Generic Haskell: practice and theory

Ralf Hinze[1] and Johan Jeuring[2,3]

[1] Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
`ralf@informatik.uni-bonn.de`
`http://www.informatik.uni-bonn.de/~ralf/`
[2] Institute of Information and Computing Sciences, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
`johanj@cs.uu.nl`
`http://www.cs.uu.nl/~johanj/`
[3] Open University, Heerlen, The Netherlands

Abstract. Generic Haskell is an extension of Haskell that supports the construction of generic programs. These lecture notes describe the basic constructs of Generic Haskell and highlight the underlying theory.

Generic programming aims at making programming more effective by making it more general. Generic programs often embody non-traditional kinds of polymorphism. Generic Haskell is an extension of Haskell [38] that supports the construction of generic programs. Generic Haskell adds to Haskell the notion of *structural polymorphism*, the ability to define a function (or a type) by induction on the structure of types. Such a function is generic in the sense that it works not only for a specific type but for a whole class of types. Typical examples include equality, parsing and pretty printing, serialising, ordering, hashing, and so on.

The lecture notes on Generic Haskell are organized into two parts. This first part motivates the need for genericity, describes the basic constructs of Generic Haskell, puts Generic Haskell into perspective, and highlights the underlying theory. The second part entitled "Generic Haskell: applications" delves deeper into the language discussing three non-trivial applications of Generic Haskell: generic dictionaries, compressing XML documents, and a generic version of the zipper data type.

The first part is organized as follows. Section 1 provides some background discussing type systems in general and the type system of Haskell in particular. Furthermore, it motivates the basic constructs of Generic Haskell. Section 2 takes a closer look at generic definitions and shows how to define some popular generic functions. Section 3 highlights the theory underlying Generic Haskell and discusses its implementation. Section 4 concludes.

1 Introduction

This section motivates and introduces the basic constructs of Generic Haskell. We start by looking at type systems.

A basic knowledge of *Haskell* is desirable, as all the examples are given either in Haskell or in Generic Haskell.

1.1 Type systems

Safe languages Most programmers probably agree that *language safety* is a good thing. Language safety is quite a colorful term meaning different things to different people. Here are a few definitions taken from Pierce's excellent text book "Types and Programming Languages" [40].

– A *safe language* is one that makes it impossible to shoot yourself in the foot while programming.
– A *safe language* is one that protects its own abstractions.
– A *safe language* is one that that prevents untrapped errors at run time.
– A *safe language* is completely defined by its programmer's manual.

The definitions put emphasis on different aspects of language safety. Quite clearly, all of these are desirable properties of a programming language.

Now, language safety can be achieved by *static type checking*, by *dynamic type checking*, or—and this is the most common case—by a combination of static and dynamic checks. The language Haskell serves as an example of the latter approach: passing an integer to a list-processing function is captured statically at compile time while taking the first element of the empty list results in a run-time error.

Static and dynamic typing It is widely accepted that static type systems are indispensable for building large and reliable software systems. The most cited benefits of static typing include:

– Programming errors are detected at an early stage.
– Type systems enforce disciplined programming.
– Types promote abstraction (abstract data types, module systems).
– Types provide machine-checkable documentation.

However, type systems are always *conservative*: they must necessarily reject programs that behave well at run time.

In a sense, generic programming is about extending the boundaries of static type systems. This is the reason why these lecture notes have little to offer for addicts of dynamically typed languages. As we will see, most generic programs can be readily implemented in a dynamically checked language. (Conceptually, a dynamic language offers one universal data type; programming a function that works for a class of data types is consequently a non-issue.)

Polymorphic type systems Polymorphism complements type security by flexibility. Polymorphic type systems like the Hindley-Milner system [33] allow the definition of functions that behave uniformly over all types. A standard example is the *length* function that computes the length of a list.

$$\begin{aligned}
&\textbf{data}\ \mathsf{List\ a} &&= \mathit{Nil} \mid \mathit{Cons}\ \mathsf{a}\ (\mathsf{List\ a})\\
&\mathit{length} &&:: \forall \mathsf{a} \,.\, \mathsf{List\ a} \to \mathsf{Int}\\
&\mathit{length}\ \mathit{Nil} &&= 0\\
&\mathit{length}\ (\mathit{Cons}\ a\ as) &&= 1 + \mathit{length}\ as
\end{aligned}$$

The first line declares the list data type, which is parametric in the type of list elements. The function *length* happens to be insensitive to the element type. This is signalled by the universal quantifier in *length*'s type signature (read: $\mathsf{List\ a} \to \mathsf{Int}$ is a valid type of *length* for all types a). Though this is not Haskell 98 syntax, we will write polymorphic types always using explicit qualifiers. Most readers probably know the universal quantifier from predicate logic. Indeed, there is a close correspondence between polymorphic type systems and systems of higher-order logic, see [47]. In light of this correspondence we note that the quantifier in *length*'s type signature is *second-order* as it ranges over sets (if we naively equate types with sets).

However, even polymorphic type systems are sometimes less flexible than one would wish. For instance, it is not possible to define a polymorphic *equality function* that works for all types.

$$\mathit{eq} :: \forall \mathsf{a} \,.\, \mathsf{a} \to \mathsf{a} \to \mathsf{Bool} \qquad \text{-- does not work}$$

The parametricity theorem [46] implies that a function of type $\forall \mathsf{a} \,.\, \mathsf{a} \to \mathsf{a} \to \mathsf{Bool}$ must necessarily be constant. As a consequence, the programmer is forced to program a separate equality function for each type from scratch. This sounds like a simple task but may, in fact, be arbitrarily involved. To illustrate some of the difficulties we will go through a series of instances of equality (and other generic functions). First, however, let us take a closer look at Haskell's type system, especially at the **data** construct.

1.2 Haskell's data construct

Haskell offers one basic construct for defining new types: a so-called *data type declaration*. In general, a **data** declaration has the following form:

$$\textbf{data}\ \mathsf{B}\ \mathsf{a}_1\ \ldots\ \mathsf{a}_m = K_1\ \mathsf{t}_{11}\ \ldots\ \mathsf{t}_{1m_1} \mid \cdots \mid K_n\ \mathsf{t}_{n1}\ \ldots\ \mathsf{t}_{nm_n}.$$

This definition simultaneously introduces a new type constructor B and n data or value constructors $K_1, \ldots, K_n$, whose types are given by

$$K_j :: \forall \mathsf{a}_1\ \ldots\ \mathsf{a}_m \,.\, \mathsf{t}_{j1} \to \cdots \to \mathsf{t}_{jm_j} \to \mathsf{B}\ \mathsf{a}_1\ \ldots\ \mathsf{a}_m.$$

The type parameters $\mathsf{a}_1, \ldots, \mathsf{a}_m$ must be distinct and may appear on the right-hand side of the declaration. If $m > 0$, then B is called a *parameterized type*. Data type declarations can be recursive, that is, B may also appear on the right-hand side. In general, data types are defined by a system of mutually recursive data type declarations. A Haskell data type is essentially a *sum of products*: the components of the sum are labelled by constructor names; the arguments of a constructor form a product.

The following sections provide several examples of data type declarations organized in increasing order of difficulty.

Finite types Data type declarations subsume enumerated types. In this special case, we only have nullary data constructors, that is, $m_1 = \cdots = m_n = 0$. The following declaration defines a simple enumerated type, the type of truth values.

$$\textbf{data}\ \mathsf{Bool} = \mathit{False} \mid \mathit{True}$$

Data type declarations also subsume record types. In this case, we have only one value constructor, that is, $n = 1$.

$$\textbf{data}\ \mathsf{Fork\ a} = \mathit{Fork}\ \mathsf{a\ a}$$

An element of Fork a is a pair whose two components both have type a. This example illustrates that we can use the same name for a type and for a data constructor. In these notes we distinguish the two by using different fonts: data constructors are set in Roman and type constructors in Sans Serif.

Haskell assigns a *kind* to each type constructor. One can think of a kind as the 'type' of a type constructor. The type constructor Fork defined above has kind $\star \to \star$. The '$\star$' kind represents nullary constructors like Char, Int or Bool. The kind $\kappa \to \nu$ represents type constructors that map type constructors of kind κ to those of kind ν. Note that the term 'type' is sometimes reserved for nullary type constructors.

The following type can be used to represent 'optional values'.

$$\textbf{data}\ \mathsf{Maybe\ a} = \mathit{Nothing} \mid \mathit{Just}\ \mathsf{a}$$

An element of type Maybe a is an 'optional a': it is either of the form *Nothing* or of the form *Just a* where a is of type a. The type constructor Maybe has kind $\star \to \star$.

Recursive types Data type declarations may be recursive or even mutually recursive. A simple recursive data type is the type of natural numbers.

$$\textbf{data}\ \mathsf{Nat} = \mathit{Zero} \mid \mathit{Succ}\ \mathsf{Nat}$$

The number 6, for instance, is given by

$$\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ \mathit{Zero}))))).$$

Strings can also be represented by a recursive data type.

$$\textbf{data}\ \mathsf{String} = \mathit{NilS} \mid \mathit{ConsS}\ \mathsf{Char\ String}$$

The type String is a binary sum. The first summand, *Nil*, is a nullary product and the second summand, *Cons*, is a binary product. Here is an example element of String:

$$\begin{array}{l} \mathit{ConsS}\ \texttt{'F'}\ (\mathit{ConsS}\ \texttt{'l'}\ (\mathit{ConsS}\ \texttt{'o'}\ (\mathit{ConsS}\ \texttt{'r'} \\ \quad (\mathit{ConsS}\ \texttt{'i'}\ (\mathit{ConsS}\ \texttt{'a'}\ (\mathit{ConsS}\ \texttt{'n'}\ \mathit{NilS})))))). \end{array}$$

The most popular data type is without doubt the type of parametric lists; it is obtained from String by abstracting over Char.

$$\textbf{data}\ \mathsf{List\ a} = \mathit{Nil} \mid \mathit{Cons}\ \mathsf{a}\ (\mathsf{List\ a})$$

The empty list is denoted *Nil*; *Cons a x* denotes the list whose first element is *a* and whose remaining elements are those of *x*. The list of the first six prime numbers, for instance, is given by

$$\mathit{Cons}\ 2\ (\mathit{Cons}\ 3\ (\mathit{Cons}\ 5\ (\mathit{Cons}\ 7\ (\mathit{Cons}\ 11\ (\mathit{Cons}\ 13\ \mathit{Nil}))))).$$

In Haskell, lists are predefined with special syntax: List a is written [a], *Nil* is replaced by [ ], and *Cons a x* by $a : x$. We will use both notations simultaneously.

The following definition introduces external binary search trees.

$$\textbf{data}\ \mathsf{Tree\ a\ b} = \mathit{Tip}\ \mathsf{a} \mid \mathit{Node}\ (\mathsf{Tree\ a\ b})\ \mathsf{b}\ (\mathsf{Tree\ a\ b})$$

We distinguish between external nodes of the form *Tip a* and internal nodes of the form *Node l b r*. The former are labelled with elements of type a while the latter are labelled with elements of type b. Here is an example element of type Tree Bool Int:

$$\mathit{Node}\ (\mathit{Tip\ True})\ 7\ (\mathit{Node}\ (\mathit{Tip\ True})\ 9\ (\mathit{Tip\ False})).$$

The type Tree has kind $\star \to \star \to \star$. Perhaps surprisingly, binary type constructors like Tree are curried in Haskell.

The following data type declaration captures multiway branching trees, also known as *rose trees* [6].

$$\textbf{data}\ \mathsf{Rose\ a} = \mathit{Branch}\ \mathsf{a}\ (\mathsf{List}\ (\mathsf{Rose\ a}))$$

A node is labelled with an element of type a and has a list of subtrees. An example element of type Rose Int is:

$$\begin{array}{l} \mathit{Branch}\ 2\ (\mathit{Cons}\ (\mathit{Branch}\ 3\ \mathit{Nil}) \\ \qquad\quad (\mathit{Cons}\ (\mathit{Branch}\ 5\ \mathit{Nil}) \\ \qquad\quad (\mathit{Cons}\ (\mathit{Branch}\ 7\ (\mathit{Cons}\ (\mathit{Branch}\ 11\ \mathit{Nil}) \\ \qquad\qquad\qquad\qquad\qquad (\mathit{Cons}\ (\mathit{Branch}\ 13\ \mathit{Nil})\ \mathit{Nil})))\ \mathit{Nil}))). \end{array}$$

The type Rose falls back on the type List. Instead, we may introduce Rose using two mutually recursive data type declarations:

$$\begin{array}{l} \textbf{data}\ \mathsf{Rose'\ a}\ \ = \mathit{Branch'}\ \mathsf{a}\ (\mathsf{Forest\ a}) \\ \textbf{data}\ \mathsf{Forest\ a} = \mathit{NilF} \mid \mathit{ConsF}\ (\mathsf{Rose'\ a})\ (\mathsf{Forest\ a}). \end{array}$$

Now Rose' depends on Forest and vice versa.

The type parameters of a data type may range over type constructors of arbitrary kinds. By contrast, Miranda (trademark of Research Software Ltd), Standard ML, and previous versions of Haskell (1.2 and before) only have first-order kinded data types. The following generalization of rose trees, that abstracts over the List data type, illustrates this feature.

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

A slight variant of this definition has been used by [37] to extend an implementation of priority queues with an efficient merge operation. The type constructor GRose has kind $(\star \to \star) \to (\star \to \star)$, that is, GRose has a so-called *second-order kind* where the order of a kind is given by

$$\begin{aligned} \mathit{order}(\star) &= 0 \\ \mathit{order}(\kappa \to \nu) &= \mathit{max}\{1 + \mathit{order}(\kappa), \mathit{order}(\nu)\}. \end{aligned}$$

Applying GRose to List yields the type of rose trees.

The following data type declaration introduces a fixed point 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* [30].

$$\begin{aligned} \mathbf{newtype}\ \mathsf{Fix\ f} &= \mathit{In}\ \mathsf{(f\ (Fix\ f))} \\ \mathbf{data}\ \mathsf{ListBase\ a\ b} &= \mathit{NilL} \mid \mathit{ConsL}\ \mathsf{a\ b} \end{aligned}$$

The kinds of these type constructors are $\mathsf{Fix} :: (\star \to \star) \to \star$ and $\mathsf{ListBase} :: \star \to (\star \to \star)$. Using Fix and $\mathsf{ListBase}$ the data type of parametric lists can alternatively be defined by

$$\mathbf{type}\ \mathsf{List\ a} = \mathsf{Fix\ (ListBase\ a)}.$$

Here is the list of the first six prime numbers written as an element of type $\mathsf{Fix\ (ListBase\ Int)}$:

$$\begin{aligned} &\mathit{In}\ (\mathit{ConsL}\ 2\ (\mathit{In}\ (\mathit{ConsL}\ 3\ (\mathit{In}\ (\mathit{ConsL}\ 5 \\ &\qquad (\mathit{In}\ (\mathit{ConsL}\ 7\ (\mathit{In}\ (\mathit{ConsL}\ 11\ (\mathit{In}\ (\mathit{ConsL}\ 13\ (\mathit{In}\ \mathit{NilL})))))))\ ))))). \end{aligned}$$

Nested types A *regular* or *uniform* data type is a recursive, parameterized type whose definition does not involve a change of the type parameter(s). The data types of the previous section are without exception regular types. This section is concerned with non-regular or *nested* types [7]. Nested data types are practically important since they can capture data-structural invariants in a way that regular data types cannot. For instance, the following data type declaration defines perfectly balanced, binary leaf trees [20]—perfect trees for short.

$$\mathbf{data}\ \mathsf{Perfect\ a} = \mathit{ZeroP}\ \mathsf{a} \mid \mathit{SuccP}\ \mathsf{(Perfect\ (Fork\ a))}$$

This equation can be seen as a bottom-up definition of perfect trees: a perfect tree is either a singleton tree or a perfect tree that contains pairs of elements. Here is a perfect tree of type Perfect Int:

```
SuccP (SuccP (SuccP (ZeroP (Fork (Fork (Fork 2 3)
                                       (Fork 5 7))
                                 (Fork (Fork 11 13)
                                       (Fork 17 19)) )))).
```

Note that the height of the perfect tree is encoded in the prefix of *SuccP* and *ZeroP* constructors.

The next data type provides an alternative to the ubiquitous list type if an efficient indexing operation is required: Okasaki's *binary random-access lists* [37] support logarithmic access to the elements of a list.

```
data Sequ a = EndS
            | ZeroS (Sequ (Fork a))
            | OneS a (Sequ (Fork a))
```

This definition captures the invariant that binary random-access lists are sequences of perfect trees stored in increasing order of height. Using this representation the sequence of the first six prime numbers reads:

```
ZeroS (OneS (Fork 2 3) (OneS (Fork (Fork 5 7) (Fork 11 13)) EndS)).
```

The types Perfect and Sequ are examples of so-called *linear nests*: the parameters of the recursive calls do not themselves contain occurrences of the defined type. A non-linear nest is the following type taken from [7]:

```
data Bush a = NilB | ConsB a (Bush (Bush a)).
```

An element of type Bush a resembles an ordinary list except that the i-th element has type Bush^i a rather than a. Here is an example element of type Bush Int:

```
ConsB 1 (ConsB (ConsB 2 NilB)
               (ConsB (ConsB (ConsB 3 NilB) NilB) NilB)).
```

Perhaps surprisingly, we will get to know a practical application of this data type in the second part of these notes, which deals with so-called *generalized tries.*

Haskell's **data** construct is surprisingly expressive. In fact, all primitive data types such as characters or integers can, in principle, be defined by a data declaration. The only exceptions to this rule are the function space constructor and Haskell's IO data type. Now, the one-million-dollar question is, of course, how can we define a function that works for all of these data types.

1.3 Towards generic programming

The basic idea of generic programming is to define a function such as equality by *induction on the structure of types.* Thus, generic equality takes three arguments, a type and two values of that type, and proceeds by case analysis on the type argument. In other words, generic equality is a function that depends on a type.

Defining a function by induction on the structure of types sounds like a hard nut to crack. We are trained to define functions by induction on the structure of values. Types are used to guide this process, but we typically think of them as separate entities. So, at first sight, generic programming appears to add an extra level of complication and abstraction to programming. However, we claim that generic programming is in many cases actually simpler than conventional programming. The fundamental reason is that genericity gives you 'a lot of things for free'—we will make this statement more precise in the course of these notes. For the moment, let us support the claim by defining two simple algorithms both in a conventional and in a generic style (data compression and equality). Of course, these are algorithms that make sense for a large class of data types. Consequently, in the conventional style we have to provide an algorithm for each instance of the class.

Towards generic data compression The first problem we look at is to encode elements of a given data type as bit streams implementing a simple form of data compression [25]. For concreteness, we assume that bit streams are given by the following data type (we use Haskell's predefined list data type here):

$$\begin{aligned}&\textbf{type}\ \mathsf{Bin} = [\mathsf{Bit}]\\&\textbf{data}\ \mathsf{Bit}\ = \mathsf{0} \mid \mathsf{1}.\end{aligned}$$

Thus, a bit stream is simply a list of bits. A real implementation might have a more sophisticated representation for Bin but that is a separate matter.

We will implement binary encoders and decoders for three different data types. We consider these types in increasing level of difficulty. The first example type is String. Supposing that *encodeChar* :: Char → Bin is an encoder for characters provided from somewhere, we can encode an element of type String as follows:

$$\begin{aligned}&\mathit{encodeString} && :: \mathsf{String} \to \mathsf{Bin}\\&\mathit{encodeString}\ \mathit{NilS} && = \mathsf{0} : [\,]\\&\mathit{encodeString}\ (\mathit{ConsS}\ c\ s) && = \mathsf{1} : \mathit{encodeChar}\ c +\!\!+\ \mathit{encodeString}\ s.\end{aligned}$$

We emit one bit to distinguish between the two constructors *NilS* and *ConsS*. If the argument is a non-empty string of the form *ConsS c s*, we (recursively) encode the components *c* and *s* and finally concatenate the resulting bit streams.

Given this scheme it is relatively simple to decode a bit stream produced by *encodeString*. Again, we assume that a decoder for characters is provided externally.

$$\begin{aligned}&\mathit{decodesString} && :: \mathsf{Bin} \to (\mathsf{String}, \mathsf{Bin})\\&\mathit{decodesString}\ [\,] && = \mathit{error}\ \texttt{"decodesString"}\end{aligned}$$

$$\begin{array}{l}
\mathit{decodesString}\ (\mathsf{0} : \mathit{bin}) = (\mathit{NilS}, \mathit{bin}) \\
\mathit{decodesString}\ (\mathsf{1} : \mathit{bin}) = \mathbf{let}\ (c, \mathit{bin}_1) = \mathit{decodesChar}\ \mathit{bin} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (s, \mathit{bin}_2) = \mathit{decodesString}\ \mathit{bin}_1 \\
\qquad\qquad\qquad\qquad\qquad\quad\ \mathbf{in}\ (\mathit{ConsS}\ c\ s, \mathit{bin}_2)
\end{array}$$

The decoder has type $\mathsf{Bin} \to (\mathsf{String}, \mathsf{Bin})$ rather than $\mathsf{Bin} \to \mathsf{String}$ to be able to compose decoders in a modular fashion: $\mathit{decodesChar} :: \mathsf{Bin} \to (\mathsf{Char}, \mathsf{Bin})$, for instance, consumes an initial part of the input bit stream and returns the decoded character together with the rest of the input stream. Here are some applications (we assume that characters are encoded in 8 bits).

$$\begin{array}{l}
\mathit{encodeString}\ (\mathit{ConsS}\ \texttt{'L'}\ (\mathit{ConsS}\ \texttt{'i'}\ (\mathit{ConsS}\ \texttt{'s'}\ (\mathit{ConsS}\ \texttt{'a'}\ \mathit{NilS})))) \\
\Longrightarrow \mathsf{1001100101100101101110011101100001100} \\
\mathit{decodesChar}\ (\mathit{tail}\ \mathsf{1001100101100101101110011101100001100}) \\
\Longrightarrow (\texttt{'L'}, \mathsf{1100101101110011101100001100}) \\
\mathit{decodesString}\ \mathsf{1001100101100101101110011101100001100} \\
\Longrightarrow (\mathit{ConsS}\ \texttt{'L'}\ (\mathit{ConsS}\ \texttt{'i'}\ (\mathit{ConsS}\ \texttt{'s'}\ (\mathit{ConsS}\ \texttt{'a'}\ \mathit{NilS}))), [\,])
\end{array}$$

Note that a string of length n is encoded using $n + 1 + 8 * n$ bits.

A string is a list of characters. We have seen that we obtain Haskell's list type by abstracting over the type of list elements. How can we encode a list of something? We could insist that the elements of the input list have already been encoded as bit streams. Then *encodeListBin* completes the task:

$$\begin{array}{ll}
\mathit{encodeListBin} & :: \mathsf{List}\ \mathsf{Bin} \to \mathsf{Bin} \\
\mathit{encodeListBin}\ \mathit{Nil} & = \mathsf{0} : [\,] \\
\mathit{encodeListBin}\ (\mathit{Cons}\ \mathit{bin}\ \mathit{bins}) & = \mathsf{1} : \mathit{bin} +\!\!+\ \mathit{encodeListBin}\ \mathit{bins}.
\end{array}$$

For encoding the elements of a list the following function proves to be useful:

$$\begin{array}{ll}
\mathit{mapList} & :: \forall \mathsf{a}_1\ \mathsf{a}_2\,.\,(\mathsf{a}_1 \to \mathsf{a}_2) \to (\mathsf{List}\ \mathsf{a}_1 \to \mathsf{List}\ \mathsf{a}_2) \\
\mathit{mapList}\ \mathit{mapa}\ \mathit{Nil} & = \mathit{Nil} \\
\mathit{mapList}\ \mathit{mapa}\ (\mathit{Cons}\ a\ \mathit{as}) & = \mathit{Cons}\ (\mathit{mapa}\ a)\ (\mathit{mapList}\ \mathit{mapa}\ \mathit{as}).
\end{array}$$

The function *mapList* is a so-called mapping function that applies a given function to each element of a given list (we will say a lot more about mapping functions in these notes). Combining *encodeListBin* and *mapList* we can encode a variety of lists:

$$\begin{array}{l}
\mathit{encodeListBin}\ (\mathit{mapList}\ \mathit{encodeChar} \\
\qquad (\mathit{Cons}\ \texttt{'A'}\ (\mathit{Cons}\ \texttt{'n'}\ (\mathit{Cons}\ \texttt{'j'}\ (\mathit{Cons}\ \texttt{'a'}\ \mathit{Nil}))))\,) \\
\Longrightarrow \mathsf{1100000101011101101010101101100001100} \\
\mathit{encodeListBin}\ (\mathit{mapList}\ \mathit{encodeInt}\ (\mathit{Cons}\ 47\ (\mathit{Cons}\ 11\ \mathit{Nil}))) \\
\Longrightarrow \mathsf{11111010000000000111010000000000000} \\
(\mathit{encodeListBin} \cdot \mathit{mapList}\ (\mathit{encodeListBin} \cdot \mathit{mapList}\ \mathit{encodeBool})) \\
\qquad (\mathit{Cons}\ (\mathit{Cons}\ \mathit{True}\ (\mathit{Cons}\ \mathit{False}\ (\mathit{Cons}\ \mathit{True}\ \mathit{Nil}))) \\
\qquad (\mathit{Cons}\ (\mathit{Cons}\ \mathit{False}\ (\mathit{Cons}\ \mathit{True}\ (\mathit{Cons}\ \mathit{False}\ \mathit{Nil}))) \\
\qquad (\mathit{Nil}))) \\
\Longrightarrow \mathsf{11110110110111000}.
\end{array}$$

Here, *encodeInt* and *encodeBool* are primitive encoders for integers and Boolean values respectively (an integer occupies 16 bits whereas a Boolean value makes do with one bit).

