Polytypic values possess polykinded types

Ralf Hinze

Institute of Information and Computing Sciences, Utrecht University, P.O.Box 80.089, 3508 TB Utrecht, The Netherlands

Abstract

A polytypic value is one that is defined by induction on the structure of types. In Haskell types are assigned so-called kinds that distinguish 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 turns out 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 data types of arbitrary kinds including first- and higher-order functors.

Haskell's type system essentially corresponds to the simply typed lambda calculus with kinds playing the rôle of types. We show that the specialization of a polytypic value to concrete instances of data types can be phrased as an interpretation of the simply typed lambda calculus. This allows us to employ logical relations to prove properties of polytypic values. Among other things, we show 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 [26] 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 [35] implies that a function of type $\forall A \, . \, A \to A \to \mathit{Bool}$

Preprint submitted to Elsevier Science 30 August 2001

must necessarily be constant. As a consequence, the programmer is forced to program a separate equality function for each type from scratch.

Polytypic programming [3,2] 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 [12] 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. 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 [16,12] 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 [22], 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 introduces the language of terms, which is based on the polymorphic lambda calculus. Section 5 explains how to define polytypic values and polykinded types. Section 6 shows how to specialize a polytypic value to concrete instances of data types. Section 7 presents several examples of polytypic functions with polykinded types, which demonstrate that the extension is useful in practice. Polytypic values enjoy polytypic properties. Section 8 shows how to express polytypic laws using logical relations. Among other things, we show that the polytypic mapping function satisfies suitable generalizations of the functorial laws. Finally, Section 9 reviews related work and Section 10 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 [31]. However, for reasons of coherence we will slightly deviate from Haskell's lexical syntax: both type constructors and type variables are written with an initial upper-case letter (in Haskell type variables begin with a lower-case letter) and both value constructors and value variables are written with an initial lower-case letter (in Haskell value constructors begin with an upper-case letter). This convention helps to easily identify values and types.

Before tackling the polytypic mapping function let us first take a look at different data types and associated monotypic mapping functions. As an aside, note that the combination of a type constructor and its mapping function is often referred to as a functor.

As a first, rather simple example consider the list data type (**data** introduces a new type and value constructors over that type).

$$\mathbf{data}\ \mathit{List}\ A = \mathit{nil} \mid \mathit{cons}\ A\ (\mathit{List}\ A)$$

Actually, *List* is not a type but a unary type constructor. In Haskell the 'type' of a type constructor is specified by the kind system. For instance, *List* has kind $\star \to \star$. The '$\star$' kind represents manifest types like *Int* or *Bool*. The kind $\mathfrak{T} \to \mathfrak{U}$ represents type constructors that map type constructors of kind $\mathfrak{T}$ to those of kind $\mathfrak{U}$. The mapping function for *List*, called map_{List}, is given by

$$\begin{aligned}
&map_{List} &&:: \forall A_1\ A_2\,.\,(A_1 \to A_2) \to (List\ A_1 \to List\ A_2)\\
&map_{List}\ map_A\ nil &&= nil\\
&map_{List}\ map_A\ (cons\ a\ as) &&= cons\ (map_A\ a)\ (map_{List}\ map_A\ as).
\end{aligned}$$

The mapping function takes a function and applies it to each element of a given list. It is perhaps unusual to call the argument function map_A. The reason for this choice will become clear as we go along. For the moment it suffices to bear in mind that the definition of map_{List} rigidly follows the structure of the data type.

The *List* type constructor is an example of a so-called regular or uniform type. Briefly, a regular type is one that can be defined as the least fixed point of a functor. Interestingly, Haskell's type system is expressive enough to rephrase *List* using an explicit fixed point operator [24]. We will repeat this construction in the following as it provides us with interesting examples of data types and associated mapping functions. First, we define the so-called base or pattern functor of *List*.

$$\mathbf{data}\ \mathit{ListF}\ A\ B = \mathit{nilF} \mid \mathit{consF}\ A\ B$$

The type *ListF* has kind $\star \to (\star \to \star)$, which shows that binary type constructors are curried in Haskell. The following definition introduces a fixed point operator on the type level (**newtype** is a variant of **data** introducing 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 the type constructor *Fix* is $(\star \to \star) \to \star$, a so-called second-order kind. In general, the order of a kind is given by $\mathit{order}(\star) = 0$ and $\mathit{order}(\mathfrak{T} \to \mathfrak{U}) = \mathit{max}\{1 + \mathit{order}(\mathfrak{T}), \mathit{order}(\mathfrak{U})\}$. It remains to define *List* as a fixed point of its base functor (**type** defines a type synonym).

$$\mathbf{type}\ \mathit{List}'\ A = \mathit{Fix}\ (\mathit{ListF}\ A)$$

Now, how can we define the mapping function for lists thus defined? For a start, we define the mapping function for the base functor.

$$\begin{aligned}
&\mathit{map}_{\mathit{ListF}} && :: \forall A_1\ A_2\,.\,(A_1 \to A_2) \to \forall B_1\ B_2\,.\,(B_1 \to B_2)\\
& && \qquad \to (\mathit{ListF}\ A_1\ B_1 \to \mathit{ListF}\ A_2\ B_2)\\
&\mathit{map}_{\mathit{ListF}}\ \mathit{map}_A\ \mathit{map}_B\ \mathit{nilF} = \mathit{nilF}\\
&\mathit{map}_{\mathit{ListF}}\ \mathit{map}_A\ \mathit{map}_B\ (\mathit{consF}\ a\ b)\\
& && = \mathit{consF}\ (\mathit{map}_A\ a)\ (\mathit{map}_B\ b)
\end{aligned}$$

Since the base functor has two type arguments, its mapping function takes two functions, map_A and map_B, and applies them to values of type A_1 and B_1, respectively. Even more interesting is the mapping function for *Fix*

$$\begin{aligned}
&\mathit{map}_{\mathit{Fix}} && :: \forall F_1\ F_2\,.\,(\forall A_1\ A_2\,.\,(A_1 \to A_2) \to (F_1\ A_1 \to F_2\ A_2))\\
& && \qquad \to (\mathit{Fix}\ F_1 \to \mathit{Fix}\ F_2)\\
&\mathit{map}_{\mathit{Fix}}\ \mathit{map}_F\ (\mathit{in}\ v) = \mathit{in}\ (\mathit{map}_F\ (\mathit{map}_{\mathit{Fix}}\ \mathit{map}_F)\ v),
\end{aligned}$$

which takes a polymorphic function as argument. In other words, $\mathit{map}_{\mathit{Fix}}$ has a so-called rank-2 type signature [21]. Though not in the current language definition of Haskell, rank-2 type signatures are supported by recent versions of the Glasgow Haskell Compiler GHC [34] and the Haskell interpreter Hugs [20]. The argument function, map_F, has a more general type than one would probably expect: it takes a function of type $A_1 \rightarrow A_2$ to a function of type $F_1\ A_1 \rightarrow F_2\ A_2$. By contrast, the mapping function for *List* (which like F has kind $\star \rightarrow \star$) takes $A_1 \rightarrow A_2$ to $\mathit{List}\ A_1 \rightarrow \mathit{List}\ A_2$. The definition below demonstrates that the extra generality is vital.

$$
\begin{aligned}
&map_{\mathit{List}'} && :: \forall A_1\ A_2 \,.\, (A_1 \rightarrow A_2) \rightarrow (\mathit{List}'\ A_1 \rightarrow \mathit{List}'\ A_2) \\
&map_{\mathit{List}'}\ map_A && = map_{\mathit{Fix}}\ (map_{\mathit{ListF}}\ map_A)
\end{aligned}
$$

The argument of map_{Fix} has type $\forall B_1\ B_2 \,.\, (B_1 \rightarrow B_2) \rightarrow (\mathit{ListF}\ A_1\ B_1 \rightarrow \mathit{ListF}\ A_2\ B_2)$, that is, F_1 is instantiated to $\mathit{ListF}\ A_1$ and F_2 to $\mathit{ListF}\ A_2$.

The list data type 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 [30].

$$
\begin{aligned}
&\textbf{data}\ \mathit{Fork}\ A = \mathit{fork}\ A\ A \\
&\textbf{data}\ \mathit{Sequ}\ A = \mathit{empty} \mid \mathit{zero}\ (\mathit{Sequ}\ (\mathit{Fork}\ A)) \mid \mathit{one}\ A\ (\mathit{Sequ}\ (\mathit{Fork}\ A))
\end{aligned}
$$

Since the type argument is changed in the recursive calls, *Sequ* is an example of a so-called nested or non-regular data type [4]. Nested data types have recently received a great deal of attention since they can capture data-structural invariants in a way that regular data types 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.

$$
\begin{aligned}
&map_{\mathit{Fork}} && :: \forall A_1\ A_2 \,.\, (A_1 \rightarrow A_2) \rightarrow (\mathit{Fork}\ A_1 \rightarrow \mathit{Fork}\ A_2) \\
&map_{\mathit{Fork}}\ map_A\ (\mathit{fork}\ a_1\ a_2) && = \mathit{fork}\ (map_A\ a_1)\ (map_A\ a_2) \\
&map_{\mathit{Sequ}} && :: \forall A_1\ A_2 \,.\, (A_1 \rightarrow A_2) \rightarrow (\mathit{Sequ}\ A_1 \rightarrow \mathit{Sequ}\ A_2) \\
&map_{\mathit{Sequ}}\ map_A\ \mathit{empty} && = \mathit{empty} \\
&map_{\mathit{Sequ}}\ map_A\ (\mathit{zero}\ as) && = \mathit{zero}\ (map_{\mathit{Sequ}}\ (map_{\mathit{Fork}}\ map_A)\ as) \\
&map_{\mathit{Sequ}}\ map_A\ (\mathit{one}\ a\ as) && = \mathit{one}\ (map_A\ a)\ (map_{\mathit{Sequ}}\ (map_{\mathit{Fork}}\ map_A)\ as)
\end{aligned}
$$

Note that map_{Sequ} requires polymorphic recursion [29]: the recursive calls have type $\forall A_1\ A_2 \,.\, (\mathit{Fork}\ A_1 \rightarrow \mathit{Fork}\ A_2) \rightarrow (\mathit{Sequ}\ (\mathit{Fork}\ A_1) \rightarrow \mathit{Sequ}\ (\mathit{Fork}\ A_2))$, which is a substitution instance of the declared type. Haskell 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 [10].

Since *Sequ* is a nested type, it cannot be expressed as a fixed point of a functor. However, it can be rephrased as a fixed point of a higher-order functor [4]. Again, we will carry out the construction to generate examples of higher-order kinded data types. The higher-order base functor associated with *Sequ* is

$$\textbf{data}\ \mathit{SequF}\ S\ A = \mathit{emptyF} \mid \mathit{zeroF}\ (S\ (\mathit{Fork}\ A)) \mid \mathit{oneF}\ A\ (S\ (\mathit{Fork}\ A)).$$

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

$$\textbf{newtype}\ \mathit{HFix}\ H\ A = \mathit{hin}\ (H\ (\mathit{HFix}\ H)\ A).$$

Since the fixed point 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 fixed point of *SequF*.

