Generics as a Library (extended abstract)

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. Typically, 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

Typically, 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 write ad-hoc polymorphic functions using type classes. Here is an example, a binary encoder:

```
class Encode t where
  encode :: t → [Bit]
instance Encode Int where
  encode = encodeInt
instance Encode Char where
  encode = encodeChar
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 [])
  plus a b  = Encode (λx → case x of Inl l → 0 : encode' a l
                                     Inr r → 1 : encode' b r)
  prod a b  = Encode (λx → encode' a (outl x) ++ encode' b (outr x))
  char      = Encode encodeChar
  int       = Encode encodeInt
  view iso a = Encode (λx → encode' a (from iso x))
```

Fig. 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 1, and an occurrence of [] with the bit 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.

Generic functions can be implemented in several ways. Hinze's "Generics for the Masses" (GM) [3] approach stands out among all the approaches to generic functional programming, because it allows writing generic functions directly in Haskell 98, whereas all other approaches need compiler support or significant language extensions.

Using his encoding, we could write the generic binary encoder presented in Figure 1. 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 that set.

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 [4–7] 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 is 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 workload on the programmer can be significantly reduced further.
- We relate our solution with 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* modular. 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 4 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 5, we relate our technique with 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
  plus :: g a → g b → g (a + b)
  prod :: g a → g b → g (a × b)
  char :: g Char
  int  :: g Int
  view :: Iso b a → g a → g b
```

Our generic binary encoder in Figure 1 is one of such instances. 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:

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

The types of the class methods follow the kinds of the data types: 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.

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:

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

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.

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 a type class 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. The definition of the dispatcher is shown in Figure 2. With the help of the class *Rep*, we can define *encode* as follows:

$$\begin{aligned}
&\mathit{encode} :: \mathit{Rep}\ t \Rightarrow t \to [\mathit{Bit}]\\
&\mathit{encode} = \mathit{encode}'\ \mathit{rep}
\end{aligned}$$

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

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

```
class Rep a where
  rep :: (Generic g) ⇒ g a
instance Rep 𝟙 where
  rep = unit
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 Char where
  rep = char
instance Rep Int where
  rep = int
instance Rep a ⇒ Rep [a] where
  rep = rList rep
```

Fig. 2. A generic dispatcher

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 try to add extra cases to generic functions by introducing subclasses:

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

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

Fig. 3. An ad-hoc dispatcher for binary encoders

By default, *list* is just *rList*. However, because *list* is a method of a type class, it can be overriden 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 $(Generic\ g) \Rightarrow g\ a$, 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 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 *overEncode*, which plays the role of *rep*, is defined as:

```
overEncode :: REncode a ⇒ Encode a
overEncode = 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 overEncode)
```

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.

The additional flexibility of our approach comes 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.

4 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*. If 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 the can be used to achieve the same. The technique was first proposed by Hughes [11] and it as 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, and use a type variable instead. In Figure 4 we see how to use this

```
instance Generic g ⇒ Over (g 𝟙) where
  over = unit
instance (Generic g, Over (g a), Over (g b)) ⇒ Over (g (a + b)) where
  over = plus over over
instance (Generic g, Over (g a), Over (g b)) ⇒ Over (g (a × b)) where
  over = prod over over
instance Generic g ⇒ Over (g Int) where
  over = int
instance Generic g ⇒ Over (g Char) where
  over = char
instance (GenericList g, Over (g a)) ⇒ Over (g [a]) where
  over = list over
```

Fig. 4. A less ad-hoc dispatcher.

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 require g to be one instance of *Generic*. However, the $[a]$ case is handled by *GenericList* and therefore g must be one instance of that type class. The remaining constraints, of the form *Over* $(g\ a)$, contain the necessary information to perform the recursive calls. Now, we can just use this dispatcher to obtain an extensible *encode*:

```
encode :: Over (Encode t) ⇒ t → [Bit]
encode = encode′ 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.

5 Discussion and Related Work

In this section we summarize our main results; then we briefly relate our technique to the *expression problem* [10]; and, 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.

Expression problem Wadler 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 [12], 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.

Other related work The paper "Derivable Type Classes" (DTCs) [8] proposes an 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 between our approach and their 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 [13] 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 [14] 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.

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. Hughes, J.: Restricted data types in haskell. In Meijer, E., ed.: Proceedings of the 1999 Haskell Workshop. Number UU-CS-1999-28 (1999)
12. Oliveira, B., Gibbons, J.: Typecase: A design pattern for type-indexed functions. In: Haskell Workshop. (2005) 98–109
13. Löh, A., Hinze, R.: Open data types and open functions. Technical Report IAI-TR-2006-2, Institut für Informatik III, Universität Bonn (2006)
14. 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