How do we decode the bit streams thus produced? The first bit tells whether the original list was empty or not, but then we are stuck: we simply do not know how many bits were spent on the first list element. The only way out of this dilemma is to use a decoder function, supplied as an additional argument, that decodes the elements of the original list.

$$
\begin{array}{lll}
\mathit{decodesList} & :: & \forall \mathsf{a} \,.\, (\mathsf{Bin} \to (\mathsf{a}, \mathsf{Bin})) \to (\mathsf{Bin} \to (\mathsf{List}\ \mathsf{a}, \mathsf{Bin})) \\
\mathit{decodesList}\ \mathit{dea}\ [\,] & = & \mathit{error}\ \texttt{"decodesList"} \\
\mathit{decodesList}\ \mathit{dea}\ (\mathsf{0} : \mathit{bin}) & = & (\mathit{Nil}, \mathit{bin}) \\
\mathit{decodesList}\ \mathit{dea}\ (\mathsf{1} : \mathit{bin}) & = & \mathbf{let}\ (a, \mathit{bin}_1) = \mathit{dea}\ \mathit{bin} \\
 & & \qquad (\mathit{as}, \mathit{bin}_2) = \mathit{decodesList}\ \mathit{dea}\ \mathit{bin}_1 \\
 & & \mathbf{in}\ (\mathit{Cons}\ a\ \mathit{as}, \mathit{bin}_2)
\end{array}
$$

This definition generalizes *decodesString* defined above; we have *decodesString* $\cong$ *decodesList decodesChar* (corresponding to String $\cong$ List Char). In some sense, the abstraction step that led from String to List is repeated here on the value level. Of course, we can also generalize *encodeString*:

$$
\begin{array}{lll}
\mathit{encodeList} & :: & \forall \mathsf{a} \,.\, (\mathsf{a} \to \mathsf{Bin}) \to (\mathsf{List}\ \mathsf{a} \to \mathsf{Bin}) \\
\mathit{encodeList}\ \mathit{ena}\ \mathit{Nil} & = & \mathsf{0} : [\,] \\
\mathit{encodeList}\ \mathit{ena}\ (\mathit{Cons}\ a\ \mathit{as}) & = & \mathsf{1} : \mathit{ena}\ a +\!\!+\ \mathit{encodeList}\ \mathit{ena}\ \mathit{as}.
\end{array}
$$

It is not hard to see that *encodeList ena* = *encodeListBin* · *mapList ena*. Encoding and decoding lists is now fairly simple:

$$
\begin{array}{l}
\mathit{encodeList}\ \mathit{encodeChar}\ (\mathit{Cons}\ \texttt{'A'}\ (\mathit{Cons}\ \texttt{'n'}\ (\mathit{Cons}\ \texttt{'j'}\ (\mathit{Cons}\ \texttt{'a'}\ \mathit{Nil})))) \\
\Longrightarrow \mathsf{1100000101011101101010101101100001100} \\
\mathit{encodeList}\ \mathit{encodeInt}\ (\mathit{Cons}\ 47\ (\mathit{Cons}\ 11\ \mathit{Nil})) \\
\Longrightarrow \mathsf{11111010000000000111010000000000000} \\
\mathit{encodeList}\ (\mathit{encodeList}\ \mathit{encodeBool}) \\
\qquad (\mathit{Cons}\ (\mathit{Cons}\ \mathit{True}\ (\mathit{Cons}\ \mathit{False}\ (\mathit{Cons}\ \mathit{True}\ \mathit{Nil}))) \\
\qquad (\mathit{Cons}\ (\mathit{Cons}\ \mathit{False}\ (\mathit{Cons}\ \mathit{True}\ (\mathit{Cons}\ \mathit{False}\ \mathit{Nil}))) \\
\qquad (\mathit{Nil}))) \\
\Longrightarrow \mathsf{11110110110111000}.
\end{array}
$$

The third data type we look at is Okasaki's *binary random-access list.* Using the recursion scheme of *encodeList* we can also program an encoder for binary random-access lists.

$$
\begin{array}{lll}
\mathit{encodeFork} & :: & \forall \mathsf{a} \,.\, (\mathsf{a} \to \mathsf{Bin}) \to (\mathsf{Fork}\ \mathsf{a} \to \mathsf{Bin}) \\
\mathit{encodeFork}\ \mathit{ena}\ (\mathit{Fork}\ a_1\ a_2) & = & \mathit{ena}\ a_1 +\!\!+\ \mathit{ena}\ a_2 \\
\mathit{encodeSequ} & :: & \forall \mathsf{a} \,.\, (\mathsf{a} \to \mathsf{Bin}) \to (\mathsf{Sequ}\ \mathsf{a} \to \mathsf{Bin}) \\
\mathit{encodeSequ}\ \mathit{ena}\ \mathit{EndS} & = & \mathsf{0} : [\,] \\
\mathit{encodeSequ}\ \mathit{ena}\ (\mathit{ZeroS}\ s) & = & \mathsf{1} : \mathsf{0} : \mathit{encodeSequ}\ (\mathit{encodeFork}\ \mathit{ena})\ s \\
\mathit{encodeSequ}\ \mathit{ena}\ (\mathit{OneS}\ a\ s) & = & \mathsf{1} : \mathsf{1} : \mathit{ena}\ a +\!\!+\ \mathit{encodeSequ}\ (\mathit{encodeFork}\ \mathit{ena})\ s
\end{array}
$$

Consider the last equation which deals with arguments of the form *OneS a s*. We emit two bits for the constructor and then (recursively) encode its components. Since *a* has type a, we apply *ena*. Similarly, since *s* has type Sequ (Fork a), we call *encodeSequ* (*encodeFork ena*). It is not hard to see that the type of the component determines the function calls in a straightforward manner. As an aside, note that *encodeSequ* requires a non-schematic form of recursion known as *polymorphic recursion* [36]. The two recursive calls are at type (Fork a → Bin) → (Sequ (Fork a) → Bin) which is a substitution instance of the declared type. Functions operating on nested types are in general polymorphically recursive. Haskell 98 allows polymorphic recursion only if an explicit type signature is provided for the function. The rationale behind this restriction is that type inference in the presence of polymorphic recursion is undecidable [17].

Though the Sequ data type is more complex than the list data type, encoding binary random-access lists is not any more difficult.

```
encodeSequ encodeChar (ZeroS (ZeroS (OneS
    (Fork (Fork 'L' 'i') (Fork 's' 'a')) EndS)))
⟹ 101011001100101001011011001110100001100
encodeSequ encodeInt (ZeroS (OneS (Fork 47 11) EndS))
⟹ 1011111101000000000011010000000000000
```

In general, a string of length n makes do with $2 * \lceil lg\ (n+1) \rceil + 1 + 8 * n$ bits. Perhaps surprisingly, encoding a binary random-access list requires fewer bits than encoding the corresponding list (if the list contains more than 8 elements).

To complete the picture here is the decoder for binary random-access lists.

```
decodesFork             :: ∀a . (Bin → (a, Bin)) → (Bin → (Fork a, Bin))
decodesFork dea bin     = let (a₁, bin₁) = dea bin
                              (a₂, bin₂) = dea bin₁
                          in (Fork a₁ a₂, bin₂)
decodesSequ             :: ∀a . (Bin → (a, Bin)) → (Bin → (Sequ a, Bin))
decodesSequ dea [ ]     = error "decodes"
decodesSequ dea (0 : bin) = (EndS, bin)
decodesSequ dea (1 : 0 : bin)
                        = let (s, bin′) = decodesSequ (decodesFork dea) bin
                          in (ZeroS s, bin′)
decodesSequ dea (1 : 1 : bin)
                        = let (a, bin₁) = dea bin
                              (s, bin₂) = decodesSequ (decodesFork dea) bin₁
                          in (OneS a s, bin₂)
```

Towards generic equality As a second example, let us work towards implementing a generic version of equality. Taking a look at several ad-hoc instances of equality will improve our understanding when we consider the generic programming extensions Generic Haskell offers.

Let us start simple: here is equality of strings.

$$\begin{aligned}
&\mathit{eqString} && :: \mathsf{String} \to \mathsf{String} \to \mathsf{Bool}\\
&\mathit{eqString}\ \mathit{NilS}\ \mathit{NilS} && = \mathit{True}\\
&\mathit{eqString}\ \mathit{NilS}\ (\mathit{ConsS}\ c'\ s') && = \mathit{False}\\
&\mathit{eqString}\ (\mathit{ConsS}\ c\ s)\ \mathit{NilS} && = \mathit{False}\\
&\mathit{eqString}\ (\mathit{ConsS}\ c\ s)\ (\mathit{ConsS}\ c'\ s') && = \mathit{eqChar}\ c\ c' \wedge \mathit{eqString}\ s\ s'
\end{aligned}$$

The function $\mathit{eqChar} :: \mathsf{Char} \to \mathsf{Char} \to \mathsf{Bool}$ is equality of characters. As usual, we assume that this function is predefined.

The type List is obtained from String by abstracting over Char. Likewise, *eqList* is obtained from *eqString* by abstracting over *eqChar*.

$$\begin{aligned}
&\mathit{eqList} && :: \forall \mathsf{a}\,.\,(\mathsf{a} \to \mathsf{a} \to \mathsf{Bool})\\
& && \qquad \to (\mathsf{List\ a} \to \mathsf{List\ a} \to \mathsf{Bool})\\
&\mathit{eqList}\ \mathit{eqa}\ \mathit{Nil}\ \mathit{Nil} && = \mathit{True}\\
&\mathit{eqList}\ \mathit{eqa}\ \mathit{Nil}\ (\mathit{Cons}\ a'\ x') && = \mathit{False}\\
&\mathit{eqList}\ \mathit{eqa}\ (\mathit{Cons}\ a\ x)\ \mathit{Nil} && = \mathit{False}\\
&\mathit{eqList}\ \mathit{eqa}\ (\mathit{Cons}\ a\ x)\ (\mathit{Cons}\ a'\ x') && = \mathit{eqa}\ a\ a' \wedge \mathit{eqList}\ \mathit{eqa}\ x\ x'
\end{aligned}$$

Similarly, the type GRose of generalized rose trees is obtained from Rose by abstracting over the list type constructor (which is of kind $\star \to \star$). Likewise, *eqGRose* abstracts over list equality (which has a polymorphic type). Thus, *eqGRose* takes a polymorphic function to a polymorphic function.

$$\begin{aligned}
&\mathit{eqGRose}\ ::\ \forall \mathsf{f}\,.\,(\forall \mathsf{a}\,.\,(\mathsf{a} \to \mathsf{a} \to \mathsf{Bool}) \to (\mathsf{f\ a} \to \mathsf{f\ a} \to \mathsf{Bool}))\\
&\qquad\qquad \to (\forall \mathsf{a}\,.\,(\mathsf{a} \to \mathsf{a} \to \mathsf{Bool})\\
&\qquad\qquad\qquad \to (\mathsf{GRose\ f\ a} \to \mathsf{GRose\ f\ a} \to \mathsf{Bool}))\\
&\mathit{eqGRose}\ \mathit{eqf}\ \mathit{eqa}\ (\mathit{GBranch}\ a\ f)\ (\mathit{GBranch}\ a'\ f')\\
&\ = \mathit{eqa}\ a\ a' \wedge \mathit{eqf}\ (\mathit{eqGRose}\ \mathit{eqf}\ \mathit{eqa})\ f\ f'
\end{aligned}$$

The function *eqGRose* has a so-called rank-2 type. In general, the rank of a type is given by

$$\begin{aligned}
&\mathit{rank}(\mathsf{C}) && = 0\\
&\mathit{rank}(\forall \mathsf{a}\,.\,\mathsf{t}) && = \mathit{max}\{1, \mathit{rank}(\mathsf{t})\}\\
&\mathit{rank}(\mathsf{t} \to \mathsf{u}) && = \mathit{max}\{\mathit{inc}\ (\mathit{rank}(\mathsf{t})), \mathit{rank}(\mathsf{u})\},
\end{aligned}$$

where $\mathit{inc}\ 0 = 0$ and $\mathit{inc}\ (n{+}1) = n{+}2$. Most implementations of Haskell support rank-2 types. The latest version of the Glasgow Haskell Compiler, GHC 5.04, even supports general rank-n types.

As a final example, consider defining equality for the fixed point operator on the type level.

$$\begin{aligned}
&\mathit{eqFix}\ ::\ \forall \mathsf{f}\,.\,(\forall \mathsf{a}\,.\,(\mathsf{a} \to \mathsf{a} \to \mathsf{Bool}) \to (\mathsf{f\ a} \to \mathsf{f\ a} \to \mathsf{Bool}))\\
&\qquad\qquad \to (\mathsf{Fix\ f} \to \mathsf{Fix\ f} \to \mathsf{Bool})\\
&\mathit{eqFix}\ \mathit{eqf}\ (\mathit{In}\ f)\ (\mathit{In}\ f') = \mathit{eqf}\ (\mathit{eqFix}\ \mathit{eqf})\ f\ f'
\end{aligned}$$

1.4 Towards Generic Haskell

In the previous section we have seen a bunch of ad-hoc instances of generic functions. Looking at the type signatures of equality we see that the type of eqT depends on the kind of T. Roughly speaking, the more complicated the kind of T, the more complicated the type of eqT. To capture the type of generic functions, Generic Haskell supports the definition of types that are defined by induction over the structure of kinds, so-called *kind-indexed types*.

Apart from the typings, it is crystal clear what the definition of eqT looks like. We first have to check whether the two arguments of equality are labelled by the same constructor. If this is the case, then their arguments are recursively tested for equality. Nonetheless, coding the equality function is boring and consequently error-prone. Fortunately, Generic Haskell allows us to capture equality once and for all.

To define generic equality and other generic functions it suffices to cover three simple, non-recursive data types: binary sums, binary products and nullary products (that is, the unit data type). Since these types are the building blocks of data declarations, Generic Haskell is then able to generate instances of equality for arbitrary user-defined types. In other words, the generic equality function works for all types of all kinds. Of course, if the user-defined type falls back on some primitive type, then the generic equality function must also supply code for this type. Thus, generic equality will include cases for Char, Int, etc. On the other hand, it will not include cases for function types or for the IO type since we cannot decide equality of functions or IO actions.

We have already mentioned the slogan that generic programming gives the programmer a lot of things for free. In our case, Generic Haskell automatically takes care of type abstraction, type application and type recursion. And it does so in a type-safe manner.

Kind-indexed types The type of a generic function is captured by a kind-indexed type which is defined by induction on the structure of kinds. Here are some examples.

```
type Encode{[⋆]} t      = t → Bin
type Encode{[κ → ν]} t  = ∀a . Encode{[κ]} a → Encode{[ν]} (t a)
type Decodes{[⋆]} t     = Bin → (t, Bin)
type Decodes{[κ → ν]} t = ∀a . Decodes{[κ]} a → Decodes{[ν]} (t a)
type Eq{[⋆]} t          = t → t → Bool
type Eq{[κ → ν]} t      = ∀a . Eq{[κ]} a → Eq{[ν]} (t a)
```

The part enclosed in {[·]} is the kind index. In each case, the equation for kind $\star$ is the interesting one. For instance, $\mathsf{t} \to \mathsf{t} \to \mathsf{Bool}$ is the type of equality for manifest types (nullary type constructors). Perhaps surprisingly, the equations for function kinds always follow the same scheme, which we will encounter time and again. We will see in Section 3.3 that this scheme is inevitable because of the way type constructors of kind $\kappa \to \nu$ are specialized.

The type signatures we have seen in the previous section can be written more succinctly using the kind-indexed types above.

$$
\begin{array}{ll}
\mathit{encodeString} & :: \mathsf{Encode}\{\![\star]\!\}\ \mathsf{String} \\
\mathit{encodeList} & :: \mathsf{Encode}\{\![\star \to \star]\!\}\ \mathsf{List} \\
\mathit{decodesString} & :: \mathsf{Decodes}\{\![\star]\!\}\ \mathsf{String} \\
\mathit{decodesList} & :: \mathsf{Decodes}\{\![\star \to \star]\!\}\ \mathsf{List} \\
\mathit{eqString} & :: \mathsf{Eq}\{\![\star]\!\}\ \mathsf{String} \\
\mathit{eqList} & :: \mathsf{Eq}\{\![\star \to \star]\!\}\ \mathsf{List} \\
\mathit{eqGRose} & :: \mathsf{Eq}\{\![(\star \to \star) \to (\star \to \star)]\!\}\ \mathsf{GRose} \\
\mathit{eqFix} & :: \mathsf{Eq}\{\![(\star \to \star) \to \star]\!\}\ \mathsf{Fix}
\end{array}
$$

In general, the equality function for type t of kind κ has type $\mathsf{Eq}\{\![\kappa]\!\}\ \mathsf{t}$.

Sums and products Recall that a Haskell data type is essentially a sum of products. To cover data types the generic programmer only has to define the generic function for binary sums and binary products (and nullary products). To this end Generic Haskell provides the following data types.

$$
\begin{array}{l}
\mathbf{data}\ \mathsf{Unit} \quad = \mathit{Unit} \\
\mathbf{data}\ \mathsf{a}\ {:}{*}{:}\ \mathsf{b} = \mathsf{a}\ {:}{*}{:}\ \mathsf{b} \\
\mathbf{data}\ \mathsf{a}\ {:}{+}{:}\ \mathsf{b} = \mathit{Inl}\ \mathsf{a} \mid \mathit{Inr}\ \mathsf{b}
\end{array}
$$

Note that the operator ':*:' is used both as a type constructor and as a data constructor (pairing).

In a sense, the generic programmer views the data declaration

$$\mathbf{data}\ \mathsf{List}\ \mathsf{a} = \mathit{Nil} \mid \mathit{Cons}\ \mathsf{a}\ (\mathsf{List}\ \mathsf{a})$$

as if it were given by the following type definition

$$\mathbf{type}\ \mathsf{List}\ \mathsf{a} = \mathsf{Unit}\ {:}{+}{:}\ \mathsf{a}\ {:}{*}{:}\ \mathsf{List}\ \mathsf{a},$$

which makes sums and products explicit (':*:' binds more tightly than ':+:').

The types Unit, ':*:', and ':+:' are isomorphic to the predefined Haskell types '()', '(,)', and *Either*. The main reason for introducing new types is that it gives the user the ability to provide special instances for '()', '(,)', and *Either*. As an example, you may want to show elements of the pair data type in a special way (by contrast, we have seen above that ':*:' is used to represent the arguments of a constructor).

Type-indexed values Given these prerequisites, the definition of generic functions is within reach. The generic programmer has to provide a type signature, which typically involves a kind-indexed type, and a set of equations, one for each type constant, where a type constant is either a primitive type like Char, Int, ‘→’ etc or one of the three types Unit, ‘:∗:’, and ‘:+:’. As an example here is the definition of the generic encoding function.

```
encode{|t :: κ|}                   :: Encode{[κ]} t
encode{|Char|}                     = encodeChar
encode{|Int|}                      = encodeInt
encode{|Unit|} Unit                = [ ]
encode{|:+:|} ena enb (Inl a)      = 0 : ena a
encode{|:+:|} ena enb (Inr b)      = 1 : enb b
encode{|:∗:|} ena enb (a :∗: b)    = ena a ++ enb b
```

Generic functions are also called *type-indexed values*; the part enclosed in {|·|} is the type index. Note that each equation is more or less inevitable. Characters and integers are encoded using the primitive functions *encodeChar* and *encodeInt*. To encode the single element of the unit type no bits are required. To encode an element of a sum 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. Since the types ‘:+:’ and ‘:∗:’ have kind $\star \to \star \to \star$, the generic instances take two additional arguments, *ena* and *enb*.

The definition of *decode* follows the same definitional pattern.

```
decodes{|t :: κ|}                  :: Decodes{[κ]} t
decodes{|Char|}                    = decodesChar
decodes{|Int|}                     = decodesInt
decodes{|Unit|} bin                = (Unit, bin)
decodes{|:+:|} dea deb [ ]         = error "decodes"
decodes{|:+:|} dea deb (0 : bin)   = let (a, bin′) = dea bin in (Inl a, bin′)
decodes{|:+:|} dea deb (1 : bin)   = let (b, bin′) = deb bin in (Inr b, bin′)
decodes{|:∗:|} dea deb bin         = let (a, bin₁) = dea bin
                                         (b, bin₂) = deb bin₁
                                     in ((a :∗: b), bin₂)
```

Generic equality is equally straightforward.

```
eq{|t :: κ|}                           :: Eq{[κ]} t
eq{|Char|}                             = eqChar
eq{|Int|}                              = eqInt
eq{|Unit|} Unit Unit                   = True
eq{|:+:|} eqa eqb (Inl a) (Inl a′)     = eqa a a′
eq{|:+:|} eqa eqb (Inl a) (Inr b′)     = False
eq{|:+:|} eqa eqb (Inr b) (Inl a′)     = False
eq{|:+:|} eqa eqb (Inr b) (Inr b′)     = eqb b b′
eq{|:∗:|} eqa eqb (a :∗: b) (a′ :∗: b′) = eqa a a′ ∧ eqb b b′
```

Generic application Given the definitions above we can encode and decode elements of arbitrary user-defined data types. The generic functions are invoked by instantiating the type-index to a specific type, where the type can be any *closed* type expression.

```
encode{|String|} (ConsS 'L' (ConsS 'i' (ConsS 's' (ConsS 'a' NilS))))
⟹ 1001100101100101101110011101100001100
decodes{|String|} 1001100101100101101110011101100001100
⟹ (ConsS 'L' (ConsS 'i' (ConsS 's' (ConsS 'a' NilS))), [])
encode{|List Char|} (Cons 'A' (Cons 'n' (Cons 'j' (Cons 'a' Nil))))
⟹ 1100000101011101101010101101100001100
encode{|List Int|} (Cons 47 (Cons 11 Nil))
⟹ 11111010000000000011101000000000000000
encode{|List (List Bool)|}
    (Cons (Cons True (Cons False (Cons True Nil)))
    (Cons (Cons False (Cons True (Cons False Nil)))
    (Nil)))
⟹ 11110110110111000.
```

In the examples above we call *encode* and *decodes* always for types of kind $\star$. However, the generic functions are far more flexible: we can call them at any type of any kind. The following session illustrates a more general use (here with generic equality).

```
eq{|List Char|} "hello" "Hello"
⟹ False
let sim c c' = eqChar (toUpper c) (toUpper c')
eq{|List|} sim "hello" "Hello"
⟹ True
```

If we instantiate the type index to a type constructor, then we have to pass an 'equality function' for the type argument as an additional parameter. Of course, we can pass any function as long as it meets the typing requirements. In the session above, we pass 'an equality test' that ignores case distinctions. Quite clearly, this gives us an extra degree of flexibility.

Generic abstraction Abstraction is at the heart of programming. Generic Haskell also supports a simple form of type abstraction. Common usages of generic functions can be captured using generic abstractions.

```
similar{|t :: ⋆ → ⋆|} :: t Char → t Char → Bool
similar{|t|}          = eq{|t|} sim
```

Note that *similar* is only applicable to type constructors of kind $\star \to \star$.

