Generics for the masses

RALF HINZE
Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
(*e-mail:* ralf@informatik.uni-bonn.de)

Abstract

A generic function is a function that can be instantiated on many data types to obtain data type specific functionality. Examples of generic functions are the functions that can be derived in Haskell, such as *show*, *read*, and '=='. The recent years have seen a number of proposals that support the definition of generic functions. Some of the proposals define new languages, some define extensions to existing languages. As a common characteristic none of the proposals can be made to work within Haskell 98: they all require something extra, either a more sophisticated type system or an additional language construct. The purpose of this paper is to show that one can, in fact, program generically within Haskell 98 obviating to some extent the need for fancy type systems or separate tools. Haskell's type classes are at the heart of this approach: they ensure that generic functions can be defined succinctly and, in particular, that they can be used painlessly. We detail three different implementations of generics both from a practical and from a theoretical perspective.

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 (Peyton Jones, 2003) it is simply not possible to define an equality test that works for all types. Polymorphism does not help: equality is not a polymorphic function since it must inspect its arguments. Static typing dictates that equality becomes a family of functions containing a tailor-made instance of equality for each type of interest. Rather annoyingly, all these instances have to be programmed.

More than a decade ago the designers of Haskell noticed and partially addressed this problem. By attaching a so-called *deriving form* to a data type 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

[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.

have to become known as *data-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.

The recent years have seen a number of proposals (Jansson & Jeuring, 1997; Hinze & Peyton Jones, 2001; Cheney & Hinze, 2002; Hinze & Jeuring, 2003b; Norell & Jansson, 2003) that support exactly this, the *definition* of generic functions. Some of the proposals define new languages, some define extensions to existing languages. However, none of the proposals can be made to work within Haskell 98: they all require something extra, either a more sophisticated type system or an additional language construct.

The purpose of this paper is to show that one can, in fact, program generically within Haskell 98 obviating to some extent the need for fancy type systems or separate tools. The proposed approach is extremely light-weight; each implementation of generics—we will introduce three major ones and a few variations—consists roughly of two dozen lines of Haskell code. The technique is surprisingly expressive: we can define all the generic functions presented, for instance, in (Hinze, 2002). Of course, there are also limitations: for instance, defining functions that involve *generic types* (Hinze *et al.*, 2004) seems out of reach. On the other hand, the code can be easily adapted to one's needs. Indeed, the reader is cordially invited to play with the material. The source code can be found at

`http://www.informatik.uni-bonn.de/~ralf/masses.tar.bz2`

We have also included several exercises to support digestion of the material and to stimulate further experiments.

The rest of the paper is structured as follows. The first part, consisting of Sections 2, 3 and 4, introduces three implementations of generics. This part is largely written in a tutorial style introducing the approach to a potential user, that is, a generic programmer. The theoretical background is then investigated in the second part, Section 5, which derives the two major approaches from first principles. The two parts are largely independent. The reader who is keen to see the inner workings may wish to skim through the first part, read the second part and then go back to the first. Finally, Section 6 summarizes the main points and Section 7 provides references for further studies and reviews related work.

2 Generic functions on types

This section discusses our first implementation of generics. Section 2.1 shows how to embed a generic definition into Haskell 98 covering what you would expect from a paper on generics. However, this is not the whole story. Whenever the user defines a new data type, she has to do a bit of extra work so that a generic function can be instantiated to that type. This extra work is detailed in Section 2.2. Furthermore, some additional code is needed, which is shared among the generic definitions. Section 2.3 provides the details.

Most if not all approaches to generics contain these three facets: code for generic definitions, per data type code, and shared library code. In most cases, however,

the per data type code is not burdened upon the generic programmer but is generated automatically. In a sense, this is the price we have to pay for staying within Haskell 98. On the other hand, since neither language extensions nor compiler modifications are required, the approach can be easily modified or extended. Section 2.4 takes a look at various extensions, some obvious and some perhaps less so.

2.1 Defining a generic function

Let us tackle a concrete problem. Suppose we want to encode elements of various data types as bit strings implementing a simple form of data compression. For simplicity, we represent a bit string by a list of bits.

$$\begin{aligned}&\textbf{type } \mathit{Bin} = [\mathit{Bit}]\\&\textbf{data } \mathit{Bit} \;= \mathtt{0} \mid \mathtt{1} \textbf{ deriving } (\mathit{Show})\\&\mathit{bits} \qquad :: (\mathit{Enum}\ \alpha) \Rightarrow \mathit{Int} \to \alpha \to \mathit{Bin}\end{aligned}$$

We assume a function *bits* that encodes an element of an enumeration type using the specified number of bits. We seek to generalize *bits* to a function *showBin* that works for arbitrary types. Here is a simple interactive session that illustrates the use of *showBin* (note that characters consume 7 bits and integers 16 bits).

```
Main⟩ showBin 3
[1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
Main⟩ showBin [3,5]
[1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0]
Main⟩ showBin "Lisa"
[1,0,0,1,1,0,0,1,1,1,0,0,1,0,1,1,1,1,1,0,0,1,1,1,1,1,0,0,0,0,1,1,0]
```

A string of length n, for instance, is encoded in $8 * n + 1$ bits.

Implementing *showBin* so that it works for arbitrary data types seems like a hard nut to crack. The good news is that it suffices to define *showBin* for primitive types and for three *elementary types*: the one-element type, the binary sum, and the binary product.

$$\begin{aligned}&\textbf{data } \mathit{Unit} \qquad = \mathit{Unit}\\&\textbf{data } \mathit{Plus}\ \alpha\ \beta = \mathit{Inl}\ \alpha \mid \mathit{Inr}\ \beta\\&\textbf{data } \mathit{Pair}\ \alpha\ \beta = \mathit{Pair}\{\,\mathit{outl} :: \alpha, \mathit{outr} :: \beta\,\}\end{aligned}$$

As an aside, the latter definition uses Haskell's record syntax to simultaneously introduce a constructor function $\mathit{Pair} :: \alpha \to \beta \to \mathit{Pair}\ \alpha\ \beta$ and two selector functions $\mathit{outl} :: \mathit{Pair}\ \alpha\ \beta \to \alpha$ and $\mathit{outr} :: \mathit{Pair}\ \alpha\ \beta \to \beta$.

Why these three types? Well, Haskell's construct for defining new types, the **data** declaration, introduces a type that is isomorphic to a sum of products. Thus, if we know how to compress sums and products, we can compress elements of an arbitrary data type. More generally, we can handle a type σ if we can handle some representation type τ that is isomorphic to σ. The details of the representation type are largely irrelevant, so we abstract away from them: When programming a generic function it suffices to know the two mappings that witness the isomorphism.

$$\textbf{data}\ \mathit{Iso}\ \alpha\ \beta = \mathit{Iso}\{\mathit{fromData} :: \beta \to \alpha, \mathit{toData} :: \alpha \to \beta\}$$

In what follows β will always be the original data type and α its representation type.

Turning to the implementation of *showBin*, we first have to provide the signature of the generic function. Rather unusually, we specify the type using a **newtype** declaration.

$$\textbf{newtype}\ \mathit{ShowBin}\ \alpha = \mathit{ShowBin}\{\,\mathit{appShowBin} :: \alpha \to \mathit{Bin}\,\}$$

An important point is that you should read the above declaration as a type signature; the **newtype** declaration is just an idiom for embedding generics in Haskell 98.

An element of *ShowBin* σ is an instance of *showBin* that encodes values of type σ as bit strings. We know that the generic function *showBin* itself cannot be a genuine polymorphic function of type $\alpha \to \mathit{Bin}$. Data compression does not work for arbitrary types, but only for types that are *representable*. Representable means that the *type* can be represented by a certain *value*. For the moment, it suffices to know that a type representation is simply an overloaded value called *rep*. The first part of the generic compression function is then given by the following definition.

$$\begin{aligned}&\mathit{showBin} :: (\mathit{Rep}\ \alpha) \Rightarrow \alpha \to \mathit{Bin}\\ &\mathit{showBin} = \mathit{appShowBin}\ \mathit{rep}\end{aligned}$$

Loosely speaking, we apply the generic function to the type representation *rep*. Of course, this is not the whole story. The code above defines only a convenient shortcut. The actual definition of *showBin* is provided by an instance declaration, but you should read it instead as just a generic definition.

$$\begin{aligned}&\textbf{instance}\ \mathit{Generic}\ \mathit{ShowBin}\ \textbf{where}\\ &\quad \mathit{unit} &&= \mathit{ShowBin}\ (\lambda x \to [\,])\\ &\quad \mathit{plus} &&= \mathit{ShowBin}\ (\lambda x \to \textbf{case}\ x\ \textbf{of}\ \mathit{Inl}\ l \to \mathtt{0} : \mathit{showBin}\ l\\ &&&\qquad\qquad\qquad\qquad\quad \mathit{Inr}\ r \to \mathtt{1} : \mathit{showBin}\ r)\\ &\quad \mathit{pair} &&= \mathit{ShowBin}\ (\lambda x \to \mathit{showBin}\ (\mathit{outl}\ x) +\!\!+ \mathit{showBin}\ (\mathit{outr}\ x))\\ &\quad \mathit{datatype}\ \mathit{iso} &&= \mathit{ShowBin}\ (\lambda x \to \mathit{showBin}\ (\mathit{fromData}\ \mathit{iso}\ x))\\ &\quad \mathit{char} &&= \mathit{ShowBin}\ (\lambda x \to \mathit{bits}\ 7\ \ x)\\ &\quad \mathit{int} &&= \mathit{ShowBin}\ (\lambda x \to \mathit{bits}\ 16\ x)\end{aligned}$$

The class *Generic* has six member functions corresponding to the elementary types, *Unit*, *Plus*, and *Pair*, and to a small selection of primitive types, *Char* and *Int*. The member function *datatype*, which slightly breaks ranks, deals with arbitrary data types. Each method binding defines the instance of the generic function for the corresponding type. Let us consider each case in turn. To encode the single element of the type *Unit* no bits are required (*read:* the instance of *showBin* for the *Unit* type is $\lambda x \to [\,]$). To encode an element of a sum type, we emit one bit for the constructor followed by the encoding of its argument. The encoding of a pair is given by the concatenation of the component's encodings. To encode an element of an arbitrary data type, we first convert the element into a sum of products, which is then encoded. Finally, characters and integers are encoded using the function *bits*.

That's it, at least, as far as the generic function is concerned. Figure 1 summarizes

```
newtype Poly α = Poly{ appPoly :: π α }
poly :: (Rep α) ⇒ π α
poly = appPoly rep
instance Generic Poly where
  unit         = Poly (...)
  plus         = Poly (... poly ... poly ...)
  pair         = Poly (... poly ... poly ...)
  datatype iso = Poly (... (fromData iso) ... poly ... (toData iso) ...)
  char         = Poly (...)
  int          = Poly (...)
```

Fig. 1. A template for generic definitions on types.

the idioms you have to use for defining a generic function *poly* of type $\pi\ \alpha$, where α marks the generic part (the parts in ellipsis have to be filled in). Before we can actually compress data to strings of bits, we first have to turn the types of the to-be-compressed values into representable types, which is what we will do next.

