A New Approach to Generic Functional Programming

Ralf Hinze

Institut für Informatik III
Universität Bonn
Römerstraße 164, 53117 Bonn, Germany

E-mail: `ralf@informatik.uni-bonn.de`
Homepage: `http://www.informatik.uni-bonn.de/~ralf/`

Abstract

This paper describes a new approach to generic functional programming, which allows us to define functions generically for all datatypes expressible in Haskell. A generic function is one that is defined by induction on the structure of types. Typical examples include pretty printers, parsers, and comparison functions. The advanced type system of Haskell presents a real challenge: datatypes may be parameterized not only by types but also by type constructors, type definitions may involve mutual recursion, and recursive calls of type constructors can be arbitrarily nested. We show that—despite this complexity—a generic function is uniquely defined by giving cases for primitive types and type constructors (such as disjoint unions and cartesian products). Given this information a generic function can be specialized to arbitrary Haskell datatypes. The key idea of the approach is to model types by terms of the simply typed λ-calculus augmented by a family of recursion operators. While conceptually simple, our approach places high demands on the type system: it requires polymorphic recursion, rank-n types, and a strong form of type constructor polymorphism. Finally, we point out connections to Haskell's class system and show that our approach generalizes type classes in some respects.

1 Introduction

Programming is sometimes a challenge and sometimes a nuisance. Recurring routine tasks such as changing the representation of data, converting the internal representation into a human readable form (pretty printing) or vice versa (parsing) fall into the latter category. This is the domain of generic programming, which aims at relieving the programmer from repeatedly writing functions of similar functionality for different user-defined datatypes. A generic functional program is essentially a collection of type-indexed values—typically functions—that are defined by induction on the structure of types. A generic function such as a pretty printer or a parser is written once and for all times; its specialization to different instances of datatypes happens without further effort from the user. This way generic programming greatly simplifies the construction and maintenance of software systems as it automatically adapts functions to changes in the representation of data.

Different approaches to generic functional programming differ in the way the structure of datatypes is modelled. The language extension PolyP [21], for instance, is based on the initial algebra semantics of datatypes. An unfortunate consequence of this choice is that the class of datatypes is restricted to so-called regular datatypes. A regular datatype is a parameterized type whose definition neither contains function spaces nor recursive calls which involve a change of the type parameter(s). Thus, compared to the class of datatypes definable in Haskell 98 [37] PolyP covers only a small, though important subset. This is a great pity. Especially, because the importance of generic programming increases with the complexity of types. We will get to know several involved datatypes, where even the skilled Haskell programmer will experience considerable hardship when trying to implement, say, comparison functions for these types.

Admittedly, Haskell has a very rich type system, which presents a real challenge for implementors of generic programming extensions: datatypes may be parameterized not only by types but also by type constructors, type definitions may involve mutual recursion, and recursive calls of type constructors can be arbitrarily nested. If we aim at extending generic programming to the full type system of Haskell, we are well-advised to model datatypes as closely and as faithfully as possible. Now, since Haskell supports both type application and (an implicit form of) type abstraction, type terms correspond essentially to terms of the simply typed λ-calculus augmented by a family of recursion operators and by additional constants denoting primitive types (such as integers) and primitive type constructors (such as disjoint unions and cartesian products). The 'types' of type terms are described using so-called kinds [28].

Though Haskell's type system is quite involved, defining generic values is comparatively easy. We will see that a generic value is uniquely defined by giving cases for primitive types and primitive type constructors. Given this information a generic value can be specialized to arbitrary Haskell datatypes. Interestingly, the process of specialization can be seen as an interpretation of simply typed λ-terms.

The main contributions of this paper are the following.

- ☐ We explain how to define values indexed by types or by type constructors of first-order kind.
- ☐ We show how to specialize generic values to concrete instances of datatypes. We identify several program-

ming language features that are required to make this translation work.

- □ We point out connections to Haskell's type classes [27] and show that our approach generalizes type classes in some respects.

The rest of this paper is organized as follows. Sec. 2 introduces Haskell's type system by means of some examples. Sec. 3 sketches our approach to generic functional programming. Sec. 4 introduces the simply typed λ-calculus and makes explicit how datatypes are modelled by λ-terms. Sec. 5 shows how to define type-indexed values and explains how to specialize a type-indexed value to concrete instances of datatypes. Sec. 6 generalizes the approach to type constructors of first-order kind. Sec. 7 works towards encoding instances of generic values in Haskell and relates type-indexed definitions to type classes. Finally, Sec. 8 reviews related work and Sec. 9 concludes.

2 Haskell's type system

This section introduces the main features of Haskell's type system by means of some examples. It is not important to understand the examples in full detail. The purpose of this section is rather to demonstrate the expressiveness of Haskell's type system and to give evidence that real use has been made of these features.

As first, rather simple examples consider the datatypes of lists and rose trees [4].

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

The two equations introduce two type constructors, *List* and *Rose*, of kind $\star \to \star$. The kind system of Haskell specifies the 'type' of a type constructor [28]. The '$\star$' kind represents nullary constructors like *Int*. The kind $\kappa_1 \to \kappa_2$ represents type constructors that map type constructors of kind κ_1 to those of kind κ_2. The kind system is necessary since Haskell supports abstraction over type constructors of arbitrary kind. The following generalization of rose trees illustrates this feature.

$$\textbf{data}\ \mathit{GRose}\ f\ a = \mathit{GBranch}\ a\ (f\ (\mathit{GRose}\ f\ a))$$

A slight variant of this definition has been used in [34] to extend an implementation of priority queues with an efficient merge operation. The type constructor *GRose* has kind $(\star \to \star) \to (\star \to \star)$ and is related to *Rose* by $\mathit{GRose}\ \mathit{List} = \mathit{Rose}$. Note that this equation states an isomorphism rather than an equality since Haskell's type system is based on name equivalence rather than on structural equivalence. Furthermore, note that *GRose* has a second-order kind. The order of a kind is given by $\mathit{order}(\star) = 0$ and $\mathit{order}(\kappa_1 \to \kappa_2) = \mathit{max}\{1 + \mathit{order}(\kappa_1), \mathit{order}(\kappa_2)\}$.

The following definition introduces a fixpoint operator on the level of types. This definition appears, for instance, in [32] where it is employed to give a generic definition of so-called cata- and anamorphisms.

$$\begin{aligned}\textbf{data}\ \mathit{Fix}\ f &= \mathit{In}\ (f\ (\mathit{Fix}\ f))\\ \textbf{data}\ \mathit{BaseList}\ a\ b &= \mathit{Nil} \mid \mathit{Cons}\ a\ b\end{aligned}$$

The kinds of these types are given by $\mathit{Fix} :: (\star \to \star) \to \star$ and $\mathit{BaseList} :: \star \to (\star \to \star)$. Note that binary type constructors like *BaseList* are, in fact, curried. Using *Fix* and *BaseList* the datatype of polymorphic lists can alternatively be defined by $\mathit{List}\ a = \mathit{Fix}\ (\mathit{BaseList}\ a)$.

The type terms on the right-hand side of datatype definitions may be arbitrarily nested. The following equations, which implement binary random-access lists [34], exemplify nested type terms involving recursion.

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

Since the type parameter of *Sequ* is changed in the recursive calls, *Sequ* is termed a nested or non-regular datatype [5]. Nested datatypes are practically important since they can capture data-structural invariants in a way that regular datatypes cannot. For instance, *Sequ* captures the invariant that binary random-access lists are sequences of perfect binary leaf trees stored in increasing order of height. We refer the interested reader to [34, 7, 6] for further examples of nested types.

The following equations employ both abstraction over type constructors and nested recursion.

$$\begin{aligned}&\textbf{data}\ \mathit{MapFork}\ m\ v\\ &\quad = \mathit{TrieFork}\ (m\ (m\ v))\\ &\textbf{data}\ \mathit{MapSequ}\ m\ v\\ &\quad = \mathit{TrieSequ}\ v\\ &\qquad\qquad (\mathit{MapSequ}\ (\mathit{MapFork}\ m)\ v)\\ &\qquad\qquad (m\ (\mathit{MapSequ}\ (\mathit{MapFork}\ m)\ v))\end{aligned}$$

The types $\mathit{MapFork}, \mathit{MapSequ} :: (\star \to \star) \to (\star \to \star)$ represent the so-called generalized tries [16] for *Fork* and *Sequ*. The type constructor *MapFork* is the type-level counterpart of the function $\mathit{twice}\ f\ x = f\ (f\ x)$, which applies a given function twice to a given value.

Here is another example for a nested datatype of second-order kind.

$$\begin{aligned}\textbf{type}\ \mathit{Square}\ a &= \mathit{Square}'\ \mathit{Nil}\ a\\ \textbf{data}\ \mathit{Square}'\ f\ a &= \mathit{Zero}\ (f\ (f\ a))\\ &\mid \mathit{Succ}\ (\mathit{Square}'\ (\mathit{Cons}\ f)\ a)\\ \textbf{data}\ \mathit{Nil}\ a &= \mathit{Nil}\\ \textbf{data}\ \mathit{Cons}\ f\ a &= \mathit{Cons}\ a\ (f\ a)\end{aligned}$$

The type constructors have kinds $\mathit{Square}, \mathit{Nil} :: \star \to \star$ and $\mathit{Square}', \mathit{Cons} :: (\star \to \star) \to (\star \to \star)$. The type *Square* implements square $n \times n$ matrices [35, 18]. In contrast to common representations, such as lists of lists, the 'squareness' constraint is automatically enforced by the type system. More examples of nested types of higher-order kinds can be found in [15, 35, 18].

Now, before going on the reader is invited to consider programming, say, comparison functions for the datatypes above. This is a non-trivial task especially for the nested and for the second-order kinded types. In the sequel we will show that defining a *generic* comparison function that works for *all* datatypes is—perhaps surprisingly—a much simpler task.

3 Generic programming in a nutshell

This section sketches the main ideas of our approach to generic functional programming primarily from the programmer's perspective. A formal treatment of the approach is deferred to Sec. 5 and Sec. 6.

