Generic Programming in 3D

Ralf Hinze [a], Andres Löh [b]

[a] *Institut für Informatik III, Universität Bonn*
Römerstraße 164, 53117 Bonn, Germany

[b] *Utrecht University, Department of Information and Computing Sciences*
PO Box 80.089, 3508 TB Utrecht, The Netherlands

Abstract

Support for generic programming consists of three essential ingredients: support for overloaded functions, a run-time type representation, and a generic view on data. Different approaches to datatype-generic programming occupy different points in this design space. In this article, we revisit the "Scrap your boilerplate" approach and identify its location within the three-dimensional design space. The characteristic features of "Scrap your boilerplate" are its two generic views, the 'spine' view for consuming and transforming data, and the 'type-spine' view for producing data. We show how to combine these views with different overloading mechanisms and type representations.

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 [44] it is simply not possible, for instance, to define an equality test that works for all non-function 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

Email addresses: `ralf@informatik.uni-bonn.de` (Ralf Hinze), `andres@cs.uu.nl` (Andres Löh).

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.[1] 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. The approaches differ wildly in syntax, expressiveness and ease of use. However, they all share a common structure. In general, support for generic programming consists of three essential ingredients:

- support for overloading,
- a run-time type representation, and
- a generic view on data.

An *overloaded* function is a function that performs case analysis on types and exhibits type-specific behaviour. As we will see in Section 2, generic functions are special cases of overloaded functions, and almost every generic function exhibits type-specific behaviour: Haskell's pretty-printer, for instance, displays pairs and lists using a special mix-fix notation.

If we write a generic function, does it only range over types (such as the pretty printer), or also over parameterised types (such as a map function that works for any container type)? The *type representation* determines the types we can analyze in overloaded functions. Ideally, a type representation is a faithful mirror of the language's type system. To be able to define such a representation, the type system must be sufficiently expressive. However, the more expressive a type system, the more difficult it is to reflect the full system onto the value level.

Finally, to make a function truly generic, we need a uniform mechanism for accessing the structure of values: a generic pretty printer works for all datatypes including types that the programmer has yet to define. Consequently, the pretty printer has to treat elements of these types in a uniform way: in Haskell, for instance, they are displayed using prefix notation.

[1] Actually, in Haskell 1.0 the compiler would always generate an instance of equality. A deriving form was used to *restrict* the instances generated to those mentioned in the form. To avoid the generation of instances altogether, the programmer had to supply an empty deriving clause.

The three ingredients are orthogonal concepts. Each concept can be realized in different ways and each combination yields a different point in the design space of generic programming. Not all combinations are equally feasible, and sometimes, additional glue is required to make a choice work.

In this article, we take a closer look at the "Scrap your boilerplate" (SYB) approach [45,36,37,29,27] and identify its location in the three-dimensional design-space. The characteristic feature of SYB are its views: For consuming and transforming values, SYB employs the 'spine' view; for producing values, SYB provides the 'type-spine' view. We show how to use different overloading mechanisms and type representations. In particular, we discuss three techniques to achieve overloading: type reflection via a datatype of type representations, Haskell's type classes, and a type-safe cast. We explain how to represent the type system of Haskell 1.0 and explore several ideas of how to cover the more expressive system of Haskell 98 (or one of its manifold extensions).

The rest of the article is organised as follows. In Section 2, we introduce the central ideas of overloading, type representations, and generic views by picking one particular point in the design space: we choose explicit type reflection to achieve support for overloading, we represent the simplest possible type language, and pick the spine view of SYB as the generic view. Using this combination, we present a few simple generic functions. Afterwards, we look at each of the three dimensions in more detail, starting with type representations (Section 3), followed by generic views (Section 4) and mechanisms for overloading (Section 5). Finally, we discuss related work in Section 6 and conclude in Section 7.

Throughout this article, we use Haskell (with GADTs, and for some parts, overlapping instances) as an implementation language. As we explain in Section 2, we make some assumptions about the language that are not supported by current Haskell explanations. However, all of the modifications are so simple that they can be translated into plain Haskell by applying a simple preprocessor. In fact, this paper is generated using lhs2TeX [25], a preprocessor that can generate both a LaTeX version and a Haskell version of all the Haskell code examples from a common source.

This article is a thoroughly revised and extended version of the lecture notes "Generic Programming, Now!" [28], which in turn are based on the papers "'Scrap Your Boilerplate' Reloaded" [29] and "'Scrap Your Boilerplate' Revolutions" [27]. While the lecture notes explore the design space in 2D, this paper additionally covers the third dimension: the mechanism for overloading (Section 5). Furthermore, we introduce and discuss a new generic view (Section 4.3), which combines the 'spine' and the 'type-spine' view.

2 A guided tour

2.1 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.

```
data Tree α = Empty | Node (Tree α) α (Tree α)
                deriving (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 ('⋙ ' is the prompt of the interpreter).

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

Here *tree* :: $[\alpha] \to$ Tree α 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.

```
⋙ tree [0 . . 9]
Node (Node (Node (Node Empty 0 Empty) 1 Empty) 2 (Node (Node Empt
y 3 Empty) 4 Empty)) 5 (Node (Node (Node Empty 6 Empty) 7 Empty) 8 (N
ode Empty 9 Empty))
```

In the sequel, we develop a replacement for *show*, a generic prettier-printer. There are several pretty-printing libraries around; since this article focuses on generic programming techniques rather than pretty-printing we pick a very simple one (see Appendix A.2), which only offers 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 the datatype 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.

Now, our goal is to define a single function that receives the type of the to-be-printed-value as an additional argument and suitably dispatches on this type argument. Unfortunately, Haskell does not 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 $\mathsf{Type} \to \alpha \to \mathsf{Text}$ where Type is a suitable type of type representations. This turns out to be too simple-minded: the parametricity theorem [51] 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 $\mathsf{Type}\ \alpha \to \alpha \to \mathsf{Text}$. The idea is that an element of type $\mathsf{Type}\ \tau$ is a representation of the type τ. Using a *generalised algebraic datatype* (GADT), 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)
```

A few remarks are in order. First, the above datatype definition introduces constructors *Char*, *Int*, *Pair*, *List*, and *Tree* with the given type signatures. The type defined is not an ordinary algebraic datatype, as the result types of the constructors are restricted: *Char*, for example, is not an element of $\mathsf{Type}\ \alpha$, but of $\mathsf{Type\ Char}$. If we analyse a value of a generalised algebraic datatype using a **case**-statement, the type checker can use this additional information.

Second, the datatype Type is a parameterised type. Haskell's type system classifies types using so-called *kinds* (the types of types). All values belong to types of kind $*$. Type functions have kind $\kappa_1 \to \kappa_2$ where κ_1 is the kind of the argument type and κ_2 is the kind of the result type. The grammar of kinds is given by

$$\kappa ::= * \mid \kappa_1 \to \kappa_2$$

Kinds play a significant role in the field of generic programming, as we will see in Section 3.

Third, the datatype definition is marked *open*. In this article, we assume that we have open datatypes and open functions [39] at our disposal. New constructors can be freely added to an open datatype without modifying code that already has been written. For instance, we can add a new constructor *Bool* to represent the type of truth values just by providing the signature

```
Bool :: Type Bool
```

We augment Type by a new constructor every time we define a new datatype in our program.

The semantics of an open datatype is the same as if it 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. The code in this article remains executable in current Haskell implementations that do not support these constructs by applying a preprocessor that collects all parts of open definitions into one place.

Using Type, each type has a unique representation: the type Int is represented by the constructor *Int*, the type (Char, Int) is represented by *Pair Char Int* and so forth. For any given τ in our family of types, Type τ comprises exactly one element (ignoring $\bot$); 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. [2]

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

The definition makes use of Haskell's record syntax. It introduces the colon ':' as an infix data constructor. Additionally, the definition brings the field selectors *val* and *type* into scope. Thus, $4711 : \mathit{Int}$ is an element of Typed Int and (47, "hello") : *Pair Int* (*List Char*) is an element of Typed (Int, [Char]). 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 ◇
```

[2] 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, namely as `t :> x`.

```
                        pretty (x : a) ♢ nl ♢
                        pretty (r : Tree a) ♢ text ")")
```
$pretty_{\mathsf{Char}} :: \mathsf{Char} \to \mathsf{Text}$
$pretty_{\mathsf{Char}}\ c = text\ (show_{\mathsf{Char}}\ c)$
$pretty_{\mathsf{Int}} :: \mathsf{Int} \to \mathsf{Text}$
$pretty_{\mathsf{Int}}\ n = text\ (show_{\mathsf{Int}}\ n)$
$align :: \mathsf{String} \to \mathsf{Text} \to \mathsf{Text}$
$align\ s\ d = indent\ (length\ s)\ (text\ s \diamondsuit d)$

The first line flags *pretty* as an *open function*. The definition of an open function need not be contiguous; the defining equations may be scattered throughout the program. In the case of overlapping patterns, the most specific match takes precedence (best-fit pattern matching). Again, the semantics of an open function is the same as if the function had been defined in one place. Open functions can be translated into ordinary Haskell by collecting and reordering the scattered cases. The details are described in a recent paper [39].

The function *pretty* makes heavy use of type annotations; its type $\mathsf{Typed}\ \alpha \to \mathsf{Text}$ is essentially an uncurried version of $\mathsf{Type}\ \alpha \to \alpha \to \mathsf{Text}$. 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 $\mathsf{Typed\ Char} \to \mathsf{Text}$. Let us consider each equation in turn. The first two equations take care of characters and integers, respectively. Pairs are enclosed in parentheses, the two elements being separated by a line-break 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 pretty-printer produces output in the following style.

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

While the layout nicely emphasizes 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.

$\mathit{pretty}\ (s : \mathit{List}\ \mathit{Char}) = \mathit{text}\ (\mathit{show}_{[\mathsf{Char}]}\ s)$

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

```
>>> pretty ([(47, "hello"), (11, "world")] : List (Pair Int (List Char)))
[ (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 : List Char)      = [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.

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 [17].

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

As an aside, note that Perfect is a so-called *nested data type* [6]. 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*.

```
>>> 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 the element *a*; its definition is given in Appendix A.1.

2.3 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*.

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* particular overloaded function. 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 $Con\ node \diamond (l : Tree\ Int) \diamond (a : Int) \diamond (r : Tree\ Int)$ 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 sequence of type assignments 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)
```

Note that the type variable α does not appear in the result type of '$\diamond$': it is existentially quantified.[3] 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 $\mathsf{Constr}\ \alpha$ comprises an element of type α, namely the original data constructor, plus some additional information about its syntax: its name and its arity.

$\mathbf{data}\ \mathsf{Constr}\ \alpha = \mathit{Descr}\{\mathit{constr} :: \alpha, \mathit{name} :: \mathsf{String}, \mathit{arity} :: \mathsf{Int}\}$

Given a value of type $\mathsf{Spine}\ \alpha$, we can easily recover the original value of type α by undoing the conversion step.

$$
\begin{aligned}
&\mathit{fromSpine} :: \mathsf{Spine}\ \alpha \to \alpha\\
&\mathit{fromSpine}\ (\mathit{Con}\ c) = \mathit{constr}\ c\\
&\mathit{fromSpine}\ (f \diamond x) \quad = (\mathit{fromSpine}\ f)\ (\mathit{val}\ x)
\end{aligned}
$$

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 $\mathsf{Typed}\ \alpha \to \mathsf{Spine}\ \alpha$. 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

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

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

$$
\begin{aligned}
&\mathbf{open}\ \mathit{toSpine} :: \mathsf{Typed}\ \alpha \to \mathsf{Spine}\ \alpha\\
&\mathit{toSpine}\ (\mathit{Empty} : \mathit{Tree}\ a) \qquad = \mathit{Con}\ \mathit{empty}\\
&\mathit{toSpine}\ (\mathit{Node}\ l\ x\ r : \mathit{Tree}\ a) = \mathit{Con}\ \mathit{node} \diamond (l : \mathit{Tree}\ a) \diamond (x : a) \diamond (r : \mathit{Tree}\ a)\\
&\mathit{empty} :: \mathsf{Constr}\ (\mathsf{Tree}\ \alpha)\\
&\mathit{empty} = \mathit{Descr}\{\mathit{constr} = \mathit{Empty}, \mathit{name} = \texttt{"Empty"}, \mathit{arity} = 0\}\\
&\mathit{node} \ \ :: \mathsf{Constr}\ (\mathsf{Tree}\ \alpha \to \alpha \to \mathsf{Tree}\ \alpha \to \mathsf{Tree}\ \alpha)\\
&\mathit{node} \ \ = \mathit{Descr}\{\mathit{constr} = \mathit{Node}, \ \ \mathit{name} = \texttt{"Node"}, \ \ \mathit{arity} = 3\}
\end{aligned}
$$

[3] 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.