Exercise 1. Implement a generic version of Haskell's comparison function *compare*:: $(Rep\ \alpha) \Rightarrow \alpha \to \alpha \to Ordering$. Follow the scheme above: first turn the signature into a **newtype** declaration, then define *compare*, and finally provide an instance of *Generic*. □

Exercise 2. Implement a function $readBin :: (Rep\ \alpha) \Rightarrow Bin \to \alpha$ that decodes a bit string that was encoded by *showBin*. □

2.2 Defining a new type

A generic function such as *showBin* can only be instantiated to a representable type. By default, only the elementary types, *Unit*, *Plus*, and *Pair*, and the primitive types *Char* and *Int* are representable. So, whenever we define a new data type and we intend to use a generic function on that type, we have to do a little bit of extra work. As an example, consider the data type of binary leaf trees.

```
data Tree α = Leaf α | Fork (Tree α) (Tree α)
```

We have to show that this type is representable. To this end we exhibit an isomorphic type built from representable type constructors. We call this type the *structure type* of *Tree*, denoted *Tree′*.

```
type Tree′ α = Plus α (Pair (Tree α) (Tree α))
```

The main work goes into defining two mappings, *fromTree* and *toTree*, which certify that $Tree\ \alpha$ and its structure type $Tree'\ \alpha$ are indeed isomorphic.

```
instance (Rep α) ⇒ Rep (Tree α) where
  rep = datatype (Iso fromTree toTree)
fromTree                :: Tree α → Tree′ α
fromTree (Leaf x)       = Inl x
fromTree (Fork l r)     = Inr (Pair l r)
toTree                  :: Tree′ α → Tree α
toTree (Inl x)          = Leaf x
toTree (Inr (Pair l r)) = Fork l r
```

Perhaps surprisingly, the structure type may contain the original type. This is valid and, in fact, the standard approach for recursive types since the original type becomes representable by virtue of the instance declaration.

Remark 1
Strictly speaking, the type *Tree* α and its structure type *Tree′* α are not isomorphic in Haskell since *Plus* is a lifted sum. This can be safely ignored if the trees on which the generic functions operate are always fully defined. We have to be careful, however, if the functions also deal with partial trees. In this case, the definition of *toTree* should be changed to

```
toTree (Inl x)              = Leaf x
toTree (Inr (∼(Pair l r))) = Fork l r
```

so that *Inr* $\perp$ is mapped to *Fork* $\perp$ $\perp$ rather than $\perp$. In general, we have to take care that the mapping *to* the data type is sufficiently lazy. □

As a second example, here is the encoding of Haskell's list data type.

```
type List′ α = Plus Unit (Pair α [α])
instance (Rep α) ⇒ Rep [α] where
  rep = datatype (Iso fromList toList)
fromList                :: [α] → List′ α
fromList []             = Inl Unit
fromList (x : xs)       = Inr (Pair x xs)
toList                  :: List′ α → [α]
toList (Inl Unit)       = []
toList (Inr (Pair x xs)) = x : xs
```

The *Unit* type is used for encoding constructors with no arguments. If a data type has more than two alternatives, or if a constructor has more than two arguments, we have to nest the binary type constructors *Plus* and *Pair* accordingly. Actually, we are more flexible than this: we can map the new type to any other type as long as the target type is an instance of *Rep*. For instance, we could map elements of some efficient sequence type (Okasaki, 1997) to lists. This flexibility is even necessary if the structure of a type is not available as in the case of *abstract types*. Figure 2 summarizes the per data type work.

Exercise 3. Turn the following types into instances of *Rep*.

```
type T' α1 ... αn = ...
instance (Rep α1, ..., Rep αn) ⇒ Rep (T α1 ... αn) where
  rep = datatype (Iso fromT toT)
fromT :: T α1 ... αn → T' α1 ... αn
fromT = ...
toT   :: T' α1 ... αn → T α1 ... αn
toT   = ...
```

Fig. 2. A template for making types representable.

```
data Shrub α β = Tip α | Node (Shrub α β) β (Shrub α β)
data Rose α    = Branch α [Rose α]
```

□

Exercise 4. Write a program that takes a data type definition and generates the Haskell source for the *Rep* instance. Use a tool such as DrIFT (2005) or the Template Haskell extension (Sheard & Peyton Jones, 2002) if you like. □

2.3 Implementing the shared library code

The implementation of light-weight generics is surprisingly concise: apart from declaring the two classes, *Generic* and *Rep*, we only provide a handful of instance declarations. To begin with, the class *Generic* accommodates the different instances of a generic function.

```
class Generic g where
  unit     :: g Unit
  plus     :: (Rep α, Rep β) ⇒ g (Plus α β)
  pair     :: (Rep α, Rep β) ⇒ g (Pair α β)
  datatype :: (Rep α) ⇒ Iso α β → g β
  char     :: g Char
  int      :: g Int
```

The class abstracts over the type constructor g, the type of a generic function. This is why *unit* has type $g\ Unit$. In the case of *Plus* and *Pair* the corresponding method has an additional context that constrains the type arguments of *Plus* and *Pair* to representable types. This context is necessary so that a generic function can recurse on the component types. In fact, the context allows us to call any generic function, so that we can easily define mutually recursive generic functions. We will see an example of this in the next section.

As a technical aside, in the introductory example in Section 2.1 the type constructor g was *ShowBin*. Since Haskell restricts class instances to data types, introduced by **data** or **newtype** declarations, the type signature of a generic function must be given by a **newtype** rather than a **type** definition.

Now, what does it mean for a type to be representable? For our purposes, this simply means that we can instantiate a generic function to that type. So an intriguing choice is to *identify* type representations with generic functions.

```
class Rep α where
  rep :: (Generic g) ⇒ g α
```

Note that the type variable g is implicitly universally quantified: the type representation must work for *all* types g that are instances of *Generic*. This is quite a strong requirement. How can we possibly define an instance of *Rep*? The answer lies in the type of *rep*: we have to use the methods of class *Generic*. Recall that *unit* has type $(Generic\ g) \Rightarrow g\ Unit$. Thus, we can turn *Unit* into an instance of *Rep*.

```
instance Rep Unit where
  rep = unit
instance (Rep α, Rep β) ⇒ Rep (Plus α β) where
  rep = plus
instance (Rep α, Rep β) ⇒ Rep (Pair α β) where
  rep = pair
instance Rep Char where
  rep = char
instance Rep Int where
  rep = int
```

Strange as the instance declarations may possibly seem, each of them has a logical explanation. A type is representable if we can instantiate a generic function to that type. But the class *Generic* just contains the instances of generic functions. Thus, each method of *Generic* with the notable exception of *datatype* gives rise to an instance declaration. We have seen in Section 2.2 that the method *datatype* is used to make an arbitrary type an instance of *Rep*. The procedure described in Section 2.2 is, in fact, dictated by *datatype*'s type $(Rep\ \alpha) \Rightarrow Iso\ \alpha\ \beta \to g\ \beta$: to make β representable we have to provide an isomorphic type α which in turn is representable.

The type of *rep*, namely, $(Rep\ \alpha, Generic\ g) \Rightarrow g\ \alpha$ is quite remarkable. In a sense, *rep* can be seen as the mother of all generic functions. This explains, in particular, the definition of *showBin* in Section 2.1: the field selector *appShowBin* has type $ShowBin\ \alpha \to (\alpha \to Bin)$; the application *appShowBin rep* implicitly instantiates *rep*'s type to $(Rep\ \alpha) \Rightarrow ShowBin\ \alpha$, which the field selector then turns to $(Rep\ \alpha) \Rightarrow \alpha \to Bin$. Note that the classes *Generic* and *Rep* are mutually recursive: each class lists the other one in a method context.

2.4 Extensions

2.4.1 Additional type cases

The class *Generic* can be seen as implementing a case analysis on types. Each method corresponds to a case branch. Types not listed as class methods are handled completely generically. However, this is not always what is wanted. As an example, recall that the encoding of a list of length n takes $n + 1$ bits plus the space for the encoding of the elements. A better method is to first encode the length of the list

and then to concatenate the encodings of the elements. In order to treat the list type as a separate case, we have to add a new method to the class *Generic*.

```
class Generic g where
  ...
  list :: (Rep α) ⇒ g [α]
  list = datatype (Iso fromList toList)
instance (Rep α) ⇒ Rep [α] where
  rep = list
```

So, the bad news is that we have to change a class definition, which suggests that *Generic* is not a good candidate for inclusion in a library (unless one can anticipate all future type cases). The good news is that by supplying a default definition for *list* this change does not affect any of the instance declarations: all the generic functions work exactly as before. In other words, the modification is a very local one but requires access to the source code. The new *ShowBin* instance overrides the default definition.

```
instance Generic ShowBin where
  ...
  list = ShowBin (λx → bits 16 (length x) ++ concatMap showBin x)
```

The technique relies on Haskell's concept of *default class methods*: only if the instance does not provide a binding for the *list* method, then the default class method is used.

Exercise 5. Adopt *readBin* to the new encoding of lists. □

2.4.2 A default type case

Using the same technique we can also implement a default or catch-all type case.

```
class Generic g where
  ...
  default :: (Rep α) ⇒ g α
  unit    = default
  plus    = default
  pair    = default
  char    = default
  int     = default
```

Now, the generic programmer can either define *unit*, *plus*, *pair*, *char*, *int* or simply *default* (in addition to *datatype*).[2] A default type case is useful for saying 'treat all the type cases not explicitly listed in the following way'. We will see an example application in Section 2.4.4.

[2] Unfortunately, if we specify all the type cases except *default*, we get a compiler warning saying that there is no explicit method nor default method for *default*.

2.4.3 Accessing constructor names

So far, the structure type captures solely the structure of a data type, hence its name. However, in Haskell there is more to a data type than this: a data constructor has a unique name, an arity, possibly a fixity, and possibly named fields. We are free to add all this information to the structure type. There are, in fact, several ways to accomplish this: we discuss one alternative in the sequel, Exercise 6 sketches a second one.

