Chapter 1

Extensible and Modular Generics for the Masses

Bruno C. d. S. Oliveira[1], Ralf Hinze[2], Andres Löh[2]

Abstract: A *generic function* is a function that is defined on the structure of data types: with a single definition, we obtain a function that works for many data types. In contrast, an *ad-hoc polymorphic* function requires a separate implementation for each data type. Previous work by Hinze on *lightweight generic programming* has introduced techniques that allow the definition of generic functions directly in Haskell. A severe drawback of these approaches is that generic functions, once defined, cannot be extended with ad-hoc behaviour for new data types, precluding the design of an extensible and modular generic programming

library based on these techniques. In this paper, we present a revised version of Hinze's *Generics for the masses* approach that overcomes this limitation. Using our new technique, writing an extensible and modular generic programming library in Haskell 98 is possible.

1.1 INTRODUCTION

A *generic*, or *polytypic*, function is a function that is defined over the structure of types: with a single definition, we obtain a function that works for many data types. Standard examples include the functions that can be derived automatically in Haskell [14], such as *show*, *read*, and '==', but there are many more.

By contrast, an *ad-hoc polymorphic* function [15] requires a separate implementation for each data type. In Haskell, we implement ad-hoc polymorphic functions using type classes. Here is an example, a binary encoder:

---

[1]Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, UK, bruno@comlab.ox.ac.uk

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

```
class Encode t where
  encode :: t → [Bit]

instance Encode Char where
  encode = encodeChar

instance Encode Int where
  encode = encodeInt

instance Encode a ⇒ Encode [a] where
  encode []     = [0]
  encode (x:xs) = 1 : (encode x ++ encode xs)
```

The **class** construct introduces an overloaded function with a type parameter t, and the **instance** statements provide implementations for a number of specific types. An instance for lists of type $[a]$ can only be given if an instance for a exists already. The function *encode* thus works on characters, integers, and lists, and on data types that are built from these types. If we call *encode*, the compiler figures out the correct implementation to use, or, if no suitable instance exists, reports a type error.

We assume that primitive bit encoders for integers and characters are provided from somewhere. Lists are encoded by replacing an occurrence of the empty list [ ]

with the bit 0, and occurrences of the list constructor (:) with the bit 1 followed by the encoding of the head element and the encoding of the remaining list.

The following example session demonstrates the use of *encode* on a list of strings (where strings are lists of characters in Haskell).

```
Main⟩ encode ["xy","x"]
[1,1,0,0,0,1,1,1,1,1,1,0,0,1,1,1,1,0,1,1,0,0,0,1,1,1,1,0,0]
```

The function *encode* can be extended at any time to work on additional data types. All we have to do is write another instance of the *Encode* class. However, each time we add a new data type and we want to encode values of that data type, we need to supply a specific implementation of encode for it.

In "Generics for the Masses" (GM) [4] a particularly lightweight approach to generic programming is presented. Using the techniques described in that paper we can write generic functions directly in Haskell 98. This contrasts with other approaches to generic programming, which usually require significant compiler support or language extensions.

In Figure 1.1, we present a generic binary encoder implemented using the GM technique. We will describe the technical details, such as the shape of class *Generic*, in Section 1.2. Let us, for now, focus on the comparison with the ad-hoc polymorphic function given above. The different methods of class *Generic* define

different cases of the generic function. For characters and integers, we assume again standard definitions. But the case for lists is now subsumed by three generic cases for unit, sum and product types. By viewing all data types in a uniform way, these three cases are sufficient to call the encoder on lists, tuples, trees, and several more complex data structures – a new instance declaration is not required.

2

```
newtype Encode a = Encode {encode' :: a → [Bit]}
instance Generic Encode where
  unit       = Encode (const [])
  plus a b   = Encode (λx → case x of Inl l  → 0 : encode' a l
                                      Inr r → 1 : encode' b r)
  prod a b   = Encode (λ(x × y) → encode' a x ++ encode' b y)
  char       = Encode encodeChar
  int        = Encode encodeInt
  view iso a = Encode (λx → encode' a (from iso x))
```

FIGURE 1.1. A generic binary encoder

However, there are situations in which a specific case for a specific data type – called an *ad-hoc case* – is desirable. For example, lists can be encoded more efficiently than shown above: instead of encoding each constructor, we can encode the length of the list followed by encodings of the elements. Or, suppose that sets are represented as trees: The same set can be represented by multiple trees, so a generic equality function should not compare sets structurally, and therefore we need an ad-hoc case for sets.

