Generic Programming, Now!

Ralf Hinze[1] and Andres Löh[1]

Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
`{ralf,loeh}@informatik.uni-bonn.de`

Abstract. Tired of writing boilerplate code? Tired of repeating essentially the same function definition for lots of different datatypes? Datatype-generic programming promises to end these coding nightmares. In these lecture notes, we present the key abstractions of datatype-generic programming, give several applications, and provide an elegant embedding of generic programming into Haskell. The embedding builds on recent advances in type theory: generalised algebraic datatypes and open datatypes. We hope to convince you that generic programming is useful and that you can use generic programming techniques today!

1 Introduction

A type system is like a suit of armour: it shields against the modern dangers of illegal instructions and memory violations, but it also restricts flexibility. The lack of flexibility is particularly vexing when it comes to implementing fundamental operations such as showing a value or comparing two values. In a statically typed language such as Haskell 98 [37] it is simply not possible, for instance, to define an equality test that works for all types. As a rule of thumb, the more expressive a type system, the more fine-grained the type information and the more difficult it becomes to write general-purpose functions.

This problem has been the focus of intensive research for more than a decade. In Haskell 1.0 and in subsequent versions of the language, the problem was only partially addressed: by attaching a so-called *deriving form* to a datatype declaration the programmer can instruct the compiler to generate an instance of equality for the new type. In fact, the deriving mechanism is not restricted to equality: parsers, pretty-printers and several other functions are derivable, as well. These functions have become known as *datatype-generic* or *polytypic* functions, functions that work for a whole family of types. Unfortunately, Haskell's deriving mechanism is closed: the programmer cannot introduce new generic functions.

A multitude of proposals have been put forward that support exactly this, the *definition* of generic functions. Some of the proposals define new languages, some define extensions to existing languages, and some define libraries within existing languages. The early proposals had a strong background in category theory; the recent years have seen a gentle shift towards type-theoretic approaches. In these lecture notes, we present a particularly pragmatic approach: we show how to *embed* generic programming into Haskell. The embedding builds upon recent advances in type theory: generalised algebraic datatypes and open datatypes. Or to put it the other way round, we propose and employ language features that are useful for generic programming. Along the way, we will identify the basic building blocks of generic programming and we will provide an overview of the overall design space.

To cut a long story short, we hope to convince you that generic programming is useful and that you can use generic programming techniques today!

To get the most out of the lecture notes you require a basic knowledge of Haskell. To this end, Section 2 provides a short overview of the language and its various extensions. (The section is, however, too dense to serve as a beginner's guide to Haskell.) Section 3 then provides a gentle introduction to the main topic of these lecture notes: we show how to define generic functions and dynamic values, and give several applications. The remaining sections are overviewed at the end of Section 3.

Contents

2 Preliminaries

2.1 Values, types and kinds

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? Haskell allows the programmer to define parametric types such as the popular datatype of lists. The list type constructor can be seen as a function on types and the kind system allows us to specify this in a precise way. Thus, a kind is simply the 'type' of a type constructor.

$$\begin{array}{c} \text{kinds} \\ \hline \text{types} \\ \hline \text{values} \end{array}$$

Types and their kinds In Haskell, new datatypes are declared using the **data** construct. Here are three examples: the type of booleans, the type of pairs and the type of lists:

$$\begin{aligned} &\mathbf{data}\ Bool &&= False \mid True \\ &\mathbf{data}\ Pair\ \alpha\ \beta &&= (\alpha, \beta) \\ &\mathbf{data}\ [\alpha] &&= Nil \mid Cons\ \alpha\ [\alpha] \end{aligned}$$

In general, a datatype comprises one or more *constructors*, and each constructor can have multiple fields. A datatype declaration of the schematic form

$$\mathbf{data}\ T\ \alpha_1\ \ldots\ \alpha_s = C_1\ \tau_{1,1}\ \ldots\ \tau_{1,m_1} \mid \cdots \mid C_n\ \tau_{n,1}\ \ldots\ \tau_{n,m_n}$$

introduces data constructors $C_1, \ldots, C_n$ with signatures

$$C_i :: \forall \alpha_1\ \ldots\ \alpha_s\ .\ \tau_{i,1} \to \cdots \to \tau_{i,m_i} \to T\ \alpha_1\ \ldots\ \alpha_s$$

The constructors *False* and *True* of *Bool* have no arguments. The list constructors *Nil* and *Cons* are written [ ] and ':' in Haskell. For the purposes of these lecture notes, we stick to the explicit names, as we will use the colon for something else.

The following alternative definition of the pair datatype

$$\mathbf{data}\ Pair\ \alpha\ \beta = Pair\{fst :: \alpha, snd :: \beta\}$$

makes use of Haskell's *record syntax*: the declaration introduces the data constructor *Pair* and two accessor functions

$$\begin{aligned} &fst &&:: \forall \alpha\ \beta\ .\ Pair\ \alpha\ \beta \to \alpha \\ &snd &&:: \forall \alpha\ \beta\ .\ Pair\ \alpha\ \beta \to \beta \end{aligned}$$

Pairs and lists are examples of parameterised datatypes or *type constructors*. The kind of types such as *Bool* is $*$, whereas the kind of a type constructor is a function of the kind of its parameters to $*$. The kind of Pair is $* \to * \to *$, the kind of $[\,]$ is $* \to *$.

In general, the order of a kind is given by

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

Haskell supports kinds of arbitrary order.

Values and their types Functions in Haskell are usually defined using pattern matching. Here is the function *length* that computes the number of elements in a list:

$$\begin{aligned} &\mathit{length} :: \forall \alpha\ .\ [\alpha] \to \mathit{Int} \\ &\mathit{length}\ \mathit{Nil} &&= 0 \\ &\mathit{length}\ (\mathit{Cons}\ x\ \mathit{xs}) &&= 1 + \mathit{length}\ \mathit{xs} \end{aligned}$$

The *patterns* on the left hand side are matched against the actual arguments from left to right. The first equation, from top to bottom, where the match succeeds is applied. The first line of the definition is the *type signature* of *length*. Haskell can infer types of functions, but we generally provide type signatures of all top-level functions. The function *length* is *parametrically polymorphic*: the type of list elements is irrelevant; the function applies to arbitrary lists.

In general, the rank of a type is given by

$$\begin{aligned} &\mathit{rank}\ (T) &&= 0 \\ &\mathit{rank}\ (\forall \alpha\ .\ \tau) &&= \mathit{max}\{1, \mathit{rank}\ (\tau)\} \\ &\mathit{rank}\ (\sigma \to \tau) &&= \mathit{max}\{\mathit{inc}\ (\mathit{rank}\ (\sigma)), \mathit{rank}\ (\tau)\}, \end{aligned}$$

where $\mathit{inc}\ 0 = 0$ and $\mathit{inc}\ (n + 1) = n + 2$. Most implementations of Haskell support rank-2 types, although the Haskell 98 standard [37] does not. Recent versions of the Glasgow Haskell Compiler (GHC) [39] support types of arbitrary rank. In Haskell, type variables that appear free in a type signature are implicitly universally quantified on the outside. For example, the type signature of *length* could have been defined as $\mathit{length} :: [\alpha] \to \mathit{Int}$.

Sometimes, we use *pattern definitions* as a form of syntactic sugar. (Pattern definitions are not currently supported by any Haskell implementation.) A definition such as

$$\mathit{Single}\ x = \mathit{Cons}\ x\ \mathit{Nil}$$

defines *Single x* to be a transparent abbreviation of *Cons x Nil*. We can use *Single* on the right-hand side of a function definition to construct a value, but also as a derived pattern on the left-hand side of a function definition to destruct a function argument.

2.2 Generalised algebraic datatypes

Using a recent version of GHC, there is an alternative way of defining datatypes: by listing the signatures of the constructors explicitly. For example, the definition of lists becomes

$$\begin{array}{l} \textbf{data}\ [\,] :: * \to * \ \textbf{where} \\ \quad \mathit{Nil}\ \ \ :: \forall \alpha\ .\ [\alpha] \\ \quad \mathit{Cons} :: \forall \alpha\ .\ \alpha \to [\alpha] \to [\alpha] \end{array}$$

The first line declares the kind of the new datatype: $[\,]$ is a type constructor that takes types of kind $*$ to types of kind $*$. The type is then inhabited by listing the signatures of the data constructors. The original datatype syntax hides the fact that the result type of all constructors is $[\alpha]$; this is made explicit here. We can now also define datatypes where this is not the case, so-called *generalised algebraic datatypes* (GADTs):

$$\begin{array}{l} \textbf{data}\ \mathit{Expr} :: * \to * \ \textbf{where} \\ \quad \mathit{Num} :: \mathit{Int} \to \mathit{Expr}\ \mathit{Int} \\ \quad \mathit{Plus}\ :: \mathit{Expr}\ \mathit{Int} \to \mathit{Expr}\ \mathit{Int} \to \mathit{Expr}\ \mathit{Int} \\ \quad \mathit{Eq}\ \ \ \ :: \mathit{Expr}\ \mathit{Int} \to \mathit{Expr}\ \mathit{Int} \to \mathit{Expr}\ \mathit{Bool} \\ \quad \mathit{If}\ \ \ \ \ :: \forall \alpha\ .\ \mathit{Expr}\ \mathit{Bool} \to \mathit{Expr}\ \alpha \to \mathit{Expr}\ \alpha \to \mathit{Expr}\ \alpha \end{array}$$

The datatype Expr represents *typed expressions*: the data constructor *Plus*, for instance, can only be applied to arithmetic expressions of type $\mathit{Expr}\ \mathit{Int}$; applying *Plus* to a Boolean expression results in a type error. It is important to note that the type Expr cannot be introduced by a standard Haskell 98 **data** declaration since the constructors have different result types.

For functions on GADTs, type signatures are mandatory. Here is an evaluator for the Expr datatype:

$$\begin{array}{ll} \mathit{eval} :: \forall \alpha\ .\ \mathit{Expr}\ \alpha \to \alpha & \\ \mathit{eval}\ (\mathit{Num}\ i) & = i \\ \mathit{eval}\ (\mathit{Plus}\ e_1\ e_2) & = \mathit{eval}\ e_1 + \mathit{eval}\ e_2 \\ \mathit{eval}\ (\mathit{Eq}\ e_1\ e_2) & = \mathit{eval}\ e_1 == \mathit{eval}\ e_2 \\ \mathit{eval}\ (\mathit{If}\ e_1\ e_2\ e_3) & = \textbf{if}\ \mathit{eval}\ e_1\ \textbf{then}\ \mathit{eval}\ e_2\ \textbf{else}\ \mathit{eval}\ e_3 \end{array}$$

Even though *eval* is assigned the type $\forall \alpha\ .\ \mathit{Expr}\ \alpha \to \alpha$, each equation – with the notable exception of the last one – has a more specific type as dictated by the type constraints. As an example, the first equation has type $\mathit{Expr}\ \mathit{Int} \to \mathit{Int}$ as *Num* constrains α to Int. The interpreter is quite notable in that it is *tag free* — that is, no explicit type information is carried at run-time. If it receives a Boolean expression, then it returns a Boolean.

In the following, we often omit universal quantifiers in type signatures: type variables that occur free in a type signature are implicitly universally quantified at the outermost level.

2.3 Open datatypes and open functions

Consider the datatype of expressions that we introduced in the previous section. The expression language supports integers, addition, equality and conditionals, but nothing else. If we want to add additional constructs to the expression language, then we have to extend the datatype.

In these lecture notes, we assume that we can extend datatypes that have been flagged as "open": new constructors can be freely added without modifying the code that already has been written. In order to mark Expr as an open datatype, we declare it as follows:

$$\textbf{open data}\ \mathit{Expr} :: * \to *$$

Constructors can then be introduced just by providing their type signatures. Here, we add three new constructors for strings, for turning numbers into strings and for concatenating strings:

$$\begin{aligned}
&\mathit{Str} && :: \mathit{String} \to \mathit{Expr}\ \mathit{String}\\
&\mathit{Show} && :: \mathit{Expr}\ \mathit{Int} \to \mathit{Expr}\ \mathit{String}\\
&\mathit{Cat} && :: \mathit{Expr}\ \mathit{String} \to \mathit{Expr}\ \mathit{String} \to \mathit{Expr}\ \mathit{String}
\end{aligned}$$

In order to extend a function, we first have to declare it as open. This is accomplished by providing a type signature flagged with the **open** keyword:

$$\textbf{open}\ \mathit{eval} :: \mathit{Expr}\ \alpha \to \alpha$$

The definition of an open function need not be contiguous; the defining equations may be scattered around the program. We can thus extend the evaluator to cover the three new constructors of the Expr datatype:

$$\begin{aligned}
&\mathit{eval}\ (\mathit{Str}\ s) && = s\\
&\mathit{eval}\ (\mathit{Show}\ e) && = \mathit{show}_{\mathit{Int}}\ (\mathit{eval}\ e)\\
&\mathit{eval}\ (\mathit{Cat}\ e_1\ e_2) && = \mathit{eval}\ e_1 +\!\!+\ \mathit{eval}\ e_2
\end{aligned}$$

The semantics of open datatypes and open functions is the same as if they had been defined closed, in a single place. Openness is therefore mainly a matter of convenience and modularity; it does not increase the expressive power of the language. We use open datatypes and open functions throughout these lecture notes, but the code remains executable in current Haskell implementations that do not support these constructs: one can apply a preprocessor that collects into one place all the constructors for open datatypes and all the defining equations for open functions.

Using open datatypes and open functions gives us both directions of extensibility mentioned in the famous *expression problem* [41]: we can add additional sorts of data, by providing new constructors, and we can add additional operations, by defining new functions. Here is another function on expressions, which turns a given expression into its string representation:

$\textbf{open}\ \mathit{string} :: \mathit{Expr}\ \alpha \to \mathit{String}$
$\mathit{string}\ (\mathit{Num}\ i) = \texttt{"(Num"} +\!\!\!+\!\!\!+\ \mathit{show}_{\mathit{Int}}\ i +\!\!\!+ \texttt{")"}$
$\mathit{string}\ (\mathit{Plus}\ e_1\ e_2) = \texttt{"(Plus"} +\!\!\!+\!\!\!+\ \mathit{string}\ e_1 +\!\!\!+\!\!\!+\ \mathit{string}\ e_2 +\!\!\!+ \texttt{")"}$
$\mathit{string}\ (\mathit{Eq}\ e_1\ e_2) = \texttt{"(Eq"} +\!\!\!+\!\!\!+\ \mathit{string}\ e_1 +\!\!\!+\!\!\!+\ \mathit{string}\ e_2 +\!\!\!+ \texttt{")"}$
$\mathit{string}\ (\mathit{If}\ e_1\ e_2\ e_3) = \texttt{"(If"} +\!\!\!+\!\!\!+\ \mathit{string}\ e_1 +\!\!\!+\!\!\!+\ \mathit{string}\ e_2 +\!\!\!+\!\!\!+\ \mathit{string}\ e_3 +\!\!\!+ \texttt{")"}$
$\mathit{string}\ (\mathit{Str}\ s) = \texttt{"(Str"} +\!\!\!+\!\!\!+\ \mathit{show}_{\mathit{String}}\ s +\!\!\!+ \texttt{")"}$
$\mathit{string}\ (\mathit{Show}\ e) = \texttt{"(Show"} +\!\!\!+\!\!\!+\ \mathit{string}\ e +\!\!\!+ \texttt{")"}$
$\mathit{string}\ (\mathit{Cat}\ e_1\ e_2) = \texttt{"(Cat"} +\!\!\!+\!\!\!+\ \mathit{string}\ e_1 +\!\!\!+\!\!\!+\ \mathit{string}\ e_2 +\!\!\!+ \texttt{")"}$

The auxiliary operator '$+\!\!\!+\!\!\!+$' concatenates two strings with an intermediate blank:

$$s_1 +\!\!\!+\!\!\!+\ s_2 = s_1 +\!\!\!+ \texttt{" "} +\!\!\!+ s_2$$

As an aside, we note that $\forall\alpha\ .\ \mathit{Expr}\ \alpha \to \mathit{String}$, the type of *string*, is isomorphic to the *existential type* $(\exists\alpha\ .\ \mathit{Expr}\ \alpha) \to \mathit{String}$, as α does not occur in the result type.

For open functions, first-fit pattern matching is not suitable. To see why, suppose that we want to provide a default definition for *string* in order to prevent pattern matching failures, stating that everything without a specific definition is ignored in the string representation:

$$\mathit{string}\ \_ = \texttt{""}$$

Using first-fit pattern matching, this equation effectively closes the definition of *string*. Later equations cannot be reached at all. Furthermore, if equations of the function definition are scattered across multiple modules, it is unclear (or at least hard to track) in which order they will be matched with first-fit pattern matching.

We therefore adopt a different scheme for open functions, called *best-fit left-to-right* pattern matching. The idea is that the most specific match rather than the first match wins. This makes the order in which equations of the open function appear irrelevant. In the example above, it ensures that the default case for *string* will be chosen only if no other equation matches. If open functions are implemented via a preprocessor, the defining equations have to be reordered in such a way that the more specific equations come first. The details of open datatypes and functions are described in a recent paper [32].

3 A guided tour

3.1 Type-indexed functions

In Haskell, showing values of a datatype is particularly easy: one simply attaches a **deriving** (*Show*) clause to the declaration of the datatype.

$\textbf{data}\ \mathit{Tree}\ \alpha = \mathit{Empty} \mid \mathit{Node}\ (\mathit{Tree}\ \alpha)\ \alpha\ (\mathit{Tree}\ \alpha)$
$\qquad\qquad\qquad \textbf{deriving}\ (\mathit{Show})$

The compiler then automatically generates a suitable *show* function. This function is used, for instance, in interactive sessions to print the result of a submitted expression (the string '*Now*⟩ ' is the prompt of the interpreter).

```
Now⟩ tree [0 .. 3]
Node (Node (Node Empty 0 Empty) 1 Empty) 2 (Node Empty 3 Empty)
```

Here *tree* :: $[\alpha] \to Tree\ \alpha$ transforms a list into a balanced tree (see Appendix A.1). The function *show* can be seen as a *pretty-printer*. The display of larger structures, however, is not especially pretty, due to lack of indentation.

```
Now⟩ tree [0 .. 9]
Node (Node (Node (Node Empty 0 Empty) 1 Empty) 2 (Node (Node Em
pty 3 Empty) 4 Empty)) 5 (Node (Node (Node Empty 6 Empty) 7 Empt
y) 8 (Node Empty 9 Empty))
```

In the sequel, we develop a replacement for *show*, a generic prettier-printer. There are several pretty-printing libraries around; since these lecture notes focus on generic programming techniques rather than pretty-printing we pick a very basic one (see Appendix A.2), which just offers basic support for indentation.

```
data Text
text   :: String → Text
nl     :: Text
indent :: Int → Text → Text
(◇)    :: Text → Text → Text
```

The function *text* converts a string to a text, where *Text* is type of documents with indentation. By convention, the string passed to *text* must not contain newline characters. The constant *nl* has to be used for that purpose. The function *indent* adds a given number of spaces after each newline. Finally, '◇' concatenates two pieces of text.