To record the properties of a data constructor we use the data type *Constr*—we confine ourselves to name and arity.

```
type Name      = String
type Arity     = Int
data Constr α = Constr{name :: Name,
                       arity :: Arity,
                       arg   :: α}
```

As an example, here is a suitable redefinition of *fromTree* and *toTree*.

```
type Tree' α = Plus (Constr α) (Constr (Pair (Tree α) (Tree α)))

fromTree               :: Tree α → Tree' α
fromTree (Leaf x)      = Inl (Constr "Leaf" 1 x)
fromTree (Fork l r)    = Inr (Constr "Fork" 2 (Pair l r))

toTree                                 :: Tree' α → Tree α
toTree (Inl (Constr n a x))            = Leaf x
toTree (Inr (Constr n a (Pair l r)))   = Fork l r
```

Note that, for reasons explained below, *toTree* simply discards the additional *Constr* wrapper. So strictly, the two functions do not define an isomorphism. This is not a problem, however, as long as we do not cheat with the constructor names (such as attaching *Constr* `"Leaf"` 1 to the representation of a *Fork* constructor).

It remains to introduce a new type case for constructors and to add *Constr* to the league of representable types.

```
class Generic g where
  ...
  constr             :: (Rep α) ⇒ g (Constr α)
  constr             = datatype (Iso arg wrap)
    where wrap a = Constr ⊥ ⊥ a
instance (Rep α) ⇒ Rep (Constr α) where
  rep                = constr
```

Note that *arg*, which is used in the default method for *constr*, is a field selector of the data type *Constr*. It is important that we have a default case for *constr* so that a generic function that does not require the additional information need not define a *constr* case. Since the helper function *wrap* necessarily adds undefined name and arity fields, the mapping *toTree* and colleagues have to ignore the decoration.

Figure 3 displays a simple pretty printer, based on Wadler's prettier printing

```
newtype Pretty α = Pretty{ appPretty :: α → Doc }
pretty :: (Rep α) ⇒ α → Doc
pretty = appPretty rep
instance Generic Pretty where
  unit    = Pretty (λx → empty)
  plus    = Pretty (λx → case x of Inl l  → pretty l
                                   Inr r → pretty r)
  pair    = Pretty (λx → pretty (outl x) ⟨⟩ line ⟨⟩ pretty (outr x))
  datatype iso
          = Pretty (λx → pretty (fromData iso x))
  char    = Pretty (λx → prettyChar x)
  int     = Pretty (λx → prettyInt x)
  list    = Pretty (λx → prettyl pretty x)
  constr  = Pretty (λx → let s = text (name x) in
                         if arity x == 0
                         then s
                         else  group (nest 1 (text "(" ⟨⟩ s ⟨⟩ line
                                              ⟨⟩ pretty (arg x) ⟨⟩ text ")")))
prettyl             :: (α → Doc) → ([α] → Doc)
prettyl p []        = text "[]"
prettyl p (a : as) = group (nest 1 (text "[" ⟨⟩ p a ⟨⟩ rest as))
  where rest []       = text "]"
        rest (x : xs) = text "," ⟨⟩ line ⟨⟩ p x ⟨⟩ rest xs
```

Fig. 3. A generic prettier printer

library (2003), that puts the additional information to good use. The *plus* case discards the constructors *Inl* and *Inr* as they are not needed for showing a value. The *constr* case signals the start of a constructed value. If the constructor is nullary, its string representation is emitted. Otherwise, the constructor name is printed followed by a space followed by the representation of its arguments. The *pair* case applies if a constructor has more than one component. In this case the components are separated by a space. Finally, *list* takes care of printing lists using standard list syntax: comma-separated elements between square brackets.

The approach above works well for pretty printing but, unfortunately, fails for parsing. The problem is that the constructor names are attached to a *value*. Consequently, this information is not available when parsing a string. The important point is that parsing *produces* (not consumes) a value, and yet it requires access to the constructor name. An alternative approach, discussed in the exercise below, is to attach the information to the *type* (more accurately, to the type representation).

Exercise 6. Augment the *datatype* method by an additional argument

$$datatype :: (Rep\ \alpha) \Rightarrow DataDescr \to Iso\ \alpha\ \beta \to g\ \beta$$

that records information about the data type and its constructors. Re-implement the pretty printer using this modification instead of the *constr* case. *Hint:* also extend *pretty* by a *DataDescr* argument. □

Exercise 7. Use the extension of the previous exercise and a parser library of your choice to implement a generic parser analogous to Haskell's *read* method. □

2.4.4 Mutual recursion

In Haskell, the *Show* class takes care of pretty printing. The class is very carefully crafted so that strings, which are lists of characters, are shown in double quotes, rather than between square brackets. It is instructive to re-program this behaviour as the new code requires all three extensions introduced above.

Basically, we have to implement a nested case analysis on types. The outer type case checks whether we have a list type; the inner type case checks whether the type argument of the list type constructor is *Char*. In our setting, a nested type case can be encoded using a pair of mutually recursive generic functions. The first realizes the outer type case.

```
instance Generic Pretty where
  ...
  list = Pretty (λx → prettyList x)
```

The instance declaration is the same as before, except that the *list* method dispatches to the second function which corresponds to the inner type case.

```
newtype PrettyList α = PrettyList{appPrettyList :: [α] → Doc}
prettyList :: (Rep α) ⇒ [α] → Doc
prettyList = appPrettyList rep
instance Generic PrettyList where
  char            = PrettyList (λx → prettyString x)
  datatype iso    = PrettyList (λx → prettyl prettyd x)
    where prettyd = pretty · fromData iso
  list            = default
  default         = PrettyList (λx → prettyl pretty x)
```

The *PrettyList* instance makes use of a default type case which implements the original behaviour (comma-separated elements between square brackets). The *datatype* method is similar to *default* except that the list elements are first converted to the structure type. Note that the *list* method must be explicitly set to *default* because it has the 'wrong' default class method: *datatype* (*Iso fromList toList*) instead of *default*. Finally, the *char* method takes care of printing strings in double quotes.

3 Generic functions on type constructors

Let us now turn to an alternative implementation of generics, which will increase flexibility at the cost of automation.

The generic functions introduced in the last section abstract over a type. For instance, *showBin* generalizes functions of type

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

to a single generic function of type

$$(Rep\ \alpha) \Rightarrow \alpha \rightarrow Bin$$

A generic function may also abstract over a *type constructor*. Take, as an example, a function that counts the number of elements contained in a data structure (a container). Such a function generalizes functions of type

$$[\alpha] \rightarrow Int, \quad Tree\ \alpha \rightarrow Int, \quad [Rose\ \alpha] \rightarrow Int$$

to a single generic function of type

$$(FRep\ \varphi) \Rightarrow \varphi\ \alpha \rightarrow Int$$

The class context makes explicit that counting elements does not work for arbitrary type constructors, but only for representable ones.

When type constructors come into play, typings often become ambiguous. Imagine applying a generic size function to a data structure of type $[Rose\ Int]$. Shall we count the number of rose trees in the list, or the number of integers in the list of rose trees? Because of this inherent ambiguity, the second implementation of generics will be more explicit about types and type representations.

This section is structured like the previous one: Section 3.1 introduces the format of generic definitions. Section 3.2 details the extra work for each newly defined data type, and Section 3.3 lists the shared library code. Finally, Section 3.4 takes a look at some extensions. Note that we shall re-use the class and method names even though the types of the class methods are slightly different.

3.1 Defining a generic function

Let us again start with a concrete example. Here is the implementation of a generic counter.

```
newtype Count α = Count{ appCount :: α → Int }
instance Generic Count where
  unit      = Count (λx → 0)
  plus a b  = Count (λx → case x of Inl l  → appCount a l
                                    Inr r → appCount b r)
  pair a b  = Count (λx → appCount a (outl x) + appCount b (outr x))
  datatype iso a
            = Count (λx → appCount a (fromData iso x))
  char      = Count (λx → 0)
  int       = Count (λx → 0)
```

The new version of the class *Generic* has the same member functions as before, but with slightly different typings: the cases corresponding to type constructors, *plus*, *pair* and *datatype*, now take explicit type arguments, a and b, which are passed to the recursive calls. Of course, we do not pass types as arguments, but rather type representations.

Though the class is a bit different, we are still able to define all the generic

functions we have seen before. In particular, we can apply *appCount* to *rep* to obtain a generic function of type $(Rep\ \alpha) \Rightarrow \alpha \to Int$. However, the result is not interesting at all: the function *appCount rep* always returns 0 (provided its argument is fully defined). Instead, we apply *appCount* to *frep*, the generic representation of a type constructor.

$$\begin{aligned} &size :: (FRep\ \varphi) \Rightarrow \varphi\ \alpha \to Int \\ &size = appCount\ (frep\ (Count\ (\lambda x \to 1))) \end{aligned}$$

Since *frep* represents a type constructor, it takes an additional argument, which specifies the action of *size* on the base type α: the function $\lambda x \to 1$ makes precise that each element of type α counts as 1. Interestingly, this is not the only option. If we pass the identity to *frep*, then we get a generic sum function.

$$\begin{aligned} &sum :: (FRep\ \varphi) \Rightarrow \varphi\ Int \to Int \\ &sum = appCount\ (frep\ (Count\ (\lambda x \to x))) \end{aligned}$$

Two generic functions for the price of one!

When *size* and *sum* are applied to some value, Haskell's type inferencer determines the particular instance of the type constructor φ. We have noted in the introduction that there are, in general, several possible alternatives for φ. If we are not happy with Haskell's choice, we can always specify the type explicitly (*list* is the representation of the list data type).

```
Main⟩ let xss = [[i * j | j ← [i..9]] | i ← [0..9]]
Main⟩ size xss
10
Main⟩ let a = Count (λx → 1)
Main⟩ appCount (list (list a)) xss
55
Main⟩ appCount (list a) xss
10
Main⟩ appCount a xss
1
```

By default, *size* calculates the size of the outer list, not the total number of elements. For the latter behaviour, we must pass an explicit type representation to *appCount*. This is something which is not possible with the first implementation of generics. Figure 4 summarizes the idioms for defining a generic function in the new style.

Exercise 8. Generalize *size* and *sum* so that they work for arbitrary numeric types.

$$\begin{aligned} &size :: (FRep\ \varphi, Num\ \eta) \Rightarrow \varphi\ \alpha \to \eta \\ &sum :: (FRep\ \varphi, Num\ \eta) \Rightarrow \varphi\ \eta \to \eta \end{aligned}$$

□

Exercise 9. The function *reducer* whose signature is given below generalizes Haskell's *foldr* function (*reducer* swaps the second and the third argument).

```
newtype Poly α = Poly{ appPoly :: π α}
instance Generic Poly where
  unit           = Poly (...)
  plus a b       = Poly (... (appPoly a) ... (appPoly b) ...)
  pair a b       = Poly (... (appPoly a) ... (appPoly b) ...)
  datatype iso a = Poly (... (fromData iso) ... (appPoly a) ... (toData iso) ...)
  char           = Poly (...)
  int            = Poly (...)
poly :: (FRep φ) ⇒ π (φ ...)
poly = appPoly (frep (Poly (...)))
```

Fig. 4. A template for generic definitions on type constructors.

```
newtype Reducer β α = Reducer{ appReducer :: α → β → β}
instance Generic (Reducer β)
reducer  :: (FRep φ) ⇒ (α → β → β) → (φ α → β → β)
reducer f = appReducer (frep (Reducer f))
```

Fill in the missing details. Use *reducer* to define a function that flattens a data structure into a list of elements. Define *sum* in terms of *reducer*. □

3.2 Introducing a new type

As before, we have to do a bit of extra work when we define a new data type. The main difference to Section 2.2 is that we must explicitly provide the structure type: the method *datatype* now expects the structure type as its second argument. At first sight, providing this information seems to be a lot less elegant, but it turns out to be fairly advantageous.