1.5 Stocktaking

A generic program is one that the programmer writes once, but which works over many different data types. Broadly speaking, generic programming aims at relieving the programmer from repeatedly writing functions of similar functionality for different user-defined data types. Examples of generic functions include equality, parsing and pretty printing, serialising, ordering, hashing, and so on. A generic function such as a pretty printer or a parser is written once and for all; its specialization to different instances of data types 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.

The basic idea of generic programming is to define a function such as equality by *induction on the structure of types*. Thus, generic equality takes three arguments, a type and two values of that type, and proceeds by case analysis on the type argument. In other words, generic equality is a function that depends on a type. Consider the structure of the language Haskell. If we ignore the module system, Haskell has the three level structure depicted on the right. The lowest level, that is, the level where computations take place, consists of *values*. The second level, which imposes structure on the value level, is inhabited by *types*. Finally, on the third level, which imposes structure on the type level, we have so-called *kinds*. Why is there a third level? We have seen that Haskell allows the programmer to define parametric types such as the popular data type of lists. The list type constructor can be seen as a function on types and the kind system allows to specify this in a precise way. Thus, a kind is simply the 'type' of a type constructor.

kinds

---

types

---

values

In ordinary programming we routinely define values that depend on values, that is, functions and types that depend on types, that is, type constructors. However, we can also imagine to have dependencies between adjacent levels. For instance, a type might depend on a value or a type might depend on a kind. The following table lists the possible combinations:

kinds depending on kinds	parametric and kind-indexed kinds
kinds depending on types	dependent kinds
types depending on kinds	polymorphic and kind-indexed types
types depending on types	parametric and type-indexed types
types depending on values	dependent types
values depending on types	polymorphic and type-indexed functions
values depending on values	ordinary functions

.

If a higher level depends on a lower level we have so-called dependent types or dependent kinds. Programming languages with dependent types are the subject of intensive research, see, for instance, [4]. Dependent types will play little rôle in these notes as generic programming is concerned with the opposite direction, where a lower level depends on the same or a higher level. However, using dependent types we can simulate generic programming, see Section 1.6. If a value depends on a type we either have a *polymorphic* or a *type-indexed* function. In both cases the function takes a type as an argument. What is the difference between the two? Now, a polymorphic function stands for an algorithm that happens to be insensitive to what type the values in some structure are. Take, for example, the *length* function that calculates the length of a list. Since it need not inspect the elements of a given list, it has type $\forall \mathsf{a} . \mathsf{List\ a} \to \mathsf{Int}$. By contrast, a type-indexed function is defined by induction on the structure of its type argument. In some sense, the type argument guides the computation which is performed on the value arguments.

A similar distinction applies to the type and to the kind level: a parametric type does not inspect its type argument whereas a type-indexed type is defined by induction on the structure of its type argument and similarly for kinds. The following table summarizes the interesting cases.

kinds	defined by induction on the structure of kinds	kind-indexed kinds
types	defined by induction on the structure of kinds	kind-indexed types
types	defined by induction on the structure of types	type-indexed types
values	defined by induction on the structure of types	type-indexed values

Amazingly, we will encounter examples of all sorts of parameterization in the lecture notes (type-indexed types and kind-indexed kinds are covered in the second part).

1.6 Related work

This section puts Generic Haskell into a broader perspective discussing its limitations, possible generalizations and variations and alternative approaches to generic programming. To illustrate the underlying ideas we will use generic equality as a running example.

Generic Haskell We have noted in the beginning of Section 1.1 that it is not possible to define a *polymorphic* equality function that works uniformly for all types.

$$eq :: \forall \mathsf{a} . \mathsf{a} \to \mathsf{a} \to \mathsf{Bool} \qquad \text{-- does not work}$$

Consequently, Generic Haskell treats *eq* as a family of functions indexed by type. Deviating from Generic Haskell's syntax *eq*'s type could be written as follows.

$$eq :: \{\!|\mathsf{a} :: \star|\!\} \to \mathsf{a} \to \mathsf{a} \to \mathsf{Bool}$$

A moment's reflection reveals that this is really a *dependent type*: the second and the third argument depend on the first argument, which is a type. Of course, since equality may be indexed by types of arbitrary kinds *eq*'s type signature is slightly more complicated.

$$eq :: \forall \kappa . \{\!|\mathsf{a} :: \kappa|\!\} \to \mathsf{Eq}\{\![\kappa]\!\}\ \mathsf{a}$$

The universal quantifier, which ranges over kinds, makes explicit that *eq* works for all kinds.

Though the construct $\{\!|\mathsf{a} :: \kappa|\!\} \rightarrow \mathsf{t}$ resembles a dependent type, type-indexed functions are not first-class citizens in Generic Haskell. For example, we cannot define a higher-order generic function that takes type-indexed functions to type-indexed functions. The reason for this restriction is that Generic Haskell implements genericity by translating a type-indexed function into a family of (higher-order) polymorphic functions, see Section 3.3.

Type classes Haskell's major innovation is its support for *overloading*, based on type classes. For example, the Haskell Prelude defines the class Eq (slightly simplified):

$$\begin{array}{l}\textbf{class}\ \mathsf{Eq\ a}\ \textbf{where}\\ \quad eq :: \mathsf{a} \rightarrow \mathsf{a} \rightarrow \mathsf{Bool}\end{array}$$

This *class declaration* defines an overloaded top-level function, called *method*, whose type is

$$eq :: \forall \mathsf{a}\,.\,(\mathsf{Eq\ a}) \Rightarrow \mathsf{a} \rightarrow \mathsf{a} \rightarrow \mathsf{Bool}.$$

Before we can use *eq* on values of, say Int, we must explain how to take equality over Int values:

$$\begin{array}{l}\textbf{instance}\ \mathsf{Eq\ Int}\ \textbf{where}\\ \quad eq = eqInt.\end{array}$$

This *instance declaration* makes Int an element of the type class Eq and says 'the *eq* function at type Int is implemented by *eqInt*'. As a second example consider equality of lists. Two lists are equal if they have the same length and corresponding elements are equal. Hence, we require equality over the element type:

$$\begin{array}{ll}\multicolumn{2}{l}{\textbf{instance}\ (\mathsf{Eq}\ a) \Rightarrow \mathsf{Eq}\ (\mathsf{List}\ a)\ \textbf{where}}\\ \quad eq\ Nil\ Nil & = True\\ \quad eq\ Nil\ (Cons\ a_2\ as_2) & = False\\ \quad eq\ (Cons\ a_1\ as_1)\ Nil & = False\\ \quad eq\ (Cons\ a_1\ as_1)\ (Cons\ a_2\ as_2) & = eq\ a_1\ a_2 \wedge eq\ as_1\ as_2.\end{array}$$

This instance declaration says 'if a is an instance of Eq, then List a is an instance of Eq, as well'.

Though type classes bear a strong resemblance to generic definitions, they do not support generic programming. A type class declaration corresponds roughly to the type signature of a generic definition—or rather, to a collection of type signatures. Instance declarations are related to the type cases of a generic definition. The crucial difference is that a generic definition works for all types, whereas instance declarations must be provided explicitly by the programmer for each newly defined data type. There is, however, one exception to this rule. For a handful of built-in classes Haskell provides special support, the so-called '**deriving**' mechanism. For instance, if you define

$$\textbf{data}\ \mathsf{List\ a} = Nil \mid Cons\ \mathsf{a}\ (\mathsf{List\ a})\ \textbf{deriving}\ (\mathsf{Eq})$$

then Haskell generates the 'obvious' code for equality. What 'obvious' means is specified informally in an Appendix of the language definition [38]. Of course, the idea suggests itself to use generic definitions for specifying default methods so that the programmer can define her own derivable classes. This idea is pursued further in [23, 1].

Haskell translates type classes and instance declarations into a family of polymorphic functions using the so-called *dictionary passing translation*, which is quite similar to the implementation technique of Generic Haskell.

Intensional type analysis The framework of intensional type analysis [16] was originally developed as a means to improve the implementation of polymorphic functions. It was heavily employed in typed intermediate languages not intended for programmers but for compiler writers. The central idea is to pass types or representation of types at run time, which can be analysed and dispatched upon. Thus, equality has the simple type

$$eq :: \forall \mathsf{a} \,.\, \mathsf{a} \to \mathsf{a} \to \mathsf{Bool} \qquad \text{-- non-parametric } \forall.$$

As in Generic Haskell equality takes an additional type argument and does a case analysis on this argument (using a **typecase**). The resulting code looks quite similar to our definition of equality. The major difference is that the type argument is interpreted at run time whereas Generic Haskell does the analysis at compile time. On the other hand, type-indexed functions are second-class citizens in Generic Haskell whereas in intensional type analysis they have first-class status. Originally, the framework was restricted to data types of kind $\star$, but recent work [49] has lifted this restriction (the generalization is, in fact, inspired by our work on Generic Haskell).

Type representations Typed intermediate languages based on intensional type analysis are expressive but rather complex languages. Perhaps surprisingly, dynamic type dispatch can be simulated in a much more modest setting: we basically require a Hindley-Milner type system augmented with existential types. The central idea is to pass *type representations* instead of types. As a first try, if Rep is the type of type representations, we could assign 'eq' the type $\forall \mathsf{a} \,.\, \mathsf{Rep} \to \mathsf{a} \to \mathsf{a} \to \mathsf{Bool}$. This approach, however, does not work. The parametricity theorem [46] implies that a function of this type must necessarily ignore its second and its third argument. The trick is to use a parametric type for type representations:

$$eq :: \forall \mathsf{a} \,.\, \mathsf{Rep}\ \mathsf{a} \to \mathsf{a} \to \mathsf{a} \to \mathsf{Bool}.$$

Here $\mathsf{Rep}\ \mathsf{t}$ is the type representation of t. In [9, 5] it is shown how to define a Rep type in Haskell (augmented with existential types). This approach is, however, restricted to types of one fixed kind.

Dependent types We have noted above that the type Generic Haskell assigns to equality resembles a dependent type. Thus, it comes as little surprise that we can simulate Generic Haskell in a dependently typed language [2]. In such a language we can define a simple, non-parametric type Rep of type representations. The correspondence between a type and its representative is established by a function $\mathit{Type} :: \mathsf{Rep} \to \star$ that maps a representation to its type. The signature of equality is then given by

$$\mathit{eq} :: (\mathsf{a} :: \mathsf{Rep}) \to \mathit{Type}\ \mathsf{a} \to \mathit{Type}\ \mathsf{a} \to \mathsf{Bool}.$$

The code of *eq* is similar to what we have seen before; only the typechecking is more involved as it requires reduction of type expressions.

Historical notes The concept of functional generic programming trades under a variety of names: F. Ruehr refers to this concept as *structural polymorphism* [42, 41], T. Sheard calls generic functions *type parametric* [44], C.B. Jay and J.R.B. Cocket use the term *shape polymorphism* [27], R. Harper and G. Morrisett [16] coined the phrase *intensional polymorphism*, and J. Jeuring invented the word *polytypism* [28].

The mainstream of generic programming is based on the initial algebra semantics of datatypes, see, for instance [15], and puts emphasis on general recursion operators like mapping functions and catamorphisms (folds). In [43] several variations of these operators are informally defined and algorithms are given that specialize these functions for given datatypes. The programming language *Charity* [10] automatically provides mapping functions and catamorphisms for each user-defined datatype. Since general recursion is not available, Charity is strongly normalizing. *Functorial ML* [26] 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* [24], a precursor of Generic Haskell, 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 [19] for a more detailed comparison).

Most approaches are restricted to first-order kinded, regular datatypes (or even subsets of this class). One notable exception is the work of F. Ruehr [42], who presents a higher-order language based on a type system related to ours. Genericity is achieved through the use of type patterns which are interpreted at run-time. By contrast, the implementation technique of Generic Haskell does not require the passing of types or representations of types at run-time.

Exercise 1. Prove that a function of type $\forall \mathsf{a} . \mathsf{a} \to \mathsf{a} \to \mathsf{Bool}$ is necessarily constant.

Exercise 2. Define an *instance* of equality for random-access lists.

Exercise 3. Implement a generic version of Haskell's *compare* function, which determines the precise ordering of two elements. Start by defining an appropriate kind-indexed type and then give equations for each of the type constants Unit, ':+:', and ':∗:'.

Exercise 4. Miranda offers a primitive function *force* $:: \forall \mathsf{a} . \mathsf{a} \rightarrow \mathsf{a}$ that *fully* evaluates its argument. Implement a generic version of *force* using Haskell's *seq* function (which is of type $\forall \mathsf{a}\ \mathsf{b} . \mathsf{a} \rightarrow \mathsf{b} \rightarrow \mathsf{b}$).

Exercise 5 (Difficult). Define a generic function that memoizes a given function. Its kind-indexed type is given by

$$\begin{aligned}&\textbf{type}\ \mathsf{Memo}\{\![\star]\!\}\ \mathsf{t} &&= \forall \mathsf{v} . (\mathsf{t} \rightarrow \mathsf{v}) \rightarrow (\mathsf{t} \rightarrow \mathsf{v})\\ &\textbf{type}\ \mathsf{Memo}\{\![\kappa \rightarrow \nu]\!\}\ \mathsf{t} &&= \forall \mathsf{a} . \mathsf{Memo}\{\![\kappa]\!\}\ \mathsf{a} \rightarrow \mathsf{Memo}\{\![\nu]\!\}\ (\mathsf{t}\ \mathsf{a}).\end{aligned}$$

Note that $\mathsf{Memo}\{\![\star]\!\}\ \mathsf{t}$ is a polymorphic type. *Hint: memo*$\{\!|\mathsf{t} :: \star|\!\}\ f$ should yield a closure that does not dependent on the actual argument of f.

2 Generic Haskell—Practice

In this section we look at generic definitions in more detail explaining the various features of Generic Haskell. In particular, we show how to define mapping functions, reductions, and pretty printers generically.

2.1 Mapping functions

A *mapping function* for a type constructor F of kind $\star \rightarrow \star$ lifts a given function of type $\mathsf{a} \rightarrow \mathsf{b}$ to a function of type $\mathsf{F}\ \mathsf{a} \rightarrow \mathsf{F}\ \mathsf{b}$. In the sequel we show how to define mapping functions so that they work for *all* types of *all* kinds. Before we tackle the generic definition, let us consider some instances first. As an aside, note that the combination of a type constructor and its mapping function is often referred to as a *functor*.

Here is again the all-time favourite, the mapping function for lists.

$$\begin{aligned}&\mathit{mapList} &&:: \forall \mathsf{a}_1\ \mathsf{a}_2 . (\mathsf{a}_1 \rightarrow \mathsf{a}_2) \rightarrow (\mathsf{List}\ \mathsf{a}_1 \rightarrow \mathsf{List}\ \mathsf{a}_2)\\ &\mathit{mapList}\ \mathit{mapa}\ \mathit{Nil} &&= \mathit{Nil}\\ &\mathit{mapList}\ \mathit{mapa}\ (\mathit{Cons}\ a\ as) &&= \mathit{Cons}\ (\mathit{mapa}\ a)\ (\mathit{mapList}\ \mathit{mapa}\ as).\end{aligned}$$

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 *mapa*. 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 *mapList* rigidly follows the structure of the data type.

We have seen in Section 1.2 that List can alternatively be defined using an explicit fixed point construction.

$$\textbf{type}\ \mathsf{List}'\ \mathsf{a} = \mathsf{Fix}\ (\mathsf{ListBase}\ \mathsf{a}).$$

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

$$\begin{aligned}
&\mathit{mapListBase} && :: \forall \mathsf{a}_1\ \mathsf{a}_2 \,.\, (\mathsf{a}_1 \to \mathsf{a}_2) \to \forall \mathsf{b}_1\ \mathsf{b}_2 \,.\, (\mathsf{b}_1 \to \mathsf{b}_2)\\
& && \qquad\qquad \to (\mathsf{ListBase}\ \mathsf{a}_1\ \mathsf{b}_1 \to \mathsf{ListBase}\ \mathsf{a}_2\ \mathsf{b}_2)\\
&\mathit{mapListBase}\ \mathit{mapa}\ \mathit{mapb}\ \mathit{NilL} = \mathit{NilL}\\
&\mathit{mapListBase}\ \mathit{mapa}\ \mathit{mapb}\ (\mathit{ConsL}\ a\ b)\\
& && = \mathit{ConsL}\ (\mathit{mapa}\ a)\ (\mathit{mapb}\ b)
\end{aligned}$$

Since the base functor has two type arguments, its mapping function takes two functions, *mapa* and *mapb*, and applies them to values of type a_1 and b_1, respectively. Even more interesting is the mapping function for Fix

$$\begin{aligned}
&\mathit{mapFix} && :: \forall \mathsf{f}_1\ \mathsf{f}_2 \,.\, (\forall \mathsf{a}_1\ \mathsf{a}_2 \,.\, (\mathsf{a}_1 \to \mathsf{a}_2) \to (\mathsf{f}_1\ \mathsf{a}_1 \to \mathsf{f}_2\ \mathsf{a}_2))\\
& && \qquad\qquad \to (\mathsf{Fix}\ \mathsf{f}_1 \to \mathsf{Fix}\ \mathsf{f}_2)\\
&\mathit{mapFix}\ \mathit{mapf}\ (\mathit{In}\ v) = \mathit{In}\ (\mathit{mapf}\ (\mathit{mapFix}\ \mathit{mapf})\ v),
\end{aligned}$$

which takes a polymorphic function as an argument. In other words, *mapFix* has a rank-2 type. The argument function, *mapf*, has a more general type than one would probably expect: it takes a function of type $\mathsf{a}_1 \to \mathsf{a}_2$ to a function of type $\mathsf{f}_1\ \mathsf{a}_1 \to \mathsf{f}_2\ \mathsf{a}_2$. By contrast, the mapping function for List (which like f has kind $\star \to \star$) takes $\mathsf{a}_1 \to \mathsf{a}_2$ to $\mathsf{List}\ \mathsf{a}_1 \to \mathsf{List}\ \mathsf{a}_2$. The definition below demonstrates that the extra generality is vital.

$$\begin{aligned}
&\mathit{mapList'} && :: \forall \mathsf{a}_1\ \mathsf{a}_2 \,.\, (\mathsf{a}_1 \to \mathsf{a}_2) \to (\mathsf{List'}\ \mathsf{a}_1 \to \mathsf{List'}\ \mathsf{a}_2)\\
&\mathit{mapList'}\ \mathit{mapa} = \mathit{mapFix}\ (\mathit{mapListBase}\ \mathit{mapa})
\end{aligned}$$

The argument of *mapFix* has type $\forall \mathsf{b}_1\ \mathsf{b}_2 \,.\, (\mathsf{b}_1 \to \mathsf{b}_2) \to (\mathsf{ListBase}\ \mathsf{a}_1\ \mathsf{b}_1 \to \mathsf{ListBase}\ \mathsf{a}_2\ \mathsf{b}_2)$, that is, f_1 is instantiated to $\mathsf{ListBase}\ \mathsf{a}_1$ and f_2 to $\mathsf{ListBase}\ \mathsf{a}_2$.

Now, let us define a generic version of *map*. What is the type of the generic mapping function? As a first attempt, we might define

$$\begin{aligned}
&\textbf{type}\ \mathsf{Map}\{[\star]\}\ \mathsf{t} && = \mathsf{t} \to \mathsf{t} \qquad\qquad\qquad\qquad \text{-- WRONG}\\
&\textbf{type}\ \mathsf{Map}\{[\kappa \to \nu]\}\ \mathsf{t} && = \forall \mathsf{a} \,.\, \mathsf{Map}\{[\kappa]\}\ \mathsf{a} \to \mathsf{Map}\{[\nu]\}\ (\mathsf{t}\ \mathsf{a}).
\end{aligned}$$

Alas, we have $\mathsf{Map}\{[\star \to \star]\}\ \mathsf{List} = \forall \mathsf{a} \,.\, (\mathsf{a} \to \mathsf{a}) \to (\mathsf{List}\ \mathsf{a} \to \mathsf{List}\ \mathsf{a})$, which is not general enough. The solution is to use a two-argument version of the kind-indexed type Map.

$$\begin{aligned}
&\textbf{type}\ \mathsf{Map}\{[\star]\}\ \mathsf{t}_1\ \mathsf{t}_2 && = \mathsf{t}_1 \to \mathsf{t}_2\\
&\textbf{type}\ \mathsf{Map}\{[\kappa \to \nu]\}\ \mathsf{t}_1\ \mathsf{t}_2 && = \forall \mathsf{a}_1\ \mathsf{a}_2 \,.\, \mathsf{Map}\{[\kappa]\}\ \mathsf{a}_1\ \mathsf{a}_2 \to \mathsf{Map}\{[\nu]\}\ (\mathsf{t}_1\ \mathsf{a}_1)\ (\mathsf{t}_2\ \mathsf{a}_2)\\
&\mathit{map}\{|\mathsf{t} :: \kappa|\} && :: \mathsf{Map}\{[\kappa]\}\ \mathsf{t}\ \mathsf{t}
\end{aligned}$$

We obtain $\mathsf{Map}\{[\star \to \star]\}\ \mathsf{List}\ \mathsf{List} = \forall \mathsf{a}_1\ \mathsf{a}_2 \,.\, (\mathsf{a}_1 \to \mathsf{a}_2) \to (\mathsf{List}\ \mathsf{a}_1 \to \mathsf{List}\ \mathsf{a}_2)$ as desired. In the base case $\mathsf{Map}\{[\star]\}\ \mathsf{t}_1\ \mathsf{t}_2$ equals the type of a conversion function. The inductive case has a very characteristic form, which we have already encountered several times. It specifies that a 'conversion function' between the type constructors t_1 and t_2 is a function that maps a conversion function between a_1 and a_2 to a conversion function between $\mathsf{t}_1\ \mathsf{a}_1$ and $\mathsf{t}_2\ \mathsf{a}_2$, for all possible instances of a_1 and a_2. Roughly speaking, $\mathsf{Map}\{[\kappa \to \nu]\}\ \mathsf{t}_1\ \mathsf{t}_2$ is the type of a 'conversion function'-transformer. It is not hard to see that the type signatures of *mapList*, *mapListBase*, and *mapFix* 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: for instance, the *map* for a second-order kinded type has a rank-2 type signature.

The definition of *map* itself is straightforward.

$$\begin{array}{ll}
map\{\!|\mathsf{t} :: \kappa|\!\} & :: \mathsf{Map}\{[\kappa]\}\ \mathsf{t}\ \mathsf{t} \\
map\{\!|\mathsf{Char}|\!\}\ c & = c \\
map\{\!|\mathsf{Int}|\!\}\ i & = i \\
map\{\!|\mathsf{Unit}|\!\}\ Unit & = Unit \\
map\{\!|{:}{+}{:}|\!\}\ mapa\ mapb\ (Inl\ a) & = Inl\ (mapa\ a) \\
map\{\!|{:}{+}{:}|\!\}\ mapa\ mapb\ (Inr\ b) & = Inr\ (mapb\ b) \\
map\{\!|{:}{*}{:}|\!\}\ mapa\ mapb\ (a\ {:}{*}{:}\ b) & = mapa\ a\ {:}{*}{:}\ mapb\ b
\end{array}$$

This definition contains all the ingredients needed to derive *map*s for arbitrary data types of arbitrary kinds. As an aside, 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 $(*)$.

$$\begin{array}{ll}
map\{\!|\mathsf{Char}|\!\} & = id \\
map\{\!|\mathsf{Int}|\!\} & = id \\
map\{\!|\mathsf{Unit}|\!\} & = id \\
map\{\!|{:}{+}{:}|\!\}\ mapa\ mapb & = mapa + mapb \\
map\{\!|{:}{*}{:}|\!\}\ mapa\ mapb & = mapa * mapb
\end{array}$$

Even more succinctly, we have $map\{\!|{:}{+}{:}|\!\} = (+)$ and $map\{\!|{:}{*}{:}|\!\} = (*)$.

As usual, to apply a generic function we simply instantiate the type-index to a closed type.

$$\begin{array}{l}
map\{\!|\mathsf{List\ Char}|\!\}\ \texttt{"hello world"} \\
\Longrightarrow \texttt{"hello world"} \\
map\{\!|\mathsf{List}|\!\}\ toUpper\ \texttt{"hello world"} \\
\Longrightarrow \texttt{"HELLO WORLD"}
\end{array}$$

We can also use *map* to define other generic functions.

$$\begin{array}{ll}
distribute\{\!|\mathsf{t} :: \star \to \star|\!\} & :: \forall \mathsf{a}\ \mathsf{b} \,.\, \mathsf{t}\ \mathsf{a} \to \mathsf{b} \to \mathsf{t}\ (\mathsf{a}, \mathsf{b}) \\
distribute\{\!|\mathsf{t}|\!\}\ x\ b & = map\{\!|\mathsf{t}|\!\}\ (\lambda a \to (a, b))\ x
\end{array}$$

The call $distribute\{\!|\mathsf{t}|\!\}\ x\ b$ pairs the value b with every element contained in the structure x.