Given this library it is a simple exercise to write a prettier-printer for trees of integers.

```
pretty_Int :: Int → Text
pretty_Int n = text (show_Int n)

pretty_TreeInt :: Tree Int → Text
pretty_TreeInt Empty        = text "Empty"
pretty_TreeInt (Node l x r) = align "(Node " (pretty_TreeInt l ◇ nl ◇
                                              pretty_Int     x ◇ nl ◇
                                              pretty_TreeInt r ◇ text ")")

align :: String → Text → Text
align s d = indent (length s) (text s ◇ d)
```

While the program does the job, it is not very general: we can print trees of integers, but not, say, trees of characters. Of course, it is easy to add another two ad-hoc definitions.

```
pretty_Char :: Char → Text
pretty_Char c = text (show_Char c)

pretty_TreeChar :: Tree Char → Text
pretty_TreeChar Empty        = text "Empty"
pretty_TreeChar (Node l x r) = align "(Node " (pretty_TreeChar l ◇ nl ◇
                                               pretty_Char     x ◇ nl ◇
                                               pretty_TreeChar r ◇ text ")")
```

The code of *pretty*$_{Tree\,Char}$ is almost identical to that of *pretty*$_{Tree\,Int}$. It seems that we actually need a *family* of pretty printers: *Tree* is a parameterised datatype and quite naturally one would like the elements contained in a tree to be pretty-printed, as well. For concreteness, let us assume that the types of interest are given by the following grammar.

$$\tau ::= Char \mid Int \mid (\tau, \tau) \mid [\tau] \mid Tree\ \tau$$

Implementing a type-indexed family of functions sounds like a typical case for Haskell's type classes, in particular, since the deriving mechanism itself relies on the class system: **deriving** (*Show*) generates an instance of Haskell's predefined *Show* class. However, this is only one of several options. In the sequel we explore a different route that does not depend on Haskell's most beloved feature. Sections 4 and 5 will then put this approach in perspective, providing an overview of the overall design space.

type-indexed functions. A simple approach to generic programming defines a family of functions indexed by type.

$$poly_{\tau} :: Poly\ \tau$$

The family contains a definition of $poly_{\tau}$ for each type τ of interest; the type of $poly_{\tau}$ is parameterised by the type index τ. For brevity, we call *poly* a *type-indexed function* (omitting the 'family of').

Now, instead of implementing a type-indexed family of pretty-printers, we define a single function that receives the type as an additional argument and suitably dispatches on this type argument. However, Haskell doesn't permit the explicit passing of types. An alternative is to pass the pretty-printer an additional argument that *represents* the type of the value we wish to convert to text. As a first try, we could assign the pretty-printer the type $Type \to \alpha \to Text$ where *Type* is the type of type representations. Unfortunately, this is too simple-minded: the parametricity theorem [42] implies that a function of this type must necessarily ignore its second parameter. This argument breaks down, however, if we additionally parameterise *Type* by the type it represents. The signature of the pretty-printer then becomes $Type\ \alpha \to \alpha \to Text$. The idea is that an element of type $Type\ \tau$ is a representation of the type τ. Using a *generalised algebraic datatype* (see Section 2.2), we can define *Type* directly in Haskell.

```
open data Type :: * → * where
  Char :: Type Char
  Int    :: Type Int
  Pair   :: Type α → Type β → Type (α, β)
  List   :: Type α → Type [α]
  Tree   :: Type α → Type (Tree α)
String :: Type String
String = List Char
```

We declare *Type* to be *open* (Section 2.3) so that we can add a new type representation whenever we define a new datatype. The derived constructor *String*, defined by a *pattern definition* (Section 2.1), is equal to *List Char* in all contexts. Recall that we allow *String* to be used on the left-hand side of equations. Each type has a unique representation: the type *Int* is represented by the constructor *Int*, the type (*String*, *Int*) is represented by *Pair String Int* and so forth. For any given τ in our family of types, *Type* τ comprises exactly one element (ignoring $\perp$); *Type* τ is a so-called *singleton type*.

In the sequel, we often need to annotate an expression with its type representation. We introduce a special type for this purpose.[1]

```
infixl 1 :
data Typed α = (:){ val :: α, type :: Type α }
```

The definition, which makes use of Haskell's record syntax, introduces the colon ':' as an infix data constructor. Thus, $4711 : \mathit{Int}$ is an element of *Typed Int* and (47, `"hello"`) : *Pair Int String* is an element of *Typed* (*Int*, *String*). It is important to note the difference between $x : t$ and $x :: \tau$. The former expression constructs a pair consisting of a value x and a representation t of its type. The latter expression is Haskell syntax for 'x has type τ'.

Given these prerequisites, we can finally define the desired pretty-printer:

```
open pretty :: Typed α → Text
pretty (c : Char)           = pretty_Char c
pretty (n : Int)            = pretty_Int n
pretty ((x, y) : Pair a b)  = align "( " (pretty (x : a)) ◇ nl ◇
                              align ", " (pretty (y : b)) ◇ text ")"
pretty (xs : List a)        = bracketed [pretty (x : a) | x ← xs]
pretty (Empty : Tree a)     = text "Empty"
pretty (Node l x r : Tree a)
  = align "(Node " (pretty (l : Tree a) ◇ nl ◇
                    pretty (x : a) ◇ nl ◇
                    pretty (r : Tree a) ◇ text ")")
```

[1] The operator ':' is predefined in Haskell for constructing lists. However, since we use type annotations much more frequently than lists, we use ':' for the former and *Nil* and *Cons* for the latter purpose. Furthermore, we agree upon the convention that the pattern $x : t$ is matched from *right to left*: first the type representation t is matched, then the associated value x. In other words: in proper Haskell source code, $x : t$ has to be written in reverse order, as `t :> x`.

We declare *pretty* to be open so that we can later extend it by additional equations. The function *pretty* makes heavy use of type annotations; its type $\mathit{Typed}\ \alpha \to \mathit{Text}$ is essentially an uncurried version of $\mathit{Type}\ \alpha \to \alpha \to \mathit{Text}$. Even though *pretty* has a polymorphic type, each equation implements a more specific case as dictated by the type annotations. For example, the first equation has type $\mathit{Typed}\ \mathit{Int} \to \mathit{Text}$.

Let us consider each equation in turn. The first two equations take care of integers and characters, respectively. Pairs are enclosed in parentheses, the two elements being separated by a linebreak and a comma. Lists are shown using *bracketed*, defined in Appendix A.2, which produces a comma-separated sequence of elements between square brackets. Finally, trees are displayed using prefix notation.

The function *pretty* is defined by explicit case analysis on the type representation. This is typical of a type-dependent function, but not compulsory: the wrapper function *show*, defined below, is given by a simple abstraction.

```
show    :: Typed α → String
show x = render (pretty x)
```

The pretty-printer produces output in the following style.

```
Now⟩ pretty (tree : Tree Int [0 . . 3])
(Node (Node (Node Empty
                  0
                  Empty)
            1
            Empty)
      2
      (Node Empty
            3
            Empty))
Now⟩ pretty ([(47, "hello"), (11, "world")] : List (Pair Int String))
[(47
 , [ 'h'
   , 'e'
   , 'l'
   , 'l'
   , 'o'])
, (11
 , [ 'w'
   , 'o'
   , 'r'
   , 'l'
   , 'd'])]
```

While the layout nicely emphasises the structure of the tree, the pretty-printed strings look slightly odd: a string is formatted as a list of characters. Fortunately, this problem is easy to remedy: we add a special case for strings.

$pretty\ (s : String) = text\ (show_{String}\ s)$

This case is more specific than the one for lists; best-fit pattern matching ensures that the right instance is chosen. Now, we get

```
Now⟩ pretty ([(47, "hello"), (11, "world")] : List (Pair Int String))
[ (47
  , "hello")
, (11
  , "world")]
```

The type of type representations is, of course, by no means specific to pretty-printing. Using type representations, we can define arbitrary type-dependent functions. Here is a second example: collecting strings.

```
open strings :: Typed α → [String]
strings (i : Int)          = Nil
strings (c : Char)         = Nil
strings (s : String)       = [s]
strings ((x, y) : Pair a b) = strings (x : a) ++ strings (y : b)
strings (xs : List a)      = concat [strings (x : a) | x ← xs]
strings (t : Tree a)       = strings (inorder t : List a)
```

The function *strings* returns the list of all strings contained in the argument structure. The example shows that we need not program every case from scratch: the *Tree* case falls back on the list case. Nonetheless, most of the cases have a rather ad-hoc flavour. Surely, there must be a more systematic approach to collecting strings.

type-polymorphic functions. A function of type

$poly :: \forall\alpha\ .\ Type\ \alpha \to Poly\ \alpha$

is called *type-polymorphic* or *intensionally polymorphic*. By contrast, a function of type $\forall\alpha\ .\ Poly\ \alpha$ is called *parametrically polymorphic*.

A note on style: if $Poly\ \alpha$ is of the form $\alpha \to \sigma$ where α does not occur in σ (*poly* is a so-called consumer), we will usually prefer the uncurried variant $poly :: \forall\alpha\ .\ Typed\ \alpha \to \sigma$ over the curried version.

3.2 Introducing new datatypes

We have declared *Type* to be open so that we can freely add new constructors to the *Type* datatype and so that we can freely add new equations to existing open functions on *Type*. To illustrate the extension of *Type*, consider the type of perfect binary trees [12].

```
data Perfect α = Zero α | Succ (Perfect (α, α))
```

As an aside, note that *Perfect* is a so-called *nested data type* [3]. To be able to pretty-print perfect trees, we add a constructor to the type *Type* of type representations and extend *pretty* by suitable equations.

```
Perfect :: Type α → Type (Perfect α)
pretty (Zero x : Perfect a) = align "(Zero " (pretty (x : a) ◇ text ")")
pretty (Succ x : Perfect a)
   = align "(Succ " (pretty (x : Perfect (Pair a a)) ◇ text ")")
```

Here is a short interactive session that illustrates the extended version of *pretty*.

```
Now⟩ pretty (perfect 4 1 : Perfect Int)
(Succ (Succ (Succ (Succ (Zero ((((1
                                 ,1)
                                ,(1
                                 ,1))
                               ,((1
                                 ,1)
                                ,(1
                                 ,1)))
                              ,(((1
                                 ,1)
                                ,(1
                                 ,1))
                               ,((1
                                 ,1)
                                ,(1
                                 ,1))))))))))
```

The function *perfect d a* generates a perfect tree of depth *d* whose leaves are labelled with *a*s.

3.3 Generic functions

Using type representations, we can program functions that work uniformly for all types of a given family, so-called *overloaded functions*. Let us now broaden the scope of *pretty* and *strings* so that they work for *all* datatypes, including types that the programmer has yet to define. For emphasis, we call these functions *generic functions*.

overloaded and generic functions. An *overloaded function* works for a fixed family of types. By contrast, a *generic function* works for all types, including types that the programmer has yet to define.

We have seen in the previous section that whenever we define a new datatype, we add a constructor of the same name to the type of type representations and we add corresponding equations to *all* generic functions. While the extension of

Type is cheap and easy (a compiler could do this for us), the extension of all type-indexed functions is laborious and difficult (can you imagine a compiler doing that?). In this section we develop a scheme so that it suffices to extend *Type* by a new constructor and to extend *one or two* particular overloaded functions. The remaining functions adapt themselves.

To achieve this goal we need to find a way to treat elements of a data type in a general, uniform way. Consider an arbitrary element of some datatype. It is always of the form $C\ e_1\ \cdots\ e_n$, a constructor applied to some values. For instance, an element of *Tree Int* is either *Empty* or of the form *Node l a r*. The idea is to make this applicative structure visible and accessible: to this end we mark the constructor using *Con* and each function application using '$\diamond$'. Additionally, we annotate the constructor arguments with their types and the constructor itself with information on its syntax. Consequently, the constructor *Empty* becomes *Con empty* and the expression *Node l a r* becomes *Con node* $\diamond$ $(l : \mathit{Tree\ Int}) \diamond (a : \mathit{Int}) \diamond (r : \mathit{Tree\ Int})$ where *empty* and *node* are the tree constructors augmented with additional information. The functions *Con* and '$\diamond$' are themselves constructors of a datatype called *Spine*.

```
infixl 0 ◇
data Spine :: * → * where
  Con :: Constr α → Spine α
  (◇)  :: Spine (α → β) → Typed α → Spine β
```

The type is called *Spine* because its elements represent the possibly partial spine of a constructor application (a constructor application can be seen as the internal node of a binary tree; the path to the leftmost leaf in a binary tree is called its *left spine*). The following table illustrates the stepwise construction of a spine.

```
node :: Constr (Tree Int → Int → Tree Int → Tree Int)
Con node :: Spine (Tree Int → Int → Tree Int → Tree Int)
Con node ◇ (l : Tree Int) :: Spine (Int → Tree Int → Tree Int)
Con node ◇ (l : Tree Int) ◇ (a : Int) :: Spine (Tree Int → Tree Int)
Con node ◇ (l : Tree Int) ◇ (a : Int) ◇ (r : Tree Int) :: Spine (Tree Int)
```

If we ignore the type constructors *Constr*, *Spine* and *Typed*, then *Con* has the type of the identity function, $\alpha \to \alpha$, and '$\diamond$' has the type of function application, $(\alpha \to \beta) \to \alpha \to \beta$. Note that the type variable α does not appear in the result type of '$\diamond$': it is existentially quantified.[2] This is the reason why we annotate the second argument with its type. Otherwise, we wouldn't be able to use it as an argument of an overloaded function (see below).

An element of type *Constr* α comprises an element of type α, namely the original data constructor, plus some additional information about its syntax: its name, its arity, its fixity and its order. The order is a pair (i, n) with $1 \leqslant i \leqslant n$, which specifies that the constructor is the ith of a total of n constructors.

[2] All type variables in Haskell are universally quantified. However, $\forall\alpha\ .\ (\sigma \to \tau)$ is isomorphic to $(\exists\alpha\ .\ \sigma) \to \tau$ provided α does not appear free in τ; this is where the term 'existential type' comes from.

```
data Constr α = Descr{ constr :: α,
                       name   :: String,
                       arity  :: Int,
                       fixity :: Fixity,
                       order  :: (Integer, Integer)}
data Fixity = Prefix Int | Infix Int | Infixl Int | Infixr Int | Postfix Int
```

Given a value of type *Spine* α, we can easily recover the original value of type α by undoing the conversion step.

```
fromSpine :: Spine α → α
fromSpine (Con c) = constr c
fromSpine (f ◇ x)  = (fromSpine f) (val x)
```

The function *fromSpine* is parametrically polymorphic; it works independently of the type in question, as it simply replaces *Con* with the original constructor and '$\diamond$' with function application.

The inverse of *fromSpine* is not polymorphic; rather, it is an overloaded function of type *Typed* $\alpha \to$ *Spine* α. Its definition, however, follows a trivial pattern (so trivial that the definition could be easily generated by a compiler): if the datatype comprises a constructor C with signature

$$C :: \tau_1 \to \cdots \to \tau_n \to \tau_0$$

then the equation for *toSpine* takes the form

$$toSpine\ (C\ x_1\ \ldots\ x_n : t_0) = Con\ c \diamond (x_1 : t_1) \diamond \cdots \diamond (x_n : t_n)$$

where c is the annotated version of C and t_i is the type representation of τ_i. As an example, here is the definition of *toSpine* for binary trees.

```
open toSpine :: Typed α → Spine α
toSpine (Empty : Tree a)      = Con empty
toSpine (Node l x r : Tree a) = Con node ◇ (l : Tree a) ◇ (x : a) ◇ (r : Tree a)
empty :: Constr (Tree α)
empty = Descr{ constr = Empty,
               name   = "Empty",
               arity  = 0,
               fixity = Prefix 10,
               order  = (0, 2)}
node  :: Constr (Tree α → α → Tree α → Tree α)
node  = Descr{ constr = Node,
               name   = "Node",
               arity  = 3,
               fixity = Prefix 10,
               order  = (1, 2)}
```

Note that this scheme works for arbitrary datatypes including generalised algebraic datatypes!

With all the machinery in place we can now turn *pretty* and *strings* into truly generic functions. The idea is to add a catch-all case to each function that takes care of all the remaining type cases in a uniform manner. Let's tackle *strings* first.

```
strings x = strings_ (toSpine x)
strings_ :: Spine α → [String]
strings_ (Con c) = []
strings_ (f ◇ x)  = strings_ f ++ strings x
```

The helper function *strings*₋ traverses the spine calling *strings* for each argument of the spine.

Actually, we can drastically simplify the definition of *strings*: every case except the one for *String* is subsumed by the catch-all case. Hence, the definition boils down to:

```
strings :: Typed α → [String]
strings (s : String) = [s]
strings x            = strings_ (toSpine x)
```

The revised definition makes clear that *strings* has only one type-specific case, namely the one for *String*. This case must be separated out, because we want to do something specific for strings, something that does not follow the general pattern.

The catch-all case for *pretty* is almost as easy. We only have to take care that we do not parenthesize nullary constructors.

```
pretty x = pretty_ (toSpine x)
pretty_ :: Spine α → Text
pretty_ (Con c) = text (name c)
pretty_ (f ◇ x)  = pretty1_ f (pretty x)
pretty1_ :: Spine α → Text → Text
pretty1_ (Con c) d = align ("(" ++ name c ++ " ") (d ◇ text ")")
pretty1_ (f ◇ x)  d = pretty1_ f (pretty x ◇ nl ◇ d)
```

Now, why are we in a better situation than before? When we introduce a new datatype such as, say, *XML*, we still have to extend the representation type with a constructor *XML* :: *Type XML* and provide cases for the data constructors of *XML* in the *toSpine* function. However, this has to be done only once per datatype, and it is so simple that it could easily be done automatically. The code for the generic functions (of which there can be many) is completely unaffected by the addition of a new datatype. As a further plus, the generic functions are unaffected by changes to a given datatype (unless they include code that is specific to the datatype). Only the function *toSpine* must be adapted to the new definition and possibly the type representation if the kind of the datatype changes.

3.4 Dynamic values

Haskell is a statically typed language. Unfortunately, one cannot guarantee the absence of run-time errors using static checks only. For instance, when we communicate with the environment, we have to check dynamically whether the imported values have the expected types. In this section we show how to embed dynamic checking in a statically typed language.

To this end we introduce a *universal datatype*, the type *Dynamic*, which encompasses all static values. To inject a static value into the universal type we bundle the value with a representation of its type, re-using the *Typed* datatype.

$$\begin{array}{l}\textbf{data}\ \mathit{Dynamic} :: * \ \textbf{where}\\ \quad \mathit{Dyn} :: \mathit{Typed}\ \alpha \to \mathit{Dynamic}\end{array}$$

Note that the type variable α does not appear in the result type: it is effectively existentially quantified. In other words, *Dynamic* is the union of all typed values. As an example, *misc* is a list of a dynamic values.

$$\begin{array}{l}\mathit{misc} :: [\mathit{Dynamic}]\\ \mathit{misc} = [\mathit{Dyn}\ (4711 : \mathit{Int}), \mathit{Dyn}\ (\texttt{"hello world"} : \mathit{String})]\end{array}$$

Since we have introduced a new type, we must extend the type of type representations.

$$\mathit{Dynamic} :: \mathit{Type}\ \mathit{Dynamic}$$

Now, we can also turn *misc* itself into a dynamic value: $\mathit{Dyn}\ (\mathit{misc}{:}\mathit{List}\ \mathit{Dynamic})$.

Dynamic values and generic functions go well together. In a sense, they are dual concepts.[3] We can quite easily extend the generic function *strings* so that it additionally works for dynamic values.

$$\mathit{strings}\ (\mathit{Dyn}\ x : \mathit{Dynamic}) = \mathit{strings}\ x$$

An element of type *Dynamic* just contains the necessary information required by *strings*. In fact, the situation is similar to the *Spine* datatype where the second argument of '$\diamond$' also has an existentially quantified type (this is why we had to add type information).

Can we also extend *toSpine* by a case for *Dynamic* so that *strings* works without any changes? Of course! As a first step we add *Type* and *Typed* to the type of representable types.

$$\begin{array}{l}\mathit{Type}\ \ :: \mathit{Type}\ \alpha \to \mathit{Type}\ (\mathit{Type}\ \alpha)\\ \mathit{Typed} :: \mathit{Type}\ \alpha \to \mathit{Type}\ (\mathit{Typed}\ \alpha)\end{array}$$

[3] The type *Dynamic* corresponds to the infinite union $\exists\alpha\ .\ \mathit{Typed}\ \alpha$; a generic function of type $\mathit{Typed}\ \alpha{\to}\sigma$ corresponds to the infinite intersection $\forall\alpha\ .\ (\mathit{Typed}\ \alpha{\to}\sigma)$ which equals $(\exists\alpha\ .\ \mathit{Typed}\ \alpha) \to \sigma$ if α does not occur in σ. Hence, a generic function of this type can be seen as taking a dynamic value as an argument.

