“Scrap Your Boilerplate” Reloaded

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

[1] Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
{ralf,loeh}@informatik.uni-bonn.de
[2] Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
bruno@comlab.ox.ac.uk

Abstract. The paper “Scrap your boilerplate” (SYB) introduces a combinator library for generic programming that offers generic traversals and queries. Classically, support for generic programming consists of two essential ingredients: a way to write (type-)overloaded functions, and independently, a way to access the structure of data types. SYB seems to lack the second. As a consequence, it is difficult to compare with other approaches such as PolyP or Generic Haskell. In this paper we reveal the structural view that SYB builds upon. This allows us to define the combinators as generic functions in the classical sense. We explain the SYB approach in this changed setting from ground up, and use the understanding gained to relate it to other generic programming approaches. Furthermore, we show that the SYB view is applicable to a very large class of data types, including generalized algebraic data types.

1 Introduction

The paper “Scrap your boilerplate” (SYB) [1] introduces a combinator library for generic programming that offers generic traversals and queries. Classically, support for generic programming consists of two essential ingredients: a way to write (type-)overloaded functions, and independently, a way to access the structure of data types. SYB seems to lacks the second, because it is entirely based on combinators.

In this paper, we make the following contributions:

– We explain the SYB approach from ground up using an explicit representa-

tion of data types, the *spine view*. Many of the SYB library functions are more easily defined in the spine view than using the combinators underlying the original SYB library.
– We compare the expressive power and applicability of the spine view to the original SYB paper, to PolyP [2] and to Generic Haskell [3, 4].
– Furthermore, we show that the SYB view is applicable to a very large class of data types, including generalized algebraic data types (GADTs) [5, 6].

We use Haskell [7] for all our examples. The source code of this paper [8] constitutes a Haskell program that can be compiled by GHC [9] in order to

test and experiment with our implementation. While our implementation is not directly usable as a separate library, because it is not extensible (new data types cannot be added in a compositional way), this deficiency is not tied to the idea of the *Spine* view: the technical report version of this paper [8] contains a slightly less elegant implementation that is extensible.

In this introduction, we explain the ingredients of a system for generic programming, and argue that the original SYB presentation does not clearly qualify as such a system. In order to better understand the concept of generic programming, let us first look at plain functional programming.

1.1 Functional programming and views

As functional programmers in a statically typed language, we are used to define functions by case analysis on a data type. In fact, it is standard practice to define a function on a data type by performing case analysis on the input. The shape of the data type guides our function definitions, and affects how easy it is to define certain functions.

As an example, assume we want to implement a priority queue supporting among others the operation

$$splitMinimum :: PriorityQueue \rightarrow Maybe\ (Int, PriorityQueue)$$

to separate the minimum from the remaining queue if the queue is not empty. We can choose a heap-structured tree to implement the priority queue, and define

$$\begin{aligned}&\textbf{data}\ \mathit{Tree}\ a &&= \mathit{Empty} \mid \mathit{Node}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a)\\ &\textbf{type}\ \mathit{PriorityQueue} &&= \mathit{Tree}\ \mathit{Int}\ \ .\end{aligned}$$

The choice of a heap as the underlying data stucture makes the implementation of *splitMinimum* slightly tricky, requiring an auxiliary operation to merge two heaps.

If, on the other hand, we choose a sorted list to represent the priority queue

$$\textbf{data}\ \mathit{PriorityQueue} = \mathit{Void} \mid \mathit{Min}\ \mathit{Int}\ \mathit{PriorityQueue}\ \ ,$$

we make our life much easier, because *splitMinimum* is now trivial to define. The price we pay is that the implementation on lists is likely to be less efficient than the one using the tree. Such different *views* on a data structure need not be mutually exclusive. Wadler and others have proposed language support for views [10, 11].

Many functions on a single data type follow common traversal and recursion patterns. Instead of defining each function by case analysis, it is possible to define combinators that capture these patterns. For instance, given functions

$$\begin{aligned}&\mathit{foldTree} :: r \to (r \to a \to r \to r) \to \mathit{Tree}\ a \to r\\ &\mathit{mapTree} :: (a \to b) \to \mathit{Tree}\ a \to \mathit{Tree}\ b\ \ ,\end{aligned}$$

we can write functions to perform an inorder traversal of the tree or to increase every label in a tree by one very concisely:

$$\begin{aligned}&\mathit{inorder} = \mathit{foldTree}\ [\,]\ (\lambda l\ x\ r \to l +\!\!+ [x] +\!\!+ r)\\ &\mathit{incTree} = \mathit{mapTree}\ (+1)\ \ .\end{aligned}$$

1.2 Generic programming

A *generic function* is a function that is defined once, but works for many data types. It can adapt itself to the structure of data types. Generic functions are also called *polytypic* or *structurally polymorphic*.

Genericity is different from *parametric polymorphism*, where the same code works for multiple types, and the structure of a data type is not available for analysis. It is also more specific than *ad-hoc polymorphism*, which allows a function to be defined for different data types, by providing one implementation for each type.

Typical examples of generic functions are equality or comparison, parsing and unparsing, serialization, traversals over large data structures and many others.

Support for generic programming consists of two essential ingredients. Firstly, support for ad-hoc polymorphism is required. This allows the programmer to write *overloaded functions*, i.e., functions that dispatch on a type argument. Secondly, we need a *generic view* on the structure of data types. In a nominal type system, types with similar structure are considered to be completely distinct. To employ generic programming, we need to lower this barrier and make the structure transparent if desired.

The two ingredients are orthogonal, and for both, there is a choice. Overloaded functions can be expressed in Haskell using the class system, using a type-safe cast operation, by reflecting the type system on the value level, or by a combination of the above. Any of these approaches has certain advantages and disadvantages, but they are mostly interchangeable and do not dramatically affect the expressivity of the generic programming system.

The structural view, on the other hand, dictates the flavour of the whole system: it affects the set of data types we can represent in the view, the class of functions we can write using case analysis on the structure, and potentially the efficiency of these functions. The structural view is used to make an overloaded function truly generic, working for a data type even if it has no ad-hoc case for that type.