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

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.

$$\begin{aligned}
&\mathit{map}_{\mathit{SequF}} :: \forall S_1\ S_2\,.\,(\forall B_1\ B_2\,.\,(B_1 \to B_2) \to (S_1\ B_1 \to S_2\ B_2)) \\
&\qquad\qquad\qquad \to \forall A_1\ A_2\,.\,(A_1 \to A_2) \to (\mathit{SequF}\ S_1\ A_1 \to \mathit{SequF}\ S_2\ A_2) \\
&\mathit{map}_{\mathit{SequF}}\ \mathit{map}_S\ \mathit{map}_A\ \mathit{emptyF} \\
&\qquad = \mathit{emptyF} \\
&\mathit{map}_{\mathit{SequF}}\ \mathit{map}_S\ \mathit{map}_A\ (\mathit{zeroF}\ \mathit{as}) \\
&\qquad = \mathit{zeroF}\ (\mathit{map}_S\ (\mathit{map}_{\mathit{Fork}}\ \mathit{map}_A)\ \mathit{as}) \\
&\mathit{map}_{\mathit{SequF}}\ \mathit{map}_S\ \mathit{map}_A\ (\mathit{oneF}\ a\ \mathit{as}) \\
&\qquad = \mathit{oneF}\ (\mathit{map}_A\ a)\ (\mathit{map}_S\ (\mathit{map}_{\mathit{Fork}}\ \mathit{map}_A)\ \mathit{as})
\end{aligned}$$

This example indicates why argument *map*s of kind $\star \to \star$ must be polymorphic: both calls of map_S are instances of the declared type. In general, the argument mapping function may be applied to many different types. Admittedly, the type signature of $\mathit{map}_{\mathit{SequF}}$ 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 $\mathit{map}_{\mathit{HFix}}$, which has rank 3. Unfortunately, no current Haskell implementation supports rank-3 type signatures. Hence, the following code cannot be executed.

$$
\begin{aligned}
&\mathit{map}_{\mathit{HFix}} :: \forall H_1\ H_2\,.\,(\forall F_1\ F_2\,.\,(\forall C_1\ C_2\,.\,(C_1 \to C_2) \to (F_1\ C_1 \to F_2\ C_2))\\
&\qquad\qquad\qquad\qquad\qquad\qquad \to \forall B_1\ B_2\,.\,(B_1 \to B_2)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \to (H_1\ F_1\ B_1 \to H_2\ F_2\ B_2))\\
&\qquad\qquad\qquad\qquad \to \forall A_1\ A_2\,.\,(A_1 \to A_2)\\
&\qquad\qquad\qquad\qquad\qquad\qquad \to (\mathit{HFix}\ H_1\ A_1 \to \mathit{HFix}\ H_2\ A_2)\\
&\mathit{map}_{\mathit{HFix}}\ \mathit{map}_H\ \mathit{map}_A\ (\mathit{hin}\ v)\\
&\qquad = \mathit{hin}\ (\mathit{map}_H\ (\mathit{map}_{\mathit{HFix}}\ \mathit{map}_H)\ \mathit{map}_A\ v)
\end{aligned}
$$

Finally, applying $\mathit{map}_{\mathit{HFix}}$ to $\mathit{map}_{\mathit{SequF}}$ we obtain the desired function.

$$
\begin{aligned}
&\mathit{map}_{\mathit{Sequ}'} :: \forall A_1\ A_2\,.\,(A_1 \to A_2) \to (\mathit{Sequ}'\ A_1 \to \mathit{Sequ}'\ A_2)\\
&\mathit{map}_{\mathit{Sequ}'} = \mathit{map}_{\mathit{HFix}}\ \mathit{map}_{\mathit{SequF}}
\end{aligned}
$$

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 as subscripts. Hence, $\mathit{map}_{T::\mathfrak{T}}$ denotes the application of the polytypic map to the type T of kind $\mathfrak{T}$. 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 '$\Box$' ('$\star$' and '$\Box$' are sometimes called sorts).

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

$$
\begin{aligned}
&\mathit{Map}_{\mathfrak{T}::\Box} && :: \mathfrak{T} \to \mathfrak{T} \to \star\\
&\mathit{Map}_{\star}\ T_1\ T_2 && = T_1 \to T_2\\
&\mathit{Map}_{\mathfrak{A}\to\mathfrak{B}}\ T_1\ T_2 && = \forall X_1\ X_2\,.\,\mathit{Map}_{\mathfrak{A}}\ X_1\ X_2 \to \mathit{Map}_{\mathfrak{B}}\ (T_1\ X_1)\ (T_2\ X_2)
\end{aligned}
$$

The first line of the definition is the so-called kind signature, which makes precise that $\mathit{Map}_{\mathfrak{T}::\Box}$ maps two types of kind $\mathfrak{T}$ to a manifest type. In the base case $\mathit{Map}_{\star}\ T_1\ T_2$ 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, $\mathit{Map}_{\mathfrak{A}\to\mathfrak{B}}\ T_1\ T_2$ is the type of a 'conversion function'-transformer. It is not hard to see that the type signatures we have encountered 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 turns out that the technique described in [12] 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 data type declarations.

$$
\begin{aligned}
&\mathbf{data}\ 1 && = ()\\
&\mathbf{data}\ A + B && = \mathit{inl}\ A \mid \mathit{inr}\ B\\
&\mathbf{data}\ A \times B && = (A, B)
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{map}_{T::\mathfrak{T}} && :: \mathit{Map}_{\mathfrak{T}}\ T\ T\\
&\mathit{map}_{1}\ () && = ()\\
&\mathit{map}_{\mathit{Int}}\ i && = i\\
&\mathit{map}_{+}\ \mathit{map}_{A}\ \mathit{map}_{B}\ (\mathit{inl}\ a) && = \mathit{inl}\ (\mathit{map}_{A}\ a)\\
&\mathit{map}_{+}\ \mathit{map}_{A}\ \mathit{map}_{B}\ (\mathit{inr}\ b) && = \mathit{inr}\ (\mathit{map}_{B}\ b)\\
&\mathit{map}_{\times}\ \mathit{map}_{A}\ \mathit{map}_{B}\ (a, b) && = (\mathit{map}_{A}\ a, \mathit{map}_{B}\ b).
\end{aligned}
$$

This straightforward definition contains all the ingredients needed to derive *map*s for arbitrary data types of arbitrary kinds (see Section 6). 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 (×).

$$
\begin{aligned}
&\mathit{map}_{1} && = \mathit{id}\\
&\mathit{map}_{\mathit{Int}} && = \mathit{id}\\
&\mathit{map}_{+}\ \mathit{map}_{A}\ \mathit{map}_{B} && = \mathit{map}_{A} + \mathit{map}_{B}\\
&\mathit{map}_{\times}\ \mathit{map}_{A}\ \mathit{map}_{B} && = \mathit{map}_{A} \times \mathit{map}_{B}
\end{aligned}
$$

3 The simply typed lambda calculus as a type language

This section introduces the language of kinds and types that we will use in the theoretical part of the paper. The type system is essentially that of Haskell smoothing away some of its irregularities. Recall that Haskell offers one basic construct for defining new types: data type declarations. In general, a **data** declaration has the following form.

$$\mathbf{data}\ B\ A_1\ \ldots\ A_m = k_1\ T_{11}\ \ldots\ T_{1m_1} \mid \cdots \mid k_n\ T_{n1}\ \ldots\ T_{nm_n}.$$

This definition simultaneously introduces a new type constructor B and n value constructors $k_1, \ldots, k_n$. The **data** construct 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.

In the sequel we review the syntax and the semantics of the simply typed lambda calculus. A basic knowledge of this material will prove useful both for specializing polytypic values and for proving properties of polytypic values. Most of the definitions are taken from the excellent textbook by J. Mitchell [27]. Cognoscenti may safely skip Sections 3.1, 3.3 and 3.4 except perhaps for notation.

3.1 Syntax

Syntactic categories The simply typed lambda calculus has a two-level structure (kinds and types—since we will use the calculus to model Haskell's type system we continue to speak of kinds and types).

kind terms	$\mathfrak{T}, \mathfrak{U} \in \mathfrak{Kind}$
type constants	$C, D \in \mathit{Const}$
type variables	$A, B \in \mathit{var}$
type terms	$T, U \in \mathit{Type}$

Note that we use upper-case Fraktur letters for kinds and upper-case Roman letters for types.

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

$$\begin{array}{rcll} \mathfrak{T}, \mathfrak{U} \in \mathfrak{Kind} & ::= & \star & \text{kind of types} \\ & | & (\mathfrak{T} \to \mathfrak{U}) & \text{function kind} \end{array}$$

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

Type terms Pseudo-type terms are built from type constants and type variables using type application and type abstraction.

$$\begin{array}{rcll} T, U \in \mathit{Type} & ::= & C & \text{type constant} \\ & | & A & \text{type variable} \\ & | & (\Lambda A :: \mathfrak{U} \,.\, T) & \text{type abstraction} \\ & | & (T \ U) & \text{type application} \end{array}$$

We assume that type abstraction extends as far to the right as possible and that type application associates to the left. For typographic simplicity, we will often omit the kind annotation in $\Lambda A :: \mathfrak{U} \,.\, T$ (especially if $\mathfrak{U} = \star$). Finally, we abbreviate nested abstractions $\Lambda A_1 \,.\, \ldots \Lambda A_m \,.\, T$ by $\Lambda A_1 \ \ldots \ A_m \,.\, T$.

The choice of *Const*, the set of type constants, is more or less arbitrary. In order to model Haskell's **data** declarations we assume that *Const* comprises at least the constants *Int*, '1', '+', '×', and *Fix*:

$$\begin{array}{rl} \mathit{Const} \supseteq & \{ \mathit{Int} :: \star, 1 :: \star, (\times) :: \star \to \star \to \star, (+) :: \star \to \star \to \star \} \\ & \cup \{ \mathit{Fix}_{\mathfrak{T}} :: (\mathfrak{T} \to \mathfrak{T}) \to \mathfrak{T} \mid \mathfrak{T} \in \mathfrak{Kind} \}. \end{array}$$

As usual, we write binary type constants infix. We assume that '×' and '+' associate to the right and that '×' binds more tightly than '+'. The set of type constants includes a family of fixed point operators indexed by kind: $\mathit{Fix}_\star$ corresponds to *Fix* and $\mathit{Fix}_{\star \to \star}$ to *HFix* introduced in Section 2. We use '0' as an abbreviation for $\mathit{Fix}_\star \ \mathit{Id}$. In the examples, we will often omit the kind annotation in $\mathit{Fix}_{\mathfrak{T}}$.

A context is a finite set of kind assumptions of the form $A :: \mathfrak{T}$. It is convenient to view a context Γ as a finite map from type variables to kinds and write $\mathit{dom}(\Gamma)$ for its domain. Likewise, we view *Const* as a finite map from type constants to kinds. A pseudo-type term T is called a type term if there is a context Γ and a kind $\mathfrak{T}$ such that $\Gamma \vdash T :: \mathfrak{T}$ is derivable using the rules depicted in Figure 1.

The equational proof system of the simply typed lambda calculus is given by the rules in Figure 2. If $\mathcal{E}$ is a possibly empty set of equations between type

$$\frac{}{\Gamma \vdash C :: \mathit{Const}(C)} \text{ (T-CONST)}$$

$$\frac{}{\Gamma \vdash A :: \Gamma(A)} \text{ (T-VAR)}$$

$$\frac{\Gamma, A :: \mathfrak{U} \vdash T :: \mathfrak{T}}{\Gamma \vdash (\Lambda A :: \mathfrak{U} \,.\, T) :: (\mathfrak{U} \to \mathfrak{T})} \text{ (T-}\to\text{-INTRO)}$$

$$\frac{\Gamma \vdash T :: (\mathfrak{U} \to \mathfrak{T}) \qquad \Gamma \vdash U :: \mathfrak{U}}{\Gamma \vdash (T\ U) :: \mathfrak{T}} \text{ (T-}\to\text{-ELIM)}$$

Fig. 1. Kinding rules.

$$\frac{}{\Gamma \vdash (\Lambda A :: \mathfrak{U} \,.\, T)\ U = T[A := U] :: \mathfrak{T}} \text{ (T-}\beta\text{)}$$

$$\frac{A \text{ not free in } T}{\Gamma \vdash \Lambda A :: \mathfrak{U} \,.\, T\ A = T :: (\mathfrak{U} \to \mathfrak{T})} \text{ (T-}\eta\text{)}$$

$$\frac{}{\Gamma \vdash \mathit{Fix}_{\mathfrak{T}}\ T = T\ (\mathit{Fix}_{\mathfrak{T}}\ T) :: \mathfrak{T}} \text{ (T-FIX)}$$

Fig. 2. Equational proof rules (the usual 'logical' rules for reflexivity, symmetry, transitivity, and congruence are omitted).

terms, we write $\Gamma \vdash_{\mathcal{E}} T_1 = T_2 :: \mathfrak{T}$ to mean that the type equation $T_1 = T_2$ is provable using the rules and the equations in $\mathcal{E}$.

The equational proof rules identify a recursive type $\mathit{Fix}_{\mathfrak{T}}\ T$ and its unfolding $T\ (\mathit{Fix}_{\mathfrak{T}}\ T)$. In general, there are two varieties of recursive types: equi-recursive types and iso-recursive types, see [8]. In the latter system $\mathit{Fix}_{\mathfrak{T}}\ T$ and $T\ (\mathit{Fix}_{\mathfrak{T}}\ T)$ must only be isomorphic rather than equal. The development in this paper is largely independent of this design choice. We use equi-recursive types because they simplify the presentation somewhat.