Reconsider the data type *Tree*. Since it is a type constructor rather than a type, we first define a 'type constructor representation'.

```
tree   :: (Generic g) ⇒ g α → g (Tree α)
tree a = datatype (Iso fromTree toTree) (a ⊕ tree a ⊗ tree a)
```

The operators '⊕' and '⊗' are convenient shortcuts for *plus* and *pair*.

```
infixr 3 ⊗
infixr 2 ⊕
a ⊕ b = plus a b
a ⊗ b = pair a b
```

The type constructor *Tree* can be seen as a function that takes types to types. Likewise, *tree* is a function that takes type representations to type representations. The structure type $a \oplus tree\ a \otimes tree\ a$ makes explicit, that *Tree* is a binary sum, that the first constructor takes a single argument of type α, and that the second constructor takes two arguments of type *Tree* α. Using *tree* we can now provide suitable instances of *Rep* and *FRep*.

```
type T' α1 ... αn = ...
t :: (Generic g) ⇒ g α1 → ··· → g αn → g (T α1 ... αn)
t a1 ... an = datatype (Iso fromT toT) (t' a1 ... an)
  -- here, t' a1 ... an is the representation of the structure type T' α1 ... αn
fromT :: T α1 ... αn → T' α1 ... αn
fromT = ...
toT   :: T' α1 ... αn → T α1 ... αn
toT   = ...
instance (Rep α1, ..., Rep αn) ⇒ Rep (T α1 ... αn) where
  rep = t rep ... rep   -- n copies of rep
  -- if T has at least one type argument:
instance (Rep α1, ..., Rep αn−1) ⇒ FRep (T a1 ... αn−1) where
  frep = t rep ... rep   -- n − 1 copies of rep
```

Fig. 5. A template for making types representable (second approach).

```
instance (Rep α) ⇒ Rep (Tree α) where
  rep = tree rep
instance FRep Tree where
  frep = tree
```

The last declaration shows that *tree* is just the *Tree* instance of *frep*. Figure 5 summarizes the per data type work.

3.3 Implementing the shared library code

The implementation of *Generic* and *Rep* reflects the change from implicit to explicit type arguments: the implicit arguments in the form of a context '(*Rep* α) $\Rightarrow$' are replaced by explicit arguments of the form '$g\ \alpha \rightarrow$'.

```
class Generic g where
  unit     :: g Unit
  plus     :: g α → g β → g (Plus α β)
  pair     :: g α → g β → g (Pair α β)
  datatype :: Iso α β → g α → g β
  char     :: g Char
  int      :: g Int
class Rep α where
  rep :: (Generic g) ⇒ g α
instance Rep Unit where
  rep = unit
instance (Rep α, Rep β) ⇒ Rep (Plus α β) where
  rep = rep ⊕ rep
instance (Rep α, Rep β) ⇒ Rep (Pair α β) where
  rep = rep ⊗ rep
```

```
instance Rep Char where
  rep = char
instance Rep Int where
  rep = int
```

Furthermore, we introduce a class that accommodates the mother of all 'type constructor representations'.

```
class FRep φ where
  frep :: (Generic g) ⇒ g α → g (φ α)
```

The class *Rep* abstracts over a type of kind $\star$, *FRep* abstracts over a type of kind $\star \to \star$. In general, we need a class Rep_κ for each kind of interest, see also Exercise 15.

Exercise 10. The first implementation of generics used implicit, the second explicit type arguments. Does it make sense to combine both?

```
class Generic g where
  unit :: g Unit
  plus :: (Rep α, Rep β) ⇒ g α → g β → g (Plus α β)
  ...
```

Hint: given this interface can you define a truly polymorphic function of type $(FRep\ \varphi) \Rightarrow \varphi\ \alpha \to Int$? □

3.4 Extensions

3.4.1 Accessing constructor names

Passing type representations explicitly pays off when it comes to adding information about constructors. In Section 2.4.3 we had to introduce a new type *Constr* to record the name and the arity of the constructor, and we had to change the representation of elements accordingly. Now, we can simply add the information to the type representation.

```
class Generic g where
  ...
  constr :: Name → Arity → g α → g α
```

Since the additional type case *constr name arity* has type $g\ \alpha \to g\ \alpha$, the representation of values is not affected. This is a huge advantage as it means that this extension works both for pretty printing and parsing.

In particular, it suffices to adapt the definition of *tree* and colleagues; the implementation of the mappings *fromTree* and *toTree* is not affected.

```
tree    :: (Generic g) ⇒ g α → g (Tree α)
tree a = datatype (Iso fromTree toTree)
                  (constr "Leaf" 1 a ⊕ constr "Fork" 2 (tree a ⊗ tree a))
```

The new definition of *tree* is a true transliteration of the data type declaration.

3.4.2 Mutual recursion

Being explicit about type representations is awkward when it comes to programming mutually recursive generic functions. With the first implementation mutual recursion was easy: the method context '$(Rep\ \alpha) \Rightarrow$' allowed us to call any generic function. Now, we are less flexible: the explicit $g\ \alpha$ argument corresponds to the immediate recursive call. So, to implement mutual recursion we have to tuple the functions involved.

$$\begin{aligned}&\textbf{newtype}\ Pretty\ \alpha = Pretty\{\ appPretty\ \ \ \ \ \ \ \ :: \alpha \to Doc,\\ &\qquad\qquad\qquad\qquad\qquad\quad appPrettyList :: [\alpha] \to Doc\}\end{aligned}$$

The following exercise asks you to re-implement the prettier printer using this record type.

Exercise 11. Re-implement the generic prettier printer of Section 2.4.4 using tupling. Try, in particular, to simulate default type cases. □

4 Abstracting over multiple type arguments

The next and final generalization, while simple to implement, is not entirely obvious: we allow the signature of a generic function to abstract over multiple type arguments. This extension is pointless for generic functions on types, but useful for generic functions on type constructors as it adds an extra degree of flexibility. Again, we shall re-use the class and method names of the previous sections even though the types are different.

4.1 Defining a generic function

Many list-processing functions can be made generic so that they work for arbitrary data types. An important example is the function *map* which applies a given function to every element of a given list:

$$\begin{aligned}&map &&:: (\alpha_1 \to \alpha_2) \to ([\alpha_1] \to [\alpha_2])\\ &map\ f\ [\,] &&= [\,]\\ &map\ f\ (x : xs) &&= f\ x : map\ f\ xs\end{aligned}$$

As a characteristic feature *map* does not change the structure of the list; only the elements of the list are modified. The list data type is the most prominent example of a *container type*. It is not hard to see that mapping functions make sense for arbitrary container types. In general, the mapping function for an n-ary container type (containing elements of n different types) takes n argument functions and applies them to the elements of the appropriate types leaving the structure of the container unchanged:

$$\begin{aligned}&Int \to Int\\ &(\alpha_1 \to \alpha_2) \to (Tree\ \alpha_1 \to Tree\ \alpha_2)\\ &(\alpha_1 \to \alpha_2) \to (\beta_1 \to \beta_2) \to (Shrub\ \alpha_1\ \beta_1 \to Shrub\ \alpha_2\ \beta_2)\end{aligned}$$

Type constructors with no arguments, ie types, are an extreme case: since the mapping function takes no arguments, it has type $T \to T$. In fact, the mapping

function for types boils down to the identity as it is not supposed to change the structure of the 'container'. We already know from Section 3.1 that generic functions sometimes have trivial instances on types: the generic counter, for instance, is the constant 0 function in this case. Nonetheless the extreme case is important as it suggests a type signature for the generic mapping function:

$$\textbf{newtype}\ \mathit{Map}\ \alpha = \mathit{Map}\{\,\mathit{appMap} :: \alpha \to \alpha\,\}$$

If we apply *appMap* to *rep*, we obtain a function of type $(\mathit{Rep}\ \alpha) \Rightarrow \alpha \to \alpha$ as desired. Applying *appMap* to *frep* yields

$$\begin{aligned}&\mathit{fmap}' \quad :: (\mathit{FRep}\ \varphi) \Rightarrow (\alpha \to \alpha) \to (\varphi\ \alpha \to \varphi\ \alpha)\\ &\mathit{fmap}'\ f = \mathit{appMap}\ (\mathit{frep}\ f)\end{aligned}$$

which is almost what we want: *fmap′* takes as a first argument a function of type $\alpha \to \alpha$ whereas the original *map* takes a function of type $\alpha_1 \to \alpha_2$. Fortunately, this problem is easy to remedy: we merely have to extend the type signature by a second type argument:

$$\textbf{newtype}\ \mathit{Map}\ \alpha_1\ \alpha_2 = \mathit{Map}\{\,\mathit{appMap} :: \alpha_1 \to \alpha_2\,\}$$

The type *Map* is now isomorphic to the function type constructor '→', so we could use '→' directly. However, for clarity, we shall stick to the more verbose type. The generic definition of the mapping function is mostly straightforward.

$$\begin{aligned}&\textbf{instance}\ \mathit{Generic}\ \mathit{Map}\ \textbf{where}\\ &\quad \mathit{unit} \quad\ = \mathit{Map}\ (\lambda x \to x)\\ &\quad \mathit{plus}\ a\ b = \mathit{Map}\ (\lambda x \to \textbf{case}\ x\ \textbf{of}\ \mathit{Inl}\ l\ \to \mathit{Inl}\ (\mathit{appMap}\ a\ l)\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{Inr}\ r \to \mathit{Inr}\ (\mathit{appMap}\ b\ r))\\ &\quad \mathit{pair}\ a\ b = \mathit{Map}\ (\lambda x \to \mathit{Pair}\ (\mathit{appMap}\ a\ (\mathit{outl}\ x))\ (\mathit{appMap}\ b\ (\mathit{outr}\ x)))\\ &\quad \mathit{datatype}\ \mathit{iso}_1\ \mathit{iso}_2\ a\\ &\qquad\qquad = \mathit{Map}\ (\lambda x \to \mathit{toData}\ \mathit{iso}_2\ (\mathit{appMap}\ a\ (\mathit{fromData}\ \mathit{iso}_1\ x)))\\ &\quad \mathit{char} \quad\ = \mathit{Map}\ (\lambda x \to x)\\ &\quad \mathit{int} \quad\ = \mathit{Map}\ (\lambda x \to x)\end{aligned}$$

The mapping function on types, *Unit*, *Char* and *Int*, is the identity; on binary types, *Plus* and *Pair*, it takes two argument functions and applies them to the components of the appropriate types. We postpone a discussion of *datatype* until the next section.