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.

$$\begin{aligned}
&\mathit{strings}\ x = \mathit{strings}_{\swarrow}\ (\mathit{toSpine}\ x)\\
&\mathit{strings}_{\swarrow} :: \mathsf{Spine}\ \alpha \to [\mathsf{String}]\\
&\mathit{strings}_{\swarrow}\ (\mathit{Con}\ c) = \mathit{Nil}\\
&\mathit{strings}_{\swarrow}\ (f \diamond x) \quad = \mathit{strings}_{\swarrow}\ f +\!\!\!+\ \mathit{strings}\ x
\end{aligned}$$

The helper function $\mathit{strings}_{\swarrow}$ traverses the left spine calling *strings* for each argument of the spine.

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

$$\begin{aligned}
&\mathit{strings} :: \mathsf{Typed}\ \alpha \to [\mathsf{String}]\\
&\mathit{strings}\ (s : \mathit{List}\ \mathit{Char}) = [s]\\
&\mathit{strings}\ x \qquad\qquad\quad\ \ = \mathit{strings}_{\swarrow}\ (\mathit{toSpine}\ x)
\end{aligned}$$

The revised definition makes clear that *strings* has only one type-specific case, namely the one for *List Char*. 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 parenthesise nullary constructors.

$$\begin{aligned}
&\mathit{pretty}\ x = \mathit{pretty}_{\swarrow}\ (\mathit{toSpine}\ x)\\
&\mathit{pretty}_{\swarrow} :: \mathsf{Spine}\ \alpha \to \mathsf{Text}\\
&\mathit{pretty}_{\swarrow}\ (\mathit{Con}\ c) = \mathit{text}\ (\mathit{name}\ c)\\
&\mathit{pretty}_{\swarrow}\ (f \diamond x) \quad = \mathit{pretty1}_{\swarrow}\ f\ (\mathit{pretty}\ x)\\
&\mathit{pretty1}_{\swarrow} :: \mathsf{Spine}\ \alpha \to \mathsf{Text} \to \mathsf{Text}\\
&\mathit{pretty1}_{\swarrow}\ (\mathit{Con}\ c)\ d = \mathit{align}\ (\texttt{"("} +\!\!\!+\ \mathit{name}\ c +\!\!\!+\ \texttt{" "})\ (d \Diamond \mathit{text}\ \texttt{")"})\\
&\mathit{pretty1}_{\swarrow}\ (f \diamond x) \quad d = \mathit{pretty1}_{\swarrow}\ f\ (\mathit{pretty}\ x \Diamond \mathit{nl} \Diamond d)
\end{aligned}$$

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 $\mathit{XML} :: \mathsf{Type}\ \mathsf{XML}$ 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.

2.4 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.

```
data Dynamic :: * where
  Dyn :: Typed α → Dynamic
```

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 dynamic values.

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

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

```
Dynamic :: Type Dynamic
```

Now, we can also turn the list *misc* itself into a dynamic value: *Dyn* (*misc* : *List Dynamic*).

Dynamic values and generic functions go well together. In a sense, they are dual concepts.[4] We can quite easily extend the generic function *strings* so that it additionally works for dynamic values.

[4] The universal type Dynamic corresponds to the infinite union $\exists\alpha$. Typed α; a generic function of type Typed $\alpha \to \sigma$ corresponds to the infinite intersection $\forall\alpha$. (Typed $\alpha \to \sigma$) which equals ($\exists\alpha$. Typed α) $\to \sigma$ 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 (see also Section 5.3.2).

$strings\ (Dyn\ x : Dynamic) = strings\ x$

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.

$Type\ \ :: \mathsf{Type}\ \alpha \to \mathsf{Type}\ (\mathsf{Type}\ \alpha)$
$Typed :: \mathsf{Type}\ \alpha \to \mathsf{Type}\ (\mathsf{Typed}\ \alpha)$

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 the type constructor $\mathsf{T} :: * \to *$ is the data constructor $T :: \mathsf{Type}\ \alpha \to \mathsf{Type}\ (\mathsf{T}\ \alpha)$.

As a second step, we provide suitable instances of *toSpine* pedantically following the general scheme given in Section 2.3 (*oftype* is the infix operator ':' augmented by additional information).

$$\begin{array}{ll}
toSpine\ (Char : Type\ Char) & = Con\ char \\
toSpine\ (Int : Type\ Int) & = Con\ int \\
toSpine\ (List\ t : Type\ (List\ a)) & = Con\ list \diamond (t : Type\ a) \quad \text{-- } t = a \\
\ldots & \\
toSpine\ ((x : t) : Typed\ a) & = Con\ oftype \diamond (x : t) \diamond (t : Type\ t) \quad \text{-- } t = a
\end{array}$$

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.

$toSpine\ (Dyn\ x : Dynamic) = Con\ dyn \diamond (x : Typed\ (type\ x))$

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 run-time (see also Section 4.1). But the whole purpose of *Dyn* is to pack a value and its type together, and therefore we 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 traverses *both* the static value *and* its type for *Dynamic*, as ':' is treated just like every other data constructor. This may or 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.

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

2.5 Recap

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 reflect types onto the value level. For each type constructor we introduce a corresponding 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 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* functions.

GADTs allow for a very direct type representation. In a less expressive type system we may have to encode types less directly or in a less type-safe manner. However, we shall see in Section 3 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.

We have used the 'spine' view, given by the type Spine and the transformations *toSpine* and *fromSpine*, to represent data in a uniform way. 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 (such as pretty-printers) but not ones that produce data (such as parsers). Again, the reason for this 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 4 shows how to overcome this limitation and introduces several variants of the spine view.

So far, we have used type reflection as mechanism for overloading. Using the datatype Type of type representations, we can reflect types as values and program functions that depend or dispatch on types. We will discuss alternative techniques in Section 5.

3 Type representations

In this section, we explore the first dimension of the design space of generic programming: the type representation. For simplicity, we consider only overloaded functions – the material still applies to generic functions. We return to truly generic functions in Section 4.

3.1 Representation types for types of a fixed kind

3.1.1 Representation type for types of kind $*$

The type Type of Section 2.1 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 an atomic type such as Char or Int, or a compound type such as [Char] or (Int, [Char]). The components of a compound type are possibly type constructors of higher kinds such as [ ] or (,). 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, consider again the declaration of Type (Section 2.1), this time making polymorphic types explicit.

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

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 $\mathsf{T}\ \alpha$, for all types α. In general, T_κ has the signature

$T_\kappa :: \mathsf{Type}_\kappa\ \mathsf{T}_\kappa$

where Type_κ is defined inductively on the structure of kinds

$\mathbf{type}\ \mathsf{Type}_*\quad \alpha = \mathsf{Type}\ \alpha$
$\mathbf{type}\ \mathsf{Type}_{\iota \to \kappa}\ \varphi = \forall \alpha\ .\ \mathsf{Type}_\iota\ \alpha \to \mathsf{Type}_\kappa\ (\varphi\ \alpha)$

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:

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

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.

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

As an aside, the type constructor Fix is Haskell 98, but the data constructor *Fix* is not, because of its rank-2 type. Using *Fix*, the representation of fixed points on the type level, 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 2.3.

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

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

Unfortunately, this type representation has its problems. For instance, we cannot compare two types for equality: the arguments of *Fix* cannot be recursively checked 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 3.2 we introduce an alternative type representation that does not suffer from this problem.

3.1.2 Representation type for types of kind $* \to *$

The generic functions of Section 2 abstract over a type of kind $*$. For instance, *pretty* generalises functions of type

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

to a single generic function of type

$\mathsf{Type}\ \alpha \to \alpha \to \mathsf{Text}$ or equivalently $\mathsf{Typed}\ \alpha \to \mathsf{Text}$

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

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

to a single generic function of type

$\mathsf{Type}'\ \varphi \to \varphi\ \alpha \to \mathsf{Int}$ or equivalently $\mathsf{Typed}'\ \varphi\ \alpha \to \mathsf{Int}$

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 $\mathsf{Type}_{* \to *}$ is not suitable, as we intend to define *size* and other generic functions by case analysis on the type constructor. Again, the elements of $\mathsf{Type}_{* \to *}$ 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 $* \to *$. The type Type' is only inhabited by two constructors since the other datatypes have kinds different from $* \to *$. Additionally, we introduce a primed variant of Typed.

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

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

```
size :: Typed' φ α → Int
size (Nil :' List)        = 0
```

$$\begin{aligned}
&\mathit{size}\ (\mathit{Cons}\ x\ \mathit{xs} :' \mathit{List}) &&= 1 + \mathit{size}\ (\mathit{xs} :' \mathit{List})\\
&\mathit{size}\ (\mathit{Empty} :' \mathit{Tree}) &&= 0\\
&\mathit{size}\ (\mathit{Node}\ l\ x\ r :' \mathit{Tree}) &&= \mathit{size}\ (l :' \mathit{Tree}) + 1 + \mathit{size}\ (r :' \mathit{Tree})
\end{aligned}$$

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* $(x : \mathit{List}\ (\mathit{Tree}\ \mathit{Int}))$. 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 $\varphi\ \tau = [\mathsf{Tree\ Int}]$. The equation has, in fact, not three but four principal solutions: $\varphi = \Lambda\alpha \to \alpha$ and $\tau = [\mathsf{Tree\ Int}]$, $\varphi = \Lambda\alpha \to [\alpha]$ and $\tau = \mathsf{Tree\ Int}$, $\varphi = \Lambda\alpha \to [\mathsf{Tree}\ \alpha]$ and $\tau = \mathsf{Int}$, and $\varphi = \Lambda\alpha \to [\mathsf{Tree\ Int}]$ and τ arbitrary. How can we represent these different container types? They can be easily expressed using functions: $\lambda a \to a$, $\lambda a \to \mathit{List}\ a$, $\lambda a \to \mathit{List}\ (\mathit{Tree}\ a)$, and $\lambda a \to \mathit{List}\ (\mathit{Tree}\ \mathit{Int})$. 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 [18] so that they become members of Type' and to include Id, given by

$$\mathbf{newtype}\ \mathsf{Id}\ \alpha = \mathit{In}_{\mathsf{Id}}\{\mathit{out}_{\mathsf{Id}} :: \alpha\}$$

as a representation of the type variable α:

$$\begin{aligned}
&\mathit{Id} &&:: \mathsf{Type}'\ \mathsf{Id}\\
&\mathit{Char}' &&:: \mathsf{Type}'\ \mathsf{Char}'\\
&\mathit{Int}' &&:: \mathsf{Type}'\ \mathsf{Int}'\\
&\mathit{List}' &&:: \mathsf{Type}'\ \varphi \to \mathsf{Type}'\ (\mathsf{List}'\ \varphi)\\
&\mathit{Tree}' &&:: \mathsf{Type}'\ \varphi \to \mathsf{Type}'\ (\mathsf{Tree}'\ \varphi)
\end{aligned}$$

The type List', defined below, is the lifted variant of $[\,]$: 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, $\mathsf{List}'\ \mathsf{Id}$, $\mathsf{List}'\ (\mathsf{Tree}'\ \mathsf{Id})$ and $\mathsf{List}'\ (\mathsf{Tree}'\ \mathsf{Int}')$. Essentially, we replace the types by their lifted counterparts and the type variable α by Id. Note that the constructors of Type' have types similar to those of Type, only the kinds differ.

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

$$\begin{aligned}
&\mathbf{newtype}\ \mathsf{Char}'\ &&\chi = \mathit{In}_{\mathsf{Char}'}\{\mathit{out}_{\mathsf{Char}'} :: \mathsf{Char}\}\\
&\mathbf{newtype}\ \mathsf{Int}' &&\chi = \mathit{In}_{\mathsf{Int}'}\{\mathit{out}_{\mathsf{Int}'} :: \mathsf{Int}\}\\
&\mathbf{data}\ \mathsf{List}'\ \alpha' &&\chi = \mathit{Nil}' \mid \mathit{Cons}'\ (\alpha'\ \chi)\ (\mathsf{List}'\ \alpha'\ \chi)\\
&\mathbf{data}\ \mathsf{Pair}'\ \alpha'\ \beta'\ &&\chi = \mathit{Pair}'\ (\alpha'\ \chi)\ (\beta'\ \chi)\\
&\mathbf{data}\ \mathsf{Tree}'\ \alpha' &&\chi = \mathit{Empty}' \mid \mathit{Node}'\ (\mathsf{Tree}'\ \alpha'\ \chi)\ (\alpha'\ \chi)\ (\mathsf{Tree}'\ \alpha'\ \chi)
\end{aligned}$$

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

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

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.