For instance, PolyP views data types as fixed points of regular functors. Therefore its approach is limited to regular data types, but the view allows access to the points of recursion and allows the definition of recursion combinators such as catamorphisms. Generic Haskell uses a sum-of-products view which is more widely applicable, but limits the class of functions we can write. The concept of generic views is explained further in a recent paper [12], and is related to *universes* in dependently-typed programming [13].

In summary, it turns out that there is a close analogy between plain functional and generic programming: the concepts of views, function definition by case analysis, and combinators occur in both settings.

1.3 Scrap your boilerplate

In analogy with the situation on plain functions, not all generic functions are defined by case analysis. Just as there are powerful combinators for ordinary functions, such combinators also exist for generic programming. In fact, the very combinators we have used above, *foldTree* and *mapTree*, are typical candidates for generalization.

The paper "Scrap your boilerplate" (SYB) describes a library for *strategic programming* [14], i.e., it offers combinators for generic traversals and queries on terms. Two central combinators of the SYB library are *everywhere* to traverse a data structure and modify it in certain places, and *everything* to traverse a data structure and collect information in the process.

The SYB approach builds completely on combinators, and some fundamental combinators are assumed to be provided by the implementation. While this is fine in practice, it makes it difficult to compare SYB with other approaches such as PolyP or Generic Haskell. The reason is that the concept of a generic view seems to be missing. Functions are never defined by case analysis on the structure of types.

However, the generic view is only hidden in the original presentation. In this paper we reveal the structure that SYB uses behind the scenes and that allows us to define the SYB combinators as generic functions by case analysis on that structure.

We will explain the SYB approach in this changed setting from ground up. The focus of the presentation is on conceptual conciseness. We do not strive to replace the original implementation, but to complement it by an alternative implementation which may be easier to understand and relate to other approaches.

1.4 Organization of this paper

The rest of this paper is organized as follows: We first describe the two orthogonal ingredients required for generic programming in our presentation of the SYB approach: overloaded functions (Section 2) and the *spine view*, the structure that is the hidden foundation of SYB (Section 3). We then review the central combinators of SYB in Section 4. Section 5 shows how we can access names of constructors.

In Section 6, we take a step back and relate SYB to other generic programming approaches. Inspired by our analysis on the expressiveness of the SYB approach, we demonstrate how to extend the spine view to generalized algebraic data types (Section 7). Section 8 discusses related work and concludes.

2 Overloaded functions

The standard way in Haskell to express an overloaded function is to use a type class. In fact, this is the way taken by the original SYB papers: in the first SYB paper, type classes are used in conjunction with a type-safe cast operation, and in the third paper, overloaded functions are expressed solely based on type classes. However, type classes leave it to the compiler to find the correct instance, and thus hide a non-trivial aspect of the program. In this paper, we prefer to be more explicit and emphasize the idea that an overloaded function dispatches on a type argument. Haskell excels at embedded languages, so it seems a good idea to try to embed the type language in Haskell. The following way to encode overloaded functions is not new: it is based on Hinze's "Fun of Programming" chapter [15] and has been used widely elsewhere [16].

The whole point of static types is that they can be used at compile time to distinguish programs, hence we certainly do not want to use an unparameterized data type *Type* to represent types. Instead, we add a parameter so that *Type* t comprises only type representations for the type t. We now need ways to construct values of type *Type* t. For instance, *Int* can be a representation of the type *Int*, so that we have *Int* :: *Type Int*. Similarly, if we have a representation r of type a, we can make *List* r a representation of type $[a]$, or formally *List* :: *Type* $a \to$ *Type* $[a]$.

The notation we use suggests that *Int* and *List* are data constructors of type *Type*, but this impossible in Haskell 98, because the result type of a constructor must always be unrestricted, i.e., *Type* a for some type variable a. Fortunately, GHC now supports *generalized algebraic data types* (GADTs) [5, 6], which lift exactly this restriction. Therefore, we can indeed define *Type* in Haskell using the following GADT:

```
data Type :: * → * where
  Int  :: Type Int
```

$$
\begin{aligned}
&\mathit{Char} :: \mathit{Type}\ \mathit{Char}\\
&\mathit{List}\ \ :: \mathit{Type}\ a \to \mathit{Type}\ [a]\\
&\mathit{Pair}\ :: \mathit{Type}\ a \to \mathit{Type}\ b \to \mathit{Type}\ (a, b)\\
&\mathit{Tree}\ :: \mathit{Type}\ a \to \mathit{Type}\ (\mathit{Tree}\ a)\ \ .
\end{aligned}
$$

This type allows us to represent integers, characters, lists, pairs, and trees – enough to give an example of a simple overloaded function that sums up all integers in a value:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{sum} :: \mathit{Type}\ a \to a \to \mathit{Int}}\\
\mathit{sum}\ \mathit{Int} & n & = n\\
\mathit{sum}\ \mathit{Char} & \_ & = 0\\
\mathit{sum}\ (\mathit{List}\ a) & \mathit{xs} & = \mathit{foldr}\ (+)\ 0\ (\mathit{map}\ (\mathit{sum}\ a)\ \mathit{xs})\\
\mathit{sum}\ (\mathit{Pair}\ a\ b) & (x, y) & = \mathit{sum}\ a\ x + \mathit{sum}\ b\ y\\
\mathit{sum}\ (\mathit{Tree}\ a) & t & = \mathit{sum}\ (\mathit{List}\ a)\ (\mathit{inorder}\ t)\ \ .
\end{array}
$$

The function *sum* works on all types that can be constructed from *Int*, *Char*, [], (,), and *Tree*, for instance, on a complex type such as $[(\mathit{Char}, \mathit{Int})]$: the expression

$$\mathit{sum}\ (\mathit{List}\ (\mathit{Pair}\ \mathit{Char}\ \mathit{Int}))\ [(\texttt{'k'}, 6), (\texttt{'s'}, 9), (\texttt{' '}, 27)]$$

evaluates to 42.