The first line looks a bit intimidating with four occurrences of the same identifier, but it exactly follows the scheme for unary type constructors: the representation of $T :: * \to *$ is $T :: \mathit{Type}\ \alpha \to \mathit{Type}\ (T\ \alpha)$.

As a second step, we provide suitable instances of *toSpine* pedantically following the general scheme given in Section 3.3 (*oftype* is the infix operator ':' augmented by additional information).

```
toSpine (Char : Type Char)      = Con char
toSpine (List t : Type (List a)) = Con list ◇ (t : Type a)   -- t = a
...
toSpine ((x : t) : Typed a)      = Con oftype ◇ (x : t) ◇ (t : Type t)   -- t = a
```

Note that t and a must be the same type representation since the type representation of $x : t$ is *Typed* t. It remains to extend *toSpine* by a *Dynamic* case.

```
toSpine (Dyn x : Dynamic) = Con dyn ◇ (x : Typed (type x))
```

It is important to note that this instance does *not* follow the general pattern for *toSpine*. The reason is that *Dyn*'s argument is existentially quantified and in general, we do not have any type information about existentially quantified types at runtime (see also Section 5.1). But the whole purpose of *Dyn* is to pack a value and its type together, and we therefore can use this type information to define *toSpine*.

To summarise, for every (closed) type with n constructors we have to add $n + 1$ equations for *toSpine*, one for the type representation itself and one for each of the n constructors.

Given these prerequisites *strings* now works without any changes. There is, however, a slight difference to the previous version: the generic case for *Dynamic* traverses *both* the static value *and* its type, as ':' is treated just like every other data constructor. This may or this may not be what is wanted.

For *pretty* we decide to give an ad-hoc type case for typed values (we want to use infix rather than prefix notation for ':') and to fall back on the generic case for dynamic values.

```
pretty ((x : t) : Typed a) = align "( " (pretty (x : t)) ◇ nl ◇    -- t = a
                             align ": " (pretty (t : Type t)) ◇ text ")"
```

Here is a short interactive session that illustrates pretty-printing dynamic values.

```
Now⟩ pretty (misc : List Dynamic)
[(Dyn (4711
       : Int))
,(Dyn ("hello world"
       : (List Char)))]
```

The constructor *Dyn* turns a static into a dynamic value. The other way round involves a dynamic type check. This operation, usually termed *cast*, takes a dynamic value and a type representation and checks whether the type representation of the dynamic value and the one supplied are identical. The type-equality check itself is given by an *overloaded function* that takes two type representations and possibly returns a *proof* of their equality (a simple truth value is not enough). The proof states that one type may be substituted for the other. We define

$$\textbf{data}\ (:=:) :: * \to * \to *\ \textbf{where}\ \mathit{Refl} :: \alpha :=: \alpha$$

This generalised algebraic datatype has the intriguing property that it is non-empty if and only if its argument types are equal.[4] Given an equality proof of α and β, we can turn any value of type α into a value of type β by pattern matching on the proof and thus making the type-equality constraint available to the type checker:

$$\begin{array}{l}
\mathit{apply} :: (\alpha :=: \beta) \to (\alpha \to \beta) \\
\mathit{apply}\ p\ x = \textbf{case}\ p\ \textbf{of}\ \{\mathit{Refl} \to x\}
\end{array}$$

The type-equality type has all the properties of a *congruence relation*. The constructor *Refl* itself serves as the proof of reflexivity. The equality type is furthermore symmetric, transitive, and congruent. Here are programs that implement the proofs of congruence for type constructors of kind $* \to *$ and $* \to * \to *$.

$$\begin{array}{lll}
\mathit{ctx}_1 & :: & (\alpha :=: \beta) \to (\psi\ \alpha :=: \psi\ \beta) \\
\mathit{ctx}_1\ p = & & \textbf{case}\ p\ \textbf{of}\ \{\mathit{Refl} \to \mathit{Refl}\} \\
\mathit{ctx}_2 & :: & (\alpha_1 :=: \beta_1) \to (\alpha_2 :=: \beta_2) \to (\psi\ \alpha_1\ \alpha_2 :=: \psi\ \beta_1\ \beta_2) \\
\mathit{ctx}_2\ p_1\ p_2 = & & \textbf{case}\ p_1\ \textbf{of}\ \{\mathit{Refl} \to \textbf{case}\ p_2\ \textbf{of}\ \{\mathit{Refl} \to \mathit{Refl}\}\}
\end{array}$$

The type-equality check is now given by

$$\begin{array}{llll}
\multicolumn{4}{l}{\mathit{equal} :: \mathit{Type}\ \alpha \to \mathit{Type}\ \beta \to \mathit{Maybe}\ (\alpha :=: \beta)} \\
\mathit{equal}\ \mathit{Int} & \mathit{Int} & = & \mathit{return}\ \mathit{Refl} \\
\mathit{equal}\ \mathit{Char} & \mathit{Char} & = & \mathit{return}\ \mathit{Refl} \\
\mathit{equal}\ (\mathit{Pair}\ a_1\ a_2) & (\mathit{Pair}\ b_1\ b_2) & = & \mathit{liftM2}\ \mathit{ctx}_2\ (\mathit{equal}\ a_1\ b_1)\ (\mathit{equal}\ a_2\ b_2) \\
\mathit{equal}\ (\mathit{List}\ a) & (\mathit{List}\ b) & = & \mathit{liftM}\ \mathit{ctx}_1\ (\mathit{equal}\ a\ b) \\
\mathit{equal}\ \_ & \_ & = & \mathit{fail}\ \texttt{"types are not unifiable"}
\end{array}$$

Since the equality check may fail, we must lift the congruence proofs into the *Maybe* monad using *return*, *liftM*, and *liftM2*. Note that the running time of the

[4] We ignore the fact here, that in Haskell every type contains the bottom element. Alternatively, we can adapt Leibniz's principle of substituting equals for equals to types and define

$$\textbf{newtype}\ \alpha :=: \beta = \mathit{Proof}\{\mathit{apply} :: \forall\varphi\ .\ \varphi\ \alpha \to \varphi\ \beta\}$$

An element of $\alpha :=: \beta$ is then a function that converts an element of type $\varphi\ \alpha$ into an element of $\varphi\ \beta$ for *any* type constructor φ. Operationally, this function is always the identity.

cast function that *equal* returns is *linear* in the size of the type (it is independent of the size of its argument structure).

The *cast* operation simply calls *equal* and then applies the conversion function to the dynamic value.

```
cast                      :: Dynamic → Type α → Maybe α
cast (Dyn (x : a)) t = fmap (λp → apply p x) (equal a t)
```

Again, we have to introduce an auxiliary datatype to direct Haskell's type-checker. Here is a short session that illustrates the use of *cast*.

```
Now⟩ let d = Dyn (4711 : Int)
Now⟩ pretty (d : Dynamic)
(Dyn (4711
      :Int))
Now⟩ d ‘cast‘ Int
Just 4711
Now⟩ fromJust (d ‘cast‘ Int) + 289
5000
Now⟩ d ‘cast‘ Char
Nothing
```

In a sense, *cast* can be seen as the dynamic counterpart of the colon operator: x ‘*cast*‘ T yields a static value of type τ if T is the representation of τ.

generic functions and dynamic values. Generics and dynamics are dual concepts:

$$\begin{array}{ll} \text{generic function:} & \forall\alpha\ .\ \mathit{Type}\ \alpha \to \sigma \\ \text{dynamic value:} & \exists\alpha\ .\ \mathit{Type}\ \alpha \times \sigma \end{array}$$

This is analogous to first-order predicate logic where $\forall x{:}T\ .\ P(x)$ is shorthand for $\forall x\ .\ T(x) \Rightarrow P(x)$ and $\exists x{:}T\ .\ P(x)$ abbreviates $\exists x\ .\ T(x) \wedge P(x)$.

3.5 Stocktaking

Before we proceed, let us step back to see what we have achieved so far.

Broadly speaking, generic programming is about defining functions that work for all types but that also exhibit type-specific behaviour. Using a GADT we have reflected types onto the value level. For each type constructor we have introduced a data constructor: types of kind $*$ are represented by constants; parameterised types are represented by functions that take type representations to type representations. Using reflected types we can program *overloaded functions*, functions that work for a fixed class of types and that exhibit type-specific behaviour. Finally, we have defined the *Spine* datatype that allows us to treat data in a uniform manner. Using this uniform view on data we can generalise overloaded functions to *generic* ones.

In general, support for generic programming consists of three essential ingredients:

– a type reflection mechanism,
– a type representation, and
– a generic view on data.

Let us consider each ingredient in turn.

Type reflection Using the type of type representations we can program functions that depend on or dispatch on types. Alternative techniques include Haskell's type classes and a type-safe cast. We stick to the GADT technique in these lecture notes.

Type representation Ideally, a representation type is a faithful mirror of the language's type system. To be able to define such a representation type or some representation type at all, the type system must be sufficiently expressive. We have seen that GADTs allow for a very direct representation; in a less expressive type system we may have to encode types less directly or in a less type-safe manner. However, the more expressive a type system, the more difficult it is to reflect the full system onto the value level. We shall see in Section 4 that there are several ways to model the Haskell type system and that the one we have used in this section is *not* the most natural or the most direct one. Briefly, the type *Type* models the type system of Haskell 1.0; it is difficult to extend to the more expressive system of Haskell 98 (or to one of its manifold extensions).

Generic view The generic view has the largest impact on the expressivity of a generic programming system: it affects the set of datatypes we can cover, the class of functions we can write and potentially the efficiency of these functions. In this section we have used the spine view to represent data in a uniform way. We shall see that this view is applicable to a large class of datatypes, including GADTs. The reason for the wide applicability is simple: a datatype definition describes how to construct data, the spine view captures just this. Its main weakness is also rooted in the 'value-orientation': one can only define generic functions that consume data (*show*) but not ones that produce data (*read*). Again, the reason for the limitation is simple: a uniform view on individual constructor applications is useful if you have data in your hands, but it is of no help if you want to construct data. Section 5 shows how to overcome this limitation and furthermore introduces alternative views.

4 Type representations

4.1 Representation types for types of a fixed kind

Representation type for types of kind $*$ The type *Type* of Section 3.1 represents types of kind $*$. A type constructor T is represented by a data constructor

T of the same name. A type of kind $*$ is either a basic type such as Char or Int, or a compound type such as $\mathit{List}\ \mathit{Char}$ or $\mathit{Pair}\ \mathit{Int}\ (\mathit{List}\ \mathit{Char})$. The components of a compound type are possibly type constructors of higher kinds such as List or Pair. These type constructors must also be represented using the type Type of type representations. Since type constructors are reflected onto the value level, the type of the data constructor T depends on the kind of the type constructor T. To see the precise relationship between the type of T and the kind of T, re-consider the declaration of Type, this time making polymorphic types explicit.

$$\begin{array}{l}
\textbf{open data}\ \mathit{Type} :: * \to * \ \textbf{where} \\
\quad \mathit{Char} :: \mathit{Type}\ \mathit{Char} \\
\quad \mathit{Int}\ \ :: \mathit{Type}\ \mathit{Int} \\
\quad \mathit{Pair} :: \forall\alpha\ .\ \mathit{Type}\ \alpha \to (\forall\beta\ .\ \mathit{Type}\ \beta \to \mathit{Type}\ (\alpha,\beta)) \\
\quad \mathit{List}\ :: \forall\alpha\ .\ \mathit{Type}\ \alpha \to \mathit{Type}\ [\alpha] \\
\quad \mathit{Tree} :: \forall\alpha\ .\ \mathit{Type}\ \alpha \to \mathit{Type}\ (\mathit{Tree}\ \alpha)
\end{array}$$

A type constructor T of higher kind is represented by a *polymorphic* function that takes a type representation for α to a type representation for $T\ \alpha$, for all types α. In general, T_κ has the signature

$$T_\kappa :: \mathit{Type}_\kappa\ T_\kappa$$

where Type_κ is defined

$$\begin{array}{l}
\textbf{type}\ \mathit{Type}_*\quad \alpha = \mathit{Type}\ \alpha \\
\textbf{type}\ \mathit{Type}_{\iota\to\kappa}\ \varphi = \forall\alpha\ .\ \mathit{Type}_\iota\ \alpha \to \mathit{Type}_\kappa\ (\varphi\ \alpha)
\end{array}$$

Thus, application on the type level corresponds to application of polymorphic functions on the value level.

So far we have only encountered first-order type constructors. Here is an example of a second-order one:

$$\textbf{newtype}\ \mathit{Fix}\ \varphi = \mathit{In}\{\mathit{out} :: \varphi\ (\mathit{Fix}\ \varphi)\}$$

The declaration introduces a fixed point operator, Fix, on the type level, whose kind is $(* \to *) \to *$. Consequently, the value counterpart of Fix has a rank-2 type: it takes a polymorphic function as an argument.

$$\mathit{Fix} :: \forall\varphi\ .\ (\forall\alpha\ .\ \mathit{Type}\ \alpha \to \mathit{Type}\ (\varphi\ \alpha)) \to \mathit{Type}\ (\mathit{Fix}\ \varphi)$$

Using *Fix*, the representation of type fixed points, we can now extend, for instance, *strings* by an appropriate case.

$$\mathit{strings}\ (\mathit{In}\ x : \mathit{Fix}\ f) = \mathit{strings}\ (x : f\ (\mathit{Fix}\ f))$$

Of course, this case is not really necessary: if we add a *Fix* equation to *toSpine*, then the specific case above is subsumed by the generic one of Section 3.3.

$$\mathit{toSpine}\ (\mathit{In}\ x : \mathit{Fix}\ f) = \mathit{Con}\ \mathit{in} \diamond (x : f\ (\mathit{Fix}\ f))$$

Here *in* is the annotated variant of *In*. Again, the definition of *toSpine* pedantically follows the general scheme.

Unfortunately, we cannot extend the definition of *equal* to cover the *Fix* case: *equal* cannot recursively check the arguments of *Fix* for equality, as they are polymorphic functions. In general, we face the problem that we cannot pattern match on polymorphic functions: *Fix List*, for instance, is not a legal pattern (*List* is not saturated). In Section 4.2 we introduce an alternative type representation that does not suffer from this problem.

Representation type for types of kind $* \to *$ The generic functions of Section 3 abstract over a type of kind $*$. For instance, *pretty* generalises functions of type

$$\mathit{Char} \to \mathit{Text}, \quad \mathit{String} \to \mathit{Text}, \quad [[\mathit{Int}]] \to \mathit{Text}$$

to a single generic function of type

$$\mathit{Type}\ \alpha \to \alpha \to \mathit{Text} \qquad \text{or equivalently} \qquad \mathit{Typed}\ \alpha \to \mathit{Text}$$

A generic function may also abstract over a type constructor of higher kind. Take, as an example, the function *size* that counts the number of elements contained in some data structure. This function generalises functions of type

$$[\alpha] \to \mathit{Int}, \quad \mathit{Tree}\ \alpha \to \mathit{Int}, \quad [\mathit{Tree}\ \alpha] \to \mathit{Int}$$

to a single generic function of type

$$\mathit{Type}'\ \varphi \to \varphi\ \alpha \to \mathit{Int} \qquad \text{or equivalently} \qquad \mathit{Typed}'\ \varphi\ \alpha \to \mathit{Int}$$

where Type' is a representation type for types of kind $* \to *$ and Typed' is a suitable type, to be defined shortly, for annotating values with these representations.

How can we represent type constructors of kind $* \to *$? Clearly, the type $\mathit{Type}_{* \to *}$ is not suitable, as we intend to define *size* and other generic functions by case analysis on the type constructor. Again, the elements of $\mathit{Type}_{* \to *}$ are polymorphic functions and pattern-matching on functions would break referential transparency. Therefore, we define a new tailor-made representation type.

```
open data Type′ :: (∗ → ∗) → ∗ where
  List :: Type′ [ ]
  Tree :: Type′ Tree
```

Think of the prime as shorthand for the kind index $* \to *$. Additionally, we introduce a primed variant of *Typed*.

```
infixl 1 :′
data Typed′ φ α = (:′){val′ :: φ α, type′ :: Type′ φ}
```

The type $Type'$ is only inhabited by two constructors since the other datatypes have kinds different from $* \to *$.

An overloaded version of *size* is now straightforward to define.

$$
\begin{array}{ll}
size :: Typed' \; \varphi \; \alpha \to Int & \\
size \; (Nil :' List) & = 0 \\
size \; (Cons \; x \; xs :' List) & = 1 + size \; (xs :' List) \\
size \; (Empty :' Tree) & = 0 \\
size \; (Node \; l \; x \; r :' Tree) & = size \; (l :' Tree) + 1 + size \; (r :' Tree)
\end{array}
$$

Unfortunately, *size* is not as flexible as *pretty*. If we have some compound data structure x, say, a list of trees of integers, then we can simply call $pretty \; (x : List \; (Tree \; Int))$. We cannot, however, use *size* to count the total number of integers, simply because the new versions of *List* and *Tree* take no arguments!

There is one further problem, which is more fundamental. Computing the size of a compound data structure is inherently ambiguous: in the example above, do we count the number of integers, the number of trees or the number of lists? Formally, we have to solve the type equation $\varphi \; \tau = List \; (Tree \; Int)$. The equation has, in fact, not three but four principal solutions: $\varphi = \Lambda\alpha \to \alpha$ and $\tau = List \; (Tree \; Int)$, $\varphi = \Lambda\alpha \to List \; \alpha$ and $\tau = Tree \; Int$, $\varphi = \Lambda\alpha \to List \; (Tree \; \alpha)$ and $\tau = Int$, and $\varphi = \Lambda\alpha \to List \; (Tree \; Int)$ and τ arbitrary. How can we represent these different container types? They can be easily expressed using functions: $\lambda a \to a$, $\lambda a \to List \; a$, $\lambda a \to List \; (Tree \; a)$, and $\lambda a \to List \; (Tree \; Int)$. Alas, we are just trying to get rid of the functional representation. There are several ways out of this dilemma. One possibility is to *lift* the type constructors [14] so that they become members of $Type'$ and to include Id, defined as

$$\mathbf{newtype} \; Id \; \alpha = In_{Id} \{ out_{Id} :: \alpha \}$$

as a representation of the type variable α:

$$
\begin{array}{ll}
Id & :: Type' \; Id \\
Char' & :: Type' \; Char' \\
Int' & :: Type' \; Int' \\
List' & :: Type' \; \varphi \to Type' \; (List' \; \varphi) \\
Tree' & :: Type' \; \varphi \to Type' \; (Tree' \; \varphi)
\end{array}
$$

The type $List'$, for instance, is the lifted variant of $List$: it takes a type constructor of kind $* \to *$ to a type constructor of kind $* \to *$. Using the lifted types we can specify the four different container types as follows: Id, $List' \; Id$, $List' \; (Tree' \; Id)$ and $List' \; (Tree' \; Int')$. Essentially, we replace the types by their lifted counterparts and the type variable α by Id. Note that the above constructors of $Type'$ are *exactly identical* to those of $Type$ except for the kinds.

