"Scrap Your Boilerplate" Revolutions

Ralf Hinze[1] and Andres Löh[1]

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

Abstract. Generic programming allows you to write a function once, and use it many times at different types. Traditionally, generic functions are defined by induction on the structure of types. "Scrap your boilerplate" (SYB) is a radically different approach that dispatches on the structure of values. In previous work, we have shown how to reconcile both approaches using the concept of generic views: many generic functions can be based either on the classical sum-of-products view or on the view that underlies SYB, the so-called 'spine' view. One distinct advantage of the spine view is its generality: it is applicable to a large class of data types, including generalised algebraic data types. Its main weakness roots in the value-orientation: one can only define generic functions that consume data (*show*) but not ones that produce data (*read*). Furthermore, functions that abstract over type constructors (*map*, *reduce*) are out of reach. In this paper, we show how to overcome both limitations. The main technical contributions are the 'type spine' view and the 'lifted spine' view.

1 Introduction

A generic function is one that the programmer writes once, but which is used over many different data types. The folklore examples are pretty printing, parsing, mapping functions, reductions, and so on. There is an impressive body of work on generic programming [1–3]. 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 two essential ingredients: a way to write overloaded functions, and independently, a way to access the structure of values in a uniform way.

Overloading is essential as almost every generic function exhibits type-specific behaviour: Haskell's pretty printer, for instance, displays pairs and lists using a special mix-fix notation.

A uniform mechanism for accessing the structure of values is essential to program the 'generic part' of a generic function: a generic pretty printer works for all data types including types that the programmer is 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.

The two ingredients are orthogonal concepts, and for both, there is a choice. In Haskell, overloaded functions can be expressed

- using the class system [4, 5],
- using a type-safe cast operation [6, 3],
- by reflecting the type structure onto the value level [7, 8],
- by specialisation [1, 9],
- or by a combination of the above [10].

Each approach has certain advantages and disadvantages. Nonetheless, they are mostly interchangeable and of similar expressiveness. For the purposes of this paper, we pick the third alternative, type reflection, as it is the most perspicuous.

The structural view, on the other hand, has a much larger impact: it affects the set of data types we can represent, the class of functions we can write and potentially the efficiency of these functions. For instance,

- PolyP [1] views data types as fixed points of regular functors,
- Generic Haskell [2] uses a sum-of-products view.

"Scrap your boilerplate" (SYB) [3] was originally introduced as a combinator library for generic programming, so it seemed to lack the structural view on data types. In a previous paper [11], we have revealed this structure:

- SYB [3] builds upon the so-called 'spine' view.

The spine view treats data uniformly as constructor applications; it is, in a sense, *value-oriented*. This is in contrast to the classical views of PolyP and Generic Haskell, which can be characterised as *type-oriented*. One distinct advantage of the spine view is its generality: it is applicable to a large class of data types, including *generalised algebraic data types* (GADTs) [12, 13]. The reason for the wide applicability is simple: a data type describes how to construct data, the spine view captures just this. Its main weakness also roots in the value-orientation: one can only define generic functions that consume data (*show*) but not ones that produce data (*read*). Again, the reason for the limitation is simple: a uniform view on individual constructor applications is useful if you have data in your hands, but it is of no help if you want to construct data. Furthermore, functions that abstract over type constructors (*map*, *reduce*) are out of reach, because type constructors comprise no values.

In this paper, we show how to overcome both limitations. The main technical contributions are the 'type spine' view for defining generic producers and the 'lifted spine' view, which renders it possible to define generic functions that abstract over type constructors.

The rest of the paper is structured as follows. In Section 2 we review the SYB approach to generic programming. We introduce the spine view and explain how to define generic consumers such as *show*. Section 3 introduces a variant of the spine view, the 'type spine' view, that allows us to write generic producers such as *read*. Section 4 then broadens the scope of SYB to generic functions that abstract over type constructors. In particular, we show how to implement classic generic functions such as *map*. Finally, Section 5 reviews related work and Section 6 concludes. For reference, Appendix A defines the helper functions that are used in the main body of the paper.

2 Recap: "Scrap Your Boilerplate" Reloaded

This section summarises the essentials ideas of the SYB approach to generic programming. The material is based on the paper " 'Scrap Your Boilerplate' Reloaded" [11]. Readers familiar with our previous work may wish to skim through Sections 2.1 and 2.2 and proceed with Section 2.3.

As noted in the introduction, support for generic programming consists of two essential ingredients: a way to write overloaded functions and a way to access the structure of values in a uniform way. Section 2.1 introduces *type reflection*, the mechanism we use to implement overloaded functions. This choice is entirely independent of the paper's main theme and has been taken with clarity in mind. Section 2.2 then reveals the generic view SYB builds upon.

2.1 Overloaded functions

Assume that you want to define a pretty printer, such as Haskell's *show* function, that works for a family of types including characters, integers, lists and pairs. The *show* function cannot be assigned the polymorphic type $\alpha \to \mathit{String}$, as *show* is not insensitive to what type its argument is. Quite on the contrary, the particular algorithm *show* invokes depends on the type: characters, for instance, are displayed differently from lists.

An obvious idea is to pass the pretty printer an additional argument that *represents* the type of the value that we wish to convert to its string representation. As a first try, we could assign the pretty printer the type $\mathit{Type} \to \alpha \to \mathit{String}$ where Type is the type of type representations. Unfortunately, this is too simple-minded: the parametricity theorem [14] 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 $\mathit{Type}\ \alpha \to \alpha \to \mathit{String}$. The idea is that an element of type $\mathit{Type}\ \tau$ is a representation of the type τ. Using a *generalised algebraic data type* [12, 13], we can define Type directly in Haskell.

```
data Type :: ∗ → ∗ where
  Char :: Type Char
  Int  :: Type Int
  List :: Type α → Type [α]
  Pair :: Type α → Type β → Type (α, β)
```

Each type has a unique representation: the type Int is represented by the constructor Int, the type $(\mathit{String}, \mathit{Int})$ is represented by $\mathit{Pair}\ (\mathit{List}\ \mathit{Char})\ \mathit{Int}$. In other words, $\mathit{Type}\ \tau$ is a so-called *singleton type*.

