Generics for the Masses

Ralf Hinze
Institut für Informatik III
Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
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 pearl 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.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages

Keywords

Generic programming, type classes, Haskell 98

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 [11] 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 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 [9, 7, 2] that support exactly this, 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 pearl 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 two major ones and a few variations—consists roughly of two dozen lines of Haskell code. The reader is cordially invited to play with the material. The source code can be found at

```
http://www.ralf-hinze.de/masses.tar.bz2
```

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

2. GENERIC FUNCTIONS ON TYPES

This section discusses the first implementation of generics. Sections 2.1 and 2.2 introduce the approach from a user's perspective, Section 2.3 details the implementation, and Sec-

[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.

tion 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.

```
type Bin = [Bit]
data Bit  = 0 | 1 deriving (Show)
bits      :: (Enum α) ⇒ Int → α → Bin
```

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, 0, 0,
 1, 1, 0, 1, 0, 0, 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. Fortunately, generic programming comes to the rescue. 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.

```
data Unit     = Unit
data Plus α β = Inl α | Inr β
data Pair α β = Pair{ outl :: α, outr :: β}
```

Why these 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. When programming a generic function it suffices to know the two mappings that witness the isomorphism.

```
data Iso α β = Iso{fromData :: β → α, toData :: α → β}
```

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.

```
newtype ShowBin α = ShowBin{ appShowBin :: α → Bin}
```

An element of *ShowBin* σ is an instance of *showBin* that encodes values of type σ as bit strings. We know that *showBin* itself cannot be a genuine polymorphic function of type $\alpha \to$ *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 generic compression function is then given by the following simple, yet slightly mysterious definition.

```
showBin :: (Rep α) ⇒ α → Bin
showBin = appShowBin rep
```

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.

```
instance Generic ShowBin where
  unit = ShowBin (λx → [ ])
  plus = ShowBin (λx → case x of Inl l  → 0 : showBin l
                                 Inr r → 1 : showBin r)
  pair = ShowBin (λx → showBin (outl x)
                       ++ showBin (outr x))
  datatype iso
       = ShowBin (λx → showBin (fromData iso x))
  char = ShowBin (λx → bits 7 x)
  int  = ShowBin (λx → bits 16 x)
```

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. 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* α) $\Rightarrow \alpha \to \alpha \to$ *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* α) $\Rightarrow$ *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*.

```
instance (Rep α) ⇒ Rep (Tree α) where
  rep = datatype (Iso fromTree toTree)
```

The main work goes into defining two mappings, *fromTree* and *toTree*, which certify that *Tree* α and its structure type *Plus* α (*Pair* (*Tree* α) (*Tree* α)) are indeed isomorphic.[2]

```
fromTree :: Tree α → Plus α (Pair (Tree α) (Tree α))
fromTree (Leaf x)       = Inl x
fromTree (Fork l r)     = Inr (Pair l r)

toTree    :: Plus α (Pair (Tree α) (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. Oh, wonders of recursion!

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

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

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

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

2.3 Implementation

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
```

[2]Strictly speaking, the type *Tree* α and its structure type *Plus* α (*Pair* (*Tree* α) (*Tree* α)) are not isomorphic in Haskell since *Plus* is a lifted sum. We simply ignore this complication here.

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.

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* instances of g. This is quite a strong requirement. How can we possibly define an instance of g? The answer is simple, yet mind-boggling: 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 seem, each 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 the type of *datatype*: we have to provide an isomorphic data type which in turn is representable.

The type of *rep*, namely, (*Rep* α, *Generic* g) $\Rightarrow$ g α is quite remarkable. In a sense, *rep* can be seen as the mother of all generic functions. This de-mystifies the definition of *showBin* in Section 2.1: the application *appShowBin rep* implicitly instantiates *rep*'s type to (*Rep* α) $\Rightarrow$ *ShowBin* α, which the field selector *appShowBin* subsequently turns to (*Rep* α) $\Rightarrow$ $\alpha \rightarrow$ *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 into a library). 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. 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 4. 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*).[3] 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.

2.4.3 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 5 sketches a second one.

[3]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*.

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 of efficiency, *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.

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 "" (−1) 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*.

Figure 1 displays a simple pretty printer, based on Wadler's prettier printing library [13], 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* (well, to the type representation).

```
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
```

Figure 1: A generic prettier printer

Exercise 5. 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. □

Exercise 6. 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 the second implementation of generics, which will increase flexibility at the cost of automation. Note that we re-use the class and method names even though the types of the class methods are slightly different.

3.1 Defining a generic function

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

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

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. Such a function generalizes functions of type

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

to a single generic function of type

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

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. The following implementation of a generic counter illustrates the point.

```
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 $(Rep\ \alpha) \Rightarrow \alpha \rightarrow Int$. However, the result is not interesting at all: the function always returns 0 (provided its argument is fully defined). Instead, we apply *appCount* to *frep*, the generic representation of a type constructor.

```
size :: (FRep φ) ⇒ φ α → Int
size = appCount (frep (Count (λx → 1)))
```

Since *frep* represents a type constructor, it takes an additional argument, which specifies the action of *size* on the base type α: the function $\lambda x \rightarrow 1$ 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.

```
sum :: (FRep φ) ⇒ φ Int → Int
sum = appCount (frep (Count (λx → x)))
```

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 before 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.