The function *sum* is an example of an ad-hoc-polymorphic function. There are a limited number of cases for different types, defining potentially unrelated behavior of *sum* for these types. The function will not work on types such as *Bool* or *Maybe* or even on a type

$$\textbf{newtype}\ \mathit{MyPair}\ a\ b = \mathit{MyPair}\ (a, b)\ \ ,$$

because Haskell has a nominal type system, hence $\mathit{MyPair}\ a\ b$ is isomorphic to, yet distinct from (a, b).

3 The spine view

In this section, we learn how to define a truly generic *sum*, which works on *Bool* and *Maybe* and *MyType*, among others.

Take a look at any Haskell value. If it is not of some abstract type, it can always be written as a data constructor applied to other values. For example, *Node Empty* 2 *Empty* is the *Node* data constructor, applied to the three values *Empty*, 2, and *Empty*. Even built-in types such as *Int* or *Char* are not fundamentally different: every literal can be seen as a nullary constructor.

Let us make the structure of constructed values visible and mark each constructor using *Constr*, and each function application using $\diamond$. The example from above becomes $\mathit{Constr}\ \mathit{Node} \diamond \mathit{Empty} \diamond 2 \diamond \mathit{Empty}$. The functions *Constr* and $(\diamond)$[3] are themselves constructors of a new data type *Spine*:[4]

$$
\begin{array}{l}
\textbf{data}\ \mathit{Spine} :: * \to *\ \textbf{where} \\
\quad \mathit{Constr} :: a \to \mathit{Spine}\ a \\
\quad (\diamond) \qquad :: \mathit{Spine}\ (a \to b) \to a \to \mathit{Spine}\ b\ \ .
\end{array}
$$

Given a value of type *Spine a*, we can recover the original value of type *a* by undoing the conversion step made before:

$$
\begin{array}{l}
\mathit{fromSpine} :: \mathit{Spine}\ a \to a \\
\mathit{fromSpine}\ (\mathit{Constr}\ c) = c \\
\mathit{fromSpine}\ (f \diamond a) \qquad = (\mathit{fromSpine}\ f)\ a\ \ .
\end{array}
$$

[3] We use $(\diamond)$ as a symbol for an infix data constructor. For our presentation, we ignore the Haskell rule that names of infix data constructors must start with a colon.

[4] Note that in contrast to *Type*, the data type *Spine* is not necessarily a *generalized* algebraic data type. The result types of the constructors are not restricted, *Spine* could therefore be defined in GHC as a normal data type with existentials. However, we prefer the GADT syntax.

The function *fromSpine* is parametrically polymorphic, i.e., it works independently of the type in question: it just replaces *Constr* with the original constructor and $(\diamond)$ with function application.

Unfortunately, *fromSpine* is the only interesting function we can write on a *Spine*. Reconsider the type of the $(\diamond)$ constructor:

$$
(\diamond) :: \mathit{Spine}\ (a \to b) \to a \to \mathit{Spine}\ b\ \ .
$$

The type *a* is not visible in the final result (it is existentially quantified in the data type), so the only thing we can do with the component of type *a* is to combine it somehow with the component of type $Spine\ (a \to b)$.

Since we intend to call overloaded functions on the value of type a, we require a representation of the type of a. Our solution is thus that together with the value of type a, we store a representation of its type. To this end, we introduce a data type for typed values[5]

$$\mathbf{data}\ \mathit{Typed}\ a = a : \mathit{Type}\ a\ \ ,$$

and then adapt $(\diamond)$ to use $\mathit{Typed}\ a$ instead of a:

$$\begin{array}{l}
\mathbf{data}\ \mathit{Spine} :: * \to *\ \mathbf{where} \\
\quad \mathit{Constr} :: a \to \mathit{Spine}\ a \\
\quad (\diamond) \qquad :: \mathit{Spine}\ (a \to b) \to \mathit{Typed}\ a \to \mathit{Spine}\ b\ \ .
\end{array}$$

Of course, we have to adapt *fromSpine* to ignore the new type annotations:

$$\begin{array}{l}
\mathit{fromSpine} :: \mathit{Spine}\ a \to a \\
\mathit{fromSpine}\ (\mathit{Constr}\ c)\ \ = c \\
\mathit{fromSpine}\ (f \diamond (a : _)) = (\mathit{fromSpine}\ f)\ a\ \ .
\end{array}$$

We can define a right inverse to *fromSpine*, as an overloaded function. For each data type, the definition follows a trivial pattern. Here are the cases for the *Int* and *Tree* types:

$$\begin{array}{lll}
\mathit{toSpine} :: \mathit{Type}\ a \to a \to \mathit{Spine}\ a & & \\
\mathit{toSpine}\ \mathit{Int} & n & = \mathit{Constr}\ n \\
\mathit{toSpine}\ (\mathit{Tree}\ a) & \mathit{Empty} & = \mathit{Constr}\ \mathit{Empty} \\
\mathit{toSpine}\ (\mathit{Tree}\ a) & (\mathit{Node}\ l\ x\ r) & = \mathit{Constr}\ \mathit{Node} \\
 & & \quad \diamond (l : \mathit{Tree}\ a) \diamond (x : a) \diamond (r : \mathit{Tree}\ a)\ \ .
\end{array}$$

With all the machinery in place, we can now write the truly generic *sum*:

$$\begin{array}{l}
\mathit{sum} :: \mathit{Type}\ a \to a \to \mathit{Int} \\
\mathit{sum}\ \mathit{Int}\ n = n \\
\mathit{sum}\ t \quad x = \mathit{sum}'\ (\mathit{toSpine}\ t\ x)
\end{array}$$

[5] We use (:) as data constructor for the type *Typed* in this paper. The *cons*-operator for lists, written (:) in Haskell, does not occur in this paper.

$$
\begin{aligned}
&sum' :: Spine\ a \to Int\\
&sum'\ (Constr\ c) \ = 0\\
&sum'\ (f \diamond (x : t)) = sum'\ f + sum\ t\ x \ \ .
\end{aligned}
$$

This function requires only a single type-specific case, namely the one for *Int*. The reason is that we want to do something specific for integers, which does not follow the general pattern, whereas the formerly explicit behavior for the types *Char*, [], (,), and *Tree* is now completely subsumed by the function sum'. Note also that in the last line of sum', the type information t for x is indispensable, as we call the generic function *sum* recursively.