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

[1] The operator ':' is predefined in Haskell for constructing lists. However, since we use type annotations much more frequently than lists, we use ':' for the former and Nil

$$\begin{aligned}&\textbf{infixl}\ 1\ (:)\\&\textbf{data}\ \mathit{Typed}\ \alpha = (:)\{\mathit{val} :: \alpha, \mathit{type} :: \mathit{Type}\ \alpha\}\end{aligned}$$

Thus, $4711{:}\mathit{Int}$ is an element of $\mathit{Typed}\ \mathit{Int}$ and $(47, \texttt{"hello"}){:}\mathit{Pair}\ \mathit{Int}\ (\mathit{List}\ \mathit{Char})$ is an element of $\mathit{Typed}\ (\mathit{Int}, \mathit{String})$. It is important to note the difference between $x : t$ and $x :: \tau$. The former expression constructs a pair consisting of a value x and a representation t of its type. The latter expression is Haskell syntax for 'x has type τ'.

Given these prerequisites, we can define the desired family of pretty printers. For concreteness, we re-implement Haskell's *showsPrec* function (the *Int* argument of *showsPrec* specifies the operator precedence of the enclosing context; *ShowS* is shorthand for $\mathit{String} \to \mathit{String}$, Hughes' efficient sequence type [15]).

$$\begin{aligned}&\mathit{showsPrec} :: \mathit{Int} \to \mathit{Typed}\ \alpha \to \mathit{ShowS}\\&\mathit{showsPrec}\ d\ (c : \mathit{Char}) &&= \mathit{showsPrec}_{\mathit{Char}}\ d\ c\\&\mathit{showsPrec}\ d\ (n : \mathit{Int}) &&= \mathit{showsPrec}_{\mathit{Int}}\ d\ n\\&\mathit{showsPrec}\ d\ (s : \mathit{List}\ \mathit{Char}) &&= \mathit{showsPrec}_{\mathit{String}}\ d\ s\\&\mathit{showsPrec}\ d\ (xs : \mathit{List}\ a) &&= \mathit{showsList}\ [\mathit{shows}\ (x : a) \mid x \leftarrow xs]\\&\mathit{showsPrec}\ d\ ((x, y) : \mathit{Pair}\ a\ b)\\&\quad = \mathit{showChar}\ \texttt{'('} \cdot \mathit{shows}\ (x : a) \cdot \mathit{showChar}\ \texttt{','}\\&\qquad\qquad \cdot \mathit{shows}\ (y : b) \cdot \mathit{showChar}\ \texttt{')'}\end{aligned}$$

The function *showsPrec* makes heavy use of type annotations; its type $\mathit{Int} \to \mathit{Typed}\ \alpha \to \mathit{ShowS}$ is essentially an uncurried version of $\mathit{Int} \to \mathit{Type}\ \alpha \to \alpha \to \mathit{ShowS}$. Even though *showsPrec* has a polymorphic type, each equation implements a more specific case as dictated by the type representation. For example, the first equation has type $\mathit{Int} \to \mathit{Typed}\ \mathit{Char} \to \mathit{ShowS}$. This is typical of functions on GADTs.

Let us consider each equation in turn. The first three equations delegate the work to tailor-made functions, $\mathit{showsPrec}_{\mathit{Char}}$, $\mathit{showsPrec}_{\mathit{Int}}$ and $\mathit{showsPrec}_{\mathit{String}}$, which are provided from somewhere. Lists are shown using *showsList*, defined in Appendix A, which produces a comma-separated sequence of elements between square brackets. Note that strings, lists of characters, are treated differently: they are shown in double quotes by virtue of the third equation. Finally, pairs are enclosed in parentheses, the two elements being separated by a comma.

The function *showsPrec* is defined by case analysis on the type representation. This is typical of an overloaded function, but not compulsory: the wrapper functions *shows* and *show*, defined below, are given by simple abstractions.

$$\begin{aligned}&\mathit{shows} &&:: \mathit{Typed}\ \alpha \to \mathit{ShowS}\\&\mathit{shows} &&= \mathit{showsPrec}\ 0\\&\mathit{show} &&:: \mathit{Typed}\ \alpha \to \mathit{String}\\&\mathit{show}\ x &&= \mathit{shows}\ x\ \texttt{""}\end{aligned}$$

---

and *Cons* for the latter purpose. Furthermore, we agree upon that the pattern $x : t$ is matched from *right to left*: first the type representation t is matched, then the associated value x.

Note that *shows* and *showsPrec* are mutually recursive.

An overloaded function is a single entity that incorporates a family of functions where each member implements some type-specific behaviour. If we wish to extend the pretty printer to other data types we have to add new constructors to the *Type* data type and new equations to *showsPrec*. As an example, consider the data type of binary trees.

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

To be able to show binary trees, we add *Tree* to the type of type representations

$$\mathit{Tree} :: \mathit{Type}\ \alpha \to \mathit{Type}\ (\mathit{Tree}\ \alpha)$$

and extend *showsPrec* by suitable equations

$$\begin{aligned}
&\mathit{showsPrec}\ d\ (\mathit{Empty} : \mathit{Tree}\ a) = \mathit{showString}\ \texttt{"Empty"}\\
&\mathit{showsPrec}\ d\ (\mathit{Node}\ l\ x\ r : \mathit{Tree}\ a)\\
&\quad = \mathit{showParen}\ (d > 10)\ (\mathit{showString}\ \texttt{"Node"} \bullet \mathit{showsPrec}\ 11\ (l : \mathit{Tree}\ a)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \bullet \mathit{showsPrec}\ 11\ (x : a)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \bullet \mathit{showsPrec}\ 11\ (r : \mathit{Tree}\ a))
\end{aligned}$$

The predefined function *showParen* b puts its argument in parentheses if b is *True*. The operator '•' separates two elements by a space, see Appendix A.

2.2 Generic functions

Using type reflection we can program an *overloaded function* that works for all types of a given family. Let us now broaden the scope of *showsPrec*, *shows* and *show* so that they work for *all* data types including types that the programmer is yet to define. For emphasis, we call such functions *generic functions*.