The specialization of the generic mapping function to unary type constructors is then given by

$$\begin{aligned}&\mathit{fmap}' \quad :: (\mathit{FRep}\ \varphi) \Rightarrow (\alpha_1 \to \alpha_2) \to (\varphi\ \alpha_1 \to \varphi\ \alpha_2)\\ &\mathit{fmap}'\ f = \mathit{appMap}\ (\mathit{frep}\ (\mathit{Map}\ f))\end{aligned}$$

The type of *fmap′* is as expected because *frep* now has the more general type $(\mathit{Generic}\ g) \Rightarrow g\ \alpha_1\ \alpha_2 \to g\ (\varphi\ \alpha_1)\ (\varphi\ \alpha_2)$, see Section 4.3.

Exercise 12. Implement a generic version of the monadic mapping function:

$$\begin{aligned}&\textbf{newtype}\ \mathit{MapM}\ \mu\ \alpha_1\ \alpha_2 = \mathit{MapM}\{\,\mathit{appMapM} :: \alpha_1 \to \mu\ \alpha_2\,\}\\ &\textbf{instance}\ (\mathit{Monad}\ \mu) \Rightarrow \mathit{Generic}\ (\mathit{MapM}\ \mu)\end{aligned}$$

□

4.2 Introducing a new type

Making new types representable works almost exactly as before except that *datatype* now takes two *iso* arguments: given isomorphisms iso_1 and iso_2 of types $Iso\ \alpha_1\ \beta_1$ and $Iso\ \alpha_2\ \beta_2$ the method $datatype\ iso_1\ iso_2$ has type $g\ \alpha_1\ \alpha_2 \to g\ \beta_1\ \beta_2$. It allows us to turn a generic function of type $g\ \alpha_1\ \alpha_2$ into a function of type $g\ \beta_1\ \beta_2$ provided α_1 and β_1 are isomorphic and α_2 and β_2. This explains the definition of the mapping function for the *datatype* case in the previous section: since *Map* is essentially '$\to$', we have to turn a function f of type $\alpha_1 \to \alpha_2$ into a function of type $\beta_1 \to \beta_2$. The composition $toData\ iso_2 \cdot f \cdot fromData\ iso_1$ does the job.

As an example, here is the type constructor representation for the data type *Tree*.

$$
\begin{aligned}
&tree &&:: (Generic\ g) \Rightarrow g\ \alpha_1\ \alpha_2 \to g\ (Tree\ \alpha_1)\ (Tree\ \alpha_2)\\
&tree\ a &&= datatype\ isoTree\ isoTree\ (a \oplus tree\ a \otimes tree\ a)\\
&isoTree &&:: Iso\ (Tree'\ \alpha)\ (Tree\ \alpha)\\
&isoTree &&= Iso\ fromTree\ toTree
\end{aligned}
$$

At first sight, it seems that we pass two copies of *isoTree* to *datatype*, but a closer inspection reveals that they are two different instances of the same polymorphic value.

4.3 Implementing the shared library code

The *Generic* class must be adapted to abstract over a binary type constructor g.

$$
\begin{aligned}
&\textbf{class}\ Generic\ g\ \textbf{where}\\
&\quad unit &&:: g\ Unit\ Unit\\
&\quad plus &&:: g\ \alpha_1\ \alpha_2 \to g\ \beta_1\ \beta_2 \to g\ (Plus\ \alpha_1\ \beta_1)\ (Plus\ \alpha_2\ \beta_2)\\
&\quad pair &&:: g\ \alpha_1\ \alpha_2 \to g\ \beta_1\ \beta_2 \to g\ (Pair\ \alpha_1\ \beta_1)\ (Pair\ \alpha_2\ \beta_2)\\
&\quad datatype &&:: Iso\ \alpha_1\ \beta_1 \to Iso\ \alpha_2\ \beta_2 \to g\ \alpha_1\ \alpha_2 \to g\ \beta_1\ \beta_2\\
&\quad char &&:: g\ Char\ Char\\
&\quad int &&:: g\ Int\ Int
\end{aligned}
$$

The mother of all generic functions, *rep*, instantiates g to two copies of the representable type:

$$
\begin{aligned}
&\textbf{class}\ Rep\ \alpha\ \textbf{where}\\
&\quad rep :: (Generic\ g) \Rightarrow g\ \alpha\ \alpha
\end{aligned}
$$

Though the class definition has changed, the instance declarations are exactly as in Section 3.3.

The *FRep* class is, however, more general than before—this was the purpose of the whole exercise.

$$
\begin{aligned}
&\textbf{class}\ FRep\ \varphi\ \textbf{where}\\
&\quad frep :: (Generic\ g) \Rightarrow g\ \alpha_1\ \alpha_2 \to g\ (\varphi\ \alpha_1)\ (\varphi\ \alpha_2)
\end{aligned}
$$

The new version of *Generic* strictly generalizes the development in Section 3 as every one-argument type signature can be rewritten as a two-argument signature that simply ignores the second argument. For example,

$$\textbf{newtype}\ \mathit{Count}\ \alpha_1\ \alpha_2 = \mathit{Count}\{\,\mathit{genericCount} :: \alpha_1 \to \mathit{Int}\,\}$$

So, at least in principle, there is no necessity to have both the one-argument and the two-argument version of *Generic*.

Exercise 13. Can you think of a generic function that is parameterized by more than two type arguments? □

Exercise 14. How would you implement

$$\mathit{apply} :: (\mathit{FRep}\ \varphi) \Rightarrow \varphi\ (\alpha \to \beta) \to (\varphi\ \alpha \to \varphi\ \beta)$$

which applies a structure of functions to a structure of arguments? We require both structures to have the same shape. *Hint:* solve the previous exercise first. □

Exercise 15. Generalize *Rep* and *FRep* to a family Rep_κ of classes indexed by the kind κ of its type argument: $\mathit{Rep} = \mathit{Rep}_\star$ and $\mathit{FRep} = \mathit{Rep}_{\star\to\star}$. *Hint:* use a kind-indexed type (Hinze, 2002). □

4.4 Example: generic ordering

Exercise 1 asked for a generic version of Haskell's comparison function suggesting the following type signature

$$\textbf{newtype}\ \mathit{Cmp}\ \alpha = \mathit{Cmp}\{\,\mathit{appCmp} :: \alpha \to \alpha \to \mathit{Ordering}\,\}$$

Intuitively, the two elements whose ordering is determined must be of the same type. Perhaps surprisingly, we obtain a more flexible variant if we abstract over two type arguments.

$$\textbf{newtype}\ \mathit{Cmp}\ \alpha_1\ \alpha_2 = \mathit{Cmp}\{\,\mathit{appCmp} :: \alpha_1 \to \alpha_2 \to \mathit{Ordering}\,\}$$

Given this type the implementation of the generic *compare* is fairly straightforward:

$$\begin{aligned}
&\textbf{instance}\ \mathit{Generic}\ \mathit{Cmp}\ \textbf{where}\\
&\quad \mathit{unit} \quad\ = \mathit{Cmp}\ (\lambda x_1\ x_2 \to \mathit{EQ})\\
&\quad \mathit{plus}\ a\ b = \mathit{Cmp}\ (\lambda x_1\ x_2 \to \textbf{case}\ (x_1, x_2)\ \textbf{of}\\
&\qquad\qquad\qquad\qquad (\mathit{Inl}\ a_1, \mathit{Inl}\ a_2) \to \mathit{appCmp}\ a\ a_1\ a_2\\
&\qquad\qquad\qquad\qquad (\mathit{Inl}\ a_1, \mathit{Inr}\ b_2) \to \mathit{LT}\\
&\qquad\qquad\qquad\qquad (\mathit{Inr}\ b_1, \mathit{Inl}\ a_2) \to \mathit{GT}\\
&\qquad\qquad\qquad\qquad (\mathit{Inr}\ b_1, \mathit{Inr}\ b_2) \to \mathit{appCmp}\ b\ b_1\ b_2)\\
&\quad \mathit{pair}\ a\ b = \mathit{Cmp}\ (\lambda x_1\ x_2 \to \textbf{case}\ \mathit{appCmp}\ a\ (\mathit{outl}\ x_1)\ (\mathit{outl}\ x_2)\ \textbf{of}\\
&\qquad\qquad\qquad\qquad \mathit{LT} \to \mathit{LT}\\
&\qquad\qquad\qquad\qquad \mathit{EQ} \to \mathit{appCmp}\ b\ (\mathit{outr}\ x_1)\ (\mathit{outr}\ x_2)\\
&\qquad\qquad\qquad\qquad \mathit{GT} \to \mathit{GT})\\
&\quad \mathit{datatype}\ \mathit{iso}_1\ \mathit{iso}_2\ a\\
&\qquad\qquad = \mathit{Cmp}\ (\lambda x_1\ x_2 \to \mathit{appCmp}\ a\ (\mathit{fromData}\ \mathit{iso}_1\ x_1)\ (\mathit{fromData}\ \mathit{iso}_2\ x_2))\\
&\quad \mathit{char} \quad\ = \mathit{Cmp}\ (\lambda x_1\ x_2 \to \mathit{compare}\ x_1\ x_2)\\
&\quad \mathit{int} \quad\ = \mathit{Cmp}\ (\lambda x_1\ x_2 \to \mathit{compare}\ x_1\ x_2)
\end{aligned}$$

Applying *appCmp* to *rep* yields the function requested in Exercise 1.

$$cmp :: (Rep\ \alpha) \Rightarrow \alpha \to \alpha \to Ordering$$
$$cmp = appCmp\ rep$$

The extra generality shows up if we lift *cmp* to type constructors:

$$fcmp :: (FRep\ \varphi) \Rightarrow (\alpha_1 \to \alpha_2 \to Ordering) \to (\varphi\ \alpha_1 \to \varphi\ \alpha_2 \to Ordering)$$
$$fcmp\ rel = appCmp\ (frep\ (Cmp\ rel))$$

The call *fcmp rel* x_1 x_2 checks whether corresponding elements in the structures x_1 and x_2 are related by *rel*. Of course, *rel* need not be the usual lexicographic ordering; it may even relate elements of different types. It is worth noting that the above implementation with the exception of *datatype* works for both the one-argument and the two-argument version of *Cmp*. In a sense, the more general type is also more natural; the first type artificially constrains the applicability of the code.

5 Background: type representations

We have seen three implementations of generics. So far we have motivated and explained the different approaches mainly from the perspective of a potential user. It is high time to look behind the scenes. In this section we shall highlight the theoretical background deriving the first two implementations from first principles.