Defining ad-hoc cases for ad-hoc polymorphic functions is trivial: we just add an **instance** declaration with the desired implementation. For the generic version of the binary encoder, the addition of a new case is, however, very difficult. Each case of the function definition is implemented a method of class *Generic*, and adding a new case later requires the modification of the class. We say that generic functions written in this style are not *extensible*, and that the GM approach is not *modular*, because non-extensibility precludes writing a generic programming library. Generic functions are more concise, but ad-hoc polymorphic functions are more flexible.

While previous foundational work [2, 7, 3, 9] provides a very strong basis for generic programming, most of it only considered non-extensible generic functions. It was realized by many authors [5, 4, 8] that this was a severe limitation.

This paper makes the following contributions:

- In Section 1.3, we give an encoding of extensible generic functions directly within Haskell 98 that is modular, overcoming the limitations of GM while retaining its advantages. An extensible generic pretty printer is presented in Section 1.4.

- In Section 1.5, we show that using a type class with two parameters, a small extension to Haskell 98, the notational overhead can be significantly reduced further.

- The fact that an extensible and modular generic programming library requires the ability to add both new generic functions and new ad-hoc cases is related

  to the expression problem [18]. We establish this relation and present other related work in Section 1.6.

But let us start with the fundamentals of the GM approach, and why extensibility in this framework is not easy to achieve.

1.2 GENERICS FOR THE MASSES

In this section we will summarise the key points of the GM approach.

1.2.1 A class for generic functions

In the GM approach to generic programming, each generic function is an instance of the class *Generic*:

class *Generic* g **where**
$$
\begin{aligned}
&\mathit{unit} && :: g\ \mathbb{1}\\
&\mathit{plus} && :: g\ a \to g\ b \to g\ (a+b)\\
&\mathit{prod} && :: g\ a \to g\ b \to g\ (a \times b)\\
&\mathit{constr} && :: \mathit{Name} \to \mathit{Arity} \to g\ a \to g\ a\\
&\mathit{constr}\ \_\ \_ && = \mathit{id}\\
&\mathit{char} && :: g\ \mathit{Char}\\
&\mathit{int} && :: g\ \mathit{Int}\\
&\mathit{view} && :: \mathit{Iso}\ b\ a \to g\ a \to g\ b
\end{aligned}
$$

Our generic binary encoder in Figure 1.1 is one such instance. The idea of *Generic* is that g represents the type of the generic function and each method of the type class represents a case of the generic function. Haskell 98 severely restricts the type terms that can appear in instance declarations. To fulfil these restrictions, we have to pack the type of the encoder in a data type *Encode*. For convenience, we define *Encode* as a record type with one field and an accessor function $\mathit{encode}' ::$

Encode $a \to (a \to [Bit])$.

The first three methods of class *Generic* are for the unit, sum and product types that are defined as follows:

$$\textbf{data}\ \mathbb{1} \quad = \mathbb{1}$$
$$\textbf{data}\ a + b = Inl\ a \mid Inr\ b$$
$$\textbf{data}\ a \times b = a \times b$$

The types of the class methods follow the kinds of the data types [3], where kinds are types of type-level terms. Types with values such as *Int*, *Char*, and $\mathbb{1}$ have kind $*$. The parameterized types $+$ and $\times$ have kind $* \to * \to *$, to reflect the fact that they are binary operators on types of kind $*$. The functions *plus* and *prod* correspondingly take additional arguments that capture the recursive calls of the generic function on the parameters of the data type.

The binary encoder is defined to encode the unit type as the empty sequence of bits. In the sum case, a 0 or 1 is generated depending on the constructor of

the input value. In the product case, we concatenate the encodings of the left and

right component.

If our generic functions require information about the constructors (such as the name and arity), we can optionally provide a definition for the function *constr*. Otherwise – such as for the binary encoder – we can just use the default implementation, which ignores the extra information.

Cases for the primitive types *Char* and *Int* are defined by providing the functions *char* and *int*, respectively.

The *view* function is the key to genericity: given an isomorphism (between the data type and a sum of products) and a representation for the isomorphic type, returns a representation for the original data type. Let us look at lists as an example. A list of type $[a]$ can be considered as a binary sum (it has two constructors), where the first argument is a unit (the $[\,]$ constructor has no arguments) and the second argument is a pair (the $(:)$ constructor has two arguments) of an element of type a and another list of type $[a]$. This motivates the following definitions:

$$\begin{aligned}
&\textbf{data}\ \mathit{Iso}\ a\ b = \mathit{Iso}\ \{\mathit{from} :: a \to b, \mathit{to} :: b \to a\}\\
&\mathit{isoList} :: \mathit{Iso}\ [a]\ (\mathbb{1} + (a \times [a]))\\
&\mathit{isoList} = \mathit{Iso}\ \mathit{fromList}\ \mathit{toList}\\
&\mathit{fromList} :: [a] \to \mathbb{1} + (a \times [a])\\
&\mathit{fromList}\ [\,] \quad\ = \mathit{Inl}\ \mathbb{1}\\
&\mathit{fromList}\ (x : xs) = \mathit{Inr}\ (x \times xs)
\end{aligned}$$

$$
\begin{aligned}
&\mathit{toList} :: \mathbb{1} + (a \times [a]) \to [a] \\
&\mathit{toList}\ (\mathit{Inl}\ \mathbb{1}) \qquad\quad = [\,] \\
&\mathit{toList}\ (\mathit{Inr}\ (x \times xs)) = x : xs
\end{aligned}
$$