2.2 Kind-indexed types and type-indexed values

In general, the definition of a type-indexed value consists of two parts: a type signature, which typically involves a kind-indexed type, and a set of equations, one for each type constant. A kind-indexed type is defined as follows:

$$\begin{aligned}
&\textbf{type}\ \mathsf{Poly}\{\![\star]\!\}\ \mathsf{t}_1\ \ldots\ \mathsf{t}_n &&= \ldots \\
&\textbf{type}\ \mathsf{Poly}\{\![\kappa \to \nu]\!\}\ \mathsf{t}_1\ \ldots\ \mathsf{t}_n &&= \forall \mathsf{a}_1\ \ldots\ \mathsf{a}_n \,.\, \mathsf{Poly}\{\![\kappa]\!\}\ \mathsf{a}_1\ \ldots\ \mathsf{a}_n \\
& && \qquad \to \mathsf{Poly}\{\![\nu]\!\}\ (\mathsf{t}_1\ \mathsf{a}_1)\ \ldots\ (\mathsf{t}_n\ \mathsf{a}_n).
\end{aligned}$$

The second clause is the same for all kind-indexed types so that the generic programmer merely has to fill out the right-hand side of the first equation. Actually, Generic Haskell offers a slightly more general form (see Section 2.4), which is the reason why we do not leave out the second clause.

Given a kind-indexed type, the definition of a type-indexed value takes on the following schematic form.

$$\begin{aligned}
&\mathit{poly}\{\!|\mathsf{t} :: \kappa|\!\} &&:: \mathsf{Poly}\{\![\kappa]\!\}\ \mathsf{t}\ \ldots\ \mathsf{t} \\
&\mathit{poly}\{\!|\mathsf{Char}|\!\} &&= \ldots \\
&\mathit{poly}\{\!|\mathsf{Int}|\!\} &&= \ldots \\
&\mathit{poly}\{\!|\mathsf{Unit}|\!\} &&= \ldots \\
&\mathit{poly}\{\!|{:}{+}{:}|\!\}\ \mathit{polya}\ \mathit{polyb} &&= \ldots \\
&\mathit{poly}\{\!|{:}{*}{:}|\!\}\ \mathit{polya}\ \mathit{polyb} &&= \ldots
\end{aligned}$$

We have one clause for each primitive type (Int, Char etc) and one clause for each of the three type constructors Unit, ':*:', and ':+:'. Again, the generic programmer has to fill out the right-hand sides. To be well-typed, the $\mathit{poly}\{\!|\mathsf{t}{::}\kappa|\!\}$ instance must have type $\mathsf{Poly}\{\![\kappa]\!\}\ \mathsf{t}\ \ldots\ \mathsf{t}$ as stated in the type signature of *poly*. Actually, the type signature can be more elaborate (we will see examples of this in Section 2.4).

The major insight of the mapping example is that a kind-indexed type can have several type arguments. Recall in this respect the type of the generic equality function:

$$\begin{aligned}
&\textbf{type}\ \mathsf{Eq}\{\![\star]\!\}\ \mathsf{t} &&= \mathsf{t} \to \mathsf{t} \to \mathsf{Bool} \\
&\textbf{type}\ \mathsf{Eq}\{\![\kappa \to \nu]\!\}\ \mathsf{t} &&= \forall \mathsf{a} \,.\, \mathsf{Eq}\{\![\kappa]\!\}\ \mathsf{a} \to \mathsf{Eq}\{\![\nu]\!\}\ (\mathsf{t}\ \mathsf{a}).
\end{aligned}$$

Interestingly, we can generalize the type since the two arguments of equality need not be of the same type.

$$\begin{aligned}
&\textbf{type}\ \mathsf{Eq}\{\![\star]\!\}\ \mathsf{t}_1\ \mathsf{t}_2 &&= \mathsf{t}_1 \to \mathsf{t}_2 \to \mathsf{Bool} \\
&\textbf{type}\ \mathsf{Eq}\{\![\kappa \to \nu]\!\}\ \mathsf{t}_1\ \mathsf{t}_2 &&= \forall \mathsf{a}_1\ \mathsf{a}_2 \,.\, \mathsf{Eq}\{\![\kappa]\!\}\ \mathsf{a}_1\ \mathsf{a}_2 \to \mathsf{Eq}\{\![\nu]\!\}\ (\mathsf{t}_1\ \mathsf{a}_1)\ (\mathsf{t}_2\ \mathsf{a}_2)
\end{aligned}$$

We can assign $\mathit{eq}\{\!|\mathsf{t} :: \kappa|\!\}$ the more general type $\mathsf{Eq}\{\![\kappa]\!\}\ \mathsf{t}\ \mathsf{t}$. Though this gives us a greater degree of flexibility, the definition of *eq* itself is not affected by this change! As an example, we could pass $\mathit{eq}\{\!|\mathsf{List}|\!\}$ the 'equality test' $\mathit{match} :: \mathsf{Female} \to \mathsf{Male} \to \mathsf{Bool}$ in order to check whether corresponding list entries match.

2.3 Embedding-projection maps

Most of the generic functions cannot sensibly be defined for the function space. For instance, equality of functions is not decidable. The mapping function *map* cannot be defined for function types since $(\rightarrow)$ is *contravariant* in its first argument:

$$\begin{aligned}
&(\rightarrow) && :: \forall \mathsf{a}_1\ \mathsf{a}_2 \,.\, (\mathsf{a}_2 \rightarrow \mathsf{a}_1) \rightarrow \\
& && \qquad \forall \mathsf{b}_1\ \mathsf{b}_2 \,.\, (\mathsf{b}_1 \rightarrow \mathsf{b}_2) \rightarrow ((\mathsf{a}_1 \rightarrow \mathsf{b}_1) \rightarrow (\mathsf{a}_2 \rightarrow \mathsf{b}_2)) \\
&(f \rightarrow g)\ h = g \cdot h \cdot f .
\end{aligned}$$

In the case of mapping functions we can remedy the situation by drawing from the theory of embeddings and projections [13]. The central idea is to supply a pair of functions, *from* and *to*, where *to* is the left-inverse of *from*, that is, $to \cdot from = id$. (If the functions additionally satisfy $from \cdot to \sqsubseteq id$, then they are called an *embedding-projection pair*.) We use the following data type to represent embedding-projection pairs (the code below make use of Haskell's record syntax).

$$\begin{aligned}
&\mathbf{data}\ \mathsf{EP}\ \mathsf{a}_1\ \mathsf{a}_2 = EP\{from :: \mathsf{a}_1 \rightarrow \mathsf{a}_2, to :: \mathsf{a}_2 \rightarrow \mathsf{a}_1\} \\
&id_E && :: \forall \mathsf{a} \,.\, \mathsf{EP}\ \mathsf{a}\ \mathsf{a} \\
&id_E && = EP\{from = id, to = id\} \\
&(\circ) && :: \forall \mathsf{a}\ \mathsf{b}\ \mathsf{c} \,.\, \mathsf{EP}\ \mathsf{b}\ \mathsf{c} \rightarrow \mathsf{EP}\ \mathsf{a}\ \mathsf{b} \rightarrow \mathsf{EP}\ \mathsf{a}\ \mathsf{c} \\
&f \circ g && = EP\{from = from\ f \cdot from\ g, to = to\ g \cdot to\ f\}
\end{aligned}$$

Here, id_E is the identity embedding-projection pair and '$\circ$' shows how to compose two embedding-projection pairs (note that the composition is reversed for the projection). In fact, id_E and '$\circ$' give rise to the category $\mathcal{C}po^e$, the category of complete partial orders and embedding-projection pairs. This category has the interesting property that the function space can be turned into a *covariant* functor.

$$\begin{aligned}
&(+_E) && :: \forall \mathsf{a}_1\ \mathsf{a}_2 \,.\, \mathsf{EP}\ \mathsf{a}_1\ \mathsf{a}_2 \rightarrow \forall \mathsf{b}_1\ \mathsf{b}_2 \,.\, \mathsf{EP}\ \mathsf{b}_1\ \mathsf{b}_2 \rightarrow \mathsf{EP}\ (\mathsf{a}_1 :\!+\!: \mathsf{b}_1)\ (\mathsf{a}_2 :\!+\!: \mathsf{b}_2) \\
&f +_E g && = EP\{from = from\ f + from\ g, to = to\ f + to\ g\} \\
&(*_E) && :: \forall \mathsf{a}_1\ \mathsf{a}_2 \,.\, \mathsf{EP}\ \mathsf{a}_1\ \mathsf{a}_2 \rightarrow \forall \mathsf{b}_1\ \mathsf{b}_2 \,.\, \mathsf{EP}\ \mathsf{b}_1\ \mathsf{b}_2 \rightarrow \mathsf{EP}\ (\mathsf{a}_1 :\!*\!: \mathsf{b}_1)\ (\mathsf{a}_2 :\!*\!: \mathsf{b}_2) \\
&f *_E g && = EP\{from = from\ f * from\ g, to = to\ f * to\ g\} \\
&(\rightarrow_E) && :: \forall \mathsf{a}_1\ \mathsf{a}_2 \,.\, \mathsf{EP}\ \mathsf{a}_1\ \mathsf{a}_2 \rightarrow \forall \mathsf{b}_1\ \mathsf{b}_2 \,.\, \mathsf{EP}\ \mathsf{b}_1\ \mathsf{b}_2 \rightarrow \mathsf{EP}\ (\mathsf{a}_1 \rightarrow \mathsf{b}_1)\ (\mathsf{a}_2 \rightarrow \mathsf{b}_2) \\
&f \rightarrow_E g && = EP\{from = to\ f \rightarrow from\ g, to = from\ f \rightarrow to\ g\}
\end{aligned}$$

Given these helper functions the generic embedding-projection map can be defined as follows.

$$\begin{aligned}
&\mathbf{type}\ \mathsf{MapE}\{\![\star]\!\}\ \mathsf{t}_1\ \mathsf{t}_2 && = \mathsf{EP}\ \mathsf{t}_1\ \mathsf{t}_2 \\
&\mathbf{type}\ \mathsf{MapE}\{\![\kappa \rightarrow \nu]\!\}\ \mathsf{t}_1\ \mathsf{t}_2 && = \forall \mathsf{a}_1\ \mathsf{a}_2 \,.\, \mathsf{MapE}\{\![\kappa]\!\}\ \mathsf{a}_1\ \mathsf{a}_2 \rightarrow \mathsf{MapE}\{\![\nu]\!\}\ (\mathsf{t}_1\ \mathsf{a}_1)\ (\mathsf{t}_2\ \mathsf{a}_2)
\end{aligned}$$

$$\begin{aligned}
&mapE\{\!|\mathsf{t} :: \kappa|\!\} && :: \mathsf{MapE}\{\![\kappa]\!\}\ \mathsf{t}\ \mathsf{t} \\
&mapE\{\!|\mathsf{Char}|\!\} && = id_E \\
&mapE\{\!|\mathsf{Int}|\!\} && = id_E \\
&mapE\{\!|\mathsf{Unit}|\!\} && = id_E \\
&mapE\{\!|\mathsf{:+:}|\!\} && = (+_E) \\
&mapE\{\!|\mathsf{:*:}|\!\} && = (*_E) \\
&mapE\{\!|\to|\!\} && = (\to_E)
\end{aligned}$$

We will see in Section 3.4 that embedding-projection maps are useful for changing the representation of data.

2.4 Reductions

The Haskell standard library defines a vast number of list processing functions. We have among others:

$$\begin{aligned}
&sum, product && :: \forall \mathsf{a}\,.\,(Num\ \mathsf{a}) \Rightarrow [\mathsf{a}] \to \mathsf{a} \\
&and, or && :: [\mathsf{Bool}] \to \mathsf{Bool} \\
&all, any && :: \forall \mathsf{a}\,.\,(\mathsf{a} \to \mathsf{Bool}) \to ([\mathsf{a}] \to \mathsf{Bool}) \\
&length && :: \forall \mathsf{a}\,.\,[\mathsf{a}] \to \mathsf{Int} \\
&minimum, maximum && :: \forall \mathsf{a}\,.\,(Ord\ \mathsf{a}) \Rightarrow [\mathsf{a}] \to \mathsf{a} \\
&concat && :: \forall \mathsf{a}\,.\,[[\mathsf{a}]] \to [\mathsf{a}].
\end{aligned}$$

These are examples of so-called *reductions*. A reduction or a crush [31] is a function that collapses a structure of values of type t (such a structure is also known as a container) into a single value of type t. This section explains how to define reductions that work for all types of all kinds. To illustrate the main idea we first discuss three motivating examples. Let us start with a generic function that counts the number of values of type Int within a given structure of some type.

Here is the type of the generic counter

$$\begin{aligned}
&\textbf{type}\ \mathsf{Count}\{\![\star]\!\}\ \mathsf{t} && = \mathsf{t} \to \mathsf{Int} \\
&\textbf{type}\ \mathsf{Count}\{\![\kappa \to \nu]\!\}\ \mathsf{t} && = \forall \mathsf{a}\,.\,\mathsf{Count}\{\![\kappa]\!\}\ \mathsf{a} \to \mathsf{Count}\{\![\nu]\!\}\ (\mathsf{t}\ \mathsf{a})
\end{aligned}$$

and here is its definition.

$$\begin{aligned}
&count\{\!|\mathsf{t} :: \kappa|\!\} && :: \mathsf{Count}\{\![\kappa]\!\}\ \mathsf{t} \\
&count\{\!|\mathsf{Char}|\!\}\ c && = 0 \\
&count\{\!|\mathsf{Int}|\!\}\ i && = 1 \\
&count\{\!|\mathsf{Unit}|\!\}\ Unit && = 0 \\
&count\{\!|\mathsf{:+:}|\!\}\ counta\ countb\ (Inl\ a) && = counta\ a \\
&count\{\!|\mathsf{:+:}|\!\}\ counta\ countb\ (Inr\ b) && = countb\ b \\
&count\{\!|\mathsf{:*:}|\!\}\ counta\ countb\ (a\ \mathsf{:*:}\ b) && = counta\ a + countb\ b
\end{aligned}$$

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

$$
\begin{array}{ll}
sum\{|\mathsf{t} :: \kappa|\} & :: \mathsf{Count}\{[\kappa]\}\ \mathsf{t} \\
sum\{|\mathsf{Char}|\}\ c & = 0 \\
sum\{|\mathsf{Int}|\}\ i & = 0 \\
sum\{|\mathsf{Unit}|\}\ Unit & = 0 \\
sum\{|{:}{+}{:}|\}\ suma\ sumb\ (Inl\ a) & = suma\ a \\
sum\{|{:}{+}{:}|\}\ suma\ sumb\ (Inr\ b) & = sumb\ b \\
sum\{|{:}{*}{:}|\}\ suma\ sumb\ (a\ {:}{*}{:}\ b) & = suma\ a + sumb\ b
\end{array}
$$

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

$$sum\{|\mathsf{t} :: \star \to \star|\} :: \forall \mathsf{a} \,.\, (\mathsf{a} \to \mathsf{Int}) \to (\mathsf{t}\ \mathsf{a} \to \mathsf{Int})$$

If we pass the identity function to sum, we obtain a function that sums up a structure of integers. Another viable choice is $const$ 1; this yields a function of type $\forall \mathsf{a} \,.\, \mathsf{t}\ \mathsf{a} \to \mathsf{Int}$ that counts the number of values of type a in a given structure of type $\mathsf{t}\ \mathsf{a}$.

$$
\begin{array}{l}
sum\{|\mathsf{List}\ \mathsf{Int}|\}\ [2, 7, 1965] \\
\Longrightarrow 0 \\
sum\{|\mathsf{List}|\}\ id\ [2, 7, 1965] \\
\Longrightarrow 1974 \\
sum\{|\mathsf{List}|\}\ (const\ 1)\ [2, 7, 1965] \\
\Longrightarrow 3
\end{array}
$$

As usual, we can use generic abstractions to capture these idioms.

$$
\begin{array}{ll}
fsum\{|\mathsf{t} :: \star \to \star|\} & :: \mathsf{t}\ \mathsf{Int} \to \mathsf{Int} \\
fsum\{|\mathsf{t}|\} & = sum\{|\mathsf{t}|\}\ id \\
fsize\{|\mathsf{t} :: \star \to \star|\} & :: \forall \mathsf{a} \,.\, \mathsf{t}\ \mathsf{a} \to \mathsf{Int} \\
fsize\{|\mathsf{t}|\} & = sum\{|\mathsf{t}|\}\ (const\ 1)
\end{array}
$$

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

$$
\begin{array}{ll}
\textbf{type}\ \mathsf{Flatten}\{[\star]\}\ \mathsf{t}\ \mathsf{x} & = \mathsf{t} \to [\mathsf{x}] \\
\textbf{type}\ \mathsf{Flatten}\{[\kappa \to \nu]\}\ \mathsf{t}\ \mathsf{x} & = \forall \mathsf{a} \,.\, \mathsf{Flatten}\{[\kappa]\}\ \mathsf{a}\ \mathsf{x} \to \mathsf{Flatten}\{[\nu]\}\ (\mathsf{t}\ \mathsf{a})\ \mathsf{x}
\end{array}
$$

makes use of a simple extension: $\mathsf{Flatten}\{[\kappa]\}\ \mathsf{t}\ \mathsf{x}$ takes an additional type parameter, x, that is passed unchanged to the base case. One can safely think of x as a type parameter that is global to the definition. The code for *flatten* is similar to the code for *sum*.

$$\begin{aligned}
&\mathit{flatten}\{\!|\mathsf{t} :: \kappa|\!\} && :: \forall \mathsf{x}\,.\,\mathsf{Flatten}\{\![\kappa]\!\}\ \mathsf{t}\ \mathsf{x}\\
&\mathit{flatten}\{\!|\mathsf{Char}|\!\}\ c && = [\,]\\
&\mathit{flatten}\{\!|\mathsf{Int}|\!\}\ i && = [\,]\\
&\mathit{flatten}\{\!|\mathsf{Unit}|\!\}\ \mathit{Unit} && = [\,]\\
&\mathit{flatten}\{\!|{:}{+}{:}|\!\}\ \mathit{fla}\ \mathit{flb}\ (\mathit{Inl}\ a) && = \mathit{fla}\ a\\
&\mathit{flatten}\{\!|{:}{+}{:}|\!\}\ \mathit{fla}\ \mathit{flb}\ (\mathit{Inr}\ b) && = \mathit{flb}\ b\\
&\mathit{flatten}\{\!|{:}{*}{:}|\!\}\ \mathit{fla}\ \mathit{flb}\ (a\ {:}{*}{:}\ b) && = \mathit{fla}\ a +\!\!+\ \mathit{flb}\ b
\end{aligned}$$

The type signature of *flatten* makes precise that its instances are parametric in the type of list elements. We have, for instance,

$$\begin{aligned}
&\mathit{flatten}\{\!|\mathsf{Char}|\!\} :: \forall \mathsf{x}\,.\,\mathsf{Char} \to [\mathsf{x}]\\
&\mathit{flatten}\{\!|\mathsf{Rose}|\!\} :: \forall \mathsf{x}\,.\,\forall \mathsf{a}\,.\,(\mathsf{a} \to [\mathsf{x}]) \to (\mathsf{Rose}\ \mathsf{a} \to [\mathsf{x}]).
\end{aligned}$$

Interestingly, the type dictates that $\mathit{flatten}\{\!|\mathsf{Char}|\!\} = \mathit{const}\ [\,]$. Like *sum*, the *flatten* function is pointless for types but useful for type constructors.

$$\begin{aligned}
&\mathit{fflatten}\{\!|\mathsf{t} :: \star \to \star|\!\} && :: \forall \mathsf{a}\,.\,\mathsf{t}\ \mathsf{a} \to [\mathsf{a}]\\
&\mathit{fflatten}\{\!|\mathsf{t}|\!\} && = \mathit{flatten}\{\!|\mathsf{t}|\!\}\ \mathit{wrap}\ \mathbf{where}\ \mathit{wrap}\ a = [\,a\,]
\end{aligned}$$

The definitions of *sum* and *flatten* exhibit a common pattern: the elements of a base type are replaced by a constant (0 and $[\,]$, respectively) and the pair constructor is replaced by a binary operator ($(+)$ and $(+\!\!+)$, respectively). The generic function *reduce* abstracts away from these particularities. Its kind-indexed type is given by

$$\begin{aligned}
&\mathbf{type}\ \mathsf{Reduce}\{\![\star]\!\}\ \mathsf{t}\ \mathsf{x} && = \mathsf{x} \to (\mathsf{x} \to \mathsf{x} \to \mathsf{x}) \to \mathsf{t} \to \mathsf{x}\\
&\mathbf{type}\ \mathsf{Reduce}\{\![\kappa \to \nu]\!\}\ \mathsf{t}\ \mathsf{x} && = \forall \mathsf{a}\,.\,\mathsf{Reduce}\{\![\kappa]\!\}\ \mathsf{a}\ \mathsf{x} \to \mathsf{Reduce}\{\![\nu]\!\}\ (\mathsf{t}\ \mathsf{a})\ \mathsf{x}.
\end{aligned}$$

Note that the type argument x is passed unchanged to the recursive calls.

$$\begin{aligned}
&\mathit{reduce}\{\!|\mathsf{t} :: \kappa|\!\} && :: \forall \mathsf{x}\,.\,\mathsf{Reduce}\{\![\kappa]\!\}\ \mathsf{t}\ \mathsf{x}\\
&\mathit{reduce}\{\!|\mathsf{Char}|\!\}\ e\ \mathit{op}\ c && = e\\
&\mathit{reduce}\{\!|\mathsf{Int}|\!\}\ e\ \mathit{op}\ i && = e\\
&\mathit{reduce}\{\!|\mathsf{Unit}|\!\}\ e\ \mathit{op}\ \mathit{Unit} && = e\\
&\mathit{reduce}\{\!|{:}{+}{:}|\!\}\ \mathit{reda}\ \mathit{redb}\ e\ \mathit{op}\ (\mathit{Inl}\ a) && = \mathit{reda}\ e\ \mathit{op}\ a\\
&\mathit{reduce}\{\!|{:}{+}{:}|\!\}\ \mathit{reda}\ \mathit{redb}\ e\ \mathit{op}\ (\mathit{Inr}\ b) && = \mathit{redb}\ e\ \mathit{op}\ b\\
&\mathit{reduce}\{\!|{:}{*}{:}|\!\}\ \mathit{reda}\ \mathit{redb}\ e\ \mathit{op}\ (a\ {:}{*}{:}\ b) && = \mathit{reda}\ e\ \mathit{op}\ a\ `\mathit{op}`\ \mathit{redb}\ e\ \mathit{op}\ b
\end{aligned}$$

Using *reduce* we can finally give generic versions of Haskell's list processing functions listed in the beginning of this section.

$$\begin{aligned}
&\mathit{freduce}\{\!|\mathsf{t} :: \star \to \star|\!\} && :: \forall \mathsf{x}\,.\,\mathsf{x} \to (\mathsf{x} \to \mathsf{x} \to \mathsf{x}) \to \mathsf{t}\ \mathsf{x} \to \mathsf{x}\\
&\mathit{freduce}\{\!|\mathsf{t}|\!\} && = \mathit{reduce}\{\!|\mathsf{t}|\!\}\ (\lambda e\ \mathit{op}\ a \to a)
\end{aligned}$$

$$
\begin{aligned}
&\mathit{fsum}\{|\mathsf{t}|\} &&= \mathit{freduce}\{|\mathsf{t}|\}\ 0\ (+)\\
&\mathit{fproduct}\{|\mathsf{t}|\} &&= \mathit{freduce}\{|\mathsf{t}|\}\ 1\ (*)\\
&\mathit{fand}\{|\mathsf{t}|\} &&= \mathit{freduce}\{|\mathsf{t}|\}\ \mathit{True}\ (\wedge)\\
&\mathit{for}\{|\mathsf{t}|\} &&= \mathit{freduce}\{|\mathsf{t}|\}\ \mathit{False}\ (\vee)\\
&\mathit{fall}\{|\mathsf{t}|\}\ f &&= \mathit{fand}\{|\mathsf{t}|\} \cdot \mathit{map}\{|\mathsf{t}|\}\ f\\
&\mathit{fany}\{|\mathsf{t}|\}\ f &&= \mathit{for}\{|\mathsf{t}|\} \cdot \mathit{map}\{|\mathsf{t}|\}\ f\\
&\mathit{fminimum}\{|\mathsf{t}|\} &&= \mathit{freduce}\{|\mathsf{t}|\}\ \mathit{maxBound}\ \mathit{min}\\
&\mathit{fmaximum}\{|\mathsf{t}|\} &&= \mathit{freduce}\{|\mathsf{t}|\}\ \mathit{minBound}\ \mathit{max}\\
&\mathit{fflatten}\{|\mathsf{t}|\} &&= \mathit{freduce}\{|\mathsf{t}|\}\ [\,]\ (+\!\!+)
\end{aligned}
$$

Typically, the two arguments of *freduce* form a *monoid*: the second argument is associative and has the first as its neutral element.