3.2 Modelling **data** *declarations*

Using the simply typed lambda calculus as a type language we can easily translate data type declarations into type terms. For instance, the type B defined by the schematic **data** declaration in the beginning of Section 3 is modelled by (we tacitly assume that the kinds of the type variables have been inferred)

$$\mathit{Fix}\ (\Lambda B \,.\, \Lambda A_1\ \ldots\ A_m \,.\, (T_{11} \times \cdots \times T_{1m_1}) + \cdots + (T_{n1} \times \cdots \times T_{nm_n})),$$

where $T_1 \times \cdots \times T_k = 1$ for $k = 0$. For simplicity, n-ary sums are reduced to binary sums and n-ary products to binary products. For instance, the **data** declaration

$$\textbf{data}\ \mathit{List}\ A = \mathit{nil} \mid \mathit{cons}\ A\ (\mathit{List}\ A)$$

is translated to

$$\mathit{Fix}\ (\Lambda \mathit{List}\,.\,\Lambda A\,.\,1 + A \times \mathit{List}\ A).$$

Interestingly, the representation of regular types such as *List* can be improved by applying a technique called lambda-dropping [6]: if $\mathit{Fix}\ (\Lambda F\,.\,\Lambda A\,.\,T)$ is regular, then it is equivalent to $\Lambda A\,.\,\mathit{Fix}\ (\Lambda B\,.\,T[F\ A := B])$ 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 $\mathit{Fix}\ (\Lambda \mathit{List}\,.\,\Lambda A\,.\,1 + A \times \mathit{List}\ A)$ is $\Lambda A\,.\,\mathit{Fix}\ (\Lambda B\,.\,1 + A \times B)$. The λ-dropped version employs the fixed point operator at kind $\star$ whereas the λ-lifted version employs the fixed point operator at kind $\star \to \star$. Nested types such as *Sequ* are not amenable to this transformation since the type argument of the nested type is changed in the recursive call(s). As an aside, note that the λ-dropped and the λ-lifted version correspond to two different methods of modelling parameterized types: families of first-order fixed points versus higher-order fixed points, see, for instance, [5].

3.3 Environment models

This section is concerned with the denotational semantics of the simply typed lambda calculus. There are two general frameworks for describing the semantics: environment models and models based on cartesian closed categories. We will use environment models for the presentation since they are somewhat easier to understand.

The definition of the semantics proceeds in three steps. First, we introduce so-called applicative structures, and then we define two conditions that an applicative structure must satisfy to qualify as a model.

Definition 1 *An applicative structure* $\mathcal{E}$ *is a tuple* $(\mathsf{E}, \mathsf{app}, \mathsf{const})$ *such that*

- $\mathsf{E} = (\mathsf{E}^{\mathfrak{T}} \mid \mathfrak{T} \in \mathfrak{Kind})$ *is a family of sets,*
- $\mathsf{app} = (\mathsf{app}_{\mathfrak{T},\mathfrak{U}} : \mathsf{E}^{\mathfrak{T}\to\mathfrak{U}} \to (\mathsf{E}^{\mathfrak{T}} \to \mathsf{E}^{\mathfrak{U}}) \mid \mathfrak{T}, \mathfrak{U} \in \mathfrak{Kind})$ *is a family of maps, and*
- $\mathsf{const} : \mathit{Const} \to \mathsf{E}$ *is a mapping from type constants to values such that* $\mathsf{const}(C) \in \mathsf{E}^{\mathit{Const}(C)}$ *for every* $C \in \mathit{dom}(\mathit{Const})$.

The first condition on models requires that equality between elements of function kinds is standard equality on functions.

Definition 2 *An applicative structure* $\mathcal{E} = (\mathsf{E}, \mathsf{app}, \mathsf{const})$ *is extensional, if* $\forall \phi_1, \phi_2 \in \mathsf{E}^{\mathfrak{T} \to \mathfrak{U}}. (\forall \alpha \in \mathsf{E}^{\mathfrak{T}}. \mathsf{app}\ \phi_1\ \alpha = \mathsf{app}\ \phi_2\ \alpha) \supset \phi_1 = \phi_2$.

A simple example for an applicative structure is the set of type terms itself: Let $\mathcal{H}$ be an infinite context that provides infinitely many type variables of each kind. An extensional applicative structure $(\mathit{Type}, \mathsf{app}, \mathsf{const})$ may then be defined by letting $\mathit{Type}^{\mathfrak{T}} = \{\, T \mid \Gamma \vdash T :: \mathfrak{T}$ for some finite $\Gamma \subseteq \mathcal{H}\,\}$, $\mathsf{app}\ T\ U = (T\ U)$, and $\mathsf{const}(C) = C$.

The second condition on models ensures that the applicative structure has enough points so that every type term containing type abstractions can be assigned a meaning in the structure. To formulate the condition we require the notion of an environment. An environment η is a map from type variables to values. If Γ is a context, then we say η satisfies Γ if $\eta(A) \in \mathsf{E}^{\Gamma(A)}$ for every $A \in \mathit{dom}(\Gamma)$. If η is an environment, then $\eta(A := \alpha)$ is the environment mapping A to α and B to $\eta(B)$ for B different from A.

Definition 3 *An applicative structure* $\mathcal{E} = (\mathsf{E}, \mathsf{app}, \mathsf{const})$ *is an environment model if it is extensional and if the clauses below define a total meaning function on terms* $\Gamma \vdash T :: \mathfrak{T}$ *and environments such that* η *satisfies* Γ.

$$\begin{aligned}
&\mathcal{E}[\![\Gamma \vdash C :: \mathit{Const}(C)]\!]\eta = \mathsf{const}(C)\\
&\mathcal{E}[\![\Gamma \vdash A :: \Gamma(A)]\!]\eta \qquad\; = \eta(A)\\
&\mathcal{E}[\![\Gamma \vdash (\Lambda A :: \mathfrak{U}.\, T) :: (\mathfrak{U} \to \mathfrak{T})]\!]\eta\\
&\qquad\qquad\qquad\qquad\qquad = \textit{the unique } \phi \in \mathsf{E}^{\mathfrak{U} \to \mathfrak{T}} \textit{ such that for all } \alpha \in \mathsf{E}^{\mathfrak{U}}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{app}_{\mathfrak{U},\mathfrak{T}}\ \phi\ \alpha = \mathcal{E}[\![\Gamma, A :: \mathfrak{U} \vdash T :: \mathfrak{T}]\!]\eta(A := \alpha)\\
&\mathcal{E}[\![\Gamma \vdash (T\ U) :: \mathfrak{T}]\!]\eta \qquad = \mathsf{app}_{\mathfrak{U},\mathfrak{T}}\ (\mathcal{E}[\![\Gamma \vdash T :: (\mathfrak{U} \to \mathfrak{T})]\!]\eta)\ (\mathcal{E}[\![\Gamma \vdash U :: \mathfrak{U}]\!]\eta)
\end{aligned}$$

Note that extensionality guarantees the uniqueness of the element ϕ whose existence is postulated in the third clause.

The set of type terms can be turned into an environment model if we identify type terms that are provably equal: Let $\mathcal{E}$ be a possibly empty set of equations between type terms. Define the equivalence class $[\,T\,] = \{\, T' \mid \Gamma \vdash_{\mathcal{E}} T = T' :: \mathfrak{T}$ for some finite $\Gamma \subseteq \mathcal{H}\,\}$ and let $\mathit{Type}^{\mathfrak{T}}/\mathcal{E} = \{[\,T\,] \mid T \in \mathit{Type}^{\mathfrak{T}}\}$, $(\mathsf{app}/\mathcal{E})\ [\,T\,]\ [\,U\,] = [\,T\ \ U\,]$, and $(\mathsf{const}/\mathcal{E})\ (C) = [\,C\,]$. Then $(\mathit{Type}/\mathcal{E}, \mathsf{app}/\mathcal{E}, \mathsf{const}/\mathcal{E})$ is an environment model.

The environment model condition is often difficult to check. An equivalent, but simpler condition is the combinatory model condition.

Definition 4 *An applicative structure* $\mathcal{E} = (\mathsf{E}, \mathsf{app}, \mathsf{const})$ *satisfies the com-*

binatory model condition if for all kinds $\mathfrak{T}$, $\mathfrak{U}$ *and* $\mathfrak{V}$ *there exist elements* $\mathbf{K}_{\mathfrak{T},\mathfrak{U}} \in \mathbf{E}^{\mathfrak{T}\to\mathfrak{U}\to\mathfrak{T}}$ *and* $\mathbf{S}_{\mathfrak{T},\mathfrak{U},\mathfrak{V}} \in \mathbf{E}^{(\mathfrak{T}\to\mathfrak{U}\to\mathfrak{V})\to(\mathfrak{T}\to\mathfrak{U})\to\mathfrak{T}\to\mathfrak{V}}$ *such that*

$$\begin{aligned}
&\mathbf{app}\ (\mathbf{app}\ \mathbf{K}\ X)\ Y &&= X\\
&\mathbf{app}\ (\mathbf{app}\ (\mathbf{app}\ \mathbf{S}\ X)\ Y)\ Z &&= \mathbf{app}\ (\mathbf{app}\ X\ Z)\ (\mathbf{app}\ Y\ Z)
\end{aligned}$$

for all X, Y *and* Z *of the appropriate kinds.*

3.4 Logical relations

Logical relations are an important tool in the study of typed lambda calculi. We will use them extensively in Section 8 to prove properties of polytypic values.

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

Definition 5 *Let* $\mathcal{E}_1$ *and* $\mathcal{E}_2$ *be applicative structures. A logical relation* $\mathcal{R} = (\mathcal{R}^{\mathfrak{T}} \mid \mathfrak{T} \in \mathfrak{Kind})$ *over* $\mathcal{E}_1$ *and* $\mathcal{E}_2$ *is a family of relations such that*

- $\mathcal{R}^{\mathfrak{T}} \subseteq \mathbf{E}_1^{\mathfrak{T}} \times \mathbf{E}_2^{\mathfrak{T}}$ *for each kind* $\mathfrak{T}$,
- $(\mathbf{const}_1(C), \mathbf{const}_2(C)) \in \mathcal{R}^{Const(C)}$ *for every type constant* $C \in dom(Const)$,
- $\mathcal{R}^{\mathfrak{T}\to\mathfrak{U}}$ *is closed under type application and type abstraction:*

$$\begin{aligned}
&(\phi_1, \phi_2) \in \mathcal{R}^{\mathfrak{T}\to\mathfrak{U}}\\
&\qquad \equiv\ \forall \alpha_1 \in \mathbf{E}_1^{\mathfrak{T}}, \alpha_2 \in \mathbf{E}_2^{\mathfrak{T}}\,.\\
&\qquad\qquad\qquad (\alpha_1, \alpha_2) \in \mathcal{R}^{\mathfrak{T}}\ \supset\ (\mathbf{app}_1\ \phi_1\ \alpha_1, \mathbf{app}_2\ \phi_2\ \alpha_2) \in \mathcal{R}^{\mathfrak{U}}.
\end{aligned}$$

Usually, a logical relation is defined on the kind constant $\star$ only; the third clause of the definition then shows how to extend the relation to functional kinds.

Now, say, we are given two models of the simply typed lambda calculus. Then Lemma 1 below shows that the meaning of a type term in one model is logically related to its meaning in the other model. This lemma is sometimes called the Basic Lemma of logical relations.

Lemma 1 *Let* $\mathcal{E}_1$ *and* $\mathcal{E}_2$ *be two environment models and let* $\mathcal{R}$ *be a logical relation over* $\mathcal{E}_1$ *and* $\mathcal{E}_2$*. Then*

$$(\mathcal{E}_1[\![\Gamma \vdash T :: \mathfrak{T}]\!]\eta_1, \mathcal{E}_2[\![\Gamma \vdash T :: \mathfrak{T}]\!]\eta_2) \in \mathcal{R}^{\mathfrak{T}},$$

for all environments η_1 *and* η_2*, satisfying* Γ*, such that* $(\eta_1(A), \eta_2(A)) \in \mathcal{R}^{\Gamma(A)}$

for all $A \in \mathit{dom}(\Gamma)$*.*

4 The polymorphic lambda calculus

We have seen in the introduction that instances of *map* require first-class polymorphism in general. For instance, $\mathit{map}_{\mathit{SequF}}$ takes a polymorphic argument to a polymorphic result. To make the use of polymorphism explicit we will use a variant of the polymorphic lambda calculus [9] both for defining and for specializing polytypic values. This section provides a brief introduction to the calculus. As an aside, note that a similar language is also used as the internal language of the Glasgow Haskell Compiler [32].

Syntactic categories The polymorphic lambda calculus has a three-level structure (kinds, type schemes, and terms) incorporating the simply typed lambda calculus on the type level.

type schemes	$R, S \in \mathit{Scheme}$
individual constants	$c, d \in \mathit{const}$
individual variables	$a, b \in \mathit{var}$
terms	$t, u \in \mathit{term}$

We use lower-case Roman letters for terms.

Type schemes Pseudo-type schemes are formed according to the following grammar.

$$\begin{array}{rcll} R, S \in \mathit{Scheme} & ::= & T & \text{type term} \\ & \mid & (R \to S) & \text{functional type} \\ & \mid & (\forall A :: \mathfrak{U} \,.\, S) & \text{polymorphic type} \end{array}$$

A pseudo-type scheme S is called a type scheme if there is a context Γ and a kind $\mathfrak{T}$ such that $\Gamma \vdash S :: \mathfrak{T}$ is derivable using the rules listed in Figures 1 and 3.

$$\frac{\Gamma \vdash R :: \star \qquad \Gamma \vdash S :: \star}{\Gamma \vdash (R \to S) :: \star} \ (\textsc{t-fun})$$

$$\frac{\Gamma, A :: \mathfrak{U} \vdash S :: \star}{\Gamma \vdash (\forall A :: \mathfrak{U} \,.\, S) :: \star} \ (\textsc{t-all})$$

Fig. 3. Additional kinding rules for type schemes.