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

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, $\mathit{size}\ (x :' \mathit{List}'\ (\mathit{Tree}'\ \mathit{Id}))$ where x is a list of trees of integers, since $\mathsf{List}'\ (\mathsf{Tree}'\ \mathsf{Id})\ \mathsf{Int}$ is different from $[\mathsf{Tree}\ \mathsf{Int}]$. However, the two types are isomorphic: $\tau \cong \tau'\ \mathsf{Id}$ where τ' is the lifted variant of τ [18]. We leave it at that for the moment and return to the problem in Section 4.

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

$$T_\kappa' :: \mathsf{Type}_\kappa'\ \mathsf{T}_\kappa'$$

where Type_κ' is defined

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

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 4.4.1).

3.1.3 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 [18]. 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 $\mathsf{Type}^{\omega} :: \omega \to *$ **where**
  $T^{\omega}_{\kappa} :: \mathsf{Type}^{\omega}_{\kappa}\ \mathsf{T}^{\omega}_{\kappa}$

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

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

The lifted variant $\mathsf{T}^{\omega}_{\kappa}$ 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: the type 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 $Out_1, \ldots, Out_n$.

3.2 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 $\sigma \to \sigma$.

3.2.1 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.

$\mathbf{open\ data}\ \mathsf{Type}_\kappa :: \kappa \to * \ \mathbf{where}$
  $T_\kappa :: \mathsf{Type}_\kappa\ \mathsf{T}_\kappa$

In this scheme $\mathsf{Int} :: *$ is represented by a data constructor of type Type_*; the type constructor $\mathsf{Tree} :: * \to *$ is represented by a data constructor of type $\mathsf{Type}_{*\to*}$ 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 may come as a surprise: we also represent type application syntactically using a family of kind-indexed constructors.

$App_{\iota,\kappa} :: \mathsf{Type}_{\iota\to\kappa}\ \varphi \to \mathsf{Type}_\iota\ \alpha \to \mathsf{Type}_\kappa\ (\varphi\ \alpha)$

The result type dictates that $App_{\iota,\kappa}$ is an element of Type_κ. Theoretically, we need an infinite number of $App_{\iota,\kappa}$ 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 the purposes of this article the following declarations suffice.

$\mathbf{open\ data}\ \mathsf{Type}_* :: * \to * \ \mathbf{where}$
  $Char_* \quad :: \mathsf{Type}_*\ \mathsf{Char}$
  $Int_* \quad :: \mathsf{Type}_*\ \mathsf{Int}$
  $App_{*,*} \quad :: \mathsf{Type}_{*\to*}\ \varphi \to \mathsf{Type}_*\ \alpha \to \mathsf{Type}_*\ (\varphi\ \alpha)$
$\mathbf{open\ data}\ \mathsf{Type}_{*\to*} :: (* \to *) \to * \ \mathbf{where}$
  $List_{*\to*} \quad :: \mathsf{Type}_{*\to*}\ [\,]$
  $Tree_{*\to*} \quad :: \mathsf{Type}_{*\to*}\ \mathsf{Tree}$
  $App_{*,*\to*} \quad :: \mathsf{Type}_{*\to*\to*}\ \varphi \to \mathsf{Type}_*\ \alpha \to \mathsf{Type}_{*\to*}\ (\varphi\ \alpha)$
$\mathbf{open\ data}\ \mathsf{Type}_{*\to*\to*} :: (* \to * \to *) \to * \ \mathbf{where}$
  $Pair_{*\to*\to*} :: \mathsf{Type}_{*\to*\to*}\ (,)$

For example, $\mathsf{Tree\ Int}$ is now represented by $Tree_{*\to*}$ '$App_{*,*}$' Int_*.[5] We have $(Pair_{*\to*\to*}$ '$App_{*,*\to*}$' $Int_*)$ '$App_{*,*}$' $Int_* :: \mathsf{Type}_*\ (\mathsf{Int}, \mathsf{Int})$. Since $App_{*,*}$ is a

[5] We can use a function or a constructor infix if we enclose it in backquotes: $Tree_{*\to*}$ '$App_{*,*}$' Int_* is the same as $App_{*,*}\ Tree_{*\to*}\ Int_*$.

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 the type system of Haskell 98.

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 3.1.2).

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

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 $\alpha \to \mathsf{Int}$ into an additional argument:

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

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.

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

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.

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

where Count_κ is defined

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

The definition looks familiar: it follows the scheme we have already encountered in Section 3.1.1 (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.

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

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 $count_\kappa$:

$$\begin{array}{lll}
\textbf{open}\ count_* :: \mathsf{Type}_*\ \alpha \to \mathsf{Count}_*\ \alpha & & \\
count_*\ (f\ `App_{*,*}`\ a) & = (count_{*\to*}\, f)\ (count_*\ a) & \\
count_*\ t & = const\ 0 & \\
\textbf{open}\ count_{*\to*} :: \mathsf{Type}_{*\to*}\ \alpha \to \mathsf{Count}_{*\to*}\ \alpha & & \\
count_{*\to*}\ List_{*\to*} & c = sum_{[]}\ .\ map_{[]}\ c & \\
count_{*\to*}\ Tree_{*\to*} & c = count_{*\to*}\ List_{*\to*}\ c\ .\ inorder & \\
count_{*\to*}\ (f\ `App_{*,*\to*}`\ a) & c = (count_{*\to*\to*}\, f)\ (count_*\ a)\ c & \\
\textbf{open}\ count_{*\to*\to*} :: \mathsf{Type}_{*\to*\to*}\ \alpha \to \mathsf{Count}_{*\to*\to*}\ \alpha & & \\
count_{*\to*\to*}\ (Pair_{*\to*\to*})\ c_1\ c_2 = \lambda(x_1, x_2) \to c_1\ x_1 + c_2\ x_2 & &
\end{array}$$

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 $\Lambda\alpha \to [\mathsf{Tree}\ \alpha]$ 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*.

```
>>> let ts = [tree [0 .. i] | i ← [0 .. 9]]
>>> size (List_{*→*}) ts
10
>>> count_{*→*} (List_{*→*}) (size (Tree_{*→*})) 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*—the term dictionary refers to the standard implementation of Haskell's type classes. 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:

$map_\kappa :: \mathsf{Type}_\kappa\ \alpha \to \mathsf{Map}_\kappa\ \alpha\ \alpha$

where Map_κ is defined

$\textbf{type}\ \mathsf{Map}_*\quad \alpha_1\ \alpha_2 = \alpha_1 \to \alpha_2$
$\textbf{type}\ \mathsf{Map}_{\iota\to\kappa}\ \varphi_1\ \varphi_2 = \forall \alpha_1\ \alpha_2\ .\ \mathsf{Map}_\iota\ \alpha_1\ \alpha_2 \to \mathsf{Map}_\kappa\ (\varphi_1\ \alpha_1)\ (\varphi_2\ \alpha_2)$

The definition of *map* itself is straightforward:

$\textbf{open}\ map_* :: \mathsf{Type}_*\ \alpha \to \mathsf{Map}_*\ \alpha\ \alpha$
$map_*\ (Int_*) \qquad\quad = id$
$map_*\ (Char_*) \qquad = id$
$map_*\ (App_{*,*}\ f\ a) = (map_{*\to*}\ f)\ (map_*\ a)$
$\textbf{open}\ map_{*\to*} :: \mathsf{Type}_{*\to*}\ \varphi \to \mathsf{Map}_{*\to*}\ \varphi\ \varphi$
$map_{*\to*}\ (List_{*\to*}) \qquad\quad = map_{[\,]}$
$map_{*\to*}\ (Tree_{*\to*}) \qquad\quad = map_{\mathsf{Tree}}$
$map_{*\to*}\ (App_{*,*\to*}\ f\ a) = (map_{*\to*\to*}\ f)\ (map_*\ a)$
$\textbf{open}\ map_{*\to*\to*} :: \mathsf{Type}_{*\to*\to*}\ \varphi \to \mathsf{Map}_{*\to*\to*}\ \varphi\ \varphi$
$map_{*\to*\to*}\ (Pair_{*\to*\to*})\ f\ g\ (a, b) = (f\ a, g\ b)$

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

3.2.2 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

$\textbf{type}\ \mathsf{Pretty}_*\quad \alpha = \alpha \to \mathsf{Text}$
$\textbf{type}\ \mathsf{Pretty}_{\iota\to\kappa}\ \varphi = \forall \alpha\ .\ \mathsf{Type}_\iota\ \alpha \to \mathsf{Pretty}_\kappa\ (\varphi\ \alpha)$

The equations for $pretty_\kappa$ are similar to those of *pretty* of Section 2.1, except for the 'type patterns': the left-hand side $pretty\ (T\ a_1\ \ldots\ a_n)$ becomes $pretty_\kappa\ T_\kappa\ a_1\ \ldots\ a_n$, where κ is the kind of T.

$\textbf{open}\ pretty_* :: \mathsf{Type}_*\ \alpha \to \mathsf{Pretty}_*\ \alpha$
$pretty_*\ (Char_*)\ c \qquad\qquad\quad = pretty_{\mathsf{Char}}\ c$
$pretty_*\ (Int_*)\quad n \qquad\qquad\quad = pretty_{\mathsf{Int}}\ \ n$
$pretty_*\ (f\ `App_{*,*}`\ a)\ x \qquad\ = pretty_{*\to*}\ f\ a\ x$

$$\begin{array}{lll}
\textbf{open}\ \mathit{pretty}_{*\to *} :: \mathsf{Type}_{*\to *}\ \alpha \to \mathsf{Pretty}_{*\to *}\ \alpha & & \\
\mathit{pretty}_{*\to *}\ (\mathit{List}_{*\to *})\ a\ \mathit{xs} & = & \mathit{bracketed}\ [\mathit{pretty}_{*}\ a\ x \mid x \leftarrow \mathit{xs}] \\
\mathit{pretty}_{*\to *}\ (\mathit{Tree}_{*\to *})\ a\ \mathit{Empty} & = & \mathit{text}\ \texttt{"Empty"} \\
\mathit{pretty}_{*\to *}\ (\mathit{Tree}_{*\to *})\ a\ (\mathit{Node}\ l\ x\ r) & & \\
\quad = \mathit{align}\ \texttt{"(Node "}\ (\mathit{pretty}_{*\to *}\ \mathit{Tree}_{*\to *}\ a\ l \quad \diamondsuit\ \mathit{nl}\ \diamondsuit & & \\
\qquad\qquad\qquad\qquad \mathit{pretty}_{*}\ a\ x\ \diamondsuit\ \mathit{nl}\ \diamondsuit & & \\
\qquad\qquad\qquad\qquad \mathit{pretty}_{*\to *}\ \mathit{Tree}_{*\to *}\ a\ r \quad \diamondsuit\ \mathit{text}\ \texttt{")"}) & & \\
\mathit{pretty}_{*\to *}\ (f\ `\mathit{App}_{*,*\to *}`\ a)\ b\ x & = & \mathit{pretty}_{*\to *\to *}\ f\ a\ b\ x
\end{array}$$

$$\begin{array}{lll}
\textbf{open}\ \mathit{pretty}_{*\to *\to *} :: \mathsf{Type}_{*\to *\to *}\ \alpha \to \mathsf{Pretty}_{*\to *\to *}\ \alpha & & \\
\mathit{pretty}_{*\to *\to *}\ (\mathit{Pair}_{*\to *\to *})\ a\ b\ (x, y) & = & \mathit{align}\ \texttt{"("}\ (\mathit{pretty}_{*}\ a\ x)\ \diamondsuit\ \mathit{nl}\ \diamondsuit \\
 & & \mathit{align}\ \texttt{", "}\ (\mathit{pretty}_{*}\ b\ y)\ \diamondsuit\ \mathit{text}\ \texttt{")"}
\end{array}$$

The equations for type application have a particularly simple form:

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

The recursive call takes two type arguments (in dictionary-passing style the second argument was $\mathit{pretty}_{\iota}\ a$, not just a). But again, this case is truly generic: it is the same for all kind-indexed functions (in type-passing style).

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, $\mathit{size} :: \mathsf{Type}_{*\to *}\ \varphi \to (\forall\alpha\ .\ \varphi\ \alpha \to \mathsf{Int})$, which is not possible in type-passing style where *size* has type $\mathsf{Type}_{*\to *}\ \varphi \to (\forall\alpha\ .\ \mathsf{Type}_{*}\ \alpha \to \varphi\ \alpha \to \mathsf{Int})$.

3.3 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 specify the instances of $* \to *$-indexed functions using suitable type abstractions: for our running example we could use representations of $\Lambda\alpha \to [\mathsf{Tree}\ \mathsf{Int}]$, $\Lambda\alpha \to \alpha$, $\Lambda\alpha \to [\alpha]$, or $\Lambda\alpha \to [\mathsf{Tree}\ \alpha]$. Interestingly, there is an alternative. We can represent an anonymous type function by an *open type term*: $\Lambda\alpha{\to}[\mathsf{Tree}\ \alpha]$, for instance, is represented by $\mathit{List}\ (\mathit{Tree}\ a)$ where a is a suitable representation of α.

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.

$$
\begin{aligned}
&\mathit{count} :: \mathsf{Type}\ \alpha \to (\alpha \to \mathsf{Int})\\
&\mathit{count}\ (\mathit{Char}) &&= \mathit{const}\ 0\\
&\mathit{count}\ (\mathit{Int}) &&= \mathit{const}\ 0\\
&\mathit{count}\ (\mathit{Pair}\ a\ b) &&= \lambda(x,y) \to \mathit{count}\ a\ x + \mathit{count}\ b\ y\\
&\mathit{count}\ (\mathit{List}\ a) &&= \mathit{sum}_{[]}\ .\ \mathit{map}_{[]}\ (\mathit{count}\ a)\\
&\mathit{count}\ (\mathit{Tree}\ a) &&= \mathit{sum}_{[]}\ .\ \mathit{map}_{[]}\ (\mathit{count}\ a)\ .\ \mathit{inorder}
\end{aligned}
$$

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 free type variables to Type. The million-dollar question is, of course, what constitutes a suitable representation of a free type variable? Now, if we extend *count* by a case for the free 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 a free type variable is a constructor that embeds a *count* instance, a function of type $\alpha \to \mathsf{Int}$, into the type of type representations.

$$\mathit{Count} :: (\alpha \to \mathsf{Int}) \to \mathsf{Type}\ \alpha$$

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

$$\mathit{count}\ (\mathit{Count}\ c) = c$$

A moment's reflection reveals that this approach is an instance of the 'embedding trick' [13] 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*:

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

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

```
>>> 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. 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 at hand. Fortunately, there is a way out of this dilemma: we parameterise Type by the type of generic functions.