It remains to define the lifted versions of the type constructors.

$$
\begin{array}{l}
\mathbf{newtype} \; Char' \; \chi = In_{Char'} \{ out_{Char'} :: Char \} \\
\mathbf{newtype} \; Int' \;\;\; \chi = In_{Int'} \{ out_{Int'} :: Int \}
\end{array}
$$

$$\begin{aligned}
&\textbf{data}\ List'\ \alpha' \quad\quad\ \chi = Nil' \mid Cons'\ (\alpha'\ \chi)\ (List'\ \alpha'\ \chi)\\
&\textbf{data}\ Pair'\ \alpha'\ \beta'\ \chi = Pair'\ (\alpha'\ \chi)\ (\beta'\ \chi)\\
&\textbf{data}\ Tree'\ \alpha' \quad\quad \chi = Empty' \mid Node'\ (Tree'\ \alpha'\ \chi)\ (\alpha'\ \chi)\ (Tree'\ \alpha'\ \chi)
\end{aligned}$$

The lifted variants of the nullary type constructors $Char$ and Int simply ignore the additional argument χ. The **data** definitions follow a simple scheme: each data constructor C with signature

$$C :: \tau_1 \to \cdots \to \tau_n \to \tau$$

is replaced by a polymorphic data constructor C' with signature

$$C' :: \forall \chi\ .\ \tau_1'\ \chi \to \cdots \to \tau_n'\ \chi \to \tau_0'\ \chi$$

where τ_i' is the lifted variant of τ_i.

The function *size* can be easily extended to Id and to the lifted types.

$$\begin{aligned}
&size\ (x :'\ Id) &&= 1\\
&size\ (c :'\ Char') &&= 0\\
&size\ (i :'\ Int') &&= 0\\
&size\ (Nil' :'\ List'\ a') &&= 0\\
&size\ (Cons'\ x\ xs :'\ List'\ a') &&= size\ (x :'\ a') + size\ (xs :'\ List'\ a')\\
&size\ (Empty' :'\ Tree'\ a') &&= 0\\
&size\ (Node'\ l\ x\ r :'\ Tree'\ a')\\
&\quad = size\ (l :'\ Tree'\ a') + size\ (x :'\ a') + size\ (r :'\ Tree'\ a')
\end{aligned}$$

The instances are similar to the ones for the unlifted types, except that *size* is now also called recursively for list elements and tree labels, that is, for components of type α'.

Unfortunately, in Haskell *size* no longer works on the original data types: we cannot call, for instance, $size\ (x :'\ List'\ (Tree'\ Id))$ where x is a list of trees of integers, since $List'\ (Tree'\ Id)\ Int$ is different from $[Tree\ Int]$. However, the two types are isomorphic: $\tau \cong \tau'\ Id$ where τ' is the lifted variant of τ [14]. We leave it at that for the moment and return to the problem later in Section 5.

We have already noted that $Type'$ is similar to $Type$ except for the kinds. This becomes even more evident when we consider the signature of a lifted type representation: the lifted version of T_κ has signature

$$T_\kappa' :: Type_\kappa'\ T_\kappa'$$

where $Type_\kappa'$ is defined

$$\begin{aligned}
&\textbf{type}\ Type_*' \quad\ \ \alpha = Type'\ \alpha\\
&\textbf{type}\ Type_{\iota\to\kappa}'\ \varphi = \forall \alpha\ .\ Type_\iota'\ \alpha \to Type_\kappa'\ (\varphi\ \alpha)
\end{aligned}$$

Defining an overloaded function that abstracts over a type of kind $* \to *$ is similar to defining a $*$-indexed function, except that one has to consider one additional case, namely Id, which defines the action of the overloaded function on the type parameter. It is worth noting that it is not necessary to define instances for the unlifted type constructors ($[\,]$ and $Tree$ in our running example), as we have done, because these instances can be automatically derived from the lifted ones by virtue of the isomorphism $\tau \cong \tau'\ Id$ (see Section 5.3).

Representation type for types of kind ω Up to now we have confined ourselves to generic functions that abstract over types of kind $*$ or $* \to *$. An obvious question is whether the approach can be generalised to *kind indices* of arbitrary kinds. This is indeed possible. However, functions that are indexed by higher kinds, for instance, by $(* \to *) \to * \to *$ are rare. For that reason, we only sketch the main points. For a formal treatment see Hinze's earlier work [14]. Assume that $\omega = \kappa_1 \to \cdots \to \kappa_n \to *$ is the kind of the type index. We first introduce a suitable type representation and lift the datatypes to kind ω by adding n type arguments of kinds $\kappa_1, \ldots, \kappa_n$.

```
open data $Type^{\omega} :: \omega \to *$ where
  $T^{\omega}_{\kappa} :: Type^{\omega}_{\kappa}\ T^{\omega}_{\kappa}$
```

where T^{ω}_{κ} is the lifted version of T_{κ} and $Type^{\omega}_{\kappa}$ is defined

```
type $Type^{\omega}_{*}\quad \alpha = Type^{\omega}\ \alpha$
type $Type^{\omega}_{\iota \to \kappa}\ \varphi = \forall \alpha\ .\ Type^{\omega}_{\iota}\ \alpha \to Type^{\omega}_{\kappa}\ (\varphi\ \alpha)$
```

The lifted variant T^{ω}_{κ} of the type T_{κ} has kind κ^{ω} where $(-)^{\omega}$ is defined inductively on the structure of kinds

$$\begin{aligned} *^{\omega} &= \omega \\ (\iota \to \kappa)^{\omega} &= \iota^{\omega} \to \kappa^{\omega} \end{aligned}$$

Types and lifted types are related as follows: τ is isomorphic to $\tau'\ Out_1\ \ldots\ Out_n$ where Out_i is the *projection type* that corresponds to the i-th argument of ω. The generic programmer has to consider the cases for the lifted type constructors plus n additional cases, one for each of the n projection types $Out_1, \ldots, Out_n$.

4.2 Kind-indexed families of representation types

We have seen that type-indexed functions may abstract over arbitrary type constructors: *pretty* abstracts over types of kind $*$, size abstracts over types of kind $* \to *$. Sometimes a type-indexed function even makes sense for types of *different* kinds. A paradigmatic example is the *mapping function*: the mapping function of a type φ of kind $* \to *$ lifts a function of type $\alpha_1 \to \alpha_2$ to a function of type $\varphi\ \alpha_1 \to \varphi\ \alpha_2$; the mapping function of a type ψ of kind $* \to * \to *$ takes two functions of type $\alpha_1 \to \alpha_2$ and $\beta_1 \to \beta_2$ respectively and returns a function of type $\psi\ \alpha_1\ \beta_1 \to \psi\ \alpha_2\ \beta_2$. As an extreme case, the mapping function of a type σ of kind $*$ is the identity of type $\sigma \to \sigma$.

Dictionary-passing style The above discussion suggests turning *map* into a *family* of overloaded functions. Since the type of the mapping functions depends on the kind of the type argument, we have, in fact, a *kind-indexed family* of overloaded functions. To make this work we have to represent types differently: we require a kind-indexed family of representation types.

$$
\begin{array}{l}
\textbf{open data}\ \mathit{Type}_\kappa :: \kappa \to * \ \textbf{where}\\
\quad T_\kappa :: \mathit{Type}_\kappa\ T_\kappa
\end{array}
$$

In this scheme $\mathit{Int} :: *$ is represented by a data constructor of type Type_*; the type constructor $\mathit{Tree} :: * \to *$ is represented by a data constructor of type $\mathit{Type}_{* \to *}$ and so forth. There is, however, a snag in it. If the representation of Tree is not a function, how can we represent the application of Tree to some type? The solution is simple: we also represent type application syntactically using a family of kind-indexed constructors.

$$\mathit{App}_{\iota,\kappa} :: \mathit{Type}_{\iota \to \kappa}\ \varphi \to \mathit{Type}_\iota\ \alpha \to \mathit{Type}_\kappa\ (\varphi\ \alpha)$$

The result type dictates that $\mathit{App}_{\iota,\kappa}$ is an element of Type_κ. Theoretically, we need an infinite number of $\mathit{App}_{\iota,\kappa}$ constructors, one for each combination of ι and κ. Practically, only a few are likely to be used, since types with a large number of type arguments are rare. For our purposes the following declarations suffice.

$$
\begin{array}{lll}
\textbf{open data} & \mathit{Type}_* :: * \to * \ \textbf{where} & \\
\quad \mathit{Char}_* & :: \mathit{Type}_*\ \mathit{Char} & \\
\quad \mathit{Int}_* & :: \mathit{Type}_*\ \mathit{Int} & \\
\quad \mathit{App}_{*,*} & :: \mathit{Type}_{* \to *}\ \varphi \to \mathit{Type}_*\ \alpha \to \mathit{Type}_*\ (\varphi\ \alpha) & \\
\textbf{open data} & \mathit{Type}_{* \to *} :: (* \to *) \to * \ \textbf{where} & \\
\quad \mathit{List}_{* \to *} & :: \mathit{Type}_{* \to *}\ [\,] & \\
\quad \mathit{Tree}_{* \to *} & :: \mathit{Type}_{* \to *}\ \mathit{Tree} & \\
\quad \mathit{App}_{*,* \to *} & :: \mathit{Type}_{* \to * \to *}\ \varphi \to \mathit{Type}_*\ \alpha \to \mathit{Type}_{* \to *}\ (\varphi\ \alpha) & \\
\textbf{open data} & \mathit{Type}_{* \to * \to *} :: (* \to * \to *) \to * \ \textbf{where} & \\
\quad \mathit{Pair}_{* \to * \to *} & :: \mathit{Type}_{* \to * \to *}\ (,) &
\end{array}
$$

For example, $\mathit{Tree}\ \mathit{Int}$ is now represented by $\mathit{Tree}_{* \to *}$ ‘$\mathit{App}_{*,*}$‘ Int_*. We have $(\mathit{Pair}_{* \to * \to *}$ ‘$\mathit{App}_{*,* \to *}$‘ $\mathit{Int}_*)$ ‘$\mathit{App}_{*,*}$‘ $\mathit{Int}_* :: \mathit{Type}_*\ (\mathit{Int}, \mathit{Int})$. Since $\mathit{App}_{*,*}$ is a data constructor, we can pattern match both on $\mathit{Tree}_{* \to *}$ ‘$\mathit{App}_{*,*}$‘ a and on $\mathit{Tree}_{* \to *}$ alone. Since Haskell allows type constructors to be partially applied, the family Type_κ is indeed a faithful representation of Haskell's type system.

It is straightforward to adapt the type-indexed functions of Section 3 to the new representation. In fact, using a handful of *pattern definitions* we can re-use the code *without* any changes.

$$
\begin{array}{ll}
\mathit{Int} & :: \mathit{Type}_*\ \mathit{Int}\\
\mathit{Int} & = \mathit{Int}_*\\
\mathit{Char} & :: \mathit{Type}_*\ \mathit{Char}\\
\mathit{Char} & = \mathit{Char}_*\\
\mathit{Pair} & :: \mathit{Type}_*\ \alpha \to \mathit{Type}_*\ \beta \to \mathit{Type}_*\ (\alpha, \beta)\\
\mathit{Pair}\ a\ b & = \mathit{Pair}_{* \to * \to *}\ \text{‘}\mathit{App}_{*,* \to *}\text{‘}\ a\ \text{‘}\mathit{App}_{*,*}\text{‘}\ b\\
\mathit{List} & :: \mathit{Type}_*\ \alpha \to \mathit{Type}_*\ [\alpha]\\
\mathit{List}\ a & = \mathit{List}_{* \to *}\ \text{‘}\mathit{App}_{*,*}\text{‘}\ a
\end{array}
$$

$$\begin{aligned} &\mathit{Tree} && :: \mathit{Type}_{*}\ \alpha \to \mathit{Type}_{*}\ (\mathit{Tree}\ \alpha) \\ &\mathit{Tree}\ a && = \mathit{Tree}_{*\to *}\ \text{‘}\mathit{App}_{*,*}\text{‘}\ a \end{aligned}$$

The definitions show that the old representation can be defined in terms of the new representation. The reverse, however, is not true: we cannot turn a polymorphic function into a data constructor.

Now, let's tackle an example of a type-indexed function that works for types of different kinds. We postpone the implementation of the mapping function until the end of the section and first re-implement the function *size* that counts the number of elements contained in a data structure (see Section 4.1).

$$\mathit{size} :: \mathit{Type}_{*\to *}\ \varphi \to \varphi\ \alpha \to \mathit{Int}$$

How can we generalise *size* so that it works for types of arbitrary kinds? The essential step is to abstract away from *size*'s action on values of type α turning the action of type $\alpha \to \mathit{Int}$ into an additional argument:

$$\mathit{count}_{*\to *} :: \mathit{Type}_{*\to *}\ \varphi \to (\alpha \to \mathit{Int}) \to (\varphi\ \alpha \to \mathit{Int})$$

We call *size*'s kind-indexed generalisation *count*. If we instantiate the second argument of $\mathit{count}_{*\to *}$ to *const* 1, we obtain the original function back. But there is also an alternative choice: if we instantiate the second argument to *id*, we obtain a generalisation of Haskell's *sum* function, which sums the elements of a container.

$$\begin{aligned} &\mathit{size} && :: \mathit{Type}_{*\to *}\ \varphi \to \varphi\ \alpha \to \mathit{Int} \\ &\mathit{size}\ f && = \mathit{count}_{*\to *}\ f\ (\mathit{const}\ 1) \\ &\mathit{sum} && :: \mathit{Type}_{*\to *}\ \varphi \to \varphi\ \mathit{Int} \to \mathit{Int} \\ &\mathit{sum}\ f && = \mathit{count}_{*\to *}\ f\ \mathit{id} \end{aligned}$$

Two generic functions for the price of one!

Let us now turn to the definition of count_{κ}. Since count_{κ} is indexed by kind it also has a kind-indexed type.

$$\mathit{count}_{\kappa} :: \mathit{Type}_{\kappa}\ \alpha \to \mathit{Count}_{\kappa}\ \alpha$$

where Count_{κ} is defined

$$\begin{aligned} &\mathbf{type}\ \mathit{Count}_{*}\ \alpha = \alpha \to \mathit{Int} \\ &\mathbf{type}\ \mathit{Count}_{\iota\to\kappa}\ \varphi = \forall \alpha\ .\ \mathit{Count}_{\iota}\ \alpha \to \mathit{Count}_{\kappa}\ (\varphi\ \alpha) \end{aligned}$$

The definition looks familiar: it follows the scheme we have already encountered in Section 4.1 (Type_{κ} is defined analogously). The first line specifies that a 'counting function' maps an element to an integer. The second line expresses that $\mathit{count}_{\iota\to\kappa}\ f$ takes a counting function for α to a counting function for $\varphi\ \alpha$, for all α. This means that the kind-indexed function count_{κ} maps type application to application of generic functions.

$$\mathit{count}_{\kappa}\ (\mathit{App}_{\iota,\kappa}\ f\ a) = (\mathit{count}_{\iota\to\kappa}\ f)\ (\mathit{count}_{\iota}\ a)$$

This case for $App_{\iota,\kappa}$ is truly generic: it is the same for all kind-indexed generic functions (in dictionary-passing style; see below) and for all combinations of ι and κ. The type-specific behaviour of a generic function is solely determined by the cases for the different type constructors. As an example, here are the definitions for $count_{\kappa}$:

```
open $count_{*} :: Type_{*}\ \alpha \to Count_{*}\ \alpha$
$count_{*}\ (f\ `App_{*,*}`\ a) = (count_{*\to*}\ f)\ (count_{*}\ a)$
$count_{*}\ t$                  $= const\ 0$
open $count_{*\to*} :: Type_{*\to*}\ \alpha \to Count_{*\to*}\ \alpha$
$count_{*\to*}\ List_{*\to*}$            $c = sum_{[\,]}\ .\ map_{[\,]}\ c$
$count_{*\to*}\ Tree_{*\to*}$            $c = count_{*\to*}\ List_{*\to*}\ c\ .\ inorder$
$count_{*\to*}\ (f\ `App_{*,*\to*}`\ a)\ c = (count_{*\to*\to*}\ f)\ (count_{*}\ a)\ c$
open $count_{*\to*\to*} :: Type_{*\to*\to*}\ \alpha \to Count_{*\to*\to*}\ \alpha$
$count_{*\to*\to*}\ (Pair_{*\to*\to*})\ c_1\ c_2 = \lambda(x_1, x_2) \to c_1\ x_1 + c_2\ x_2$
```

Note that we have to repeat the generic $App_{\iota,\kappa}$ case for every instance of ι and κ. The catch-all case for types of kind $*$ determines that elements of types of kind $*$ such as Int or $Char$ are mapped to 0.

Taking the size of a compound data structure such as a list of trees of integers is now much easier than before: the count function for $\Lambda\alpha \to List\ (Tree\ \alpha)$ is the unique function that maps c to $count_{*\to*}\ (List_{*\to*})\ (count_{*\to*}\ (Tree_{*\to*})\ c)$. Here is a short interactive session that illustrates the use of *count* and *size*.

```
Now⟩ let $ts = [tree\ [0\,..\,i] \mid i \leftarrow [0\,..\,9]]$
Now⟩ $size\ (List_{*\to*})\ ts$
10
Now⟩ $count_{*\to*}\ (List_{*\to*})\ (size\ (Tree_{*\to*}))\ ts$
55
```

The fact that $count_{*\to*}$ is parameterised by the action on α allows us to mimic type abstraction by abstraction on the value level. Since $count_{*\to*}$ receives the $*$-instance of the count function as an argument, we say that *count* is defined in *dictionary-passing style*. There is also an alternative, type-passing style, which we discuss in a moment, where the type representation itself is passed as an argument.

The definition of the mapping function is analogous to the definition of *size* except for the type. Recall that the mapping function of a type φ of kind $* \to *$ lifts a function of type $\alpha_1 \to \alpha_2$ to a function of type $\varphi\ \alpha_1 \to \varphi\ \alpha_2$. The instance is doubly polymorphic: both the argument and the result type of the argument function may vary. Consequently, we assign *map* a kind-indexed type that has two type arguments:

```
$map_{\kappa} :: Type_{\kappa}\ \alpha \to Map_{\kappa}\ \alpha\ \alpha$
```

where Map_{κ} is defined

$$\begin{aligned}
&\textbf{type}\ \mathit{Map}_{*}\ \alpha_1\ \alpha_2 = \alpha_1 \to \alpha_2\\
&\textbf{type}\ \mathit{Map}_{\iota\to\kappa}\ \varphi_1\ \varphi_2 = \forall \alpha_1\ \alpha_2\ .\ \mathit{Map}_{\iota}\ \alpha_1\ \alpha_2 \to \mathit{Map}_{\kappa}\ (\varphi_1\ \alpha_1)\ (\varphi_2\ \alpha_2)
\end{aligned}$$

The definition of *map* itself is straightforward:

$$\begin{aligned}
&\textbf{open}\ \mathit{map}_{*} :: \mathit{Type}_{*}\ \alpha \to \mathit{Map}_{*}\ \alpha\ \alpha\\
&\mathit{map}_{*}\ \mathit{Int}_{*} = \mathit{id}\\
&\mathit{map}_{*}\ \mathit{Char}_{*} = \mathit{id}\\
&\mathit{map}_{*}\ (\mathit{App}_{*,*}\ f\ a) = (\mathit{map}_{*\to*}\ f)\ (\mathit{map}_{*}\ a)\\
&\textbf{open}\ \mathit{map}_{*\to*} :: \mathit{Type}_{*\to*}\ \varphi \to \mathit{Map}_{*\to*}\ \varphi\ \varphi\\
&\mathit{map}_{*\to*}\ \mathit{List}_{*\to*} = \mathit{map}_{[\,]}\\
&\mathit{map}_{*\to*}\ \mathit{Tree}_{*\to*} = \mathit{map}_{\mathit{Tree}}\\
&\mathit{map}_{*\to*}\ (\mathit{App}_{*,*\to*}\ f\ a) = (\mathit{map}_{*\to*\to*}\ f)\ (\mathit{map}_{*}\ a)\\
&\textbf{open}\ \mathit{map}_{*\to*\to*} :: \mathit{Type}_{*\to*\to*}\ \varphi \to \mathit{Map}_{*\to*\to*}\ \varphi\ \varphi\\
&\mathit{map}_{*\to*\to*}\ \mathit{Pair}_{*\to*\to*}\ f\ g\ (a, b) = (f\ a, g\ b)
\end{aligned}$$

Each instance simply defines the mapping function for the respective type.

kind-indexed functions. A kind-indexed family of type-polymorphic functions

$$\mathit{poly}_{\kappa} :: \forall \alpha\ .\ \mathit{Type}_{\kappa}\ \alpha \to \mathit{Poly}_{\kappa}\ \alpha$$

contains a definition of poly_{κ} for each kind κ of interest. The type representation Type_{κ} and the type Poly_{κ} are indexed by kind, as well. For brevity, we call poly_{κ} a *kind-indexed function* (omitting the 'family of type-polymorphic').