Terms Pseudo-terms are given by the grammar

$$\begin{array}{rcll} t, u \in \mathit{term} & ::= & c & \text{constant} \\ & | & a & \text{variable} \\ & | & (\lambda a :: S \,.\, t) & \text{abstraction} \\ & | & (t\ u) & \text{application} \\ & | & (\lambda A :: \mathfrak{U} \,.\, t) & \text{universal abstraction} \\ & | & (t\ R) & \text{universal application.} \end{array}$$

Here, $\lambda A :: \mathfrak{U} \,.\, t$ denotes universal abstraction (forming a polymorphic value) and $t\ R$ denotes universal application (instantiating a polymorphic value). Note that we use the same syntax for value abstraction $\lambda a :: S \,.\, t$ (here a is a value variable) and universal abstraction $\lambda A :: \mathfrak{U} \,.\, t$ (here A is a type variable). We assume that the set *const* of value constants includes at least the polymorphic fixed point operator

$$\mathit{fix} :: \forall A \,.\, (A \to A) \to A$$

and suitable functions for each of the other type constants C in $\mathit{dom}(\mathit{Const})$ (such as () for '1', *inl*, *inr*, and **case** for '+', and *outl*, *outr*, and (_, _) for '×'). We use $\bot :: \forall A \,.\, A$ as an abbreviation for $\Lambda A \,.\, \mathit{fix}\ A\ \mathit{id}$. To improve readability we will usually omit the type argument of *fix* and $\bot$.

To give the typing rules we have to extend the notion of context. A context is a finite set of kind assumptions $A :: \mathfrak{T}$ and type assumptions $a :: S$. We say a context Γ is closed if Γ is either empty, or if $\Gamma = \Gamma_1, A :: \mathfrak{T}$ with Γ_1 closed, or if $\Gamma = \Gamma_1, a :: S$ with Γ_1 closed and $\mathit{free}(S) \subseteq \mathit{dom}(\Gamma)$. In the following we assume that contexts are closed. This restriction is necessary to prevent non-sensible terms such as $\lambda A :: \star \,.\, a :: A$ where the value variable a carries the type variable A out of scope. A pseudo-term t is called a term if there is some context Γ and some type scheme S such that $\Gamma \vdash t :: S$ is derivable using the typing rules depicted in Figure 4. Note that rule (CONV) allows us to interchange provably equal types.

The equational proof system of the polymorphic lambda calculus is given by the rules in Figure 5. When we discuss the specialization of polytypic

$$\frac{}{\Gamma \vdash c :: \mathit{const}(c)} \; (\textsc{var})$$

$$\frac{}{\Gamma \vdash a :: \Gamma(a)} \; (\textsc{const})$$

$$\frac{\Gamma, a :: S \vdash t :: R}{\Gamma \vdash (\lambda a :: S \,.\, t) :: (S \to R)} \; (\to\text{-}\textsc{intro})$$

$$\frac{\Gamma \vdash t :: (S \to R) \qquad \Gamma \vdash u :: S}{\Gamma \vdash (t\ u) :: R} \; (\to\text{-}\textsc{elim})$$

$$\frac{\Gamma, A :: \mathfrak{U} \vdash t :: S}{\Gamma \vdash (\lambda A :: \mathfrak{U} \,.\, t) :: (\forall A :: \mathfrak{U} \,.\, S)} \; (\forall\text{-}\textsc{intro})$$

$$\frac{\Gamma \vdash t :: (\forall A :: \mathfrak{U} \,.\, S) \qquad \Gamma \vdash R :: \mathfrak{U}}{\Gamma \vdash (t\ R) :: S[\, A := R\,]} \; (\forall\text{-}\textsc{elim})$$

$$\frac{\Gamma \vdash t :: R \qquad \Gamma \vdash R = S :: \star}{\Gamma \vdash t :: S} \; (\textsc{conv})$$

Fig. 4. Typing rules.

$$\frac{}{\Gamma \vdash (\lambda a :: S \,.\, t)\ u = t[\, a := u\,] :: R} \; (\beta)$$

$$\frac{a \text{ not free in } t}{\Gamma \vdash \lambda a :: S \,.\, t\ a = t :: (S \to R)} \; (\eta)$$

$$\frac{}{\Gamma \vdash (\lambda A :: \mathfrak{U} \,.\, t)\ R = t[\, A := R\,] :: S[\, A := R\,]} \; (\beta)_\forall$$

$$\frac{A \text{ not free in } t}{\Gamma \vdash \lambda A :: \mathfrak{U} \,.\, t\ A = t :: (\forall A :: \mathfrak{U} \,.\, S)} \; (\eta)_\forall$$

$$\frac{}{\Gamma \vdash \mathit{fix}\ R\ f = f\ (\mathit{fix}\ R\ f) :: R} \; (\textsc{fix})$$

Fig. 5. Equational proof rules (the usual 'logical' rules for reflexivity, symmetry, transitivity, and congruence are omitted).

values we will consider type schemes and terms modulo provable equality. Let $\mathcal{H}$ be an infinite context that provides type variables of each kind and variables of each type scheme and let $\mathcal{E}$ be a set of equations between type schemes and/or between terms. Analogous to $\mathit{Type}^{\mathfrak{T}}$, $[\, T\,]$ and $\mathit{Type}^{\mathfrak{T}}/\mathcal{E}$ we define $\mathit{Scheme}^{\mathfrak{T}} = \{\, S \mid \Gamma \vdash S :: \mathfrak{T} \text{ for some finite } \Gamma \subseteq \mathcal{H}\,\}$, the equivalence class $[\, S\,] = \{\, S' \mid \Gamma \vdash_{\mathcal{E}} S = S' :: \mathfrak{T} \text{ for some finite } \Gamma \subseteq \mathcal{H}\,\}$, $\mathit{Scheme}^{\mathfrak{T}}/\mathcal{E} = \{[\, S\,] \mid S \in \mathit{Scheme}^{\mathfrak{T}}\}$, and $\mathit{Term}^{S} = \{\, t \mid \Gamma \vdash t :: S \text{ for some finite } \Gamma \subseteq \mathcal{H}\,\}$, $[\, t\,] = \{\, t' \mid \Gamma \vdash_{\mathcal{E}} t = t' :: S \text{ for some finite } \Gamma \subseteq \mathcal{H}\,\}$, and $\mathit{Term}^{S}/\mathcal{E} = \{[\, t\,] \mid t \in \mathit{Term}^{S}\}$. Note that $[\, S_1\,] = [\, S_2\,]$ implies $\mathit{Term}^{S_1} = \mathit{Term}^{S_2}$ because of rule

(CONV). The set $\mathcal{E}$ of equations might include, for instance, $t = \bot :: 0$ to formalize that the type '0' contains only a single element.

5 Defining polytypic values

Let us now extend the polymorphic lambda calculus by polytypic definitions.

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 $\mathfrak{T} \to \mathfrak{U}$ are specialized. We will return to this point in Section 6. In general, a polykinded type definition has the following schematic form.

$$
\begin{aligned}
&\mathit{Poly}_{\mathfrak{T}::\Box} &&:: \mathfrak{T} \to \cdots \to \mathfrak{T} \to \star \\
&\mathit{Poly}_{\star} &&= \Lambda X_1 \ \ldots \ X_n \, . \ldots \\
&\mathit{Poly}_{\mathfrak{A}\to\mathfrak{B}} &&= \Lambda X_1 \ \ldots \ X_n \, . \, \forall A_1 \ \ldots \ A_n \, . \, \mathit{Poly}_{\mathfrak{A}} \ A_1 \ \ldots \ A_n \\
& && \qquad \to \mathit{Poly}_{\mathfrak{B}} \ (X_1 \ A_1) \ \ldots \ (X_n \ A_n)
\end{aligned}
$$

The kind signature makes precise that the kind-indexed type $\mathit{Poly}_{\mathfrak{T}::\Box}$ maps n types of kind $\mathfrak{T}$ to a manifest type (for $\mathit{Map}_{\mathfrak{T}::\Box}$ 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.

$$
\begin{aligned}
&\mathit{poly}_{T::\mathfrak{T}} &&:: \mathit{Poly}_{\mathfrak{T}} \ T \ \ldots \ T \\
&\mathit{poly}_{1} &&= \ldots \\
&\mathit{poly}_{\mathit{Int}} &&= \ldots \\
&\mathit{poly}_{+} &&= \lambda A_1 \ \ldots \ A_n \, . \, \lambda \mathit{poly}_A :: (\mathit{Poly}_{\star} \ A_1 \ \ldots \ A_n) \, . \\
& && \qquad \lambda B_1 \ \ldots \ B_n \, . \, \lambda \mathit{poly}_B :: (\mathit{Poly}_{\star} \ B_1 \ \ldots \ B_n) \, . \ldots \\
&\mathit{poly}_{\times} &&= \lambda A_1 \ \ldots \ A_n \, . \, \lambda \mathit{poly}_A :: (\mathit{Poly}_{\star} \ A_1 \ \ldots \ A_n) \, . \\
& && \qquad \lambda B_1 \ \ldots \ B_n \, . \, \lambda \mathit{poly}_B :: (\mathit{Poly}_{\star} \ B_1 \ \ldots \ B_n) \, . \ldots
\end{aligned}
$$

Again, the polytypic programmer has to fill out the right-hand sides. To be well-typed, the poly_C instances must have type $\mathit{Poly}_{\mathit{Const}\,(C)} \ C \ \ldots \ C$ as stated in the type signature.

Using the syntax of the polymorphic lambda calculus the definition of the polytypic mapping function takes on the following form.

$$
\begin{aligned}
& \mathit{Map}_{\mathfrak{T}::\Box} && :: \mathfrak{T} \to \mathfrak{T} \to \star \\
& \mathit{Map}_{\star} && = \Lambda X_1\ X_2 \,.\, X_1 \to X_2 \\
& \mathit{Map}_{\mathfrak{A}\to\mathfrak{B}} && = \Lambda X_1\ X_2 \,.\, \forall A_1\ A_2 \,.\, \mathit{Map}_{\mathfrak{A}}\ A_1\ A_2 \to \mathit{Map}_{\mathfrak{B}}\ (X_1\ A_1)\ (X_2\ A_2) \\
& \mathit{map}_1 && = \lambda u :: 1 \,.\, u \\
& \mathit{map}_{\mathit{Int}} && = \lambda i :: \mathit{Int} \,.\, i \\
& \mathit{map}_{+} && = \lambda A_1\ A_2 \,.\, \lambda \mathit{map}_A :: (A_1 \to A_2) \,.\, \lambda B_1\ B_2 \,.\, \lambda \mathit{map}_B :: (B_1 \to B_2) \,. \\
& && \qquad \lambda s :: (A_1 + B_1) \,.\, \textbf{case}\ s\ \textbf{of}\ \{ \mathit{inl}\ a \Rightarrow \mathit{inl}\ (\mathit{map}_A\ a); \\
& && \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{inr}\ b \Rightarrow \mathit{inr}\ (\mathit{map}_B\ b) \} \\
& \mathit{map}_{\times} && = \lambda A_1\ A_2 \,.\, \lambda \mathit{map}_A :: (A_1 \to A_2) \,.\, \lambda B_1\ B_2 \,.\, \lambda \mathit{map}_B :: (B_1 \to B_2) \,. \\
& && \qquad \lambda p :: (A_1 \times B_1) \,.\, (\mathit{map}_A\ (\mathit{outl}\ p), \mathit{map}_B\ (\mathit{outr}\ p))
\end{aligned}
$$

6 Specializing polytypic values

This section is concerned with the specialization of polytypic values to concrete instances of data types. We have seen in Section 2 that the structure of each instance of map_T 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 phrased 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: roughly speaking, type application is interpreted by value application, type abstraction by value abstraction, and type recursion by value recursion.