In order to use generic functions on a data type, the programmer must define such an isomorphism once. Afterwards, all generic functions can be used on the data type by means of the *view* case. The function *rList* – also within the programmer's responsibility – captures how to apply *view* in the case of lists:

$$
\begin{aligned}
&\mathit{rList} :: \mathit{Generic}\ g \Rightarrow g\ a \to g\ [a] \\
&\mathit{rList}\ a = \mathit{view}\ \mathit{isoList}\ (\mathit{unit}\ \text{`}\mathit{plus}\text{`}\ (a\ \text{`}\mathit{prod}\text{`}\ \mathit{rList}\ a))
\end{aligned}
$$

The first argument of *view* is the isomorphism for lists defined above. The second argument reflects the list-isomorphic type $\mathbb{1} + (a \times [a])$. Using *rList*, we can apply any generic function to a list type, by viewing any list as a sum of products and then using the generic definitions for the unit, sum and product types.

The *view* case of the encoder applies the *from* part of the isomorphism to convert the type of the input value and then calls the encoder recursively on that value. Generally, functions such as the encoder where the type variable appears only in the argument position are called generic *consumers* and require only the *from* part of the isomorphism. Functions that *produce* or *transform* values generically make also use of the *to* component.

We have demonstrated that a fixed amount of code (the isomorphism and the *view* invocation) is sufficient to adapt all generic functions to work on a new data

type. This is a huge improvement over ad-hoc polymorphic functions, which have to be extended one by one to work on an additional data type.

1.2.2 Using generic functions

In order to call a generic function such as $encode'$, we have to supply a suitable value of type *Encode*. As a simple example, suppose that we want to call the function $encode'$ on a pair of an integer and a character. We then use *prod int char* to build the desired value of type $Encode\ (Int \times Char)$, and use $encode'$ to extract the function:

$$Main\rangle\ encode'\ (prod\ int\ char)\ (1 \times \texttt{'x'})$$
$$[1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1]$$

Similarly, if we want to encode a list of strings, we can make use of function *rList* to build the argument to $encode'$:

```
Main⟩ encode' (rList (rList char)) ["xy","x"]
[1,1,0,0,0,1,1,1,1,1,1,0,0,1,1,1,1,0,1,1,0,0,0,1,1,1,1,0,0]
```

The argument to *encode'* is dictated by the type at which we call the generic function. We can therefore use the type class *Rep*, shown in Figure 1.2, to infer this so-called *type representation* automatically for us. We call such a type class a *dispatcher*, because it selects the correct case of a generic function depending on the type context in which it is used. Note that the dispatcher works for any g that is an instance of *Generic*. Therefore, it needs to be defined only once for all generic functions. With the dispatcher, we can define *encode* as follows:

```
encode :: Rep t ⇒ t → [Bit]
encode = encode' rep
```

Here, the type representation is implicitly passed via the type class. The function *encode* can be used with the same convenience as any ad-hoc overloaded function, but it is truly generic.

1.3 EXTENSIBLE GENERIC FUNCTIONS

This section consists of two parts: in the first part, we demonstrate how the non-

extensibility of GM functions leads to non-modularity. In the second part, we show how to overcome this limitation.

1.3.1 The modularity problem

Suppose that we want to encode lists, and that we want to use a different encoding of lists than the one derived generically: a list can be encoded by encoding its length, followed by the encodings of all the list elements. For long lists, this


```
class Rep a where
  rep :: (Generic g) ⇒ g a
instance Rep 𝟙 where
  rep = unit
instance Rep Char where
  rep = char
instance Rep Int where
  rep = int
instance (Rep a, Rep b) ⇒ Rep (a + b) where
  rep = plus rep rep
```

```
instance (Rep a, Rep b) ⇒ Rep (a × b) where
  rep = prod rep rep
instance Rep a ⇒ Rep [a] where
  rep = rList rep
```

FIGURE 1.2. A generic dispatcher

encoding is more efficient than to separate any two subsequent elements of the lists and to mark the end of the list.