Before explaining how to define type-indexed values let us first take a closer look at datatype definitions. Haskell's **data** construct combines several features in a single coherent form: type abstraction, n-ary disjoint sums, n-ary cartesian products, and type recursion. The following rewritings of *List* and *GRose* make the structure of the datatype definitions more explicit.

$$\begin{aligned} \mathit{List}\ a &= 1 + a \times \mathit{List}\ a \\ \mathit{GRose}\ f\ a &= a \times f\ (\mathit{GRose}\ f\ a) \end{aligned}$$

Here, 1 denotes the unit type, and '+' and '×' are more conventional notation for binary disjoint sums and binary cartesian products. For simplicity, we assume that n-ary sums are reduced to binary sums and n-ary products to binary products. We treat 1, '+', and '×' as if they were given by the following datatype declarations.

$$\begin{aligned} \mathbf{data}\ 1 &= () \\ \mathbf{data}\ a_1 + a_2 &= \mathit{Inl}\ a_1 \mid \mathit{Inr}\ a_2 \\ \mathbf{data}\ a_1 \times a_2 &= (a_1, a_2) \end{aligned}$$

Now, to define a type-indexed value it suffices to specify cases for the primitive types (say, 1 and *Int*) and for the primitive type constructors (say, '+' and '×'). As an example, the following equations define the generic function $enc\langle a\rangle$, which encodes elements of type a as bit strings implementing a simple form of data compression [23]. The type argument of *enc* is written in angle brackets to distinguish it from the value argument.

$$\begin{aligned} \mathbf{type}\ \mathit{Bin} &= [\mathit{Bit}] \\ \mathbf{data}\ \mathit{Bit} &= 0 \mid 1 \\ enc\langle a :: \star\rangle &:: a \to \mathit{Bin} \\ enc\langle 1\rangle\ () &= [\,] \\ enc\langle \mathit{Int}\rangle\ n &= encInt\ n \\ enc\langle a_1 + a_2\rangle\ (\mathit{Inl}\ x_1) &= 0 : enc\langle a_1\rangle\ x_1 \\ enc\langle a_1 + a_2\rangle\ (\mathit{Inr}\ x_2) &= 1 : enc\langle a_2\rangle\ x_2 \\ enc\langle a_1 \times a_2\rangle\ (x_1, x_2) &= enc\langle a_1\rangle\ x_1 + \!\!+\ enc\langle a_2\rangle\ x_2 \end{aligned}$$

The type signature of *enc* makes explicit that the type of $enc\langle a\rangle$ depends on the type parameter a. Each equation is more or less inevitable. To encode the single element of the unit type no bits are required. Integers are encoded using the primitive function *encInt*, whose existence we assume. To encode an element of a disjoint union we emit one bit for the constructor followed by the encoding of its argument. Finally, the encoding of a pair is given by the concatenation of the component's encodings.[1]

This simple definition contains all ingredients needed to derive specializations for compressing elements of arbitrary datatypes. For instance, $enc\langle \mathit{Sequ\ Int}\rangle$ of type $\mathit{Sequ\ Int} \to \mathit{Bin}$ compresses random-access lists with integer elements and $enc\langle \mathit{GRose\ List\ Int}\rangle$ compresses generalized rose trees with integer labels.

Remark 1 *Generic values cannot directly be implemented in Haskell or Standard ML. The reason is simply that* $enc\langle t\rangle$ *is a* value *which depends on the* type t. *Value-type dependencies cannot be expressed in current functional languages. Even if we circumvented this problem by using encodings into a universal datatype* [45] *or by using dynamic types and a* **typecase** [1], *the result would be rather inefficient because enc would repeatedly interpret its type argument. By specializing* $enc\langle t\rangle$ *for a given t we remove this interpretative layer. As an aside, one could argue that value-type dependencies are present in the second-order λ-calculus* [11] *since a polymorphic function depends on the type argument supplied. The dependence is, however, quite loose since a polymorphic function uses 'the same algorithm' at each type. Note that a function of type* $\forall a.a \to \mathit{Bit}$ *is necessarily a constant function—this is a simple consequence of the parametricity theorem* [44].

In order to specialize $enc\langle t\rangle$ we cannot simply unfold or partially evaluate the definition of *enc*. To see why consider specializing $enc\langle \mathit{Sequ\ Int}\rangle$: to define $enc\langle \mathit{Sequ\ Int}\rangle$ we require $enc\langle \mathit{Sequ}\ (\mathit{Fork}^n\ \mathit{Int})\rangle$ for each $n \geqslant 1$. If we simply unfold the definition, we will in general not obtain a finite representation of $enc\langle t\rangle$.

The key idea of the specialization is to mimic the structure of types at the value level. For instance, $enc\langle \mathit{Sequ\ Int}\rangle$ should be compositionally defined in terms of specializations for the constituent types, say, *encSequ* and *encInt*. Since *Sequ* is a function on types, *encSequ* is consequently a function on encoders. Then the encoder for the type application *Sequ Int* is given by the application of *encSequ* to *encInt*. In a nutshell, type abstraction is mapped to value abstraction, type application to value application, and type recursion to value recursion. To exemplify, for *GRose*, *Fork*, and *Sequ* we can automatically derive the following specializations (for clarity, we use the original constructor names).

$$\begin{aligned} &encGRose :: \forall f.(\forall b.(b \to \mathit{Bin}) \to (f\ b \to \mathit{Bin})) \\ &\qquad\qquad \to (\forall a.(a \to \mathit{Bin}) \to (\mathit{GRose}\ f\ a \to \mathit{Bin})) \\ &encGRose\ encf\ enca\ (\mathit{GBranch}\ x\ ts) \\ &\qquad = enca\ x + \!\!+\ encf\ (encGRose\ encf\ enca)\ ts \\ &encFork :: \forall a.(a \to \mathit{Bin}) \to (\mathit{Fork}\ a \to \mathit{Bin}) \\ &encFork\ enca\ (\mathit{Node}\ x_1\ x_2) \\ &\qquad = enca\ x_1 + \!\!+\ enca\ x_2 \\ &encSequ :: \forall a.(a \to \mathit{Bin}) \to (\mathit{Sequ}\ a \to \mathit{Bin}) \\ &encSequ\ enca\ \mathit{Empty} \\ &\qquad = [0] \\ &encSequ\ enca\ (\mathit{Zero}\ s) \\ &\qquad = 1 : 0 : encSequ\ (encFork\ enca)\ s \\ &encSequ\ enca\ (\mathit{One}\ x\ s) \\ &\qquad = 1 : 1 : enca\ x + \!\!+\ encSequ\ (encFork\ enca)\ s \end{aligned}$$

The types of the functions are motivated and explained only in Sec. 5.3.

Many list processing functions can be generalized to arbitrary datatypes. Consider, for instance, the polymorphic function $length :: \forall x.\mathit{List}\ x \to \mathit{Int}$, which computes the length of a list.[2] A *length* or rather a *size* function can also be defined for rose trees, for random-access lists, and for many other datatypes. The recipe for defining these functions is simple: in each case we count the number of elements of type x in a given value of type $f\ x$. This suggests that we should be able to program a generic function $size\langle f\rangle :: \forall x.f\ x \to \mathit{Int}$, which works for all f. Note that the type signature of *size* is more involved than the signature of *enc* since *size* is indexed by a type constructor of kind $\star \to \star$ rather than by a type of kind $\star$. The type of *size* ensures that we can determine the size of a list or a rose tree

[1] The definition of *enc* exhibits $\Theta(n^2)$ worst-case behaviour, but this is easy to remedy.

[2] In Haskell there is no syntax for universal quantification. For clarity, however, we will always write quantification explicitly when giving the type of functions.

but not the size of a Boolean or an integer. Now, in order to define $size\langle f\rangle$ generically for all f we must explicate the structure of type functions of kind $\star \to \star$. To this end we lift the primitive type constructors 1, *Int*, '+', and '×' to a function level.

$$\begin{array}{lcl} \underline{1}\ a & = & 1 \\ \underline{Int}\ a & = & Int \\ (f_1 \underline{+} f_2)\ a & = & f_1\ a + f_2\ a \\ (f_1 \underline{\times} f_2)\ a & = & f_1\ a \times f_2\ a \end{array}$$

Note that '+' has kind $\star \to (\star \to \star)$ whereas the lifted version has kind $(\star \to \star) \to ((\star \to \star) \to (\star \to \star))$. Interestingly, every type constructor of kind $\star \to \star$ can be expressed in terms of $\underline{1}$, $\underline{Int}$, '$\underline{+}$', '$\underline{\times}$', and the identity type given by $Id\ a = a$—this claim will be justified in Sec. 6.1. Thus, $size\langle f\rangle$ is uniquely defined by the following equations.

$$\begin{array}{lcl} size\langle f :: \star \to \star\rangle & :: & \forall x.f\ x \to Int \\ size\langle Id\rangle\ x & = & 1 \\ size\langle \underline{1}\rangle\ () & = & 0 \\ size\langle \underline{Int}\rangle\ n & = & 0 \\ size\langle f_1 \underline{+} f_2\rangle\ (Inl\ x_1) & = & size\langle f_1\rangle\ x_1 \\ size\langle f_1 \underline{+} f_2\rangle\ (Inr\ x_2) & = & size\langle f_2\rangle\ x_2 \\ size\langle f_1 \underline{\times} f_2\rangle\ (x_1, x_2) & = & size\langle f_1\rangle\ x_1 + size\langle f_2\rangle\ x_2 \end{array}$$

Again, each of the equations is more or less inevitable. A value of type $Id\ x = x$ contains exactly only one element of type x; values of type $\underline{1}\ x = 1$ or $\underline{Int}\ x = Int$ contain no elements of type x. To determine the size of a structure of type $(f_1 \underline{+} f_2)\ x$ we must either calculate the size of a structure of type $f_1\ x$ or that of a structure of type $f_2\ x$ depending on which component of the disjoint sum the argument comes from. Finally, the size of a structure of type $(f_1 \underline{\times} f_2)\ x$ is given by the sum of the size of the two components.

Specializing *size* to concrete instances of datatypes works essentially as before. For instance, for *List* and *GRose* we obtain the following specializations.

$$\begin{array}{lcl} sizeList & :: & \forall a.(\forall x.a\ x \to Int) \to (\forall x.List\ (a\ x) \to Int) \\ \multicolumn{3}{l}{sizeList\ sizea\ Nil} \\ & = & 0 \\ \multicolumn{3}{l}{sizeList\ sizea\ (Cons\ x\ xs)} \\ & = & sizea\ x + sizeList\ sizea\ xs \\ sizeGRose & :: & \forall f.(\forall b.(\forall x.b\ x \to Int) \\ & & \qquad \to (\forall x.(f\ x)\ (b\ x) \to Int)) \\ & & \quad \to (\forall a.(\forall x.a\ x \to Int) \\ & & \qquad \to (\forall x.GRose\ (f\ x)\ (a\ x) \to Int)) \\ \multicolumn{3}{l}{sizeGRose\ sizef\ sizea\ (GBranch\ x\ ts)} \\ & = & sizea\ x + sizef\ (sizeGRose\ sizef\ sizea)\ ts \end{array}$$

Again, the definitions rigidly follow the structure of types: since *List* is a function on types, *sizeList* transforms 'size functions' into 'size functions'. More precisely, if *sizea* determines the size of a structure of type $a\ x$, then *sizeList sizea* determines the size of a structure of type $List\ (a\ x)$—the typings will be explained more thoroughly in Sec. 6.3. Now, to obtain the length function for lists we simply pass the size function for *Id*, i.e., $sizeId\ x = 1$, to *sizeList*. Likewise, to determine the size of a structure of type, say, *GRose List x* we call *sizeGRose sizeList sizeId*.

4 Kinds and types

This section introduces kind and type terms. Kind terms are formed according to the following grammar.

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

We agree upon that '$\to$' associates to the right.