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

A generic function then has type PType Poly α → Poly α for some suitable type Poly. As before, the representation of a free type variable is a constructor of the inverse type, except that now we additionally abstract away from Poly.

```
PVar :: π α → PType π 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*.

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

$\ggg$ *pcount* (a) *ts*
1
$\ggg$ *pcount* $(PList\ a)$ *ts*
10
$\ggg$ *pcount* $(PList\ (PTree\ a))$ *ts*
55
$\ggg$ **let** $a = PVar\ (In_{\mathsf{Count}}\ id)$
$\ggg$ *:type a*
PType Count Int
$\ggg$ *pcount* $(PList\ (PTree\ a))$ *ts*
165

Note that the type of a now limits the applicability of the free 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** $a = PVar\ (In_{\mathsf{Count}}\ (const\ 1))$
psum $f = pcount\ (f\ a)$ **where** $a = PVar\ (In_{\mathsf{Count}}\ id)$

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

$\ggg$ **let** $ts = [tree\ [0\,..\,i] \mid i \leftarrow [0\,..\,9 :: \mathsf{Int}]]$
$\ggg$ *psize* $(\lambda a \to PList\ (PTree\ PInt))$ *ts*
0
$\ggg$ *psize* $(\lambda a \to a)$ *ts*
1
$\ggg$ *psize* $(\lambda a \to PList\ a)$ *ts*
10
$\ggg$ *psize* $(\lambda a \to PList\ (PTree\ a))$ *ts*
55

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* first-order types (standard Hindley-Milner types):

psize $:: (\mathsf{PType\ Count}\ \alpha\ \ \to \mathsf{PType\ Count}\ \beta) \to (\beta \to \mathsf{Int})$
psum $:: (\mathsf{PType\ Count\ Int} \to \mathsf{PType\ Count}\ \beta) \to (\beta \to \mathsf{Int})$

The functions also possess second-order types ($F\omega$ types), which are different from the types above:

psize $:: \forall\varphi\ .\ \mathsf{PType}_{*\to*}\ \mathsf{Count}\ \varphi \to (\forall\alpha\ .\ \varphi\ \alpha\ \ \to \mathsf{Int})$
psum $:: \forall\varphi\ .\ \mathsf{PType}_{*\to*}\ \mathsf{Count}\ \varphi \to (\forall\alpha\ .\ \varphi\ \mathsf{Int} \to \mathsf{Int})$

Using $F\omega$ types, however, the above calls do not type-check, since Haskell employs a kinded first-order unification of types [34].

The other representation types, Type' and Type_κ, can be extended in an analogous manner to support open type terms.

3.4 Summary

In this section, we have presented type representations for several different sets of types. The straightforward representation type Type reflects types of kind $*$, and is therefore suitable to define overloaded or generic functions on datatypes of kind $*$. By lifting the type constructors, we can define several variants of Type, and thereby represent types of an arbitrary, but fixed kind.

The type Type models type terms as first-order algebraic terms. Haskell's type language, however, has a kinded, higher-order term structure. This language can be faithfully represented using a kind-indexed family of representation types. Type application is represented syntactically using kind-indexed constructors $App_{\iota,\kappa}$. Overloaded and generic functions that range over types of different kinds (such as *map*) can then be defined as a kind-indexed family of functions.

We have investigated how to define the 'type application' case of a generic function: in dictionary-passing style, type application is mapped to application of generic functions; in type-passing style, we pass the type representation itself as an argument rather than the recursive call. Both approaches have their merits: while type-passing style allows the definition of mutually recursive generic functions with ease, dictionary-passing style permits the definition of truly polymorphic generic functions.

Haskell does not support type-level lambdas. Nevertheless, we can simulate the effect of type-level lambdas with open type terms, type terms with free type variables. This requires us to parametrise the representation types by the type of generic functions. This extension features a number of advantages: We can represent partial application of type constructors using free type variables obviating the need for a kind-indexed family. Furthermore, by providing instances for the free type variables we can easily modify or customise the behaviour of generic functions.

4 Views

In Section 3 we thoroughly investigated various type representations. The examples in that section are without exception overloaded functions. In this section we discuss techniques to turn these overloaded functions into truly generic ones exploring the second dimension of the design space. 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 2.3 we introduced the spine view, which views data as constructor applications. Of course, this is not the only generic view. PolyP [33], for instance, views data types as fixed points of regular functors; Generic Haskell [23] 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 a *single* element.

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

```
data View :: * → * where
  View :: Type β → (α → β) → (α ← β) → View α
type α ← β = β → α
```

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 α → View α
```

that maps a type to its structural representation. The view function can then be used in the catch-all case of a generic function. Let us rewrite the catch-all case of *strings* (defined in Section 2.3) so that it makes use of a generic view.

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

Using the *fromData* conversion function, $x : t$ is converted to its structural representation $\mathit{fromData}\ x : u$, 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.

$\mathit{strings}\ (x : \mathit{Spine}\ a) = \mathit{strings}_{\checkmark}\ x$

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

$\mathbf{data}\ \mathsf{View}' :: (* \to *) \to *\ \mathbf{where}$
$\quad \mathit{View}' :: \mathsf{Type}'\ \psi \to (\varphi \mathrel{\dot{\to}} \psi) \to (\varphi \mathrel{\dot{\leftarrow}} \psi) \to \mathsf{View}'\ \varphi$
$\mathbf{type}\ \varphi \mathrel{\dot{\to}} \psi = \forall \alpha\ .\ \varphi\ \alpha \to \psi\ \alpha$
$\mathbf{type}\ \varphi \mathrel{\dot{\leftarrow}} \psi = \forall \alpha\ .\ \psi\ \alpha \to \varphi\ \alpha$

The view function is now of type

$\mathit{view}' :: \mathsf{Type}'\ \varphi \to \mathsf{View}'\ \varphi$

and is used as follows:

$\mathit{map}\ f\ m\ x = \mathbf{case}\ \mathit{view}'\ f\ \mathbf{of}$
$\qquad\qquad\qquad \mathit{View}'\ g\ \mathit{fromData}\ \mathit{toData} \to$
$\qquad\qquad\qquad\quad (\mathit{toData} \cdot \mathit{map}\ g\ m \cdot \mathit{fromData})\ x$

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

4.1 Spine view

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

$\mathit{spine}\quad :: \mathsf{Type}\ \alpha \to \mathsf{View}\ \alpha$
$\mathit{spine}\ a = \mathit{View}\ (\mathit{Spine}\ a)\ (\lambda x \to \mathit{toSpine}\ (x : a))\ \mathit{fromSpine}$

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 has to consider only 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 2.2)

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

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 a typed representation of expressions:

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

The relevant equations for *toSpine* are

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

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 6.6.1) currently does not support **deriving** (*Show*) for GADTs. When we turned Dynamic into a representable type (Section 2.4), 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:

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

The equation for *toSpine*

$\mathit{toSpine}\ (\mathit{Apply}\ f\ x : \mathsf{Expr}\ b)$
$\quad = \mathit{Con}\ \mathit{apply} \diamond (f : \mathit{Expr}\ (a \to b)) \diamond (x : \mathit{Expr}\ a)$ -- 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.[6]

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 [16] is only applicable to Haskell 98 types excluding GADTs and existential types (however, we will show in Section 4.3 how to extend a 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 sections we show how to overcome both limitations.

4.2 The type-spine view

A *generic consumer* is a function of type $\mathsf{Type}\ \alpha \to \alpha \to \tau$ ($\cong \mathsf{Typed}\ \alpha \to \tau$), where the type we abstract over occurs in an argument position and possibly in the result type τ. We have seen in Section 2.3 that the generic part of a consumer follows the general pattern below.

open $\mathit{consume} :: \mathsf{Type}\ \alpha \to \alpha \to \tau$
$\ldots$
$\mathit{consume}\ a\ x = \mathit{consume}_{\swarrow}\ (\mathit{toSpine}\ (x : a))$

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

```
consume↙ :: Spine α → τ
consume↙ ... = ...
```

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

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

The helper function $\mathit{produce}_{\swarrow}$ 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 $\mathit{produce}_{\swarrow} :: \forall\alpha\ .\ \tau \to \mathsf{Spine}\ \alpha$, then the composition $\mathit{fromSpine}\ .\ \mathit{produce}_{\swarrow}$ would yield a parametrically polymorphic function of type $\forall\alpha\ .\ \tau \to \alpha$, which is the type of an unsafe cast operation. Furthermore, 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 4.1): a datatype is essentially a sequence of signatures. This motivates the following definitions.

```
type Datatype α = [Signature α]
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 α rather than Typed α. 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*.

Other than the spine view, the type-spine view encodes not only the structure of a single constructor application, but contains information about all the constructors in the list of signatures. 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 α.

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

Here, *int* maps a character to its annotated variant; *nil*, *cons* and *pair* are the annotated versions of *Nil*, *Cons* and '(,)'. As an aside, the second equation produces a rather long list (and so would a case for *Char*); it is 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          = Nil
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 $generate_{\swarrow}$ constructs all terms that conform to a given signature. The right-hand side of the second equation essentially computes the cartesian product of $generate_{\swarrow}$ *s d* and *generate a d*. Here is a short interactive session that illustrates the use of *generate*.

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

As a second example, let us define a generic parser. We implement a parser similar to Haskell's *read* function that is the left-inverse of *pretty*. For the implementation, we use a standard combinator-parsing library [9], see also Appendix A.3.

```
data ReadP :: ∗ → ∗
instance Monad ReadP
token  :: String → ReadP String
pfail  :: ReadP α
(+++)  :: ReadP α → ReadP α → ReadP α
sepBy  :: ReadP α → ReadP β → ReadP [α]
```

Parsers have type $\mathsf{ReadP}\ \alpha$ if they parse a string producing a result of type α. Note that ReadP is an instance of the Monad class. The function *token* reads a specific string and consumes trailing spaces. The parser *pfail* always fails; the operator $(+\!\!+\!\!+)$ represents binary choice. Finally, *sepBy* parses sequences of αs that are separated by βs.

The generic parser *parse* takes an additional Boolean argument that indicates whether the context requires a non-atomic expression to be enclosed in parentheses.

```
open parse :: Type α → Bool → ReadP α
parse (Char)      d     = parse_Char
parse (Int)       d     = parse_Int
parse (List Char) d     = parse_String
parse (List a)    d
   = parseParens False (do token "["
                           xs ← sepBy (parse a False) (token ",")
                           token "]"
                           return xs)
parse (Pair a b)  d
   = parseParens True  (do x ← parse a False
                           token ","
                           y ← parse b False
                           return (x, y))
parse a           d
   = foldr (+++) pfail
        [parseParens (arity′ s > 0 ∧ d) (parse_✓ s) | s ← datatype a]
```

The overall structure is similar to that of *pretty*. The first three equations delegate the work to tailor-made parsers. We assume that each of these tailor-made parsers can also deal with optional pairs of parentheses and trailing spaces (for instance, $parse_{\mathsf{Int}}$ should be able to parse `(␣(4711␣))␣␣`). Lists consist of a comma-separated list of elements (calling *parse* on the element type a) between square brackets. The function *parseParen* b (defined in Appendix A.3) takes care of optional ($b = False$) or mandatory parentheses ($b = True$). Pairs are read using the usual mix-fix notation. 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 context requires parentheses. The parser for α is the choice between all parsers for the individual constructors of α. The auxiliary function $parse_{\checkmark}$ parses a single constructor application.

