Generics as a Library

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

[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

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 a customizable 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 a customizable generic programming library in Haskell 98 is possible.

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 in Haskell [1], such as *show*, *read*, and '$==$', but there are many more.

By contrast, an *ad-hoc polymorphic* function [2] 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:

```
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)
```

```
newtype Encode a = Encode{ encode′ :: a → [Bit] }
instance Generic Encode where
  unit       = Encode (const [ ])
  char       = Encode encodeChar
  int        = Encode encodeInt
  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)
  view iso a = Encode (λx → encode′ a (from iso x))
```

FIGURE 1. A generic binary encoder

This function works on all data types built from integers, characters and lists. We assume that primitive bit encoders for integers and characters are provided from somewhere. Lists are encoded by replacing an occurrence of [ ] with the bit 0, and occurrences of (:) with the bit 1 followed by the encoding of the head element and the encoding of the remaining list.

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) [3] 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, 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 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.

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 consider a representation of sets as balanced trees. The same set can be represented by multiple trees, so a generic equality function must not compare sets structurally, therefore we need an ad-hoc case for set representations.

Defining ad-hoc cases for ad-hoc polymorphic functions is trivial. For the generic version of the binary encoder, the addition of a new case is, however, very difficult. Each case is 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 [4, 5, 6, 7] provides a very strong basis for generic programming, most of it only considered non-extensible generic functions. It was realized by many authors [8, 3, 9] that this was a severe limitation.

This paper makes the following contributions:

- We give an encoding of extensible generic functions directly within Haskell 98 that is modular, overcoming the limitations of GM while retaining its advantages.
- We show that by using already implemented language extensions, the notational overhead can be significantly reduced further.
- We relate our solution to the expression problem [10].

The rest of the paper is structured as follows. In Section 2 we repeat the fundamentals of the GM approach, and demonstrate why extensibility is not easy to achieve. In Section 3, we give a new encoding of generic functions in Haskell 98 that is based on GM *and* that is modular. In Section 4 we look at the problem of pretty printing values generically and show how we can code a modular pretty printer with our encoding. A disadvantage of this encoding is that it requires the programmer to write a relatively large amount of boilerplate code per generic function. In Section 5 we therefore show how we can employ some widely used and implemented extensions of the Haskell language to reduce the workload on the programmer significantly. Finally, in Section 6, we relate our technique to the expression problem and discuss other related work.

2 GENERICS FOR THE MASSES

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

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
  unit    :: g 𝟙
  char    :: g Char
```

$$\begin{aligned}
&\mathit{int} && :: g\ \mathit{Int} \\
&\mathit{plus} && :: g\ a \to g\ b \to g\ (a + b) \\
&\mathit{prod} && :: g\ a \to g\ b \to g\ (a \times b) \\
&\mathit{constr} && :: \mathit{Name} \to \mathit{Arity} \to g\ a \to g\ a \\
&\mathit{constr}\ \_\ \_ = \mathit{id} \\
&\mathit{view} && :: \mathit{Iso}\ b\ a \to g\ a \to g\ b
\end{aligned}$$

Our generic binary encoder in Figure 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. The first three methods are for the unit, sum and product types that are defined as follows:

$$\begin{aligned}
&\mathbf{data}\ \mathbb{1} && = \mathbb{1} \\
&\mathbf{data}\ a + b && = \mathit{Inl}\ a \mid \mathit{Inr}\ b \\
&\mathbf{data}\ a \times b && = a \times b
\end{aligned}$$

The types of the class methods follow the kinds of the data types [6]: for parameterized types such as $+$ or $\times$, the function takes additional arguments that capture the recursive calls of the generic function on the parameters of the data type.

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, we can just use the default implementation, which just ignores the extra information.

We can define cases for primitive types *Char* and *Int* by providing, respectively, the functions *char* and *int*.

Finally, the *view* function allows us to use generic functions on many Haskell data types, given an isomorphism between the data type and its structural representation. Here is an example of the isomorphism for the data type of lists:

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

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. 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.

```
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 2. A generic dispatcher

2.2 Using generic functions

In order to call a generic function such as *encode′*, we have to provide a suitable value of type *Encode*. We can use the type class *Rep*, shown in Figure 2, to infer this so-called *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* and therefore it only needs to be defined once for all generic functions. With the help of the class *Rep*, 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. In order to extend the function with new type cases we need to create a representation for that type. For example,

```
rList :: Generic g ⇒ g a → g [a]
rList a = view isoList (unit ‘plus‘ (a ‘prod‘ rList a))
```

is the representation for lists. The embedding into *Generic* is via *view* and the previously defined isomorphism on lists.