Given a fixed set of type variables X and a fixed set of primitive type constructors $C = \{1, Int, (+), (\times)\}$ type terms are formed according to the following grammar.

$$T \ ::= \ X \mid C \mid (T\ T) \mid (\Lambda X :: K \to T) \mid (\mu T)$$

Here, $t_1\ t_2$ denotes type application, $\Lambda x :: \kappa \to t$ denotes type abstraction, and μt denotes the fixpoint of t. We agree upon that type application associates to the left and that type abstraction extends as far to the right as possible. We abbreviate $\Lambda x_1 :: \kappa_1 \to \cdots \Lambda x_m :: \kappa_m \to t$ by $\Lambda x_1 :: \kappa_1 \ \ldots \ x_m :: \kappa_m \to t$ and write $(+)\ t_1\ t_2$ as $t_1 + t_2$ and similarly for '×'. Note that in a Λ-abstraction the bound type variable is annotated with its kind. For reasons of readability we will usually omit the kind annotation. Furthermore, note that the choice of C is more or less arbitrary; we only require the primitive types to have first-order kinds or kind $\star$, see Sec. 6.5. For instance, C could additionally include the function space constructor '$\to$'. We have only omitted '$\to$' because most of the generic functions cannot sensibly be defined for the function space.

The kinds of the primitives are given by

$$\begin{array}{lcl} 1, Int & :: & \star \\ (+), (\times) & :: & \star \to \star \to \star \ . \end{array}$$

The kinds of type terms are determined by the rules depicted in Fig. 1. The notation $\Gamma \vdash t :: \kappa$ means that the statement $t :: \kappa$ is derivable from the set of kind assumptions Γ. If $t :: \kappa_1 \to \cdots \to \kappa_n \to \star$ is derivable, we say that t has arity n. It is worth mentioning that 'μ' is polymorphic with respect to the kinds, i.e., it represents a family of fixpoint operators.

In essence, Haskell types are represented by terms of the simply typed λ-calculus with kinds playing the rôle of types. The translation of the datatype declarations given in Sec. 2 into type terms is fairly straightforward. Here are some examples.

$$\begin{array}{lcl} List & = & \Lambda a \to \mu(\Lambda \ell \to 1 + a \times \ell) \\ Fork & = & \Lambda a \to a \times a \\ Sequ & = & \mu(\Lambda s\ a \to 1 + s\ (Fork\ a) + a \times s\ (Fork\ a)) \end{array}$$

First-order kinded, regular types, such as *List* and *Rose*, can be modelled using a fixpoint operator of kind $\star \to \star$ while nested types, such as *Sequ*, require an operator of kind $(\star \to \star) \to (\star \to \star)$.

Remark 2 *Type recursion is expressed using a fixpoint operator. Whilst this is sufficient for the datatype definitions introduced in Sec. 2, it does not generalize easily to handle mutually recursive types. A viable alternative is to consider systems of recursion equations:*

$$x_1 = t_1; \ldots ; x_n = t_n \ ,$$

where the x_i are type variables and the t_i are type terms. The approach to generic programming works equally well if we use recursion equations instead of a fixpoint operator. The development, however, becomes more verbose and provides no additional insights.

$$\frac{(x :: \kappa) \in \Gamma}{\Gamma \vdash x :: \kappa} \qquad \frac{\Gamma \vdash t_1 :: \kappa_2 \to \kappa_1 \qquad \Gamma \vdash t_2 :: \kappa_2}{\Gamma \vdash (t_1\ t_2) :: \kappa_1} \qquad \frac{\Gamma, x :: \kappa_1 \vdash t :: \kappa_2}{\Gamma \vdash (\Lambda x :: \kappa_1 \to t) :: (\kappa_1 \to \kappa_2)} \qquad \frac{\Gamma \vdash t :: \kappa \to \kappa}{\Gamma \vdash (\mu t) :: \kappa}$$

Figure 1: Kind rules.

In the following sections we require the notion of Böhm tree which we introduce next. Böhm trees can be considered as a kind of 'infinite normal form' for type terms and are obtained by unwinding type terms ad infinitum.

Definition 1 *The* convertibility relation *on type terms, denoted '$\leftrightarrow$', is given by the following axioms*

$$\begin{aligned} (\Lambda x \to t)\ u &\leftrightarrow t\,[x := u] && (\beta)\\ \Lambda x \to t\ x &\leftrightarrow t\,, \quad x \textit{ not free in } t && (\eta)\\ \mu t &\leftrightarrow t\ (\mu t) && (\mu) \end{aligned}$$

plus the usual 'logical' rules for reflexivity, symmetry, transitivity, and congruence.

Definition 2 *A* head normal form *(hnf) is a type term of the form $\Lambda x_1\ \dots\ x_m \to z\ u_1\ \dots\ u_n$ with $m, n \geqslant 0$, $z \in C \cup X$, and z has arity n. A type term t has hnf u if $t \leftrightarrow u$ and u is an hnf.*

The definition of hnf is a bit unusual in that we additionally require hnfs to be η-expanded (in order to guarantee that Böhm trees are well-defined). The term $(+)$, for instance, has hnf $\Lambda x_1\ x_2 \to x_1 + x_2$. Not every type term has an hnf, consider, for instance $\mu(\Lambda a \to a)$. Type terms that have no hnf are in a sense pointless and can be excluded by a simple syntactic restriction, which we will adopt in the sequel: t in μt must have the form $\Lambda x_1\ \dots\ x_m \to c\ u_1\ \dots\ u_n$ with $c \in C$ of arity n.

Definition 3 *The* Böhm tree *of the type term t, denoted* BT(t), *is a labelled tree defined as follows: if t has hnf $\Lambda x_1\ \dots\ x_m \to z\ u_1\ \dots\ u_n$, then*

$$\begin{tikzcd}[column sep=small] \mathrm{BT}(t) = \Lambda x_1 \dots x_m \to & & z \arrow[dl, no head] \arrow[dr, no head] & \\ & \mathrm{BT}(u_1) & \cdots & \mathrm{BT}(u_n)\ . \end{tikzcd}$$

For a more formal treatment of Böhm trees we refer the reader to [2]. Now, from a generic programming point of view we can identify type terms that have the same Böhm trees, i.e., $t = u$ if $\mathrm{BT}(t) = \mathrm{BT}(u)$. We have, for instance, $\mathit{Rose} = \mathit{GRose}\ (\Lambda a \to \mathit{Fix}\ (\mathit{BaseList}\ a))$. Since generic values are defined by induction on the *structure* of types, the structure is all that matters.

5 Type-indexed values

The framework is developed in three steps: (1) we characterize the set of normal forms of types of kind $\star$, (2) we give a prototype for generic values indexed by types of this kind, and (3) we show how to promote a generic value thus defined to types of arbitrary kind.

5.1 Normal forms of types of kind $\star$

Types of kind $\star$ have a very simple normal form. Consider the Böhm tree of a type of kind $\star$. Clearly, the root of the tree cannot be labelled with a type abstraction. Instead, it must be labelled with a primitive type constructor, say, c. Moreover, if c has arity n, the root must have n direct successors. Thus, the normal form of type terms of kind $\star$ is described by the following grammar.

$$T_\star \ ::= \ 1 \mid \mathit{Int} \mid (T_\star + T_\star) \mid (T_\star \times T_\star)$$

It is understood that $T_\star$ contains *type terms* of kind $\star$. The set of all *type trees* (finite and infinite) that can be formed according to this grammar is denoted $T_\star^\infty$. We have $T_\star^\infty = \{\mathrm{BT}(t) \mid \emptyset \vdash t :: \star\}$.

5.2 Defining $\star$-indexed values

The characterization of normal forms motivates the following prototype for type-indexed values.

$$\begin{aligned} \mathit{poly}\langle a :: \star\rangle &:: F\ a\\ \mathit{poly}\langle 1\rangle &= \mathit{poly}_1\\ \mathit{poly}\langle \mathit{Int}\rangle &= \mathit{poly}_{\mathit{Int}}\\ \mathit{poly}\langle a_1 + a_2\rangle &= \mathit{poly}_+\ (\mathit{poly}\langle a_1\rangle)\ (\mathit{poly}\langle a_2\rangle)\\ \mathit{poly}\langle a_1 \times a_2\rangle &= \mathit{poly}_\times\ (\mathit{poly}\langle a_1\rangle)\ (\mathit{poly}\langle a_2\rangle) \end{aligned}$$

Here, poly is the name of the type-indexed value; a, a_1, and a_2 are type variables of kind $\star$; F, poly_1, $\mathit{poly}_{\mathit{Int}}$, poly_+, and $\mathit{poly}_\times$ are the ingredients that have to be supplied by the generic programmer. The type of $\mathit{poly}\langle a\rangle$ is given by $F\ a$, where F is a type constructor of kind $\star \to \star$. Unlike the type index F may also contain function types, $t_1 \to t_2$, and universally quantified types, $\forall x.t$. The poly_c values must have the following types:

$$\begin{aligned} \mathit{poly}_1 &:: F\ 1\\ \mathit{poly}_{\mathit{Int}} &:: F\ \mathit{Int}\\ \mathit{poly}_+ &:: \forall x_1, x_2. F\ x_1 \to F\ x_2 \to F\ (x_1 + x_2)\\ \mathit{poly}_\times &:: \forall x_1, x_2. F\ x_1 \to F\ x_2 \to F\ (x_1 \times x_2)\ . \end{aligned}$$

In the latter two cases x_1 and x_2 are universally qualified since poly_+ and $\mathit{poly}_\times$ have to work for all possible argument types.

It is instructive to see how the example of Sec. 3 maps to the formalism above: $\mathit{enc}\langle a\rangle$ has type $F\ a = a \to \mathit{Bin}$ and the functions enc_1, $\mathit{enc}_{\mathit{Int}}$, enc_+, and $\mathit{enc}_\times$ are given by

$$\begin{aligned} \mathit{enc}_1\ () &= [\,]\\ \mathit{enc}_{\mathit{Int}}\ n &= \mathit{encInt}\ n\\ \mathit{enc}_+\ \mathit{enca}_1\ \mathit{enca}_2\ (\mathit{Inl}\ x_1) &= 0 : \mathit{enca}_1\ x_1\\ \mathit{enc}_+\ \mathit{enca}_1\ \mathit{enca}_2\ (\mathit{Inr}\ x_2) &= 1 : \mathit{enca}_2\ x_2\\ \mathit{enc}_\times\ \mathit{enca}_1\ \mathit{enca}_2\ (x_1, x_2) &= \mathit{enca}_1\ x_1 + \!\!\! + \ \mathit{enca}_2\ x_2\ . \end{aligned}$$

The definition of type-indexed values is inductive on the structure of $T_\star$: we have one equation for each primitive type constructor. Now, a standard result from the theory of infinite trees [9] guarantees that a generic value like *poly* possesses a unique (least) extension in $T_\star^\infty$. In that sense, *poly* is uniquely defined by its action on primitive type constructors, i.e., by $poly_1$, $poly_{Int}$, $poly_+$, and $poly_\times$.