Before we discuss the formal definitions let us take a look at an example first. Consider specializing *map* for the type *Matrix* given by $\Lambda A \,.\, \mathit{List}\ (\mathit{List}\ A)$. The instance $\mathit{map}_{\mathit{Matrix}}$ is given by

$$
\begin{aligned}
& \mathit{map}_{\mathit{Matrix}} :: \forall A_1\ A_2 \,.\, (A_1 \to A_2) \to (\mathit{Matrix}\ A_1 \to \mathit{Matrix}\ A_2) \\
& \mathit{map}_{\mathit{Matrix}} = \lambda A_1\ A_2 .\ \lambda \mathit{map}_A :: (A_1 \to A_2) \,.\, \mathit{map}_{\mathit{List}}\ (\mathit{List}\ A_1)\ (\mathit{List}\ A_2) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{map}_{\mathit{List}}\ A_1\ A_2\ \mathit{map}_A).
\end{aligned}
$$

The specialization of the type application $\mathit{List}\ A$ is given by the lambda term $\mathit{map}_{\mathit{List}}\ A_1\ A_2\ \mathit{map}_A$, which is a combination of universal application and value application. Thus, if we aim at phrasing the specialization of *map* as a model of the simply typed lambda calculus we must administer both the actual instance of *map* and its type. This observation suggests to represent instances as triples $(S_1, S_2; \mathit{map}_S)$ where S_1 and S_2 are type schemes of some kind, say, $\mathfrak{T}$ and map_S is a value term of type $Map_{\mathfrak{T}}\ S_1\ S_2$. Of course, we have to work with equivalence classes of type schemes and terms. Let $\mathcal{E}$ be a set of equations specifying identities between type schemes and/or between terms. The applicative structure $\mathcal{M} = (\mathbf{M}, \mathbf{app}, \mathbf{const})$ is then given by

$$
\begin{aligned}
\mathbf{M}^{\mathfrak{T}} \quad &= ([S_1], [S_2] \in \mathit{Scheme}^{\mathfrak{T}}/\mathcal{E}; \mathit{Term}^{Map_{\mathfrak{T}}\ S_1\ S_2}/\mathcal{E}) \\
\mathbf{app}_{\mathfrak{T},\mathfrak{U}}\, &([R_1], [R_2]; [t])\ ([S_1], [S_2]; [u]) \\
&= ([R_1\ S_1], [R_2\ S_2]; [t\ S_1\ S_2\ u]) \\
\mathbf{const}(C) &= ([C], [C]; [map_C]).
\end{aligned}
$$

Note that the semantic application function **app** uses both the type and the value component of its second argument. It is not hard to see that the result term $t\ S_1\ S_2\ u$ is well-typed: t has type $\forall A_1\ A_2 . Map_{\mathfrak{T}}\ A_1\ A_2 \rightarrow Map_{\mathfrak{U}}\ (R_1\ A_1)\ (R_2\ A_2)$, S_1 and S_2 have kind $\mathfrak{T}$, and u has type $Map_{\mathfrak{T}}\ S_1\ S_2$. It is important to note that the definition of $Map_{\mathfrak{T}\rightarrow\mathfrak{U}}$ and **app** go hand in hand. This explains, in particular, why the definition of $Poly_{\mathfrak{T}\rightarrow\mathfrak{U}}$ is fixed for functional kinds.

Now, does $\mathcal{M}$ also constitute a model? To this end we have to show that $\mathcal{M}$ is extensional and that is satisfies the combinatorial model condition. The first condition is easy to check. To establish the second condition we define combinators (omitting type and kind annotations)

$$
\begin{aligned}
\mathbf{K}_{\mathfrak{T},\mathfrak{U}} \ &= ([K_{\mathfrak{T},\mathfrak{U}}], [K_{\mathfrak{T},\mathfrak{U}}]; [\lambda A_1\ A_2 . \lambda map_A . \lambda B_1\ B_2 . \lambda map_B . map_A]) \\
\mathbf{S}_{\mathfrak{T},\mathfrak{U},\mathfrak{V}} &= ([S_{\mathfrak{T},\mathfrak{U},\mathfrak{V}}], [S_{\mathfrak{T},\mathfrak{U},\mathfrak{V}}]; \\
&\qquad [\lambda A_1\ A_2 . \lambda map_A . \lambda B_1\ B_2 . \lambda map_B . \Lambda C_1\ C_2 . \lambda map_C . \\
&\qquad\qquad (map_A\ C_1\ C_2\ map_C)\ (B_1\ C_1)\ (B_2\ C_2)\ (map_B\ C_1\ C_2\ map_C)])
\end{aligned}
$$

where K and S are given by

$$
\begin{aligned}
K_{\mathfrak{T},\mathfrak{U}} \ &= \Lambda A :: \mathfrak{T} . \Lambda B :: \mathfrak{U} . A \\
S_{\mathfrak{T},\mathfrak{U},\mathfrak{V}} &= \Lambda A :: (\mathfrak{T} \rightarrow \mathfrak{U} \rightarrow \mathfrak{V}) . \Lambda B :: (\mathfrak{T} \rightarrow \mathfrak{U}) . \Lambda C :: \mathfrak{T} . (A\ C)\ (B\ C).
\end{aligned}
$$

It is straightforward to prove that the combinatory laws are indeed satisfied.

It remains to provide interpretations for the fixed point operators $Fix_{\mathfrak{T}}$. The definition is essentially the same for all polytypic values. This is why the polytypic programmer need not supply instances for $Fix_{\mathfrak{T}}$ by hand. Here is the definition of $map_{Fix_{\mathfrak{T}}}$.

$$
\begin{aligned}
map_{Fix_{\mathfrak{T}}} = \ &\lambda F_1\ F_2 . \lambda map_F :: (Map_{\mathfrak{T}\rightarrow\mathfrak{T}}\ F_1\ F_2) . \\
&\quad \mathit{fix}\ (map_F\ (Fix_{\mathfrak{T}}\ F_1)\ (Fix_{\mathfrak{T}}\ F_2))
\end{aligned}
$$

Note that $map_{Fix_{\mathfrak{T}}}$ essentially equals *fix*—if we ignore type abstractions and type applications for a moment. Let us briefly check that the definition of $map_{Fix_{\mathfrak{T}}}$ is well-typed. The universal application $map_F\ (Fix_{\mathfrak{T}}\ F_1)\ (Fix_{\mathfrak{T}}\ F_2)$ has type $Map_{\mathfrak{T}}\ (Fix_{\mathfrak{T}}\ F_1)\ (Fix_{\mathfrak{T}}\ F_2) \to Map_{\mathfrak{T}}\ (F_1\ (Fix_{\mathfrak{T}}\ F_1))\ (F_2\ (Fix_{\mathfrak{T}}\ F_2))$. Since $Fix_{\mathfrak{T}}\ F_i = F_i\ (Fix_{\mathfrak{T}}\ F_i)$, we can use rule (CONV) to infer the type $Map_{\mathfrak{T}}\ (Fix_{\mathfrak{T}}\ F_1)\ (Fix_{\mathfrak{T}}\ F_2) \to Map_{\mathfrak{T}}\ (Fix_{\mathfrak{T}}\ F_1)\ (Fix_{\mathfrak{T}}\ F_2)$. Consequently, $fix\ (map_F\ (Fix_{\mathfrak{T}}\ F_1)\ (Fix_{\mathfrak{T}}\ F_2))$ has type $Map_{\mathfrak{T}}\ (Fix_{\mathfrak{T}}\ F_1)\ (Fix_{\mathfrak{T}}\ F_2)$ as desired.

Now, let us turn to the general case. The definitions for arbitrary polytypic values are very similar to the ones for *map*. The applicative structure $\mathcal{P} = (\mathbf{P}, \mathbf{app}, \mathbf{const})$ induced by the polytypic value $poly_{T::\mathfrak{T}} :: Poly_{\mathfrak{T}}\ T\ \ldots\ T$ is given by

$$\begin{aligned}
\mathbf{P}^{\mathfrak{T}} &= ([S_1], \ldots, [S_n] \in Scheme^{\mathfrak{T}}/\mathcal{E};\ Term^{Poly_{\mathfrak{T}}\ S_1\ \ldots\ S_n}/\mathcal{E}) \\
&\mathbf{app}_{\mathfrak{T},\mathfrak{U}}\,([R_1], \ldots, [R_n]; [t])\ ([S_1], \ldots, [S_n]; [u]) \\
&= ([R_1\ S_1], \ldots, [R_n\ S_n]; [t\ S_1\ \ldots\ S_n\ u]) \\
\mathbf{const}(C) &= ([C], \ldots, [C]; [poly_C]).
\end{aligned}$$

where $poly_{Fix_{\mathfrak{T}}}$ is defined

$$\begin{aligned}
poly_{Fix_{\mathfrak{T}}} = \lambda F_1\ \ldots\ F_n\,.\,\lambda poly_F :: (Poly_{\mathfrak{T}\to\mathfrak{T}}\ F_1\ \ldots\ F_n)\,.& \\
fix\ (poly_F\ (Fix_{\mathfrak{T}}\ F_1)\ \ldots\ (Fix_{\mathfrak{T}}\ F_n)).&
\end{aligned}$$

Three remarks are in order. First, the value domain $\mathbf{P}^{\mathfrak{T}}$ is a so-called dependent product: the type of the last component depends on the first n components. A similar structure has also been used to give a semantics to Standard ML's module system, see [28]. Second, if T is a closed type term, then $\mathcal{P}[\![\emptyset \vdash T :: \mathfrak{T}]\!]\eta$ is of the form $([T], \ldots, [T]; [poly_T])$ where $poly_T$ is the desired instance. As an aside, note that this is in agreement with *poly*'s type signature $poly_{T::\mathfrak{T}} :: Poly_{\mathfrak{T}}\ T\ \ldots\ T$. Third, a polytypic value can be specialized to a type but not to a type scheme. This restriction is, however, quite mild. Haskell, for instance, does not allow universal quantifiers in **data** declarations.

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

$$\begin{aligned}
poly_{\Lambda A\,.\,F\ (G\ A)} = \lambda A_1\ \ldots\ A_n\,.\,\lambda poly_A\,.& \\
poly_F\ (G\ A_1)\ \ldots\ (G\ A_n)\ (poly_G\ A_1\ \ldots\ A_n\ poly_A).&
\end{aligned}$$

Writing type and function composition as usual this implies, in particular, that $map_{F\cdot G} = map_F \cdot map_G$. Perhaps surprisingly, this relationship holds for all polytypic values, not only for mapping functions. A similar observation is that $poly_{\Lambda A\,.\,A} = \lambda A\,.\,\lambda poly_A\,.\,poly_A$ for all polytypic values. Abbreviating $\Lambda A\,.\,A$ by Id we have, in particular, that $map_{Id} = id$. As an aside, note that these polytypic identities are not to be confused with the familiar functorial laws $map_F\ id = id$ and $map_F\ (\varphi \cdot \psi) = map_F\ \varphi \cdot map_F\ \psi$ (see Section 8.1), which are base-level identities.

7 Examples

This section presents further examples of polytypic values with polykinded types (for reasons of readability we use again Haskell notation).

7.1 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 $equal_T$ has type $T \to T \to Bool$, which determines the following definition.

$$
\begin{aligned}
&Equal_{\mathfrak{T}::\Box} && :: \mathfrak{T} \to \star \\
&Equal_{\star}\ X && = X \to X \to Bool \\
&Equal_{\mathfrak{A}\to\mathfrak{B}}\ X && = \forall A\,.\,Equal_{\mathfrak{A}}\ A \to Equal_{\mathfrak{B}}\ (X\ A).
\end{aligned}
$$

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.

$$
\begin{aligned}
&equal_{T::\mathfrak{T}} && :: Equal_{\mathfrak{T}}\ T \\
&equal_1\ u_1\ u_2 && = true \\
&equal_{Int}\ i_1\ i_2 && = equalInt\ i_1\ i_2 \\
&equal_+\ equal_A\ equal_B\ (inl\ a_1)\ (inl\ a_2) && = equal_A\ a_1\ a_2 \\
&equal_+\ equal_A\ equal_B\ (inl\ a_1)\ (inr\ b_2) && = false \\
&equal_+\ equal_A\ equal_B\ (inr\ b_1)\ (inl\ a_2) && = false \\
&equal_+\ equal_A\ equal_B\ (inr\ b_2)\ (inr\ b_2) && = equal_B\ b_1\ b_2 \\
&equal_\times\ equal_A\ equal_B\ (a_1, b_1)\ (a_2, b_2) && = equal_A\ a_1\ a_2 \wedge equal_B\ b_1\ b_2
\end{aligned}
$$

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

$$
equal_{F::\star\to\star} :: \forall A\,.\,(A \to A \to Bool) \to (F\ A \to F\ A \to Bool)
$$

This gives us an extra degree of flexibility: $equal_F\ op\ x_1\ x_2$ checks whether corresponding elements in x_1 and x_2 are related by op. Of course, op need not be an equality operator. PolyLib [17] defines an analogous function but with a more general type:

$$pequal_{F::\star\to\star} :: \forall A_1\ A_2 \,.\, (A_1 \to A_2 \to Bool) \to (F\ A_1 \to F\ A_2 \to Bool).$$

Here, the element types need not be identical. And, in fact, $equal_{T::\mathfrak{T}}$ can be assigned the more general type $PEqual_{\mathfrak{T}}\ T\ T$ given by

$$\begin{aligned}
&PEqual_{\mathfrak{T}::\Box} &&:: \mathfrak{T} \to \mathfrak{T} \to \star\\
&PEqual_{\star}\ X_1\ X_2 &&= X_1 \to X_2 \to Bool\\
&PEqual_{\mathfrak{A}\to\mathfrak{B}}\ X_1\ X_2 &&= \forall A_1\ A_2 \,.\, PEqual_{\mathfrak{A}}\ A_1\ A_2 \to PEqual_{\mathfrak{B}}\ (X_1\ A_1)\ (X_2\ A_2),
\end{aligned}$$

which gives us an even greater degree of flexibility.