Why are we in a better situation than before? If we encounter a new data type such as *Maybe*, we still have to extend the representation type with a constructor $Maybe :: Type\ a \to Type\ (Maybe\ a)$ and provide a case for the constructor *Maybe* in the *toSpine* function. However, this has to be done only once per data type, and it is so simple that it could easily be done automatically. The code for the generic functions (of which there can be many) is completely unaffected by the addition of a new data type.

4 Generic queries and traversals

In this section, we implement the two central SYB combinators *everything* and *everywhere* that are used to construct generic queries and traversals.

4.1 Generic queries

A query is an overloaded function that returns a result of a specific type:

$$\mathbf{type}\ Query\ r = \forall a. Type\ a \to a \to r \ \ .$$

We have already seen an example of a query, namely the *sum* function from Section 3. There are many more applications of queries: computation of the size of a structure, collection of names, collection of free variables, building a finite map, finding a specific element etc.

If we look back at the generic *sum* function, we see that it performs several tasks at once, leading to a relatively complex definition: integers are preserved, while in general, constructors are replaced by 0; the subresults are added; finally, a recursive traversal is performed over the entire data structure.

In the following, we describe how to separate these different activities into different functions, and at the same time abstract from the specific problem of summing up values.

If we already have a query, we can define a derived query that applies the original query to all immediate children of a given constructor:

$$
\begin{aligned}
&\mathit{mapQ} :: \mathit{Query}\ r \rightarrow \mathit{Query}\ [r] \\
&\mathit{mapQ}\ q\ t = \mathit{mapQ}'\ q \circ \mathit{toSpine}\ t \\
&\mathit{mapQ}' :: \mathit{Query}\ r \rightarrow (\forall a. \mathit{Spine}\ a \rightarrow [r])
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{mapQ}'\ q\ (\mathit{Constr}\ c) &&= [\,] \\
&\mathit{mapQ}'\ q\ (f \diamond (x : t)) &&= \mathit{mapQ}'\ q\ f +\!\!+ [q\ t\ x]\ .
\end{aligned}
$$

The results of the original query q are collected in a list. The combinator *mapQ* does not traverse the input data structure. The traversal is the job of *everything′*, which is defined in terms of *mapQ*:

$$
\begin{aligned}
&\mathit{everything}' :: \mathit{Query}\ r \rightarrow \mathit{Query}\ [r] \\
&\mathit{everything}'\ q\ t\ x = [q\ t\ x] +\!\!+ \mathit{concat}\ (\mathit{mapQ}\ (\mathit{everything}'\ q)\ t\ x)\ .
\end{aligned}
$$

Here, we apply the given query q to the entire argument x, and then recurse for the immediate children. The SYB version of *everything* fuses *everything′* with an application of *foldl1*, using a binary operator to combine all the elements of the nonempty list returned by *everything′*:

$$
\begin{aligned}
&\mathit{everything} :: (r \rightarrow r \rightarrow r) \rightarrow \mathit{Query}\ r \rightarrow \mathit{Query}\ r \\
&\mathit{everything}\ \mathit{op}\ q\ t\ x = \mathit{foldl1}\ \mathit{op}\ ([q\ t\ x] +\!\!+ \mathit{mapQ}\ (\mathit{everything}\ \mathit{op}\ q)\ t\ x)\ .
\end{aligned}
$$

In order to express the query *sum* in terms of *everything*, we need a simple query *sumQ* expressing that we want to count integers:

$$
\begin{aligned}
&\mathit{sumQ} :: \mathit{Query}\ \mathit{Int} \\
&\mathit{sumQ}\ \mathit{Int}\ n = n \\
&\mathit{sumQ}\ t\quad x = 0 \\
&\mathit{sum} :: \mathit{Query}\ \mathit{Int} \\
&\mathit{sum} = \mathit{everything}\ (+)\ \mathit{sumQ}\ .
\end{aligned}
$$

4.2 Generic traversals

While a query computes an answer of a fixed type from an input, a traversal is an overloaded function that preserves the type of its input:

$$\textbf{type}\ \mathit{Traversal} = \forall a.\mathit{Type}\ a \to a \to a\ .$$

The counterpart of $mapQ$ is $mapT$. It applies a given traversal to the immediate children of a constructor, then rebuilds a value of the same constructor from the results:

$$\begin{aligned}
&mapT :: \mathit{Traversal} \to \mathit{Traversal}\\
&mapT\ h\ t = \mathit{fromSpine} \circ mapT'\ h \circ \mathit{toSpine}\ t\\
&mapT' :: \mathit{Traversal} \to (\forall a.\mathit{Spine}\ a \to \mathit{Spine}\ a)\\
&mapT'\ h\ (\mathit{Constr}\ c)\ \ = \mathit{Constr}\ c\\
&mapT'\ h\ (f \diamond (x : t)) = mapT'\ h\ f \diamond (h\ t\ x : t)\ .
\end{aligned}$$

The function $mapT$ not only consumes a value of the type argument, but also produces one. Therefore we call not only *toSpine* on the input value, but also *fromSpine* before returning the result. The calls to *fromSpine* and *toSpine* are

determined by the type of the generic function that is defined. The general principle is described elsewhere [3, Chapter 11].