```
parse_✓ :: Signature α → ReadP α
parse_✓ (Sig c) = do x ← ident
                     if x == name c then return (constr c) else pfail
```

```
parse↙ (s □ a) = do f ← parse↙ s
                    x ← parse a True
                    return (f x)
```

The constructor itself is a single alphanumeric identifier parsed by *ident* (also defined in Appendix A.3). The case for '□' calls $parse_{\swarrow}$ and *parse* recursively, and applies the results to each other.

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

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

The function *parse* is defined by explicit case analysis on the type representation. This is typical of generic functions, but not compulsory: the wrapper function *read*, which simplifies the use of the generic parser, is given by a simple abstraction:

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

The library function *readP_to_S* turns the abstract parser of type ReadP α into a function of type String → [(α, String)] that produces a list of possible results (and, in case the result corresponds to a partial parse, the rest of the input).

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

```
open 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 $produce_{\swarrow}$ takes care of a single constructor.

The type-spine view is complementary to the spine view, but independent of it. The former is used for generic producers, the latter 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:

```
datatype (Perfect a) = [Sig zero □ a, Sig succ □ Perfect (Pair a a)]
```

The scheme can even be extended to generalised algebraic datatypes. However, since Datatype α is a homogeneous list, we have to partition the constructors according to their result types. Consider the expression datatype of Section 4.1. 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 and made the output of *pretty* somewhat less pretty):

```
⋙ let gen a d = putStrLn (render (pretty (generate a d : List a)))
⋙ 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)))]
⋙ 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)))]
⋙ gen (Expr Char) 4
[]
```

The last call shows that there are no character expressions of depth 4 (in fact, there are no character expressions of any depth).

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.

4.3 Spine-like sum-of-products view

We have introduced the spine view for generic consumers and transformers, and the type-spine view for generic producers, but it remains unsatisfactory that there is no single view that can handle both kinds of generic functions.

In this section, we present a view that makes explicit the sum structure (the choice between the different constructors) and the product structure (the combination of arguments for each constructor) of a datatype. It merges elements of both the spine view and the type-spine view to achieve this goal. The resulting spine-like sum-of-products view is the view introduced in the RepLib library by Weirich [54], where it is used together with class-based overloading rather than explicit type reflection (see also Section 6).

We introduce a variant of Datatype, called Constructors:

```
type Constructors α = [Constructor α]
data Constructor :: ∗ → ∗ where
  Constructor :: Product α → (α → β) → (Maybe α ← β) → Constructor β
```

Compared to Signature, the datatype Constructor contains more information about each constructor: it defines embeddings (partial views) between the datatype and types representing the product structure of the constructors. These embeddings can be used both to consume and to produce values of the original type. Because not all values of the type belong to a certain constructor, one direction of the conversion can fail.

The product structure is a list of type representations, encoded as a nested pair where the left component is always an element of the list, and the right component is either the unit type, indicating the end of the list, or another pair.

```
data Product :: ∗ → ∗ where
  PNil  :: Product ()
  PCons :: Type α → Product β → Product (α, β)
```

The counterpart of *datatype* is an open overloaded function *constructors* that computes an element of type Constructors α from a type representation.

```
open constructors :: Type a → Constructors α
```

As an example, here are the defining equations for Booleans and binary trees:

```
constructors (Bool) =
  [Constructor PNil toFalse fromFalse,
   Constructor PNil toTrue fromTrue]
  where toFalse ()          = False
        fromFalse False = Just ()
        fromFalse _         = Nothing
        toTrue ()           = True
        fromTrue True   = Just ()
        fromTrue _          = Nothing
```

```
constructors (Tree a) =
  [Constructor PNil toEmpty fromEmpty,
   Constructor (PCons (Tree a) (PCons a (PCons (Tree a) PNil)))
               toNode fromNode]
  where toEmpty ()                = Empty
        fromEmpty Empty        = Just ()
        fromEmpty _               = Nothing
        toNode (l, (x, (r, ())))  = Node l x r
        fromNode (Node l x r) = Just (l, (x, (r, ())))
        fromNode _                = Nothing
```

The transformation is straightforward: the arguments of the constructors are transformed into nested pairs. For each constructor, there are specific conversion functions, where the back direction additionally tests whether the value in question is actually constructed by the right constructor.

For brevity, we have not included information about the names of constructors or their arity (as in Section 2.3 for values of type Constr). It would be easy to augment Constructor by another argument that adds this information.[7]

The typical shape of a generic function now consists of three layers:

```
open generic :: Type α → ...
...
generic a = ... generic₊ (constructors a) ...
```

[7] Unfortunately, we cannot reuse the original Constr datatype because that contains the constructor itself as a function, and therefore has an incompatible type.

$$
\begin{array}{l}
\mathit{generic}_{+} :: \mathsf{Constructors}\ \alpha \to \ldots \\
\mathit{generic}_{+}\ \mathit{Nil} \qquad\qquad\qquad\qquad\qquad\quad = \ldots \\
\mathit{generic}_{+}\ (\mathit{Cons}\ (\mathsf{Constructor}\ b\ \mathit{to}\ \mathit{from})\ \mathit{cs}) = \ldots \mathit{generic}_{\times}\ b \ldots \mathit{generic}_{+}\ \mathit{cs}
\end{array}
$$

$$
\begin{array}{l}
\mathit{generic}_{\times} :: \mathsf{Product}\ \alpha \to \ldots \\
\mathit{generic}_{\times}\ \mathit{PNil} \qquad\quad = \ldots \\
\mathit{generic}_{\times}\ (\mathit{PCons}\ a\ b) = \ldots \mathit{generic}\ a \ldots \mathit{generic}_{\times}\ b \ldots
\end{array}
$$

The catch-all case of the generic function invokes *constructors* and delegates the traversal of the constructors – the sum structure of the datatype – to $\mathit{generic}_{+}$. In $\mathit{generic}_{+}$ we have the opportunity to perform some action for each constructor and to invoke $\mathit{generic}_{\times}$ to traverse the arguments of that constructor – the product structure of the datatype. Finally, $\mathit{generic}_{\times}$ recursively invokes the original generic function *generic*, for each constructor argument encountered.

As an example of a generic consumer, let us consider generic equality:

$$
\begin{array}{l}
\mathbf{open}\ \mathit{equal} :: \mathsf{Type}\ \alpha \to \alpha \to \alpha \to \mathsf{Bool} \\
\mathit{equal}\ (\mathit{Int}) \quad\ m\ n = m == n \\
\mathit{equal}\ (\mathit{Char})\ c\quad d = c == d \\
\mathit{equal}\ (a) \qquad\ x\quad y = \mathit{equal}_{+}\ (\mathit{constructors}\ a)\ x\ y
\end{array}
$$

$$
\begin{array}{l}
\mathit{equal}_{+} :: \mathsf{Constructors}\ \alpha \to \alpha \to \alpha \to \mathsf{Bool} \\
\mathit{equal}_{+}\ (\mathit{Cons}\ (\mathsf{Constructor}\ b\ \mathit{toCon}\ \mathit{fromCon})\ \mathit{cs})\ x\ y = \\
\quad \mathbf{case}\ (\mathit{fromCon}\ x, \mathit{fromCon}\ y)\ \mathbf{of} \\
\qquad (\mathit{Just}\ x_{\times},\ \mathit{Just}\ y_{\times}) \to \qquad \mathit{equal}_{\times}\ b\ x_{\times}\ y_{\times} \\
\qquad (\mathit{Nothing}, \mathit{Nothing}) \to \qquad \mathit{equal}_{+}\ \mathit{cs}\ x\ y \\
\qquad \_ \qquad\qquad\qquad\quad\ \to \qquad \mathit{False}
\end{array}
$$

$$
\begin{array}{l}
\mathit{equal}_{\times} :: \mathsf{Product}\ \alpha \to \alpha \to \alpha \to \mathsf{Bool} \\
\mathit{equal}_{\times}\ \mathit{PNil} \qquad\quad ()\qquad ()\qquad = \mathit{True} \\
\mathit{equal}_{\times}\ (\mathit{PCons}\ a\ b)\ (x, r)\ (y, s) = \mathit{equal}\ a\ x\ y \land \mathit{equal}_{\times}\ b\ r\ s
\end{array}
$$

This function differs from the examples we have seen before in that it consumes *two* arguments of the ‘generic’ type. Implementing such a function 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 *equal* must either involve a dynamic type-equality check, or the type of equality must be generalised to

$$\mathit{equal} :: \mathsf{Type}\ \alpha \to \alpha \to \mathsf{Type}\ \beta \to \beta \to \mathsf{Bool}$$

While the latter twist works fine for *equal*, it is not without problems for other, similar functions such as generic comparison, as in general we have to relate elements of different types.

Note that a case for the empty list of constructors is not required in *equal*$_+$, because we stop traversing the list as soon as one of the two arguments matches the current constructor, and each of the arguments must match one constructor in the list. On the negative side, since *equal*$_+$ blindly searches the list of constructors for a matching entry, the running time of *equal* is not only determined by the size of its arguments, but also by the size of the datatypes involved.

As an example of a generic producer, let us define *some*, a function that returns some value of a given type. This function may be used, for instance, in a web application to fill an HTML form with default values [47].

```
open some :: Type α → α
some (Int)  = 0
some (Char) = ' '
some a      = some_+ (constructors a)
some_+ :: Constructors α → α
some_+ (Cons (Constructor b toCon fromCon) _) = toCon (some_× b)
some_× :: Product α → α
some_× PNil         = ()
some_× (PCons a b) = (some a, some_× b)
```

Using a technique similar to that in Section 4.2 we can broaden the scope of the spine-like 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 4.1. We have three different result types, Expr Bool, Expr Int and Expr α, and consequently three equations for *constructors*.

```
constructors (Expr Bool) =  [cEq, cIf Bool]
constructors (Expr Int)  =  [cNum, cPlus, cIf Int]
constructors (Expr a)    =  [cIf a]
cNum = Constructor (PCons Int PNil)
                   toNum fromNum
  where toNum (n, ())       = Num n
        fromNum (Num n) = Just (n, ())
        fromNum _           = Nothing
cPlus  = Constructor (PCons (Expr Int) (PCons (Expr Int) PNil))
                     toPlus fromPlus
  where toPlus (x_1, (x_2, ()))     = Plus x_1 x_2
        fromPlus (Plus x_1 x_2) = Just (x_1, (x_2, ()))
        fromPlus _                  = Nothing
cEq    = Constructor (PCons (Expr Int) (PCons (Expr Int) PNil))
                     toEq fromEq
  where toEq (x₁, (x₂, ()))  = Eq x₁ x₂
        fromEq (Eq x₁ x₂) = Just (x₁, (x₂, ()))
        fromEq _          = Nothing
cIf a  = Constructor (PCons (Expr Bool)
                            (PCons (Expr a) (PCons (Expr a) PNil)))
                      toIf fromIf
  where toIf (x₁, (x₂, (x₃, ())))    = If x₁ x₂ x₃
        fromIf (If x₁ x₂ x₃)        = Just (x₁, (x₂, (x₃, ())))
        fromIf _                    = Nothing
```

For the details we refer to the description in Section 4.2.

The spine-like sum-of-products view is very similar to the sum-of-products view of Generic Haskell [41], the main difference being that Constructors's sum structure is less direct: while the product structure is encoded as a nested pair, the sum is not encoded as a nested binary sum, but simply as a list. The gain is that only one type conversion is necessary, and building subsets of the constructors such as required for the encoding of a GADT is less work.

4.4 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.

```
data BalTree α = Empty | Node Int (BalTree α) α (BalTree α)
```

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 that abstract from types of kind $*$, but not acceptable for generic functions that abstract from type constructors of kind $\kappa_1 \to \kappa_2$. For instance, a generic version of *sum* must consider the label of type α = 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:

```
infixr →'
newtype (φ →' ψ) χ = Fun{ app :: φ χ → ψ χ }
```

For technical reasons, '$\rightarrow'$' must be defined by a **newtype** rather than a **type** declaration.[8] As an example, here are variants of *Nil′* and *Cons′*:

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

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

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

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 $\mathsf{Spine}'\ \varphi$; the constructor '$\diamond'$' applies an element of type $\mathsf{Spine}'\ (\varphi \rightarrow' \psi)$ to a $\mathsf{Typed}'\ \varphi$ yielding an element of type $\mathsf{Spine}'\ \psi$.