7.2 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.

$$\mathbf{type}\ F_1 \stackrel{.}{\to} F_2 = \forall A \,.\, F_1\ A \to F_2\ A.$$

A natural transformation between functors F_1 and F_2 is simply a polymorphic function of type $F_1 \stackrel{.}{\to} F_2$. 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 $(F_1 \stackrel{.}{\to} F_2) \to (H\ F_1 \stackrel{.}{\to} H\ F_2)$. Now, the polytypic map gives us a function of type

$$\begin{aligned}
map_H :: \forall F_1\ F_2 \,.\, &(\forall B_1\ B_2 \,.\, (B_1 \to B_2) \to (F_1\ B_1 \to F_2\ B_2))\\
&\to (\forall A_1\ A_2 \,.\, (A_1 \to A_2) \to (H\ F_1\ A_1 \to H\ F_2\ A_2)).
\end{aligned}$$

Given a natural transformation α of type $F_1 \stackrel{.}{\to} F_2$ there are basically two alternatives for constructing the required function of type $\forall B_1\ B_2 \,.\, (B_1 \to B_2) \to (F_1\ B_1 \to F_2\ B_2)$: $\lambda h \,.\, \alpha \cdot map_{F_1}\ h$ or $\lambda h \,.\, map_{F_2}\ h \cdot \alpha$. The naturality of α, however, implies that both alternatives are equal. Consequently, the higher-order map is given by

$$
\begin{aligned}
&\mathit{hmap}_{H::(\star\to\star)\to\star\to\star} && :: \forall F_1\ F_2 \,.\, (F_1 \mathbin{\dot{\to}} F_2) \to (H\ F_1 \mathbin{\dot{\to}} H\ F_2)\\
&\mathit{hmap}_H\ (\alpha :: F_1 \mathbin{\dot{\to}} F_2) && = \mathit{map}_H\ (\lambda h \,.\, \alpha \cdot \mathit{map}_{F_1}\ h)\ \mathit{id}.
\end{aligned}
$$

Using polytypic definitions similar to the one in Section 2 we can also implement embedding-projection maps [14] of type $\mathit{MapE}_\star\ X_1\ X_2 = (X_1 \to X_2, X_2 \to X_1)$, monadic maps [7,25] of type $\mathit{MapM}_\star\ X_1\ X_2 = X_1 \to M\ X_2$ for some monad M, and arrow maps [18] of type $\mathit{MapA}_\star\ X_1\ X_2 = X_1 \leadsto X_2$ for some arrow type $(\leadsto)$.

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 $\mathit{List}\ A_1$ and $\mathit{List}\ A_2$ and applies the function to corresponding elements producing a list of type $\mathit{List}\ A_3$. The type of the polytypic *zip* is essentially a three parameter variant of *Map*.

$$
\begin{aligned}
&\mathit{Zip}_{\mathfrak{T}::\Box} && :: \mathfrak{T} \to \mathfrak{T} \to \mathfrak{T} \to \star\\
&\mathit{Zip}_\star\ X_1\ X_2\ X_3 && = X_1 \to X_2 \to X_3\\
&\mathit{Zip}_{\mathfrak{A}\to\mathfrak{B}}\ X_1\ X_2\ X_3 && = \forall A_1\ A_2\ A_3 \,.\, \mathit{Zip}_{\mathfrak{A}}\ A_1\ A_2\ A_3\\
& && \qquad\qquad \to \mathit{Zip}_{\mathfrak{B}}\ (X_1\ A_1)\ (X_2\ A_2)\ (X_3\ A_3)
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{zip}_{T::\mathfrak{T}} && :: \mathit{Zip}_{\mathfrak{T}}\ T\ T\ T\\
&\mathit{zip}_1\ ()\ () && = ()\\
&\mathit{zip}_{\mathit{Int}}\ i_1\ i_2 && = \mathbf{if}\ \mathit{equalInt}\ i_1\ i_2\ \mathbf{then}\ i_1\ \mathbf{else}\ \bot\\
&\mathit{zip}_+\ \mathit{zip}_A\ \mathit{zip}_B\ (\mathit{inl}\ a_1)\ (\mathit{inl}\ a_2) && = \mathit{inl}\ (\mathit{zip}_A\ a_1\ a_2)\\
&\mathit{zip}_+\ \mathit{zip}_A\ \mathit{zip}_B\ (\mathit{inl}\ a_1)\ (\mathit{inr}\ b_2) && = \bot\\
&\mathit{zip}_+\ \mathit{zip}_A\ \mathit{zip}_B\ (\mathit{inr}\ b_1)\ (\mathit{inl}\ a_2) && = \bot\\
&\mathit{zip}_+\ \mathit{zip}_A\ \mathit{zip}_B\ (\mathit{inr}\ b_1)\ (\mathit{inr}\ b_2) && = \mathit{inr}\ (\mathit{zip}_B\ b_1\ b_2)\\
&\mathit{zip}_\times\ \mathit{zip}_A\ \mathit{zip}_B\ (a_1, b_1)\ (a_2, b_2) && = (\mathit{zip}_A\ a_1\ a_2, \mathit{zip}_B\ b_1\ b_2)
\end{aligned}
$$

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 $\mathit{Zip}_\star\ X_1\ X_2\ X_3 = X_1 \to X_2 \to \mathit{Maybe}\ X_3$, which uses the exception monad *Maybe* to signal incompatibility of the argument structures, see [11].

A reduction or a crush [23] 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

$$\begin{aligned}
&\mathit{Count}_{\mathfrak{T}::\Box} &&:: \mathfrak{T} \to \star \\
&\mathit{Count}_{\star}\ X &&= X \to \mathit{Int} \\
&\mathit{Count}_{\mathfrak{A}\to\mathfrak{B}}\ X &&= \forall A\,.\,\mathit{Count}_{\mathfrak{A}}\ A \to \mathit{Count}_{\mathfrak{B}}\ (X\ A)
\end{aligned}$$

and here is its definition.

$$\begin{aligned}
&\mathit{count}_{T::\mathfrak{T}} &&:: \mathit{Count}_{\mathfrak{T}}\ T \\
&\mathit{count}_{1}\ u &&= 0 \\
&\mathit{count}_{\mathit{Int}}\ i &&= 1 \\
&\mathit{count}_{+}\ \mathit{count}_A\ \mathit{count}_B\ (\mathit{inl}\ a) &&= \mathit{count}_A\ a \\
&\mathit{count}_{+}\ \mathit{count}_A\ \mathit{count}_B\ (\mathit{inr}\ b) &&= \mathit{count}_B\ b \\
&\mathit{count}_{\times}\ \mathit{count}_A\ \mathit{count}_B\ (a, b) &&= \mathit{count}_A\ a + \mathit{count}_B\ b.
\end{aligned}$$

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

$$\begin{aligned}
&\mathit{size}_{T::\mathfrak{T}} &&:: \mathit{Count}_{\mathfrak{T}}\ T \\
&\mathit{size}_{1}\ u &&= 0 \\
&\mathit{size}_{\mathit{Int}}\ i &&= 0 \\
&\mathit{size}_{+}\ \mathit{size}_A\ \mathit{size}_B\ (\mathit{inl}\ a) &&= \mathit{size}_A\ a \\
&\mathit{size}_{+}\ \mathit{size}_A\ \mathit{size}_B\ (\mathit{inr}\ b) &&= \mathit{size}_B\ b \\
&\mathit{size}_{\times}\ \mathit{size}_A\ \mathit{size}_B\ (a, b) &&= \mathit{size}_A\ a + \mathit{size}_B\ b.
\end{aligned}$$

It is not hard to see that $\mathit{size}_T\ t$ returns 0 for all types T of kind $\star$ (provided t is finite and 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

$$\mathit{size}_{F::\star\to\star} :: \forall A\,.\,(A \to \mathit{Int}) \to (F\ A \to \mathit{Int})$$

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 $\forall A \,.\, F\ A \to \mathit{Int}$ that counts the number of values of type X in a given structure of type $F\ X$.

$$
\begin{aligned}
&\mathit{fsum}_{F::\star\to\star} &&:: F\ \mathit{Int} \to \mathit{Int}\\
&\mathit{fsum}_{F} &&= \mathit{size}_F\ \mathit{id}\\
&\mathit{fsize}_{F::\star\to\star} &&:: \forall A \,.\, F\ A \to \mathit{Int}\\
&\mathit{fsize}_{F} &&= \mathit{size}_F\ (\mathit{const}\ 1)
\end{aligned}
$$

Using a similar approach we can also flatten a structure into a list of elements. The type of the polytypic flattening function

$$
\begin{aligned}
&\mathit{Flatten}^{Z}_{\mathfrak{T}::\square} &&:: \mathfrak{T} \to \star\\
&\mathit{Flatten}^{Z}_{\star}\ X &&= X \to [Z]\\
&\mathit{Flatten}^{Z}_{\mathfrak{A}\to\mathfrak{B}}\ X &&= \forall A \,.\, \mathit{Flatten}^{Z}_{\mathfrak{A}}\ A \to \mathit{Flatten}^{Z}_{\mathfrak{B}}\ (X\ A)
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{flatten}_{T::\mathfrak{T}} &&:: \forall Z \,.\, \mathit{Flatten}^{Z}_{\mathfrak{T}}\ T\\
&\mathit{flatten}_{1}\ () &&= [\,]\\
&\mathit{flatten}_{\mathit{Int}}\ i &&= [\,]\\
&\mathit{flatten}_{+}\ \mathit{flatten}_A\ \mathit{flatten}_B\ (\mathit{inl}\ a) &&= \mathit{flatten}_A\ a\\
&\mathit{flatten}_{+}\ \mathit{flatten}_A\ \mathit{flatten}_B\ (\mathit{inr}\ b) &&= \mathit{flatten}_B\ b\\
&\mathit{flatten}_{\times}\ \mathit{flatten}_A\ \mathit{flatten}_B\ (a, b) &&= \mathit{flatten}_A\ a +\!\!+ \mathit{flatten}_B\ b
\end{aligned}
$$

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.

$$
\begin{aligned}
&\mathit{fflatten}_{F::\star\to\star} &&:: \forall A \,.\, F\ A \to \mathit{List}\ A\\
&\mathit{fflatten}_{F} &&= \mathit{flatten}_F\ \mathit{wrap}\ \textbf{where}\ \mathit{wrap}\ a = \mathit{cons}\ a\ \mathit{nil}
\end{aligned}
$$

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.

$$
\begin{array}{ll}
\mathit{Reduce}^{Z}_{\mathfrak{T}::\Box} & :: \mathfrak{T} \to \star \\
\mathit{Reduce}^{Z}_{\star}\ X & = X \to Z \\
\mathit{Reduce}^{Z}_{\mathfrak{A}\to\mathfrak{B}}\ X & = \forall A\,.\,\mathit{Reduce}^{Z}_{\mathfrak{A}}\ A \to \mathit{Reduce}^{Z}_{\mathfrak{B}}\ (X\ A) \\
\mathit{reduce}_{T::\mathfrak{T}} & :: \forall Z\,.\,Z \to (Z \to Z \to Z) \to \mathit{Reduce}^{Z}_{\mathfrak{T}}\ T \\
\mathit{reduce}_{T}\ e\ \mathit{op} & = \mathit{red}_{T} \\
\quad \mathbf{where} & \\
\quad \mathit{red}_{T::\mathfrak{T}} & :: \mathit{Reduce}^{Z}_{\mathfrak{T}}\ T \\
\quad \mathit{red}_{1}\ () & = e \\
\quad \mathit{red}_{\mathit{Int}}\ i & = e \\
\quad \mathit{red}_{+}\ \mathit{red}_{A}\ \mathit{red}_{B}\ (\mathit{inl}\ a) & = \mathit{red}_{A}\ a \\
\quad \mathit{red}_{+}\ \mathit{red}_{A}\ \mathit{red}_{B}\ (\mathit{inr}\ b) & = \mathit{red}_{B}\ b \\
\quad \mathit{red}_{\times}\ \mathit{red}_{A}\ \mathit{red}_{B}\ (a, b) & = \mathit{op}\ (\mathit{red}_{A}\ a)\ (\mathit{red}_{B}\ b)
\end{array}
$$

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

$$
\begin{array}{ll}
\mathit{red}_{1} & = \mathit{const}\ e \\
\mathit{red}_{\mathit{Int}} & = \mathit{const}\ e \\
\mathit{red}_{+}\ \mathit{red}_{A}\ \mathit{red}_{B} & = \mathit{red}_{A} \triangledown \mathit{red}_{B} \\
\mathit{red}_{\times}\ \mathit{red}_{A}\ \mathit{red}_{B} & = \mathit{uncurry}\ \mathit{op} \cdot (\mathit{red}_{A} \times \mathit{red}_{B})
\end{array}
$$

Here, $(\triangledown)$ is the so-called junction operator [2]. The type of reduce_{F} where F is a unary type constructor is quite general.

$$\mathit{reduce}_{F::\star\to\star} :: \forall Z\,.\,Z \to (Z \to Z \to Z) \to (\forall A\,.\,(A \to Z) \to (F\ A \to Z))$$

Fig. 6 lists some typical applications of reduce_{F} and reduce_{G} where G is a binary type constructor. Further examples can be found, for instance, in [23] and [17].