The class *Generic* is the base class of all generic functions, and its methods are limited. If we want to design a generic programming library, it is mandatory that we constrain ourselves to a limited set of frequently used types. Still, we might hope to add an extra case by introducing subclasses:

```
class Generic g ⇒ GenericList g where
  list :: g a → g [a]
  list = rList
```

This declaration introduces a class *GenericList* as a subclass of *Generic*: we can

only instantiate *GenericList* for type g that are also instances of class *Generic*. The subclass contains a single method *list*. By default, *list* is defined to be just *rList*. However, the default definition of *list* can be overridden in an instance declaration. For example, here is how to define the more efficient encoding for lists:

instance *GenericList Encode* **where**
$\quad list\ a = Encode\ (\lambda x \rightarrow encodeInt\ (length\ x) +\!\!+ concatMap\ (encode'\ a)\ x)$

Our extension breaks down, however, once we try to adapt the dispatcher: the method *rep* of class *Rep* has the type $Generic\ g \Rightarrow g\ a$, and we cannot easily replace the context *Generic* with something more specific without modifying the *Rep* class. Therefore, the only methods of a type class depending on the type

class *RepEncode t* **where**
$\quad repEncode :: Encode\ t$

instance *RepEncode* $\mathbb{1}$ **where**
$\quad repEncode = Encode\ (encode'\ unit)$

instance *RepEncode Int* **where**
$\quad repEncode = Encode\ (encode'\ int)$

```
instance RepEncode Char where
  repEncode = Encode (encode' char)
instance (RepEncode a, RepEncode b) ⇒ RepEncode (a + b) where
  repEncode = Encode (encode' (plus repEncode repEncode))
instance (RepEncode a, RepEncode b) ⇒ RepEncode (a × b) where
  repEncode = Encode (encode' (prod repEncode repEncode))
```

FIGURE 1.3. An ad-hoc dispatcher for binary encoders

variable g that we can use at the definitions of *rep* are those of *Generic* – any uses of methods from subclasses of *Generic* will result in type errors. In particular, we cannot adapt the instance of *Rep* for lists to make use of *list* rather than *rList*.

Consequently, generic functions in the GM approach are not extensible. This rules out modularity: all cases that can appear in a generic function must be turned into methods of class *Generic*, and as we have already argued, this is impossible: it may be necessary to add specific behaviour on user-defined or abstract types that are simply not known to the library writer.

1.3.2 Ad-hoc dispatchers

The problem with the GM approach is that the generic dispatcher forces a specific dispatching behaviour on all generic functions. A simple solution to this problem is to specialize the dispatcher *Rep* to the generic function in question. This means that we now need one dispatcher for each generic function, but it also means that extensibility is no longer a problem. Figure 1.3 shows what we obtain by specializing *Rep* to the binary encoder. In the instances, we use $encode'$ to extract the value from the **newtype** and redirect the call to the appropriate case in *Generic*. The specialized dispatcher can be used just as the general dispatcher before, to define a truly generic binary encoder:

$$encode :: RepEncode\ t \Rightarrow t \rightarrow [Bit]$$
$$encode = encode'\ repEncode$$

It is now trivial to extend the dispatcher to new types. Consider once more the ad-hoc case for encoding lists, defined by providing an **instance** declaration for *GenericList Encode*. The corresponding dispatcher extension is performed as follows:

```
instance RepEncode a ⇒ RepEncode [a] where
  repEncode = Encode (encode' (list repEncode))
```

Let us summarize. By specializing dispatchers to specific generic functions, we obtain an encoding of generic functions in Haskell that is just as expressive as the GM approach and shares the advantage that the code is pure Haskell 98. Additionally, generic functions with specialized dispatchers are extensible: we can place the type class *Generic* together with functions such as *encode* in a library that is easy to use and extend by programmers.

1.4 EXAMPLE: AN EXTENSIBLE GENERIC PRETTY PRINTER

In this section we show how to define a *extensible generic pretty printer*. This example is based on the non-modular version presented in GM (originally based on Wadler's work [19]).

1.4.1 A generic pretty printer

In Figure 1.4 we present an instance of *Generic* that defines a generic pretty printer. The pretty printer makes use of Wadler's pretty printing combinators. These combinators generate a value of type *Doc* that can be rendered into a string

afterwards. For the structural cases, the *unit* function just returns an empty document; *plus* decomposes the sum and pretty prints the value; for products, we pretty print the first and second components separated by a line. For base types *char* and *int* we assume existing pretty printers *prettyChar* and *prettyInt*. The *view* case just uses the isomorphism to convert between the user defined type and its structural representation. Finally, since pretty printers require extra constructor information, the function *constr* calls *prettyConstr*, which pretty prints constructors.