Before we proceed let us take a look at further examples of type-indexed values. The first example, *dec*, is essentially the inverse of *enc*: it takes a bit string, decodes a prefix of that string, and returns the decoded value coupled with the unused suffix. Formally, the two functions are related by $dec\langle a\rangle\ (enc\langle a\rangle\ x +\!\!+\ bs) = (x, bs)$.[3]

$$\begin{array}{lcl}
dec\langle a :: \star\rangle & :: & Bin \to (a, Bin)\\
dec\langle 1\rangle\ bs & = & ((), bs)\\
dec\langle Int\rangle\ bs & = & decInt\ bs\\
dec\langle a_1 + a_2\rangle\ [\,] & = & error\ \texttt{"dec"}\\
dec\langle a_1 + a_2\rangle\ (0 : bs) & = & \mathbf{let}\ (x_1, bs') = dec\langle a_1\rangle\ bs\\
 & & \mathbf{in}\ (Inl\ x_1, bs')\\
dec\langle a_1 + a_2\rangle\ (1 : bs) & = & \mathbf{let}\ (x_2, bs') = dec\langle a_2\rangle\ bs\\
 & & \mathbf{in}\ (Inr\ x_2, bs')\\
dec\langle a_1 \times a_2\rangle\ bs & = & \mathbf{let}\ (x_1, bs_1) = dec\langle a_1\rangle\ bs\\
 & & \phantom{\mathbf{let}}\ (x_2, bs_2) = dec\langle a_2\rangle\ bs_1\\
 & & \mathbf{in}\ ((x_1, x_2), bs_2)
\end{array}$$

The functions *enc* and *dec* can be seen as very simple printers and parsers. *Pretty* printers and parsers, which produce and process a human readable format, can be implemented if we give the generic programmer additionally access to the constructor names. In [17] we use such an extension to define Haskell's *show* function in a generic way.

Comparison functions are typical examples of type-indexed values. The following program realizes Haskell's *compare* function, which determines the precise ordering of two elements.

$$\begin{array}{lcl}
\mathbf{data}\ Ordering & = & LT \mid EQ \mid GT\\
cmp\langle a :: \star\rangle & :: & a \to a \to Ordering\\
cmp\langle 1\rangle\ ()\ () & = & EQ\\
cmp\langle Int\rangle\ m\ n & = & cmpInt\ m\ n\\
cmp\langle a_1 + a_2\rangle\ (Inl\ x_1)\ (Inl\ y_1) & = & cmp\langle a_1\rangle\ x_1\ y_1\\
cmp\langle a_1 + a_2\rangle\ (Inl\ x_1)\ (Inr\ y_2) & = & LT\\
cmp\langle a_1 + a_2\rangle\ (Inr\ x_2)\ (Inl\ y_1) & = & GT\\
cmp\langle a_1 + a_2\rangle\ (Inr\ x_2)\ (Inr\ y_2) & = & cmp\langle a_2\rangle\ x_2\ y_2\\
cmp\langle a_1 \times a_2\rangle\ (x_1, x_2)\ (y_1, y_2) & & \\
\quad = cmp\langle a_1\rangle\ x_1\ y_1\ \text{`}lexord\text{`}\ cmp\langle a_2\rangle\ x_2\ y_2 & &
\end{array}$$

The helper function *lexord* used in the last equation implements the lexicographic product of two orderings.

$$\begin{array}{lcl}
lexord & :: & Ordering \to Ordering \to Ordering\\
lexord\ LT\ ord & = & LT\\
lexord\ EQ\ ord & = & ord\\
lexord\ GT\ ord & = & GT
\end{array}$$

5.3 Specializing $\star$-indexed values

The purpose of a generic value is to be specialized. We have already discussed the key idea of promoting a generic value to types of arbitrary kinds: type abstraction is interpreted by value abstraction, type application by value application, and type recursion by value recursion. The promoted version of $poly\langle\cdot\rangle$, which we denote $poly\langle\!\langle\cdot\rangle\!\rangle$, is consequently given by the following equations.

$$\begin{array}{lcl}
poly\langle\!\langle t :: \kappa\rangle\!\rangle\ \varrho & :: & F\langle\!\langle t :: \kappa\rangle\!\rangle\\
poly\langle\!\langle c\rangle\!\rangle\ \varrho & = & poly_c\\
poly\langle\!\langle x\rangle\!\rangle\ \varrho & = & \varrho\ x\\
poly\langle\!\langle t_1\ t_2\rangle\!\rangle\ \varrho & = & (poly\langle\!\langle t_1\rangle\!\rangle\ \varrho)\ (poly\langle\!\langle t_2\rangle\!\rangle\ \varrho)\\
poly\langle\!\langle \Lambda x \to t\rangle\!\rangle\ \varrho & = & \lambda v \to poly\langle\!\langle t\rangle\!\rangle\ (\varrho(x := v))\\
poly\langle\!\langle \mu t\rangle\!\rangle\ \varrho & = & fix\ (poly\langle\!\langle t\rangle\!\rangle\ \varrho)
\end{array}$$

Here, ϱ ranges over environments, which map type variables to value variables, and $\varrho(x := v)$ is syntax for extending the environment ϱ by the binding $x := v$. The function *fix* is the polymorphic fixpoint operator on the value level. The definition of $poly\langle\!\langle t\rangle\!\rangle$ is inductive on the structure of type terms. We can, in fact, view the definition as an interpretation of the simply typed λ-calculus. The generic value $poly\langle\cdot\rangle$ is related to $poly\langle\!\langle\cdot\rangle\!\rangle$ by

$$poly\langle t :: \star\rangle \;=\; poly\langle\!\langle t :: \star\rangle\!\rangle\ \epsilon\ , \qquad (1)$$

where ϵ denotes the empty environment. Thus, in order to specialize $poly\langle t\rangle$ we simply specialize $poly\langle\!\langle t\rangle\!\rangle$. Note that Eq. (1) can be seen as the soundness condition of the specialization.

It remains to specify $F\langle\!\langle t :: \kappa\rangle\!\rangle$, which is defined by induction on the structure of kinds.

$$\begin{array}{lcl}
F\langle\!\langle u :: \star\rangle\!\rangle & = & F\ u\\
F\langle\!\langle u :: \kappa_1 \to \kappa_2\rangle\!\rangle & = & \forall x.F\langle\!\langle x :: \kappa_1\rangle\!\rangle \to F\langle\!\langle u\ x :: \kappa_2\rangle\!\rangle
\end{array}$$

If u is a type constructor of kind $\kappa_1 \to \kappa_2$, then $poly\langle\!\langle u\rangle\!\rangle$ is a function that maps values of type $F\langle\!\langle x :: \kappa_1\rangle\!\rangle$ to values of type $F\langle\!\langle u\ x :: \kappa_2\rangle\!\rangle$, for all x. Again, it is important that x is universally quantified since u may be applied to different types. The nesting of universal quantifiers is dictated by the kind: if κ has order n, then $F\langle\!\langle u :: \kappa\rangle\!\rangle$ is a rank-n type [30]—assuming that $F\ a$ has rank 0. For instance, for F given by $F\ a = a \to Bin$ we have

$$\begin{array}{ll}
 & F\langle\!\langle GRose :: (\star \to \star) \to (\star \to \star)\rangle\!\rangle\\
= & \forall f.F\langle\!\langle f :: \star \to \star\rangle\!\rangle \to F\langle\!\langle GRose\ f :: \star \to \star\rangle\!\rangle\\
= & \forall f.(\forall b.F\langle\!\langle b :: \star\rangle\!\rangle \to F\langle\!\langle f\ b :: \star\rangle\!\rangle)\\
 & \qquad \to (\forall a.F\langle\!\langle a :: \star\rangle\!\rangle \to F\langle\!\langle GRose\ f\ a :: \star\rangle\!\rangle)\\
= & \forall f.(\forall b.F\ b \to F\ (f\ b)) \to (\forall a.F\ a \to F\ (GRose\ f\ a))\\
= & \forall f.(\forall b.(b \to Bin) \to (f\ b \to Bin))\\
 & \qquad \to (\forall a.(a \to Bin) \to (GRose\ f\ a \to Bin))\ .
\end{array}$$

Since *GRose* has an order-2 kind, $F\langle\!\langle GRose\rangle\!\rangle$ is a rank-2 type.

Let us consider some examples. Fig. 2 lists the specializations of *enc* to the datatypes introduced in Sec. 2. For clarity, the definitions use the original constructor names and functions are written in an equational style. The specializations illustrate several interesting points. For instance, the function *encSequ* makes use of polymorphic recursion [33]: the recursive call has type $\forall a.(Fork\ a \to Bin) \to (Sequ\ (Fork\ a) \to Bin)$, which is a substitution instance of the declared type. In general, polymorphic recursion is required whenever the type recursion is nested. Several functions have rank-2 type signatures; *encMapFork* shows in a nutshell why this is necessary: the argument *encm* is applied at two different instances: the inner call has type $\forall a.(a \to Bin) \to (m\ a \to Bin)$ while the outer call has type $\forall a.(m\ a \to Bin) \to (m\ (m\ a) \to Bin)$. The functions *encMapSequ* and *encSquare'* even combine polymorphic recursion and the specialized use of a polymorphic argument.

[3]We tacitly assume that the predefined functions *encInt* and *decInt* satisfy this relationship.

```
encList                              :: ∀a.(a → Bin) → (List a → Bin)
encList enca                         =  encl
    where encl Nil                   =  [0]
          encl (Cons x xs)           =  1 : enca x ++ encl xs
encRose                              :: ∀a.(a → Bin) → (Rose a → Bin)
encRose enca                         =  encr
    where encr (Branch x ts)         =  enca x ++ encList encr ts
encGRose                             :: ∀f.(∀b.(b → Bin) → (f b → Bin))
                                          → (∀a.(a → Bin) → (GRose f a → Bin))
encGRose encf enca                   =  encg
    where encg (GBranch x ts)        =  enca x ++ encf encg ts
encFix                               :: ∀f.(∀a.(a → Bin) → (f a → Bin)) → (Fix f → Bin)
encFix encf                          =  encr
    where encr (In x)                =  encf encr x
encBaseList                          :: ∀a.(a → Bin) → (∀b.(b → Bin) → (BaseList a b → Bin))
encBaseList enca encb Nil            =  [0]
encBaseList enca encb (Cons x y)     =  1 : enca x ++ encb y
encFork                              :: ∀a.(a → Bin) → (Fork a → Bin)
encFork enca (Node x1 x2)            =  enca x1 ++ enca x2
encSequ                              :: ∀a.(a → Bin) → (Sequ a → Bin)
encSequ enca Empty                   =  [0]
encSequ enca (Zero s)                =  1 : 0 : encSequ (encFork enca) s
encSequ enca (One x s)               =  1 : 1 : enca x ++ encSequ (encFork enca) s
encMapFork                           :: ∀m.(∀w.(w → Bin) → (m w → Bin))
                                          → (∀v.(v → Bin) → (MapFork m v → Bin))
encMapFork encm encv (TrieFork tf)   =  encm (encm encv) tf
encMapSequ                           :: ∀m.(∀w.(w → Bin) → (m w → Bin))
                                          → (∀v.(v → Bin) → (MapSequ m v → Bin))
encMapSequ encm encv (TrieSequ te tz to) = encv te
                                     ++ encMapSequ (encMapFork encm) encv tz
                                     ++ encm (encMapSequ (encMapFork encm) encv) to
encSquare                            :: ∀a.(a → Bin) → (Square a → Bin)
encSquare enca m                     =  encSquare' encNil enca m
encSquare'                           :: ∀f.(∀b.(b → Bin) → (f b → Bin))
                                          → (∀a.(a → Bin) → (Square' f a → Bin))
encSquare' encf enca (Zero m)        =  0 : encf (encf enca) m
encSquare' encf enca (Succ m)        =  1 : encSquare' (encCons encf) enca m
encNil                               :: ∀a.(a → Bin) → (Nil a → Bin)
encNil enca Nil                      =  []
encCons                              :: ∀f.(∀b.(b → Bin) → (f b → Bin))
                                          → (∀a.(a → Bin) → (Cons f a → Bin))
encCons encf enca (Cons x xs)        =  enca x ++ encf enca xs
```

Figure 2: Specializing *enc* to the types of Sec. 2.

