Deriving Generic Functions by Example

Neil Mitchell

University of York, UK
http://www.cs.york.ac.uk/~ndm/

Abstract. A function is said to be *generic* if it operates over values of *any* data type. For example, a generic equality function can test pairs of booleans, integers, lists, trees etc. In most languages programmers must define generic functions multiple times, specialised for each data type. Alternatively, a tool could be used to specify the relationship between the data type and the implementation, but this relationship may be complex. This paper describes a solution: given a single example of the generic function on one data type, we can infer the relationship between a data type and the implementation. We have used our method in the Derive tool, allowing the implementation of 60% of the generic functions to be inferred.

1 Introduction

Haskell [10] is a modern functional programming language. In Haskell a generic function can be defined using a type class [12] (using the **class** keyword), and implementations can be provided for specific types (using the **instance** keyword). Generic equality is defined by the Eq class:

```
class Eq α where
  (≡) :: α → α → Bool
```

The Eq class has an operator, (≡). All types that are instances of the class Eq have the (≡) operator available. We can define a Haskell data type, along with an instance for Eq, as follows:

```
data WritingImplement = Pencil        -- a pencil
                      | Pen Colour    -- a pen, plus its colour

instance Eq WritingImplement where
  (Pencil) ≡ (Pencil) = True
  (Pen x)  ≡ (Pen y)  = x ≡ y
  _        ≡ _        = False
```

The data type we introduce is WritingImplement. A WritingImplement is either a pencil, or a pen with a colour. Pencil and Pen are referred to as data constructors. The first line of the Eq instance says that any two Pencil values are equal. The second line says that for two Pen values to be equal, their colour

```
data DataName = First
              | Second Any
              | Third  Any Any
              | Fourth Any Any
```

Fig. 1. The DataName data type.

fields must be equal. The final line says that any remaining pairs of values are not equal. Any instance of Eq follows naturally from the data structure: for two values to be equal they must have the same constructor, and all their fields must be equal.

Writing an Eq instance for one data type is simple. However, as the complexity and number of data types increases, so does the effort required. The standard solution is to express the *relationship* between any data type and the instance that corresponds to it, which we call an *instance generator*. In standard tools such as DrIFT [13], the author of an instance generator must be familiar with both the representation of a data type, and various code-generation functions. The result is that specifying an instance generator is not as straightforward as one might hope.

Using the techniques described in this paper, instance generators can often be automatically inferred from a single example instance. To define *all* Eq instances, an example must be given on the DataName type, provided by a library (see Figure 1):

```
instance Eq DataName where
  (First       ) ≡ (First       ) = True
  (Second x1   ) ≡ (Second y1   ) = x1 ≡ y1 ∧ True
  (Third  x1 x2) ≡ (Third  y1 y2) = x1 ≡ y1 ∧ x2 ≡ y2 ∧ True
  (Fourth x1 x2) ≡ (Fourth y1 y2) = x1 ≡ y1 ∧ x2 ≡ y2 ∧ True
  _              ≡ _              = False
```

The DataName instance follows the same pattern as for WritingImplement, but with the addition of ∧ True, whose purpose is explained in §2.3.

This paper contributes a method for inferring a relationship between a value and a piece of program code, without resorting to unguided search. In our experience, over 60% of Haskell class instances can be determined using this technique.

1.1 Roadmap

This paper first describes how to derive instance generators automatically in §2. §3 discusses which generic functions are applicable for this scheme, and §4 gives a more complex example. §5 presents related work, before §6 concludes.

2 The Method

The central idea of automatic inference of instance generators is that an instance generator is a function from a data type $\mathcal{D}$, to a piece of code $\mathcal{C}$, namely $\mathsf{gen}{::}\mathcal{D} \rightarrow \mathcal{C}$. By applying gen to a specific data type, the appropriate code will be generated.

2.1 Data Type Fragments

Rather than infer the entire gen function in one step, we instead infer many functions from $\mathcal{D}$ to each element of $\mathcal{C}$'s abstract syntax tree, then combine them to form gen. Each smaller function may depend only on some fragment of the data type – for example one particular line may depend on one particular data constructor, rather than the entire data type. A fragment of a data type may be one of the following:

A constant: A node of the abstract syntax tree can be a constant, meaning that in all cases it will generate the same value. In the Eq example things such as True, $(\equiv)$, $(\wedge)$ etc. are constant. Anything which does not have an alternative generation function is assumed to be constant.