As an aside, note that the definition of *fflatten* has a quadratic running time. Exercise 8 seeks to remedy this defect.

2.5 Pretty printing

Generic functions are defined by induction on the *structure* of types. Annoyingly, this is not quite enough. Consider, for example, the method *showsPrec* of the Haskell class *Show*. To be able to give a generic definition for *showsPrec*, the *names* of the constructors, and their fixities, must be made available.

To this end we provide one additional type case.

$$\mathit{poly}\{|\mathsf{Con}\ c|\}\ \mathit{polya} = \ldots$$

Roughly speaking, this case is invoked whenever we pass by a constructor. Quite unusual, the variable c that appears in the type index is bound to a *value* of type ConDescr and provides the required information about the name of a constructor, its arity etc.

$$
\begin{aligned}
&\textbf{data}\ \mathsf{ConDescr} &&= \mathit{ConDescr}\{\ \mathit{conName} :: \mathsf{String},\\
& && \qquad\qquad\quad \mathit{conType} :: \mathsf{String},\\
& && \qquad\qquad\quad \mathit{conArity} :: \mathsf{Int},\\
& && \qquad\qquad\quad \mathit{conLabels} :: \mathsf{Bool},\\
& && \qquad\qquad\quad \mathit{conFixity} :: \mathsf{Fixity}\}\\
&\textbf{data}\ \mathsf{Fixity} &&= \mathit{Nonfix}\\
& &&\mid \mathit{Infix}\{\mathit{prec} :: \mathsf{Int}\}\\
& &&\mid \mathit{Infixl}\{\mathit{prec} :: \mathsf{Int}\}\\
& &&\mid \mathit{Infixr}\{\mathit{prec} :: \mathsf{Int}\}
\end{aligned}
$$

The *Con* data type itself is a simple wrapper type.

$$\textbf{data}\ \mathsf{Con\ a} = \mathit{Con}\ \mathsf{a}$$

Using *conName* and *conArity* we can implement a simple variant of Haskell's *showsPrec* function (ShowS, *shows*, *showChar*, and *showString* are predefined in

Haskell).

```
type Shows{[⋆]} t              = t → ShowS
type Shows{[κ → ν]} t          = ∀a . Shows{[κ]} a → Shows{[ν]} (t a)
gshows{|t :: κ|}               :: Shows{[κ]} t
gshows{|:+:|} sa sb (Inl a)    = sa a
gshows{|:+:|} sa sb (Inr b)    = sb b
gshows{|Con c|} sa (Con a)
    | conArity c == 0          = showString (conName c)
    | otherwise                = showChar '(' · showString (conName c)
                                   · showChar ' ' · sa a · showChar ')'
gshows{|:*:|} sa sb (a :*: b)  = sa a · showChar ' ' · sb b
gshows{|Unit|} Unit            = showString ""
gshows{|Char|}                 = shows
gshows{|Int|}                  = shows
```

The first and the second equation discard the constructors *Inl* and *Inr*. They are not required since the constructor names can be accessed via the type pattern Con *c*. If the constructor is nullary, its string representation is emitted. Otherwise, the constructor name is printed followed by a space followed by the representation of its arguments. The fourth equation applies if a constructor has more than one component. In this case the components are separated by a space.

In a nutshell, via ':+:' we get to the constructors, Con signals that we hit a constructor, and via ':*:' we get to the arguments of a constructor. Or, to put it differently, the generic programmer views, for instance, the list data type

```
data List a = Nil | Cons a (List a)
```

as if it were given by the following type definition.

```
type List a = (Con Unit) :+: (Con (a :*: List a))
```

As a simple example, the list *Cons* 1 *Nil* is represented by *Inr* (*Con* (1 :*: *Inl* (*Con Unit*))).

It should be noted that descriptors of type *ConDescr* appear only in the type index; they have no counterpart on the value level as value constructors are encoded using *Inl* and *Inr*. If a generic definition does not include a case for the type pattern Con *c*, then we tacitly assume that *poly*{|Con *c*|} *polya* = *polya*. (Actually, the default definition is slightly more involved since a and Con a are different types: the data constructor *Con* must be wrapped and unwrapped at the appropriate places. Section 3.4 explains how to accomplish this representation change in a systematic manner.) Now, why does the type Con *c* incorporate information about the constructor? One might suspect that it is sufficient to supply this information on the value level. Doing so would work for *show*, but would fail for a generic version of *read*, which converts a string to a value.

Consider the ‘Con’ case:

```
greads{|Con c|} ra s = [(x, s₃) | (s₁, s₂) ← lex s,
                                   s₁ == conName c,
                                   (x, s₃) ← ra s₂].
```

The important point is that *greads produces* (not consumes) the value, and yet it requires access to the constructor name.

The *gshows* function generates one long string, which does not look pretty at all when printed out. We can do better using Wadler's pretty printing combinators [48].

```
data Doc
empty  :: Doc
(◇)    :: Doc → Doc → Doc
string :: String → Doc
nl     :: Doc
nest   :: Int → Doc → Doc
group  :: Doc → Doc
render :: Int → Doc → String
```

The value *empty* represents the empty document, the operator ‘◇’ catenates two documents, and *string* converts a string to an atomic document. The document *nl* denotes a potential line break. The function *nest* increases the indentation for all line breaks within its document argument. The function *group* marks its argument as a unit: it is printed out on a single line by converting all its potential line breaks into single spaces if this is possible without exceeding a given line-width limit. Finally, *render w d* converts a document to a string respecting the line width w.

The generic function *ppPrec*{|t|} d x takes a precedence level d (a value from 0 to 10), a value x of type t and returns a document of type Doc. The function essentially follows the structure of *gshows* except that it replaces the ShowS functions by pretty printing combinators.

```
type Pretty{[⋆]} t              = Int → t → Doc
type Pretty{[κ → ν]} t          = ∀a . Pretty{[κ]} a → Pretty{[ν]} (t a)
ppPrec{|t :: κ|}                :: Pretty{[κ]} t
ppPrec{|:+:|} ppa ppb d (Inl a) = ppa d a
ppPrec{|:+:|} ppa ppb d (Inr b) = ppb d b
ppPrec{|Con c|} ppa d (Con a)
  | conArity c == 0             = string (conName c)
  | otherwise                   = group (nest 2 (ppParen (d > 9) doc))
  where doc                     = string (conName c) ◇ nl ◇ ppa 10 a
ppPrec{|:*:|} ppa ppb d (a :*: b) = ppa d a ◇ nl ◇ ppb d b
ppPrec{|Unit|} d Unit           = empty
```

$$\begin{array}{ll}
\mathit{ppPrec}\{\!|\mathsf{Int}|\!\}\ d\ i & = \mathit{string}\ (\mathit{show}\ i) \\
\mathit{ppPrec}\{\!|\mathsf{Char}|\!\}\ d\ c & = \mathit{string}\ (\mathit{show}\ c) \\
\mathit{ppParen} & :: \mathsf{Bool} \to \mathsf{Doc} \to \mathsf{Doc} \\
\mathit{ppParen}\ \mathit{False}\ d & = d \\
\mathit{ppParen}\ \mathit{True}\ d & = \mathit{string}\ \texttt{"("} \diamond d \diamond \mathit{string}\ \texttt{")"}
\end{array}$$

The helper function *ppParen* encloses its second argument in parenthesis if its first evaluates to *True*.

2.6 Running Generic Haskell

This section explains how to use the Generic Haskell compiler called `gh`. It has two basic modes of operation. If `gh` is called without arguments, the user is prompted for a file name (relative to the working directory). The compiler then processes the file and generates an output file with the same basename as the input file, but the extension '`.hs`'. Generic Haskell source files typically have the extension '`.ghs`'. Alternatively, input files can be specified on the command line. A typical invocation is:

```
path_to_gh/bin/gh -L path_to_gh/lib/ your_file.ghs
```

A number of command line options are available:

```
Usage: gh [options...] files...
  -v      --verbose      (number of v's controls the verbosity)
  -m      --make         follow dependencies
  -V      --version      show version info
  -h, -?  --help         show help
          --cut=N        cut computations after N iterations
  -C      --continue     continue after errors
  -L DIR  --library=DIR  add DIR to search path
```

The first level of verbosity (no `-v` flag) produces only error messages. The second level (`-v`) additionally provides diagnostic information and warnings. The third level (`-vv`) produces debugging information.

The `-m` (or `--make`) option instructs the compiler to chase module dependencies and to automatically process those modules which require compilation.

The option `--cut=N` stops the compilation after `N` iterations of the specialization mechanism (the default is to stop after 50 iterations). The first level of verbosity (`-v`) can be used to report the number of required iterations.

The `-C` (or `--continue`) option forces the compiler to continue compilation even when an error is encountered. This can be used to generate further error messages or to inspect the generated code.

The Generic Haskell compiler compiles '`.ghs`' files and produces '`.hs`' files which can subsequently be compiled using a Haskell compiler. In addition, the compiler produces '`.ghi`' interface files, which will be used in subsequent compilations to avoid unnecessary recompilation. As an example, Figure 1 displays the source code for the generic version of Haskell's *shows* function. Note that *type* arguments are enclosed in `{|·|}` brackets, while `{[·]}` embraces *kind* arguments.

```
type Shows {[ * ]} t    =  t -> ShowS
type Shows {[ k -> l ]} t
       =  forall a . Shows {[ k ]} a -> Shows {[ l ]} (t a)

gshows {| t :: k |}     :: Shows {[ k ]} t
gshows {| :+: |} sa sb (Inl a)  =  sa a
gshows {| :+: |} sa sb (Inr b)  =  sb b
gshows {| Con c|} sa (Con a)
  | conArity c == 0     =  showString (conName c)
  | otherwise           =  showChar '(' . showString (conName c)
                           . showChar ' ' . sa a . showChar ')'
gshows {| :*: |} sa sb (a :*: b)
                        =  sa a . showChar ' ' . sb b
gshows {| Unit |} Unit  =  showString ""
gshows {| Char |}       =  shows
gshows {| Int |}        =  shows

data Tree a b           =  Tip a | Node (Tree a b) b (Tree a b)

main                    =  putStrLn (gshows {| Tree Int String |} ex "")

ex                      :: Tree Int String
ex                      =  Node (Tip 1) "hello"
                                (Node (Tip 2) "world" (Tip 3))
```

Fig. 1. A generic implementation of the *shows* function.

The Generic Haskell compiler generates ordinary Haskell code (Haskell 98 augmented with rank-2 types), which can be run or compiled using the Glasgow Haskell Compiler, Hugs, or any other Haskell compiler. You only have to ensure that the path to `GHPrelude.hs` (and to other Generic Haskell libraries), which can be found in the `lib` subdirectory, is included in your compiler's search path.

The Generic Haskell compiler is shipped with an extensive library, which provides further examples of generic functions.

Exercise 6. Let M be a monad. A *monadic mapping function* for a type constructor F of kind $\star \to \star$ lifts a given function of type $\mathsf{a} \to \mathsf{M}\ \mathsf{b}$ to a function of type $\mathsf{F}\ \mathsf{a} \to \mathsf{M}\ (\mathsf{F}\ \mathsf{b})$. Define a generic version of the monadic map. First define a kind-indexed type and then give equations for each of the type constants.

The standard mapping function is essentially determined by its type (each equation is more or less inevitable). Does this also hold for monadic maps?

Exercise 7. 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 two lists of type $\mathsf{List}\ \mathsf{a}_1$ and $\mathsf{List}\ \mathsf{a}_2$ and pairs corresponding elements producing a list of type $\mathsf{List}\ (\mathsf{a}_1, \mathsf{a}_2)$. Define a generic version of *zip* that works for all types of all kinds. *Hint:* the kind-indexed type of *zip* is essentially a three parameter variant of Map.

Exercise 8. The implementation of *fflatten* given in Section 2.4 has a quadratic running time since the computation of $x +\!\!+ y$ takes time proportional to the length of x. Using the well-known technique of *accumulation* [6] one can improve the running time to $O(n)$. This technique can be captured using a generic function, called a *right reduction*. Its kind-indexed type is given by

$$\begin{aligned}
&\textbf{type}\ \mathsf{Reducer}\{\![\star]\!\}\ \mathsf{t}\ \mathsf{x} &&= \mathsf{t} \to \mathsf{x} \to \mathsf{x}\\
&\textbf{type}\ \mathsf{Reducer}\{\![\kappa \to \nu]\!\}\ \mathsf{t}\ \mathsf{x} &&= \forall \mathsf{a}\,.\,\mathsf{Reducer}\{\![\kappa]\!\}\ \mathsf{a}\ \mathsf{x} \to \mathsf{Reducer}\{\![\nu]\!\}\ (\mathsf{t}\ \mathsf{a})\ \mathsf{x}
\end{aligned}$$

The second argument of type x is the accumulator. Fill in the definition of *reducer* and define *fflatten* in terms of *reducer*. Why is *reducer* called a right reduction? Also define its dual, a *left reduction*.

Exercise 9. The generic function *fsize* computes the size of a container type. Can you define a function that computes its height? *Hint:* you have to make use of the constructor case Con.

Exercise 10 (this may take some time). In Section 1.3 we have implemented a simple form of data compression. A more sophisticated scheme could use Huffman coding, emitting variable-length codes for the constructors. Implement a function that given a sample element of a data type counts the number of occurrences of each constructor. *Hint:* consider the type ConDescr. Which information is useful for your task?

3 Generic Haskell—Theory

This section highlights the theory underlying Generic Haskell (most of the material is taken from [22]).

We have already indicated that Generic Haskell takes a transformational approach to generic programming: a generic function is translated into a family of polymorphic functions. We will see in this section that this transformation can be phrased as an interpretation of the simply typed lambda calculus (types are simply typed lambda terms with kinds playing the rôle of types). To make this idea precise we switch from Haskell to the polymorphic lambda calculus. The simply typed lambda calculus and the polymorphic lambda calculus are introduced in Sections 3.1 and 3.2, respectively. Section 3.3 then shows how to specialize generic functions to instances of data types.

The polymorphic lambda calculus is the language of choice for the theoretical treatment of generic definitions. Though Haskell comes quite close to this ideal language, there is one fundamental difference between Haskell and (our presentation) of the polymorphic lambda calculus. In Haskell, each data declaration introduces a *new* type; two types are equal iff they have the same name. By contrast, in the polymorphic lambda calculus two types are equal iff they have the same structure. Section 3.4 explains how to adapt the technique of Section 3.3 to a type system based on name equivalence.

3.1 The simply typed lambda calculus as a type language

This section introduces the language of kinds and types that we will use in the sequel. The type system is essentially that of Haskell smoothing away some of its irregularities. We have see in Section 1.2 that Haskell offers one basic construct for defining new types: data type declarations.

$$\textbf{data}\ \mathsf{B}\ \mathsf{a}_1\ \ldots\ \mathsf{a}_m = K_1\ \mathsf{t}_{11}\ \ldots\ \mathsf{t}_{1m_1} \mid \cdots \mid K_n\ \mathsf{t}_{n1}\ \ldots\ \mathsf{t}_{nm_n}.$$

From a language design point of view the **data** construct is quite a beast as it 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 when we discuss the specialization of type-indexed values. Most of the definitions are taken from the excellent textbook by J. Mitchell [34].

Syntax 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	$\kappa, \nu \in \mathsf{Kind}$
type constants	$\mathsf{C}, \mathsf{D} \in \mathsf{Const}$
type variables	$\mathsf{a}, \mathsf{b} \in \mathsf{Var}$
type terms	$\mathsf{t}, \mathsf{u} \in \mathsf{Type}$

Note that we use Greek letters for kinds and Sans Serif letters for types.

Kind terms are formed according to the following grammar.

$$\begin{array}{rll} \kappa, \nu \in \mathsf{Kind} ::= & \star & \text{kind of types} \\ \mid & (\kappa \to \nu) & \text{function kind} \end{array}$$

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

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

$$\begin{array}{rll} \mathsf{t}, \mathsf{u} \in \mathsf{Type} ::= & \mathsf{C} & \text{type constant} \\ \mid & \mathsf{a} & \text{type variable} \\ \mid & (\Lambda \mathsf{a} :: \nu \,.\, \mathsf{t}) & \text{type abstraction} \\ \mid & (\mathsf{t}\ \mathsf{u}) & \text{type application} \end{array}$$

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 $\Lambda \mathsf{a} :: \nu \,.\, \mathsf{t}$ (especially if $\nu = \star$). Finally, we abbreviate nested abstractions $\Lambda \mathsf{a}_1 \,.\, \ldots \Lambda \mathsf{a}_m \,.\, \mathsf{t}$ by $\Lambda \mathsf{a}_1 \; \ldots \; \mathsf{a}_m \,.\, \mathsf{t}$.

The choice of Const, the set of type constants, is more or less arbitrary. However, in order to model Haskell's data declarations we assume that Const comprises at least the constants Char, Int, Unit, ':+:', ':∗:', and a family of fixed point operators:

$$\begin{aligned}\mathsf{Const} \supseteq \{ & \mathsf{Char} :: \star, \mathsf{Int} :: \star, \mathsf{Unit} :: \star, (:\!\ast\!:) :: \star \to \star \to \star, (:\!+\!:) :: \star \to \star \to \star \} \\ \cup \{ & \mathsf{Fix}_\kappa :: (\kappa \to \kappa) \to \kappa \mid \kappa \in \mathsf{Kind} \}.\end{aligned}$$

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. In the examples, we will often omit the kind annotation in Fix_κ.

In order to define 'well-kinded' type terms we need the notion of a context. A context is a finite set of kind assumptions of the form $\mathsf{a} :: \kappa$. It is convenient to view a context Γ as a finite map from type variables to kinds and write $dom(\Gamma)$ 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 κ such that $\Gamma \vdash \mathsf{t} :: \kappa$ is derivable using the rules depicted in Figure 2.

$$\frac{}{\Gamma \vdash \mathsf{C} :: \mathsf{Const}(\mathsf{C})} \quad (\textsc{t-const})$$

$$\frac{}{\Gamma \vdash \mathsf{a} :: \Gamma(\mathsf{a})} \quad (\textsc{t-var})$$

$$\frac{\Gamma, \mathsf{a} :: \nu \vdash \mathsf{t} :: \kappa}{\Gamma \vdash (\Lambda \mathsf{a} :: \nu \,.\, \mathsf{t}) :: (\nu \to \kappa)} \quad (\textsc{t-}\!\to\!\textsc{-intro})$$

$$\frac{\Gamma \vdash \mathsf{t} :: (\nu \to \kappa) \qquad \Gamma \vdash \mathsf{u} :: \nu}{\Gamma \vdash (\mathsf{t}\ \mathsf{u}) :: \kappa} \quad (\textsc{t-}\!\to\!\textsc{-elim})$$

Fig. 2. Kinding rules.

The equational proof system of the simply typed lambda calculus is given by the rules in Figure 3. Let $\mathcal{E}$ be a possibly empty set of equations between type terms. We write $\Gamma \vdash_{\mathcal{E}} \mathsf{t}_1 = \mathsf{t}_2 :: \kappa$ to mean that the type equation $\mathsf{t}_1 = \mathsf{t}_2$ is provable using the rules and the equations in $\mathcal{E}$.

The equational proof rules identify a recursive type $\mathsf{Fix}_\kappa\ \mathsf{t}$ and its unfolding $\mathsf{t}\ (\mathsf{Fix}_\kappa\ \mathsf{t})$. In general, there are two flavours of recursive types: equi-recursive types and iso-recursive types, see [12]. In the latter system $\mathsf{Fix}_\kappa\ \mathsf{t}$ and $\mathsf{t}\ (\mathsf{Fix}_\kappa\ \mathsf{t})$ must only be isomorphic rather than equal. The subsequent development is largely independent of this design choice. We use equi-recursive types because they simplify the presentation somewhat.

$$\frac{}{\Gamma \vdash ((\Lambda \mathsf{a} :: \nu \,.\, \mathsf{t})\ \mathsf{u} = \mathsf{t}[\mathsf{a} := \mathsf{u}]) :: \kappa}\ (\text{T-}\beta)$$

$$\frac{\mathsf{a}\text{ not free in }\mathsf{t}}{\Gamma \vdash (\Lambda \mathsf{a} :: \nu \,.\, \mathsf{t}\ \mathsf{a} = \mathsf{t}) :: (\nu \to \kappa)}\ (\text{T-}\eta)$$

$$\frac{}{\Gamma \vdash (\mathsf{Fix}_\kappa\ \mathsf{t} = \mathsf{t}\ (\mathsf{Fix}_\kappa\ \mathsf{t})) :: \kappa}\ (\text{T-FIX})$$

Fig. 3. Equational proof rules (the usual 'logical' rules for reflexivity, symmetry, transitivity, and congruence are omitted).

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

$$\mathsf{Fix}\ (\Lambda \mathsf{B} \,.\, \Lambda \mathsf{a}_1\ \ldots\ \mathsf{a}_m \,.\, (\mathsf{t}_{11} \mathbin{:\!*\!:} \cdots \mathbin{:\!*\!:} \mathsf{t}_{1m_1}) \mathbin{:\!+\!:} \cdots \mathbin{:\!+\!:} (\mathsf{t}_{n1} \mathbin{:\!*\!:} \cdots \mathbin{:\!*\!:} \mathsf{t}_{nm_n})),$$

where $\mathsf{t}_1 \mathbin{:\!*\!:} \cdots \mathbin{:\!*\!:} \mathsf{t}_k = \mathsf{Unit}$ 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

$$\textbf{data}\ \mathsf{List}\ \mathsf{a} = \mathit{Nil} \mid \mathit{Cons}\ \mathsf{a}\ (\mathsf{List}\ \mathsf{a})$$

is translated to

$$\mathsf{Fix}\ (\Lambda \mathsf{List} \,.\, \Lambda \mathsf{a} \,.\, \mathsf{Unit} \mathbin{:\!+\!:} \mathsf{a} \mathbin{:\!*\!:} \mathsf{List}\ \mathsf{a}).$$

Interestingly, the representation of regular types such as List can be improved by applying a technique called *lambda-dropping* [11]: if $\mathsf{Fix}\ (\Lambda \mathsf{f} \,.\, \Lambda \mathsf{a} \,.\, \mathsf{t})$ is regular, then it is equivalent to $\Lambda \mathsf{a} \,.\, \mathsf{Fix}\ (\Lambda \mathsf{b} \,.\, \mathsf{t}[\mathsf{f}\ \mathsf{a} := \mathsf{b}])$ where $\mathsf{t}[\mathsf{t}_1 := \mathsf{t}_2]$ denotes the type term, in which all occurrences of t_1 are replaced by t_2. For instance, the λ-dropped version of $\mathsf{Fix}\ (\Lambda \mathsf{List} \,.\, \Lambda \mathsf{a} \,.\, \mathsf{Unit} \mathbin{:\!+\!:} \mathsf{a} \mathbin{:\!*\!:} \mathsf{List}\ \mathsf{a})$ is $\Lambda \mathsf{a} \,.\, \mathsf{Fix}\ (\Lambda \mathsf{b} \,.\, \mathsf{Unit} \mathbin{:\!+\!:} \mathsf{a} \mathbin{:\!*\!:} \mathsf{b})$. The λ-dropped version employs the fixed point operator at kind $\star$ whereas the original, the so-called 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, [8].

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. An applicative structure is similar to an algebra, in that we need a set of 'values' for each kind and an interpretation for each type constant. Additionally, we have to give meaning to type abstraction and type application.