6 Values indexed by first-order kinded types

In the previous section we have considered values indexed by types of kind $\star$. For values that are indexed by type constructors, such as *size*, some additional machinery is needed. We will develop the main ideas for type indices of kind $\star \to \star$. The straightforward extension to first-order kinds is explained in Sec. 6.4. Sec. 6.5 discusses the difficulties in extending the approach to higher-order kinds. We proceed as in the previous section: (1) we characterize the set of normal forms of types of kind $\star \to \star$, (2) we give a prototype for generic values indexed by types of this kind, and (3) we show how to promote a generic value to types of arbitrary kind.

6.1 Normal forms of types of kind $\star \to \star$

Recall that a type constructor of kind $\star \to \star$ is a function on types of kind $\star$. Consequently, when defining $(\star \to \star)$-indexed values we have to use type patterns that range over functions. It appears that these functions can be most conveniently expressed if we lift types to a function level. Formally, lifting maps a type $t :: \kappa$ to a type $\uparrow t :: \uparrow\kappa$ where $\uparrow\kappa$ is defined as follows.

$$\begin{aligned}\uparrow\star &= \star \to \star\\ \uparrow\kappa_1 \to \kappa_2 &= (\uparrow\kappa_1) \to (\uparrow\kappa_2)\end{aligned}$$

The lifted version $\uparrow t$ of type t is given by (we assume that for each type variable x of kind κ there is a lifted variable named $\underline{x}$ of kind $\uparrow\kappa$)

$$\begin{aligned}\uparrow c &= \underline{c}\\ \uparrow x &= \underline{x}\\ \uparrow t_1\ t_2 &= (\uparrow t_1)\ (\uparrow t_2)\\ \uparrow \Lambda x \to t &= \Lambda \underline{x} \to \uparrow t\\ \uparrow \mu t &= \mu(\uparrow t)\ .\end{aligned}$$

The lifted versions, $\underline{c}$, of the primitive type constructors have already been defined in Sec. 3. The lifted version of *List*, for instance, reads

$$\underline{List} = \Lambda \underline{a} \to \mu(\Lambda \underline{\ell} \to \underline{1} \mathbin{\underline{+}} \underline{a} \mathbin{\underline{\times}} \underline{\ell})\ .$$

Expanding the lifted primitives we obtain

$$\underline{List} = \Lambda \underline{a} \to \mu(\Lambda \underline{\ell}\ z \to 1 + \underline{a}\ z \times \underline{\ell}\ z)\ .$$

The lifted and the unlifted version of a type are closely related: if t has kind $\star \to \star$, then

$$t = (\uparrow t)\ Id\ . \qquad (2)$$

This relation will be employed later when we define $poly\langle\cdot\rangle$ in terms of $poly\langle\!\langle\cdot\rangle\!\rangle$.

Using the notion of lifting we can easily characterize the set of normal forms of types of kind $\star \to \star$. Assume that we are given a type t of kind $\star \to \star$. Applying η-expansion we have $t = \Lambda a \to t\ a$. The body of the abstraction has kind $\star$ and we know from the previous section the shape of its normal form. The free variable, a, is treated as an additional constant of kind $\star$. Now, to make the passing of a explicit we abstract a out. The abstraction process replaces the primitive type constructors by their lifted versions and a by $Id = \Lambda a \to a$. This motivates the following characterization.

$$T_{\star\to\star} ::= Id \mid \underline{1} \mid \underline{Int} \mid (T_{\star\to\star} \mathbin{\underline{+}} T_{\star\to\star}) \mid (T_{\star\to\star} \mathbin{\underline{\times}} T_{\star\to\star})$$

We can, in fact, view Id, $\underline{1}$, $\underline{Int}$, '$\underline{+}$', and '$\underline{\times}$' as a tiny combinator language for defining type constructors of kind $\star \to \star$.

6.2 Defining $(\star \to \star)$-indexed values

The characterization of normal forms suggests the following prototype for values indexed by types of kind $\star \to \star$.

$$\begin{aligned}poly\langle f :: \star \to \star\rangle &:: H\ f\\ poly\langle Id\rangle &= poly_{Id}\\ poly\langle \underline{1}\rangle &= poly_{\underline{1}}\\ poly\langle \underline{Int}\rangle &= poly_{\underline{Int}}\\ poly\langle f_1 \mathbin{\underline{+}} f_2\rangle &= poly_{\underline{+}}\ (poly\langle f_1\rangle)\ (poly\langle f_2\rangle)\\ poly\langle f_1 \mathbin{\underline{\times}} f_2\rangle &= poly_{\underline{\times}}\ (poly\langle f_1\rangle)\ (poly\langle f_2\rangle)\end{aligned}$$

The type of $poly\langle f\rangle$ is given by $H\ f$, where H is a type constructor of kind $(\star \to \star) \to \star$. The prototype for $\star \to \star$-indexed values is nearly identical to the prototype for $\star$-indexed values: the primitive type constructors are merely replaced by their lifted versions and we have one additional case for the identity type.

Let us consider how the example of Sec. 3 fits into this scheme: $size\langle f\rangle$ has type $H\ f = \forall x.f\ x \to Int$ and the functions $size_{Id}$, $size_{\underline{1}}$, $size_{\underline{Int}}$, $size_{\underline{+}}$, and $size_{\underline{\times}}$ are given by

$$\begin{aligned}size_{Id}\ x &= 1\\ size_{\underline{1}}\ () &= 0\\ size_{\underline{Int}}\ n &= 0\\ size_{\underline{+}}\ sizea_1\ sizea_2\ (Inl\ x_1) &= sizea_1\ x_1\\ size_{\underline{+}}\ sizea_1\ sizea_2\ (Inr\ x_2) &= sizea_2\ x_2\\ size_{\underline{\times}}\ sizea_1\ sizea_2\ (x_1, x_2) &= sizea_1\ x_1 + sizea_2\ x_2\ .\end{aligned}$$

Note that $size\langle f\rangle$ is not only a generic, but also a polymorphic function. This combination is, however, not cogent: the generic function $sum\langle f\rangle$, which sums a structure of integers, has the monomorphic type $f\ Int \to Int$.

In the rest of this section we present two further examples of generic values. The most paradigmatic example of a $(\star \to \star)$-indexed value is probably *map*, which applies a given function to each element of type x in a given structure of type $f\ x$.

$$\begin{aligned}map\langle f :: \star \to \star\rangle &:: \forall x, y.(x \to y) \to (f\ x \to f\ y)\\ map\langle Id\rangle\ \varphi\ x &= \varphi\ x\\ map\langle K\ t\rangle\ \varphi\ x &= x\\ map\langle f_1 \mathbin{\underline{+}} f_2\rangle\ \varphi\ (Inl\ x_1) &= Inl\ (map\langle f_1\rangle\ \varphi\ x_1)\\ map\langle f_1 \mathbin{\underline{+}} f_2\rangle\ \varphi\ (Inr\ x_2) &= Inr\ (map\langle f_2\rangle\ \varphi\ x_2)\\ map\langle f_1 \mathbin{\underline{\times}} f_2\rangle\ \varphi\ (x_1, x_2) &= (map\langle f_1\rangle\ \varphi\ x_1, map\langle f_2\rangle\ \varphi\ x_2)\end{aligned}$$

Note that the 'type pattern' $K\ t = \Lambda a \to t$ covers both $\underline{1}$ and $\underline{Int}$. Furthermore, note that Haskell provides a class *Functor* for mapping functions. Alas, the user must program instances of *Functor* by hand—which is for the most part tedious but sometimes quite involving.

The function *size* is an instance of a more general concept termed reduction or crush [31]. A reduction is a function of type $f\ x \to x$, which collapses a structure of values of type x into a single value of type x. To define a reduction we require two ingredients: a value $e :: x$ and a binary operation $op :: x \to x \to x$. Usually but not necessarily e is the neutral element of op.

$$\begin{aligned}&reduce\langle f :: \star \to \star\rangle :: \forall x.x \to (x \to x \to x) \to (f\ x \to x)\\ &reduce\langle Id\rangle\ e\ op\ x = x\\ &reduce\langle K\ t\rangle\ e\ op\ x = e\\ &reduce\langle f_1 \mathbin{\underline{+}} f_2\rangle\ e\ op\ (Inl\ x_1) = reduce\langle f_1\rangle\ e\ op\ x_1\\ &reduce\langle f_1 \mathbin{\underline{+}} f_2\rangle\ e\ op\ (Inr\ x_2) = reduce\langle f_2\rangle\ e\ op\ x_2\\ &reduce\langle f_1 \mathbin{\underline{\times}} f_2\rangle\ e\ op\ (x_1, x_2)\\ &\quad = reduce\langle f_1\rangle\ e\ op\ x_1\ \text{`}op\text{`}\ reduce\langle f_2\rangle\ e\ op\ x_2\end{aligned}$$

A number of useful functions can be defined in terms of $\mathit{reduce}\langle f\rangle$ and $\mathit{map}\langle f\rangle$, see, for instance, [31, 22, 19].