```
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.

Exercise 7. Generalize *size* and *sum* so that they work for arbitrary numeric types.

```
size :: (FRep φ, Num η) ⇒ φ α → η
sum :: (FRep φ, Num η) ⇒ φ η → η
```
□

Exercise 8. The function *reducer* whose signature is given below generalizes Haskell's *foldr* function (*reducer* swaps the second and the third argument).

```
newtype Reducer β α
          = Reducer{ appReducer :: α → β → β}
instance Generic (Reducer β) where
  . . .
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 define 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*.

```
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*.

3.3 Implementation

The implementation of *Generic* and *Rep* reflects the change from implicit to explicit type arguments: the implicit arguments in the form of a context '$(Rep\ \alpha) \Rightarrow$' 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)
```

Exercise 9. 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)
  ...                                                  □
```

Exercise 10. Some generic functions require abstraction over two type parameters.

```
class Generic g where
  unit :: g Unit Unit
  plus :: g α₁ α₂ → g β₁ β₂ → g (Plus α₁ β₁) (Plus α₂ β₂)
  ...
class Rep α where
  rep  :: (Generic g) ⇒ g α α
```

Implement a generic mapping function using this interface (generalizing Haskell's *fmap*). □

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. 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 $g\ \alpha \rightarrow g\ \alpha$, 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 a bit of a pain 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.

```
newtype Pretty α = Pretty{ appPretty     :: α → Doc,
                           appPrettyList :: [α] → Doc}
```

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. STOCK TAKING

We have presented two 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 difficult to use (mutual recursion requires tupling).

The two approaches only differ in the way type representations are passed around: the first implementation passes them implicitly via $Rep\ \alpha$ contexts, the second 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 jolly 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 [4]. It has, however, also its limitations. Using the class *Generic* we can only define functions that abstract over one type parameter. In order to implement functions that abstract over two type arguments such as *map*, we need an additional type class (see Exercise 10). In fact, we need one separate class for each arity. This has the unfortunate consequence that there isn't a single type representation, which is awkward for implementing dynamic values. Furthermore, we cannot define generic functions that involve *generic types* [8], types that are defined by induction on the structure of types.

5. WHERE TO GO FROM HERE

Got interested in generic programming? There is quite a wealth of material on the subject. For a start, we recommend studying the tutorials [1, 7, 6]. Further reading includes [9, 3].

The particular implementation described in this pearl is inspired by Weirich's paper [14]. Weirich gives an implementation in Haskell augmented by rank-2 types. The essence of this pearl is that Haskell's class system can be used to avoid higher-order ranks.

If you are willing to go beyond Haskell 98, then there is a lot more to discover. Using *rank-2 types* we can implement *higher-order generic functions*. This extension is vital for implementing generic traversals [12, 5, 10]. Using *existential types* we can combine generic functions with dynamic values [2, 5]. Dynamic type checking is indispensable for programs that interact with the environment.

Acknowledgements

I am grateful to Jeremy Gibbons, Fermin Reig, Stephanie Weirich, and the anonymous referees for pointing out several typos and for valuable suggestions regarding grammar and presentation.

6. REFERENCES

[1] Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic Programming — An Introduction —. In S. Doaitse Swierstra, Pedro R. Henriques, and Jose N. Oliveira, editors, *3rd International Summer School on Advanced Functional Programming, Braga, Portugal*, volume 1608 of *Lecture Notes in Computer Science*, pages 28–115. Springer-Verlag, Berlin, 1999.

[2] James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In Manuel M.T. Chakravarty, editor, *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop*, pages 90–104. ACM Press, October 2002.

[3] Ralf Hinze. A new approach to generic functional programming. In Thomas W. Reps, editor, *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00), Boston, Massachusetts, January 19-21*, pages 119–132, January 2000.

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

[5] Ralf Hinze. Fun with phantom types. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, pages 245–262. Palgrave Macmillan, 2003. ISBN 1-4039-0772-2 hardback, ISBN 0-333-99285-7 paperback.

[6] Ralf Hinze and Johan Jeuring. Generic Haskell: Applications. In Roland Backhouse and Jeremy Gibbons, editors, *Generic Programming: Advanced Lectures*, volume 2793 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[7] Ralf Hinze and Johan Jeuring. Generic Haskell: Practice and theory. In Roland Backhouse and Jeremy Gibbons, editors, *Generic Programming: Advanced Lectures*, volume 2793 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

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

[9] Patrik Jansson and Johan Jeuring. PolyP—a polytypic programming language extension. In *Conference Record 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 470–482. ACM Press, January 1997.

[10] Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In Kathleen Fisher, editor, *Proceedings of the 2004 International Conference on Functional Programming, Snowbird, Utah, September 19–22, 2004*, September 2004.

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

[12] Simon Peyton Jones and Ralf Lämmel. Scrap your boilerplate: a practical approach to generic programming. In *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003), New Orleans*, January 2003.

[13] Philip Wadler. A prettier printer. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, Cornerstones of Computing, pages 223–243. Palgrave Macmillan Publishers Ltd, March 2003.

[14] Stephanie Weirich. Higher-order intensional type analysis in type-erasure semantics. Available from `http://www.cis.upenn.edu/~sweirich/papers/erasure/erasure-paper-july03.pdf`, 2003.