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

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

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.

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

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

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

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

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

The first line extends the type representation Type' (see Section 3.1.2) by an additional constructor.

Given these prerequisites we can turn *size* (see Section 3.1.2) into a generic function.

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

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 $\mathsf{Spine}'\ \varphi$ (this will change in a moment), so the first equation applies, which in turn delegates the work to the helper function $\mathit{size}_{\swarrow}$.

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

The implementation of $\mathit{size}_{\swarrow}$ 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:

$$\begin{aligned}
&\mathit{map} :: \mathsf{Type}'\ \varphi \to (\alpha \to \beta) \to (\varphi\ \alpha \to \varphi\ \beta)\\
&\mathit{map}\ \mathit{Id} && m = \mathit{In}_{\mathsf{Id}} \mathbin{.} m \mathbin{.} \mathit{out}_{\mathsf{Id}}\\
&\mathit{map}\ (\mathit{Spine}'\ a') && m = \mathit{map}_{\swarrow}\ m\\
&\mathit{map}\ a' && m = \mathbf{case}\ \mathit{spine}'\ a'\ \mathbf{of}\\
& && \qquad \mathit{View}'\ b'\ \mathit{from}\ \mathit{to} \to \mathit{to} \mathbin{.} \mathit{map}\ b'\ m \mathbin{.} \mathit{from}\\
&\mathit{map}_{\swarrow} :: (\alpha \to \beta) \to (\mathsf{Spine}'\ \varphi\ \alpha \to \mathsf{Spine}'\ \varphi\ \beta)\\
&\mathit{map}_{\swarrow}\ m\ (\mathit{Con}'\ c) &&= \mathit{Con}'\ c\\
&\mathit{map}_{\swarrow}\ m\ (f \diamond'\ (x :'\ a')) &&= \mathit{map}_{\swarrow}\ m\ f \diamond'\ (\mathit{map}\ a'\ m\ x :'\ a')
\end{aligned}$$

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

4.4.1 Bridging the gap

We have noted in Section 3.1.2 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 $\tau'\ \mathsf{Id}$ is isomorphic to τ [18]. (This relation only holds for Haskell 98 types, not for GADTs; see also below.) Even more, $\tau'\ \mathsf{Id}$ 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 $fromList\ In_{\mathsf{Id}}$ and $toList\ out_{\mathsf{Id}}$ exhibit the isomorphism between $[\,]$ and $\mathsf{List}'\ \mathsf{Id}$.

$fromList :: (\alpha \rightarrow \alpha'\ \chi) \rightarrow ([\alpha] \rightarrow \mathsf{List}'\ \alpha'\ \chi)$
$fromList\ from\ Nil \qquad\qquad = Nil'$
$fromList\ from\ (Cons\ x\ xs) = Cons'\ (from\ x)\ (fromList\ from\ xs)$
$toList \quad :: (\alpha'\ \chi \rightarrow \alpha) \rightarrow (\mathsf{List}'\ \alpha'\ \chi \rightarrow [\alpha])$
$toList\ to\ Nil' \qquad\qquad = Nil$
$toList\ to\ (Cons'\ x\ xs) = Cons\ (to\ x)\ (toList\ to\ xs)$

Operationally, if the types $\tau'\ \mathsf{Id}$ and τ have the same run-time representation, then $fromList\ In_{\mathsf{Id}}$ and $toList\ out_{\mathsf{Id}}$ are identity functions (the Haskell Report [44] guarantees this for In_{Id} and out_{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.

$spine'\ List = View'\ (List'\ Id)\ (fromList\ In_{\mathsf{Id}})\ (toList\ out_{\mathsf{Id}})$

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

$\ggg$ **let** $ts = [tree\ [0\,..\,i :: \mathsf{Int}] \mid i \leftarrow [0\,..\,9]]$
$\ggg\ size\ (ts :'\ List)$
10
$\ggg\ size\ (fromList\ (fromTree\ In_{\mathsf{Int}'})\ ts :'\ List'\ (Tree'\ Int'))$
0
$\ggg\ size\ (In_{\mathsf{Id}}\ ts :'\ Id)$
1

$\ggg$ $size\ (fromList\ In_{\mathsf{Id}}\ ts :' List'\ Id)$
10
$\ggg$ $size\ (fromList\ (fromTree\ In_{\mathsf{Id}})\ ts :' List'\ (Tree'\ Id))$
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 [34], the calls almost always additionally involve a change on the value level. The type equation $\varphi\ \tau = [\mathsf{Tree\ Int}]$ is solved by setting $\varphi = [\,]$ and $\tau = \mathsf{Tree\ Int}$, 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, $[\mathsf{Tree\ Int}]$ must be rearranged so that the unification with $\varphi\ \tau$ yields the desired choice.

4.4.2 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 2.2), we introduce a lifted variant

$\mathbf{data}\ \mathsf{Perfect}'\ \alpha'\ \chi = Zero'\ (\alpha'\ \chi)\ |\ Succ'\ (\mathsf{Perfect}'\ (\mathsf{Pair}'\ \alpha'\ \alpha')\ \chi)$
$Perfect\ \ :: \mathsf{Type}'\ \mathsf{Perfect}$
$Perfect' :: \mathsf{Type}'\ \varphi \to \mathsf{Type}'\ (\mathsf{Perfect}'\ \varphi)$
$toSpine'\ (Zero'\ x :' Perfect'\ a') = Con'\ zero' \diamond' (x :' a')$
$toSpine'\ (Succ'\ x :' Perfect'\ a') = Con'\ succ' \diamond' (x :' Perfect'\ (Pair'\ a'\ a'))$

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

$spine'\ (Perfect)$
$\quad = View'\ (Perfect'\ Id)\ (fromPerfect\ In_{\mathsf{Id}})\ (toPerfect\ out_{\mathsf{Id}})$
$fromPerfect :: (\alpha \to \alpha'\ \chi) \to (\mathsf{Perfect}\ \alpha \to \mathsf{Perfect}'\ \alpha'\ \chi)$
$fromPerfect\ from\ (Zero\ x) = Zero'\ (from\ x)$
$fromPerfect\ from\ (Succ\ x) = Succ'\ (fromPerfect\ (fromPair\ from\ from)\ x)$
$toPerfect\quad :: (\alpha'\ \chi \to \alpha) \to (\mathsf{Perfect}'\ \alpha'\ \chi \to \mathsf{Perfect}\ \alpha)$
$toPerfect\ to\ (Zero'\ x) = Zero\ (to\ x)$
$toPerfect\ to\ (Succ'\ x) = Succ\ (toPerfect\ (toPair\ to\ to)\ x)$

The following interactive session shows some examples involving perfect trees.

$\ggg$ $size\ (Succ\ (Zero\ (1,2)) :' Perfect)$
2
$\ggg$ $map\ (Perfect)\ (+1)\ (Succ\ (Zero\ (1,2)))$
$Succ\ (Zero\ (2,3))$

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 4.1. 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 $(\alpha \to \beta) \to (\mathsf{Expr}\ \alpha \to \mathsf{Expr}\ \beta)$: How could we possibly turn expressions of type Expr α into expressions of type Expr β? The type Expr β 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* [38,20] or an *indexed type* [55].

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 τ' 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 $* \to *$.

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 $* \to *$.

4.5 Summary

In this section, we have made the concept of a generic view explicit, and we have compared several variants of the spine view.

The original spine view of SYB [45] provides a view on a single element of a datatype. It models the way that values are constructed. The strengths of the spine view are its simplicity and that it covers a large class of datatypes: nested datatypes and GADTs pose no problems; only values of existential types lacking some form of run-time type information cannot be turned into a spine. The weakness of the view is its value-orientation: functions that destruct a value are easy to write, but functions that construct a value require information about the structure of the complete datatype, which is not provided by the spine view. The spine view operates on datatypes of kind $*$.

The type-spine view (also originating from a SYB paper [36]) contains information about the constructors of a datatype and is specifically tailored to provide the functionality that the original spine view lacks: while it is now easy to write generic producers of values, the type-spine view cannot express generic consumers. Again, the view is applicable to nested datatypes. With a bit of effort the view can be extended to GADTs. This view is slightly more involved than the spine view. Both views nicely complement each other; most generic functions on types of kind $*$ can be written using either view.

The spine-like sum-of-products view can be seen as a combination of the two previous views. The information about the constructors of a datatype (the sum structure) is combined with information about individual values (the product structure). As a consequence, using this view both producers and consumers can be defined generically. Other than the original spine view, the spine-like sum-of-products view is not value-oriented – the shape of the type is separated from the generic value. For this reason, the spine-like sum-of-products view can be used to define consumers that combine several arguments of the generic type, such as generic equality. The view is again applicable to many datatypes of kind $*$. Nested types pose no difficulties. GADTs can be handled in the same manner as for the type-spine view. The spine-like sum-of-products view is the view of RepLib [54] and shares most of its properties with the sum-of-products view of Generic Haskell [41] (although GADTs are not yet supported in Generic Haskell). The disadvantage of the sum-of-products views is that they are less direct than the spine and the type-spine views.

Finally, we have discussed how the spine view can be lifted to datatypes of higher kind such as $* \rightarrow *$. This requires quite some effort, but the development is completely mechanical and could be easily automated. We can reuse the view mechanism to exploit the isomorphism between datatypes and their lifted counterparts. The resulting view is almost as general as the original, but it no longer works for GADTs.

5 Overloading

Most generic functions exhibit type-specific behaviour requiring support for overloading. In this section, we compare three approaches to overloading exploring the third and last dimension of the design space of generic programming.

We have already seen that GADTs enable us to reflect types. We summarise the pros and cons of this approach in Section 5.1. Besides, we take a look at the classic construct that Haskell offers to achieve overloading: *type classes* (Section 5.2). Finally, we discuss the use of a *type-safe cast* to perform a limited amount of run-time analysis on types (Section 5.3).

Some approaches to generic programming are based on preprocessing and circumvent the need for overloading by *specializing* generic functions for the cases that are used in a program. We briefly consider these approaches in our discussion of related work (Section 6).

In the following, we consider and evaluate the different approaches in turn. Like in the previous sections, we move through one dimension of our three-dimensional design space while keeping the two other dimensions fixed. Here, we represent only closed type terms of kind $*$ and we confine ourselves to overloaded functions (no generic view).

5.1 Type reflection

Reflecting types using the GADT Type is the approach to overloading that we have used so far. Each type has a representation as a Haskell value. Here is the definition of Type again:

```
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)
```

The advantage of the GADT encoding is its simplicity and its directness: type representations have structure; they can be combined by constructor application and analyzed by pattern matching. In particular, pattern matching is a very convenient way to write overloaded functions, not the least because Haskell's pattern matching language is very expressive: it allows, for instance, nested and overlapping patterns. At the same time, the encoding is type-safe, because the type parameter of Type records the type that is being reflected.

There are a number of disadvantages, too. We require generalised algebraic datatypes, a relatively recent addition to the Haskell type system, that is not yet supported by many implementations. Also, the explicitness of the reflection mechanisms implies that the programmer must manually provide the correct type arguments. All overloaded and generic functions take a type argument, even if the reflected type is uniquely determined by the context. Perhaps the most significant disadvantage of this reflection mechanism is that the selection of the correct case is shifted to run-time: this implies that we only learn at run-time if a type case is missing (reported as a pattern match failure). For Haskell programmers, who are used to the fact that type-level computations are performed at compile time, this may come as an unpleasant surprise.

If the set of datatypes we deal with is known in advance, GADTs are very suitable for expressing generic functions. However, the introduction of new datatypes requires the modification of Type and a few other existing definitions. Throughout this article, we therefore assume that the programming language supports open datatypes and functions.

5.2 Type classes

Type classes have the potential to alleviate some of the disadvantages of the GADT approach discussed in the previous section. Instances for type classes are inferred statically and implicitly by the compiler, and a missing type case, that is, a missing instance is reported as a compile-time error. Furthermore, type classes can be easily extended with new instances obviating the need for open datatypes.

Type classes are a powerful and versatile construct. Therefore, it is perhaps not surprising that there is more than one way in which type classes can be employed in this context. We consider two variations. First, we show how to combine type classes with the GADT approach to gain more convenience. Thereafter, we analyse the classic approach of defining overloaded functions via type classes.