Type-passing style The functions above are defined in dictionary-passing style, as instances of overloaded functions are passed around. An alternative scheme passes the type representation instead. We can use it, for instance, to define $*$-indexed functions in a less verbose way. To illustrate, let us re-define the overloaded function *pretty* in type-passing style. Its kind-indexed type is given by

$$\begin{aligned}
&\textbf{type}\ \mathit{Pretty}_{*}\ \alpha = \alpha \to \mathit{Text}\\
&\textbf{type}\ \mathit{Pretty}_{\iota\to\kappa}\ \varphi = \forall \alpha\ .\ \mathit{Type}_{\iota}\ \alpha \to \mathit{Pretty}_{\kappa}\ (\varphi\ \alpha)
\end{aligned}$$

The equations for pretty_{κ} are similar to those of *pretty* of Section 3.1, except for the 'type patterns': the left-hand side $\mathit{pretty}\ (T\ a_1\ \ldots\ a_n)$ becomes $\mathit{pretty}_{\kappa}\ T_{\kappa}\ a_1\ \ldots\ a_n$, where κ is the kind of T.

$$\begin{aligned}
&\textbf{open}\ \mathit{pretty}_{*} :: \mathit{Type}_{*}\ \alpha \to \mathit{Pretty}_{*}\ \alpha\\
&\mathit{pretty}_{*}\ \mathit{Char}_{*}\ c = \mathit{pretty}_{\mathit{Char}}\ c\\
&\mathit{pretty}_{*}\ \mathit{Int}_{*}\ n = \mathit{pretty}_{\mathit{Int}}\ n\\
&\mathit{pretty}_{*}\ (f\ \text{`}\mathit{App}_{*,*}\text{`}\ a)\ x = \mathit{pretty}_{*\to*}\ f\ a\ x\\
&\textbf{open}\ \mathit{pretty}_{*\to*} :: \mathit{Type}_{*\to*}\ \alpha \to \mathit{Pretty}_{*\to*}\ \alpha
\end{aligned}$$

$$\begin{array}{l}
pretty_{*\to*}\ List_{*\to*}\ a\ xs = bracketed\ [pretty_{*}\ a\ x \mid x \leftarrow xs] \\
pretty_{*\to*}\ Tree_{*\to*}\ a\ Empty = text\ \texttt{"Empty"} \\
pretty_{*\to*}\ Tree_{*\to*}\ a\ (Node\ l\ x\ r) \\
\quad = align\ \texttt{"(Node "}\ (pretty_{*\to*}\ Tree_{*\to*}\ a\ l \diamond nl \diamond \\
\qquad\qquad pretty_{*}\ a\ x \diamond nl \diamond \\
\qquad\qquad pretty_{*\to*}\ Tree_{*\to*}\ a\ r \diamond text\ \texttt{")"}) \\
pretty_{*\to*}\ (f\ `App_{*,*\to*}`\ a)\ b\ x = pretty_{*\to*\to*}\ f\ a\ b\ x
\end{array}$$

$$\begin{array}{l}
\textbf{open}\ pretty_{*\to*\to*} :: Type_{*\to*\to*}\ \alpha \to Pretty_{*\to*\to*}\ \alpha \\
pretty_{*\to*\to*}\ Pair_{*\to*\to*}\ a\ b\ (x, y) = align\ \texttt{"("}\ (pretty_{*}\ a\ x) \diamond nl \diamond \\
\qquad align\ \texttt{", "}\ (pretty_{*}\ b\ y) \diamond text\ \texttt{")"}
\end{array}$$

The equations for type application have a particularly simple form.

$$poly_{\kappa}\ (App_{\iota,\kappa}\ f\ a) = poly_{\iota\to\kappa}\ f\ a$$

Type-passing style is preferable to dictionary-passing style for implementing *mutually recursive* generic functions. In dictionary-passing style we have to tuple the functions into a single dictionary (analogous to the usual implementation of Haskell's type classes). On the other hand, using dictionary-passing style we can define truly polymorphic generic functions such as, for example, $size :: Type_{*\to*}\ \varphi \to \forall\alpha\ .\ \varphi\ \alpha \to Int$, which is not possible in type-passing style where *size* has type $Type_{*\to*}\ \varphi \to \forall\alpha\ .\ Type_{*}\ \alpha \to \varphi\ \alpha \to Int$.

dictionary- and type-passing style. A kind-indexed family of overloaded functions is said to be defined in *dictionary-passing style* if the instances for type functions receive as an argument the instance (the dictionary) for the type parameter. If instead the type representation itself is passed, then the family is defined in *type-passing style*.

4.3 Representations of open type terms

Haskell's type system is somewhat peculiar, as it features type application but not type abstraction. If Haskell had type-level lambdas, we could determine the instances of $* \to *$-indexed functions using suitable type abstractions: for our running example we could use representations of $\Lambda\alpha \to List\ (Tree\ Int)$, $\Lambda\alpha \to \alpha$, $\Lambda\alpha \to List\ \alpha$, or $\Lambda\alpha \to List\ (Tree\ \alpha)$. Interestingly, there is an alternative. We can represent an anonymous type function by an *open type term*: $\Lambda\alpha \to List\ (Tree\ \alpha)$, for instance, is represented by $List\ (Tree\ a)$ where a is a suitable representation of α.

Representation types for types of a fixed kind To motivate the representation of *free* type variables, let us work through a concrete example. Consider the following version of *count* that is defined on *Type*, the original type of type representations.

$$\begin{array}{l}
count :: Type\ \alpha \to (\alpha \to Int) \\
count\ (Char) = const\ 0
\end{array}$$

$$\begin{aligned}
&count\ (Int) &&= const\ 0\\
&count\ (Pair\ a\ b) &&= \lambda(x,y) \to count\ a\ x + count\ b\ y\\
&count\ (List\ a) &&= sum_{[\,]}\ .\ map_{[\,]}\ (count\ a)\\
&count\ (Tree\ a) &&= sum_{[\,]}\ .\ map_{[\,]}\ (count\ a)\ .\ inorder
\end{aligned}$$

As it stands, *count* is point-free, but also pointless, as it always returns the constant 0 (unless the argument is not fully defined, in which case *count* is undefined, as well). We shall see in a moment that we can make *count* more useful by adding a representation of unbound type variables to *Type*. The million-dollar question is, of course, what constitutes a suitable representation of an unbound type variable? Now, if we extend *count* by a case for the unbound type variable, its meaning must be provided from somewhere. An intriguing choice is therefore to identify the type variable with its meaning. Thus, the representation of an open type variable is a constructor that embeds a *count* instance, a function of type $\alpha \to Int$, into the type of type representations.

$$Count :: (\alpha \to Int) \to Type\ \alpha$$

Since the 'type variable' carries its own meaning, the *count* instance is particularly simple.

$$count\ (Count\ c) = c$$

A moment's reflection reveals that this approach is an instance of the 'embedding trick' [9] for higher-order abstract syntax: *Count* is the pre-inverse or right inverse of *count*. Using *Count* we can specify the action on the free type variable when we call *count*:

```
Now⟩ let ts = [tree [0 .. i] | i ← [0 .. 9 :: Int]]
Now⟩ let a = Count (const 1)
Now⟩ count (List (Tree Int)) ts
0
Now⟩ count a ts
1
Now⟩ count (List a) ts
10
Now⟩ count (List (Tree a)) ts
55
```

Using a different instance we can also sum the elements of a data structure:

```
Now⟩ let a = Count id
Now⟩ count (Pair Int Int) (47, 11)
0
Now⟩ count (Pair a   Int) (47, 11)
47
Now⟩ count (Pair Int a  ) (47, 11)
```

```
11
Now⟩ count (Pair a    a  ) (47, 11)
58
```

The approach would work perfectly well if *count* were the only generic function. But it is not:

```
Now⟩ pretty (4711 : a)
*** Exception: Non-exhaustive patterns in function pretty
```

If we pass *Count* to a different generic function, we get a run-time error. Unfortunately, the problem is not easy to remedy, as it is impossible to define a suitable *Count* instance for *pretty*. We simply have not enough information to hand. There are at least two ways out of this dilemma: we can augment the representation of unbound type variables by the required information, or we can use a different representation type that additionally abstracts over the type of a generic function. Let us consider each alternative in turn.

To define a suitable equation for *pretty* or other generic functions, we basically need the representation of the instance type. Therefore we define:

```
infixl ‘Use‘
Use :: Type α → Instance α → Type α
```

where *Instance* gathers instances of generic functions:

```
open data Instance :: ∗ → ∗ where
  Pretty :: (α → Text) → Instance α
  Count  :: (α → Int)  → Instance α
```

Using the new representation, *Count c* becomes *a* ‘*Use*‘ *Count c*, where *a* is the representation of *c*'s instance type. Since *Use* couples each instance with a representation of the instance type, we can easily extend *count* and *pretty*:

```
count (Use a d) = case d of {Count c → c; otherwise → count a}
pretty (Use a d) = case d of {Pretty p → p; otherwise → pretty a}
```

The definitional scheme is the same for each generic function: we first check whether the instance matches the generic function at hand, otherwise we recurse on the type representation. It is important to note that the scheme is independent of the number of generic functions, in fact, the separate *Instance* type was introduced to make the pattern matching more robust. A type representation that involves *Use* such as *Int* ‘*Use*‘ *Count c* ‘*Use*‘ *Pretty p* :: *Type Int* can be seen as a mini-environment that determines the action of the listed generic functions at this point. The above instances of *count* and *pretty* effectively perform an environment look-up at runtime.

Let us now turn to the second alternative. The basic idea is to parameterise *Type* by the type of generic functions.

```
open data PType :: (∗ → ∗) → ∗ → ∗ where
  PChar :: PType poly Char
  PInt  :: PType poly Int
  PPair :: PType poly α → PType poly β → PType poly (α, β)
  PList :: PType poly α → PType poly [α]
  PTree :: PType poly α → PType poly (Tree α)
```

A generic function then has type *PType Poly* $\alpha \to$ *Poly* α for some suitable type *Poly*. As before, the representation of an unbound type variable is a constructor of the inverse type, except that now we additionally abstract away from *Poly*.

```
PVar :: poly α → PType poly α
```

Since we abstract over *Poly*, we make do with a single constructor: *PVar* can be used to embed instances of arbitrary generic functions.

The definition of *count* can be easily adapted to the new representation (for technical reasons, we have to introduce a **newtype** for *count*'s type).

```
newtype Count α = In_Count { out_Count :: α → Int }
pcount :: PType Count α → (α → Int)
pcount (PVar c)     = out_Count c
pcount (PChar)      = const 0
pcount (PInt)       = const 0
pcount (PPair a b)  = λ(x, y) → pcount a x + pcount b y
pcount (PList a)    = sum_[] . map_[] (pcount a)
pcount (PTree a)    = sum_[] . map_[] (pcount a) . inorder
```

The code is almost identical to what we have seen before, except that the type signature is more precise.

Here is an interactive session that illustrates the use of *pcount*.

```
Now⟩ let ts = [tree [0 . . i] | i ← [0 . . 9 :: Int]]
Now⟩ let a = PVar (In_Count (const 1))
Now⟩ :type a
a :: ∀α . PType Count α
Now⟩ pcount (PList (PTree PInt)) ts
0
Now⟩ pcount (a) ts
1
Now⟩ pcount (PList a) ts
10
Now⟩ pcount (PList (PTree a)) ts
55
Now⟩ let a = PVar (In_Count id)
Now⟩ :type a
PType Count Int
```

```
Now⟩ pcount (PList (PTree a)) ts
165
```

Note that the type of a now limits the applicability of the unbound type variable: passing it to *pretty* would result in a static type error.

We can also capture our standard idioms, counting elements and summing up integers, as abstractions.

```
psize f  = pcount (f a) where a = PVar (In_Count (const 1))
psum f = pcount (f a) where a = PVar (In_Count id)
```

Given these definitions, we can represent type constructors of kind $* \to *$ by ordinary, value-level λ-terms.

```
Now⟩ let ts = [tree [0 .. i] | i ← [0 .. 9 :: Int]]
Now⟩ psize (λa → PList (PTree PInt)) ts
0
Now⟩ psize (λa → a) ts
1
Now⟩ psize (λa → PList a) ts
10
Now⟩ psize (λa → PList (PTree a)) ts
55
Now⟩ psum (λa → PPair PInt PInt) (47, 11)
0
Now⟩ psum (λa → PPair a    PInt) (47, 11)
47
Now⟩ psum (λa → PPair PInt a   ) (47, 11)
11
Now⟩ psum (λa → PPair a    a   ) (47, 11)
58
```

It is somewhat surprising that the expressions above type-check, in particular, as Haskell does not support anonymous type functions. The reason is that we can assign *psize* and *psum* Hindler-Milner types:

```
psize :: (PType Count α   → PType Count β) → (β → Int)
psum :: (PType Count Int → PType Count β) → (β → Int)
```

The functions also possess $F\omega$ types, which are different from the types above:

```
psize :: ∀φ . PType_{*→*} Count φ → (∀α . φ α   → Int)
psum :: ∀φ . PType_{*→*} Count φ → (∀α . φ Int → Int)
```

Using $F\omega$ types, however, the above calls do not type-check, since Haskell employs a kinded first-order unification of types.

Kind-indexed families of representation types The other representation types, Type' and Type_κ, can be extended in an analogous manner to support open type terms. For instance, for Type_κ we basically have to introduce kind-indexed versions of *Use* and *Instance*.

$$\begin{array}{l}
\textbf{open data}\ \mathit{Instance}_\kappa :: \kappa \to * \ \textbf{where} \\
\quad \mathit{Poly}_\kappa :: \mathit{Poly}_\kappa\ \alpha \to \mathit{Instance}_\kappa\ \alpha \\
\mathit{Use}_\kappa :: \mathit{Type}_\kappa\ \alpha \to \mathit{Instance}_\kappa\ \alpha \to \mathit{Type}_\kappa\ \alpha \\
\mathit{poly}_\kappa\ (\mathit{Use}_\kappa\ a\ d) = \textbf{case}\ d\ \textbf{of}\ \{\mathit{Poly}_\kappa\ p \to p; \mathit{otherwise} \to \mathit{poly}_\kappa\ a\}
\end{array}$$

The reader may wish to fill in the gory details and to work through the implementation of the other combinations.

5 Views

In Section 4 we thoroughly investigated the design space of type representations. The examples in that section are without exception overloaded functions. In this section we explore various techniques to turn these overloaded functions into truly generic ones. Before we tackle this, let us first discuss the difference between *nominal* and *structural* type systems.

Haskell has a *nominal type system*: each **data** declaration introduces a new type that is incompatible with all the existing types. Two types are equal if and only if they have the same name. By contrast, in a *structural type system* two types are equal if they have the same structure. In a language with a structural type system there is no need for a generic view; a generic function can be defined exhaustively by induction on the structure of types.

For nominal systems the key to genericity is a uniform view on data. In Section 3.3 we introduced the spine view, which views data as constructor applications. Of course, this is not the only generic view. PolyP [26], for instance, views data types as fixed points of regular functors; Generic Haskell [19] uses a sum-of-products view. We shall see that these two approaches can be characterised as *type-oriented*: they provide a uniform view on all elements of a datatype. By contrast, the spine view is *value-oriented*: it provides a uniform view on single elements.

View For the following it is useful to make the concept of a view explicit.

$$\begin{array}{l}
\textbf{infixr}\ 5 \to \\
\textbf{infixl}\ 5 \leftarrow \\
\textbf{type}\ \alpha \leftarrow \beta = \beta \to \alpha \\
\textbf{data}\ \mathit{View} :: * \to * \ \textbf{where} \\
\quad \mathit{View} :: \mathit{Type}\ \beta \to (\alpha \to \beta) \to (\alpha \leftarrow \beta) \to \mathit{View}\ \alpha
\end{array}$$

A view consists of three ingredients: a so-called *structure type* that constitutes the actual view on the original datatype, and two functions that convert to and fro. To define a view the generic programmer simply provides a view function

$view :: Type\ \alpha \to View\ \alpha$

that maps a type to its structural representation. The view function can then be used in the catch-all case of a generic function. Take as an example the modified definition of *strings* (the original catch-all case is defined in Section 3.1).

```
strings (x : t) = case view t of
                    View u fromData toData → strings (fromData x : u)
```

Using one of the conversion functions, $x : t$ is converted to its structural representation $fromData\ x : u$, on which *strings* is called recursively. Because of the recursive call, the definition of *strings* must contain additional case(s) that deal with the structure type. For the spine view, a single equation suffices.

$strings\ (x : Spine\ a) = strings_{-}\ x$

Lifted view For the type $Type'$ of lifted type representations, we can set up similar machinery.

```
infixr 5 →̇
infixl 5 ←̇
type φ →̇ ψ = ∀α . φ α → ψ α
type φ ←̇ ψ = ∀α . ψ α → φ α
data View′ :: (∗ → ∗) → ∗ where
  View′ :: Type′ ψ → (φ →̇ ψ) → (φ ←̇ ψ) → View′ φ
```

The view function is now of type

$view' :: Type'\ \varphi \to View'\ \varphi$

and is used as follows:

```
map f m x = case view′ f of
              View′ g fromData toData → (toData . map g m . fromData) x
```

In this case, we require both the *fromData* and the *toData* function.

5.1 Spine view

The spine view of the type τ is simply $Spine\ \tau$:

```
spine   :: Type α → View α
spine a = View (Spine a) (λx → toSpine (x : a)) fromSpine
```

Recall that *fromSpine* is parametrically polymorphic, while *toSpine* is an overloaded function. The definition of *toSpine* follows a simple pattern: if the datatype comprises a constructor C with signature

$$C :: \tau_1 \to \cdots \to \tau_n \to \tau_0$$

then the equation for *toSpine* takes the form

$$\mathit{toSpine}\ (C\ x_1\ \ldots\ x_n : t_0) = \mathit{Con}\ c \diamond (x_1 : t_1) \diamond \cdots \diamond (x_n : t_n)$$

where c is the annotated version of C and t_i is the type representation of τ_i. The equation is only valid if $\mathit{vars}\ (t_1) \cup \cdots \cup \mathit{vars}\ (t_n) \subseteq \mathit{vars}\ (t_0)$, that is, if C's type signature contains no existentially quantified type variables (see also below).

The spine view is particularly easy to use: the generic part of a generic function only has to consider two cases: *Con* and '$\diamond$'.