We have seen in the previous section that whenever we define a new data type, we add a constructor of the same name to the type of type representations and we add corresponding equations to *all* overloaded functions that are defined by explicit case analysis. While the extension of *Type* is cheap and easy (a compiler could do this for us), the extension of all overloaded functions is laborious and difficult (can you imagine a compiler doing that?). In this section we shall develop a scheme so that it suffices to extend *Type* by a new constructor and to extend a *single* 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 data type. 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 '◊'. Additionally, we annotate the constructor arguments with their types and the constructor itself with information on its syntax. As an example, *Empty* becomes *Con empty* and *Node l a r* becomes *Con node*◊(*l*:*Tree Int*)◊(*a*:*Int*)◊(*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 data type 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. The following table illustrates the stepwise construction of a spine.

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

Note that the type variable α does not appear in the result type of '$\diamond$': it is existentially quantified. 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.

Elements of type *Constr* α comprise an element of type α, namely the original data constructor, plus some additional information about its syntax: for the purposes of this paper we confine ourselves to the name of the constructor.

```
data Constr α = Constr{ constr :: α, name :: String }
```

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

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

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

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

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

then the equation for *toSpine* takes the form

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

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

```
toSpine :: Typed α → Spine α
toSpine (Empty : Tree a)       = Con empty
toSpine (Node l x r : Tree a) = Con node ◇ (l : Tree a) ◇ (x : a) ◇ (r : Tree a)
```

The smart constructors *empty* and *node* are given by

```
empty :: Constr (Tree α)
empty = Constr{constr = Empty, name = "Empty"}
node  :: Constr (Tree α → α → Tree α → Tree α)
node  = Constr{constr = Node, name = "Node"}
```

With all the machinery in place we can now turn *showsPrec* into a truly generic function. The idea is to add a catch-all case that takes care of all the remaining type cases in a uniform manner.

```
showsPrec d x = showParen (arity x > 0 ∧ d > 10) (shows_ (toSpine x))
shows_ :: Spine α → ShowS
shows_ (Con c) = showString (name c)
shows_ (f ◇ x)  = shows_ f • showsPrec 11 x
```

The catch-all case displays its argument x using prefix notation. It first converts x into a spine, which the helper function *shows_* then traverses. Note that in the last equation x is of type *Typed* α; at this point we require the type information so that we can call *showsPrec* recursively. The *Tree* instance of *showsPrec* is subsumed by this general pattern, so the two *Tree* equations can be safely removed.

The function *arity* used above computes the arity of a data constructor. Its implementation follows the same definitional scheme as *showsPrec*:

```
arity :: Typed α → Int
arity = arity_ · toSpine
arity_ :: Spine α → Int
arity_ (Con c) = 0
arity_ (f ◇ x)  = arity_ f + 1
```

Interestingly, *arity* exhibits no type-specific behaviour; it is completely generic.

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

2.3 Discussion

The key to genericity is a uniform view on data. In the previous section we have introduced the spine view, which views data as constructor applications. Of course, this is not the only generic view. PolyP [1], for instance, views data types as fixed points of regular functors; Generic Haskell [2] uses a sum-of-products view. These two approaches can be characterised as *type-oriented*: they provide a uniform view on all elements of a data type. By contrast, the spine view is *value-oriented*: it provides a uniform view on single elements.

The spine view is particularly easy to use: the generic part of a generic function only has to consider two cases: *Con* and '$\diamond$'. By contrast, Generic Haskell distinguishes three cases, PolyP even six.

A further advantage of the spine view is its generality: it is applicable to a large class of data types. Nested data types [16], for instance, pose no problems: the type of perfect binary trees [17]

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

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

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

The equations follow exactly the general scheme introduced in Section 2.2. The scheme is even applicable to *generalised algebraic data types*. Consider as an example a typed representation of expressions.

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

The relevant equations for *toSpine* are

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

Given this definition we can apply *show* to values of type *Expr* without further ado. Note in this respect that the Glasgow Haskell Compiler (GHC) currently does not support **deriving** (*Show*) for GADTs. We can also turn *Type* itself into a representable type (recall that *Type* is a GADT). One may be tempted to consider this an intellectual curiosity, but it is not. The possibility to reflect *Type* is vital for implementing dynamic values.

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

Note that the type variable α does not appear in the result type: it is existentially quantified. However, since α is accompanied by a representation of its type, we can define a suitable *toSpine* instance.

```
Dynamic :: Type Dynamic
Type    :: Type α → Type (Type α)
Typed   :: Type α → Type (Typed α)
toSpine (Dyn x : Dynamic)  = Con dyn ◊ (x : Typed (type x))
toSpine ((x : t) : Typed a)  = Con hastype ◊ (x : t) ◊ (t : Type t)  -- t = a
toSpine (Char : Type Char) = Con char
...
```

It is important to note that the first instance does *not* follow the general pattern for *toSpine*. This points out the only limitation of the spine view: it can, in general, not cope with existentially quantified types. Consider, as an example, the following extension of the expression data type:

```
Apply :: Expr (α → β) → Expr α → Expr β
```

The equation for *toSpine*

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

is not legal Haskell, as a, the representation of α, appears free on the right-hand side. The only way out of this dilemma is to augment x by a representation of its type, as in *Dynamic*.[2]

To make a long story short: a data declaration describes how to construct data, the spine view captures just this. Consequently, it is applicable to almost every data type declaration. The classic views are much more restricted: Generic Haskell's sum-of-products view is only applicable to Haskell 98 types excluding GADTs and existential types; PolyP is even restricted to fixed points of regular functors excluding nested data types and higher-order kinded types.

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

On the other hand, the classic views provide more information as they represent the complete data type, 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 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 *map* and *reduce*) are out of reach for SYB. The latter deficiency is somewhat ironic as these functions are the classic examples of generics. In the following two sections we show how to overcome both limitations.