Suppose that we add a new data type *Tree* for representing labelled binary trees. Furthermore, the nodes have an auxiliary integer value that can be used to track the maximum depth of the subtrees.

```
data Tree a = Empty | Fork Int (Tree a) a (Tree a)
```

Now, we want to use our generic functions with *Tree*. As we have explained before, what we need to do is to add a subclass of *Generic* with a case for the new data type and provide a suitable *view*.

```
class Generic g ⇒ GenericTree g where
  tree :: g a → g (Tree a)
  tree a = view isoTree (constr "Empty" 0 unit `plus`
                         constr "Fork"  4
                           (int `prod` (rTree a `prod` (a `prod` rTree a))))
```

(We omit the boilerplate definition of *isoTree*). Providing a pretty printer for *Tree* amounts to declaring an empty instance of *GenericTree* – that is, using the default definition for *tree*.


```
newtype Pretty a = Pretty {pretty' :: a → Doc}
instance Generic Pretty where
  unit        = Pretty (const empty)
  char        = Pretty (prettyChar)
  int         = Pretty (prettyInt)
  plus a b    = Pretty (λx → case x of Inl l  → pretty' a l
                                       Inr r → pretty' b r)
  prod a b    = Pretty (λ(x × y) → pretty' a x ◇ line ◇ pretty' b y)
  view iso a  = Pretty (pretty' a ∘ from iso)
  constr n ar a = Pretty (prettyConstr n ar a)
prettyConstr n ar a x = let s = text n in
  if ar == 0 then s
             else group (nest 1 (text "(" ◇ s ◇ line ◇ pretty' a x ◇ text ")"))
```

FIGURE 1.4. A generic prettier printer

```
Main⟩ let t = Fork 1 (Fork 0 Empty 'h' Empty) 'i' (Fork 0 Empty '!' Empty)
Main⟩ render 80 (pretty' (tree char) t)
(Fork 1 (Fork 0 Empty 'h' Empty) 'i' (Fork 0 Empty '!' Empty))
Main⟩ let i = Fork 1 (Fork 0 Empty 104 Empty) 105 (Fork 0 Empty 33 Empty)
Main⟩ render 80 (pretty' (tree (Pretty (λx → text [chr x]))) i)
(Fork 1 (Fork 0 Empty h Empty) i (Fork 0 Empty ! Empty))
Main⟩ render 80 (pretty t)
(Fork 1 (Fork 0 Empty 'h' Empty) 'i' (Fork 0 Empty '!' Empty))
```

FIGURE 1.5. A sample interactive sesssion

instance *GenericTree Pretty*

We now demonstrate the use of generic functions, and the pretty printer in particular, by showing the outcome of a console session in Figure 1.5.

The first use of *pretty'* prints the tree *t* using the generic functionality given by *tree* and *char*. More interestingly, the second example (on a tree of integers), shows that we can override the generic behaviour for the integer parameter by providing a user-defined function instead of *int* – in this case, we interpret an integer as the code of a character using the function *chr*.

Whenever the extra flexibility provided by the possibility of overriding the generic behaviour is not required (as in the first call of *pretty'*), we can provide a dispatcher such as the one presented in Figure 1.6 and just use the convenient *pretty* function.


```
class RepPretty a where
  repPretty     :: Pretty a
  repPrettyList :: Pretty [a]
  repPrettyList = Pretty (pretty' (list repPretty))
instance RepPretty 𝟙 where
  repPretty     = Pretty (pretty' repPretty)
instance RepPretty Char where
  repPretty     = Pretty (pretty' char)
  repPrettyList = Pretty prettyString
```

```
instance RepPretty Int where
  repPretty      = Pretty (pretty' int)
instance (RepPretty a, RepPretty b) ⇒ RepPretty (a + b) where
  repPretty      = Pretty (pretty' (plus repPretty repPretty))
instance (RepPretty a, RepPretty b) ⇒ RepPretty (a × b) where
  repPretty      = Pretty (pretty' (prod repPretty repPretty))
instance RepPretty a ⇒ RepPretty (Tree a) where
  repPretty      = Pretty (pretty' (tree repPretty))
pretty :: RepPretty t ⇒ t → Doc
pretty = pretty' repPretty
```