A number: The code x_1 is parameterised by the number 1. We use the notation $1 \mapsto x[\![\#]\!]$ to denote the function, where # is the parameterised number, and 1 is the parameter. Any detected number is either a parameterised value, or a constant.

A constructor: The code Third is parameterised by the constructor Third, being the name of the constructor. We use the notation $\mathsf{Third} \mapsto [\![\mathsf{ctorname}]\!]$ to denote this function.

A data type: In the first example the whole code is parameterised by the data type. One particular place parameterised directly on the data type is Eq DataName, which becomes $\mathsf{DataName} \mapsto \mathsf{Eq}\ [\![\mathsf{dataname}]\!]$. If the name of the data type changes to WritingImplement, then the function will generate Eq WritingImplement instead.

2.2 The Map Pattern

The examples shown previously map data type fragments to fixed-sized portions of the abstract syntax tree. But consider the pattern (Third x_1 x_2), we require n variables, where n is the arity of the constructor. To solve this problem, we can construct a generalised map from a list of generation functions. This construction requires two conditions: (1) one item of the list has a function which generates all abstract syntax trees in the list; (2) their parameter values are sequential – either consecutive integers or constructors in their definition order.

$[1 \mapsto x[\![\#]\!], 2 \mapsto x[\![\#]\!]]$

Take the above list of functions. Picking either function, when applying it to the other parameter, we obtain the correct value. The parameters are clearly sequential. We can rewrite this list using a MAP as:

$2 \mapsto [\![\textsc{map}\ [1..\#]\ (x[\![\#]\!])]\!]$

MAP takes two arguments, a range of items to use as the parameter, and an expression. Within the second argument, # is bound to each number in the range in turn. The resultant function is parameterised by the highest number in the sequence. We can also have a MAP which operates over constructors, in which case the resultant MAP is parameterised by the entire data type.

The following two examples *cannot* be generalised to a MAP:

$[1 \mapsto x[\![\#]\!], 3 \mapsto x[\![\#]\!]]$
$[1 \mapsto x[\![\#]\!], 2 \mapsto y[\![\#]\!]]$

In the first, the parameters are not sequential. In the second, neither function produces the correct result on both items.

The MAP pattern provides the key power to generalising an example from one specific data type, to any data type. In order to increase the chance of finding a suitable generalisation, we allow each node in the abstract syntax tree to be represented by a list of possible parameterised functions [11].

2.3 The Fold Pattern

One common pattern in programming is the *fold* [4]. Consider the Eq example from before. If we move to using prefix notation for $(\wedge)$ and explicit bracketing we obtain:

$(\mathsf{Third}\ x_1\ x_2) \equiv (\mathsf{Third}\ y_1\ y_2) = (\wedge)\ (x_1 \equiv y_1)\ ((\wedge)\ (x_2 \equiv y_2)\ \mathsf{True})$
-- or, expressed as a fold
$(\mathsf{Third}\ x_1\ x_2) \equiv (\mathsf{Third}\ y_1\ y_2) = \mathsf{foldr}\ (\wedge)\ \mathsf{True}\ [x_1 \equiv y_1, x_2 \equiv y_2]$

The original structure is not a list, and would not be found as a MAP. However, we can convert this function to a fold, recovering a list structure. A fold takes a list, and replaces each cons by a function (i.e. $(\wedge)$) and the nil by a value (i.e. True). By automatically converting the code to use fold, a MAP is recovered. Using a FOLDR rule (fold associating to the right), the result of the right hand side is:

$2 \mapsto [\![\textsc{foldr}\ (\wedge)\ \mathsf{True}\ [\![\textsc{map}\ [1..\#]\ (x[\![\#]\!] \equiv y[\![\#]\!])]\!]]\!]$

The FOLDR rule is a foldr function, but applied at generation time. The fold encoding explains the redundant $\wedge$ True at the end of each line in the Eq example. In order for the base case (True) to share the same fold definition as above, we require a redundant conjunct. In the code generated for instances $\wedge$ True is removed, using algebraic simplification.