3 Extension I: the type spine view

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

$$
\begin{aligned}
&\mathit{consume} :: \mathit{Type}\ \alpha \to \alpha \to \tau\\
&\ldots\\
&\mathit{consume}\ a\ x\ \ = \mathit{consume}_{\_}\ (\mathit{toSpine}\ (x : a))\\
&\mathit{consume}_{\_}\ \ :: \mathit{Spine}\ \alpha \to \tau\\
&\mathit{consume}_{\_}\ \ldots\ = \ldots
\end{aligned}
$$

The element x is converted to the spine representation, over which the helper function $\mathit{consume}_{\_}$ then recurses. By duality, we would expect that a generic producer of type $\mathit{Type}\ \alpha \to \tau \to \alpha$, where α appears in the result type *but not* in τ, takes on the following form.

$$
\begin{aligned}
&\mathit{produce} :: \mathit{Type}\ \alpha \to \tau \to \alpha\\
&\ldots\\
&\mathit{produce}\ a\ t\ \ = \mathit{fromSpine}\ (\mathit{produce}_{\_}\ t)\\
&\mathit{produce}_{\_}\ \ :: \tau \to \mathit{Spine}\ \alpha\quad \text{-- does not work}\\
&\mathit{produce}_{\_}\ \ldots\ = \ldots
\end{aligned}
$$

The helper function $\mathit{produce}_{\_}$ generates an element in spine representation, which *fromSpine* converts back. Unfortunately, this approach does not work. The formal reason is that *toSpine* and *fromSpine* are different beasts: *toSpine* is an overloaded function, while *fromSpine* is parametrically polymorphic. If it were possible to define $\mathit{produce}_{\_} :: \tau \to \mathit{Spine}\ \alpha$, then the composition $\mathit{fromSpine} \cdot \mathit{produce}_{\_}$ would yield a parametrically polymorphic function of type $\tau \to \alpha$. And, indeed, a closer inspection of the catch-all case 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 data type, information that the spine view does not provide. Consider in this respect the syntactic form of a GADT (eg *Type* itself or *Expr* in Section 2.3): a data type is essentially a sequence of signatures. This motivates the following definitions.

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

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

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

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

char   :: Char → Constr Char
char c = Constr{constr = c, name = show_Char c}

int    :: Int → Constr Int
int i  = Constr{constr = i, name = show_Int  i}
```

Here, *nil*, *cons* and *pair* are the annotated versions of *Nil*, *Cons* and '(,)'. The function *datatype* plays the same role for producers as *toSpine* plays for *consumers*.

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

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

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

The helper function *generate_* constructs all terms that conform to a given signature. The right-hand side of the second equation essentially computes the

cartesian product of $generate_{\_}\ s\ d$ and $generate\ a\ d$. Here is a short interactive session that illustrates the use of *generate* (we assume a suitable *Bool* instance of *datatype*).

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

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

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

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

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

Finally, $arity'$ determines the arity of a constructor.

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

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

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

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

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

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

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

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

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

The type spine view shares the major advantage of the spine view: it is applicable to a large class of data types. Nested data types 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 data types. Since *Datatype* α is a homogeneous list, we have to partition the constructors according to their result types. Re-consider the expression data type of Section 2.3. 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):

```
Main⟩ let gen a d = putStrLn (show (generate a d : List a))
Main⟩ 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))]
Main⟩ 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))]
Main⟩ gen (Expr Char) 4
[]
```

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

In general, for each constructor C with signature

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

we add an element of the form

$$Sig\ c \square t_1 \square \cdots \square t_n$$

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

4 Extension II: the lifted spine view

The generic functions of the previous two sections abstract over a type. For instance, *shows* generalises functions of type

$$Char \to ShowS, \quad String \to ShowS, \quad [[Int]] \to ShowS$$

to a single generic function of type

$$Type\ \alpha \to \alpha \to ShowS \qquad \cong Typed\ \alpha \to ShowS$$

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

$$[\alpha] \to Int, \quad Tree\ \alpha \to Int, \quad [\,Tree\ \alpha\,] \to Int$$

to a single generic function of type

$$Type'\ \varphi \to \varphi\ \alpha \to Int \qquad \cong Typed'\ \varphi\ \alpha \to Int$$

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

The original spine view is not appropriate in this context as it cannot capture type abstractions. To illustrate, consider a variant of *Tree* whose inner nodes are annotated with an integer, say, a balance factor.

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

If we call the generic function on a value of type $BalTree\ Int$, then the two integer components are handled in a uniform way. This is fine for generic functions on types, but not acceptable for generic functions on type constructors. In the Sections 4.1 and 4.2 we introduce suitable variants of *Type* and *Spine* that can be used to define the latter brand of generic functions.

4.1 Lifted types

To represent type constructors of kind $* \to *$ we introduce a new tailor-made representation type.

```
data Type' :: (* → *) → * where
  List :: Type' []
  Tree :: Type' Tree

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

The type is only inhabited by two constructors since the other data types listed in *Type* have kinds different from $* \to *$.

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

```
size :: Typed' φ α → Int
size (Nil :' List)          = 0
size (Cons x xs :' List)    = 1 + size (xs :' List)
size (Empty :' Tree)        = 0
size (Node l x r :' Tree)   = size (l :' Tree) + 1 + size (r :' Tree)
```

Unfortunately, the overloaded function *size* is not as flexible as *shows*. If we have some compound data structure x, say, a list of trees of integers, then we

can simply call *shows* $(x : List\ (Tree\ Int))$. We cannot, however, use *size* to count the total number of integers, simply because the new versions of $List$ and $Tree$ take no arguments.