6.3 Specializing $(\star \to \star)$-indexed values

Promoting $\mathit{poly}\langle\cdot\rangle$ to types of arbitrary kind proceeds exactly as before except that we are now working on a function level, i.e., we work with lifted kinds and types. To begin with, the type of the promoted version is given by $H\langle\!\langle\cdot\rangle\!\rangle$, which is inductively defined as follows.

$$\begin{aligned} H\langle\!\langle u :: \uparrow\star\rangle\!\rangle &= H\ u \\ H\langle\!\langle u :: \uparrow\kappa_1 \to \kappa_2\rangle\!\rangle &= \forall x. H\langle\!\langle x :: \uparrow\kappa_1\rangle\!\rangle \to H\langle\!\langle u\ x :: \uparrow\kappa_2\rangle\!\rangle \end{aligned}$$

The promoted version of $\mathit{poly}\langle\cdot\rangle$ reads

$$\begin{aligned} \mathit{poly}\langle\!\langle t :: \kappa\rangle\!\rangle\ \varrho &:: H\langle\!\langle t :: \kappa\rangle\!\rangle \\ \mathit{poly}\langle\!\langle \underline{c}\rangle\!\rangle\ \varrho &= \mathit{poly}_{\underline{c}} \\ \mathit{poly}\langle\!\langle \underline{x}\rangle\!\rangle\ \varrho &= \varrho\ \underline{x} \\ \mathit{poly}\langle\!\langle t_1\ t_2\rangle\!\rangle\ \varrho &= (\mathit{poly}\langle\!\langle t_1\rangle\!\rangle\ \varrho)\ (\mathit{poly}\langle\!\langle t_2\rangle\!\rangle\ \varrho) \\ \mathit{poly}\langle\!\langle \Lambda\underline{x} \to t\rangle\!\rangle\ \varrho &= \lambda v \to \mathit{poly}\langle\!\langle t\rangle\!\rangle\ (\varrho(\underline{x} := v))\ . \\ \mathit{poly}\langle\!\langle \mu t\rangle\!\rangle\ \varrho &= \mathit{fix}\ (\mathit{poly}\langle\!\langle t\rangle\!\rangle\ \varrho) \end{aligned}$$

Note that $\mathit{poly}\langle\!\langle\cdot\rangle\!\rangle$ depends only on $\mathit{poly}_{\underline{c}}$ but not on poly_{Id}. The latter value is used when defining $\mathit{poly}\langle\cdot\rangle$ in terms of $\mathit{poly}\langle\!\langle\cdot\rangle\!\rangle$.

$$\mathit{poly}\langle t :: \star \to \star\rangle \ = \ \mathit{poly}\langle\!\langle\uparrow t :: \uparrow\star \to \star\rangle\!\rangle\ \epsilon\ \mathit{poly}_{Id} \qquad (3)$$

Thus, in order to specialize $\mathit{poly}\langle t\rangle$ we specialize $\mathit{poly}\langle\!\langle\uparrow t\rangle\!\rangle$, which is defined by induction on the structure of lifted types. The resulting function has type $\forall x. H\ x \to H\ ((\uparrow t)\ x)$. Supplying poly_{Id} as the first argument we obtain a value of type $H\ ((\uparrow t)\ Id)$, which is equal to $H\ t$. Again, we can view Eq. (3) as the soundness condition of the specialization.

One practical problem remains: the type of $\mathit{poly}\langle\!\langle\cdot\rangle\!\rangle$ uses lifted types. Consider, for instance, the type signature of $\mathit{size}\langle\!\langle\underline{List}\rangle\!\rangle$.

$$\mathit{size}\langle\!\langle\underline{List}\rangle\!\rangle :: \forall\underline{a}.(\forall x.\underline{a}\ x \to \mathit{Int}) \to (\forall x.\underline{List}\ \underline{a}\ x \to \mathit{Int})$$

If we intend to present the specialized program to a Haskell compiler, we must find a way of expressing $\underline{List}$ in terms of $List$. For type constructors of kind $\star \to \star$ the relationship is quite simple: we have, for instance, $\underline{List}\ \underline{a} = \Lambda z \to List\ (\underline{a}\ z)$, i.e., $\underline{List}\ \underline{a}$ equals the type composition $List \cdot \underline{a}$. Given this relation we can rewrite the type signature above.

$$\mathit{size}\langle\!\langle\underline{List}\rangle\!\rangle :: \forall\underline{a}.(\forall x.\underline{a}\ x \to \mathit{Int}) \to (\forall x.List\ (\underline{a}\ x) \to \mathit{Int})$$

For the general case we must delve a bit into the theory of combinators. To begin with we require type-level counterparts of the combinators K and S.

$$\begin{aligned} K\ t &= \Lambda x \to t \\ S\ t_1\ t_2 &= \Lambda x \to (t_1\ x)\ (t_2\ x) \end{aligned}$$

Next, we relate types of kind $\uparrow\kappa$ to types of kind $\star \to \kappa$.

Definition 4 *The relation $(\sim_\kappa) \subseteq (\uparrow\kappa) \times (\star \to \kappa)$ is defined by induction on the structure of kinds.*

$$\begin{aligned} t \sim_\star t' &\iff t = t' \\ t \sim_{\kappa_1 \to \kappa_2} t' &\iff \forall x, x'.(x \sim_{\kappa_1} x' \implies t\ x \sim_{\kappa_2} S\ t'\ x') \end{aligned}$$

In the base case '$\sim_\star$' relates types of the same kind, so we require them to be equal. 'Type functions' are related if related arguments are mapped to related results.

Proposition 1 *Let t be a type of kind κ, then*

$$\uparrow t \ \sim_\kappa\ K\ t\ .$$

For types of kind $\star$ we have $\uparrow t = K\ t$, i.e., the lifted version simply ignores its additional argument—which comes as little surprise. For types of kind $\star \to \star$ Prop. 1 implies $(\uparrow t)\ x = S\ (K\ t)\ x = t \cdot x$. Setting $x = Id$ we obtain $(\uparrow t)\ Id = t$, which shows that Eq. (2) is a direct consequence of Prop. 1. Now, let us apply Prop. 1 to rewrite the type signature of, say, $\mathit{size}\langle\!\langle\underline{Fix}\rangle\!\rangle$:

$$\begin{aligned} \mathit{size}\langle\!\langle\underline{Fix}\rangle\!\rangle :: \forall\underline{f}.(\forall\underline{a}.&(\forall x.\underline{a}\ x \to \mathit{Int}) \\ &\to (\forall x.\underline{f}\ \underline{a}\ x \to \mathit{Int})) \\ &\to (\forall x.\underline{Fix}\ \underline{f}\ x \to \mathit{Int}) \end{aligned}$$

For the second-order type Fix Prop. 1 reads

$$\underline{f}\ \underline{a} = \Lambda x \to (f\ x)\ (\underline{a}\ x) \implies \underline{Fix}\ \underline{f} = \Lambda x \to Fix\ (f\ x)\ .$$

Given this relationship we can replace $\underline{Fix}\ \underline{f}$ and $\underline{f}\ \underline{a}$ in $\mathit{size}\langle\!\langle\underline{Fix}\rangle\!\rangle$'s type signature.

$$\begin{aligned} \mathit{size}\langle\!\langle\underline{Fix}\rangle\!\rangle :: \forall f.(\forall\underline{a}.&(\forall x.\underline{a}\ x \to \mathit{Int}) \\ &\to (\forall x.(f\ x)\ (\underline{a}\ x) \to \mathit{Int})) \\ &\to (\forall x.Fix\ (f\ x) \to \mathit{Int}) \end{aligned}$$

Note that the rewrite involves the change of a bound variable: $\underline{f}$ of kind $(\star \to \star) \to (\star \to \star)$ is replaced by f of kind $\star \to (\star \to \star)$. This change is, however, perfectly fine since $\underline{f}$ is only instantiated to lifted types (by construction) and we know by Prop. 1 that lifted types of this kind can be expressed in terms of the original types. In other words, if we consistently change the types of the $\mathit{poly}\langle\!\langle\cdot\rangle\!\rangle$ functions, the resulting program is well-typed.

It is high time to consider some examples. Fig. 3 lists the specializations of *size* to the datatypes of Sec. 2. The code looks pretty similar to the code generated for *enc*, see Fig. 2. This is not surprising since the 'code generation' is completely independent of the kind of the type index. Only the types are more involved: since *size* has already a rank-1 type, $\mathit{size}\langle\!\langle\underline{F}\rangle\!\rangle$ is assigned a rank-$(n+1)$ type if F has an order-n kind.

In Sec. 5.3 we have already discussed two of the features the type system of the host language must support: rank-n types and polymorphic recursion. Here, we require an additional feature: a strong form of type constructor polymorphism [6]. Consider the call $\mathit{sizeSequ}\ (\mathit{sizeFork}\ \mathit{sizea})$ in the definition of *sizeSequ*: *sizeSequ* requires an argument of type $\forall x.a\ x \to \mathit{Int}$ while $\mathit{sizeFork}\ \mathit{sizea}$ has type $\forall x.Fork\ (a'\ x) \to \mathit{Int}$. To determine whether the call is well-typed we must solve the equation $a\ x = Fork\ (a'\ x)$. Setting $a = \Lambda z \to Fork\ (a'\ z)$ the call and the definition of *sizeSequ* can be type-checked. Clearly, some kind of higher-order unification is required here. Unfortunately, we know of no practical language that supports this feature. Haskell, for instance, uses a kinded first-order unification [28], which reduces $a\ x = Fork\ (a'\ x)$ to $a = Fork$ and $x = a'\ x$. The latter equation is, however, not solvable. The lack of type constructor polymorphism is also noted in [6], which suggests that generalizing Haskell's type system might be worthwhile. Let us remark that this problem disappears if we switch to a language with explicit type annotations. Suitable candidates are the intermediate language of the Glasgow Haskell Compiler [38], which is based on the second-order λ-calculus [11], or the language Henk [39], which is based on Barendregt's λ-cube [3].

```
sizeList                          :: ∀a.(∀x.a x → Int) → (∀x.List (a x) → Int)
sizeList sizea                    =  sizel
    where sizel Nil               =  0
          sizel (Cons x xs)       =  sizea x + sizel xs
sizeRose                          :: ∀a.(∀x.a x → Int) → (∀x.Rose (a x) → Int)
sizeRose sizea                    =  sizer
    where sizer (Branch x ts)     =  sizea x + sizeList sizer ts
sizeGRose                         :: ∀f.(∀b.(∀x.b x → Int) → (∀x.(f x) (b x) → Int))
                                       → (∀a.(∀x.a x → Int) → (∀x.GRose (f x) (a x) → Int))
sizeGRose sizef sizea             =  sizeg
    where sizeg (GBranch x ts)    =  sizea x + sizef sizeg ts
sizeFix                           :: ∀f.(∀b.((∀x.b x → Int) → (∀x.(f x) (b x) → Int)))
                                       → (∀x.Fix (f x) → Int)
sizeFix sizef                     =  sizer
    where sizer (In x)            =  sizef sizer x
sizeBaseList                      :: ∀a.(∀x.a x → Int) → (∀b.(∀x.b x → Int) → (∀x.BaseList (a x) (b x) → Int))
sizeBaseList sizea sizeb Nil      =  0
sizeBaseList sizea sizeb (Cons x y) = sizea x + sizeb y
sizeFork                          :: ∀a.(∀x.a x → Int) → (∀x.Fork (a x) → Int)
sizeFork sizea (Node x1 x2)       =  sizea x1 + sizea x2
sizeSequ                          :: ∀a.(∀x.a x → Int) → (∀x.Sequ (a x) → Int)
sizeSequ sizea Empty              =  0
sizeSequ sizea (Zero s)           =  sizeSequ (sizeFork sizea) s
sizeSequ sizea (One x s)          =  sizea x + sizeSequ (sizeFork sizea) s
sizeMapFork                       :: ∀m.(∀w.(∀x.w x → Int) → (∀x.(m x) (w x) → Int))
                                        → (∀v.(∀x.v x → Int) → (∀x.MapFork (m x) (v x) → Int))
sizeMapFork sizem sizev (TrieFork tf)
                                  =  sizem (sizem sizev) tf
sizeMapSequ                       :: ∀m.(∀w.(∀x.w x → Int) → (∀x.(m x) (w x) → Int))
                                        → (∀v.(∀x.v x → Int) → (∀x.MapSequ (m x) (v x) → Int))
sizeMapSequ sizem sizev (TrieSequ te tz to)
                                  =  sizev te
                                  +  sizeMapSequ (sizeMapFork sizem) sizev tz
                                  +  sizem (sizeMapSequ (sizeMapFork sizem) sizev) to
sizeSquare                        :: ∀a.(∀x.a x → Int) → (∀x.Square (a x) → Int)
sizeSquare sizea m                =  sizeSquare' sizeNil sizea m
sizeSquare'                       :: ∀f.(∀b.(∀x.b x → Int) → (∀x.(f x) (b x) → Int))
                                       → (∀a.(∀x.a x → Int) → (∀x.Square' (f x) (a x) → Int))
sizeSquare' sizef sizea (Zero m)  =  sizef (sizef sizea) m
sizeSquare' sizef sizea (Succ m)  =  sizeSquare' (sizeCons sizef) sizea m
sizeNil                           :: ∀a.(∀x.a x → Int) → (∀x.Nil (a x) → Int)
sizeNil sizea Nil                 =  0
sizeCons                          :: ∀f.(∀b.(∀x.b x → Int) → (∀x.(f x) (b x) → Int))
                                       → (∀a.(∀x.a x → Int) → (∀x.Cons (f x) (a x) → Int))
sizeCons sizef sizea (Cons x xs)  =  sizea x + sizef sizea xs
```