2.4 Eliminating Number Parameters

Initially functions may be parameterised by the whole data type, constructors or numbers. The MAP rule converts a list of functions parameterised by constructors to a function parameterised by the whole data type, but leaves numbers parameterised by numbers. A number parameter is replaced by a constructor with a relationship to that number. Taking the example of x_1 x_2 from earlier, we can write any of:

```
2      ↦ ⟦MAP [1 .. #         ] (x⟦#⟧)⟧
Third  ↦ ⟦MAP [1 .. ctorarity ] (x⟦#⟧)⟧
Third  ↦ ⟦MAP [1 .. ctorindex] (x⟦#⟧)⟧
Fourth ↦ ⟦MAP [1 .. ctorarity ] (x⟦#⟧)⟧
```

Instead of parameterising by the number 2, we can parameterise by a constructor. The number 2 can be obtained in three ways: it is both the arity, and zero-based index of Third, and also the arity of Fourth.

2.5 Combining Generation Functions

The calculation of generation functions proceeds in a bottom-up manner. Once all the children of a node in the abstract syntax tree have generation functions, they are combined to form a generation function for the whole node. A set of child generation functions can only be combined if all the non-constant functions share the same parameter. For example:

```
(Third ↦ ⟦ctorname⟧) (Third ↦ ⟦MAP [1 .. ctorarity] (x⟦#⟧)⟧)
  -- becomes
Third ↦ ⟦ctorname⟧ ⟦MAP [1 .. ctorarity] (x⟦#⟧)⟧
```

2.6 The Result

After applying the rules, the resultant function for the Eq example is:

```
instance Eq ⟦dataname⟧ where
  ⟦MAP ctors (
    (⟦ctorname⟧ ⟦MAP [1 .. ctorarity] (x⟦#⟧)) ≡
    (⟦ctorname⟧ ⟦MAP [1 .. ctorarity] (y⟦#⟧)) =
    ⟦FOLDR (∧) True ⟦MAP [1 .. ctorarity] (x⟦#⟧ ≡ y⟦#⟧)⟧⟧
  )⟧
  _ ≡ _ = False
```

Applying this function to WritingImplement will produce the instance we originally specified. Writing the generation function directly poses a number of difficulties:

1. There is no available language in which to write a generator. The DrIFT preprocessor [13] allows a generator to be specified, but requires the author to learn many new library functions.
2. Details such as the associativity of (∧) can be omitted from an example, but are required when expressing a FOLDR directly.
3. Manually written generators may not produce type-safe instances.
4. The results of a manual generator require testing against examples.
5. The complexity, compared to a single example, is much higher.

3 Limitations of Automatic Derivation

The instance generation scheme given is not complete – there exist instances whose generator cannot be determined. The Derive tool [9] is a program for generating instances for user defined data types. Of the 24 instances supported by the Derive tool, 15 are expressed by example, while 9 require manually written instance generators. There are several reasons some instances cannot be determined:

Non-inductive definitions: For example, the Binary class serialises a value to disk. For each value, a tag is written to indicate the constructor. If a data type has only one constructor, the tag is omitted. These instances are not inductive – the single constructor does not follow the same pattern.

Type-based definitions: For example, the Monoid class requires items of the same type to be processed using mappend, but items of a different type use mempty. Automatic derivation has no notion of type-specific behaviour.

Record-based definitions: Haskell provides records, which allow fields to be labelled. The Show class outputs the field name if present, but the examples have no notion of label-specific behaviour. By extending DataName, record definitions could be determined, but this change would increase the complexity of all other example instances.

4 Generation of Standard Classes

Many instance generators can be expressed by example – including some from the standard Haskell libraries (Enum, Ord, Bounded) and publicly distributed libraries (Serial, Arbitrary). The Data class was introduced in Scrap Your Boilerplate [6], and allows Haskell programmers to write concise queries and transformations. The fundamental operation is gfoldl, which involves a fold over each value, and the application of an argument to join the fields. An example instance can be given as:

```
instance Data DataName where
  gfoldl k r (First        ) = r First
  gfoldl k r (Second x1    ) = r Second `k` x1
  gfoldl k r (Third  x1 x2 ) = r Third  `k` x1 `k` x2
  gfoldl k r (Fourth x1 x2 ) = r Fourth `k` x1 `k` x2
```

The generator function is inferred as:

```
instance Data ⟦dataname⟧ where
  ⟦MAP ctors (
    gfoldl k r
      (⟦ctorname⟧ ⟦MAP [1..ctorarity] (x⟦#⟧)⟧) =
      ⟦FOLDR k (r ⟦ctorname⟧)
        ⟦MAP [1..ctorarity] (x⟦#⟧)⟧
      ⟧
  )⟧
```

5 Related Work

The purpose of this work is to find a pattern, and generalise that pattern to other situations. Genetic algorithms [2] are often used to automatically find a pattern in a data set. Genetic algorithms work by evolving a hypothesis (a gene sequence) and testing on a sample problem. They are well suited to search problems where the utility function is continuous – close hypotheses have similar fitness. The main difference from this paper is that the hypothesis is random, whereas ours is strongly directed by the shape of the example.