There is one further problem, which is more fundamental. Computing the size of a compound data structure is inherently ambiguous: in the example above, shall we count the number of integers, the number of trees or the number of lists? Formally, we have to solve the type equation $\varphi\ \tau = List\ (Tree\ Int)$. The equation has, in fact, not three but four principal solutions: $\varphi = \Lambda\alpha \to \alpha$ and $\tau = List\ (Tree\ Int)$, $\varphi = \Lambda\alpha \to List\ \alpha$ and $\tau = Tree\ Int$, $\varphi = \Lambda\alpha \to List\ (Tree\ \alpha)$ and $\tau = Int$, and $\varphi = \Lambda\alpha \to List\ (Tree\ Int)$ and τ arbitrary. How can we represent these different container types? One possibility is to *lift* the type constructors [9] so that they become members of $Type'$ and to include Id, the identity type, as a representation of the type variable α:

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

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

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

$$\begin{aligned}
&\mathbf{newtype}\ Id\ &&\chi = In_{Id}\ \ \{out_{Id}\ &&:: \chi\ \ \}\\
&\mathbf{newtype}\ Char'\ &&\chi = In_{Char'}\{out_{Char'}\ &&:: Char\}\\
&\mathbf{newtype}\ Int'\ &&\chi = In_{Int'}\ \ \{out_{Int'}\ &&:: Int\ \ \}\\
&\mathbf{data}\ List'\ \alpha'\ &&\chi = Nil' \mid Cons'\ (\alpha'\ \chi)\ (List'\ \alpha'\ \chi)\\
&\mathbf{data}\ Pair'\ \alpha'\ \beta'\ &&\chi = Pair'\ (\alpha'\ \chi)\ (\beta'\ \chi)\\
&\mathbf{data}\ Tree'\ \alpha'\ &&\chi = Empty' \mid Node'\ (Tree'\ \alpha'\ \chi)\ (\alpha'\ \chi)\ (Tree'\ \alpha'\ \chi)
\end{aligned}$$

The lifted variants of the nullary type constructors Int and $Char$ 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.

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

The instances are similar to the ones for the unlifted types except that *size* is now also called recursively for 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 is a list of trees of integers, since $\mathit{List}'\ (\mathit{Tree}'\ \mathit{Id})\ \mathit{Int}$ is different from $[\mathit{Tree}\ \mathit{Int}]$. We address this problem later in Section 4.3 after we have introduced the lifted spine view.

4.2 Lifted spine view

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 $\forall\chi.(\tau'_1 \mathbin{\dot\to} \cdots \mathbin{\dot\to} \tau'_n \mathbin{\dot\to} \tau'_0)\ \chi$, using the lifted functions space:

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

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

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

Now, 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.

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

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

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

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

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

$$C' :: \forall\chi.\tau'_1\ \chi \to \cdots \to \tau'_n\ \chi \to \tau'_0\ \chi$$

gives rise to an equation of the form

$$\mathit{toSpine'}\ (C'\ x_1\ \ldots\ x_n :' t'_0) = \mathit{Con}\ c'\ \diamond\ (x_1 : t'_1)\ \diamond \cdots \diamond\ (x_n : t'_n)$$

where c' is the variant of C' that uses the lifted function space and t'_i is the type representation of the lifted type τ'_i. As an example, here is the instance for lifted lists.

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

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

The *Spine′* data type provides the generic view that allows us to implement the 'generic part' of a generic function. The following declarations make the concept of a generic view explicit.

$$\begin{aligned}
&\mathbf{infixr}\ 5\ \twoheadrightarrow\\
&\mathbf{infixl}\ 5\ \twoheadleftarrow\\
&\mathbf{type}\ \varphi \twoheadrightarrow \psi = \forall\alpha.\varphi\ \alpha \to \psi\ \alpha\\
&\mathbf{type}\ \varphi \twoheadleftarrow \psi = \forall\alpha.\psi\ \alpha \to \varphi\ \alpha\\
&\mathbf{data}\ \mathit{View'} :: (* \to *) \to * \ \mathbf{where}\\
&\quad \mathit{View'} :: \mathit{Type'}\ \psi \to (\varphi \twoheadrightarrow \psi) \to (\varphi \twoheadleftarrow \psi) \to \mathit{View'}\ \varphi
\end{aligned}$$

A view consists of three ingredients: a so-called *structure type* that provides a uniform view on the original type and two functions that convert to and fro. In our case, the structure view of φ is simply $\mathit{Spine'}\ \varphi$.

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

Given these prerequisites we can finally turn *size* into a generic function.

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

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 $Spine'\ \varphi$ (this will change in the next section), so the first equation applies, which in turn delegates the work to the helper function $size\_$.

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

The implementation of $size\_$ is entirely straightforward: it traverses the spine summing up the sizes of the constructors 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:

$$
\begin{aligned}
&map :: Type'\ \varphi \to (\alpha \to \beta) \to (\varphi\ \alpha \to \varphi\ \beta) \\
&map\ Id \qquad\qquad\ m = In_{Id} \cdot m \cdot out_{Id} \\
&map\ (Spine'\ a')\ m = map\_\ m \\
&map\ a' \qquad\qquad\ m = \mathbf{case}\ spineView\ a'\ \mathbf{of} \\
&\qquad\qquad\qquad\qquad View'\ b'\ from\ to \to to \cdot map\ b'\ m \cdot from \\
&map\_ :: (\alpha \to \beta) \to (Spine'\ \varphi\ \alpha \to Spine'\ \varphi\ \beta) \\
&map\_\ m\ (Con'\ c) \qquad\ = Con'\ c \\
&map\_\ m\ (f\ \diamond'\ (x :'\ a')) = map\_\ m\ f\ \diamond'\ (map\ a'\ m\ x :'\ a')
\end{aligned}
$$

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

4.3 Bridging the gap

We have noted in Section 4.1 that the generic size function does not work on the original, unlifted types as they are different from the lifted ones. However, both are closely related: if τ' is the lifted variant of τ, then $\tau'\ Id$ is isomorphic to τ [9]. Even more, $\tau'\ Id$ and τ can share the same run-time representation, since Id is defined by a **newtype** declaration and since the lifted data type τ' has exactly the same structure as the original data type τ.

As an example, the functions $fromList\ In_{Id}$ and $toList\ out_{Id}$ exhibit the isomorphism between $[\,]$ and $List'\ Id$.

$$
\begin{aligned}
&fromList :: (\alpha \to \alpha'\ \chi) \to ([\alpha] \to List'\ \alpha'\ \chi) \\
&fromList\ from\ Nil \qquad\ = Nil' \\
&fromList\ from\ (Cons\ x\ xs) = Cons'\ (from\ x)\ (fromList\ from\ xs)
\end{aligned}
$$

```
toList    :: (α' χ → α) → (List' α' χ → [α])
toList to Nil'          = Nil
toList to (Cons' x xs) = Cons (to x) (toList to xs)
```

Operationally, if the types $\tau'\ Id$ and τ have the same run-time representation, then $fromList\ In_{Id}$ and $toList\ out_{Id}$ are identity functions (the Haskell Report [18] 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 (the equation below must be inserted before the catch-all case).

```
spineView List = View' (List' Id) (fromList In_Id) (toList out_Id)
```

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

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

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 [19], the calls almost always involve a change on the value level: The type equation $\varphi\ \tau = List\ (Tree\ Int)$, for instance, is solved setting $\varphi = List$ and $\tau = Tree\ Int$, 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 desired call.