In the following section, we will show why the GM approach is not modular, and present a way to overcome this problem.

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.

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 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 can add an extra case by introducing subclasses:

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

By default, *list* is just *rList*. However, because *list* is a default method of a type class, it can be overridden in the instances. For example, here is how to define the more efficient encoding for lists:

```
instance GenericList Encode where
  list a = Encode (λx → encodeInt (length x) ++
                              concatMap (encode′ a) x)
```

Our extension breaks down, however, when we try to adapt the dispatcher: the method *rep* has type *Generic g ⇒ g a*, and we cannot easily replace the context *Generic* with something more specific.

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.

3.2 Ad-hoc dispatchers

The problem with the GM approach is that the generic dispatcher is actually too general, and forces a specific dispatching behaviour on all generic functions. The solution to this problem is simple, yet intriguing: in order to make a generic function extensible, we specialize *Rep* to the generic function in question. Figure 3

```
class REncode t where
  encode :: t → [Bit]
instance REncode 𝟙 where
  encode = encode′ unit
instance REncode Int where
  encode = encode′ int
instance REncode Char where
  encode = encode′ char
instance (REncode a, REncode b) ⇒ REncode (a + b) where
  encode = encode′ (plus repEncode repEncode)
instance (REncode a, REncode b) ⇒ REncode (a × b) where
  encode = encode′ (prod repEncode repEncode)
```

FIGURE 3. An ad-hoc dispatcher for binary encoders

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 function *repEncode*, which plays the role of *rep*, is defined as:

```
repEncode :: REncode a ⇒ Encode a
repEncode = Encode encode
```

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 REncode a ⇒ REncode [a] where
  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 equally 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.

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 [11]).

```
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 4. A generic prettier printer

4.1 A generic pretty printer

In Figure 4 we present an instance of *Generic* that encodes a generic pretty printer. The pretty printer is defined using Wadler's pretty printing combinators. These combinators generate a value of *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*.

```
instance GenericTree Pretty
```

We demonstrate the use of the pretty printer by showing the outcome of a console session:

```
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 [Char.chr x]))) i)
(Fork 1 (Fork 0 Empty h Empty) i (Fork 0 Empty ! Empty))
```

The function *render* takes the number of columns available for rendering and given a document it pretty prints it. The first use of *render* creates a document using *pretty′* and, by using the generic functionality provided by *tree* and *char*, pretty prints the tree *t*. More interestingly, the second example shows that if we have a *Tree Int*, when we override the generic behaviour for the *Int* parameter, the function that is used to pretty print the auxiliary values of *Int* is still the one provided by *int* and not the one used by the integer parameters.

Whenever, the extra flexibility provided by the possibility of overriding the generic behaviour is not required (like in the first use of *render*) we can provide a dispatcher such as the one presented in Figure 5 and just use the convenient *pretty* function.

```
Main⟩ render 80 (pretty t)
(Fork 1 (Fork 0 Empty 'h' Empty) 'i' (Fork 0 Empty '!' Empty))
```

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 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, unlike with *GenericTree Pretty*, we override the default definition.

```
instance GenericList Pretty where
  list p = Pretty (λx →
    case x of []       → text "[]"
```

```
class RPretty a where
  pretty     :: a → Doc
  prettyList :: [a] → Doc
  prettyList = pretty′ (list repPretty)
instance RPretty 𝟙 where
  pretty     = pretty′ repPretty
instance RPretty Char where
  pretty     = pretty′ char
  prettyList = prettyString
instance RPretty Int where
  pretty     = pretty′ int
instance (RPretty a, RPretty b) ⇒ RPretty (a + b) where
  pretty     = pretty′ (plus repPretty repPretty)
instance (RPretty a, RPretty b) ⇒ RPretty (a × b) where
  pretty     = pretty′ (prod repPretty repPretty)
instance RPretty a ⇒ RPretty (Tree a) where
  pretty     = pretty′ (tree repPretty)
repPretty :: RPretty t ⇒ Pretty t
repPretty = Pretty pretty
```

FIGURE 5. An ad-hoc dispatcher for pretty printers

```
              (a : as) → group (nest 1 (text "[" ◇ pretty′ p a ◇ rest as)))
    where rest []       = text "]"
          rest (x : xs) = text "," ◇ line ◇ pretty′ p x ◇ rest xs
```

Using this, we can extend the dispatcher in Figure 5 with an instance for lists that uses Haskell's standard notation.