The area of optical character recognition [5] has some similar characteristics – a page is analysed to look for common patterns (pictures or text passages), which can be processed further. This is related to the process of using the fold pattern (§2.3), where a repeating pattern is detected. The difference is that character recognition works on image data, which does not have the same precision as program code.

The closest work we are aware of is that of the theorem proving community. Induction is a very common tactic for writing proofs, and well supported in systems such as HOL Light [3]. Typically the user must suggest the use of induction, which the system checks for validity. Automatic inference of an induction argument has been tried [8], but is rarely successful.

The concepts in this paper are applicable outside the domain of instances in Haskell. Any programming language operation that exhibits some degree of uniformity could be automated. To give one example: the object-orientated community have embraced design patterns [1], which involve many recurring patterns.

6 Conclusions and Future Work

We have presented a mechanism for automatically deriving instance generators for Haskell type classes. Our technique has been implemented in the Derive tool [9], where 60% of instance generators are specified by example. The ease of creating new instances has enabled several users to contribute instance generators to the Derive tool. We see several lines of future work:

- Using automatic instance generation allows the underlying tool to change the API for specifying instances, without requiring human intervention to modify the generators – they can simply be regenerated. This freedom allows instances to be expressed in new ways. Currently an instance is a fragment of compile time code, but using Haskell's reflection capabilities [7], instances could be derived at run-time, removing the inconvenience of a separate preprocessor.
- The provided data type (Figure 1) allows many instances to be inferred – but more would be desirable. One approach to specifying more instances would be to augment the existing data type with additional features, such as record names (see §3). An alternative approach would be to introduce new data types with features specifically targeted for certain types of definition. Care

would have to be taken to ensure that these extensions do not substantially increase the complexity of writing examples.

Computers are ideally suited to applying repetitive patterns, but specifying these patterns can be complex and error prone. By specifying the result, instead of the pattern, a user can focus on what they want, rather than the mechanism by which this is realized.

Acknowledgements Thanks to Matt Naylor and Chris Smith for helpful suggestions on the presentation of this work. Thanks to Stefan O'Rear for work on the Derive tool.

References

1. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Professional, 1995.
2. David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley Professional, January 1989.
3. John Harrison. HOL light: A tutorial introduction. In Mandayam Srivas and Albert Camilleri, editors, *Proc. Formal Methods in Computer-Aided Design, FMCAD'96*, volume 1166 of *LNCS*, pages 265–269. Spinger-Verlag, 1996.
4. Graham Hutton. A tutorial on the universality and expressiveness of fold. *JFP*, 9(4):355–372, July 1999.
5. S Impedovo, L Ottaviano, and S Occhinegro. Optical character recognition – A survey. *International Journal of Pattern Recognition and Artificial Intelligence*, 5:1–24, 1991.
6. Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *Proc. TLDI '03*, 38(3):26–37, March 2003.
7. Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proc. ICFP '04*, pages 244–255. ACM Press, 2004.
8. Sava Mintchev. Mechanized reasoning about functional programs. In K. Hammond, D. N. Turner, and P. M. Sansom, editors, *Functional Programming*, pages 151–166. Springer, Berlin, Heidelberg, 1994.
9. Neil Mitchell and Stefan O'Rear. Derive - project home page. `http://www.cs.york.ac.uk/~ndm/derive/`, March 2007.
10. Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, 2003.
11. Philip Wadler. How to replace failure by a list of successes. In *Proc. FPCA '85*, pages 113–128. Springer-Verlag, 1985.
12. Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. POPL '89*, pages 60–76, New York, NY, USA, 1989. ACM Press.
13. Noel Winstanley. Reflections on instance derivation. In *1997 Glasgow Workshop on Functional Programming*, September 1997.