4.4 Discussion

The lifted spine view is almost as general as the original spine view: it is applicable to all data types that are definable in Haskell 98. In particular, nested data types can be handled with ease. As an example, for the data type *Perfect*, see Section 2.3, we introduce a lifted variant

```
data Perfect' α' χ = Zero' (α' χ) | Succ' (Perfect' (Pair' α' α') χ)

Perfect  :: Type' Perfect
Perfect' :: Type' φ → Type' (Perfect' φ)

toSpine' (Zero' x :' Perfect' a') = Con' zero' ◊' (x :' a')
toSpine' (Succ' x :' Perfect' a') = Con' succ' ◊' (x :' Perfect' (Pair' a' a'))
```

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

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

The following interactive session shows some examples involving perfect trees.

```
Main⟩ size (Succ (Zero (1, 2)) :' Perfect)
2
Main⟩ map (Perfect) (+1) (Succ (Zero (1, 2)))
Succ (Zero (2, 3))
```

We have seen in Section 2.3 that the spine view is also applicable to *generalised algebraic data types*. This does not hold for the lifted spine view, as it is not possible to generalise *map* to GADTs. Consider the expression data type of Section 2.3. Though *Expr* is parameterised, it is not a container type: an element of *Expr Int*, for instance, is an expression that evaluates to an integer; it is not a data structure that contains integers. This means, in particular, that we cannot define a mapping function $(\alpha \to \beta) \to (Expr\ \alpha \to Expr\ \beta)$: How could we possibly turn expressions of type $Expr\ \alpha$ into expression of type $Expr\ \beta$? The type $Expr\ \beta$ might not even be inhabited: there are, for instance, no terms of type *Expr String*. Since the type argument of *Expr* is not related to any component, *Expr* is also called a *phantom type* [20].

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

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

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

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

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

In the preceding section we have seen two examples of generic consumers (or transformers). As in the first-order case generic producers are out of reach and for exactly the same reason: *fromSpine'* is a polymorphic function while *toSpine'* is overloaded. Of course, the solution to the problem suggests itself: we must also

lift the type spine view to type constructors of kind $* \to *$. In a sense, the spine view really comprises two views: one for consumers (and transformers) and one for pure producers.

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. The answer is an emphatic "Yes!". Let us briefly sketch the main steps, for a formal treatment see Hinze's earlier work [9]. Assume that $\kappa = \kappa_1 \to \cdots \to \kappa_n \to *$ is the kind of the type index. We first introduce a suitable type representation and lift the data types to kind κ by adding n type arguments of kind $\kappa_1, \ldots, \kappa_n$. Types and lifted types are related as follows: τ is isomorphic to $\tau'\ Out_1\ \ldots\ Out_n$ where Out_i is the *projection type* that corresponds to the i-th argument of κ. The spine representation must be lifted accordingly. 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, generic functions that are indexed by higher-order kinds, for instance, $(* \to *) \to * \to *$ are rare. In practice, most generic functions are indexed by a first-order kind such as $*$ or $* \to *$.

5 Related Work

Scrap your boilerplate The SYB approach has been developed by Lämmel and Peyton Jones in a series of papers [3, 21, 5]. The original approach is *combinator-based*: the user writes generic functions by combining a few generic primitives. The first paper [3] introduces two main combinators: a type-safe cast for defining ad-hoc cases and a generic recursion operator, called *gfoldl*, for implementing the generic part. It turns out that *gfoldl* is essentially the catamorphism of the *Spine* data type [11]: *gfoldl* equals the catamorphism composed with *toSpine*. The second paper [21] adds a function called *gunfold* to the set of predefined combinators, which is required for defining generic producers. The name suggests that the new combinator is the anamorphism of the *Spine* type, but it is not: *gunfold* is actually the catamorphism of *Signature*, introduced in Section 3.

Relating approaches to generic programming There is a wealth of material on the subject of generic programming. The tutorials [22, 2, 23] provide an excellent overview of the field. We have noted in the introduction that support for generic programming consists of two essential ingredients: a way to write overloaded functions, and independently, a way to access the structure of values in a uniform way. The different approaches to generic programming can be faithfully classified along these two dimensions. Figure 1 provides an overview of the design space. The two ingredients 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 [24, 7, 25–27]	–	–	–
fixed point	Reloaded [11]	PolyP [28, 29]	–	PolyP [1]
sum-of-products	LIGD [8, 20]	DTC [30], GC [34], GM [10]	–	GH [31, 2, 32, 33]
spine	Reloaded [11], this paper	SYB [5], Reloaded [35]	SYB [3, 21]	–

Fig. 1. Generic programming: the design space.

- *type reflection*: This is the approach we have used in this paper. Its origins can be traced back to the work on intensional type analysis [24, 7, 25–27] (ITA). 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.
- *type classes* [4]: 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 [34] (GC). Clean's type classes are indexed by kind so that a single generic function can be applied to type constructors of different kinds. A pure Haskell 98 implementation of generics (GM) is described by Hinze [10]. The implementation builds upon a class-based encoding of the type *Type* of type representations.
- *type-safe cast* [6]: 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 [3] is based on casts.
- *specialisation* [9]: 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 [1] or *Generic Haskell* [33], that are set up as preprocessors or compilers.