FIGURE 1.6. An ad-hoc dispatcher for pretty printers

1.4.2 Showing lists

For user-defined types like *Tree*, our generic pretty printer can just reuse the generic functionality and the results will be very similar to the ones we get if

we just append **deriving** *Show* to our data type definitions. However, this does not work for built-in lists. The problem with lists is that they use a special mix-fix notation instead of the usual alphabetic and prefix constructors. Fortunately, we have seen in Section 1.3 that we can combine ad-hoc polymorphic functions with generic functions. We shall do the same here: we define an instance of *GenericList Pretty* but, deviating from *GenericTree Pretty*, we override the default definition.

```
instance GenericList Pretty where
  list p = Pretty (λx →
    case x of []       → text "[]"
              (a : as) → group (nest 1 (text "[" ◇ pretty' p a ◇ rest as)))
    where rest []       = text "]"
          rest (x : xs) = text "," ◇ line ◇ pretty' p x ◇ rest xs
```


We can now extend the dispatcher in Figure 1.6 with an instance for lists that uses Haskell's standard notation.

```
instance RepPretty a ⇒ RepPretty [a] where
```

$pretty = pretty'\ (list\ repPretty)$

Unfortunately, we are not done yet. In Haskell there is one more special notation involving lists: strings are just lists of characters, but we want to print them using the conventional string notation. So, not only do we need to treat lists in a special manner, but we also need to handle lists of characters specially. We thus have to implement a nested case analysis on types. We anticipated this possibility in Figure 1.6 and included a function *repPrettyList*. The basic idea is that *repPrettyList* behaves as expected for all lists except the ones with characters, where it uses *prettyString*.This is the same as Haskell does in the *Show* class. Finally, we modify *RepPretty* $[a]$ to redirect the call to *prettyList* and we are done.

instance *RepPretty* $a \Rightarrow$ *RepPretty* $[a]$ **where**
  $repPretty = repPrettyList$

In the pretty printer presented in GM, supporting the list notation involved adding an extra case to *Generic*, which required us to have access to the source code where *Generic* was originally declared. In contrast, with our solution, the addition of a special case for lists did not involve any change to our original *Generic* class or even its instance for *Pretty*.

The additional flexibility of ad-hoc dispatchers comes at a price: while in GM the responsibility of writing the code for the dispatchers was on the library

writer side, now this responsibility is on the user of the library, who has to write additional boilerplate code. Still, it is certainly preferable to define an ad-hoc dispatcher than to define the function as an ad-hoc polymorphic function, being forced to give an actual implementation for each data type. Yet, it would be even better if we could somehow return to a single dispatcher that works for all generic functions and restore the definition of the dispatcher to the library code.

In the next section we will see an alternative encoding that requires a single generic dispatcher only and still allows for modular and extensible functions. The price to pay for this is that the code requires a small extension to Haskell 98.

1.5 MAKING AD-HOC DISPATCHERS LESS AD-HOC

In this section we present another way to write extensible generic functions, which requires only one generic dispatcher just like the original GM approach. It relies, however, on an extension to Haskell 98: multi-parameter type classes, which are widely used and supported by the major Haskell implementations.

Recall the discussion at the end of Section 1.3.1. There, we have shown that the problem with GM's dispatcher is that it fixes the context of method *rep* to the class *Generic*. This happens because the type variable *g*, which abstracts over the "generic function", is universally quantified in the class method *rep*. However,

```
instance Generic g ⇒ GRep g 𝟙 where
  grep = unit
instance Generic g ⇒ GRep g Int where
  grep = int
instance Generic g ⇒ GRep g Char where
  grep = char
instance (Generic g, GRep g a, GRep g b) ⇒ GRep g (a + b) where
  grep = plus grep grep
instance (Generic g, GRep g a, GRep g b) ⇒ GRep g (a × b) where
  grep = prod grep grep
instance (GenericList g, GRep g a) ⇒ GRep g [a] where
  grep = list grep
instance (GenericTree g, GRep g a) ⇒ GRep g (Tree a) where
  grep = tree grep
```

FIGURE 1.7. A less ad-hoc dispatcher.

since we want to use subclasses of *Generic* to add additional cases to generic functions, the context of *rep* must be flexible. We therefore must be able to abstract from the specific type class *Generic*. Our solution for this problem is to change the quantification of g: instead of universally quantifying g at the method *rep* we can quantify it on the type class itself.

```
class GRep g a where
  grep :: g a
```

The type class *GRep g a* is a variation of *Rep a* with the proposed change of quantification. The fact that g occurs at the top-level gives us the extra flexibility that we need to provide more refined contexts to the method *grep* (which corresponds to the method *rep*).