```
instance RPretty a ⇒ RPretty [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 we need to treat lists in a special manner, but we also need to handle lists of characters specially. This basically means that we need to implement a nested case analysis on types. We anticipated this possibility in Figure 5 and included a function *prettyList*, which helps us tackling that problem. The basic idea is that *prettyList* behaves as expected for all lists except the ones with characters, where it uses *prettyString*. This is just like what *Haskell* does in the *Show* class. Now, modify of *RPretty* $[a]$ in order redirect the call to *prettyList* and we are done.

```
instance RPretty a ⇒ RPretty [a] where
  pretty = prettyList
```

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 our approach comes, however, at a price: using ad-hoc dispatchers requires the programmer to write boilerplate code that is not required for the original GM encoding. We now need to add one dispatcher for each extensible generic function. This code is highly trivial; 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.

In the next section we will see an alternative encoding that avoids the duplication of dispatchers.

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 extensions to the class system, in particular *undecidable instances* that are not standardized, but implemented in GHC and widely used.

Recall the discussion at the end of Section 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*. Since we use subclasses of *Generic* to add additional cases to generic functions, the context of *rep* must be flexible. We thus must abstract from the specific type class *Generic*. Haskell does not support abstraction over type classes, but there is a trick that can be used to achieve the same effect. The technique was first proposed by Hughes [12] and it has been used in Lämmel and Peyton Jones [9]. The key idea is to use a type class

```
class Over t where
  over :: t
```

where *over* is achieving something like object-oriented overloading. If we represent a type class such as *REncode* by a dictionary type such as *Encode*, then a constraint of the form *Over* (*Encode a*) plays a similar role as a constraint of the form *REncode a*, only that we can abstract from the specific type *Encode* in question, with type variable instead. In Figure 6 we see how to use this idea to capture all ad-hoc dispatchers in a single definition. The type constructor *g* represents the "type class" that we want to abstract from. The structural cases $\mathbb{1}$, $+$ and $\times$ together with the base cases *int* and *char* are all handled in *Generic*, therefore we

$\textbf{instance } \mathit{Generic}\ g \Rightarrow \mathit{Over}\ (g\ \mathbb{1})\ \textbf{where}$
$\quad \mathit{over} = \mathit{unit}$
$\textbf{instance } \mathit{Generic}\ g \Rightarrow \mathit{Over}\ (g\ \mathit{Int})\ \textbf{where}$
$\quad \mathit{over} = \mathit{int}$
$\textbf{instance } \mathit{Generic}\ g \Rightarrow \mathit{Over}\ (g\ \mathit{Char})\ \textbf{where}$
$\quad \mathit{over} = \mathit{char}$
$\textbf{instance } (\mathit{Generic}\ g, \mathit{Over}\ (g\ a), \mathit{Over}\ (g\ b)) \Rightarrow \mathit{Over}\ (g\ (a + b))\ \textbf{where}$
$\quad \mathit{over} = \mathit{plus\ over\ over}$
$\textbf{instance } (\mathit{Generic}\ g, \mathit{Over}\ (g\ a), \mathit{Over}\ (g\ b)) \Rightarrow \mathit{Over}\ (g\ (a \times b))\ \textbf{where}$
$\quad \mathit{over} = \mathit{prod\ over\ over}$
$\textbf{instance } (\mathit{GenericList}\ g, \mathit{Over}\ (g\ a)) \Rightarrow \mathit{Over}\ (g\ [a])\ \textbf{where}$
$\quad \mathit{over} = \mathit{list\ over}$
$\textbf{instance } (\mathit{GenericTree}\ g, \mathit{Over}\ (g\ a)) \Rightarrow \mathit{Over}\ (g\ (\mathit{Tree}\ a))\ \textbf{where}$
$\quad \mathit{over} = \mathit{tree\ over}$

FIGURE 6. A less ad-hoc dispatcher.

require g to be one instance of *Generic*. However, for $[a]$ and *Tree a* the argument g must be, respectively, constrained by *GenericList* and *GenericTree* since those are the type classes that handle those types. The remaining constraints, of the form $\mathit{Over}\ (g\ a)$, contain the necessary information to perform the recursive calls. Now, we can just use this dispatcher to obtain an extensible *encode*:

$\mathit{encode} :: \mathit{Over}\ (\mathit{Encode}\ t) \Rightarrow t \to [\mathit{Bit}]$
$\mathit{encode} = \mathit{encode}'\ \mathit{over}$

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

$\mathit{pretty} :: \mathit{Over}\ (\mathit{Pretty}\ t) \Rightarrow t \to \mathit{Doc}$
$\mathit{pretty} = \mathit{pretty}'\ \mathit{over}$

This approach requires about the same amount of work from the programmer as the original GM technique, but it is modular, and allows us to write a generic programming library.

6 DISCUSSION AND RELATED WORK

In this section we summarize our main results; then we briefly relate our technique to the *expression problem* [10]; finally, we discuss some other closely related work.

In the original GM, it is shown how to encode generic functions in Haskell 98. However functions defined with that encoding are not extensible: we can define new generic functions easily, but adding new cases (or variants) would involve the modification of existing code. With the two encodings that we introduce, generic functions can be extended with new cases, while retaining the simplicity and expressiveness of the GM approach. 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 at:

http://web.comlab.ox.ac.uk/oucl/work/bruno.oliveira/Generics.tar.gz

6.1 Expression problem

Wadler [11] 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 easy 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 [13], 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.

6.2 Other related work

Generic Haskell (GH) [7] is a tool that supports generic programming in a Haskell-like language. The tool 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. 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) [8] 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 2.1) are automatically generated by the compiler. However, the approach permits only generic functions on types of kind $\star$, and the DTC implementation lacks the ability to access constructor information, precluding the definition of generic parsers or pretty printers.

Lämmel and Peyton Jones [9] 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 $\star$. The 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. The possibility to abstract over a type class is essential to their approach, whereas it is optional, yet helpful, for ours.

Löh and Hinze [14] 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 [15] present a language where it is possible to define extensible generic functions on types of any kind, while guaranteeing static safety. Therefore, it is not a novelty that we can define such flexible generic functions. However, 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.

ACKNOWLEDGEMENTS

We would like to thank Jeremy Gibbons and Fermín Reig for valuable suggestions and discussions about this work. This work was partially funded by the *EPSRC*

Datatype-Generic Programming and the *DFG "A generic functional programming language"* projects.

REFERENCES

[1] Peyton Jones, S., ed.: Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press (2003)

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

[3] Hinze, R.: Generics for the masses. In: International Conference on Functional Programming, ACM Press (2004) 236–243

[4] Bird, R., de Moor, O., Hoogendijk, P.: Generic functional programming with types and relations. Journal of Functional Programming **6** (1996) 1–28

[5] Jansson, P.: Functional Polytypic Programming. PhD thesis, Chalmers University of Technology (2000)

[6] Hinze, R.: Polytypic values possess polykinded types. In Backhouse, R., Oliveira, J.N., eds.: Proceedings of the Fifth International Conference on Mathematics of Program Construction, July 3–5, 2000. Volume 1837 of Lecture Notes in Computer Science., Springer-Verlag (2000) 2–27

[7] Löh, A.: Exploring Generic Haskell. PhD thesis, Utrecht University (2004)

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

[9] Lämmel, R., Peyton Jones, S.: Scrap your boilerplate with class: extensible generic functions. In: Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2005), ACM Press (2005) 204–215

[10] Wadler, P.: The expression problem. Java Genericity Mailing list (1998)

[11] Wadler, P.: A prettier printer. In Gibbons, J., de Moor, O., eds.: The Fun of Programming, Palgrave Macmillan (2003) 223–244

[12] Hughes, J.: Restricted data types in Haskell. In Meijer, E., ed.: Proceedings of the 1999 Haskell Workshop. Number UU-CS-1999-28 (1999)

[13] Oliveira, B., Gibbons, J.: Typecase: A design pattern for type-indexed functions. In: Haskell Workshop. (2005) 98–109

[14] Löh, A., Hinze, R.: Open data types and open functions. Technical Report IAI-TR-2006-3, Institut für Informatik III, Universität Bonn (2006)

[15] Vytiniotis, D., Washburn, G., Weirich, S.: An open and shut typecase. In: TLDI '05: Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation, New York, NY, USA, ACM Press (2005) 13–24