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

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

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

Definition 2. *An applicative structure* $\mathcal{A} = (\mathbf{A}, \mathbf{app}, \mathbf{const})$ *is extensional, if* $\forall \phi_1, \phi_2 \in \mathbf{A}^{\kappa \to \nu} \,.\, (\forall \alpha \in \mathbf{A}^{\kappa} \,.\, \mathbf{app}\ \phi_1\ \alpha = \mathbf{app}\ \phi_2\ \alpha) \supset \phi_1 = \phi_2$.

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 $(\mathsf{Type}, \mathbf{app}, \mathbf{const})$ may then be defined by letting $\mathsf{Type}^{\kappa} = \{\, \mathsf{t} \mid \Gamma \vdash \mathsf{t} :: \kappa$ for some finite $\Gamma \subseteq \mathcal{H} \,\}$, $\mathbf{app}\ \mathsf{t}\ \mathsf{u} = (\mathsf{t}\ \mathsf{u})$, and $\mathbf{const}(\mathsf{C}) = \mathsf{C}$.

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 mapping from type variables to values. If Γ is a context, then we say η satisfies Γ if $\eta(\mathsf{a}) \in \mathbf{A}^{\Gamma(\mathsf{a})}$ for every $\mathsf{a} \in \mathit{dom}(\Gamma)$. If η is an environment, then $\eta(\mathsf{a} := \alpha)$ is the environment mapping a to α and b to $\eta(\mathsf{b})$ for b different from a.

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

$$\begin{aligned}
&\mathcal{A}[\![\Gamma \vdash \mathsf{C} :: \mathsf{Const}(\mathsf{C})]\!]\eta = \mathbf{const}(\mathsf{C}) \\
&\mathcal{A}[\![\Gamma \vdash \mathsf{a} :: \Gamma(\mathsf{a})]\!]\eta \quad\;\; = \eta(\mathsf{a}) \\
&\mathcal{A}[\![\Gamma \vdash (\Lambda \mathsf{a} :: \nu \,.\, \mathsf{t}) :: (\nu \to \kappa)]\!]\eta \\
&\qquad\qquad\qquad\qquad = \textit{the unique } \phi \in \mathbf{A}^{\nu \to \kappa} \textit{ such that for all } \alpha \in \mathbf{A}^{\nu} \\
&\qquad\qquad\qquad\qquad\quad\; \mathbf{app}_{\nu,\kappa}\ \phi\ \alpha = \mathcal{A}[\![\Gamma, \mathsf{a} :: \nu \vdash \mathsf{t} :: \kappa]\!]\eta(\mathsf{a} := \alpha) \\
&\mathcal{A}[\![\Gamma \vdash (\mathsf{t}\ \mathsf{u}) :: \kappa]\!]\eta \qquad = \mathbf{app}_{\nu,\kappa}\ (\mathcal{A}[\![\Gamma \vdash \mathsf{t} :: (\nu \to \kappa)]\!]\eta)\ (\mathcal{A}[\![\Gamma \vdash \mathsf{u} :: \nu]\!]\eta)
\end{aligned}$$

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. Then we define the equivalence class of types $[\mathsf{t}] = \{\, T' \mid$

$\Gamma \vdash_{\mathcal{E}} \mathsf{t} = T' :: \kappa$ for some finite $\Gamma \subseteq \mathcal{H}\}$ and let $\mathsf{Type}^{\kappa}/\mathcal{E} = \{[\mathsf{t}] \mid \mathsf{t} \in \mathsf{Type}^{\kappa}\}$, $(\mathbf{app}/\mathcal{E})\ [\mathsf{t}]\ [\mathsf{u}] = [\mathsf{t}\ \mathsf{u}]$, and $(\mathbf{const}/\mathcal{E})\ (\mathsf{C}) = [\mathsf{C}]$. Then the applicative structure $(\mathsf{Type}/\mathcal{E}, \mathbf{app}/\mathcal{E}, \mathbf{const}/\mathcal{E})$ 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* $\mathcal{A} = (\mathbf{A}, \mathbf{app}, \mathbf{const})$ *satisfies the combinatory model condition if for all kinds* κ, ν *and* μ *there exist elements* $\mathbf{K}_{\kappa,\nu} \in \mathbf{A}^{\kappa\to\nu\to\kappa}$ *and* $\mathbf{S}_{\kappa,\nu,\mu} \in \mathbf{A}^{(\kappa\to\nu\to\mu)\to(\kappa\to\nu)\to\kappa\to\mu}$ *such that*

$$\begin{aligned} \mathbf{app}\ (\mathbf{app}\ \mathbf{K}\ \mathsf{a})\ \mathsf{b} &= \mathsf{a} \\ \mathbf{app}\ (\mathbf{app}\ (\mathbf{app}\ \mathbf{S}\ \mathsf{a})\ \mathsf{b})\ \mathsf{c} &= \mathbf{app}\ (\mathbf{app}\ \mathsf{a}\ \mathsf{c})\ (\mathbf{app}\ \mathsf{b}\ \mathsf{c}) \end{aligned}$$

for all a, b *and* c *of the appropriate kinds.*

3.2 The polymorphic lambda calculus

We have seen in Section 1.3 that instances of generic functions require first-class polymorphism. For instance, *eqGRose* 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 [14] (also known as $F\omega$) both for defining and for specializing type-indexed 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 [39].

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	$\mathsf{r}, \mathsf{s} \in \mathsf{Scheme}$
individual constants	$c, d \in \mathit{const}$
individual variables	$a, b \in \mathit{var}$
terms	$t, u \in \mathit{term}$

We use Roman letters for terms.

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

$\mathsf{r}, \mathsf{s} \in \mathsf{Scheme} ::= \mathsf{t}$	type term
$\mid\ (\mathsf{r} \to \mathsf{s})$	function type
$\mid\ (\forall \mathsf{a} :: \nu\ .\ \mathsf{s})$	polymorphic type

A pseudo-type scheme s is called a type scheme if there is a context Γ and a kind κ such that $\Gamma \vdash \mathsf{s} :: \kappa$ is derivable using the rules listed in Figures 2 and 4.

Pseudo-terms are given by the grammar

$t, u \in \mathit{term} ::= c$	constant
$\mid\ a$	variable
$\mid\ (\lambda a :: \mathsf{s}\ .\ t)$	abstraction
$\mid\ (t\ u)$	application
$\mid\ (\lambda \mathsf{a} :: \nu\ .\ t)$	universal abstraction
$\mid\ (t\ \mathsf{r})$	universal application.

$$\frac{\Gamma \vdash \mathsf{r} :: \star \qquad \Gamma \vdash \mathsf{s} :: \star}{\Gamma \vdash (\mathsf{r} \to \mathsf{s}) :: \star} \;(\textsc{t-fun})$$

$$\frac{\Gamma, \mathsf{a} :: \nu \vdash \mathsf{s} :: \star}{\Gamma \vdash (\forall \mathsf{a} :: \nu \,.\, \mathsf{s}) :: \star} \;(\textsc{t-all})$$

Fig. 4. Additional kinding rules for type schemes.

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

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

and suitable functions for each of the other type constants C in $\mathit{dom}(\mathsf{Const})$ (such as *Unit* for 'Unit', *Inl*, *Inr*, and **case** for ':+:', and *outl*, *outr*, and (_ :*: _) for ':*:'). To improve readability we will usually omit the type argument of *fix*.

To give the typing rules we have to extend the notion of context. A context is a finite set of kind assumptions $\mathsf{a} :: \kappa$ and type assumptions $a :: \mathsf{s}$. We say a context Γ is closed if Γ is either empty, or if $\Gamma = \Gamma_1, \mathsf{a} :: \kappa$ with Γ_1 closed, or if $\Gamma = \Gamma_1, a :: \mathsf{s}$ with Γ_1 closed and $\mathit{free}(\mathsf{s}) \subseteq \mathit{dom}(\Gamma_1)$. In the following we assume that contexts are closed. This restriction is necessary to prevent non-sensible terms such as $a :: \mathsf{a} \vdash \lambda\mathsf{a} :: \star \,.\, a$ 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 $\Gamma \vdash t :: \mathsf{s}$ is derivable using the typing rules depicted in Figure 5. 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 6. When we discuss the specialization of type-indexed 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 the entities Type^{κ}, $[\mathsf{t}]$ and $\mathsf{Type}^{\kappa}/\mathcal{E}$ we define $\mathsf{Scheme}^{\kappa} = \{\, \mathsf{s} \mid \Gamma \vdash \mathsf{s} :: \kappa \text{ for some finite } \Gamma \subseteq \mathcal{H} \,\}$, the equivalence class $[\mathsf{s}] = \{\, \mathsf{s}' \mid \Gamma \vdash_{\mathcal{E}} \mathsf{s} = \mathsf{s}' :: \kappa \text{ for some finite } \Gamma \subseteq \mathcal{H} \,\}$, $\mathsf{Scheme}^{\kappa}/\mathcal{E} = \{\, [\mathsf{s}] \mid \mathsf{s} \in \mathsf{Scheme}^{\kappa} \,\}$, and $\mathit{Term}^{\mathsf{s}} = \{\, t \mid \Gamma \vdash t :: \mathsf{s} \text{ for some finite } \Gamma \subseteq \mathcal{H} \,\}$, $[t] = \{\, t' \mid \Gamma \vdash_{\mathcal{E}} t = t' :: \mathsf{s} \text{ for some finite } \Gamma \subseteq \mathcal{H} \,\}$, and $\mathit{Term}^{\mathsf{s}}/\mathcal{E} = \{\, [t] \mid t \in \mathit{Term}^{\mathsf{s}} \,\}$. Note that $[\mathsf{s}_1] = [\mathsf{s}_2]$ implies $\mathit{Term}^{\mathsf{s}_1} = \mathit{Term}^{\mathsf{s}_2}$ because of rule (CONV).

3.3 Specializing type-indexed values

This section is concerned with the specialization of type-indexed values to concrete instances of data types. We have seen in Section 2.1 that the structure

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

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

$$\frac{\mathit{\Gamma}, a :: \mathsf{s} \vdash t :: \mathsf{r}}{\mathit{\Gamma} \vdash (\lambda a :: \mathsf{s} \,.\, t) :: (\mathsf{s} \to \mathsf{r})} \ (\to\textsc{-intro})$$

$$\frac{\mathit{\Gamma} \vdash t :: (\mathsf{s} \to \mathsf{r}) \qquad \mathit{\Gamma} \vdash u :: \mathsf{s}}{\mathit{\Gamma} \vdash (t\ u) :: \mathsf{r}} \ (\to\textsc{-elim})$$

$$\frac{\mathit{\Gamma}, \mathsf{a} :: \nu \vdash t :: \mathsf{s}}{\mathit{\Gamma} \vdash (\lambda \mathsf{a} :: \nu \,.\, t) :: (\forall \mathsf{a} :: \nu \,.\, \mathsf{s})} \ (\forall\textsc{-intro})$$

$$\frac{\mathit{\Gamma} \vdash t :: (\forall \mathsf{a} :: \nu \,.\, \mathsf{s}) \qquad \mathit{\Gamma} \vdash \mathsf{r} :: \nu}{\mathit{\Gamma} \vdash (t\ \mathsf{r}) :: \mathsf{s}[\mathsf{a} := \mathsf{r}]} \ (\forall\textsc{-elim})$$

$$\frac{\mathit{\Gamma} \vdash t :: \mathsf{r} \qquad \mathit{\Gamma} \vdash (\mathsf{r} = \mathsf{s}) :: \star}{\mathit{\Gamma} \vdash t :: \mathsf{s}} \ (\textsc{conv})$$

Fig. 5. Typing rules.

$$\frac{}{\mathit{\Gamma} \vdash ((\lambda a :: \mathsf{s} \,.\, t)\ u = t[a := u]) :: \mathsf{r}} \ (\beta)$$

$$\frac{a \text{ not free in } t}{\mathit{\Gamma} \vdash (\lambda a :: \mathsf{s} \,.\, t\ a = t) :: (\mathsf{s} \to \mathsf{r})} \ (\eta)$$

$$\frac{}{\mathit{\Gamma} \vdash ((\lambda \mathsf{a} :: \nu \,.\, t)\ \mathsf{r} = t[\mathsf{a} := \mathsf{r}]) :: \mathsf{s}[\mathsf{a} := \mathsf{r}]} \ (\beta)_\forall$$

$$\frac{\mathsf{a} \text{ not free in } t}{\mathit{\Gamma} \vdash (\lambda \mathsf{a} :: \nu \,.\, t\ \mathsf{a} = t) :: (\forall \mathsf{a} :: \nu \,.\, \mathsf{s})} \ (\eta)_\forall$$

$$\frac{}{\mathit{\Gamma} \vdash (\mathit{fix}\ \mathsf{r}\ f = f\ (\mathit{fix}\ \mathsf{r}\ f)) :: \mathsf{r}} \ (\textsc{fix})$$

Fig. 6. Equational proof rules (the usual 'logical' rules for reflexivity, symmetry, transitivity, and congruence are omitted).

of each instance of $map\{\![\mathsf{t}]\!\}$ 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 type-indexed values. In fact, the process of specialization can be phrased as an interpretation of the simply typed lambda calculus. The generic programmer specifies the interpretation of type constants. Given this information the meaning of a type term—that is, the specialization of a type-indexed 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 $\Lambda \mathsf{a} \,.\, \mathsf{List}\ (\mathsf{List}\ \mathsf{a})$. The instance map_{Matrix} is given by

$$\begin{aligned}
&map_{\mathsf{Matrix}} :: \forall \mathsf{a}_1\ \mathsf{a}_2 \,.\, (\mathsf{a}_1 \to \mathsf{a}_2) \to (\mathsf{Matrix}\ \mathsf{a}_1 \to \mathsf{Matrix}\ \mathsf{a}_2)\\
&map_{\mathsf{Matrix}} = \lambda \mathsf{a}_1\ \mathsf{a}_2 .\ \lambda map_{\mathsf{a}} :: (\mathsf{a}_1 \to \mathsf{a}_2) \,.\, map_{\mathsf{List}}\ (\mathsf{List}\ \mathsf{a}_1)\ (\mathsf{List}\ \mathsf{a}_2)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (map_{\mathsf{List}}\ \mathsf{a}_1\ \mathsf{a}_2\ map_{\mathsf{a}}).
\end{aligned}$$

The specialization of the type application $\mathsf{t} = \mathsf{List}\ \mathsf{a}$ is given by the lambda term $t = map_{\mathsf{List}}\ \mathsf{a}_1\ \mathsf{a}_2\ map_{\mathsf{a}}$, which is a combination of universal application and value application. Likewise, the specialization of the application $\mathsf{List}\ \mathsf{t}$ is given by $map_{\mathsf{List}}\ (\mathsf{List}\ \mathsf{a}_1)\ (\mathsf{List}\ \mathsf{a}_2)\ t$. Consequently, 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 $(\mathsf{s}_1, \mathsf{s}_2; map_{\mathsf{s}})$ where s_1 and s_2 are type schemes of some kind, say, κ and map_{s} is a value term of type $\mathsf{Map}\{\![\kappa]\!\}\ \mathsf{s}_1\ \mathsf{s}_2$. 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. Possible identities include $outl\ (t, u) = t$ and $outr\ (t, u) = u$. The applicative structure $\mathcal{M} = (\mathbf{M}, \mathbf{app}, \mathbf{const})$ is then given by

$$\begin{aligned}
&\mathbf{M}^{\kappa} \qquad\ = ([\mathsf{s}_1], [\mathsf{s}_2] \in \mathsf{Scheme}^{\kappa}/\mathcal{E};\ Term^{\mathsf{Map}\{\![\kappa]\!\}\ \mathsf{s}_1\ \mathsf{s}_2}/\mathcal{E})\\
&\mathbf{app}_{\kappa,\nu}\ ([\mathsf{r}_1], [\mathsf{r}_2]; [t])\ ([\mathsf{s}_1], [\mathsf{s}_2]; [u])\\
&\qquad\quad\ = ([\mathsf{r}_1\ \mathsf{s}_1], [\mathsf{r}_2\ \mathsf{s}_2]; [t\ \mathsf{s}_1\ \mathsf{s}_2\ u])\\
&\mathbf{const}(\mathsf{C}) = ([\mathsf{C}], [\mathsf{C}]; [map\{\![\mathsf{C}]\!\}]).
\end{aligned}$$

Note that the semantic application function **app** uses both the type and the value component of its second argument. It is instructive to check that the resulting term $t\ \mathsf{s}_1\ \mathsf{s}_2\ u$ is indeed well-typed: t has type $\forall \mathsf{a}_1\ \mathsf{a}_2 \,.\, \mathsf{Map}\{\![\kappa]\!\}\ \mathsf{a}_1\ \mathsf{a}_2 \to \mathsf{Map}\{\![\nu]\!\}\ (\mathsf{r}_1\ \mathsf{a}_1)\ (\mathsf{r}_2\ \mathsf{a}_2)$, s_1 and s_2 have kind κ, and u has type $\mathsf{Map}\{\![\kappa]\!\}\ \mathsf{s}_1\ \mathsf{s}_2$. It is important to note that the definitions of $\mathsf{Map}\{\![\kappa \to \nu]\!\}$ and **app** go hand in hand. This explains, in particular, why the definition of $\mathsf{Poly}\{\![\kappa \to \nu]\!\}$ is fixed for function kinds.

Now, does $\mathcal{M}$ also constitute a model? To this end we have to show that $\mathcal{M}$ is extensional and that it 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)

$$\begin{aligned}
\mathbf{K}_{\kappa,\nu} &= ([\mathsf{K}_{\kappa,\nu}], [\mathsf{K}_{\kappa,\nu}]; [\lambda \mathsf{a}_1\ \mathsf{a}_2 \,.\, \lambda map_{\mathsf{a}} \,.\, \lambda \mathsf{b}_1\ \mathsf{b}_2 \,.\, \lambda map_{\mathsf{b}} \,.\, map_{\mathsf{a}}]) \\
\mathbf{S}_{\kappa,\nu,\mu} &= ([\mathsf{S}_{\kappa,\nu,\mu}], [\mathsf{S}_{\kappa,\nu,\mu}]; \\
&\qquad [\lambda \mathsf{a}_1\ \mathsf{a}_2 \,.\, \lambda map_{\mathsf{a}} \,.\, \lambda \mathsf{b}_1\ \mathsf{b}_2 \,.\, \lambda map_{\mathsf{b}} \,.\, \lambda \mathsf{c}_1\ \mathsf{c}_2 \,.\, \lambda map_{\mathsf{c}} \,. \\
&\qquad\quad (map_{\mathsf{a}}\ \mathsf{c}_1\ \mathsf{c}_2\ map_{\mathsf{c}})\ (\mathsf{b}_1\ \mathsf{c}_1)\ (\mathsf{b}_2\ \mathsf{c}_2)\ (map_{\mathsf{b}}\ \mathsf{c}_1\ \mathsf{c}_2\ map_{\mathsf{c}})])
\end{aligned}$$

where K and S are given by

$$\begin{aligned}
\mathsf{K}_{\kappa,\nu} &= \Lambda \mathsf{a} :: \kappa \,.\, \Lambda \mathsf{b} :: \nu \,.\, \mathsf{a} \\
\mathsf{S}_{\kappa,\nu,\mu} &= \Lambda \mathsf{a} :: (\kappa \to \nu \to \mu) \,.\, \Lambda \mathsf{b} :: (\kappa \to \nu) \,.\, \Lambda \mathsf{c} :: \kappa \,.\, (\mathsf{a}\ \mathsf{c})\ (\mathsf{b}\ \mathsf{c}).
\end{aligned}$$

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

It remains to provide interpretations for the fixed point operators Fix_κ. The definition is essentially the same for all type-indexed values. This is why the generic programmer need not supply instances for Fix_κ by hand. Here is the definition of $map\{\!|\mathsf{Fix}_\kappa|\!\}$.

$$\begin{aligned}
map\{\!|\mathsf{Fix}_\kappa|\!\} = \lambda \mathsf{f}_1\ \mathsf{f}_2 \,.\, \lambda map_{\mathsf{f}} :: (\mathsf{Map}\{\![\kappa \to \kappa]\!\}\ \mathsf{f}_1\ \mathsf{f}_2) \,. \\
fix\ (map_{\mathsf{f}}\ (\mathsf{Fix}_\kappa\ \mathsf{f}_1)\ (\mathsf{Fix}_\kappa\ \mathsf{f}_2))
\end{aligned}$$

Note that $map\{\!|\mathsf{Fix}_\kappa|\!\}$ essentially equals fix—if we ignore type abstractions and type applications. Let us check that the definition of $map\{\!|\mathsf{Fix}_\kappa|\!\}$ is well-typed. The universal application $map_{\mathsf{f}}\ (\mathsf{Fix}_\kappa\ \mathsf{f}_1)\ (\mathsf{Fix}_\kappa\ \mathsf{f}_2)$ has type

$$\mathsf{Map}\{\![\kappa]\!\}\ (\mathsf{Fix}_\kappa\ \mathsf{f}_1)\ (\mathsf{Fix}_\kappa\ \mathsf{f}_2) \to \mathsf{Map}\{\![\kappa]\!\}\ (\mathsf{f}_1\ (\mathsf{Fix}_\kappa\ \mathsf{f}_1))\ (\mathsf{f}_2\ (\mathsf{Fix}_\kappa\ \mathsf{f}_2)).$$

Since we have $\mathsf{Fix}_\kappa\ \mathsf{f}_i = \mathsf{f}_i\ (\mathsf{Fix}_\kappa\ \mathsf{f}_i)$, we can use rule (CONV) to infer the type $\mathsf{Map}\{\![\kappa]\!\}\ (\mathsf{Fix}_\kappa\ \mathsf{f}_1)\ (\mathsf{Fix}_\kappa\ \mathsf{f}_2) \to \mathsf{Map}\{\![\kappa]\!\}\ (\mathsf{Fix}_\kappa\ \mathsf{f}_1)\ (\mathsf{Fix}_\kappa\ \mathsf{f}_2)$. Consequently, $fix\ (map_{\mathsf{f}}\ (\mathsf{Fix}_\kappa\ \mathsf{f}_1)\ (\mathsf{Fix}_\kappa\ \mathsf{f}_2))$ has type $\mathsf{Map}\{\![\kappa]\!\}\ (\mathsf{Fix}_\kappa\ \mathsf{f}_1)\ (\mathsf{Fix}_\kappa\ \mathsf{f}_2)$ as desired.

Now, let us turn to the general case of generic functions. The definitions for arbitrary type-indexed values are very similar to the ones for map. The applicative structure $\mathcal{P} = (\mathbf{P}, \mathbf{app}, \mathbf{const})$ induced by the type-indexed value $poly\{\!|\mathsf{t} :: \kappa|\!\} :: \mathsf{Poly}\{\![\kappa]\!\}\ \mathsf{t}\ \ldots\ \mathsf{t}$ is given by

$$\begin{aligned}
\mathbf{P}^\kappa &= ([\mathsf{s}_1], \ldots, [\mathsf{s}_n] \in \mathsf{Scheme}^\kappa/\mathcal{E};\ Term^{\mathsf{Poly}\{\![\kappa]\!\}\ \mathsf{s}_1\ \ldots\ \mathsf{s}_n}/\mathcal{E}) \\
\mathbf{app}_{\kappa,\nu}\ & ([\mathsf{r}_1], \ldots, [\mathsf{r}_n]; [t])\ ([\mathsf{s}_1], \ldots, [\mathsf{s}_n]; [u]) \\
&= ([\mathsf{r}_1\ \mathsf{s}_1], \ldots, [\mathsf{r}_n\ \mathsf{s}_n]; [t\ \mathsf{s}_1\ \ldots\ \mathsf{s}_n\ u]) \\
\mathbf{const}(\mathsf{c}) &= ([\mathsf{c}], \ldots, [\mathsf{c}]; [poly\{\!|\mathsf{c}|\!\}]).
\end{aligned}$$

where $poly\{\!|\mathsf{Fix}_\kappa|\!\}$ is defined

$$\begin{aligned}
poly\{\!|\mathsf{Fix}_\kappa|\!\} = \lambda \mathsf{f}_1\ \ldots\ \mathsf{f}_n \,.\, \lambda poly_{\mathsf{f}} :: (\mathsf{Poly}\{\![\kappa \to \kappa]\!\}\ \mathsf{f}_1\ \ldots\ \mathsf{f}_n) \,. \\
fix\ (poly_{\mathsf{f}}\ (\mathsf{Fix}_\kappa\ \mathsf{f}_1)\ \ldots\ (\mathsf{Fix}_\kappa\ \mathsf{f}_n)).
\end{aligned}$$

Three remarks are in order. First, the value domain $\mathbf{P}^\kappa$ 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 [35]. Second, if t is a closed type term, then $\mathcal{P}[\![\emptyset \vdash \mathsf{t}::\kappa]\!]\eta$ is of the form $([\mathsf{t}], \ldots, [\mathsf{t}]; [\mathit{poly}\{\![\mathsf{t}]\!\}])$ where $\mathit{poly}\{\![\mathsf{t}]\!\}$ is the desired instance. As an aside, note that this is in agreement with *poly*'s type signature $\mathit{poly}\{\![\mathsf{t} :: \kappa]\!\} :: \mathsf{Poly}\{\![\kappa]\!\}\ \mathsf{t}\ \ldots\ \mathsf{t}$. Third, a type-indexed 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. (Recall that we need type schemes to be able to assign types to instances of generic functions.)