Using $mapT$, we can build bottom-up or top-down variants of *everywhere*, which apply the given traversal recursively:

$$\begin{aligned}
&\mathit{everywhere}_{\mathrm{BU}} :: \mathit{Traversal} \to \mathit{Traversal}\\
&\mathit{everywhere}_{\mathrm{BU}}\ f\ t = f\ t \circ mapT\ (\mathit{everywhere}_{\mathrm{BU}}\ f)\ t\\
&\mathit{everywhere}_{\mathrm{TD}} :: \mathit{Traversal} \to \mathit{Traversal}\\
&\mathit{everywhere}_{\mathrm{TD}}\ f\ t = mapT\ (\mathit{everywhere}_{\mathrm{TD}}\ f)\ t \circ f\ t\ .
\end{aligned}$$

There are many applications of traversals, such as renaming variables in an abstract syntax tree, annotating a structure with additional information, optimizing or simplifying a structure etc. Here is a simplified example of a transformation performed by the Haskell refactorer HaRe [17], which rewrites a Haskell **if** construct into an equivalent **case** expression according to the rule

$$\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2 \quad \leadsto \quad \textbf{case } e \textbf{ of } \mathit{True} \to e_1; \mathit{False} \to e_2 \enspace .$$

We assume a suitable abstract syntax for Haskell. The rewrite rule is captured by the traversal

$$\begin{array}{l}
\mathit{ifToCaseT} :: \mathit{Traversal} \\
\mathit{ifToCaseT}\ \mathit{HsExp}\ (\mathit{HsIf}\ e\ e_1\ e_2) = \\
\quad \mathit{HsCase}\ e\ [\mathit{HsAlt}\ (\mathit{HsPLit}\ (\mathit{HsBool}\ \mathit{True}))\ e_1, \\
\qquad\qquad\quad \mathit{HsAlt}\ (\mathit{HsPLit}\ (\mathit{HsBool}\ \mathit{False}))\ e_2] \\
\mathit{ifToCaseT}\ _ \qquad e \qquad\qquad\qquad = e \enspace .
\end{array}$$

The traversal can be applied to a complete Haskell program using

$$\mathit{ifToCase} = \mathit{everywhere}_{\mathrm{BU}}\ \mathit{ifToCaseT} \enspace .$$

5 Generically showing values

We have seen that we can traverse data types in several ways, performing potentially complex calculations in the process. However, we cannot reimplement Haskell's *show* function, even though it looks like a *Query String*. The reason is that there is no way to access the name of a constructor. We have a case for constructors, *Constr*, in our *Spine* data type, but there is really not much we can do at this point. So far, we have either invented a constant value ([] in the case of *mapQ*), or applied the constructor itself again (in the case of *mapT*).

But it is easy to provide additional information for each constructor. When we define *toSpine* for a specific data type, whether manually or automatically, we have information about the constructors of the data type available, so why not use it? Let us therefore modify *Spine* once more:

$$\begin{array}{l}
\textbf{data}\ \mathit{Spine} :: * \to * \ \textbf{where} \\
\quad \mathit{As} :: a \to \mathit{ConDescr} \to \mathit{Spine}\ a \\
\quad (\diamond) :: \mathit{Spine}\ (a \to b) \to \mathit{Typed}\ a \to \mathit{Spine}\ b \enspace .
\end{array}$$

We have renamed *Constr* to *As*, as we intend to use it as a binary operator which takes a constructor function and information about the constructor. In this paper, we use only the name to describe a constructor,

$$\textbf{type}\ ConDescr = String\ \ ,$$

but we could include additional information such as its arity, the name of the type, the "house number" of the constructor and so on. Adapting *Spine* means that the generation of *toSpine* has to be modified as well. We show as an example how to do this for the type *Tree*:

$$\begin{aligned}
&toSpine\ (Tree\ a)\ Empty &&= Empty\ \text{`}As\text{`}\ \texttt{"Empty"}\\
&toSpine\ (Tree\ a)\ (Node\ l\ x\ r) &&= Node\ \text{`}As\text{`}\ \texttt{"Node"}\\
& &&\quad \diamond (l : Tree\ a) \diamond (x : a) \diamond (r : Tree\ a)\ \ .
\end{aligned}$$

With the new version of *Spine*, the function *show* is straightforward to write:

$$\begin{aligned}
&show :: Type\ a \to a \to String\\
&show\ t\ x = show'\ (toSpine\ t\ x)
\end{aligned}$$

$$\begin{aligned}
&show' :: Spine\ a \to String\\
&show'\ (_\ \text{`}As\text{`}\ c) &&= c\\
&show'\ (f \diamond (a : t)) &&= \texttt{"("} +\!\!+\ show'\ f +\!\!+ \texttt{" "} +\!\!+\ show\ t\ a +\!\!+ \texttt{")"}\ \ .
\end{aligned}$$

The result of the call

$$show\ (Tree\ Int)\ (Node\ (Node\ Empty\ 1\ Empty)\ 2\ (Node\ Empty\ 3\ Empty))$$

is "$(((Node\ (((Node\ Empty)\ 1)\ Empty))\ 2)\ (((Node\ Empty)\ 3)\ Empty))$".

Even though we have information about constructors, we cannot define a generic *read* without further extensions. In the next section, we will discuss this and other questions regarding the expressivity of the SYB approach.

6 SYB in context

In the previous sections, we have introduced the SYB approach on the basis of the *Spine* data type. Generic functions are overloaded functions that make use of the *Spine* view by calling *toSpine* on their type argument.

We have seen that we can define useful and widely applicable combinators such as *everything* and *everywhere* using some basic generic functions. As long as we stay within the expressivity of these combinators, it is possible to perform generic programming avoiding explicit case analysis on types.

In this section, we want to answer how expressive the *Spine* view is in comparison to both the original presentation of SYB, which uses only a given set of combinators, and in relation to other views, as they are employed by other approaches to generic programming such as PolyP and Generic Haskell.