The approaches differ mostly in syntax and style, but less in expressiveness—except perhaps for specialisation, which cannot cope with higher-order generic functions. The second dimension, the generic view, has a much larger impact: we have seen that it affects the set of data types we can represent, 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 underly ITA are structural.
- *fixed point view*: PolyP [1] 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 data types of kind $* \to *$ that are regular can be represented, excluding nested data types and higher-order kinded data types. Its particular strength is that recursion patterns such as cata- or anamorphisms can be expressed generically, because each data type 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 [28] embeds PolyP program into Haskell augmented by multiple parameter type classes with functional dependencies [36]. Oliveira and Gibbons [29] present a light-weight variant of PolyP that works within Haskell 98.
- *sum-of-products view*: Generic Haskell [2, 32, 33] (GH) builds upon this view. It is applicable to all data types definable in Haskell 98. Generic Haskell is a full-fledged implementation of generics based on ideas by Hinze [31, 37] that features generic functions, generic types and various extensions such as default cases and constructor cases [38]. Generic Haskell supports the definition of functions that work for all types of all kinds, such as, for example, a generalised mapping function.
- *spine views*: The spine view treats data uniformly as constructor applications. The different spine views have been extensively discussed in Sections 2.3, 3 and 4.4.

6 Conclusion

The SYB approach to generic programming was originally presented as an implementation of *strategic programming* in Haskell. Strategic programming [39] is an idiom for processing and querying complex, compound data such as, for example, abstract syntax trees. Because of this background and because of the particular presentation as a combinator library, the approach seemed to be tied

to generic consumers indexed by types of kind $*$. This paper makes the following contributions revealing the full potential of the SYB approach.

- The 'type spine' view allows us to implement generic producers in the same elegant manner as generic consumers that build upon the spine view. The type spine view can be seen as the hidden structure that underlies the *gunfold* combinator.
- Functions that abstract over type constructors can be handled using the technique of *lifting*. Previously, these functions were thought to be out of reach for the SYB approach. For reasons of space, we have confined ourselves to type indices of kind $* \to *$. Lifting, however, works for indices of arbitrary kinds.

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. The generic programmer only has to consider two cases plus one case for each argument of the type index (that is, n additional cases for indices of kind $\kappa_1 \to \cdots \to \kappa_n \to *$). The spine view and the type spine view cover almost all data types *including* generalised algebraic data types, but excluding existential types. For principal reasons, the lifted spine view is more restricted: generic functions that abstract over type constructors can be instantiated to arbitrary *container types* but not to *phantom types* (GADTs).

We have left a couple of topics for future work. The overloading technique used in this paper, type reflection, hinders in its present form the formulation of the approach as a re-usable library: the encoding of overloaded functions using explicit type arguments requires the extensibility of the *Type* data type and of functions such as *toSpine*. Using the concepts of open data types and open functions [40] this limitation can be overcome. We plan to build an industrial-strength library based on this extension. Type reflection has at least one distinct advantage over a class-based approach: we expect that it is much easier to prove algebraic properties of generic functions in this setting. We believe that the work of Reig [41] could be recast using our approach, leading to shorter and more concise proofs.

Acknowledgements

We would like to thank Jeremy Gibbons for correcting the English and Stefan Holdermans for many suggestions regarding 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.

References

1. Jansson, P., Jeuring, J.: PolyP—a polytypic programming language extension. In: Conference Record 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97), Paris, France, ACM Press (1997) 470–482

2. Hinze, R., Jeuring, J.: Generic Haskell: Practice and theory. In Backhouse, R., Gibbons, J., eds.: Generic Programming: Advanced Lectures. Volume 2793 of Lecture Notes in Computer Science. Springer-Verlag (2003) 1–56
3. Peyton Jones, S., Lämmel, R.: 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. (2003) 26–37
4. Hall, C.V., Hammond, K., Peyton Jones, S.L., Wadler, P.L.: Type classes in Haskell. ACM Transactions on Programming Languages and Systems **18**(2) (1996) 109–138
5. Lämmel, R., Peyton Jones, S.: Scrap your boilerplate with class: extensible generic functions. In Pierce, B., ed.: Proceedings of the 2005 International Conference on Functional Programming, Tallinn, Estonia, September 26–28, 2005. (2005)
6. Weirich, S.: Type-safe cast: functional pearl. In: Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '00). Volume (**35**)9 of ACM SIGPLAN Notices., N.Y., ACM Press (2000) 58–67
7. Crary, K., Weirich, S., Morrisett, G.: 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., ACM Press (1999) 301–312
8. Cheney, J., Hinze, R.: A lightweight implementation of generics and dynamics. In Chakravarty, M.M., ed.: Proceedings of the 2002 ACM SIGPLAN Haskell Workshop, ACM Press (2002) 90–104
9. Hinze, R.: A new approach to generic functional programming. In Reps, T.W., ed.: Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00), Boston, Massachusetts, January 19-21. (2000) 119–132
10. Hinze, R.: Generics for the masses. In Fisher, K., ed.: Proceedings of the 2004 International Conference on Functional Programming, Snowbird, Utah, September 19–22, 2004, ACM Press (2004) 236–243
11. Hinze, R., Löh, A., Oliveira, B.C.d.S.: "Scrap Your Boilerplate" reloaded. In Wadler, P., Hagiya, M., eds.: Proceedings of the Eighth International Symposium on Functional and Logic Programming (FLOPS 2006), 24-26 April 2006, Fuji Susono, Japan. Lecture Notes in Computer Science, Springer-Verlag (2006)
12. Xi, H., Chen, C., Chen, G.: Guarded recursive datatype constructors. In: Proceedings of the ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL 2003), ACM Press (2003) 224–235
13. Peyton Jones, S., Washburn, G., Weirich, S.: Wobbly types: Type inference for generalised algebraic data types. Technical Report MS-CIS-05-26, University of Pennsylvania (2004)
14. Wadler, P.: Theorems for free! In: The Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA'89), London, UK, Addison-Wesley Publishing Company (1989) 347–359
15. Hughes, R.J.M.: A novel representation of lists and its application to the function "reverse". Information Processing Letters **22**(3) (1986) 141–144
16. Bird, R., Meertens, L.: Nested datatypes. In Jeuring, J., ed.: Fourth International Conference on Mathematics of Program Construction, MPC'98, Marstrand, Sweden. Volume 1422 of Lecture Notes in Computer Science., Springer-Verlag (1998) 52–67
17. Hinze, R.: Functional Pearl: Perfect trees and bit-reversal permutations. Journal of Functional Programming **10**(3) (2000) 305–317