Figure 3: Specializing *size* to the types of Sec. 2.

6.4 Generalizing to first-order kinds

So far we have considered generic values indexed by type constructors of kind $\star \to \star$. The generalization to type indices of first-order kind is straightforward. Assume that the type index has kind $\star^n \to \star$, which abbreviates $\underbrace{\star \to \cdots \to \star}_{n \text{ times}} \to \star$. We now redefine lifting and put $\uparrow\star = \star^n \to \star$; the case $n = 1$ specializes to the preceding treatment. The lifted version of a primitive type constructor c of arity m is given by

$$\begin{aligned}&\underline{c}\, f_1 \ldots f_m \\ &= \Lambda a_1 \ldots a_n \to c\,(f_1\, a_1 \ldots a_n) \ldots (f_m\, a_1 \ldots a_n)\ .\end{aligned}$$

To define a value indexed by a type of kind $\star^n \to \star$ the programmer must provide cases for each of the lifted primitive type constructors and additionally for n 'projection types': $\pi_i^n = \Lambda a_1 \ldots a_n \to a_i$ with $1 \leqslant i \leqslant n$ (for $n = 1$ we have $\pi_1^1 = \Lambda a_1 \to a_1 = Id$). Specializing a generic value works exactly as before except that the 'initial call' now uses $poly_{\pi_1^n}$, $\ldots$, $poly_{\pi_n^n}$.

$$\begin{aligned}&poly\langle t :: \star^n \to \star\rangle \\ &\quad = poly\langle\!\langle \uparrow t :: \uparrow\star^n \to \star\rangle\!\rangle\ \epsilon\ poly_{\pi_1^n} \ldots poly_{\pi_n^n}\end{aligned}$$

6.5 Limitations of the approach

Let us briefly discuss the difficulties in extending generic programming to higher-order kinds. Interestingly, the approach still works for type indices of second-order kind. As an example, consider a generic value indexed by a type constructor of kind $(\star \to \star) \to \star \to \star$. To characterize the set of normal forms we proceed as before: η-expanding a given type t of that kind we obtain $t = \Lambda a_1\ a_2 \to t\ a_1\ a_2$. The body of the abstraction has kind $\star$ and its normal form can be characterized in the usual way—the variables $a_1 :: \star \to \star$ and $a_2 :: \star$ are simply treated as additional constants. Abstracting a_1 and a_2 out yields the following grammar

$$\begin{aligned}T_{(\star\to\star)\to\star\to\star}\ ::=\ &\pi_1^2 \\ |\ &\pi_2^2\ T_{(\star\to\star)\to\star\to\star} \\ |\ &\underline{1} \\ |\ &\underline{Int} \\ |\ &(T_{(\star\to\star)\to\star\to\star} \mathbin{\underline{+}} T_{(\star\to\star)\to\star\to\star}) \\ |\ &(T_{(\star\to\star)\to\star\to\star} \mathbin{\underline{\times}} T_{(\star\to\star)\to\star\to\star})\ ,\end{aligned}$$

where $\pi_1^2 = \Lambda a_1\ a_2 \to a_1$, $\pi_2^2\ f_1 = \Lambda a_1\ a_2 \to a_2\ (f_1\ a_1\ a_2)$, and the lifted type constructors are defined as in Sec. 6.4. Thus, the type patterns for $((\star \to \star) \to \star \to \star)$-indexed values are similar to the ones in the first-order case except that the 'projection types' are slightly more complicated.

Unfortunately, this scheme breaks down at the third level. To illustrate, consider a type $t = \Lambda a \to t\ a$ of kind $\kappa \to \star$ with $\kappa = (\star \to \star) \to \star$. The normal form of $t\ a$ is described by the following grammar.

$$T_\star\ ::=\ a\ T_{\star\to\star} \mid 1 \mid Int \mid T_\star + T_\star \mid T_\star \times T_\star$$

Note that $T_\star$ refers to $T_{\star\to\star}$. This means that a generic value, which is indexed by a type constructor of kind $\kappa \to \star$, cannot be inductively defined on the structure of types.

For similar reasons the set C of primitive type constructors must not contain types of second-order kind or higher. To see why assume that $Fix :: (\star \to \star) \to \star$ is primitive. Since Fix's argument is a type constructor, we can no longer define generic values inductively: $enc\langle Fix\ f\rangle$, for instance, cannot fall back on $enc\langle f\rangle$ since f has not kind $\star$. To summarize: our approach is limited to type indices up to second-order kinds and to primitive types up to first-order kinds. Whether these restrictions are severe in practice remains to be seen. We suspect that this is not the case. Finally, let us stress that these restrictions do not affect the ability to specialize generic values, which works for types of arbitrary kinds.

7 Towards Haskell

The specialization of generic values as described in Sec. 5.3 and 6.3 places high demands on the type system: it requires polymorphic recursion, rank-n types, and a strong form of type constructor polymorphism. Haskell 98 [37], for instance, only supports polymorphic recursion. In this section we show that the requirement for rank-n types can be alleviated since they can be encoded using so-called dictionaries. This encoding also provides an interesting link to Haskell's type classes.

Recent extensions of Haskell as implemented in GHC [36] and in Hugs 98 [29] provide rank-2 type signatures and local universal quantification in datatypes. Using the latter feature we can circumvent rank-n types. The idea is very simple: instead of passing a polymorphic value directly as an argument we pass a dictionary that contains the value as the single component. Of course, we require not only a single dictionary but a kind-indexed family of dictionaries. The following definitions introduce suitable dictionaries for the *enc* function.

$$\begin{aligned}&\textbf{type}\ Enc\langle\star\rangle\ a &&= a \to Bin \\ &\textbf{type}\ Enc\langle\kappa_1 \to \kappa_2\rangle\ a &&= \forall x.EncD\langle\kappa_1\rangle\ x \to Enc\langle\kappa_2\rangle\ (a\ x) \\ &\textbf{data}\ EncD\langle\kappa\rangle\ a &&= EncD_\kappa\{enc_\kappa :: Enc\langle\kappa\rangle\ a\}\end{aligned}$$

The promoted function $enc\langle\!\langle t :: \kappa\rangle\!\rangle$ now has type $Enc\langle\kappa\rangle\ t$. Fig. 4 displays the specializations of *enc* for some types. The dictionary translation is interesting in at least two respects. First, it suggests an easy way of dealing with mutually recursive generic definitions, see [17] for examples. We simply use dictionaries with multiple entries, one for each recursive function. Second, it relates generic definitions to Haskell's type classes [13], which use a similar implementation. Recall that an overloaded function is translated into a non-overloaded function that takes as an additional argument the dictionary containing the operations of the class. For instance, the class definition

$$\textbf{class}\ Eq\ a\ \textbf{where}\ (==), (/=) :: a \to a \to Bool$$

gives rise to the following dictionary type.

$$\textbf{data}\ EqD\ a\ =\ EqD\{(==), (/=) :: a \to a \to Bool\}$$

Each instance declaration of the form **instance** $Eq\ T$ defines an element of $EqD\ T$ and each declaration of the form **instance** $(Eq\ a) \Rightarrow Eq\ (T\ a)$ defines a function of type $\forall a.EqD\ a \to EqD\ (T\ a)$. Thus, generic definitions and type classes use the same mechanism on the implementation level. Using the **deriving** construct instance declarations can even be automatically generated for user-defined, first-order kinded datatypes. However, instance declarations are too limited to handle types of higher-order kinds. For instance, **deriving** *Eq* for *MapFork* or *MapSequ* fails with a compile-time error in Haskell 98. Thus, generic definitions generalize type classes in some respects: they allow us to define values generically for *all* types—while class instances must be programmed by hand, except for some predefined classes like *Eq*—and the specialization is not restricted to first-order kinded types. Note that instance declarations can be mimicked in our framework by so-called ad-hoc definitions, see [17]. For instance, an ad-hoc compression scheme for lists, which yields better compression rates than the generic scheme, can be defined by

$$\begin{aligned} enc\langle List\ a\rangle\ xs &= encInt\ (length\ xs) \\ &\mathbin{+\!\!+}\ concat\ (map\ (enc\langle a\rangle)\ xs)\ . \end{aligned}$$

This equation extends the definition of *enc* given in Sec. 3 and specifies an exception to the general scheme.

8 Related work

Generic programming The concept of generic functional programming trades under a variety of names: F. Ruehr refers to this concept as *structural polymorphism* [41, 40], T. Sheard calls generic functions *type parametric* [43], C.B. Jay and J.R.B. Cocket use the term *shape polymorphism* [25], R. Harper and G. Morrisett [14] coined the phrase *intensional polymorphism*, and J. Jeuring invented the word *polytypism* [26].