6.1 The original presentation

As described in the section of implementing SYB in the original paper, it turns out that *mapT* and *mapQ* are both instances of a function that is called *gfoldl*. We can define *gfoldl*, too. To do this, let us define the ordinary fold (or catamorphism, if you like) of the *Spine* type:

$$\begin{aligned}
&\mathit{foldSpine} :: (\forall a.a \to r\ a) \to (\forall a\ b.r\ (a \to b) \to \mathit{Typed}\ a \to r\ b) \to \\
&\qquad\qquad \mathit{Spine}\ a \to r\ a \\
&\mathit{foldSpine}\ \mathit{constr}\ (\blacklozenge)\ (c\ \text{`}\mathit{As}\text{'}\ _) \quad = \mathit{constr}\ c \\
&\mathit{foldSpine}\ \mathit{constr}\ (\blacklozenge)\ (f \diamond (x : t)) = (\mathit{foldSpine}\ \mathit{constr}\ (\blacklozenge)\ f) \blacklozenge (x : t) \ .
\end{aligned}$$

The definition follows the catamorphic principle of replacing data constructors with functions. The SYB *gfoldl* is just *foldSpine* composed with *toSpine*:

$$\begin{aligned}
&\mathit{gfoldl} :: \mathit{Type}\ a \to (\forall a.a \to r\ a) \to (\forall a\ b.r\ (a \to b) \to \mathit{Typed}\ a \to r\ b) \to \\
&\qquad\quad a \to r\ a \\
&\mathit{gfoldl}\ t\ \mathit{constr}\ \mathit{app} = \mathit{foldSpine}\ \mathit{constr}\ \mathit{app} \circ \mathit{toSpine}\ t \ .
\end{aligned}$$

It is therefore clear that our approach via the *Spine* type and the original SYB approach via *gfoldl* are in principle equally expressive, because the *Spine* type can be recovered from *gfoldl*.

However, we believe that the presence of the explicit data type *Spine* makes the definitions of some generic functions easier, especially if they do not directly fall in the range of any of the simpler combinators.

The original SYB paper describes only generic functions that either consume a value based on its type (queries, consumers), or that consume a value based on its type and build up a similar value at the same time (traversals). There are also generic functions that construct values based on a type (producers). Such functions include the already mentioned generic *read*, used to parse a string into a value of a data type, or *some*, a function that produces some non-bottom value of a given data type. We cannot define such functions without further help: The definition of *some* would presumably follow the general pattern of overloaded functions on spines, the shape of the final case dictated by the type of *some* (cf. Section 4.2):

$$\begin{aligned}
&\mathit{some} :: \mathit{Type}\ a \to a\\
&\mathit{some}\ \ldots = \ldots\\
&\mathit{some}\ t \quad = \mathit{fromSpine}\ \mathit{some}'
\end{aligned}$$

But we cannot define $\mathit{some}' :: \mathit{Spine}\ a$, because that would yield $\mathit{fromSpine}\ \mathit{some}' :: \forall a.a$, which has to be $\perp$ according to the parametricity theorem [18]. Due to the well-definedness of *fromSpine*, some' would have to be $\perp$, too.

It is nevertheless possible to define $\mathit{some} :: \mathit{Type}\ a \to a$, but only if *Type* is augmented with more information about the type it represents and its constructors. Due to space limitations, the implementation is deferred to the technical report [8], where we give an equivalent of *gunfold* from the second SYB paper [19].

We *can*, however, define functions on multiple type arguments without further additions. The definition of generic equality is very straightforward using the spine view:

$$\begin{aligned}
&\mathit{eq} :: \mathit{Type}\ a \to \mathit{Type}\ b \to a \to b \to \mathit{Bool}\\
&\mathit{eq}\ t_1\ t_2\ x\ y = \mathit{eq}'\ (\mathit{toSpine}\ t_1\ x)\ (\mathit{toSpine}\ t_2\ y)\\
&\mathit{eq}' :: \mathit{Spine}\ a \to \mathit{Spine}\ b \to \mathit{Bool}\\
&\mathit{eq}'\ (_\ \text{`}\mathit{As}\text{`}\ c_1) \qquad (_\ \text{`}\mathit{As}\text{`}\ c_2) \qquad = c_1 == c_2\\
&\mathit{eq}'\ (f_1 \diamond (a_1 : t_1))\ (f_2 \diamond (a_2 : t_2)) = \mathit{eq}'\ f_1\ f_2 \wedge \mathit{eq}\ t_1\ t_2\ a_1\ a_2\\
&\mathit{eq}'\ _ \qquad\qquad\qquad _ \qquad\qquad\qquad = \mathit{False}\ .
\end{aligned}$$

The generalized type of *eq* avoids the necessity of a type-level equality test. In the second SYB paper, *eq* is defined in terms of a combinator called *zipWithQ*. Although we can mirror the definition of *zipWithQ*, we believe that the direct definition is much clearer.

6.2 Other views and their strengths and weaknesses

Let us now look at two other approaches to generic programming, PolyP and Generic Haskell. They are also based on overloaded functions, but they do not represent values using *Spine*. A different choice of view affects the class of generic functions that can be written, how easily they can be expressed, and the data types that can be represented.

PolyP In PolyP [2], data types of kind $* \rightarrow *$ are viewed as fixed points of regular pattern functors. The regular functors in turn are of kind $* \rightarrow * \rightarrow *$ and represented as lifted sums of products. The view makes use of the following type definitions:

$$
\begin{array}{ll}
\mathbf{data}\ \mathit{Fix}\ f & = \mathit{In}\ (f\ (\mathit{Fix}\ f)) \\
\mathbf{type}\ \mathit{LSum}\ f\ g\ a\ r & = \mathit{Either}\ (f\ a\ r)\ (g\ a\ r) \\
\mathbf{type}\ \mathit{LProd}\ f\ g\ a\ r & = (f\ a\ r, g\ a\ r) \\
\mathbf{type}\ \mathit{LUnit}\ a\ r & = () \\
\mathbf{type}\ \mathit{Par}\ a\ r & = a \\
\mathbf{type}\ \mathit{Rec}\ a\ r & = r\ .
\end{array}
$$

Here, *Fix* is a fixed-point computation on the type level. The type constructors *LSum*, *LProd*, and *LUnit* are lifted variants of the binary sum type *Either*, the binary product type (,), and the unit type (). Finally we have *Par* to select the parameter, and *Rec* to select the recursive call.