18. Peyton Jones, S.: Haskell 98 Language and Libraries. Cambridge University Press (2003)
19. Jones, M.P.: A system of constructor classes: overloading and implicit higher-order polymorphism. Journal of Functional Programming **5**(1) (1995) 1–35
20. Hinze, R.: Fun with phantom types. In Gibbons, J., de Moor, O., eds.: The Fun of Programming. Palgrave Macmillan (2003) 245–262 ISBN 1-4039-0772-2 hardback, ISBN 0-333-99285-7 paperback.
21. Lämmel, R., Peyton Jones, S.: Scrap more boilerplate: reflection, zips, and generalised casts. In Fisher, K., ed.: Proceedings of the 2004 International Conference on Functional Programming, Snowbird, Utah, September 19–22, 2004. (2004) 244–255
22. Backhouse, R., Jansson, P., Jeuring, J., Meertens, L.: Generic Programming — An Introduction —. In Swierstra, S.D., Henriques, P.R., Oliveira, J.N., eds.: 3rd International Summer School on Advanced Functional Programming, Braga, Portugal. Volume 1608 of Lecture Notes in Computer Science. Springer-Verlag, Berlin (1999) 28–115
23. Hinze, R., Jeuring, J.: Generic Haskell: Applications. In Backhouse, R., Gibbons, J., eds.: Generic Programming: Advanced Lectures. Volume 2793 of Lecture Notes in Computer Science. Springer-Verlag (2003) 57–97
24. Harper, R., Morrisett, G.: Compiling polymorphism using intensional type analysis. In: 22nd Symposium on Principles of Programming Languages, POPL '95. (1995) 130–141
25. Crary, K., Weirich, S.: Flexible type analysis. In: Proceedings ICFP 1999: International Conference on Functional Programming, ACM Press (1999) 233–248
26. Trifonov, V., Saha, B., Shao, Z.: Fully reflexive intensional type analysis. In: Proceedings ICFP 2000: International Conference on Functional Programming, ACM Press (2000) 82–93
27. Weirich, S.: Encoding intensional type analysis. In: European Symposium on Programming. Volume 2028 of LNCS., Springer-Verlag (2001) 92–106
28. Norell, U., Jansson, P.: Polytypic programming in Haskell. In Trinder, P., Michaelson, G., Peña, R., eds.: Implementation of Functional Languages: 15th International Workshop, IFL 2003, Edinburgh, UK, September 8-11, 2003. (2003) 168–184
29. Oliveira, B.C., Gibbons, J.: Typecase: A design pattern for type-indexed functions. In Leijen, D., ed.: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell, Tallinn, Estonia. (2005) 98–109
30. Hinze, R., Peyton Jones, S.: Derivable type classes. In Hutton, G., ed.: Proceedings of the 2000 ACM SIGPLAN Haskell Workshop. Volume 41.1 of Electronic Notes in Theoretical Computer Science., Elsevier Science (2001) The preliminary proceedings appeared as a University of Nottingham technical report.
31. Hinze, R.: Polytypic values possess polykinded types. Science of Computer Programming **43** (2002) 129–159
32. Löh, A.: Exploring Generic Haskell. PhD thesis, Utrecht University (2004)
33. Löh, A., Jeuring, J.: The Generic Haskell user's guide, version 1.42 - Coral release. Technical Report UU-CS-2005-004, Universiteit Utrecht (2005)
34. Alimarine, A., Plasmeijer, R.: A generic programming extension for Clean. In Arts, T., Mohnen, M., eds.: Proceedings of the 13th International workshop on the Implementation of Functional Languages, IFL'01, Älvsjö, Sweden (2001) 257–278
35. Hinze, R., Löh, A., Oliveira, B.C.: "Scrap Your Boilerplate" reloaded. Technical Report IAI-TR-2006-1, Institut für Informatik III, Universität Bonn (2006)
36. Jones, M.P.: Type classes with functional dependencies. In Smolka, G., ed.: Proceedings of the 9th European Symposium on Programming, ESOP 2000, Berlin,

Germany. Volume 1782 of Lecture Notes in Computer Science., Springer-Verlag (2000) 230–244
37. Hinze, R., Jeuring, J., Löh, A.: Type-indexed data types. Science of Computer Programming **51** (2004) 117–151
38. Clarke, D., Löh, A.: Generic Haskell, specifically. In Gibbons, J., Jeuring, J., eds.: Proceedings of the IFIP TC2 Working Conference on Generic Programming, Schloss Dagstuhl, Kluwer Academic Publishers (2002) 21–48
39. Visser, E.: Language independent traversals for program transformation. In Jeuring, J., ed.: Proceedings of the 2nd Workshop on Generic Programming, Ponte de Lima, Portugal. (2000) 86–104 The proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-2000-19.
40. Hinze, R., Löh, A.: Open data types and open functions. Technical Report IAI-TR-2006-2, Institut für Informatik III, Universität Bonn (2006)
41. Reig, F.: Generic proofs for combinator-based generic programs. In Loidl, H.W., ed.: Trends in Functional Programming. Volume 5. Intellect (2006)
42. Braun, W., Rem, M.: A logarithmic implementation of flexible arrays. Memorandum MR83/4, Eindhoven University of Technology (1983)

A Library

The function *tree* turns a list of elements into a balanced binary tree, a so-called *Braun tree* [42].

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

The type *ShowS* is Haskell's type of pretty printers. The operator '•' separates two elements of this type by a space.

```
(•) :: ShowS → ShowS → ShowS
s₁ • s₂ = s₁ · showChar ' ' · s₂
```

The function *showsList* produces a comma-separated sequence of elements between square brackets.

```
showsList :: [ShowS] → ShowS
showsList Nil          = showString "[]"
showsList (Cons x xs) = showChar '[' · x
                        · foldr (·) id [showChar ',' · s | s ← xs]
                        · showChar ']'
```

The type *ReadS* is Haskell's parser type. The function *alt* implements the alternation of a list of parsers.

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

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

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