The mainstream of generic programming is based on the initial algebra semantics of datatypes, see, for instance [12], and puts emphasis on general recursion operators like *map* and catamorphisms (folds). In [42] several variations of these operators are informally defined and algorithms are given that specialize these functions for given datatypes. The programming language *Charity* [8] automatically provides *map* and catamorphisms for each user-defined datatype. Since general recursion is not available, Charity is strongly normalizing. *Functorial ML* [24] has a similar functionality, but a different background. It is based on the theory of *shape polymorphism*, in which values are separated into shape and contents. The polytypic programming language extension *PolyP* [21]—already mentioned in the introduction—offers a special construct for defining generic functions. The generic definitions are similar to ours (modulo notation) except that the generic programmer must additionally consider cases for type composition and for type recursion (see [20] for a more detailed comparison).

All the approaches are restricted to first-order kinded, regular datatypes (or even subsets of this class). One notable exception is the work of F. Ruehr [41], who presents a higher-order language based on a type system related to ours (only type recursion is missing). Genericity is achieved through the use of type patterns which are interpreted at run-time. By contrast, the technique presented here does not require the passing of types or representations of types at run-time. This also distinguishes our approach from the work on intensional polymorphism [14, 10] where a **typecase** is used for defining type-dependent operations.

This paper can be regarded as a successor to [20], where a similar approach restricted to first-order kinded types is presented. A companion paper [17] contains a concrete proposal for a generic programming extension of Haskell, which is based on the theoretical framework developed here.

Type systems Several variations of the type system given in Sec. 4 have been described in the literature, see [3] for a good survey article. For instance, if we drop the fixpoint operator from the type language, we obtain the system $\lambda\underline{\omega}$, which forms one corner of Barendregt's λ-cube. Dropping type application and abstraction yields the system $\lambda\mu$, which supports structural equivalence of types. This system is, however, restricted to types of kind $\star$ and cannot handle parametric types. To the best of the author's knowledge the combination of $\lambda\underline{\omega}$ and $\lambda\mu$ (with a polymorphic fixpoint operator) is original.

Type inference algorithms for languages with generic constructs have been developed by F. Ruehr [41], C.B. Jay *et.al.* [24], and P. Jansson and J. Jeuring [21]. Note that our system does not permit type reconstruction in general. Consider the expression *size xs* where *xs* has type *List* (*Rose Int*). Should *size* count the number of integers or the number of rose trees in the list?

9 Conclusion

We have presented a new approach to generic functional programming, which is both simpler—from the generic programmer's point of view—and considerably more general than previous work—the complete type system of Haskell is covered (previous approaches were limited to first-order kinded, regular types). The basic idea is to model types by terms of the simply typed λ-calculus augmented by a family of recursion operators. Specializing a generic value can be seen as an interpretation of simply typed λ-terms. The generated code places high demands on the type system of the underlying language: polymorphic recursion, rank-n types, and a strong form of type constructor polymorphism are required. We have shown that rank-n types can be circumvented using a dictionary translation, which provides an interesting link to Haskell's type classes. In particular, generic definitions generalize Haskell's **deriving** construct.

Acknowledgements

I am grateful to Ian Bayley, Johan Jeuring, Lambert Meertens, and Fritz Ruehr for suggesting dozens of improvements regarding contents and presentation. Thanks are also due to four anonymous referees for their detailed and helpful comments.

References

[1] M. Abadi, L. Cardelli, B. Pierce, and D. Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, January 1995.

[2] H. P. Barendregt. *The Lambda Calculus — Its Syntax and Semantics.* North-Holland, Amsterdam New York Oxford, revised edition, 1984.

[3] H.P. Barendregt. Lambda calculi with types. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 2, Background: Computational Structures*, pages 118–309. Clarendon Press, Oxford, 1992.

[4] Richard Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall Europe, London, 2nd edition, 1998.

$$
\begin{array}{lcl}
\mathbf{data}\ \mathit{EncD}_\star\ t & = & \mathit{EncD}_\star\{\mathit{enc}_\star :: t \to \mathit{Bin}\} \\
\mathbf{data}\ \mathit{EncD}_{\star\to\star}\ t & = & \mathit{EncD}_{\star\to\star}\{\mathit{enc}_{\star\to\star} :: \forall x.\mathit{EncD}_\star\ x \to (t\ x \to \mathit{Bin})\} \\
\mathit{encGRose} & :: & \forall f, a.\mathit{EncD}_{\star\to\star}\ f \to \mathit{EncD}_\star\ a \to (\mathit{GRose}\ f\ a \to \mathit{Bin}) \\
\mathit{encGRose}\ \mathit{df}\ \mathit{da} & = & \mathit{encg} \\
\quad \mathbf{where}\ \mathit{encg}\ (\mathit{GBranch}\ x\ \mathit{ts}) & = & \mathit{enc}_\star\ \mathit{da}\ x + \!\!+\ \mathit{enc}_{\star\to\star}\ \mathit{df}\ (\mathit{EncD}_\star\{\mathit{enc}_\star = \mathit{encg}\})\ \mathit{ts} \\
\mathit{encFork} & :: & \forall a.\mathit{EncD}_\star\ a \to (\mathit{Fork}\ a \to \mathit{Bin}) \\
\mathit{encFork}\ \mathit{da}\ (\mathit{Node}\ x_1\ x_2) & = & \mathit{enc}_\star\ \mathit{da}\ x_1 + \!\!+\ \mathit{enc}_\star\ \mathit{da}\ x_2 \\
\mathit{encSequ} & :: & \forall a.\mathit{EncD}_\star\ a \to (\mathit{Sequ}\ a \to \mathit{Bin}) \\
\mathit{encSequ}\ \mathit{da}\ \mathit{Empty} & = & [0] \\
\mathit{encSequ}\ \mathit{da}\ (\mathit{Zero}\ s) & = & 1 : 0 : \mathit{encSequ}\ (\mathit{EncD}_\star\{\mathit{enc}_\star = \mathit{encFork}\ \mathit{da}\})\ s \\
\mathit{encSequ}\ \mathit{da}\ (\mathit{One}\ x\ s) & = & 1 : 1 : \mathit{enc}_\star\ \mathit{da}\ x + \!\!+\ \mathit{encSequ}\ (\mathit{EncD}_\star\{\mathit{enc}_\star = \mathit{encFork}\ \mathit{da}\})\ s
\end{array}
$$

Figure 4: Dictionary implementation of *enc*.

[5] 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.

[6] Richard Bird and Ross Paterson. De Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, January 1999.

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

[8] Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report 92/480/18, Dept. of Computer Science, Univ. of Calgary, June 1992.

[9] Bruno Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25(2):95–169, March 1983.

[10] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. *ACM SIGPLAN Notices*, 34(1):301–312, 1999.

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

[12] T. Hagino. *Category Theoretic Approach to Data Types.* PhD thesis, University of Edinburgh, 1987.

[13] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, March 1996.

[14] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In ACM, editor, *Conference record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'95, San Francisco, California*, pages 130–141. ACM-Press, 1995.

[15] Ralf Hinze. Numerical representations as higher-order nested datatypes. Technical Report IAI-TR-98-12, Institut für Informatik III, Universität Bonn, December 1998.

[16] Ralf Hinze. Generalizing generalized tries. *Journal of Functional Programming*, 1999. Accepted for publication.

[17] Ralf Hinze. A generic programming extension for Haskell. In Erik Meijer, editor, *Proceedings of the 3rd Haskell Workshop, Paris, France*, September 1999. The proceedings appear as a technical report of Universiteit Utrecht, UU-CS-1999-28.

[18] Ralf Hinze. Manufacturing datatypes. In Chris Okasaki, editor, *Proceedings of the Workshop on Algorithmic Aspects of Advanced Programming Languages, WAAAPL'99, Paris, France*, pages 1–16, September 1999. The proceedings appear as a technical report of Columbia University, CUCS-023-99, also available from `http://www.cs.columbia.edu/~cdo/waaapl.html`.

[19] Ralf Hinze. Polytypic functions over nested datatypes. *Discrete Mathematics and Theoretical Computer Science*, 3(4):159–180, 1999.

[20] Ralf Hinze. Polytypic programming with ease (extended abstract). In *4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99), Tsukuba, Japan*, Lecture Notes in Computer Science. Springer-Verlag, November 1999. To appear.

[21] 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.

[22] 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.

[23] Patrik Jansson and Johan Jeuring. Polytypic compact printing and parsing. In S. Doaitse Swierstra, editor, *Proceedings European Symposium on Programming, ESOP'99*, volume 1576 of *Lecture Notes in Computer Science*, pages 273–287, Berlin, 1999. Springer-Verlag.

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

[25] C.B. Jay and J.R.B. Cocket. Shapely types and shape polymorphism. In D. Sanella, editor, *Programming Languages and Systems — ESOP'94: 5th European Symposium on Programming, Edinburgh, UK, Proceedings*, volume 788 of *Lecture Notes in Computer Science*, pages 302–316, Berlin, 11–13 April 1994. Springer-Verlag.

[26] Johan Jeuring and Patrik Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Tutorial Text 2nd International School on Advanced Functional Programming, Olympia, WA, USA*, volume 1129 of *Lecture Notes in Computer Science*, pages 68–114. Springer-Verlag, 1996.

[27] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1):1–35, January 1995.

[28] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In *First International Spring School on Advanced Functional Programming Techniques*, volume 925 of *Lecture Notes in Computer Science*, pages 97–136. Springer-Verlag, 1995.

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

[30] Nancy Jean McCracken. The typechecking of programs with implicit type structure. In Gilles Kahn, David B. MacQueen, and Gordon D. Plotkin, editors, *Semantics of Data Types: International Symposium, Sophia-Antipolis, France*, volume 173 of *Lecture Notes in Computer Science*, pages 301–315. Springer-Verlag, 1984.

[31] 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.

[32] 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.

[33] 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.

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

[35] Chris Okasaki. From fast exponentiation to square matrices: An adventure in types. In Peter Lee, editor, *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming, Paris, France*, pages 28–35, September 1999.

[36] Simon Peyton Jones. Explicit quantification in Haskell, 1998. Available from `http://research.microsoft.com/Users/simonpj/Haskell/quantification.html`.

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

[38] 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.

[39] Simon L. Peyton Jones and Erik Meijer. Henk: A typed intermediate language. In *Proceedings of the Types in Compilation Workshop, Amsterdam, June*, 1997. Available from `http://www.cs.bc.edu/~muller/TIC97/`.

[40] Fritz Ruehr. Structural polymorphism. In Roland Backhouse and Tim Sheard, editors, *Informal Proceedings Workshop on Generic Programming, WGP'98, Marstrand, Sweden, 18 June 1998.* Dept. of Computing Science, Chalmers Univ. of Techn. and Göteborg Univ., June 1998.

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

[42] Tim Sheard. Automatic generation and use of abstract structure operators. *ACM Transactions on Programming Languages and Systems*, 13(4):531–557, October 1991.

[43] Tim Sheard. Type parametric programming. Technical report CS/E 93-018, Oregon Graduate Institute of Science and Technology, Department of Computer Science and Engineering, Portland, OR, USA, November 1993.

[44] 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.

[45] Zhe Yang. Encoding types in ML-like languages. *SIGPLAN Notices*, 34(1):289–300, January 1999.