In Figure 1.7 we see how to use this idea to capture all ad-hoc dispatchers in a single definition. The instances of *GRep* look just like the instances of *Rep* except that they have the extra parameter g at the top-level. The structural cases $\mathbb{1}$, $+$ and $\times$ together with the base cases *int* and *char* are all handled in *Generic*, therefore we require g to be an instance of *Generic*. However, for $[a]$ and *Tree a* the argument g must be constrained by *GenericList* and *GenericTree*, respectively, since these are the type classes that handle those types. The remaining constraints, of the form *GRep g a*, contain the necessary information to perform the recursive

calls. Now, we can just use this dispatcher to obtain an extensible *encode* by specializing the argument g to *Encode*:

$$encode :: GRep\ Encode\ t \Rightarrow t \rightarrow [Bit]$$
$$encode = encode'\ grep$$

For pretty printers we can just use the same dispatcher, but this time using *Pretty* instead of *Encode*:

$$pretty :: GRep\ Pretty\ t \Rightarrow t \rightarrow Doc$$
$$pretty = pretty'\ grep$$

This approach avoids the extra boilerplate of the solution with ad-hoc dispatchers presented in Sections 1.3 and 1.4 requiring a similar amount of work to the original GM technique. Still it is modular, and allows us to write a generic programming library.

In a previous version of this paper [13] we used a trick proposed by Hughes [6] and also used by Lämmel and Peyton Jones [8] that simulated abstraction over type classes using a class of the form:

class *Over t* **where**

over :: *t*

While the same effect could be achieved using *Over* instead of *GRep*, this would be more demanding on the type system since the instances would not be legal in Haskell 98: For example, the instance for *GRep g Int* would become *Over* (*g Int*). The former is legal in Haskell 98, the latter is not. Moreover, *GRep* is also more finely typed allowing us to precisely specify the kind of the "type class" that we are abstracting from.

Since the publication of the original version of this paper, Sulzmann and Wang [16] have shown how to add extensible superclasses to the Haskell language, which would constitute another solution to the extensibility problem for GM generic functions.

1.6 DISCUSSION AND RELATED WORK

In this section we briefly relate our technique to the *expression problem* [18] and discuss some other closely related work.

1.6.1 Expression problem

Wadler [18] identified the need for extensibility in two dimensions (adding new variants *and* new functions) as a problem and called it the expression problem.

According to him, a solution for the problem should allow the definition of a data type, the addition of new variants to such a data type as well as the addition of new functions over that data type. A solution should not require recompilation of existing code, and it should be statically type safe: applying a function to a variant for which that function is not defined should result in a compile-time error. Our solution accomplishes all of these for the particular case of generic functions. It should be possible to generalize our technique in such a way that it can be applied to other instances of the expression problem. For example, the work of Oliveira and Gibbons [11], which generalizes the GM technique as a design pattern, could be recast using the techniques of this paper.

Let us analyze the role of each type class of our solution in the context of the expression problem. The class *Generic* plays the role of a data type definition and declares the variants that *all* functions should be defined for. The subclasses of *Generic* represent extra variants that we add: not all functions need to be defined for those variants, but if we want to use a function with one of those, then we need to provide the respective case. The instances of *Generic* and subclasses are the bodies of our extensible functions. Finally, the dispatcher allows us to encode the dispatching behaviour for the extensible functions: if we add a new variant and we want to use it with our functions, we must add a new instance for that variant.

1.6.2 Other related work

Generic Haskell (GH) [9] is a compiler for a language extension of Haskell that supports generic programming. The compiler can generate Haskell code that can then be used with a Haskell compiler. Like our approach, GH uses sums of products for viewing user defined types. GH can generate the boilerplate code required for new data types automatically. With our approach we need to manually provide this code, but we could employ an additional tool to facilitate its generation. However, our generic functions are *extensible*; at any point we can add an extra ad-hoc case for some generic function. We believe this is of major importance since, as we have been arguing, extensible functions are crucial for a modular generic programming library. This is not the case for GH since all the special cases need to be defined at once. Also, since GH is an external tool it is less convenient to use. With our approach, all we have to do is to import the modules with the generic library.

"Derivable Type Classes" (DTCs) [5] is a proposed extension to Haskell that allows us to write generic default cases for methods of a type class. In this approach, data types are viewed as if constructed by binary sums and binary products, which makes it a close relative of both our approach and GM. The main advantage of DTCs is that it is trivial to add ad-hoc cases to generic functions, and

the isomorphisms between data types and their structural representations (see Section 1.2.1) are automatically generated by the compiler. However, the approach permits only generic functions on unparameterized types (types of kind $\star$), and the DTC implementation lacks the ability to access constructor information, precluding the definition of generic parsers or pretty printers. The generic extension to Clean [1] uses the ideas of DTCs and allows the definition of generic functions on types of any kind.

Lämmel and Peyton Jones [8] present another approach to generic programming based on type classes. The idea is similar to DTCs in the sense that one type class is defined for each generic function and that default methods are used to provide the generic definition. Overriding the generic behaviour is as simple as providing an instance with the ad-hoc definition. The approach shares DTC's limitation to generic functions on types of kind $\star$. One difference to our approach is that data types are not mapped to a common structure consisting of sums and products. Instead, generic definitions make use of a small set of combinators. An-

other difference is that their approach relies on some advanced extensions to the type class system, while our approach requires only a multi-parameter type class or even just Haskell 98.