A generic function such as *showBin* can be seen as a function that is parameterized by a type and proceeds by case analysis on the type. Of course, Haskell 98 like most other languages neither allows the programmer to pass types nor to analyze them. However, what we can do is to pass and analyze *representations* of types.

As a first try, we could invent a type *Rep* and assign *showBin* the type $Rep \to \alpha \to Bin$. A moment's reflection, however, reveals that this won't work. The parametricity theorem (Wadler, 1989) implies that a function of this type must necessarily ignore its second argument. The trick described in (Cheney & Hinze, 2002) is to use a parametric type for type representations:[3]

$$showBin :: \forall\alpha\,.\,Rep\ \alpha \to \alpha \to Bin$$

The idea is that an element of $Rep\ \tau$ is the *unique* representation of τ. Interestingly, we can define *Rep* in Haskell using a recent extension called *generalized algebraic data types* (Hinze, 2003; Peyton Jones *et al.*, 2004):

$$\textbf{data}\ Rep :: \star \to \star\ \textbf{where}$$
$$\quad Int\ \ :: Rep\ Int$$
$$\quad Pair :: \forall\alpha\ \beta\,.\,Rep\ \alpha \to Rep\ \beta \to Rep\ (\alpha, \beta)$$

The declaration introduces the type constructor *Rep* and two data constructors *Int* and *Pair*. For brevity, we shall use this stripped-down version of *Rep* that comprises only one primitive and one elementary type as a running example. Note that *Rep* is not an ordinary parameterized data type since the result types of *Int* and *Pair* are not of the form *Rep* α.

[3] From now on we shall be explicit about universal quantification. In particular, as we shall use polymorphic functions of higher ranks and local universal quantification.

Using *Rep* we can easily implement a generic version of *showBin*.

$$\begin{aligned}
&\mathit{showBin} && :: \forall\tau\,.\,\mathit{Rep}\ \tau \to \tau \to \mathit{Bin}\\
&\mathit{showBin}\ (\mathit{Int}) \quad i && = \mathit{bits}\ 16\ i\\
&\mathit{showBin}\ (\mathit{Pair}\ a\ b)\ (x, y) && = \mathit{showBin}\ a\ x +\!\!+ \mathit{showBin}\ b\ y
\end{aligned}$$

Since a type is represented by a value, the type case boils down to an ordinary case, which is a good thing because we can use all the conveniences of pattern matching such as default cases or nested patterns.

It is important to note, however, that the case analysis is unusual in that each branch has a different type: the first equation instantiates the type of *showBin* to *Int*, the second to (α, β). This is why generalized algebraic data types are a non-trivial extension of Haskell 98. Since we want to do without any extensions, we have to encode *Rep* somehow.

5.1 Background: encodings of data types

Before we proceed let us briefly review representations of data types. The best known scheme for representing data types in System F (Girard, 1972) was discovered independently by Leivant (1983) and Böhm and Berarducci (1985). In this scheme the recursive type $T \cong F\ T$ is represented by the space of polymorphic functions $\forall\tau\,.\,(F\ \tau \to \tau) \to \tau$. Consider as a simple example the unary representation of the natural numbers.

$$\begin{aligned}
&\textbf{data}\ \mathit{Nat} :: \star\ \textbf{where}\\
&\quad \mathit{Zero} :: \mathit{Nat}\\
&\quad \mathit{Succ} :: \mathit{Nat} \to \mathit{Nat}
\end{aligned}$$

We have $\mathit{Nat} \cong F\ \mathit{Nat}$ where $F\ \alpha = 1 + \alpha$. Using the laws of exponentials, in particular, $1 \to C \cong C$ and $(A + B) \to C \cong (A \to C) \times (B \to C)$, we can slightly simplify the encoding: $F\ \tau \to \tau = (1 + \tau) \to \tau \cong (1 \to \tau) \times (\tau \to \tau) \cong \tau \times (\tau \to \tau)$. Thus, we obtain the following definition of *Nat*.

$$\begin{aligned}
&\textbf{newtype}\ \mathit{Nat} && = \mathit{Nat}\{\mathit{fold} :: \forall \mathit{nat}\,.\,\mathit{Algebra}\ \mathit{nat} \to \mathit{nat}\}\\
&\textbf{data}\ \mathit{Algebra}\ \mathit{nat} && = \mathit{With}\{\mathit{foldZero} :: \mathit{nat},\\
&&&\qquad\quad \mathit{foldSucc} :: \mathit{nat} \to \mathit{nat}\}
\end{aligned}$$

The helper data type *Algebra*, which implements $F\ \tau \to \tau$, is pretty much a transliteration of the original **data** declaration, except that *Nat* has been replaced by *nat*. Interestingly, an element of *Nat* can be seen as a *fold* or *catamorphism* (Hinze, 2005): it evaluates the natural number it represents using a given algebra. As an aside, since $\tau \times (\tau \to \tau) \to \tau \cong (\tau \to \tau) \to (\tau \to \tau)$, the above definition of *Nat* is isomorphic to the type of Church numerals, which is why this scheme is often called *Church encoding*.

The constructors *Zero* and *Succ* are represented by

$$\begin{aligned}
&\mathit{zero} && = \mathit{Nat}\ (\lambda a \to \mathit{foldZero}\ a)\\
&\mathit{succ}\ n && = \mathit{Nat}\ (\lambda a \to \mathit{foldSucc}\ a\ (\mathit{fold}\ n\ a))
\end{aligned}$$

Read the definitions as folds or catamorphisms: the successor function, for instance,

first evaluates its argument (*fold n a*) and then applies the appropriate component of the algebra to the result (*foldSucc a*). Here is the definition of addition using this encoding:

$$\begin{aligned}&(+) :: \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat}\\ &m + n = \mathit{fold}\ m\ \mathit{With}\{\mathit{foldZero} = n,\\ &\qquad\qquad\qquad\qquad \mathit{foldSucc} = \mathit{succ}\}\end{aligned}$$

A second scheme for representing data types is due to Parigot (1992). Here the recursive type $T \cong F\ T$ is represented by the type $U = \forall\tau\,.\,(F\ U \to \tau) \to \tau$, which is recursive, as well. The Parigot encoding of the natural numbers is very similar to the Church encoding:

$$\begin{aligned}&\mathbf{newtype}\ \mathit{Nat}\ = \mathit{Nat}\{\,\mathit{case} :: \forall \mathit{nat}\,.\,\mathit{Case}\ \mathit{nat} \to \mathit{nat}\,\}\\ &\mathbf{data}\ \mathit{Case}\ \mathit{nat} = \mathit{Of}\{\,\mathit{caseZero} :: \mathit{nat},\\ &\qquad\qquad\qquad\qquad \mathit{caseSucc} :: \mathit{Nat} \to \mathit{nat}\,\}\quad \text{-- } \mathbf{NB}.\ \mathit{Nat} \to \mathit{nat} \text{ instead}\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{-- of } \mathit{nat} \to \mathit{nat}\end{aligned}$$

Again, note that the helper type *Case*, which implements $F\ U \to \tau$, is almost a transliteration of the original **data** declaration, except that now only the occurrences of *Nat* in the result types of the constructors are replaced by *nat*. The major difference to the Church encoding is that an element of *Nat* now implements a **case**-*analysis* rather than a *fold*. Consequently, the representations of *Zero* and *Succ* simply select a case branch.

$$\begin{aligned}&\mathit{zero}\quad = \mathit{Nat}\ (\lambda c \to \mathit{caseZero}\ c)\\ &\mathit{succ}\ n = \mathit{Nat}\ (\lambda c \to \mathit{caseSucc}\ c\ n)\end{aligned}$$

The definition of addition is more or less a transliteration of the usual recursive definition.

$$\begin{aligned}&(+) :: \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat}\\ &m + n = \mathit{case}\ m\ \mathit{Of}\{\,\mathit{caseZero} = n,\\ &\qquad\qquad\qquad\qquad \mathit{caseSucc} = \lambda m' \to \mathit{succ}\ (m' + n)\,\}\end{aligned}$$

An advantage of the Parigot encoding is that the predecessor can be computed in constant time: $\mathit{pred}\ n = \mathit{case}\ n\ \mathit{Of}\{\,\mathit{caseZero} = \mathit{zero}, \mathit{caseSucc} = \mathit{id}\,\}$. On the other hand, it requires System F to be extended by recursive types.

5.2 Parigot encoding of Rep

Now, let us apply the encodings to the type *Rep* of type representations. It may come as a surprise that we can actually do this as *Rep* is a *generalized algebraic data type*, one that is not given as the fixed point of some type constructor F. The point is that F is not needed to make this work: we can directly transliterate the **data** declaration of *Rep*, which defines a higher-order algebra, into an *Algebra* or *Case* record.

Here is the Parigot encoding of *Rep* (we start with the Parigot encoding as this one is easier to understand):

```
newtype Rep τ = Rep{ case :: ∀rep . Case rep → rep τ }
data Case rep  = Of{ caseInt  :: rep Int,
                     casePair :: ∀α β . Rep α → Rep β → rep (α, β) }
int       :: Rep Int
int       = Rep (λc → caseInt c)
pair      :: ∀α β . Rep α → Rep β → Rep (α, β)
pair a b  = Rep (λc → casePair c a b)
```

Typical of the Parigot encoding, the recursive arguments of the constructors have type *Rep* rather than *rep*. Furthermore, *Rep* and *Case* are defined by mutual recursion.

Here is the *showBin* function adapted to the Parigot encoding.

```
newtype ShowBin α = ShowBin{ appShowBin :: α → Bin }
showBin'   :: ∀τ . Rep τ → ShowBin τ
showBin' t = case t Of{ caseInt  = ShowBin (λi → bits 16 i),
                        casePair = λa b → ShowBin (λ(x, y) →
                                     showBin a x ++ showBin b y) }
showBin    :: ∀τ . Rep τ → (τ → Bin)
showBin t  = appShowBin (showBin' t)
```

We have successfully eliminated the generalized algebraic data type. However, the program above is still not legal Haskell 98 since it uses local universal quantification: *Case* is a record with a polymorphic component. Worse still, *Rep* τ is a polymorphic function that takes a polymorphic record as an argument. On top of this, a generic function such as *showBin* takes an argument of type *Rep* τ.

How can we possibly encode this tower of polymorphic functions in Haskell 98? The essential clue to solving this puzzle is to recall that Haskell has two different kinds of records: data types with a single constructor and type classes. A type class declaration introduces a record type, each instance declaration defines a record of that type. Though type classes are second-class citizens, they have two distinct advantages over ordinary records: they are created and passed implicitly and they may contain polymorphic components.

The first feature is a convenience: an element of *Rep* τ represents a type, so what could be more natural to use the class system to automatically create and pass elements of type *Rep* τ, in particular, as the creation is dictated by type.