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, for instance,

$$\begin{aligned}\mathit{poly}\{\![\Lambda\mathsf{a} \,.\, \mathsf{f}\ (\mathsf{g}\ \mathsf{a})]\!\} = \lambda\mathsf{a}_1\ \ldots\ \mathsf{a}_n \,.\, \lambda\mathit{poly}_{\mathsf{a}} \,.&\\ \mathit{poly}\{\![\mathsf{f}]\!\}\ (\mathsf{g}\ \mathsf{a}_1)\ \ldots\ (\mathsf{g}\ \mathsf{a}_n)\ (\mathit{poly}\{\![\mathsf{g}]\!\}\ \mathsf{a}_1\ \ldots\ \mathsf{a}_n\ \mathit{poly}_{\mathsf{a}}).&\end{aligned}$$

Writing type and function composition as usual this implies, in particular, that $\mathit{map}\{\![\mathsf{f} \cdot \mathsf{g}]\!\} = \mathit{map}\{\![\mathsf{f}]\!\} \cdot \mathit{map}\{\![\mathsf{g}]\!\}$. Perhaps surprisingly, this relationship holds for all type-indexed values, not only for mapping functions. A similar observation is that $\mathit{poly}\{\![\Lambda\mathsf{a} \,.\, \mathsf{a}]\!\} = \lambda\mathsf{a} \,.\, \lambda\mathit{poly}_{\mathsf{a}} \,.\, \mathit{poly}_{\mathsf{a}}$ for all type-indexed values. Abbreviating $\Lambda\mathsf{a} \,.\, \mathsf{a}$ by Id we have, in particular, that $\mathit{map}\{\![\mathsf{Id}]\!\} = \mathit{id}$. As an aside, note that these generic identities are not to be confused with the familiar functorial laws $\mathit{map}\{\![\mathsf{f}]\!\}\ \mathit{id} = \mathit{id}$ and $\mathit{map}\{\![\mathsf{f}]\!\}\ (\varphi \cdot \psi) = \mathit{map}\{\![\mathsf{f}]\!\}\ \varphi \cdot \mathit{map}\{\![\mathsf{f}]\!\}\ \psi$, which are base-level identities.

3.4 Bridging the gap

The polymorphic lambda calculus is the language of choice for the theoretical treatment of generic definitions as it offers rank-n polymorphism, which is required for specializing higher-order kinded data types. We additionally equipped it with a liberal notion of type equivalence so that we can interpret the type definition $\mathsf{List\ a} = \mathsf{Unit}\ {:+:}\ \mathsf{a}\ {:*:}\ \mathsf{List\ a}$ as an equality rather than as an isomorphism.

Haskell—or rather, extensions of Haskell come quite close to this ideal language. The Glasgow Haskell Compiler, GHC, [45], the Haskell B. Compiler, HBC, [3] and the Haskell interpreter Hugs [29] provide rank-2 type signatures. The latest version of the Glasgow Haskell Compiler, GHC 5.04, even supports general rank-n types. There is, however, one fundamental difference between Haskell and (our presentation) of the polymorphic lambda calculus: Haskell's notion of type equivalence is based on *name equivalence* while the polymorphic lambda calculus employs *structural equivalence*. In the sequel we explain the difference between the two views and show how to adapt the specialization to a type system that is based on name equivalence.

Generic representation types Consider again the Haskell data type of parametric lists:

$$\textbf{data}\ \mathsf{List\ a} = \mathit{Nil} \mid \mathit{Cons}\ \mathsf{a}\ (\mathsf{List\ a}).$$

We have modelled this declaration (see Section 3.1) by the type term

$$\mathsf{Fix}\ (\Lambda\mathsf{List}\,.\,\Lambda\mathsf{a}\,.\,\mathsf{Unit}\ {:}{+}{:}\ \mathsf{a}\ {:}{*}{:}\ \mathsf{List}\ \mathsf{a}).$$

Since the equivalence of type terms is based on structural equivalence, we have, in particular, that $\mathsf{List\ a} = \mathsf{Unit}\ {:}{+}{:}\ \mathsf{a}\ {:}{*}{:}\ \mathsf{List\ a}$ (using (T-FIX), see Figure 3). The specialization of generic values described in the previous section makes essential use of this fact: the List instance of *poly* given by (omitting type abstractions and type applications)

$$\mathit{fix}\ (\lambda \mathit{poly}_{\mathsf{List}}\,.\,\lambda \mathit{poly}_{\mathsf{a}}\,.\,\mathit{poly}_{:+:}\ \mathit{poly}_{\mathsf{Unit}}\ (\mathit{poly}_{:*:}\ \mathit{poly}_{\mathsf{a}}\ (\mathit{poly}_{\mathsf{List}}\ \mathit{poly}_{\mathsf{a}})))$$

only works under the proviso that $\mathsf{List\ a} = \mathsf{Unit}\ {:}{+}{:}\ \mathsf{a}\ {:}{*}{:}\ \mathsf{List\ a}$. Alas, in Haskell List a is not equivalent to Unit :+: a :*: List a as each **data** declaration introduces a new distinct type. Even the type Liste defined by

$$\textbf{data}\ \mathsf{Liste\ a} = \mathit{Vide} \mid \mathit{Constructeur}\ \mathsf{a}\ (\mathsf{Liste\ a})$$

is not equivalent to List though only the names of the data constructors are different. Furthermore, Haskell's **data** construct works with n-ary sums and products whereas generic definitions operate on binary sums and products. The bottom line of all this is that when generating instances we additionally have to introduce conversion functions which perform the impedance-matching. Fortunately, this can be done in a systematic way.

To begin with we introduce so-called *generic representation types*, which mediate between the two representations. For instance, the generic representation type for List, which we will call List°, is given by

$$\textbf{type}\ \mathsf{List}^\circ\ \mathsf{a} = \mathsf{Unit}\ {:}{+}{:}\ \mathsf{a}\ {:}{*}{:}\ \mathsf{List\ a}.$$

As to be expected our generic representation type constructors are just unit, binary sum and binary product. In particular, there is no recursion operator. We observe that List° is a non-recursive type synonym: List (not List°) appears on the right-hand side. So List° is not a recursive type; rather, it expresses the 'top layer' of a list structure.

The type constructor List° is (more or less) isomorphic to List. To make the isomorphism explicit, let us write functions that convert to and fro:

$$\begin{aligned}
&\mathit{from}_{\mathsf{List}} && :: \forall \mathsf{a}\,.\,\mathsf{List\ a} \to \mathsf{List}^\circ\ \mathsf{a}\\
&\mathit{from}_{\mathsf{List}}\ \mathit{Nil} && = \mathit{Inl}\ \mathit{Unit}\\
&\mathit{from}_{\mathsf{List}}\ (\mathit{Cons}\ x\ xs) && = \mathit{Inr}\ (x\ {:}{*}{:}\ xs)\\
&\mathit{to}_{\mathsf{List}} && :: \forall \mathsf{a}\,.\,\mathsf{List}^\circ\ \mathsf{a} \to \mathsf{List\ a}\\
&\mathit{to}_{\mathsf{List}}\ (\mathit{Inl}\ \mathit{Unit}) && = \mathit{Nil}\\
&\mathit{to}_{\mathsf{List}}\ (\mathit{Inr}\ (x\ {:}{*}{:}\ xs)) && = \mathit{Cons}\ x\ xs.
\end{aligned}$$

Though these are non-generic functions, it is not hard to generate them mechanically. That is what we turn our attention to now.

Since the generic definitions work with *binary* sums and *binary* products, algebraic data types with many constructors, each of which has many fields, must be encoded as nested uses of sum and product. There are many possible encodings. For concreteness, we use a simple *linear encoding*: for

$$\textbf{data}\ \mathsf{B}\ \mathsf{a}_1\ \ldots\ \mathsf{a}_m = K_1\ \mathsf{t}_{11}\ \ldots\ \mathsf{t}_{1m_1} \mid \cdots \mid K_n\ \mathsf{t}_{n1}\ \ldots\ \mathsf{t}_{nm_n}$$

we generate:

$$\textbf{type}\ \mathsf{B}^{\circ}\ \mathsf{a}_1\ \ldots\ \mathsf{a}_m = \Sigma\ (\Pi\ \mathsf{t}_{11}\ \ldots\ \mathsf{t}_{1m_1})\ \cdots\ (\Pi\ \mathsf{t}_{n1}\ \ldots\ \mathsf{t}_{nm_n})$$

where 'Σ' and 'Π' are defined

$$\Sigma\ \mathsf{t}_1\ \ldots\ \mathsf{t}_n = \begin{cases} \mathsf{t}_1 & \text{if } n = 1 \\ \mathsf{t}_1 \mathbin{:+:} \Sigma\ \mathsf{t}_2\ \ldots\ \mathsf{t}_n & \text{if } n > 1 \end{cases}$$

$$\Pi\ \mathsf{t}_1\ \ldots\ \mathsf{t}_n = \begin{cases} \mathsf{Unit} & \text{if } n = 0 \\ \mathsf{t}_1 & \text{if } n = 1 \\ \mathsf{t}_1 \mathbin{:*:} \Pi\ \mathsf{t}_2\ \ldots\ \mathsf{t}_n & \text{if } n > 1. \end{cases}$$

Note that this encoding corresponds closely to the scheme introduced in Section 3.1 except that here the argument types of the constructors are *not* recursively encoded. The conversion functions $\mathit{from}_{\mathsf{B}}$ and to_{B} are then given by

$$\begin{array}{ll}
\mathit{from}_{\mathsf{B}} & :: \forall \mathsf{a}_1\ \ldots\ \mathsf{a}_m\,.\,\mathsf{B}\ \mathsf{a}_1\ \ldots\ \mathsf{a}_m \to \mathsf{B}^{\circ}\ \mathsf{a}_1\ \ldots\ \mathsf{a}_m \\
\mathit{from}_{\mathsf{B}}\ (K_1\ x_{11}\ \ldots\ x_{1m_1}) & = \mathit{in}^n_1\ (\mathit{tuple}\ x_{11}\ \ldots\ x_{1m_1}) \\
\ldots & \\
\mathit{from}_{\mathsf{B}}\ (K_n\ x_{n1}\ \ldots\ x_{nm_n}) & = \mathit{in}^n_n\ (\mathit{tuple}\ x_{n1}\ \ldots\ x_{nm_n}) \\
\mathit{to}_{\mathsf{B}} & :: \forall \mathsf{a}_1\ \ldots\ \mathsf{a}_m\,.\,\mathsf{B}^{\circ}\ \mathsf{a}_1\ \ldots\ \mathsf{a}_m \to \mathsf{B}\ \mathsf{a}_1\ \ldots\ \mathsf{a}_m \\
\mathit{to}_{\mathsf{B}}\ (\mathit{in}^n_1\ (\mathit{tuple}\ x_{11}\ \ldots\ x_{1m_1})) & = K_1\ x_{11}\ \ldots\ x_{1m_1} \\
\ldots & \\
\mathit{to}_{\mathsf{B}}\ (\mathit{in}^n_n\ (\mathit{tuple}\ x_{n1}\ \ldots\ x_{nm_n})) & = K_n\ x_{n1}\ \ldots\ x_{nm_n}
\end{array}$$

where

$$\mathit{in}^n_i\ t = \begin{cases} t & \text{if } n = 1 \\ \mathit{Inl}\ t & \text{if } n > 1 \wedge i = 1 \\ \mathit{Inr}\ (\mathit{in}^{n-1}_{i-1}\ t) & \text{if } n > 1 \wedge i > 1 \end{cases}$$

$$\mathit{tuple}\ t_1\ \ldots\ t_n = \begin{cases} \mathit{Unit} & \text{if } n = 0 \\ t_1 & \text{if } n = 1 \\ (t_1 \mathbin{:*:} \mathit{tuple}\ t_2\ \ldots\ t_n) & \text{if } n > 1. \end{cases}$$

An alternative encoding, which is based on a binary sub-division scheme, is given in [18]. Most generic functions are insensitive to the translation of sums and products. Two notable exceptions are *encode* and *decodes*, for which the binary sub-division scheme is preferable (the linear encoding aggravates the compression rate).

Specializing generic values Assume for the sake of example that we want to specialize the generic functions *encode* and *decodes* introduced in Section 1.3 to the List data type. Recall the types of the generic values (here expressed as type synonyms):

$$\begin{aligned} &\mathbf{type}\ \mathsf{Encode}\ \mathsf{a}\ \ = \mathsf{a} \to \mathsf{Bin} \\ &\mathbf{type}\ \mathsf{Decodes}\ \mathsf{a} = \mathsf{Bin} \to (\mathsf{a}, \mathsf{Bin}). \end{aligned}$$

Since List° involves only the type constants Unit, ':+:' and ':∗:' (and the type variables List and a), we can easily specialize *encode* and *decodes* to $\mathsf{List}^{\circ}\ \mathsf{a}$: the instances have types $\mathsf{Encode}\ (\mathsf{List}^{\circ}\ \mathsf{a})$ and $\mathsf{Decodes}\ (\mathsf{List}^{\circ}\ \mathsf{a})$, respectively. However, we want to generate functions of type $\mathsf{Encode}\ (\mathsf{List}\ \mathsf{a})$ and $\mathsf{Decodes}\ (\mathsf{List}\ \mathsf{a})$. Now, we already know how to convert between $\mathsf{List}^{\circ}\ \mathsf{a}$ and $\mathsf{List}\ \mathsf{a}$. So it remains to lift $\mathit{from}_{\mathsf{List}}$ and $\mathit{to}_{\mathsf{List}}$ to functions of type $\mathsf{Encode}\ (\mathsf{List}\ \mathsf{a}) \to \mathsf{Encode}\ (\mathsf{List}^{\circ}\ \mathsf{a})$ and $\mathsf{Encode}\ (\mathsf{List}^{\circ}\ \mathsf{a}) \to \mathsf{Encode}\ (\mathsf{List}\ \mathsf{a})$. But this lifting is exactly what a mapping function does! In particular, since Encode and $\mathsf{Decodes}$ involve function types and we have to convert to and fro, we can use the embedding-projection maps of Section 2.3 for this purpose.

For *mapE* we have to package the two conversion functions into a single value:

$$\begin{aligned} &\mathit{conv}_{\mathsf{List}} :: \forall \mathsf{a} \,.\, \mathsf{EP}\ (\mathsf{List}\ \mathsf{a})\ (\mathsf{List}^{\circ}\ \mathsf{a}) \\ &\mathit{conv}_{\mathsf{List}} = \mathit{EP}\{\mathit{from} = \mathit{from}_{\mathsf{List}}, \mathit{to} = \mathit{to}_{\mathsf{List}}\}. \end{aligned}$$

Then $\mathit{encode}_{\mathsf{List}}$ and $\mathit{decodes}_{\mathsf{List}}$ are given by

$$\begin{aligned} &\mathit{encode}_{\mathsf{List}}\ \mathit{ena}\ \ = \mathit{to}\ (\mathit{mapE}\{\!|\mathsf{Encode}|\!\}\ \mathit{conv}_{\mathsf{List}})\ (\mathit{encode}\{\!|\mathsf{List}^{\circ}|\!\}\ \mathit{ena}) \\ &\mathit{decodes}_{\mathsf{List}}\ \mathit{dea} = \mathit{to}\ (\mathit{mapE}\{\!|\mathsf{Decodes}|\!\}\ \mathit{conv}_{\mathsf{List}})\ (\mathit{decodes}\{\!|\mathsf{List}^{\circ}|\!\}\ \mathit{dea}). \end{aligned}$$

Consider the definition of $\mathit{encode}_{\mathsf{List}}$. The specialization $\mathit{encode}\{\!|\mathsf{List}^{\circ}|\!\}\ \mathit{ena}$ yields a function of type $\mathsf{Encode}\ (\mathsf{List}^{\circ}\ \mathsf{a})$; the call $\mathit{to}\ (\mathit{mapE}\{\!|\mathsf{Encode}|\!\}\ \mathit{conv}_{\mathsf{List}})$ then converts this function into a value of type $\mathsf{Encode}\ (\mathsf{List}\ \mathsf{a})$ as desired.

In general, the translation proceeds as follows. For each generic definition we generate the following.

– A type synonym $\mathsf{Poly} = \mathsf{Poly}\{\![\star]\!\}$ for the type of the generic value.
– An embedding-projection map, $\mathit{mapE}\{\!|\mathsf{Poly}|\!\}$.
– Generic instances for Unit, ':+:', ':∗:' and possibly other primitive types.

For each data type declaration B we generate the following.

– A type synonym, B°, for B's generic representation type.
– An embedding-projection pair $\mathit{conv}_{\mathsf{B}}$ that converts between $\mathsf{B}\ \mathsf{a}_1\ \ldots\ \mathsf{a}_m$ and $\mathsf{B}^{\circ}\ \mathsf{a}_1\ \ldots\ \mathsf{a}_m$.

$$\begin{aligned} &\mathit{conv}_{\mathsf{B}} :: \forall \mathsf{a}_1\ \ldots\ \mathsf{a}_m \,.\, \mathsf{EP}\ (\mathsf{B}\ \mathsf{a}_1\ \ldots\ \mathsf{a}_m)\ (\mathsf{B}^{\circ}\ \mathsf{a}_1\ \ldots\ \mathsf{a}_m) \\ &\mathit{conv}_{\mathsf{B}} = \mathit{EP}\{\mathit{from} = \mathit{from}_{\mathsf{B}}, \mathit{to} = \mathit{to}_{\mathsf{B}}\} \end{aligned}$$

The instance of *poly* for type $\mathsf{B} :: \kappa$ is then given by (using Haskell syntax)

$$\begin{aligned}
&\mathit{poly}_{\mathsf{B}} :: \mathsf{Poly}\{\![\kappa]\!\}\ \mathsf{B}\ \ldots\ \mathsf{B}\\
&\mathit{poly}_{\mathsf{B}}\ \mathit{poly}_{\mathsf{a}_1}\ \ldots\ \mathit{poly}_{\mathsf{a}_m}\\
&\qquad = \mathit{to}\ (\mathit{mapE}\{\!|\mathsf{Poly}|\!\}\ \mathit{conv}_{\mathsf{B}}\ \ldots\ \mathit{conv}_{\mathsf{B}})\ (\mathit{poly}\{\!|\mathsf{B}^\circ|\!\}\ \mathit{poly}_{\mathsf{a}_1}\ \ldots\ \mathit{poly}_{\mathsf{a}_m}).
\end{aligned}$$

If $\mathsf{Poly}\{\![\kappa]\!\}\ \mathsf{B}\ \ldots\ \mathsf{B}$ has a rank of 2 or below, we can express $\mathit{poly}_{\mathsf{B}}$ directly in Haskell. Figures 7 and 8 show several examples of specializations all expressed in Haskell.

Generating embedding-projection maps We are in a peculiar situation: in order to specialize a generic value *poly* to some data type B, we have to specialize another generic value, namely, *mapE* to *poly*'s type Poly. This works fine if Poly like Encode only involves primitive types. So let us make this assumption for the moment. Here is a version of *mapE* tailored to Haskell's set of primitive types:

$$\begin{aligned}
&\mathit{mapE}\{\!|\mathsf{t} :: \kappa|\!\} &&:: \mathsf{MapE}\{\![\kappa]\!\}\ \mathsf{t}\ \mathsf{t}\\
&\mathit{mapE}\{\!|\mathsf{Char}|\!\} &&= \mathit{id}_E\\
&\mathit{mapE}\{\!|\mathsf{Int}|\!\} &&= \mathit{id}_E\\
&\mathit{mapE}\{\!|{\rightarrow}|\!\}\ \mathit{mapE}_{\mathsf{a}}\ \mathit{mapE}_{\mathsf{b}} &&= \mathit{mapE}_{\mathsf{a}} \rightarrow_E \mathit{mapE}_{\mathsf{b}}\\
&\mathit{mapE}\{\!|\mathsf{IO}|\!\}\ \mathit{mapE}_{\mathsf{a}} &&= \mathit{EP}\{\mathit{from} = \mathit{fmap}\ (\mathit{from}\ \mathit{mapE}_{\mathsf{a}}),\\
& && \qquad\quad \mathit{to} = \mathit{fmap}\ (\mathit{to}\ \mathit{mapE}_{\mathsf{a}})\}.
\end{aligned}$$

Note that in the last equation *mapE* falls back on the 'ordinary' mapping function *fmap*. (In Haskell, *fmap*, a method of the Functor class, is an overloaded version of *map*, which is confined to lists.) In fact, we can alternatively define

$$\mathit{mapE}\{\!|\mathsf{IO}|\!\} = \mathit{liftE}$$

where

$$\begin{aligned}
&\mathit{liftE} &&:: \forall \mathsf{f}\,.\,(\mathsf{Functor}\ \mathsf{f}) \Rightarrow \forall \mathsf{a}\ \mathsf{a}^\circ\,.\,\mathsf{EP}\ \mathsf{a}\ \mathsf{a}^\circ \rightarrow \mathsf{EP}\ (\mathsf{f}\ \mathsf{a})\ (\mathsf{f}\ \mathsf{a}^\circ)\\
&\mathit{liftE}\ \mathit{mapE}_{\mathsf{a}} &&= \mathit{EP}\{\mathit{from} = \mathit{fmap}\ (\mathit{from}\ \mathit{mapE}_{\mathsf{a}}), \mathit{to} = \mathit{fmap}\ (\mathit{to}\ \mathit{mapE}_{\mathsf{a}})\}.
\end{aligned}$$

Now, the $\mathsf{Poly} :: \pi$ instance of *mapE* is given by

$$\begin{aligned}
&\mathit{mapE}_{\mathsf{Poly}} &&:: \mathsf{MapE}\{\![\pi]\!\}\ \mathsf{Poly}\ \mathsf{Poly}\\
&\mathit{mapE}_{\mathsf{Poly}}\ \mathit{mapE}_{\mathsf{a}_1}\ \ldots\ \mathit{mapE}_{\mathsf{a}_k} &&= \mathit{mapE}\{\!|\mathsf{Poly}\ \mathsf{a}_1\ \ldots\ \mathsf{a}_k|\!\}\ \rho.
\end{aligned}$$

where $\rho = (\mathsf{a}_1 := \mathit{mapE}_{\mathsf{a}_1}, \ldots, \mathsf{a}_k := \mathit{mapE}_{\mathsf{a}_k})$ is an environment mapping type variables to terms. We use for the first time an explicit environment in order to be able to extend the definition to polymorphic types. Recall that the specialization of generic values does not work for polymorphic types. However, we allow polymorphic types to occur in the type signature of a generic value. Fortunately, we can extend *mapE* so that it works for universal quantification over types of kind $\star$ and kind $\star \rightarrow \star$.