5.2.1 A class of representable types

A very simple use of type classes in addition to GADT-based reflection is to derive values of type Type automatically:

```
class Rep a where
  rep :: Type a
```

Instances of this class are trivial to define:

```
instance Rep Int where
  rep = Int
instance Rep Char where
  rep = Char
instance (Rep α) ⇒ Rep [α] where
  rep = List rep
instance (Rep α, Rep β) ⇒ Rep (α, β) where
  rep = Pair rep rep
instance (Rep α) ⇒ Rep (Tree α) where
  rep = Tree rep
```

With the help of

```
typed :: (Rep α) ⇒ α → Typed α
typed a = a : rep
```

we can now define a class-based version of a generic function such as *pretty*:

```
cpretty :: (Rep α) ⇒ α → Text
cpretty x = pretty (typed x)
```

The type argument of *cpretty* is now implicit: at the call site, the compiler automatically inserts the correct class instance. However, the other problems of reflected types still persist: the type Type is still present behind the scenes, so we still need GADT support and open datatypes. Furthermore, overloaded functions are still defined using pattern matching on Type, and pattern match failures are still reported as run-time errors.

5.2.2 One class per overloaded function

Haskell's classic approach to overloading is to introduce one type class per overloaded function or per group of related overloaded functions.

As an example, for *strings* from Section 2, we introduce the following class declaration:

```
class Strings a where
  strings :: a → [String]
```

Each type case becomes one instance declaration. Cases for parameterised types correspond to instances with preconditions:

```
instance Strings Int where
  strings i   = Nil
instance Strings Char where
  strings c   = Nil
instance (Strings α) ⇒ Strings [α] where
  strings xs = concat [strings x | x ← xs]
```

An important case for the *strings* function is the one for strings, that is, for the type [Char]. What used to be a nested pattern now turns into an instance that overlaps with the instance for lists:

```
instance Strings [Char] where
  strings s   = [s]
```

Actually, this declaration requires 'overlapping instances', an extension to the class system of Haskell 98. At the call site, the most specific instance that applies is selected by the compiler.

If an overloaded function is declared using type classes, the type arguments are implicit and automatically inferred by the compiler. Furthermore, missing instances are detected statically and an error is reported at the call site. Furthermore, the overloaded function is extensible, because new instances can be added at any time, without touching existing code.

Special care must be taken if the class-based approach to reflection is to be used together with a view. To understand why, recall the definition of views from Section 4:

```
data View :: ∗ → ∗ where
  View :: Type β → (α → β) → (α ← β) → View α
```

A view for type α consists of three components: a structure type β, plus conversion functions to and fro. Since we want to use type classes rather than the Type datatype for type reflection, we have to replace Type by a suitable class constraint.

```
data View :: ∗ → ∗ where
  View :: (C β) ⇒ (α → β) → (α ← β) → View α
```

The constraint is required, because in the generic case, we want to invoke the view and call the function recursively on the structure type. But, what is a suitable choice for C?

Assuming the spine view for the moment and a class HasSpine for the now also class-based function *spine* (cf. Section 4.1), we obtain the following generic case for *strings*:

```
instance (HasSpine α) ⇒ Strings α where
  strings x = case spine of
                View from to → strings (from x)
```

For the above definition to be type-correct, C β must imply Strings β (we call *strings* on the result of *from*). In other words, C must be Strings, or Strings must be a superclass of C. But the same is true for *every* generic function. Whenever we want to define a generic case for a function in a class D, we require that the class D is a superclass of C. We cannot simply let C be a specific class Generic, because Haskell requires us to declare all its superclasses the moment we define Generic. As a consequence, the number of generic functions we can write is fixed, which means that we cannot add new generic functions at a later stage.

The same problem shows up when we adapt the spine view to the class-based approach. Recall that the Spine datatype in its original form contains a type representation:

```
data Spine :: ∗ → ∗ where
  Con :: Constr α → Spine α
  (◇)   :: Spine (α → β) → Typed α → Spine β
```

The reason for including a type representation of α is that we want to call generic functions on the value of that type, but we cannot say in advance which generic function that might be. The type Spine therefore becomes

```
data Spine :: ∗ → ∗ where
  Con :: Constr α → Spine α
  (◇)   :: (C α) ⇒ Spine (α → β) → α → Spine β
```

and as before we do not know what to use as a suitable class C.

There are at least two solutions to this problem: we can detach the superclass relation from the class declaration, or we can abstract away from the type class C. Neither solution is directly expressible in Haskell, each requires an extension to the class system. We discuss each solution in turn.

Flexible superclasses Sulzmann and Wang [49] propose a language extension that detaches the superclass relation from the declaration of classes. In their language extension, the superclass hierarchy can be extended at any time using rules of the form

```
rule Generic β ⟹ Strings β
```

which encodes exactly the statement from above that Generic β (or C β) implies Strings β. In the system of Sulzmann and Wang, the superclass relation may even be cyclic. Interestingly, their proposed translation scheme internally makes use of reflected types.

A consequence of this approach is that a datatype can only be an instance of Generic if it is an instance of *all* generic classes. Unfortunately, this negates one of the advantages of the class-based approach, namely that the domain of a generic function can be defined precisely and individually.

Class abstraction Lämmel and Peyton Jones [37] describe a different solution to the above problem: rather than looking for a single class like Generic to take the place of C, they abstract away from C. Haskell does not support abstraction from type classes, but let us assume for a moment that it does:

```
data AView :: (∗ → ctx) → ∗ → ∗ where
  AView :: (γ β) ⇒ (α → β) → (α ← β) → AView γ α
```

The type AView receives an additional context parameter γ of kind $* \to$ ctx, which represents a class that has one parameter of kind $*$. This parameter propagates to a number of other definitions. For instance, the Spine class becomes ASpine:

```
class ASpine γ α where
  aspine :: AView γ α
```

and the generic case of *strings* requires a modification, as well:

```
instance (ASpine Strings α) ⇒ Strings α where
  strings (x :: α) = case aspine :: AView Strings α of
                       AView from to → strings (from x)
```

At this point we make use of the abstraction: we instantiate γ to Strings so that we can recursively call *strings* after applying the view. For other generic classes, the context argument is instantiated differently. Note that mutually recursive (groups of) generic functions have to be merged into one class, because we abstract only from a single class γ.

Encoding class abstraction in Haskell There is a well-known trick [32,37] to encode class abstraction in Haskell: classes are reflected as datatypes. As an example, for the class Strings we define:

```
data StringsD α = StringsD {stringsD :: α → Text}
```

All class methods (in this case, we only have the *stringsD* method) are stored as components in the so-called *dictionary type* StringsD. The kind ctx is simulated by kind $*$, and a class constraint of the form $(\gamma\ \alpha)$ becomes $(\mathsf{Sat}\ \gamma\ \alpha)$, where Sat is a class that just contains the dictionary (this trick requires multiple-parameter type classes):

$$
\begin{array}{l}
\textbf{class}\ \mathsf{Sat}\ \gamma\ \alpha\ \textbf{where} \\
\quad \mathit{dict} :: \gamma\ \alpha \\
\textbf{data}\ \mathsf{AView} :: (* \to *) \to * \to *\ \textbf{where} \\
\quad \mathit{AView} :: (\mathsf{Sat}\ \gamma\ \beta) \Rightarrow (\alpha \to \beta) \to (\alpha \leftarrow \beta) \to \mathsf{AView}\ \gamma\ \alpha
\end{array}
$$

In the recursive calls, we do not access the class method directly, but instead we extract it from the dictionary:

$$
\begin{array}{l}
\textbf{instance}\ (\mathsf{ASpine}\ \mathsf{StringsD}\ \alpha) \Rightarrow \mathsf{Strings}\ \alpha\ \textbf{where} \\
\quad \mathit{strings}\ (x :: \alpha) = \textbf{case}\ \mathit{aspine} :: \mathsf{AView}\ \mathsf{StringsD}\ \alpha\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad \mathit{AView}\ \mathit{from}\ \mathit{to} \to \mathit{stringsD}\ \mathit{dict}\ (\mathit{from}\ x)
\end{array}
$$

To summarise, type classes are an excellent method of type reflection for overloaded functions. For generic functions, the class system requires a number of extensions: we need overlapping instances in order to express type-specific cases corresponding to nested patterns[9] and we need extensible superclasses. The simulation of class abstraction is quite subtle and requires a deep technical understanding of the type class mechanism. On the positive side, type arguments remain implicit and are inferred automatically. Furthermore, overloaded functions are extensible. Since there are no values that reflect types, there is no need for an open datatype. Missing cases of overloaded or generic functions are reported during compilation.

5.3 Type-safe cast

A third possibility to perform analysis on types is via a built-in type-safe cast operation. In the following, we will look at three variants of a type-safe cast.

5.3.1 Polymorphic cast

Let us assume that we have a *cast operator* of type

$\mathit{cast} :: \alpha \to \mathsf{Maybe}\ \beta$

[9] The generic case does not necessarily require overlapping patterns if we make use of default methods.

The function tests at run-time whether an element of α is in fact an element of β, and it returns *Nothing* if the test fails. With *cast* as a language built-in, we can write functions of polymorphic types that exhibit type-specific behaviour.

For instance, here is a variant of *strings* from Section 2:

```
strings :: α → [String]
strings x =
  case cast x of
    Just (c :: Char) → Nil
    Nothing          →
      case cast x of
        Just (i :: Int) → Nil
        Nothing         →
          case cast x of
            Just (s :: [Char]) → [s]
            Nothing            →
              case cast x of
                Just xs → concat [strings x | x ← xs]
```

A few remarks are in order. The repeated calls to *cast* are necessary because each call is at a different result type. In the first three cases, we annotate the pattern with the type (recall that $e :: \tau$ is a Haskell type annotation, not a pattern of type Typed) because the type cannot be inferred from the context. It is unusual that the presence of type annotations can influence the result of a function call (especially if there are no type classes involved), which indicates that *cast* would indeed be a very powerful function to add to the language. In the fourth case, we assume that we can cast the value x to a list type without specifying the element type explicitly. We then call *strings* recursively on the elements. Note that the function *strings* is incomplete, so if none of the four cases match, an error is reported at run-time.

To avoid the nested case structure, we can define combinators that extend existing functions by new type-specific behaviour. For instance, the original SYB paper [45] introduces an extension operator *extQ* that adds a new type-specific case to a function of type $\alpha \to \gamma$:

```
extQ :: (α → γ) → (β → γ) → (α → γ)
extQ t s x = case cast x of
               Just y   → s y
               Nothing → t x
```

Using *extQ*, we can rewrite *strings* in a slightly more appealing way:

```
strings = error "strings is undefined for this type"
          `extQ` (λxs            → concat [strings x | x ← xs])
          `extQ` (λ(s :: [Char]) → [s])
          `extQ` (λ(i :: Int)    → Nil)
          `extQ` (λ(c :: Char)   → Nil)
```

For defining overloaded functions of a more complex type (generic transformers, for instance), it is necessary to generalize the cast operator to

$gcast :: \gamma\ \alpha \to \mathsf{Maybe}\ (\gamma\ \beta)$

that can convert between α and β in an arbitrary context γ. The function *gcast* is general enough to define *cast*, by instantiating γ to the identity type.

5.3.2 Cast on dynamic values

Using polymorphic types for functions that are not parametrically polymorphic is suspicious. An alternative is to restrict the cast operation to values of type Dynamic, where Dynamic is a built-in, abstract type. Assuming a case-like *cast* construct that allows pattern-matching on types, we can define *strings* in a style similar to the GADT approach:

```
strings :: Dynamic → [String]
strings x =
  cast x of
    (c  :: Char)   → Nil
    (i  :: Int)    → Nil
    (s  :: [Char]) → [s]
    (xs :: [β])    → concat [strings (dynamic (x :: β)) | x ← xs]
```

The functional programming language Clean [46] provides such a built-in interface to dynamic values.

A dedicated pattern-matching construct has the advantage that type patterns are explicit, and that it provides a more concise syntax. As with the *cast* operator, the passing of a type argument to the overloaded function is not required. However, we now have to explicitly convert between values of concrete types and values of type Dynamic.

5.3.3 Class-based cast

In Haskell, we can use the class system to assign *cast* (and *gcast*) more restricted types:

```
cast  :: (Typeable α, Typeable β) ⇒ α → Maybe β
gcast :: (Typeable α, Typeable β) ⇒ γ α → Maybe (γ β)
```

Apart from the types, the code of the generic *strings* function is identical to the one given in Section 5.3.1, requiring a nested **case** or the introduction of an extension operator such as *extQ*.

However, it is now problematic to pattern match on a partially polymorphic case: Recall that one of the four cases in our *strings* example matches lists of an arbitrary element type. For this case, *cast* introduces a constraint Typeable β for the desired result type, and Typeable $[\beta]$ cannot be resolved without knowing β. We can fix this problem by introducing another cast operation, specific for types of kind $* \rightarrow *$:

$$gcast_{*\rightarrow*} :: (\mathsf{Typeable}_{*\rightarrow*}\ \varphi, \mathsf{Typeable}_{*\rightarrow*}\ \psi) \Rightarrow \gamma\ (\varphi\ \alpha) \rightarrow \mathsf{Maybe}\ (\gamma\ (\psi\ \alpha))$$

Likewise, in order to define a case for pairs, we need a cast operator $gcast_{*\rightarrow*\rightarrow*}$. In fact, we need cast operators and variants of Typeable for all kinds that occur in the cases of overloaded functions. Even having all those special purpose cast-operators in place, it is still somewhat tricky to put them to use [36].