As an example, our type *Tree* has pattern functor *TreeF*:

$$\mathbf{data}\ \mathit{TreeF}\ a\ r = \mathit{EmptyF} \mid \mathit{NodeF}\ r\ a\ r\ .$$

We have (modulo $\perp$) that $\mathit{Tree}\ a \cong \mathit{Fix}\ (\mathit{TreeF}\ a)$. Furthermore, we can view *TreeF* as a binary sum (it has two constructors), where the right component is

a nested binary product (*NodeF* has three fields). The recursive argument r is represented by *Rec*, the parameter to *Tree* by *Par*:

$$\mathbf{type}\ \mathit{TreeFS}\ a\ r = \mathit{LSum}\ \mathit{LUnit}\ (\mathit{LProd}\ \mathit{Rec}\ (\mathit{LProd}\ \mathit{Par}\ \mathit{Rec}))\ a\ r\ .$$

Again, we have (modulo $\perp$) an isomorphism $\mathit{TreeF}\ a\ r \cong \mathit{TreeFS}\ a\ r$.

The view of PolyP has two obvious disadvantages: first, due to its two-level nature, it is relatively complicated; second, it is quite limited in its applicability. Only data types of kind $* \rightarrow *$ that are regular can be represented.

On the other hand, many generic functions on data types of kind $* \rightarrow *$ are definable. PolyP can express functions to parse, compare, unify, or print values generically. Its particular strength is that recursion patterns such as cata- or anamorphisms can be expressed generically, because each data type is viewed as a fixed point, and the points of recursion are visible.

Generic Haskell In contrast to PolyP, Generic Haskell [3, 4] uses a view that is much more widely applicable and is slightly easier to handle: all data types are (unlifted) sums of products. The data type *Tree* is viewed as the isomorphic

$$\textbf{type}\ \mathit{TreeS}\ a = \mathit{Either}\ ()\ (\mathit{Tree}\ a, (a, \mathit{Tree}\ a))\ .$$

The original type *Tree* appears in *TreeS*, there is no special mechanism to treat recursion differently. This has a clear advantage, namely that the view is applicable to nested and mutually recursive data types of arbitrary kinds. In fact, in Generic Haskell all Haskell 98 data types can be represented. The price is that recursion patterns such as cata- or anamorphisms cannot be defined directly.

On the other hand, generic functions in Generic Haskell can be defined such that they work on types of all kinds. It is therefore significantly more powerful than PolyP. In Generic Haskell we can, for instance, define a generic *map* that works for generalized rose trees, a data type of kind $(* \to *) \to * \to *$:

$$\textbf{data}\ \mathit{Rose}\ f\ a = \mathit{Fork}\ a\ (f\ (\mathit{Rose}\ f\ a))\ .$$

Scrap your boilerplate The *Spine* view is not so much based on the structure of types, but on the structure of values. It emphasizes the structure of a constructor application. We have already noticed that this limits the generic functions that can be written. Pure producers such as *read* or *some* require additional information. Furthermore, all generic functions work on types of kind $*$. It is not possible to define a generic version of *map* for type constructors, or to define a recursion pattern such as a catamorphism generically.

But the *Spine* view also has two noticeable advantages over the other views discussed. Firstly, the view is simple, and the relation between a value and its spine representation is very direct. As a consequence, the transformation functions *fromSpine* and *toSpine* are quite efficient, and it is easy to deforest the *Spine* data structure.

Secondly, as every (non-abstract) Haskell value is a constructor application, the view is very widely applicable. Not only all Haskell 98 data types of all kinds can be represented, the *Spine* view is general enough to represent data types containing existentials and even GADTs without any further modifications. This is particularly remarkable, because at the moment, GHC does not support automatic derivation of classes for GADTs. The methods of the classes *Eq*, *Ord*, and *Show* can easily be defined using the SYB approach. Thus, there is no theoretical problem to allow derivation of these classes also for GADTs. We discuss this newly found expressive power further in the next section.

7 Scrap your boilerplate for "Scrap your boilerplate"

There are almost no limits to the data types we can represent using *Spine*. One very interesting example is the GADT of types itself, namely *Type*. This allows us to instantiate generic functions on type *Type*. Consequently, we can show types by invoking the generic *show* function, or compute type equality using the generic equality *eq*! Both are useful in the context of dynamically typed values:

$$\begin{aligned}&\mathbf{data}\ \mathit{Dynamic}\ \mathbf{where}\\&\quad \mathit{Dyn} :: t \to \mathit{Type}\ t \to \mathit{Dynamic}\ .\end{aligned}$$

The difference between the types *Dynamic* and *Typed* is that *Dynamic* contains an existential quantification.

Before we can actually use generic functions on *Dynamic*, we require that *Type* has a constructor for types and dynamic values:

$$\begin{aligned}&\mathbf{data}\ \mathit{Type} :: * \to * \ \mathbf{where}\\&\quad \ldots\\&\quad \mathit{Type} \quad\ \ :: \mathit{Type}\ a \to \mathit{Type}\ (\mathit{Type}\ a)\\&\quad \mathit{Dynamic} :: \mathit{Type}\ \mathit{Dynamic}\ .\end{aligned}$$

The function *toSpine* also requires cases for *Type* and *Dynamic*, but converting types or dynamics into the *Spine* view is entirely straightforward, as the following example cases demonstrate:

$$\begin{aligned}&\mathit{toSpine}\ (\mathit{Type}\ a')\ (\mathit{Type}\ a) \ \ = \mathit{Type}\ \text{`}\mathit{As}\text{`}\ \texttt{"Type"} \diamond (a : \mathit{Type}\ a)\\&\mathit{toSpine}\ \mathit{Dynamic}\ (\mathit{Dyn}\ x\ t) = \mathit{Dyn}\ \text{`}\mathit{As}\text{`}\ \texttt{"Dyn"} \diamond (x : t) \diamond (t : \mathit{Type}\ t)\ .\end{aligned}$$

In the first line above, a' is always equal to a, but the Haskell type system does not know that, so we do not enforce it in the program. The output of *show Dynamic* (*Dyn* (*Node Empty* 2 *Empty*) (*Tree Int*)) is now the string "((*Dyn* (((*Node Empty*) 2) *Empty*)) (*Tree Int*))", and comparing the dynamic value to itself using *eq Dynamic Dynamic* yields indeed *True*, incorporating a run-time type equality test.