$$\begin{aligned}
&\mathit{mapE}\{\!|\mathsf{C}|\!\}\ \rho &&= \mathit{mapE}\{\!|\mathsf{C}|\!\}\\
&\mathit{mapE}\{\!|\mathsf{a}|\!\}\ \rho &&= \rho(\mathsf{a})\\
&\mathit{mapE}\{\!|\mathsf{t}\ \mathsf{u}|\!\}\ \rho &&= (\mathit{mapE}\{\!|\mathsf{t}|\!\}\ \rho)\ (\mathit{mapE}\{\!|\mathsf{u}|\!\}\ \rho)\\
&\mathit{mapE}\{\!|\forall \mathsf{a} :: \star\,.\,\mathsf{t}|\!\}\ \rho &&= \mathit{mapE}\{\!|\mathsf{t}|\!\}\ \rho(\mathsf{a} := \mathit{id}_E)\\
&\mathit{mapE}\{\!|\forall \mathsf{f} :: \star \rightarrow \star\,.\,(\mathsf{Functor}\ \mathsf{f}) \Rightarrow \mathsf{t}|\!\}\ \rho &&= \mathit{mapE}\{\!|\mathsf{t}|\!\}\ \rho(\mathsf{f} := \mathit{liftE}).
\end{aligned}$$

```
{- binary encoding -}
type Encode a             = a → Bin
encode_Unit               :: Encode Unit
encode_Unit               = λUnit → []
encode_:+:                :: ∀a . Encode a → (∀b . Encode b → Encode (a :+: b))
encode_:+: encode_a encode_b = λs → case s of { Inl a → 0 : encode_a a;
                                                 Inr b → 1 : encode_b b }
encode_:*:                :: ∀a . Encode a → (∀b . Encode b → Encode (a :*: b))
encode_:*: encode_a encode_b = λ(a :*: b) → encode_a a ++ encode_b b
mapE_Encode               :: ∀a b . EP a b → EP (Encode a) (Encode b)
mapE_Encode m             = EP{from = λh → h · to m, to = λh → h · from m}
{- equality -}
type Equal a1 a2          = a1 → a2 → Bool
equal_Unit                :: Equal Unit Unit
equal_Unit                = λUnit Unit → True
equal_:+:                 :: ∀a1 a2 . Equal a1 a2 → (∀b1 b2 . Equal b1 b2
                                  → Equal (a1 :+: b1) (a2 :+: b2))
equal_:+: equal_a equal_b = λs1 s2 → case (s1, s2) of
                                  {(Inl a1, Inl a2) → equal_a a1 a2;
                                   (Inl a1, Inr b2) → False;
                                   (Inr b1, Inl a2) → False;
                                   (Inr b1, Inr b2) → equal_b b1 b2}
equal_:*:                 :: ∀a1 a2 . Equal a1 a2 → (∀b1 b2 . Equal b1 b2
                                  → Equal (a1 :*: b1) (a2 :*: b2))
equal_:*: equal_a equal_b = λ(a1 :*: b1) (a2 :*: b2) → equal_a a1 a2 ∧ equal_b b1 b2
mapE_Equal                :: ∀a1 b1 . EP a1 b1 → (∀a2 b2 . EP a2 b2
                                  → EP (Equal a1 a2) (Equal b1 b2))
mapE_Equal m1 m2          = EP{from = λh → λa1 a2 → h (to m1 a1) (to m2 a2),
                               to = λh → λb1 b2 → h (from m1 b1) (from m2 b2)}
{- generic representation types -}
type Maybe° a             = Unit :+: a
from_Maybe                :: ∀a . Maybe a → Maybe° a
from_Maybe Nothing        = Inl Unit
from_Maybe (Just a)       = Inr a
to_Maybe                  :: ∀a . Maybe° a → Maybe a
to_Maybe (Inl Unit)       = Nothing
to_Maybe (Inr a)          = Just a
conv_Maybe                :: ∀a . EP (Maybe a) (Maybe° a)
conv_Maybe                = EP{from = from_Maybe, to = to_Maybe}
```

Fig. 7. Specializing generic values in Haskell (part 1).

$$
\begin{array}{ll}
\textbf{type}\ \mathsf{List}^{\circ}\ \mathsf{a} & = \mathsf{Unit}\ {:}{+}{:}\ \mathsf{a}\ {:}{*}{:}\ \mathsf{List\ a} \\
\mathit{from}_{\mathsf{List}} & ::\ \forall \mathsf{a}\,.\,\mathsf{List\ a} \rightarrow \mathsf{List}^{\circ}\ \mathsf{a} \\
\mathit{from}_{\mathsf{List}}\ [\,] & = \mathit{Inl\ Unit} \\
\mathit{from}_{\mathsf{List}}\ (a : as) & = \mathit{Inr}\ (a\ {:}{*}{:}\ as) \\
\mathit{to}_{\mathsf{List}} & ::\ \forall \mathsf{a}\,.\,\mathsf{List}^{\circ}\ \mathsf{a} \rightarrow \mathsf{List\ a} \\
\mathit{to}_{\mathsf{List}}\ (\mathit{Inl\ Unit}) & = [\,] \\
\mathit{to}_{\mathsf{List}}\ (\mathit{Inr}\ (a\ {:}{*}{:}\ as)) & = a : as \\
\mathit{conv}_{\mathsf{List}} & ::\ \forall \mathsf{a}\,.\,\mathsf{EP}\ (\mathsf{List\ a})\ (\mathsf{List}^{\circ}\ \mathsf{a}) \\
\mathit{conv}_{\mathsf{List}} & = \mathit{EP}\{\mathit{from} = \mathit{from}_{\mathsf{List}}, \mathit{to} = \mathit{to}_{\mathsf{List}}\} \\
\textbf{type}\ \mathsf{GRose}^{\circ}\ \mathsf{f\ a} & = \mathsf{a}\ {:}{*}{:}\ \mathsf{f\ (GRose\ f\ a)} \\
\mathit{from}_{\mathsf{GRose}} & ::\ \forall \mathsf{f\ a}\,.\,\mathsf{GRose\ f\ a} \rightarrow \mathsf{GRose}^{\circ}\ \mathsf{f\ a} \\
\mathit{from}_{\mathsf{GRose}}\ (\mathit{GBranch}\ a\ ts) & = (a\ {:}{*}{:}\ ts) \\
\mathit{to}_{\mathsf{GRose}} & ::\ \forall \mathsf{f\ a}\,.\,\mathsf{GRose}^{\circ}\ \mathsf{f\ a} \rightarrow \mathsf{GRose\ f\ a} \\
\mathit{to}_{\mathsf{GRose}}\ (a\ {:}{*}{:}\ ts) & = \mathit{GBranch}\ a\ ts \\
\mathit{conv}_{\mathsf{GRose}} & ::\ \forall \mathsf{f\ a}\,.\,\mathsf{EP}\ (\mathsf{GRose\ f\ a})\ (\mathsf{GRose}^{\circ}\ \mathsf{f\ a}) \\
\mathit{conv}_{\mathsf{GRose}} & = \mathit{EP}\{\mathit{from} = \mathit{from}_{\mathsf{GRose}}, \mathit{to} = \mathit{to}_{\mathsf{GRose}}\}
\end{array}
$$

{- specializing binary encoding -}

$$
\begin{array}{ll}
\mathit{encode}_{\mathsf{Maybe}} & ::\ \forall \mathsf{a}\,.\,\mathsf{Encode\ a} \rightarrow \mathsf{Encode\ (Maybe\ a)} \\
\mathit{encode}_{\mathsf{Maybe}}\ \mathit{encode}_{\mathsf{a}} & = \mathit{to}\ (\mathit{mapE}_{\mathsf{Encode}}\ \mathit{conv}_{\mathsf{Maybe}})\ (\mathit{encode}_{:+:}\ \mathit{encode}_{\mathsf{Unit}}\ \mathit{encode}_{\mathsf{a}}) \\
\mathit{encode}_{\mathsf{List}} & ::\ \forall \mathsf{a}\,.\,\mathsf{Encode\ a} \rightarrow \mathsf{Encode\ (List\ a)} \\
\mathit{encode}_{\mathsf{List}}\ \mathit{encode}_{\mathsf{a}} & = \mathit{to}\ (\mathit{mapE}_{\mathsf{Encode}}\ \mathit{conv}_{\mathsf{List}})\ (\\
 & \qquad \mathit{encode}_{:+:}\ \mathit{encode}_{\mathsf{Unit}}\ (\mathit{encode}_{:*:}\ \mathit{encode}_{\mathsf{a}}\ (\mathit{encode}_{\mathsf{List}}\ \mathit{encode}_{\mathsf{a}}))) \\
\mathit{encode}_{\mathsf{GRose}} & ::\ \forall \mathsf{f}\,.\,(\forall \mathsf{b}\,.\,\mathsf{Encode\ b} \rightarrow \mathsf{Encode\ (f\ b)}) \\
 & \qquad\quad \rightarrow (\forall \mathsf{a}\,.\,\mathsf{Encode\ a} \rightarrow \mathsf{Encode\ (GRose\ f\ a)}) \\
\mathit{encode}_{\mathsf{GRose}}\ \mathit{encode}_{\mathsf{f}}\ \mathit{encode}_{\mathsf{a}} & \\
 & = \mathit{to}\ (\mathit{mapE}_{\mathsf{Encode}}\ \mathit{conv}_{\mathsf{GRose}})\ (\\
 & \qquad \mathit{encode}_{:*:}\ \mathit{encode}_{\mathsf{a}}\ (\mathit{encode}_{\mathsf{f}}\ (\mathit{encode}_{\mathsf{GRose}}\ \mathit{encode}_{\mathsf{f}}\ \mathit{encode}_{\mathsf{a}})))
\end{array}
$$

{- specializing equality -}

$$
\begin{array}{ll}
\mathit{equal}_{\mathsf{Maybe}} & ::\ \forall \mathsf{a}_1\ \mathsf{a}_2\,.\,\mathsf{Equal}\ \mathsf{a}_1\ \mathsf{a}_2 \rightarrow \mathsf{Equal}\ (\mathsf{Maybe}\ \mathsf{a}_1)\ (\mathsf{Maybe}\ \mathsf{a}_2) \\
\mathit{equal}_{\mathsf{Maybe}}\ \mathit{equal}_{\mathsf{a}} & = \mathit{to}\ (\mathit{mapE}_{\mathsf{Equal}}\ \mathit{conv}_{\mathsf{Maybe}}\ \mathit{conv}_{\mathsf{Maybe}})\ (\mathit{equal}_{:+:}\ \mathit{equal}_{\mathsf{Unit}}\ \mathit{equal}_{\mathsf{a}}) \\
\mathit{equal}_{\mathsf{List}} & ::\ \forall \mathsf{a}_1\ \mathsf{a}_2\,.\,\mathsf{Equal}\ \mathsf{a}_1\ \mathsf{a}_2 \rightarrow \mathsf{Equal}\ (\mathsf{List}\ \mathsf{a}_1)\ (\mathsf{List}\ \mathsf{a}_2) \\
\mathit{equal}_{\mathsf{List}}\ \mathit{equal}_{\mathsf{a}} & = \mathit{to}\ (\mathit{mapE}_{\mathsf{Equal}}\ \mathit{conv}_{\mathsf{List}}\ \mathit{conv}_{\mathsf{List}})\ (\\
 & \qquad \mathit{equal}_{:+:}\ \mathit{equal}_{\mathsf{Unit}}\ (\mathit{equal}_{:*:}\ \mathit{equal}_{\mathsf{a}}\ (\mathit{equal}_{\mathsf{List}}\ \mathit{equal}_{\mathsf{a}}))) \\
\mathit{equal}_{\mathsf{GRose}} & ::\ \forall \mathsf{f}_1\ \mathsf{f}_2\,.\,(\forall \mathsf{b}_1\ \mathsf{b}_2\,.\,\mathsf{Equal}\ \mathsf{b}_1\ \mathsf{b}_2 \rightarrow \mathsf{Equal}\ (\mathsf{f}_1\ \mathsf{b}_1)\ (\mathsf{f}_2\ \mathsf{b}_2)) \\
 & \qquad\qquad \rightarrow (\forall \mathsf{a}_1\ \mathsf{a}_2\,.\,\mathsf{Equal}\ \mathsf{a}_1\ \mathsf{a}_2 \\
 & \qquad\qquad\qquad\qquad \rightarrow \mathsf{Equal}\ (\mathsf{GRose}\ \mathsf{f}_1\ \mathsf{a}_1)\ (\mathsf{GRose}\ \mathsf{f}_2\ \mathsf{a}_2)) \\
\mathit{equal}_{\mathsf{GRose}}\ \mathit{equal}_{\mathsf{f}}\ \mathit{equal}_{\mathsf{a}} & \\
 & = \mathit{to}\ (\mathit{mapE}_{\mathsf{Equal}}\ \mathit{conv}_{\mathsf{GRose}}\ \mathit{conv}_{\mathsf{GRose}})\ (\\
 & \qquad \mathit{equal}_{:*:}\ \mathit{equal}_{\mathsf{a}}\ (\mathit{equal}_{\mathsf{f}}\ (\mathit{equal}_{\mathsf{GRose}}\ \mathit{equal}_{\mathsf{f}}\ \mathit{equal}_{\mathsf{a}})))
\end{array}
$$

Fig. 8. Specializing generic values in Haskell (part 2).

Two remarks are in order.

Haskell has neither type abstraction nor an explicit recursion operator, so these cases can be omitted from the definition.

Unfortunately, we cannot deal with polymorphic types in general. Consider, for instance, the type $\mathsf{Poly\ a} = \forall \mathsf{f} . \mathsf{f\ a} \to \mathsf{f\ a}$. There is no mapping function that works uniformly for all f. For that reason we have to restrict f to instances of $\mathsf{Functor}$ so that we can use the overloaded *liftE* function. For polymorphic types where the type variable ranges over types of kind $\star$ things are simpler: since the mapping function for a manifest type is always the identity, we can use id_E.

Now, what happens if Poly involves a user-defined data type, say B? In this case we have to specialize *mapE* to B. It seems that we are trapped in a vicious circle. One possibility to break the spell is to implement *mapE* for the B data type 'by hand'. Fortunately *mapE* is very well-behaved, so the code generation is straightforward. The embedding-projection map for the data type $\mathsf{B} :: \kappa$

$$\mathbf{data}\ \mathsf{B}\ \mathsf{a}_1\ \ldots\ \mathsf{a}_m = K_1\ \mathsf{t}_{11}\ \ldots\ \mathsf{t}_{1m_1} \mid \cdots \mid K_n\ \mathsf{t}_{n1}\ \ldots\ \mathsf{t}_{nm_n}$$

is given by

$$\begin{array}{ll}
mapE_{\mathsf{B}} & :: \mathsf{MapE}\{\![\kappa]\!\}\ \mathsf{B}\ \mathsf{B} \\
mapE_{\mathsf{B}}\ mapE_{\mathsf{a}_1}\ \ldots\ mapE_{\mathsf{a}_m} & \\
 & = EP\{from = from_{\mathsf{B}}, to = to_{\mathsf{B}}\} \\
\quad \mathbf{where} & \\
\quad from_{\mathsf{B}}\ (K_1\ x_{11}\ \ldots\ x_{1m_1}) & = K_1\ (from\{\![\mathsf{t}_{11}]\!\}\ \rho\ x_{11})\ \ldots\ (from\{\![\mathsf{t}_{1m_1}]\!\}\ \rho\ x_{1m_1}) \\
\quad \ldots & \\
\quad from_{\mathsf{B}}\ (K_n\ x_{n1}\ \ldots\ x_{nm_n}) & = K_n\ (from\{\![\mathsf{t}_{n1}]\!\}\ \rho\ x_{n1})\ \ldots\ (from\{\![\mathsf{t}_{nm_n}]\!\}\ \rho\ x_{nm_n}) \\
\quad to_{\mathsf{B}}\ (K_1\ x_{11}\ \ldots\ x_{1m_1}) & = K_1\ (to\{\![\mathsf{t}_{11}]\!\}\ \rho\ x_{11})\ \ldots\ (to\{\![\mathsf{t}_{1m_1}]\!\}\ \rho\ x_{1m_1}) \\
\quad \ldots & \\
\quad to_{\mathsf{B}}\ (K_n\ x_{n1}\ \ldots\ x_{nm_n}) & = K_n\ (to\{\![\mathsf{t}_{n1}]\!\}\ \rho\ x_{n1})\ \ldots\ (to\{\![\mathsf{t}_{nm_n}]\!\}\ \rho\ x_{nm_n})
\end{array}$$

where $from\{\![\mathsf{t}]\!\}\ \rho = from\ (mapE\{\![\mathsf{t}]\!\}\ \rho)$, $to\{\![\mathsf{t}]\!\}\ \rho = to\ (mapE\{\![\mathsf{t}]\!\}\ \rho)$, and $\rho = (\mathsf{a}_1 := mapE_{\mathsf{a}_1}, \ldots, \mathsf{a}_m := mapE_{\mathsf{a}_m})$. For example, for Encode and $\mathsf{Decodes}$ we obtain

$$\begin{array}{ll}
mapE_{\mathsf{Encode}} & :: \forall \mathsf{a}\ \mathsf{a}^\circ . \mathsf{EP\ a\ a}^\circ \to \mathsf{EP\ (Encode\ a)\ (Encode\ a}^\circ) \\
mapE_{\mathsf{Encode}}\ mapE_{\mathsf{a}} & = mapE_{\to}\ mapE_{\mathsf{a}}\ id_E \\
mapE_{\mathsf{Decodes}} & :: \forall \mathsf{a}\ \mathsf{a}^\circ . \mathsf{EP\ a\ a}^\circ \to \mathsf{EP\ (Decodes\ a)\ (Decodes\ a}^\circ) \\
mapE_{\mathsf{Decodes}}\ mapE_{\mathsf{a}} & = mapE_{\to}\ id_E\ (mapE_{(,)}\ mapE_{\mathsf{a}}\ id_E)
\end{array}$$

where the mapping function $mapE_{(,)}$ is generated according to the scheme above:

$$\begin{array}{ll}
mapE_{(,)} & :: \forall \mathsf{a}\ \mathsf{a}^\circ . \mathsf{EP\ a\ a}^\circ \to \forall \mathsf{b}\ \mathsf{b}^\circ . \mathsf{EP\ b\ b}^\circ \to \mathsf{EP\ (a, b)\ (a}^\circ, \mathsf{b}^\circ) \\
mapE_{(,)}\ mapE_{\mathsf{a}}\ mapE_{\mathsf{b}} & = EP\{from = from_{(,)}, to = to_{(,)}\} \\
\quad \mathbf{where}\ from_{(,)}\ (a, b) & = (from\ mapE_{\mathsf{a}}\ a, from\ mapE_{\mathsf{b}}\ b) \\
\qquad\qquad to_{(,)}\ (a, b) & = (to\ mapE_{\mathsf{a}}\ a, to\ mapE_{\mathsf{b}}\ b).
\end{array}$$

Interestingly, the Generic Haskell compiler does not treat *mapE* in a special way. Rather, it uses the same specialization mechanism also for *mapE* instances.

This is possible because *mapE*'s type involves only the type EP, for which we can specialize *mapE* by hand.

4 Conclusion

We have presented Generic Haskell, an extension of Haskell that allows the definition of generic programs. We have shown how to implement typical examples of generic programs such as equality, pretty printing, mapping functions and reductions. The central idea is to define a generic function by induction on the structure of types. 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 real challenge: how can we define generic functions and how can we assign types to these functions? It turns out that type-indexed values possess kind-indexed types, types that are defined by induction on the structure of kinds.

Though generic programming adds an extra level of abstraction to programming, it is in many cases simpler than conventional programming. The fundamental reason is that genericity gives you a lot of things for free. For instance, the generic programmer only has to provide cases for primitive types and for binary sums and products. Generic Haskell automatically takes care of type abstraction, type application, and type recursion.

Generic Haskell takes a transformational approach to generic programming: a generic function is translated into a family of polymorphic functions. We have seen that this transformation can be phrased as an interpretation of the simply typed lambda calculus. One of the benefits of this approach—not mentioned in these notes—is that it is possible to adapt one of the main tools for studying typed lambda calculi, logical relations, to generic reasoning, see [22]. To prove a generic property it suffices to prove the assertion for type constants. Everything else is taken care of automatically.

Finally, we have shown how to adapt the technique of specialization to Haskell, whose type system is based on name equivalence.

Acknowledgement. Thanks are due to Andres Löh for implementing the Generic Haskell compiler.

References

1. Artem Alimarine and Rinus Plasmeijer. A generic programming extension for Clean. In Th. Arts and M. Mohnen, editors, *Proceedings of the 13th International workshop on the Implementation of Functional Languages, IFL'01*, pages 257–278, lvsj, Sweden, September 2001.
2. Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In Jeremy Gibbons and Jeuring Jeuring, editors, *Pre-Proceedings of IFIP TC2 Working Conf. on Generic Programming, WCGP'02, Dagstuhl, 11–12 July 2002*, 2002. (Final Proceedings to be published by Kluwer Acad. Publ.).

3. Lennart Augustsson. *The Haskell B. Compiler (HBC)*, 1998. Available from `http://www.cs.chalmers.se/~augustss/hbc/hbc.html`.
4. Lennart Augustsson. Cayenne – a language with dependent types. *SIGPLAN Notices*, 34(1):239–250, January 1999.
5. Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. In Simon Peyton Jones, editor, *Proceedings of the 2002 International Conference on Functional Programming, Pittsburgh, PA, USA, October 4-6, 2002*, pages 157–166. ACM Press, October 2002.
6. Richard Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall Europe, London, 2nd edition, 1998.
7. 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.
8. Richard Bird and Ross Paterson. Generalised folds for nested datatypes. *Formal Aspects of Computing*, 11(2):200–222, 1999.
9. James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In Manuel M.T. Chakravarty, editor, *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop*, pages 90–104. ACM Press, October 2002.
10. Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report 92/480/18, Dept. of Computer Science, Univ. of Calgary, June 1992.
11. 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.
12. 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.
13. G. Gierz, K.H. Hofmann, K Keimel, J.D. Lawson, M. Mislove, and D.S. Scott. *A Compendium of Continuous Lattices.* Springer-Verlag, 1980.
14. Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.
15. T. Hagino. *Category Theoretic Approach to Data Types.* PhD thesis, University of Edinburgh, 1987.
16. Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *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.
17. Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.
18. Ralf Hinze. A generic programming extension for Haskell. In Erik Meijer, editor, *Proceedings of the 3rd Haskell Workshop, Paris, France*, September 1999. The proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-1999-28.
19. Ralf Hinze. Polytypic programming with ease (extended abstract). 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 21–36. Springer-Verlag, November 1999.

20. Ralf Hinze. Functional Pearl: Perfect trees and bit-reversal permutations. *Journal of Functional Programming*, 10(3):305–317, May 2000.
21. 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.
22. Ralf Hinze. Polytypic values possess polykinded types. *Science of Computer Programmming*, 43:129–159, 2002.
23. Ralf Hinze and Simon Peyton Jones. Derivable type classes. In Graham Hutton, editor, *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41.1 of Electronic Notes in Theoretical Computer Science. Elsevier Science, August 2001. The preliminary proceedings appeared as a University of Nottingham technical report.
24. 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.
25. 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.
26. C.B. Jay, G. Bellè, and E. Moggi. Functorial ML. *Journal of Functional Programming*, 8(6):573–619, November 1998.
27. 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.
28. 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.
29. M.P. Jones and J.C. Peterson. *Hugs 98 User Manual*, May 1999. Available from `http://www.haskell.org/hugs`.
30. G. Malcolm. *Algebraic data types and program transformation.* PhD thesis, University of Groningen, 1990.
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. Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.
34. John C. Mitchell. *Foundations for Programming Languages.* The MIT Press, Cambridge, MA, 1996.
35. Eugenio Moggi. A cateogry-theoretic account of program modules. *Mathematical Structures in Computer Science*, 1(1):103–139, March 1991.

36. 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.
37. Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998.
38. Simon Peyton Jones. *Haskell 98 Language and Libraries.* Cambridge University Press, 2003.
39. 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.
40. Benjamin C. Pierce. *Types and programming languages.* MIT Press, Cambridge, Mass., 2002.
41. 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.
42. Karl Fritz Ruehr. *Analytical and Structural Polymorphism Expressed using Patterns over Types.* PhD thesis, University of Michigan, 1992.
43. Tim Sheard. Automatic generation and use of abstract structure operators. *ACM Transactions on Programming Languages and Systems*, 13(4):531–557, October 1991.
44. 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.
45. The GHC Team. *The Glasgow Haskell Compiler User's Guide, Version 5.04*, 2003. Available from `http://www.haskell.org/ghc/documentation.html`.
46. 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.
47. Philip Wadler. The Girard-Reynolds isomorphism. In N. Kobayashi and B. C. Pierce, editors, *Proc. of 4th Int. Symp. on Theoretical Aspects of Computer Science, TACS 2001, Sendai, Japan, 29–31 Oct. 2001*, volume 2215 of *Lecture Notes in Computer Science*, pages 468–491. Springer-Verlag, Berlin, 2001.
48. Philip Wadler. A prettier printer. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, Cornerstones of Computing, pages 223–243. Palgrave Macmillan Publishers Ltd, March 2003.
49. Stephanie Weirich. Higher-order intensional type analysis. In D. Le Métayer, editor, *Proceedings of the 11th European Symposium on Programming, ESOP 2002*, volume 2305 of *Lecture Notes in Computer Science*, pages 98–114, 2002.