8 Properties of polytypic values

This section investigates another important aspect of polytypism: polytypic reasoning. If you want to prove a property of a polytypic value, you have to reason polytypically. Like the program the proof will be parametric in the underlying data type. This section introduces a fundamental polytypic proof method based on logical relations. The section is structured as follows. Section 8.1 shows how to generalize the functorial laws to data types of arbitrary

$$
\begin{array}{ll}
\mathit{fsum}_{F::\star\to\star} & :: \forall N\,.\,(\mathit{Num}\ N) \Rightarrow F\ N \to N \\
\mathit{fsum}_{F} & = \mathit{reduce}_F\ 0\ (+)\ \mathit{id} \\
\mathit{fsize}_{F::\star\to\star} & :: \forall A\,.\,(\mathit{Num}\ N) \Rightarrow F\ A \to N \\
\mathit{fsize}_{F} & = \mathit{reduce}_F\ 0\ (+)\ (\mathit{const}\ 1) \\
\mathit{fand}_{F::\star\to\star} & :: F\ \mathit{Bool} \to \mathit{Bool} \\
\mathit{fand}_{F} & = \mathit{reduce}_F\ \mathit{true}\ (\wedge)\ \mathit{id} \\
\mathit{fall}_{F::\star\to\star} & :: \forall A\,.\,(A \to \mathit{Bool}) \to (F\ A \to \mathit{Bool}) \\
\mathit{fall}_{F}\ p & = \mathit{reduce}_F\ \mathit{true}\ (\wedge)\ p \\
\mathit{fflatten}_{F::\star\to\star} & :: \forall A\,.\,F\ A \to [A] \\
\mathit{fflatten}_{F} & = \mathit{reduce}_F\ [\,]\ (+\!\!+)\ \mathit{wrap} \\
\mathit{biflatten}_{G::\star\to\star\to\star} & :: \forall A\ B\,.\,G\ A\ B \to [A+B] \\
\mathit{biflatten}_{G} & = \mathit{reduce}_G\ [\,]\ (+\!\!+)\ (\mathit{wrap}\cdot \mathit{inl})\ (\mathit{wrap}\cdot \mathit{inr}) \\
\mathbf{data}\ \mathit{Shape}\ A & = \mathit{empty} \mid \mathit{var}\ A \mid \mathit{bin}\ (\mathit{Shape}\ A)\ (\mathit{Shape}\ A) \\
\mathit{shape}_{F::\star\to\star} & :: \forall A\,.\,F\ A \to \mathit{Tree}\ A \\
\mathit{shape}_{F} & = \mathit{reduce}_F\ \mathit{empty}\ \mathit{bin}\ \mathit{var}
\end{array}
$$

Fig. 6. Examples of reductions.

kinds and sketches the proof of correctness. Section 8.2 explains how to deal with fixed point operators in a generic way. Further examples of polytypic proofs are provided in Section 8.3.

8.1 Functorial laws

To classify as a functor the mapping function of a unary type constructor must satisfy the so-called functorial laws (for reasons of readability we switch again to Haskell syntax):

$$
\begin{aligned}
\mathit{map}_F\ \mathit{id} &= \mathit{id} \\
\mathit{map}_F\ (\varphi \cdot \psi) &= \mathit{map}_F\ \varphi \cdot \mathit{map}_F\ \psi\ ,
\end{aligned}
$$

that is, map_F preserves identity and composition. If the type constructor is binary, the functor laws take the form

$$
\begin{aligned}
\mathit{map}_G\ \mathit{id}\ \mathit{id} &= \mathit{id} \\
\mathit{map}_G\ (\varphi_1 \cdot \psi_1)\ (\varphi_2 \cdot \psi_2) &= \mathit{map}_G\ \varphi_1\ \varphi_2 \cdot \mathit{map}_G\ \psi_1\ \psi_2\ .
\end{aligned}
$$

How can we generalize these laws to data types of arbitrary kinds? Since map_T 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_T does not preserve identity; it *is* the identity.

$$\begin{aligned} &map_T = id \\ &map_T = map_T \cdot map_T \end{aligned}$$

The pendant of the second law states that map_T is idempotent (which is a simple consequence of the first law). These 'base cases' suggest to rephrase the functorial laws as logical relations. Let $\mathcal{M}$ be the model induced by map. The polytypic version of the first functorial law then states that $\mathcal{M}[\![T :: \mathfrak{T}]\!] \in \mathcal{I}^{\mathfrak{T}}$ for all closed type terms T, where the unary logical relation $\mathcal{I}$ is given by (we write $[\,t\,]$ simply as t)

$$\begin{aligned} &\mathcal{I}^{\mathfrak{T}} && \subseteq \ \mathbf{M}^{\mathfrak{T}} \\ &(D, R; m) \in \mathcal{I}^{\star} && \equiv \quad D = R \ \cap \ m = id :: D \to R. \end{aligned}$$

Similarly, the polytypic version of the second functorial law expresses that $(\mathcal{M}[\![T :: \mathfrak{T}]\!], \mathcal{M}[\![T :: \mathfrak{T}]\!], \mathcal{M}[\![T :: \mathfrak{T}]\!]) \in \mathcal{C}^{\mathfrak{T}}$ for all closed type terms T, where the ternary logical relation $\mathcal{C}$ is given by

$$\begin{aligned} &\mathcal{C}^{\mathfrak{T}} \ \subseteq \ \mathbf{M}^{\mathfrak{T}} \times \mathbf{M}^{\mathfrak{T}} \times \mathbf{M}^{\mathfrak{T}} \\ &((D_1, R_1; m_1), (D_2, R_2; m_2), (D_3, R_3; m_3)) \in \mathcal{C}^{\star} \\ &\qquad \equiv \ R_1 = D_2 \ \cap \ D_3 = D_1 \ \cap \ R_3 = R_2 \ \cap \ m_2 \cdot m_1 = m_3 :: D_3 \to R_3. \end{aligned}$$

The reader should convince herself that the monotypic functorial laws are indeed instances of the polytypic laws.

Turning to the proof of the polytypic laws we must show that the logical relations $\mathcal{I}$ and $\mathcal{C}$ relate type constants, that is, $\mathbf{const}_{\mathcal{M}}(C) \in \mathcal{I}^{Const(C)}$ and $(\mathbf{const}_{\mathcal{M}}(C), \mathbf{const}_{\mathcal{M}}(C), \mathbf{const}_{\mathcal{M}}(C)) \in \mathcal{C}^{Const(C)}$. It is straightforward to establish these conditions for the type constants '1', *Int*, '+', and '$\times$' as the properties are either trivial or follow directly from the functorial laws of '+' and '$\times$'. It remains to verify the conditions for the fixed point operators. Recall that the specialization of the fixed point operator $Fix_{\mathfrak{T}}$ is the same for all polytypic values. This suggests that we ought to be able to construct a generic proof that is independent of a particular logical relation. That is what we turn our attention to now.

The usual approach is to impose two further conditions on logical relations: they must be pointed and directed complete [1]. Then one can invoke fixed point induction (also known as Scott induction) to show that fixed point operators are logically related. However, one quickly realizes that the ordinary fixed point induction rule is not sufficient for this purpose. Consider proving $\textbf{const}_{\mathcal{M}}(\mathit{Fix}_{\mathfrak{T}}) \in \mathcal{I}^{\mathfrak{T}}$: under the precondition that $\mathit{map}_T = \mathit{id} \supset \mathit{map}_F\ T\ T\ \mathit{map}_T = \mathit{id}$ for all $\mathit{map}_T :: T \to T$ we have to show that $\mathit{fix}\ (\mathit{map}_F\ (\mathit{Fix}\ F)\ (\mathit{Fix}\ F)) = \mathit{id}$ for all $\mathit{map}_F :: \forall A_1\ A_2 . (A_1 \to A_2) \to (F\ A_1 \to F\ A_2)$. Now, note that fixed points are formed both on the term and on the type level whereas the standard fixed point induction rule deals with the term level only.

A way out of this dilemma is to postulate the following variant of the fixed point induction rule, which we call polytypic fixed point induction rule for want of a better name. Let P be an equation or a conjunction of equations, then

$$\frac{\begin{array}{l}\Gamma \vdash P[A_1 := 0, \ldots, A_n := 0, a := \bot] \\ \Gamma, P[A_1 := C_1, \ldots, A_n := C_n, a := c] \vdash \\ \qquad\qquad P[A_1 := F_1\ C_1, \ldots, A_n := F_n\ C_n, a := f\ C_1\ \ldots\ C_n\ c]\end{array}}{\Gamma \vdash P[A_1 = \mathit{Fix}\ F_1, \ldots, A_n := \mathit{Fix}\ F_n, a := \mathit{fix}\ (f\ (\mathit{Fix}\ F_1)\ (\mathit{Fix}\ F_n))]}$$

The first hypothesis formalizes that P is pointed. Note that in the second hypothesis the type constants $C_1, \ldots, C_n$ and the term constant c must not appear in Γ. This is a way of expressing universal quantification in our equational setting. Given this rule it is not hard to show that an arbitrary unary logical relation relates fixed points provided the relation is pointed, that is, $(0, \ldots, 0; \bot) \in \mathcal{R}^\star$. For k-ary logical relation we require a k-argument generalization of the above rule.

Two remarks are in order. First, though the polytypic fixed point induction rule may look unusual, it is, in fact, a special instance of the rule for simultaneous fixed points (if we treat types and terms alike, see below). To highlight this connection consider solving the recursion equations $a_1 = f_1\ a_1; \ldots; a_n = f_n\ a_n; a = f\ a_1\ \ldots\ a_n\ a$. Since the $a_1, \ldots, a_n$ do not depend on each other or on a, we can solve the equations using iterated fixed points: $a_1 = \mathit{fix}\ f_1; \ldots; a_n = \mathit{fix}\ f_n; a = \mathit{fix}\ (f\ (\mathit{fix}\ f_1)\ \ldots\ (\mathit{fix}\ f_n))$. Now, this is exactly the form used in the conclusion of the rule above.

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

Second, if we want to provide a model for our variant of the polymorphic lambda calculus, then the model must, of course, satisfy the postulated polytypic fixed point induction rule. Suitable models are, for instance, models based on universal domains such as $P\omega$, see [1,27]. These models allow to interpret types as certain elements (closures or finitary projections) of the universal domain, so that type recursion can be interpreted by the (untyped) least fixed point operator operator. Then the polytypic fixed point induction rule is, in fact, a variant of the rule for simultaneous fixed points.

8.3 Examples

8.3.1 Mapping functions

The functorial laws are captured by the logical relations $\mathcal{I}$ and $\mathcal{C}$, which we have discussed at length in Section 8.1.

$$\begin{aligned} &\mathcal{I}^{\mathfrak{T}} && \subseteq \ \mathbf{M}^{\mathfrak{T}} \\ &(D, R; m) \in \mathcal{I}^{\star} && \equiv \quad D = R \ \cap \ m = id :: D \to R. \end{aligned}$$

$$\begin{aligned} &\mathcal{C}^{\mathfrak{T}} \ \subseteq \ \mathbf{M}^{\mathfrak{T}} \times \mathbf{M}^{\mathfrak{T}} \times \mathbf{M}^{\mathfrak{T}} \\ &((D_1, R_1; m_1), (D_2, R_2; m_2), (D_3, R_3; m_3)) \in \mathcal{C}^{\star} \\ &\qquad \equiv \ R_1 = D_2 \ \cap \ D_3 = D_1 \ \cap \ R_3 = R_2 \ \cap \ m_2 \cdot m_1 = m_3 :: D_3 \to R_3. \end{aligned}$$

It remains to show that $\mathcal{I}$ and $\mathcal{C}$ are pointed, that is, $\perp = id :: 0 \to 0$ and $\perp \cdot \perp = \perp :: 0 \to 0$. The second equation is trivially true and the first is a simple consequence of $t = \perp :: 0$.