8 Conclusions

The SYB approach has been developed by Peyton Jones and Lämmel in a series of papers [1, 19, 20]. Originally, it was an implementation of *strategic programming* [14] in Haskell, intended for traversing and querying complex, compound data such as abstract syntax trees.

The ideas underlying the generic programming extension PolyP [2] go back to the categorical notions of functors and catamorphisms, which are independent of the data type in question [21]. Generic Haskell [22] was motivated by the desire to overcome the restrictions of PolyP.

Due to the different backgrounds, it is not surprising that SYB and generic programming have remained difficult to compare for a long time. The recent work on *generic views* [12, 23] has been an attempt to unify different approaches. We believe that we bridged the gap in this paper for the first time, by presenting the *Spine* data type which encodes the SYB approach faithfully.

Our implementation handles the two central ingredients of generic programming differently from the original SYB paper: we use overloaded functions with explicit type arguments instead of overloaded functions based on a type-safe cast [1] or a class-based extensible scheme [20]; and we use the explicit spine view rather than a combinator-based approach. Both changes are independent of each other, and have been made with clarity in mind: we think that the structure of the SYB approach is more visible in our setting, and that the relations to PolyP and Generic Haskell become clearer. We have revealed that while the spine view is limited in the class of generic functions that can be written, it is applicable to a very large class of data types, including GADTs.

Our approach cannot be used easily as a library, because the encoding of overloaded functions using explicit type arguments requires the extensibility of the *Type* data type and of functions such as *toSpine*. One can, however, incorporate *Spine* into the SYB library while still using the techniques of the SYB papers to encode overloaded functions (see the technical report [8] for more details).

In this paper, we do not use classes at all, and we therefore expect that it is easier to prove algebraic properties about SYB (such as $mapT\ copy = copy$ where $copy\ \_ = id$ is the identity traversal) in this setting. For example, we believe that the work of Reig [24] could be recast using our approach, leading to shorter and more concise proofs.

Acknowledgements We thank Jeremy Gibbons, Ralf Lämmel, Pablo Nogueira, Simon Peyton Jones, Fermin Reig, and the four anonymous referees for several helpful remarks.

References

1. Lämmel, R., Peyton Jones, S.: Scrap your boilerplate: a practical design pattern for generic programming. In: Types in Language Design and Implementation. (2003)
2. Jansson, P., Jeuring, J.: PolyP – a polytypic programming language extension. In: Conference Record 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France, ACM Press (1997) 470–482
3. Löh, A.: Exploring Generic Haskell. PhD thesis, Utrecht University (2004)
4. Löh, A., Jeuring, J., Clarke, D., Hinze, R., Rodriguez, A., de Wit, J.: The Generic Haskell user's guide, version 1.42 (Coral). Technical Report UU-CS-2005-004, Institute of Information and Computing Sciences, Utrecht University (2005)

5. Xi, H., Chen, C., Chen, G.: Guarded recursive datatype constructors. In: Proceedings of the ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL 2003), ACM Press (2003) 224–235
6. Peyton Jones, S., Washburn, G., Weirich, S.: Wobbly types: Type inference for generalised algebraic data types. Technical Report MS-CIS-05-26, University of Pennsylvania (2004)
7. Peyton Jones, S., ed.: Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press (2003)
8. Hinze, R., Löh, A., Oliveira, B.: "Scrap Your Boilerplate" reloaded. Technical report, Universität Bonn (2006) Available from `http://www.informatik.uni-bonn.de/~loeh/SYB0.html`.
9. GHC Team: The Glasgow Haskell Compiler User's Guide. (2005) Available from `http://haskell.org/ghc/docs/latest/users_guide.ps.gz`.
10. Wadler, P.: Views: a way for pattern matching to cohabit with data abstraction. In: Principles of Programming Languages, ACM Press (1987) 307–313
11. Burton, F.W., Meijer, E., Sansom, P., Thompson, S., Wadler, P.: Views: an extension to Haskell pattern matching. Available from `http://www.haskell.org/development/views.html` (1996)

12. Holdermans, S., Jeuring, J., Löh, A.: Generic views on data types. Technical Report UU-CS-2005-012, Utrecht University (2005)
13. Benke, M., Dybjer, P., Jansson, P.: Universes for generic programs and proofs in dependent type theory. Nordic Journal of Computing **10** (2003) 265–289
14. Visser, E.: Language independent traversals for program transformation. In Jeuring, J., ed.: Workshop on Generic Programming (WGP'00), Ponte de Lima, Portugal, Technical Report UU-CS-2000-19, Department of Information and Computing Sciences, Universiteit Utrecht (2000)
15. Hinze, R.: Fun with phantom types. In Gibbons, J., de Moor, O., eds.: The Fun of Programming. Palgrave (2003) 245–262
16. Oliveira, B., Gibbons, J.: Typecase: A design pattern for type-indexed functions. In: Haskell Workshop. (2005) 98–109
17. Li, H., Reinke, C., Thompson, S.: Tool support for refactoring functional programs. In Jeuring, J., ed.: Haskell Workshop, Association for Computing Machinery (2003) 27–38
18. Wadler, P.: Theorems for free! In: Functional Programming and Computer Architecture. (1989)
19. Lämmel, R., Peyton Jones, S.: Scrap more boilerplate: reflection, zips, and generalised casts. In: Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2004), ACM Press (2004) 244–255
20. 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
21. Backhouse, R., Jansson, P., Jeuring, J., Meertens, L.: Generic programming: An introduction. In Swierstra, S.D., Henriques, P.R., Oliveira, J.N., eds.: Advanced Functional Programming. Volume 1608 of Lecture Notes in Computer Science., Springer-Verlag (1999) 28–115
22. 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
23. Holdermans, S.: Generic views. Master's thesis, Utrecht University (2005)
24. Reig, F.: Generic proofs for combinator-based generic programs. In Loidl, H.W., ed.: Trends in Functional Programming. Volume 5. Intellect (2006)