Löh and Hinze [10] propose an extension to Haskell that allows the definition

of extensible data types and extensible functions. With the help of this extension, it is also possible to define extensible generic functions, on types of any kind, in Haskell. While their proposed language modification is relatively small, our solution has the advantage of being usable right now. Furthermore, we can give more safety guarantees: in our setting, a call to an undefined case of a generic function is a static error; with open data types, it results in a pattern match failure.

Vytiniotis and others [17] present a language where it is possible to define extensible generic functions on types of any kind, while guaranteeing static safety. While it is not a novelty that we can define such flexible generic functions, we believe it is the first time that a solution with all these features is presented in Haskell, relying solely on implemented language constructs or even solely on Haskell 98.

1.7 CONCLUSIONS

In the GM approach defining generic functions in Haskell 98 is possible but it is impossible to extend them in a modular way. The ability to define extensible generic function is very important since, in practice, most generic functions have ad-hoc cases. In this paper we presented two variations of GM that allow the definition of generic functions that are both extensible and modular. The first variation, like the original GM, can be encoded using Haskell 98 only but requires

extra boilerplate code not present in the original approach. The second variation requires a multi-parameter type class, an extension to Haskell 98 that is supported by the major Haskell implementations. This variation still allows extensibility and does not add any significant boilerplate over the original GM. One important aspect of the GM and our encoding is that dispatching generic functions is resolved statically: calling a generic function on a case that is not defined for it is a compile-time error.

Based on the results of this paper, we are currently in the process of assembling a library of frequently used generic functions. For the interested reader, the Haskell source code for this paper can be found online [12].

ACKNOWLEDGEMENTS

We would like to thank Jeremy Gibbons and Fermín Reig for valuable suggestions and discussions about this work. We would also like to thank the anonymous referees whose reviews greatly contributed for improving the presentation of this paper. This work was partially funded by the *EPSRC Datatype-Generic Programming* and the *DFG "A generic functional programming language"* projects.

REFERENCES

[1] Artem Alimarine and Marinus J. Plasmeijer. A generic programming extension for Clean. In *Implementation of Functional Languages*, pages 168–185, 2001.

[2] Richard Bird, Oege de Moor, and Paul Hoogendijk. Generic functional programming with types and relations. *Journal of Functional Programming*, 6(1):1–28, 1996.

[3] Ralf Hinze. Polytypic values possess polykinded types. In Roland Backhouse and J. N. Oliveira, editors, *Proceedings of the Fifth International Conference on Mathematics of Program Construction, July 3–5, 2000*, volume 1837 of *Lecture Notes in Computer Science*, pages 2–27. Springer-Verlag, 2000.

[4] Ralf Hinze. Generics for the masses. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 236–243. ACM Press, 2004.

[5] Ralf Hinze and Simon Peyton Jones. Derivable type classes. In Graham Hutton, editor, *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41.1 of Electronic Notes in Theoretical Computer Science. Elsevier Science, August 2001. The preliminary proceedings appeared as a University of Nottingham technical report.

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

[7] Patrik Jansson. *Functional Polytypic Programming*. PhD thesis, Chalmers University of Technology, May 2000.

[8] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)*, pages 204–215. ACM Press, September 2005.

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

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

[11] B. Oliveira and J. Gibbons. Typecase: A design pattern for type-indexed functions. In *Haskell Workshop*, pages 98–109, 2005.

[12] Bruno Oliveira, Ralf Hinze, and Andres Löh. Source code accompanying "Extensible and modular generics for the masses".
Available from http://web.comlab.ox.ac.uk/oucl/work/bruno.oliveira/.

[13] Bruno Oliveira, Ralf Hinze, and Andres Löh. Generics as a library. In *Seventh Symposium on Trends in Functional Programming*, 2006.

[14] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[15] Christopher Strachey. Fundamental concepts in programming languages. Lecture Notes, International Summer School in Computer Programming, Copenhagen, August 1967. Reprinted in *Higher-Order and Symbolic Computation*, 13(1/2), pp. 1–49, 2000.

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

[17] Dimitrios Vytiniotis, Geoffrey Washburn, and Stephanie Weirich. An open and shut typecase. In *TLDI '05: Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 13–24, New York, NY, USA, 2005. ACM Press.

[18] Philip Wadler. The expression problem. Java Genericity Mailing list, November 1998.

[19] Philip Wadler. A prettier printer. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, pages 223–244. Palgrave Macmillan, 2003.