A further advantage of the spine view is its generality: it is applicable to a large class of datatypes. Nested datatypes, for instance, pose no problems: the type of perfect binary trees (see Section 3.2)

$$\textbf{data}\ \mathit{Perfect}\ \alpha = \mathit{Zero}\ \alpha \mid \mathit{Succ}\ (\mathit{Perfect}\ (\alpha, \alpha))$$

gives rise to the following two equations for *toSpine*:

$$\begin{aligned}
&\mathit{toSpine}\ (\mathit{Zero}\ x : \mathit{Perfect}\ a) = \mathit{Con}\ \mathit{zero} \diamond (x : a)\\
&\mathit{toSpine}\ (\mathit{Succ}\ x : \mathit{Perfect}\ a) = \mathit{Con}\ \mathit{succ} \diamond (x : \mathit{Perfect}\ (\mathit{Pair}\ a\ a))
\end{aligned}$$

The equations follow exactly the general scheme above. We have also seen that the scheme is applicable to *generalised algebraic datatypes*. Consider as an example the typed representation of expressions (see Section 2.2).

$$\begin{aligned}
&\textbf{data}\ \mathit{Expr} :: * \to *\ \textbf{where}\\
&\quad \mathit{Num} :: \mathit{Int} \to \mathit{Expr}\ \mathit{Int}\\
&\quad \mathit{Plus} :: \mathit{Expr}\ \mathit{Int} \to \mathit{Expr}\ \mathit{Int} \to \mathit{Expr}\ \mathit{Int}\\
&\quad \mathit{Eq} \quad :: \mathit{Expr}\ \mathit{Int} \to \mathit{Expr}\ \mathit{Int} \to \mathit{Expr}\ \mathit{Bool}\\
&\quad \mathit{If} \quad\ :: \mathit{Expr}\ \mathit{Bool} \to \mathit{Expr}\ \alpha \to \mathit{Expr}\ \alpha \to \mathit{Expr}\ \alpha
\end{aligned}$$

The relevant equations for *toSpine* are

$$\begin{aligned}
&\mathit{toSpine}\ (\mathit{Num}\ i : \mathit{Expr}\ \mathit{Int}) &&= \mathit{Con}\ \mathit{num} \diamond (i : \mathit{Int})\\
&\mathit{toSpine}\ (\mathit{Plus}\ e_1\ e_2 : \mathit{Expr}\ \mathit{Int}) &&= \mathit{Con}\ \mathit{plus} \diamond (e_1 : \mathit{Expr}\ \mathit{Int}) \diamond (e_2 : \mathit{Expr}\ \mathit{Int})\\
&\mathit{toSpine}\ (\mathit{Eq}\ e_1\ e_2 : \mathit{Expr}\ \mathit{Bool}) &&= \mathit{Con}\ \mathit{eq} \diamond (e_1 : \mathit{Expr}\ \mathit{Int}) \diamond (e_2 : \mathit{Expr}\ \mathit{Int})\\
&\mathit{toSpine}\ (\mathit{If}\ e_1\ e_2\ e_3 : \mathit{Expr}\ a)\\
&\quad = \mathit{Con}\ \mathit{if} \diamond (e_1 : \mathit{Expr}\ \mathit{Bool}) \diamond (e_2 : \mathit{Expr}\ a) \diamond (e_3 : \mathit{Expr}\ a)
\end{aligned}$$

Given this definition we can apply *pretty* to values of type *Expr* without further ado. Note in this respect that the Glasgow Haskell Compiler (GHC) currently does not support **deriving** (*Show*) for GADTs. When we turned *Dynamic* into a representable type (Section 3.4), we discussed one limitation of the spine view: it cannot, in general, cope with existentially quantified types. Consider, as another example, the following extension of the expression datatype:

$$\mathit{Apply} :: \mathit{Expr}\ (\alpha \to \beta) \to \mathit{Expr}\ \alpha \to \mathit{Expr}\ \beta$$

The equation for *toSpine*

```
toSpine (Apply f x : Expr b)
   = Con apply ◇ (f : Expr (a → b)) ◇ (x : Expr a)   -- not legal Haskell
```

is not legal Haskell, as a, the representation of α, appears free on the right-hand side. The only way out of this dilemma is to augment x by a representation of its type, as in *Dynamic*.[5]

To summarise: a **data** declaration describes how to construct data; the spine view captures just this. Consequently, it is applicable to almost every datatype declaration. The other views are more restricted: Generic Haskell's original sum-of-products view is only applicable to Haskell 98 types excluding GADTs and existential types (however, we will show in Section 5.4 how to extend the sum-of-products view to GADTs); PolyP is even restricted to fixed points of regular functors excluding nested datatypes and higher kinded types.

On the other hand, the classic views provide more information, as they represent the complete datatype, not just a single constructor application. The spine view effectively restricts the class of functions we can write: one can only define generic functions that consume or transform data (such as *show*) but not ones that produce data (such as *read*). The uniform view on individual constructor applications is useful if you have data in your hands, but it is of no help if you want to construct data. We make this more precise in the following section.

Furthermore, functions that abstract over type constructors (such as *size* or *map*) are out of reach for the spine view. In the following two sections we show how to overcome both limitations.

5.2 The type-spine view

A *generic consumer* is a function of type $Type\ \alpha \to \alpha \to \tau$ ($\cong Typed\ \alpha \to \tau$), where the type we abstract over occurs in an argument position and possibly in the result type τ. We have seen in Section 3.3 that the generic part of a consumer follows the general pattern below.

```
consume :: Type α → α → τ
...
consume a x  = consume_ (toSpine (x : a))
consume_   :: Spine α → τ
consume_ ... = ...
```

The element x is converted to the spine representation, over which the helper function *consume*$_\_$ then recurses. By duality, we would expect that a generic

[5] Type-theoretically, we have to turn the existential quantifier $\exists\alpha\ .\ \tau$ into an intensional quantifier $\exists\alpha\ .\ Type\ \alpha \times \tau$. This is analogous to the difference between parametrically polymorphic functions of type $\forall\alpha\ .\ \tau$ and overloaded functions of type $\forall\alpha\ .\ Type\ \alpha \to \tau$.

producer of type $Type\ \alpha \to \tau \to \alpha$, where α appears in the result type *but not* in τ, takes on the following form.

```
produce :: Type α → τ → α
...
produce a t   = fromSpine (produce_ t)
produce_   :: τ → Spine α   -- does not work
produce_ ... = ...
```

The helper function *produce_* generates an element in spine representation, which *fromSpine* converts back. Unfortunately, this approach does not work. The formal reason is that *toSpine* and *fromSpine* are different beasts: *toSpine* is an overloaded function, while *fromSpine* is parametrically polymorphic. If it were possible to define $produce\_ :: \forall\alpha\ .\ \tau \to Spine\ \alpha$, then the composition $fromSpine\ .\ produce\_$ would yield a parametrically polymorphic function of type $\forall\alpha\ .\ \tau \to \alpha$, which is the type of an unsafe cast operation. And, indeed, a closer inspection of the catch-all case of *produce* reveals that a, the type representation of α, does not appear on the right-hand side. However, as we already know, a truly polymorphic function cannot exhibit type-specific behaviour.

Of course, this does not mean that we cannot define a function of type $Type\ \alpha \to \tau \to \alpha$. We just require additional information about the datatype, information that the spine view does not provide. Consider in this respect the syntactic form of a GADT (eg *Type* itself or *Expr* in Section 2.2): a datatype is essentially a sequence of signatures. This motivates the following definitions.

```
type Datatype α = [Signature α]
infixl 0 □
data Signature :: ∗ → ∗ where
  Sig  :: Constr α → Signature α
  (□)  :: Signature (α → β) → Type α → Signature β
```

The type *Signature* is almost identical to the *Spine* type, except for the second argument of '□', which is of type $Type\ \alpha$ rather than $Typed\ \alpha$. Thus, an element of type *Signature* contains the types of the constructor arguments, but not the arguments themselves. For that reason, *Datatype* is called the *type-spine view*.

This view is similar to the sum-of-products view (see Section 5.4): the list encodes the sum, the constructor '□' corresponds to a product and *Sig* is like the unit element. To be able to use the type spine view, we additionally require an overloaded function that maps a type representation to an element of type $Datatype\ \alpha$.

```
open datatype :: Type α → Datatype α
datatype (Bool)     = [Sig false, Sig true]
datatype (Char)     = [Sig (char c) | c ← [minBound .. maxBound]]
datatype (Int)      = [Sig (int i) | i ← [minBound .. maxBound]]
datatype (List a)   = [Sig nil, Sig cons □ a □ List a]
datatype (Pair a b) = [Sig pair □ a □ b]
datatype (Tree a)   = [Sig empty, Sig node □ Tree a □ a □ Tree a]
```

Here, *char* maps a character to its annotated variant and likewise *int*; *nil*, *cons* and *pair* are the annotated versions of *Nil*, *Cons* and '(,)'. As an aside, the second and the third equation produce rather long lists; they are only practical in a lazy setting. The function *datatype* plays the same role for producers as *toSpine* plays for *consumers*.

The first example of a generic producer is a simple test-data generator. The function *generate a d* yields all terms of the data type α up to a given finite depth d.

```
generate :: Type α → Int → [α]
generate a 0       = [ ]
generate a (d + 1) = concat [generate_ s d | s ← datatype a]

generate_ :: Signature α → Int → [α]
generate_ (Sig c) d = [constr c]
generate_ (s □ a) d = [f x | f ← generate_ s d, x ← generate a d]
```

The helper function *generate_* constructs all terms that conform to a given signature. The right-hand side of the second equation essentially computes the cartesian product of *generate_ s d* and *generate a d*. Here is a short interactive session that illustrates the use of *generate*.

```
Now⟩ generate (List Bool) 3
[[ ], [False], [False, False], [False, True], [True], [True, False], [True, True]]
Now⟩ generate (List (List Bool)) 3
[[ ], [[ ]], [[ ], [ ]], [[False]], [[False], [ ]], [[True]], [[True], [ ]]]
```

As a second example, let us define a generic parser. For concreteness, we reimplement Haskell's *readsPrec* function of type $Int \to ReadS\ \alpha$. The *Int* argument specifies the operator precedence of the enclosing context; *ReadS* abbreviates $String \to [Pair\ \alpha\ String]$, the type of backtracking parsers [25].

```
open readsPrec :: Type α → Int → ReadS α
readsPrec (Char)     d = readsPrec_Char d
readsPrec (Int)      d = readsPrec_Int d
readsPrec (String)   d = readsPrec_String d
readsPrec (List a)   d = readsList (reads a)
readsPrec (Pair a b) d
  = readParen False (λs₀ → [((x, y), s₅) | ("(", s₁) ← lex     s₀,
                                           (x,   s₂) ← reads a s₁,
                                           (",", s₃) ← lex     s₂,
                                           (y,   s₄) ← reads b s₃,
                                           (")", s₅) ← lex     s₄])
readsPrec a          d
  = alt [readParen (arity′ s > 0 ∧ d > 10) (reads_ s) | s ← datatype a]
```

The overall structure is similar to that of *pretty*. The first three equations delegate the work to tailor-made parsers. Given a parser for elements, *readsList*, defined in Appendix A.3, parses a list of elements. Pairs are read using the usual mix-fix notation. The predefined function *readParen b* takes care of optional ($b = False$) or mandatory ($b = True$) parentheses. The catch-all case implements the generic part: constructors in prefix notation. Parentheses are mandatory if the constructor has at least one argument and the operator precedence of the enclosing context exceeds 10 (the precedence of function application is 11). The parser for α is the alternation of all parsers for the individual constructors of α (*alt* is defined in Appendix A.3). The auxiliary function *reads*₋ parses a single constructor application.

```
reads_ :: Signature α → ReadS α
reads_ (Sig c) s₀ = [(constr c, s₁) | (t, s₁) ← lex s₀, name c == t]
reads_ (s □ a) s₀ = [(f x, s₂) | (f, s₁) ← reads_ s s₀,
                                  (x, s₂) ← readsPrec a 11 s₁]
```

Finally, *arity′* determines the arity of a constructor.

```
arity′ :: Signature α → Int
arity′ (Sig c) = 0
arity′ (s □ a) = arity′ s + 1
```

As for *pretty*, we can define suitable wrapper functions that simplify the use of the generic parser.

```
reads :: Type α → ReadS α
reads a  = readsPrec a 0

read  :: Type α → String → α
read a s = case [x | (x, t) ← reads a s, ("", "") ← lex t] of
             [x] → x
             []  → error "read: no parse"
             _   → error "read: ambiguous parse"
```

From the code of *generate* and *readsPrec* we can abstract a general definitional scheme for generic producers.

```
produce :: Type α → τ → α
...
produce a t   = ...[...produce_ s t ... | s ← datatype a]

produce_   :: Signature α → τ → α
produce_ ... = ...
```

The generic case is a two-step procedure: the list comprehension processes the list of constructors; the helper function *produce*₋ takes care of a single constructor.

The type-spine view is complementary to the spine view, but independent of it. The latter is used for generic producers, the former for generic consumers or transformers. This is in contrast to Generic Haskell's sum-of-products view or PolyP's fixed point view where a single view serves both purposes.

The type-spine view shares the major advantage of the spine view: it is applicable to a large class of datatypes. Nested datatypes such as the type of perfect binary trees can be handled easily:

$$\mathit{datatype}\ (\mathit{Perfect}\ a) = [\mathit{Sig}\ \mathit{zero} \mathbin{\square} a, \mathit{Sig}\ \mathit{succ} \mathbin{\square} \mathit{Perfect}\ (\mathit{Pair}\ a\ a)]$$

The scheme can even be extended to generalised algebraic datatypes. Since *Datatype* α is a homogeneous list, we have to partition the constructors according to their result types. Consider the expression datatype of Section 2.2. We have three different result types, *Expr Bool*, *Expr Int* and *Expr* α, and consequently three equations for *datatype*.

```
datatype (Expr Bool)
   = [Sig eq □ Expr Int □ Expr Int,
      Sig if □ Expr Bool □ Expr Bool □ Expr Bool]
datatype (Expr Int)
   = [Sig num □ Int,
      Sig plus □ Expr Int □ Expr Int,
      Sig if □ Expr Bool □ Expr Int □ Expr Int]
datatype (Expr a)
   = [Sig if □ Expr Bool □ Expr a □ Expr a]
```

The equations are ordered from specific to general; each right-hand side lists all the constructors that have the given result type *or* a more general one. Consequently, the *If* constructor, which has a polymorphic result type, appears in every list. Given this declaration we can easily generate well-typed expressions (for reasons of space we have modified *generate Int* so that only 0 is produced):

```
Now⟩ let gen a d = putStrLn (show (generate a d : List a))
Now⟩ gen (Expr Int) 4
[(Num 0), (Plus (Num 0) (Num 0)), (Plus (Num 0) (Plus (Num 0) (Num
0))), (Plus (Plus (Num 0) (Num 0)) (Num 0)), (Plus (Plus (Num 0) (Num
0)) (Plus (Num 0) (Num 0))), (If (Eq (Num 0) (Num 0)) (Num 0) (Num
0)), (If (Eq (Num 0) (Num 0)) (Num 0) (Plus (Num 0) (Num 0))), (If (Eq
(Num 0) (Num 0)) (Plus (Num 0) (Num 0)) (Num 0)), (If (Eq (Num 0)
(Num 0)) (Plus (Num 0) (Num 0)) (Plus (Num 0) (Num 0)))]
Now⟩ gen (Expr Bool) 4
[(Eq (Num 0) (Num 0)), (Eq (Num 0) (Plus (Num 0) (Num 0))), (Eq (Plus
(Num 0) (Num 0)) (Num 0)), (Eq (Plus (Num 0) (Num 0)) (Plus (Num 0)
(Num 0))), (If (Eq (Num 0) (Num 0)) (Eq (Num 0) (Num 0)) (Eq (Num 0)
(Num 0)))]
Now⟩ gen (Expr Char) 4
[]
```

The last call shows that there are no character expressions of depth 4.

In general, for each constructor C with signature

$$C :: \tau_1 \to \cdots \to \tau_n \to \tau_0$$

we add an element of the form

$$\mathit{Sig}\ c \mathbin{\square} t_1 \mathbin{\square} \cdots \mathbin{\square} t_n$$

to each right-hand side of *datatype* t provided τ_0 is more general than τ.

5.3 Lifted spine view

We have already mentioned that the original spine view is not suitable for defining $* \to *$-indexed functions, as it cannot capture type abstractions. To illustrate, consider a variant of *Tree* whose inner nodes are annotated with an integer, say, a balance factor.

$$\textbf{data}\ \mathit{BalTree}\ \alpha = \mathit{Empty} \mid \mathit{Node}\ \mathit{Int}\ (\mathit{BalTree}\ \alpha)\ \alpha\ (\mathit{BalTree}\ \alpha)$$

If we call the generic function on a value of type *BalTree Int*, then the two integer components are handled in a uniform way. This is fine for generic functions on types, but not acceptable for generic functions on type constructors. For instance, a generic version of *sum* must consider the label of type $\alpha = \mathit{Int}$, but ignore the balance factor of type *Int*. In the sequel we introduce a suitable variant of *Spine* that can be used to define the latter brand of generic functions.

A constructor of a lifted type has the signature $\forall \chi\ .\ \tau_1'\ \chi \to \cdots \to \tau_n'\ \chi \to \tau_0'\ \chi$ where the type variable χ marks the parametric components. We can write the signature more perspicuously as $\forall \chi\ .\ (\tau_1' \to' \cdots \to' \tau_n' \to' \tau_0')\ \chi$, using the lifted function space:

$$\begin{aligned}
&\textbf{infixr} \to' \\
&\textbf{newtype}\ (\varphi \to' \psi)\ \chi = \mathit{Fun}\{\, \mathit{app} :: \varphi\ \chi \to \psi\ \chi \}
\end{aligned}$$

For technical reasons, '$\to'$' must be defined by a **newtype** rather than a **type** declaration.[6] As an example, here are variants of Nil' and Cons':

$$\begin{aligned}
&\mathit{nil}' &&:: \forall \chi\ .\ \forall \alpha'\ .\ (\mathit{List}'\ \alpha')\ \chi \\
&\mathit{nil}' &&= \mathit{Nil}' \\
&\mathit{cons}' &&:: \forall \chi\ .\ \forall \alpha'\ .\ (\alpha' \to' \mathit{List}'\ \alpha' \to' \mathit{List}'\ \alpha')\ \chi \\
&\mathit{cons}' &&= \mathit{Fun}\ (\lambda x \to \mathit{Fun}\ (\lambda xs \to \mathit{Cons}'\ x\ xs))
\end{aligned}$$

An element of a lifted type can always be put into the applicative form c' '*app*' e_1 '*app*' $\cdots$ '*app*' e_n. As in the first-order case we can make this structure visible and accessible by marking the constructor and the function applications.

$$\begin{aligned}
&\textbf{data}\ \mathit{Spine}' :: (* \to *) \to * \to *\ \textbf{where} \\
&\quad \mathit{Con}' :: (\forall \chi\ .\ \varphi\ \chi) \to \mathit{Spine}'\ \varphi\ \alpha \\
&\quad (\diamond') \ :: \mathit{Spine}'\ (\varphi \to' \psi)\ \alpha \to \mathit{Typed}'\ \varphi\ \alpha \to \mathit{Spine}'\ \psi\ \alpha
\end{aligned}$$

[6] In Haskell, types introduced by **type** declarations cannot be partially applied.

The structure of Spine' is very similar to that of Spine, except that we are now working in a higher realm: Con' takes a *polymorphic function* of type $\forall\chi\,.\,\varphi\ \chi$ to an element of $\mathit{Spine}'\ \varphi$; the constructor '$\diamond'$' applies an element of type $\mathit{Spine}'\ (\varphi \rightarrow' \psi)$ to a $\mathit{Typed}'\ \varphi$ yielding an element of type $\mathit{Spine}'\ \psi$.