The second feature is essential for the transition to Haskell 98. It is well-known that the translation of type classes to System F, the so-called *dictionary translation*, requires records with polymorphic components. Here we employ this fact to encode polymorphic records. Since both *Rep* and *Case* contain polymorphic functions, we turn both types into type classes.

```
class Rep τ where
  case      :: (Case rep) ⇒ rep τ
class Case rep where
  caseInt   :: rep Int
  casePair  :: ∀α β . (Rep α, Rep β) ⇒ rep (α, β)
```

Of course, *Rep* and *Case* are still mutually recursive. The encodings of *Int* and *Pair* are now given by two instance declarations.

```
instance Rep Int where
  case = caseInt
instance (Rep α, Rep β) ⇒ Rep (α, β) where
  case = casePair
```

The method definitions are unusually short (SPJ: "Ralf stopped writing.") since all the arguments are now passed implicitly. For completeness, here is the code of *showBin* using type classes.

```
instance Case ShowBin where
  caseInt   = ShowBin (λi → bits 16 i)
  casePair = ShowBin (λ(x, y) → showBin x ++ showBin y)
showBin :: ∀τ . (Rep τ) ⇒ τ → Bin
showBin = appShowBin case
```

It's also much shorter than the previous version because again most of the arguments are passed implicitly.

If we compare the code above to the definitions in Section 2, we find that we have, modulo naming of classes and methods, derived the first implementation of generics.

5.3 Church encoding of Rep

Here is the Church encoding of *Rep*:

```
data Rep τ        = Rep{fold :: ∀rep . Algebra rep → rep τ}
data Algebra rep = With{foldInt   :: rep Int,
                        foldPair :: ∀α β . rep α → rep β → rep (α, β)}
int       :: Rep Int
int       = Rep (λc → foldInt c)
pair      :: ∀α β . Rep α → Rep β → Rep (α, β)
pair a b = Rep (λc → foldPair c (fold a c) (fold b c))
```

As we already know, the difference to the Parigot encoding is subtle: the two types are no longer mutually recursive, since the arguments of the constructors have type *rep* rather than *Rep*. Furthermore, *pair* now recursively evaluates its arguments. Because *pair* already does most of the hard work, *foldPair* in the code below only needs to apply its arguments to the components of the pair: a and b are no longer type representations, but already appropriate instances of $showBin'$.

```
showBin'   :: ∀τ . Rep τ → ShowBin τ
showBin' t = fold t With{foldInt   = ShowBin (λi → bits 16 i),
                         foldPair  = λa b → ShowBin (λ(x, y) →
                                       appShowBin a x ++ appShowBin b y)}
showBin    :: ∀τ . Rep τ → (τ → Bin)
showBin t  = appShowBin (showBin' t)
```

Since *Algebra* is the algebra of types (more accurately, of type representations), *showBin′* can be seen as a *type fold*.

As in the previous section, we can use type classes to turn the above code into a legal Haskell 98 program.

```
class Rep τ where
  fold     :: (Algebra rep) ⇒ rep τ
class Algebra rep where
  foldInt  :: rep Int
  foldPair :: ∀α β . rep α → rep β → rep (α, β)
int        = foldInt
infixr 3 ⊗
a ⊗ b      = foldPair a b
instance Rep Int where
  fold     = int
instance (Rep α, Rep β) ⇒ Rep (α, β) where
  fold     = fold ⊗ fold
```

Adapting *showBin* to the class-based version is largely a matter of routine.

```
instance Algebra ShowBin where
  foldInt      = ShowBin (λi → bits 16 i)
  foldPair a b = ShowBin (λ(x, y) → appShowBin a x ++ appShowBin b y)
showBin :: ∀τ . (Rep τ) ⇒ τ → Bin
showBin = appShowBin fold
```

All in all, we have derived the second implementation of generics.

One of the pros of the second variant, discussed in Section 3, is that it allows us to define generic functions on type constructors. This feature is, in fact, achieved via a neat trick. The first thing to note is that the generic functions still analyse types, not type constructors (we could have introduced a generalized algebraic data type to represent type constructors, but we didn't). The basic idea is that a type constructor of kind $\star \to \star$, a function on types, can be represented by an *open type*, a type that contains a single free type variable. Assume, for the sake of example, that we want to encode solely the structure of a container ignoring its elements. Using informal syntax, this could be implemented as follows (the type constructor is $\Lambda a \,.\, int \otimes a$):

```
showBin (int ⊗ a) where showBin (a) = λx → [ ]
```

The free type variable a marks the positions of the elements that are ignored; the action of *showBin* on the free type variable a is defined in the **where** clause.

Though the original *Rep* type only admits closed terms, the class-based variant is more flexible: the informal syntax above corresponds to the legal Haskell 98 fragment below

$$\mathit{appShowBin}\ (\mathit{int} \otimes a)\ \mathbf{where}\ a = \mathit{ShowBin}\ (\lambda x \to [\,])$$

The reason why this works is that Haskell's class system silently adds code: the type of a causes *int* and '$\otimes$' to be instantiated to *ShowBin*; consequently both are passed the dictionary of the *ShowBin* instance of *Algebra*. In other words, the occurrence of *int* in the expression above is not the polymorphic value but rather a suitable *ShowBin* instance. That said, it is clear that we could, in principle, mimic this behaviour in the record-based variant of the Church encoding, but it would be quite inconvenient to do so.

6 Stock taking

We have presented three implementations of generics. The first one in Section 2 is slightly easier to use (mutually recursive definitions are straightforward) but more restricted (generic functions on type constructors are not supported). The second one in Section 3 is very flexible (supports generic functions on both types and type constructors) but slightly more inconvenient to use (mutual recursion requires tupling). The third implementation in Section 4 supersedes the second by generalizing the signature of generic functions to abstract over multiple type arguments.

The two main approaches only differ in the way type representations are passed around: the first implementation, the Parigot encoding, passes them implicitly via *Rep* α contexts; the second, the Church encoding, passes them explicitly as arguments of type $g\ \alpha$. Being explicit has one further advantage besides greater expressiveness: we can change the representation of types without changing the representation of the underlying values. This is very useful for adding information about data constructors.

The class-based implementation of generics is surprisingly expressive: we can define all the generic functions presented, for instance, in (Hinze, 2002). It has, however, also its limitations. Using a single *Generic* class we can only define functions that abstract over a fixed number of type arguments. In principle, we need one separate class for each arity. In practice, a single class that abstracts over *two* arguments might be sufficient: type signatures that abstract over one argument only can be rewritten as two-argument signatures that ignore the second parameter; generic functions that need abstraction over three or more parameters are quite rare (but see Exercises 13 and 14). Using several different *Generic* classes has the unfortunate consequence that there isn't a single type representation, which is awkward for implementing dynamic values. Finally, none of the approaches can define generic functions that involve *generic types* (Hinze *et al.*, 2004), types that are defined by induction on the structure of types.

The particular implementation described in this paper is inspired by Weirich's paper (2003). Weirich gives an implementation in Haskell augmented by rank-2 types. The essence of this paper is that Haskell's class system can be used to avoid higher-order ranks.

There is yet another encoding of type representations, described in (Cheney & Hinze, 2002). The idea is to simulate the original *Rep* type, which is a generalized algebraic data type, using equality constraints. To illustrate the idea, the type signature

$$Pair :: Rep\ \alpha \to Rep\ \beta \to Rep\ (\alpha, \beta)$$

can be rewritten as

$$Pair :: (\tau == (\alpha, \beta)) \Rightarrow Rep\ \alpha \to Rep\ \beta \to Rep\ \tau$$

A witness for the equality constraint is then passed to each occurrence of *Pair*, either implicitly or explicitly. The constraint can subsequently be used to convert an element of τ into a pair, or vice versa.

7 Further reading and related work

Have you got interested in generic programming? There is a wealth of material on the subject. For a start, we recommend studying the tutorials (Backhouse *et al.*, 1999; Hinze & Jeuring, 2003b; Hinze & Jeuring, 2003a). Further reading includes (Jansson & Jeuring, 1997; Hinze, 2000). Let us now take a closer look at related work.

Generic type classes Haskell's major innovation was its support for ad-hoc overloading in the form of type classes. Type classes bear a strong resemblance to generic definitions: A type class declaration corresponds to the type signature of a generic definition—or rather, to a collection of type signatures. An instance declaration is related to a type case of a generic definition. The crucial difference to generic programming is that an instance declaration must be written by hand for each newly defined data type, whereas a generic definition automatically works for all (representable) types. We have mentioned in the introduction that Haskell provides special support for a handful of built-in classes: by attaching a **deriving** clause to a **data** declaration, the Haskell compiler is instructed to generate the 'obvious' code for these classes. What 'obvious' means is specified informally in an Appendix of the language definition (Peyton Jones, 2003). Of course, the idea suggests itself to use generic definitions for specifying *default methods* so that the programmer can define her own derivable classes. This extension is detailed in (Hinze & Peyton Jones, 2001) and partially supported by the Glasgow Haskell Compiler (The GHC Team, 2005). A similar, but more expressive variant of *generic type classes* is implemented in Clean (Alimarine & Plasmeijer, 2001). The overall programming style is very similar (modulo syntax) to what we have seen here. A distinct advantage over our proposal is that the per data type code is generated automatically. Furthermore, extra type cases are easily handled by providing additional instance

declarations. On the other hand, some extensions do not seem to fit well into the class framework: for instance, to provide access to the names of constructors, values must be embedded in types. Since neither Haskell nor GHC's internal language support this, access to constructor names is currently not supported. With *type representations* this is not an issue: a type representation is a value, which can be easily augmented by additional data.

PolyP The Haskell extension PolyP (Jansson & Jeuring, 1997) was one of the first attempts to produce a generic programming language. It is simpler and less powerful than the approach described here as it is restricted to generic functions that abstract over *regular* type constructors of kind $\star \to \star$. The original implementation of PolyP is set up as a preprocessor that translates PolyP code into Haskell. A later version (Norell & Jansson, 2003) embeds PolyP program into Haskell augmented by multiple parameter type classes with functional dependencies (Jones, 2000). A disadvantage of the latter approach is that many type classes propagate into the types of generic functions. For instance, the generic counter has type

$$psum :: (FunctorOf\ f\ \ d, P\_fmap2\ f, P\_fsum\ f) \Rightarrow d\ Int \to Int$$

An advantage of PolyP is that it can define various recursion operators such as *cata-* or *anamorphisms* (Meijer *et al.*, 1991). This is not possible here as it requires a different representation of types. However, Oliveira and Gibbons (2005) show how to adopt our approach to PolyP resulting in a purely Haskell 98 implementation, called Light PolyP. In Light PolyP the types of generic functions are much closer to what one would expect without loosing any expressive power.