We conclude that using a type-safe cast is less convenient than explicit type reflection for the definition of overloaded functions. If we add syntactic sugar for pattern matching as in Section 5.3.2, some of the convenience can be restored. Still, compared to the class-based approach, we lose all the static guarantees: a missing type case will be reported as a run-time error.

6 Related Work

There is a wealth of material on the subject of generic programming. Several tutorials [4,23,22] provide an excellent overview of the field.

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

- support for overloading (Section 5),
- a type representation (Section 3), and
- a generic view on data (Section 4).

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 three 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

view(s)	representation of overloaded functions			
	type reflection	type classes	type-safe cast	specialisation
none	ITA [15,12,11,50,52]	–	–	–
fixed point	–	PolyP [42,43]	–	PolyP [33]
spine	Reloaded [29], Revolutions [27]	SYB [37], Reloaded [30]	SYB [45,36]	–
sum-of-products	LIGD [8,20]	DTC [31], GC [1], GM [21]	–	GH [19,23,40,41]
spine-like sum-of-products	this article	RepLib [54]	–	–

Fig. 1. Generic programming: the design space.

- *type reflection*: This is the approach we have used in the main bulk of this article. Its origins can be traced back to the work on intensional type analysis (ITA) [15,12,11,50,52]. ITA is intensively used in typed intermediate languages, 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 [8] present a library for generics and dynamics (LIGD) that uses an *encoding* of type representations in Haskell 98 augmented by existential types.

  In dependently-typed programming languages, a universe construction can be used for type reflection [5]. In style, this approach is very similar to our use of GADTs. The choice of universe determines which types can be reflected. Representing Haskell's type system within a dependently-typed language is possible without problems [2]. Universes can also represent the inductive families of modern dependently-typed languages. Using such universes for generic programming is the topic of ongoing research [3].
- *type classes* [14]: 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 (GC) [1]. 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 [21].

The implementation builds upon a class-based encoding of the type Type of type representations.

- *type-safe cast* [53]: 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 [45] is based on casts.
- *specialisation* [18]: 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 [33] or *Generic Haskell* [41] 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 third 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 [33] 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 [42] embeds PolyP programs into Haskell augmented by multiple parameter type classes with functional dependencies [35]. Oliveira and Gibbons [43] present a lightweight variant of PolyP that works within Haskell 98.
- *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 [45,36,37]. The original approach is *combinator-based*: the user writes generic functions by combining a few generic primitives. The first paper [45] 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 catamor-

phism of the Spine datatype [29]: *gfoldl* equals the catamorphism composed with *toSpine*. The second paper [36] 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 the type Signature, introduced in Section 4.2.

- *sum-of-products view*: Generic Haskell [23,40,41] (GH) builds upon this view. In its original form it is applicable to all datatypes definable in Haskell 98. Generic Haskell is a full-fledged implementation of generics based on ideas by Hinze [19,24] that features generic functions, generic types and various extensions such as default cases and constructor cases [10]. Generic Haskell supports the definition of functions that work for all types of all kinds, such as, for example, a generalised mapping function.
- *spine-like sum-of-products view:* This is the view that the RepLib library [54] uses. It can express the same functions as GH's sum-of-products view, but in style, it is more closely related to the SYB views. In particular, the sum structure is represented somewhat indirectly using a list. We have seen in Section 4.3 that this view can be generalised to GADTs. Actually, the same technique can also be applied to generalize the GH view, but the resulting code is more verbose.

7 Conclusion

The essence of the SYB approach to generic programming are its two views:

- The 'spine' view faithfully encodes the structure of a value as an application of a constructor to its arguments. Using the view, we can implement generic consumers and transformers.
- The 'type-spine' view allows us to implement generic producers in the same elegant manner as generic consumers that build upon the spine view. While the spine view can be seen as encoding the product structure of a single constructor, the type-spine view offers access to the sum structure of the datatype.

Using one of the different spine views one can program almost all of the standard examples of generic functions.

The spine views are attractive for at least two reasons: they are easy to use and they are widely applicable. For instance, for type indices of kind $*$ the programmer only has to consider two cases. The spine view and the type-spine view cover almost all datatypes *including* generalised algebraic datatypes, but excluding existential types. We have shown that both views can be combined into the 'spine-like sum-of-products' view.

While the views are characteristic for the SYB approach, they are not tied to it. The design space of generic programming is actually three-dimensional: a generic programming approach is classified not only by its view, but also by its approach to overloading, and by the datatypes it can represent. Identifying the three dimensions allows us to better categorize and relate different approaches.

Functions that abstract over type constructors can be handled using the technique of *lifting*. Originally, such functions were thought to be out of reach for the SYB approach. Lifting also requires the adaption of the view and leads to the 'lifted spine view'. Unfortunately, the lifted spine view is, for fundamental reasons, more restricted than the other views: generic functions that abstract over type constructors can be instantiated to arbitrary *container types* but not to *phantom types* (GADTs).

Besides type indices of higher kind, we have shown how to represent open type terms. This allows us to flexibly specify the behaviour of generic functions through free type variables, thereby providing a powerful way to parameterize such functions.

Of the overloading mechanisms we discussed, it is easy to see that the use of a type-safe cast is the least useful. While we advocate the use of reflection, many other approaches use actually type classes. The main reason for this choice is that classes are better supported by current implementations. GADT-support is still new, and for overloaded functions to be extensible, we require open datatypes and open functions [39], which are currently not implemented. Nonetheless, we believe that explicit type arguments make the structure of generic definitions more obvious. Furthermore, we expect that it is easier to prove algebraic properties of generic functions in this setting. We believe that the work of Reig [48] could be recast using our approach, leading to shorter and more concise proofs.

Acknowledgements

We would like to thank Roland Backhouse, Jeremy Gibbons, Stefan Holdermans and the anonymous referees of MPC and this special issue for many suggestions regarding language, style and presentation. Andres Löh would like to thank Tarmo Uustalu for pointing him to the question of how to write generic functions that work on GADTs and for several enlightening discussions on the topic. This work has been supported by the DFG grant "Eine generische funktionale Programmiersprache" (HI 669/7-2).

A Library

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

A.1 Binary trees

The function *inorder*, used in Section 2.1, 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 2.1, turns a list of elements into a balanced binary tree, a so-called *Braun tree* [7].

```
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 2.2, generates a perfect tree of depth d whose leaves are labelled with the element a.

```
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 2, 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 library ReadP (Text.ParserCombinators.ReadP), which is part of Haskell's Hierachical Libraries, comes with a number of predefined combinators:

```
string      :: String → ReadP String
satisfy     :: (Char → Bool) → ReadP Char
munch       :: (Char → Bool) → ReadP String
skipSpaces  :: ReadP ()
readP_to_S  :: ReadP α → (String → [(α, String)])
```

The parser *string* tries to parse the string provided as an argument. The parser *satisfy* parses a single character, but only if it satisfies the given property. Likewise, *munch* parses as many characters as possible that satisfy the given property. The parser *skipSpaces* consumes as much whitespace as possible. Finally, the conversion function *readP_to_S* runs a parser and transforms it from the abstract type ReadP into a concrete function.

The following derived functions are used in Section 4.2 to define the generic parser.

The function *parseParens* parses what its second argument parses, but surrounded with balanced pairs of parentheses. If the first argument is *True*, at least one pair of parentheses is mandatory.

```
parseParens :: Bool → ReadP α → ReadP α
parseParens True  p = between (token "(") (token ")")
                              (parseParens False p)
parseParens False p = p +++ parseParens True p
```

The function *token* reads a specific string and consumes trailing spaces.

```
token :: String → ReadP String
token x = do string x
             skipSpaces
             return x
```

Finally, the function *ident* reads an identifier and discards trailing whitespace. The first character must be an underscore or a letter, the remaining characters may also be numbers.

```
ident :: ReadP String
ident = do c  ← satisfy (λx → x == '_' ∨ isAlpha x)
           cs ← munch (λx → x == '_' ∨ isAlphaNum x)
           skipSpaces
           return (c:cs)
```

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 2001*, volume 2312 of *Lecture Notes in Computer Science*, pages 257–278, Älvsjö, Sweden, September 2001. Springer-Verlag.

[2] Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In Jeremy Gibbons and Jeuring Jeuring, editors, *Pre-Proceedings of IFIP TC2 Working Conf. on Generic Programming, WCGP'02, Dagstuhl, 11–12 July 2002*, 2002. (Final Proceedings to be published by Kluwer Acad. Publ.).

[3] Thorsten Altenkirch, Conor McBride, and Peter Morris. Generic programming ith dependent types. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Datatype-Generic Programming 2006*, volume 4719 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007. To appear.

[4] 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.

[5] Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*, 10(4):265–289, 2003.

[6] 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.

[7] W. Braun and M. Rem. A logarithmic implementation of flexible arrays. Memorandum MR83/4, Eindhoven University of Technology, 1983.

[8] 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.

[9] Koen Claessen. Parallel parsing processes. *Journal of Functional Programming*, 14(6):741–757, 2004.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] Ralf Hinze. Functional Pearl: Perfect trees and bit-reversal permutations. *Journal of Functional Programming*, 10(3):305–317, May 2000.

[18] 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.

[19] Ralf Hinze. Polytypic values possess polykinded types. *Science of Computer Programming*, 43:129–159, 2002.

[20] 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.

[21] Ralf Hinze. Generics for the masses. *J. Functional Programming*, 16(4&5):451–483, July&September 2006.

[22] 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.

[23] 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.

[24] Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. *Science of Computer Programming*, 51:117–151, 2004.

[25] Ralf Hinze and Andres Löh. lhs2. `http://people.cs.uu.nl/andres/lhs2tex/`.

[26] Ralf Hinze and Andres Löh. Open data types and open functions. Technical Report IAI-TR-2006-3, Institut für Informatik III, Universität Bonn, February 2006.

[27] 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.

[28] Ralf Hinze and Andres Löh. Generic programming, now! In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Datatype-Generic Programming 2006*, volume 4719 of *Lecture Notes in Computer Science*, pages 150–208. Springer-Verlag, 2007. To appear.

[29] 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.

[30] 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.

[31] 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.

[32] J. Hughes. Restricted data types in Haskell. In E. Meijer, editor, *Proceedings of the 1999 Haskell Workshop*, number UU-CS-1999-28, 1999.

[33] 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.

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

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] Andres Löh. *Exploring Generic Haskell.* PhD thesis, Utrecht University, 2004.

[41] 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.

[42] 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*, volume 3145 of *Lecture Notes in Computer Science*, pages 168–184. Springer-Verlag, September 2003.

[43] 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.

[44] Simon Peyton Jones. *Haskell 98 Language and Libraries.* Cambridge University Press, 2003.

[45] 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.

[46] M. Pil. Dynamic types and type dependent functions. In Kevin Hammond, Antony J. T. Davie, and Chris Clack, editors, *Implementation of Functional Languages, 10th International Workshop, IFL'98, London, UK, September 9-11, Selected Papers*, volume 1595 of *Lecture Notes in Computer Science*, pages 169–185. Springer, 1999.

[47] Rinus Plasmeijer and Peter Achten. Generic editors for the world wide web. In ZoltÃ¡n HorvÃ¡th, editor, *Central European Functional Programming School – Revised Lecture Notes*, volume 4164 of *Lecture Notes in Computer Science*, pages 1–34. Springer, 2005.

[48] Fermín Reig. Generic proofs for combinator-based generic programs. In Hans-Wolfgang Loidl, editor, *Trends in Functional Programming*, volume 5. Intellect, 2006.

[49] Martin Sulzmann and Meng Wang. Modular generic programming with extensible superclasses. In *WGP '06: Proceedings of the 2006 ACM SIGPLAN workshop on Generic Programming*, pages 55–65. ACM Press, 2006.

[50] 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.

[51] 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.

[52] Stephanie Weirich. Encoding intensional type analysis. In *European Symposium on Programming*, volume 2028 of *LNCS*, pages 92–106. Springer-Verlag, 2001.

[53] Stephanie Weirich. Type-safe cast. *Journal of Functional Programming*, 14(6):681–695, November 2004.

[54] Stephanie Weirich. RepLib: a library for derivable type classes. In *Haskell '06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 1–12. ACM Press, 2006.

[55] Christoph Zenger. Indexed types. *Theoretical Computer Science*, 187(1–2):147–165, November 1997.