Turning to the conversion functions, *fromSpine′* is again *polymorphic*.

$$\begin{aligned}
&\mathit{fromSpine}' :: \mathit{Spine}'\ \varphi\ \alpha \rightarrow \varphi\ \alpha\\
&\mathit{fromSpine}'\ (\mathit{Con}'\ c) = c\\
&\mathit{fromSpine}'\ (f \diamond' x) \quad = \mathit{fromSpine}'\ f\ \text{`}\mathit{app}\text{`}\ \mathit{val}'\ x
\end{aligned}$$

Its inverse is an *overloaded* function that follows a pattern similar to *toSpine*: each constructor C' with signature

$$C' :: \forall\chi\,.\,\tau_1'\ \chi \rightarrow \cdots \rightarrow \tau_n'\ \chi \rightarrow \tau_0'\ \chi$$

gives rise to an equation of the form

$$\mathit{toSpine}'\ (C'\ x_1\ \ldots\ x_n :' t_0') = \mathit{Con}\ c' \diamond (x_1 : t_1') \diamond \cdots \diamond (x_n : t_n')$$

where c' is the variant of C' that uses the lifted function space and t_i' is the type representation of the lifted type τ_i'. As an example, here is the instance for lifted lists.

$$\begin{aligned}
&\mathit{toSpine}' :: \mathit{Typed}'\ \varphi\ \alpha \rightarrow \mathit{Spine}'\ \varphi\ \alpha\\
&\mathit{toSpine}'\ (\mathit{Nil}' :' \mathit{List}'\ a') \qquad\qquad = \mathit{Con}'\ \mathit{nil}'\\
&\mathit{toSpine}'\ (\mathit{Cons}'\ x\ \mathit{xs} :' \mathit{List}'\ a') = \mathit{Con}'\ \mathit{cons}' \diamond' (x :' a') \diamond' (\mathit{xs} :' \mathit{List}'\ a')
\end{aligned}$$

The equations are surprisingly close to those of *toSpine*; pretty much the only difference is that *toSpine′* works on lifted types.

Let us make the generic view explicit. In our case, the structure view of φ is simply $\mathit{Spine}'\ \varphi$.

$$\begin{aligned}
&\mathit{Spine}' :: \mathit{Type}'\ \varphi \rightarrow \mathit{Type}'\ (\mathit{Spine}'\ \varphi)\\
&\mathit{spine}' :: \mathit{Type}'\ \varphi \rightarrow \mathit{View}'\ \varphi\\
&\mathit{spine}'\ a' = \mathit{View}'\ (\mathit{Spine}'\ a')\ (\lambda x \rightarrow \mathit{toSpine}'\ (x :' a'))\ \mathit{fromSpine}'
\end{aligned}$$

Given these prerequisites we can turn *size* (see Section 4.1) into a generic function.

$$\begin{aligned}
&\mathit{size}\ (x :' \mathit{Spine}'\ a') = \mathit{size}_{\_}\ x\\
&\mathit{size}\ (x :' a') \qquad\quad = \mathbf{case}\ \mathit{spine}'\ a'\ \mathbf{of}\\
&\qquad\qquad\qquad\qquad\quad \mathit{View}'\ b'\ \mathit{from}\ \mathit{to} \rightarrow \mathit{size}\ (\mathit{from}\ x :' b')
\end{aligned}$$

The catch-all case applies the spine view: the argument x is converted to the structure type, on which *size* is called recursively. Currently, the structure type is always of the form $\mathit{Spine}'\ \varphi$ (this will change in a moment), so the first equation applies, which in turn delegates the work to the helper function $\mathit{size}_{\_}$.

$$\begin{array}{l}
\mathit{size}_{\_} :: \mathit{Spine}'\ \varphi\ \alpha \to \mathit{Int} \\
\mathit{size}_{\_}\ (\mathit{Con}'\ c) = 0 \\
\mathit{size}_{\_}\ (f \diamond' x)\ \ = \mathit{size}_{\_}\ f + \mathit{size}\ x
\end{array}$$

The implementation of *size*$_{\_}$ is entirely straightforward: it traverses the spine, summing up the sizes of the constructor's arguments. It is worth noting that the catch-all case of *size* subsumes all the previous instances except the one for *Id*, as we cannot provide a *toSpine*′ instance for the identity type. In other words, the generic programmer has to take care of essentially three cases: *Id*, *Con*′ and '$\diamond'$'.

As a second example, here is an implementation of the generic mapping function:

$$\begin{array}{lll}
\mathit{map} :: \mathit{Type}'\ \varphi \to (\alpha \to \beta) \to (\varphi\ \alpha \to \varphi\ \beta) & & \\
\mathit{map}\ \mathit{Id} & m = & \mathit{In}_{\mathit{Id}}\ .\ m\ .\ \mathit{out}_{\mathit{Id}} \\
\mathit{map}\ (\mathit{Spine}'\ a') & m = & \mathit{map}_{\_}\ m \\
\mathit{map}\ a' & m = & \mathbf{case}\ \mathit{spine}'\ a'\ \mathbf{of} \\
 & & \quad \mathit{View}'\ b'\ \mathit{from}\ \mathit{to} \to \mathit{to}\ .\ \mathit{map}\ b'\ m\ .\ \mathit{from}
\end{array}$$

$$\begin{array}{l}
\mathit{map}_{\_} :: (\alpha \to \beta) \to (\mathit{Spine}'\ \varphi\ \alpha \to \mathit{Spine}'\ \varphi\ \beta) \\
\mathit{map}_{\_}\ m\ (\mathit{Con}'\ c) \qquad\quad = \mathit{Con}'\ c \\
\mathit{map}_{\_}\ m\ (f \diamond' (x :' a')) = \mathit{map}_{\_}\ m\ f \diamond' (\mathit{map}\ a'\ m\ x :' a')
\end{array}$$

The definition is stunningly simple: the argument function m is applied in the *Id* case; the helper function *map*$_{\_}$ applies *map* to each argument of the constructor. Note that the mapping function is of type $\mathit{Type}'\ \varphi \to (\alpha \to \beta) \to (\varphi\ \alpha \to \varphi\ \beta)$ rather than $(\alpha \to \beta) \to (\mathit{Typed}'\ \varphi\ \alpha \to \varphi\ \beta)$. Both variants are interchangeable, so picking one is just a matter of personal taste.

Bridging the gap We have noted in Section 4.1 that the generic size function does not work on the original, unlifted types, as they are different from the lifted ones. However, both are closely related: if τ' is the lifted variant of τ, then $\tau'\ \mathit{Id}$ is isomorphic to τ [14]. (This relation only holds for Haskell 98 types, not for GADTs; see also below.) Even more, $\tau'\ \mathit{Id}$ and τ can share the same run-time representation, since *Id* is defined by a **newtype** declaration and since the lifted datatype τ' has exactly the same structure as the original datatype τ.

As an example, the functions $\mathit{fromList}\ \mathit{In}_{\mathit{Id}}$ and $\mathit{toList}\ \mathit{out}_{\mathit{Id}}$ exhibit the isomorphism between $[\,]$ and $\mathit{List}'\ \mathit{Id}$.

$$\begin{array}{l}
\mathit{fromList} :: (\alpha \to \alpha'\ \chi) \to ([\alpha] \to \mathit{List}'\ \alpha'\ \chi) \\
\mathit{fromList}\ \mathit{from}\ \mathit{Nil} \qquad\qquad = \mathit{Nil}' \\
\mathit{fromList}\ \mathit{from}\ (\mathit{Cons}\ x\ \mathit{xs}) = \mathit{Cons}'\ (\mathit{from}\ x)\ (\mathit{fromList}\ \mathit{from}\ \mathit{xs}) \\
\mathit{toList} \quad :: (\alpha'\ \chi \to \alpha) \to (\mathit{List}'\ \alpha'\ \chi \to [\alpha]) \\
\mathit{toList}\ \mathit{to}\ \mathit{Nil}' \qquad\qquad = \mathit{Nil} \\
\mathit{toList}\ \mathit{to}\ (\mathit{Cons}'\ x\ \mathit{xs}) = \mathit{Cons}\ (\mathit{to}\ x)\ (\mathit{toList}\ \mathit{to}\ \mathit{xs})
\end{array}$$

Operationally, if the types $\tau'\ \mathit{Id}$ and τ have the same run-time representation, then $\mathit{fromList}\ \mathit{In}_{\mathit{Id}}$ and $\mathit{toList}\ \mathit{out}_{\mathit{Id}}$ are identity functions (the Haskell Report [37] guarantees this for $\mathit{In}_{\mathit{Id}}$ and $\mathit{out}_{\mathit{Id}}$).

We can use the isomorphism to broaden the scope of generic functions to unlifted types. To this end we simply re-use the view mechanism.

```
spine′ List = View′ (List′ Id) (fromList In_Id) (toList out_Id)
```

The following interactive session illustrates the use of *size*.

```
Now⟩ let ts = [tree [0 . . i :: Int] | i ← [0 . . 9]]
Now⟩ size (ts :′ List)
10
Now⟩ size (fromList (fromTree In_Int′) ts :′ List′ (Tree′ Int′))
0
Now⟩ size (In_Id ts :′ Id)
1
Now⟩ size (fromList In_Id ts :′ List′ Id)
10
Now⟩ size (fromList (fromTree In_Id) ts :′ List′ (Tree′ Id))
55
```

With the help of the conversion functions we can implement each of the four different views on a list of trees of integers. Since Haskell employs a kinded first-order unification of types [27], the calls almost always additionally involve a change on the value level. The type equation $\varphi\ \tau = \mathit{List}\ (\mathit{Tree}\ \mathit{Int})$ is solved by setting $\varphi = \mathit{List}$ and $\tau = \mathit{Tree}\ \mathit{Int}$, that is, Haskell picks one of the four higher-order unifiers. Only in this particular case we need not change the representation of values: *size* (*ts* :′ *List*) implements the intended call. In the other cases, $\mathit{List}\ (\mathit{Tree}\ \mathit{Int})$ must be rearranged so that the unification with $\varphi\ \tau$ yields the desired choice.

Discussion The lifted spine view is almost as general as the original spine view: it is applicable to all datatypes that are definable in Haskell 98. In particular, nested datatypes can be handled with ease. As an example, for the datatype *Perfect* (see Section 3.2), we introduce a lifted variant

```
data Perfect′ α′ χ = Zero′ (α′ χ) | Succ′ (Perfect′ (Pair′ α′ α′) χ)
Perfect  :: Type′ Perfect
Perfect′ :: Type′ φ → Type′ (Perfect′ φ)
toSpine′ (Zero′ x :′ Perfect′ a′) = Con′ zero′ ◇′ (x :′ a′)
toSpine′ (Succ′ x :′ Perfect′ a′) = Con′ succ′ ◇′ (x :′ Perfect′ (Pair′ a′ a′))
```

and functions that convert between the lifted and the unlifted variant.

```
spine′ (Perfect)
   = View′ (Perfect′ Id) (fromPerfect In_Id) (toPerfect out_Id)
fromPerfect :: (α → α′ χ) → (Perfect α → Perfect′ α′ χ)
fromPerfect from (Zero x) = Zero′ (from x)
fromPerfect from (Succ x) = Succ′ (fromPerfect (fromPair from from) x)
toPerfect     :: (α′ χ → α) → (Perfect′ α′ χ → Perfect α)
toPerfect to (Zero′ x) = Zero (to x)
toPerfect to (Succ′ x) = Succ (toPerfect (toPair to to) x)
```

The following interactive session shows some examples involving perfect trees.

```
Now⟩ size (Succ (Zero (1,2)) :′ Perfect)
2
Now⟩ map (Perfect) (+1) (Succ (Zero (1,2)))
Succ (Zero (2,3))
```

We have seen that the spine view is also applicable to *generalised algebraic datatypes*. This does not hold for the lifted spine view, as it is not possible to generalise *size* or *map* to GADTs. Consider the expression datatype of Section 2.2. Though $Expr$ is parameterised, it is not a container type: an element of $Expr\ Int$, for instance, is an expression that evaluates to an integer; it is not a data structure that contains integers. This means, in particular, that we cannot define a mapping function $(\alpha \to \beta) \to (Expr\ \alpha \to Expr\ \beta)$: How could we possibly turn expressions of type $Expr\ \alpha$ into expressions of type $Expr\ \beta$? The type $Expr\ \beta$ might not even be inhabited: there are, for instance, no terms of type $Expr\ String$. Since the type argument of $Expr$ is not related to any component, $Expr$ is also called a *phantom type* [31, 16].

It is instructive to see where the attempt to generalise *size* or *map* to GADTs fails technically. We can, in fact, define a lifted version of the $Expr$ type (we confine ourselves to one constructor).

```
data Expr′ :: (∗ → ∗) → ∗ → ∗ where
  Num′ :: Int′ χ → Expr′ Int′ χ
```

However, we cannot establish an isomorphism between $Expr$ and $Expr'\ Id$: the following code simply does not type-check.

```
fromExpr :: (α → α′ χ) → (Expr α → Expr′ α′ χ)
fromExpr from (Num i) = Num′ (In_Int′ i)   -- wrong: does not type-check
```

The isomorphism between τ and $\tau'\ Id$ only holds for Haskell 98 types.

We have seen two examples of generic consumers or transformers. As in the first-order case, generic producers are out of reach, and for exactly the same reason: *fromSpine′* is a polymorphic function while *toSpine′* is overloaded. Of course, the solution to the problem suggests itself: we must also lift the type-spine view to type constructors of kind $* \to *$. In a sense, the spine view really comprises two views: one for consumers and transformers and one for pure producers.

The spine view can even be lifted to *kind indices* of arbitrary kinds. The generic programmer then has to consider two cases for the spine view and additionally n cases, one for each of the n projection types $Out_1, \ldots, Out_n$.

Introducing lifted types for each possible type index sounds like a lot of work. Note, however, that the declarations can be generated completely mechanically (a compiler could do this easily). Furthermore, we have already noted that generic functions that are indexed by higher kinds, for instance, by $(* \to *) \to * \to *$ are rare. In practice, most generic functions are indexed by a first-order kind such as $*$ or $* \to *$.

5.4 Sum of products

Let us now turn to the 'classic' view of generic programming: the *sum-of-products view*, which is inspired by the semantics of datatypes. Consider the schematic form of a Haskell 98 **data** declaration.

$$\textbf{data}\ T\ \alpha_1\ \ldots\ \alpha_s = C_1\ \tau_{1,1}\ \ldots\ \tau_{1,m_1} \mid \cdots \mid C_n\ \tau_{n,1}\ \ldots\ \tau_{n,m_n}$$

The **data** construct combines several features in a single coherent form: type abstraction, n-ary disjoint sums, n-ary cartesian products and type recursion. We have already the machinery in place to deal with type abstraction (type application) and type recursion: using type reflection, the type-level constructs are mapped onto value abstraction and value recursion. It remains to model n-ary sums and n-ary products. The basic idea is to reduce the n-ary constructs to binary sums and binary products.

```
infixr 7 ×
infixr 6 +
data Zero
data Unit   = Unit
data α + β = Inl α | Inr β
data α × β = Pair{ outl :: α, outr :: β}
```

The *Zero* datatype, the empty sum, is used for encoding datatypes with no constructors; the *Unit* datatype, the empty product, is used for encoding constructors with no arguments. If a datatype has more than two alternatives or a constructor more than two arguments, then the binary constructors '+' and '×' are nested accordingly. With respect to the nesting there are several choices: we can use a right-deep or a left-deep nesting, a list-like nesting or a (balanced) tree-like nesting [33]. For the following examples, we choose – more or less arbitrarily – a tree-like encoding.

We first add suitable constructors to the type of type representations.

```
infixr 7 ×
infixr 6 +
0    :: Type Zero
1    :: Type Unit
(+) :: Type α → Type β → Type (α + β)
(×) :: Type α → Type β → Type (α × β)
```

The view function for the sum-of-products view is slightly more elaborate than the one for the spine view, as each datatype has a tailor-made structure type: *Bool* has the structure type *Unit* + *Unit*, $[\alpha]$ has *Unit* + α × $[\alpha]$ and finally *Tree* α has *Unit* + *Tree* α × α × *Tree* α.

```
structure :: Type α → View α
structure Bool = View (1 + 1) fromBool toBool
```

```
    where
    fromBool :: Bool → Unit + Unit
    fromBool False    = Inl Unit
    fromBool True     = Inr Unit
    toBool    :: Unit + Unit → Bool
    toBool (Inl Unit) = False
    toBool (Inr Unit) = True
  structure (List a) = View (1 + a × List a) fromList toList
    where
    fromList :: [α] → Unit + α × [α]
    fromList Nil              = Inl Unit
    fromList (Cons x xs)      = Inr (Pair x xs)
    toList    :: Unit + α × [α] → [α]
    toList (Inl Unit)         = Nil
    toList (Inr (Pair x xs))  = Cons x xs
  structure (Tree a) = View (1 + Tree a × a × Tree a) fromTree toTree
    where
    fromTree :: Tree α → Unit + Tree α × α × Tree α
    fromTree Empty                   = Inl Unit
    fromTree (Node l x r)            = Inr (Pair l (Pair x r))
    toTree    :: Unit + Tree α × α × Tree α → Tree α
    toTree (Inl Unit)                = Empty
    toTree (Inr (Pair l (Pair x r))) = Node l x r
```

Two points are worth noting. First, we only provide structure types for concrete types that are given by a **data** or a **newtype** declaration. Abstract types including primitive types such as *Char* or *Int* cannot be treated generically; for these types the generic programmer has to provide ad-hoc cases. Second, the structure types are not recursive: they express just the top 'layer' of a data element. The tail of the encoded list, for instance, is again of type $[\alpha]$, the original list datatype. We could have used explicit recursion operators but these are clumsy and hard to use in practice. Using an implicit approach to recursion has the advantage that there is no problem with mutually recursive datatypes, nor with datatypes with many parameters.

A distinct advantage of the sum-of-products view is that it provides more information than the spine view, as it represents the complete data type, not just a single constructor application. Consequently, the sum-of-products view can be used for defining both consumers and producers. The function *memo*, which memoises a given function, is an intriguing example of a function that both analyses and synthesises values of the generic type.

```
  memo :: Type α → (α → ν) → (α → ν)
  memo Char   f c         = f c   -- no memoisation
  memo Int    f i         = f i   -- no memoisation
  memo 1      f Unit      = $f_{Unit}$
    where $f_{Unit}$           = f Unit
```

```
memo (a + b) f (Inl x)      = f_Inl x
  where f_Inl               = memo a (λx → f (Inl x))
memo (a + b) f (Inr y)      = f_Inr y
  where f_Inr               = memo b (λy → f (Inr y))
memo (a × b) f (Pair x y)   = (f_Pair x) y
  where f_Pair              = memo a (λx → memo b (λy → f (Pair x y)))
memo a         f x          = f_View x
  where f_View              = case structure a of
                                View b from to → memo b (f . to) . from
```

To see how *memo* works, note that the helper definitions f_{Unit}, f_{Inl}, f_{Inr}, f_{Pair} and f_{View} do not depend on the actual argument of f. Thus, once f is given, they can be readily computed. Memoisation relies critically on the fact that they are computed only on demand and then at most once. This is guaranteed if the implementation is *fully lazy*. Usually, memoisation is defined as the composition of a function that constructs a memo table and a function that queries the table [13]. If we fuse the two functions, thereby eliminating the memo data structure, we obtain the *memo* function above. Despite appearances, the memo data structures did not vanish into thin air. Rather, they are now built into the closures. For instance, the memo table for a disjoint union is a pair of memo tables. The closure for *memo* $(a + b)$ f consequently contains a pair of memoised functions, namely f_{Inl} and f_{Inr}.

The sum-of-products view is also preferable when the generic function has to relate different elements of a datatype, the paradigmatic example being ordering.

```
compare :: Type α → α → α → Ordering
compare Char    c1          c2          = compare_Char c1 c2
compare Int     i1          i2          = compare_Int  i1 i2
compare 1       Unit        Unit        = EQ
compare (a + b) (Inl x1)    (Inl x2)    = compare a x1 x2
compare (a + b) (Inl x1)    (Inr y2)    = LT
compare (a + b) (Inr y1)    (Inl x2)    = GT
compare (a + b) (Inr y1)    (Inr y2)    = compare b y1 y2
compare (a × b) (Pair x1 y1) (Pair x2 y2)
    = case compare a x1 x2 of
        EQ → compare b y1 y2
        ord → ord
compare a       x1          x2
    = case structure a of
        View b from to → compare b (from x1) (from x2)
```

The central part of the definition is the case for sums: if the constructors are equal, then we recurse on the arguments, otherwise we immediately return the relative ordering (assuming *Inl* < *Inr*). The case for products implements the so-called *lexicographic ordering*: the ordering of two pairs is determined by the first elements, only if they are equal, we recurse on the second elements.