8.3.2 Reductions

Using a minor variant of $\mathcal{C}$ we can also relate reductions and mapping functions. Let $\mathcal{R}_Z$ be the applicative structure induced by *reduce*. Like the polykinded type $Reduce_{\mathfrak{T}}^{Z}$ the structure is parameterized by the result type of *reduce*. The ternary logical relation $\mathcal{F}_Z$ is defined

$$\begin{aligned} &\mathcal{F}_Z^{\mathfrak{T}} \ \subseteq \ \mathbf{R}_Z^{\mathfrak{T}} \times \mathbf{M}^{\mathfrak{T}} \times \mathbf{R}_Z^{\mathfrak{T}} \\ &((T; r), (D, R; m), (U; s)) \in \mathcal{F}_Z^{\star} \\ &\qquad \equiv \ D = U \ \cap \ R = T \ \cap \ r \cdot m = s :: U \to Z. \end{aligned}$$

Now, given an element $e :: Z$ and an operation $op :: Z \to Z \to Z$, we have

$$(\mathcal{R}_{Z,e,op}[\![T :: \mathfrak{T}]\!], \mathcal{M}[\![T :: \mathfrak{T}]\!], \mathcal{R}_{Z,e,op}[\![T :: \mathfrak{T}]\!]) \in \mathcal{F}_Z^{\mathfrak{T}}.$$

Note that the interpretation is additionally parameterized by the element e and the operation op. An immediate consequence of this property is

$$reduce_F\ e\ op\ \varphi \cdot map_F\ \psi = reduce_F\ e\ op\ (\varphi \cdot \psi)\ ,$$

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 $C \in dom(Const)$. Using the point-free definitions of map and red this amounts to showing that

$$\begin{aligned} const\ e \cdot id &= const\ e \\ (\varphi_1 \triangledown \varphi_2) \cdot (\psi_1 + \psi_2) &= (\varphi_1 \cdot \psi_1) \triangledown (\varphi_2 \cdot \psi_2) \\ uncurry\ op \cdot (\varphi_1 \times \varphi_2) \cdot (\psi_1 \times \psi_2) &= uncurry\ op \cdot ((\varphi_1 \cdot \psi_1) \times (\varphi_2 \cdot \psi_2))\ . \end{aligned}$$

All three conditions hold.

Previous approaches to polytypic programming [16,11] 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 $fsize_{F_1 \cdot F_2} = fsum_{F_1} \cdot map_{F_1}\ (fsize_{F_2})$. Interestingly, using $reduce$-map fusion this equation can be derived from the definitions of $fsize$ and $fsum$ given in Fig. 6.

$$\begin{aligned} & fsize_{F_1 \cdot F_2} \\ = \ & \{ \text{ definition } fsize \ \} \\ & reduce_{F_1 \cdot F_2}\ 0\ (+)\ (const\ 1) \\ = \ & \{ \ poly_{F_1 \cdot F_2} = poly_{F_1} \cdot poly_{F_2} \ \} \\ & reduce_{F_1}\ 0\ (+)\ (reduce_{F_2}\ 0\ (+)\ (const\ 1)) \\ = \ & \{ \text{ definition } fsize \ \} \\ & reduce_{F_1}\ 0\ (+)\ (fsize_{F_2}) \\ = \ & \{ \ reduce\text{-}map \text{ fusion } \} \\ & reduce_{F_1}\ 0\ (+)\ id \cdot map_{F_1}\ (fsize_{F_2}) \\ = \ & \{ \text{ definition } fsum \ \} \\ & fsum_{F_1} \cdot map_{F_1}\ (fsize_{F_2}) \end{aligned}$$

As a final example let us generalize the fusion law for reductions given by L. Meertens in [23]. To this end we use the logical relation $\mathcal{V}_{Z,Z',h}$ defined by

$$\begin{aligned} \mathcal{V}^{\mathfrak{T}}_{Z,Z',h} &\subseteq \mathbf{R}^{\mathfrak{T}}_{Z} \times \mathbf{R}^{\mathfrak{T}}_{Z'} \\ ((T;r),(U;s)) \in \mathcal{V}^{\star}_{Z,Z',h} &\equiv\ T = U\ \cap\ h \cdot r = s :: U \to Z'. \end{aligned}$$

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

$$
\begin{aligned}
&\quad h \perp = \perp \\
&\cap\, h\ e = e' \\
&\cap\, h\ (op\ x\ y) = op'\ (h\ x)\ (h\ y) \\
&\supset (\mathcal{R}_{Z,e,op}[\![T :: \mathfrak{T}]\!], \mathcal{R}_{Z',e',op'}[\![T :: \mathfrak{T}]\!]) \in \mathcal{V}^{\mathfrak{T}}_{Z,Z',h}.
\end{aligned}
$$

We can apply this law, for instance, to prove that $length \cdot fflatten_F = fsize_F$.

9 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 [16,12] were restricted in that they only allowed to parameterize values by types of one fixed kind. Three notable exceptions are Functorial ML (FML) [19], the work of F. Ruehr [33], and the work of P. Hoogendijk and R. Backhouse [15]. 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 $(\mathfrak{A} \rightarrow \star) \rightarrow \star$, where $\mathfrak{A}$ 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. Finally, definitions of polytypic values that are indexed by relators of different arities can be found in the work of P. Hoogendijk and R. Backhouse on commuting data types [15].

The results in this paper improve upon my earlier work on polytypic programming [12] in the following respects. As remarked above the previous work considered only polytypic values indexed by types of one fixed kind. Furthermore, the approach could only handle type indices of second-order kind or less and type constants (that is, primitive type constructors) were restricted to first-order kind or kind $\star$. Using polykinded types all these restrictions can be dropped.

An earlier version of this paper appeared in [13]. The main improvement over the earlier version is that we phrase the specialization of a polytypic value as a model of the simply typed lambda calculus. As a major benefit of this approach we can now use standard logical relations to state and to prove properties of polytypic values, whereas in the conference version we had to introduce a tailor-made, but somewhat ad-hoc variant of logical relations. The definition of the applicative structure in Section 6 is heavily inspired by E. Moggi's module categories, see [28], which are used to provide a category-theoretic explanation of Standard ML's module system.

10 Conclusion

Haskell possesses a rich type system, which essentially corresponds 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 turns out 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 phrased 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. Thanks are furthermore due to Fritz Ruehr for his helpful comments on an earlier draft of this paper. Furthermore, I am grateful to the five anonymous referees of MPC 2000, who went over and above the call of duty, providing numerous constructive comments for the revision of the conference version of this paper. Finally, I am indebted to Eugenio Moggi, who suggested to use module categories for the specialization of polytypic values.

References

[1] Roberto Amadio, Kim B. Bruce, and Giuseppe Longo. The finitary projection model for second order lambda calculus and solutions to higher order domain equations. In *Proceedings of the Symposium on Logic in Computer Science, Cambridge, Massachusetts*, pages 122–130. IEEE Computer Society, June 1986.

[2] Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic Programming — An Introduction —. In S. Doaitse Swierstra, Pedro R. Henriques, and Jose N. Oliveira, editors, *3rd International Summer School on Advanced Functional Programming, Braga, Portugal*, volume 1608 of *Lecture Notes in Computer Science*, pages 28–115. Springer-Verlag, Berlin, 1999.

[3] Richard Bird, Oege de Moor, and Paul Hoogendijk. Generic functional programming with types and relations. *Journal of Functional Programming*, 6(1):1–28, January 1996.

[4] Richard Bird and Lambert Meertens. Nested datatypes. In J. Jeuring, editor, *Fourth International Conference on Mathematics of Program Construction, MPC'98, Marstrand, Sweden*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer-Verlag, June 1998.

[5] Richard Bird and Ross Paterson. Generalised folds for nested datatypes. *Formal Aspects of Computing*, 11(2):200–222, 1999.

[6] Olivier Danvy. An extensional characterization of lambda-lifting and lambda-dropping. In Aart Middeldorp and Taisuke Sato, editors, *4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99), Tsukuba, Japan*, volume 1722 of *Lecture Notes in Computer Science*, pages 241–250. Springer-Verlag, November 1999.

[7] M.M. Fokkinga. Monadic maps and folds for arbitrary datatypes. Technical Report Memoranda Informatica 94-28, University of Twente, June 1994.

[8] Vladimir Gapeyev, Michael Y. Levin, and Benjamin C. Pierce. Recursive subtyping revealed (functional pearl). In *Proceedings of the ACM Sigplan International Conference on Functional Programming (ICFP-00)*, volume 35 of *ACM Sigplan Notices*, pages 221–232, New York, September 2000. ACM Press.

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

[10] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[11] Ralf Hinze. Polytypic functions over nested datatypes. *Discrete Mathematics and Theoretical Computer Science*, 3(4):193–214, September 1999.

[12] Ralf Hinze. A new approach to generic functional programming. In Thomas W. Reps, editor, *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL' 00), Boston, Massachusetts, January 19-21*, pages 119–132, January 2000.

[13] 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), July 3-5, 2000*, volume 1837 of *Lecture Notes in Computer Science*, pages 2–27. Springer-Verlag, July 2000.

[14] Ralf Hinze. Polytypic programming with ease. *Journal of Functional and Logic Programming*, 2001. To appear.

[15] Paul Hoogendijk and Roland Backhouse. When do datatypes commute? In Eugenio Moggi and Giuseppe Rosolini, editors, *Proceedings of the 7th International Conference on Category Theory and Computer Science (Santa Margherita Ligure, Italy, September 4–6)*, volume 1290 of *Lecture Notes in Computer Science*, pages 242–260. Springer-Verlag, 1997.

[16] Patrik Jansson and Johan Jeuring. PolyP—a polytypic programming language extension. In *Conference Record 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'97, Paris, France*, pages 470–482. ACM-Press, January 1997.

[17] Patrik Jansson and Johan Jeuring. PolyLib—A library of polytypic functions. In Roland Backhouse and Tim Sheard, editors, *Informal Proceedings Workshop on Generic Programming, WGP'98, Marstrand, Sweden.* Department of Computing Science, Chalmers University of Technology and Göteborg University, June 1998.

[18] Patrik Jansson and Johan Jeuring. Calculating polytypic data conversion programs. *Science of Computer Programmming*, 2000. To appear.

[19] C.B. Jay, G. Bellè, and E. Moggi. Functorial ML. *Journal of Functional Programming*, 8(6):573–619, November 1998.

[20] M.P. Jones and J.C. Peterson. *Hugs 98 User Manual*, May 1999. Available from `http://www.haskell.org/hugs`.

[21] Daniel Leivant. Polymorphic type inference. In *Proc. 10th Symposium on Principles of Programming Languages*, 1983.

[22] Nancy Jean McCracken. *An Investigation of a Programming Language with a Polymorphic Type Structure.* PhD thesis, Syracuse University, June 1979.

[23] Lambert Meertens. Calculate polytypically! In H. Kuchen and S.D. Swierstra, editors, *Proceedings 8th International Symposium on Programming Languages: Implementations, Logics, and Programs, PLILP'96, Aachen, Germany*, volume 1140 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, September 1996.

[24] Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to exponential types. In *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*, pages 324–333. ACM-Press, June 1995.

[25] Erik Meijer and Johan Jeuring. Merging monads and folds for functional programming. In J. Jeuring and E. Meijer, editors, *1st International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden*, volume 925 of *Lecture Notes in Computer Science*, pages 228–266. Springer-Verlag, Berlin, 1995.

[26] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[27] John C. Mitchell. *Foundations for Programming Languages.* The MIT Press, Cambridge, MA, 1996.

[28] Eugenio Moggi. A cateogry-theoretic account of program modules. *Mathematical Structures in Computer Science*, 1(1):103–139, March 1991.

[29] Alan Mycroft. Polymorphic type schemes and recursive definitions. In M. Paul and B. Robinet, editors, *Proceedings of the International Symposium on Programming, 6th Colloquium, Toulouse, France*, volume 167 of *Lecture Notes in Computer Science*, pages 217–228, 1984.

[30] Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998.

[31] Simon Peyton Jones and John Hughes, editors. *Haskell 98 — A Non-strict, Purely Functional Language*, February 1999. Available from `http://www.haskell.org/definition/`.

[32] Simon L. Peyton Jones. Compiling Haskell by program transformation: A report from the trenches. In Hanne Riis Nielson, editor, *Programming Languages and Systems—ESOP'96, 6th European Symposium on Programming, Linköping, Sweden, 22–24 April*, volume 1058 of *Lecture Notes in Computer Science*, pages 18–44. Springer-Verlag, 1996.

[33] Karl Fritz Ruehr. *Analytical and Structural Polymorphism Expressed using Patterns over Types.* PhD thesis, University of Michigan, 1992.

[34] The GHC Team. *The Glasgow Haskell Compiler User's Guide, Version 4.04*, September 1999. Available from `http://www.haskell.org/ghc/documentation.html`.

[35] Philip Wadler. Theorems for free! In *The Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA'89), London, UK*, pages 347–359. Addison-Wesley Publishing Company, September 1989.