Generic Haskell Generic Haskell (Löh, 2004; Löh & Jeuring, 2005) is a full-fledged implementation of generics based on ideas by Hinze (2002; 2004) that features generic functions, generic types and various extensions such as default cases and constructor cases (Clarke & Löh, 2002). Generic Haskell supports the definition of functions that work for all types of all kinds, such as, for example, a generalized mapping function. Default cases and constructor cases allow the generic programmer to refine the behaviour of a generic functions for some specific data types (additional type cases) or even for some specific data constructors. Generic Haskell like PolyP is a preprocessor that translates generic programs into Haskell 98 augmented by rank-n types. It works by program specialization: for every data type, a generic function is applied to, Generic Haskell generates a tailor-made instance. Since the type case analysis is performed at compile-time, the resulting code is more efficient.

Intensional type analysis Closely related to generic programming is the work on *intensional type analysis* (Harper & Morrisett, 1995; Crary *et al.*, 1998; Crary & Weirich, 1999; Trifonov *et al.*, 2000; Weirich, 2001). Intensional type analysis is used in typed intermediate languages in compilers for polymorphic languages, among others to be able to optimise code for polymorphic functions. Loosely speaking, intensional type analysis relates to generic programming in the same way the Parigot

encoding relates to the Church encoding: intensional type analysis centers around type case, while generic programming deals with type catamorphisms.

Scrap your boilerplate A different approach to generic programming, called 'scrap your boilerplate' henceforth SYB, was developed by Peyton Jones and Lämmel in a series of papers (2003; 2004; 2005). Originally, the approach was an implementation of *strategic programming* (Visser, 2000) in Haskell and was then extended to cover more generic grounds. Briefly, strategic programming is an idiom for processing and querying complex, compound data such as terms or object structures. The SYP approach is essentially combinator-based: the user writes generic functions by combining a few generic primitives. This is one of its strengths (generic traversals can often be written succinctly and perspicuously), but also its main weakness: the definition of more complex functions (for instance, a function that traverses several structures simultaneously) requires a considerable level of sophistication. On a more principal note, it is not clear, whether the set of predefined combinators is sufficient to define all generic functions of interest. Indeed, each new paper introduces a few additional combinators. The approach is restricted to generic functions on types, generic functions on type constructors such as *map* or *size* are out of reach. The implementation relies in an essential way on rank-2 polymorphism and various other extensions (such as type-safe cast and recursive dictionaries), so that the approach is not suitable for Haskell 98. Furthermore, it requires additional compiler support for instantiating the primitives to every data type of interest. With respect to efficiency, SYB seems to be advantageous, since the generic functions work directly on the original data, whereas our approach requires a mediating data structure, the representation type. On the other hand, SYB makes heavy use of higher-order functions and potentially costly run-time type tests, which may outweigh the savings. The SYB approach can be simulated to some extent in our framework if one is willing to go beyond Haskell 98, see (Hinze, 2003). In general, using *rank-2 types* we can implement *higher-order generic functions*.

Acknowledgements

I am grateful to Andres Löh, Jeremy Gibbons, Simon Peyton Jones, Bruno Oliveira, Fermin Reig, Stephanie Weirich, and the anonymous referees of ICFP 2004 and of this special issue for pointing out several typos and for valuable suggestions regarding grammar and, in particular, presentation.

References

Alimarine, A. and Plasmeijer, R. (2001) A generic programming extension for Clean. Arts, T. and Mohnen, M. (eds), *Proceedings of the 13th International workshop on the Implementation of Functional Languages, IFL'01* pp. 257–278.

Backhouse, R., Jansson, P., Jeuring, J. and Meertens, L. (1999) Generic Programming — An Introduction —. Swierstra, S. D., Henriques, P. R. and Oliveira, J. N. (eds), *3rd International Summer School on Advanced Functional Programming, Braga, Portugal.* Lecture Notes in Computer Science 1608, pp. 28–115. Springer-Verlag.

Böhm, C. and Berarducci, A. (1985) Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science* **39**(2-3):135–154.

Cheney, J. and Hinze, R. (2002) A lightweight implementation of generics and dynamics. Chakravarty, M. M. (ed), *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop* pp. 90–104. ACM Press.

Clarke, D. and Löh, A. (2002) Generic Haskell, specifically. Gibbons, J. and Jeuring, J. (eds), *Proceedings of the IFIP TC2 Working Conference on Generic Programming, Schloss Dagstuhl* pp. 21–48. Kluwer Academic Publishers.

Crary, K. and Weirich, S. (1999) Flexible type analysis. *Proceedings ICFP 1999: International Conference on Functional Programming* pp. 233–248. ACM Press.

Crary, K., Weirich, S. and Morrisett, J. G. (1998) Intensional polymorphism in type-erasure semantics. *Proceedings ICFP 1998: International Conference on Functional Programming* pp. 301–312. ACM Press.

DrIFT. (2005) *DrIFT Home Page.* `http://repetae.net/john/computer/haskell/DrIFT/`.

Girard, J.-Y. (1972) *Interprétation foncionnelle et élimination des coupures de l'arithmétique d'order supérieur.* PhD thesis, Université de Paris VII.

Harper, R. and Morrisett, G. (1995) Compiling polymorphism using intensional type analysis. *22nd Symposium on Principles of Programming Languages, POPL '95* pp. 130–141.

Hinze, R. (2000) A new approach to generic functional programming. 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* pp. 119–132.

Hinze, R. (2002) Polytypic values possess polykinded types. *Science of Computer Programming* **43**:129–159.

Hinze, R. (2003) Fun with phantom types. Gibbons, J. and de Moor, O. (eds), *The Fun of Programming*, pp. 245–262. Palgrave Macmillan. ISBN 1-4039-0772-2 hardback, ISBN 0-333-99285-7 paperback.

Hinze, R. (2005) Theoretical Pearl: Church numerals, twice! *J. Functional Programming* **15**(1):1–13.

Hinze, R. and Jeuring, J. (2003a) Generic Haskell: Applications. Backhouse, R. and Gibbons, J. (eds), *Generic Programming: Advanced Lectures.* Lecture Notes in Computer Science 2793, pp. 57–97. Springer-Verlag.

Hinze, R. and Jeuring, J. (2003b) Generic Haskell: Practice and theory. Backhouse, R. and Gibbons, J. (eds), *Generic Programming: Advanced Lectures.* Lecture Notes in Computer Science 2793, pp. 1–56. Springer-Verlag.

Hinze, R. and Peyton Jones, S. (2001) Derivable type classes. Hutton, G. (ed), *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, vol. 41.1 of Electronic Notes in Theoretical Computer Science. Elsevier Science. The preliminary proceedings appeared as a University of Nottingham technical report.

Hinze, R., Jeuring, J. and Löh, A. (2004) Type-indexed data types. *Science of Computer Programming* **51**:117–151.

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

Jones, M. P. (2000) Type classes with functional dependencies. Smolka, G. (ed), *Proceedings of the 9th European Symposium on Programming, ESOP 2000, Berlin, Germany.* Lecture Notes in Computer Science 1782, pp. 230–244. Springer-Verlag.

Lämmel, R. and Peyton Jones, S. (2004) Scrap more boilerplate: reflection, zips, and

generalised casts. Fisher, K. (ed), *Proceedings of the 2004 International Conference on Functional Programming, Snowbird, Utah, September 19–22, 2004* pp. 244–255.

Lämmel, R. and Peyton Jones, S. (2005) Scrap your boilerplate with class: extensible generic functions. Pierce, B. (ed), *Proceedings of the 2005 International Conference on Functional Programming, Tallinn, Estonia, September 26–28, 2005*.

Leivant, D. (1983) Reasoning about functional programs and complexity classes associated with type disciplines. *Proceedings 24th Annual IEEE Symposium on Foundations of Computer Science, FOCS'83, Tucson, AZ, USA* pp. 460–469. IEEE Computer Society Press.

Löh, A. (2004) *Exploring Generic Haskell*. PhD thesis, Utrecht University.

Löh, A. and Jeuring, J. (2005) *The Generic Haskell user's guide, Version 1.42 - Coral release*. Tech. rept. UU-CS-2005-004. Universiteit Utrecht.

Meijer, E., Fokkinga, M. and Paterson, R. (1991) Functional programming with bananas, lenses, envelopes and barbed wire. *5th ACM Conference on Functional Programming Languages and Computer Architecture, FPCA'91, Cambridge, MA, USA*. Lecture Notes in Computer Science 523, pp. 124–144. Springer-Verlag.

Norell, U. and Jansson, P. (2003) Polytypic programming in Haskell. Trinder, P., Michaelson, G. and Peña, R. (eds), *Implementation of Functional Languages: 15th International Workshop, IFL 2003, Edinburgh, UK, September 8-11, 2003* pp. 168–184.

Okasaki, C. (1997) Catenable double-ended queues. *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming* pp. 66–74. *ACM SIGPLAN Notices,* 32(8), August 1997.

Oliveira, B. C. and Gibbons, J. (2005) *TypeCase: A design pattern for type-indexed functions*. Submitted for publication.

Parigot, M. (1992) Recursive programming with proofs. *Theoretical Computer Science* **94**(2):335–356.

Peyton Jones, S. (2003) *Haskell 98 Language and Libraries*. Cambridge University Press.

Peyton Jones, S. and Lämmel, R. (2003) Scrap your boilerplate: a practical approach to generic programming. *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003), New Orleans* pp. 26–37.

Peyton Jones, S., Washburn, G. and Weirich, S. (2004) *Wobbly types: type inference for generalised algebraic data types*. Submitted for publication.

Sheard, T. and Peyton Jones, S. (2002) Template metaprogramming for Haskell. Chakravarty, M. M. T. (ed), *ACM SIGPLAN Haskell Workshop 02* pp. 1–16. ACM Press.

The GHC Team. (2005) *The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.4*. Available from `http://www.haskell.org/ghc/documentation.html`.

Trifonov, V., Saha, B. and Shao, Z. (2000) Fully reflexive intensional type analysis. *Proceedings ICFP 2000: International Conference on Functional Programming* pp. 82–93. ACM Press.

Visser, E. (2000) Language independent traversals for program transformation. Jeuring, J. (ed), *Proceedings of the 2nd Workshop on Generic Programming, Ponte de Lima, Portugal* pp. 86–104. The proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-2000-19.

Wadler, P. (1989) Theorems for free! *The Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA'89), London, UK* pp. 347–359. Addison-Wesley Publishing Company.

Wadler, P. (2003) A prettier printer. Gibbons, J. and de Moor, O. (eds), *The Fun of*

Programming. Cornerstones of Computing, pp. 223–243. Palgrave Macmillan Publishers Ltd.

Weirich, S. (2001) Encoding intensional type analysis. *European Symposium on Programming*. LNCS 2028, pp. 92–106. Springer-Verlag.

Weirich, S. (2003) *Higher-Order Intensional Type Analysis in Type-Erasure Semantics*. available from
`http://www.cis.upenn.edu/~sweirich/papers/erasure/erasure-paper-july03.pdf`.