Implementing *compare* using the spine view faces the problem that the elements of a spine possess existentially quantified types: even if we know that the constructors of two values are identical, we cannot conclude that the types of corresponding arguments are the same — and, indeed, this property fails, for instance, for the type *Dynamic*. Consequently, a spine-based implementation of *compare* must either involve a dynamic type-equality check, or the type of compare must be generalised to

```
compare :: Type α → α → Type β → β → Ordering
```

The latter twist is not without problems, as we have to relate elements of different types.

The sum-of-products view in its original form is more restricted than the spine view: it is only applicable to Haskell 98 datatypes. However, using a similar technique to that in Section 5.2 we can to broaden the scope of the sum-of-products view to include generalised algebraic datatypes. A GADT introduces a family of Haskell 98 types indexed by the type argument of the GADT. If we partition the constructors according to their result types, we can provide an individual view for each instance. Consider the expression datatype of Section 2.2. We have three different result types, *Expr Bool*, *Expr Int* and *Expr α*, and consequently three equations for *structure*.

```
structure (Expr Bool) = View expr fromExpr toExpr
  where
  expr                       = Expr Int × Expr Int +
                               Expr Bool × Expr Bool × Expr Bool
  fromExpr (Eq x₁ x₂)        = Inl (Pair x₁ x₂)
  fromExpr (If x₁ x₂ x₃)     = Inr (Pair x₁ (Pair x₂ x₃))
  toExpr (Inl (Pair x₁ x₂)) = Eq x₁ x₂
  toExpr (Inr (Pair x₁ (Pair x₂ x₃)))
                             = If x₁ x₂ x₃
structure (Expr Int)  = View expr fromExpr toExpr
  where
  expr                       = Int +
                               Expr Int × Expr Int +
                               Expr Bool × Expr Int × Expr Int
  fromExpr (Num i)           = Inl i
  fromExpr (Plus x₁ x₂)      = Inr (Inl (Pair x₁ x₂))
  fromExpr (If x₁ x₂ x₃)     = Inr (Inr (Pair x₁ (Pair x₂ x₃)))
  toExpr (Inl i)             = Num i
  toExpr (Inr (Inl (Pair x₁ x₂)))
                             = Plus x₁ x₂
  toExpr (Inr (Inr (Pair x₁ (Pair x₂ x₃))))
                             = If x₁ x₂ x₃
structure (Expr a)    = View expr fromExpr toExpr
```

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{where}} \\
\mathit{expr} & = \mathit{Expr}\ \mathit{Bool} \times \mathit{Expr}\ a \times \mathit{Expr}\ a \\
\mathit{fromExpr}\ (\mathit{If}\ x_1\ x_2\ x_3) & = \mathit{Pair}\ x_1\ (\mathit{Pair}\ x_2\ x_3) \\
\mathit{toExpr}\ (\mathit{Pair}\ x_1\ (\mathit{Pair}\ x_2\ x_3)) & = \mathit{If}\ x_1\ x_2\ x_3
\end{array}
$$

For the details we refer to the description of *datatype* in Section 5.2.

5.5 Lifted sums of products

The sum-of-products view can be quite easily adapted to the type Type' of lifted type representations. We only have to lift the type constructors of the structure types.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{infixr}\ 7 \times'} \\
\multicolumn{2}{l}{\mathbf{infixr}\ 6 +'} \\
\multicolumn{2}{l}{\mathbf{data}\ \mathit{Zero}'\ \alpha} \\
\mathbf{data}\ \mathit{Unit}'\ \alpha & = \mathit{Unit}' \\
\mathbf{data}\ (\varphi +' \psi)\ \alpha & = \mathit{Inl}'\ (\varphi\ \alpha) \mid \mathit{Inr}'\ (\psi\ \alpha) \\
\mathbf{data}\ (\varphi \times' \psi)\ \alpha & = \mathit{Pair}'\{\mathit{outl}' :: \varphi\ \alpha, \mathit{outr}' :: \psi\ \alpha\}
\end{array}
$$

The reader may wish to fill in the details.

6 Related work

There is a wealth of material on the subject of generic programming. The tutorials of previous summer schools [2, 19, 18] provide an excellent overview of the field.

We have seen that support for generic programming consists of three essential ingredients:

– a type reflection mechanism,
– a type representation, and
– a generic view on data.

The first two items provide a way to write overloaded functions, and the third a way to access the structure of values in a uniform way. The different approaches to generic programming can be faithfully classified along these dimensions. Figure 1 provides an overview of the design space. Since the type representation is closely coupled to the generic view, we have omitted the representation dimension. The two remaining dimensions are largely independent of each other and for each there are various choices. Overloaded functions can be expressed using

– *type reflection*: This is the approach we have used in these lecture notes. Its origins can be traced back to the work on intensional type analysis [11, 8, 7, 40, 43] (ITA). ITA is intensively used in typed intermediate languages,

view(s)	representation of overloaded functions			
	type reflection	type classes	type-safe cast	specialisation
none	ITA [11, 8, 7, 40, 43]	–	–	–
fixed point	Reloaded [22]	PolyP [35, 36]	–	PolyP [26]
sum-of-products	LIGD [5, 16]	DTC [24], GC [1], GM [17]	–	GH [15, 19, 33, 34]
spine	Reloaded [22], Revolutions [21]	SYB [30], Reloaded [23]	SYB [38, 29]	–

Fig. 1. Generic programming: the design space.

in particular, for optimising purely polymorphic functions. Type reflection avoids the duplication of features: a type case, for instance, boils down to an ordinary **case** expression. Cheney and Hinze [5] present a library for generics and dynamics (LIGD) that uses an *encoding* of type representations in Haskell 98 augmented by existential types.

– *type classes* [10]: Type classes are Haskell's major innovation for supporting ad-hoc polymorphism. A type class declaration corresponds to the type signature of an overloaded value — or rather, to a collection of type signatures. An instance declaration is related to a type case of an overloaded value. For a handful of built-in classes, Haskell provides support for genericity: by attaching a **deriving** clause to a **data** declaration the Haskell compiler automatically generates an appropriate instance of the class. *Derivable type classes* (DTC) generalise this feature to arbitrary user-defined classes. A similar, but more expressive variant is implemented in Generic Clean [1] (GC). Clean's type classes are indexed by kind so that a single generic function can be applied to type constructors of different kinds. A pure Haskell 98 implementation of generics (GM) is described by Hinze [17]. The implementation builds upon a class-based encoding of the type *Type* of type representations.
– *type-safe cast* [44]: A cast operation converts a value from one type to another, provided the two types are identical at run-time. A cast can be seen as a type-case with exactly one branch. The original SYB paper [38] is based on casts.
– *specialisation* [14]: This implementation technique transforms an overloaded function into a family of polymorphic functions (*dictionary translation*). While the other techniques can be used to write a library for generics, specialisation is mainly used for implementing full-fledged generic programming systems such as PolyP [26] or *Generic Haskell* [34], that are set up as preprocessors or compilers.

The approaches differ mostly in syntax and style, but less in expressiveness — except perhaps for specialisation, which cannot cope with higher-order generic functions. The second dimension, the generic view, has a much larger impact: we have seen that it affects the set of datatypes we can cover, the class of functions we can write and potentially the efficiency of these functions.

– *no view*: Haskell has a *nominal type system*: each **data** declaration introduces a new type that is incompatible with all the existing types. Two types are equal if and only if they have the same name. By contrast, in a *structural type system* two types are equal if they have the same structure. In a language with a structural type system, there is no need for a generic view; a generic function can be defined exhaustively by induction on the structure of types. The type systems that underlie ITA are structural.
– *fixed point view*: PolyP [26] views data types as fixed points of regular functors, which are in turn represented as lifted sums of products. This view is quite limited in applicability: only datatypes of kind $* \to *$ that are regular can be represented, excluding nested datatypes and higher kinded datatypes. Its particular strength is that recursion patterns such as cata- or anamorphisms can be expressed generically, because each datatype is viewed as a fixed point, and the points of recursion are visible. The original implementation of PolyP is set up as a preprocessor that translates PolyP code into Haskell. A later version [35] embeds PolyP programs into Haskell augmented by multiple parameter type classes with functional dependencies [28]. Oliveira and Gibbons [36] present a lightweight variant of PolyP that works within Haskell 98.
– *sum-of-products view*: Generic Haskell [19, 33, 34] (GH) builds upon this view. In its original form it is applicable to all datatypes definable in Haskell 98. We have seen in Section 5.4 that it can be generalised to GADTs. Generic Haskell is a full-fledged implementation of generics based on ideas by Hinze [15, 20] that features generic functions, generic types and various extensions such as default cases and constructor cases [6]. Generic Haskell supports the definition of functions that work for all types of all kinds, such as, for example, a generalised mapping function.
– *spine views*: The spine view treats data uniformly as constructor applications. The SYB approach has been developed by Lämmel and Peyton Jones in a series of papers [38, 29, 30]. The original approach is *combinator-based*: the user writes generic functions by combining a few generic primitives. The first paper [38] introduces two main combinators: a type-safe cast for defining ad-hoc cases and a generic recursion operator, called *gfoldl*, for implementing the generic part. It turns out that *gfoldl* is essentially the catamorphism of the *Spine* datatype [22]: *gfoldl* equals the catamorphism composed with *toSpine*. The second paper [29] adds a function called *gunfold* to the set of predefined combinators, which is required for defining generic producers. The name suggests that the new combinator is the anamorphism of the *Spine* type, but it is not: *gunfold* is actually the catamorphism of *Signature*, introduced in Section 5.2.

A Library

This appendix presents some auxilliary functions used in the main part of the chapter, but relegated here so as not to disturb the flow.

A.1 Binary trees

The function *inorder*, used in Section 3.1, yields the elements of a tree in symmetric order.

```
inorder :: ∀α . Tree α → [α]
inorder Empty        = Nil
inorder (Node l a r) = inorder l ++ [a] ++ inorder r
```

The function *tree*, also used in Section 3.1, turns a list of elements into a balanced binary tree, a so-called *Braun tree* [4].

```
tree :: ∀α . [α] → Tree α
tree x
  | null x                = Empty
  | otherwise             = Node (tree x₁) a (tree x₂)
  where (x₁, Cons a x₂)   = splitAt (length x `div` 2) x
```

The function *perfect* d a, used in Section 3.2, generates a perfect tree of depth d whose leaves are labelled with as.

```
perfect :: ∀α . Int → α → Perfect α
perfect 0       a = Zero a
perfect (n + 1) a = Succ (perfect n (a, a))
```

A.2 Text with indentation

The pretty-printing library, used in Section 3, is implemented as follows.

```
data Text = Text String
          | NL
          | Indent Int Text
          | Text :◇ Text
text   = Text
nl     = NL
indent = Indent
(◇)    = (:◇)
```

Each *Text*-generating function is implemented by a corresponding data constructor. The main work is done by the function *render*, which can be seen as an interpreter for *Text*-documents.

```
render′ :: Int → Text → String → String
render′ i (Text s)   x    = s ++ x
render′ i NL         x    = "\n" ++ replicate i ' ' ++ x
render′ i (Indent j d) x  = render′ (i + j) d x
render′ i (d1 :◇ d2) x    = render′ i d1 (render′ i d2 x)
render :: Text → String
render d                  = render′ 0 d ""
```

The functions *append* and *bracketed* are derived combinators:

```
append :: [Text] → Text
append = foldr (◇) (text "")
bracketed :: [Text] → Text
bracketed Nil           = text "[]"
bracketed (Cons d ds)   = align "[ " d
                          ◇ append [nl ◇ align ", " d | d ← ds] ◇ text "]"
```

The function *append* concatenates a list of documents; *bracketed* produces a comma-separated sequence of elements between square brackets.

Finally, we provide a *Show* instance for *Text*, which renders a text as a string (this instance is particularly useful for interactive sessions).

```
instance Show Text where
  showsPrec p x = render′ 0 x
```

A.3 Parsing

The type *ReadS* is Haskell's parser type. The function *alt*, used in Section 5.2, implements the alternation of a list of parsers.

```
alt :: [ReadS α] → ReadS α
alt rs = λs → concatMap (λr → r s) rs
```

Give a parser for elements, *readsList*, also used in Section 5.2, parses a list of elements written as a comma-separated sequence between square brackets.

```
readsList :: ReadS α → ReadS [α]
readsList r = readParen False (λs → [x | ("[", s1) ← lex s, x ← readl s1])
  where readl  s = [(Nil,          s1) | ("]", s1) ← lex    s]
                ++ [(Cons x xs, s2) | (x,   s1) ← r      s,
                                      (xs,  s2) ← readl′ s1]
        readl′ s = [(Nil,          s1) | ("]", s1) ← lex    s]
                ++ [(Cons x xs, s3) | (",", s1) ← lex    s,
                                      (x,   s2) ← r      s1,
                                      (xs,  s3) ← readl′ s2]
```

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, Älvsjö, Sweden, September 2001.
2. Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic Programming: An Introduction. In S. Doaitse Swierstra, Pedro R. Henriques, and Jose N. Oliveira, editors, *3rd International Summer School on Advanced Functional Programming, Braga, Portugal*, volume 1608 of *Lecture Notes in Computer Science*, pages 28–115. Springer-Verlag, Berlin, 1999.
3. Richard Bird 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.
4. W. Braun and M. Rem. A logarithmic implementation of flexible arrays. Memorandum MR83/4, Eindhoven University of Technology, 1983.
5. 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.
6. Dave Clarke and Andres Löh. Generic Haskell, specifically. In Jeremy Gibbons and Johan Jeuring, editors, *Proceedings of the IFIP TC2 Working Conference on Generic Programming, Schloss Dagstuhl*, pages 21–48. Kluwer Academic Publishers, July 2002.
7. Karl Crary and Stephanie Weirich. Flexible type analysis. *ACM SIGPLAN Notices*, 34(9):233–248, September 1999. Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France.
8. Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, MD*, volume (**34**)1 of *ACM SIGPLAN Notices*, pages 301–312. ACM Press, June 1999.
9. Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, St. Petersburg Beach, Florida, United States*, pages 284–294, 1996.
10. Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, March 1996.
11. Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *22nd Symposium on Principles of Programming Languages, POPL '95*, pages 130–141, 1995.
12. Ralf Hinze. Functional Pearl: Perfect trees and bit-reversal permutations. *Journal of Functional Programming*, 10(3):305–317, May 2000.
13. Ralf Hinze. Memo functions, polytypically! In Johan Jeuring, editor, *Proceedings of the 2nd Workshop on Generic Programming, Ponte de Lima, Portugal*, pages 17–32, July 2000. The proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-2000-19.
14. 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.

15. Ralf Hinze. Polytypic values possess polykinded types. *Science of Computer Programming*, 43:129–159, 2002.
16. Ralf Hinze. Fun with phantom types. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, pages 245–262. Palgrave Macmillan, 2003. ISBN 1-4039-0772-2 hardback, ISBN 0-333-99285-7 paperback.
17. Ralf Hinze. Generics for the masses. *J. Functional Programming*, 16(4&5):451–483, July&September 2006.
18. Ralf Hinze and Johan Jeuring. Generic Haskell: Applications. In Roland Backhouse and Jeremy Gibbons, editors, *Generic Programming: Advanced Lectures*, volume 2793 of *Lecture Notes in Computer Science*, pages 57–97. Springer-Verlag, 2003.
19. Ralf Hinze and Johan Jeuring. Generic Haskell: Practice and theory. In Roland Backhouse and Jeremy Gibbons, editors, *Generic Programming: Advanced Lectures*, volume 2793 of *Lecture Notes in Computer Science*, pages 1–56. Springer-Verlag, 2003.
20. Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. *Science of Computer Programming*, 51:117–151, 2004.
21. Ralf Hinze and Andres Löh. "Scrap Your Boilerplate" revolutions. In Tarmo Uustalu, editor, *8th International Conference on Mathematics of Program Construction (MPC '06)*, volume 4014 of *Lecture Notes in Computer Science*, pages 180–208. Springer-Verlag, July 2006.
22. Ralf Hinze, Andres Löh, and Bruno C.d.S. Oliveira. "Scrap Your Boilerplate" reloaded. In Philip Wadler and Masimi Hagiya, editors, *Proceedings of the Eighth International Symposium on Functional and Logic Programming (FLOPS 2006), 24-26 April 2006, Fuji Susono, Japan*, volume 3945 of *Lecture Notes in Computer Science*, pages 13–29. Springer-Verlag, April 2006.
23. Ralf Hinze, Andres Löh, and Bruno C.d.S. Oliveira. "Scrap Your Boilerplate" reloaded. Technical Report IAI-TR-2006-2, Institut für Informatik III, Universität Bonn, January 2006.
24. 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.
25. Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.
26. 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.
27. Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1):1–35, January 1995.
28. Mark P. Jones. Type classes with functional dependencies. In G. Smolka, editor, *Proceedings of the 9th European Symposium on Programming, ESOP 2000, Berlin, Germany*, volume 1782 of *Lecture Notes in Computer Science*, pages 230–244. Springer-Verlag, March 2000.
29. Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In Kathleen Fisher, editor, *Proceedings of the 2004 International Conference on Functional Programming, Snowbird, Utah, September 19–22, 2004*, pages 244–255, September 2004.

30. Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In Benjamin Pierce, editor, *Proceedings of the 2005 International Conference on Functional Programming, Tallinn, Estonia, September 26–28, 2005*, September 2005.
31. Daan Leijen and Erik Meijer. Domain-specific embedded compilers. In *Proceedings of the 2nd Conference on Domain-Specific Languages*, pages 109–122, Berkeley, CA, October 1999. USENIX Association.
32. Andres Löh and Ralf Hinze. Open data types and open functions. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, Venice, Italy*, pages 133–144. ACM Press, July 2006.
33. Andres Löh. *Exploring Generic Haskell*. PhD thesis, Utrecht University, 2004.
34. Andres Löh and Johan Jeuring. The Generic Haskell user's guide, version 1.42 - Coral release. Technical Report UU-CS-2005-004, Universiteit Utrecht, January 2005.
35. Ulf Norell and Patrik Jansson. Polytypic programming in Haskell. In Phil Trinder, Greg Michaelson, and Ricardo Peña, editors, *Implementation of Functional Languages: 15th International Workshop, IFL 2003, Edinburgh, UK, September 8-11, 2003*, pages 168–184, September 2003.
36. Bruno C.d.S. Oliveira and Jeremy Gibbons. TypeCase: A design pattern for type-indexed functions. In Daan Leijen, editor, *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell, Tallinn, Estonia*, pages 98–109, September 2005.
37. Simon Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.
38. Simon Peyton Jones and Ralf Lämmel. Scrap your boilerplate: a practical approach to generic programming. In *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003), New Orleans*, pages 26–37, January 2003.
39. The GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.4.1*, 2005. Available from `http://www.haskell.org/ghc/`.
40. Valery Trifonov, Bratin Saha, and Zhong Shao. Fully reflexive intensional type analysis. In *Proceedings ICFP 2000: International Conference on Functional Programming*, pages 82–93. ACM Press, 2000.
41. Philip Wadler. The expression problem. Note to Java Genericity mailing list, 12 November 1998.
42. 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.
43. Stephanie Weirich. Encoding intensional type analysis. In *European Symposium on Programming*, volume 2028 of *LNCS*, pages 92–106. Springer-Verlag, 2001.
44. Stephanie Weirich. Type-safe cast. *Journal of Functional Programming*, 14(6):681–695, November 2004.