Comparing Datatype Generic Libraries in Haskell

Alexey Rodriguez Yakushev

Johan Jeuring

Patrik Jansson

Alex Gerdes

Oleg Kiselyov

Bruno C.D.S. Oliveira

Department of Information and Computing Sciences,
Utrecht University

Technical Report UU-CS-2011-020

www.cs.uu.nl

ISSN: 0924-3275

June 2011

Under consideration for publication in J. Functional Programming

Comparing Datatype-Generic Libraries in Haskell

ALEXEY RODRIGUEZ YAKUSHEV
Vector Fabrics B.V., The Netherlands
(*e-mail:* alexey.rodriguez@gmail.com)

JOHAN JEURING
Utrecht University & Open Universiteit, The Netherlands
(*e-mail:* johanj@cs.uu.nl)

PATRIK JANSSON
Chalmers University of Technology & University of Gothenburg, Sweden
(*e-mail:* patrikj@chalmers.se)

ALEX GERDES
Open Universiteit, The Netherlands
(*e-mail:* alex.gerdes@ou.nl)

OLEG KISELYOV
FNMOC, CA, USA
(*e-mail:* oleg@okmij.org)

BRUNO C. D. S. OLIVEIRA
ROSAEC Center, Seoul National University, South Korea
(*e-mail:* bruno@ropas.snu.ac.kr)

Abstract

Datatype-generic programming is parametrizing programs by the structure, or "shape" of datatypes, letting us write, for example, generic traversal or pretty-printing once and apply them to any data type whose shape we can represent. Although more than two decades old, the field has been vigorously growing in recent years, particularly in Haskell. There are more than ten datatype-generic programming libraries in Haskell, not counting proposed language extensions.

The proliferation of the libraries poses the problem of comparing them, to help the users choose the right library for their tasks and to attempt to unify some of them. In this paper we develop an extensive test suite for comparing datatype-generic libraries in a typed functional language. We introduce a broad set of criteria and develop a collection of characteristic examples covering most of the facets of datatype-generic programming. We have implemented the examples for ten existing Haskell generic programming libraries and report for the first time the comprehensive evaluation of the libraries against the broad common standard.

1 Introduction

Every programmer is familiar with the tedium of writing yet another traversal, pretty-printing or update function for a new datatype. Each of these functions has a similar form across datatypes; they should be written only once, and then specialised to the structure, or shape of a given datatype. Parametrizing programs by (the structure of) the datatype is called *datatype-generic programming* (DGP) (Gibbons 2007). Larger applications of datatype-generic programming include XML tools, testing frameworks, debuggers, and data-conversion tools.

Although dating to more than 20 years ago, datatype-generic programming has flourished in the last decade, in particular in functional programming languages like Haskell and Clean (Alimarine & Plasmeijer 2001). Haskell alone counts more than a dozen datatype-generic libraries, and there are a few Haskell-like languages such as PolyP (Jansson & Jeuring 1997) and Generic Haskell (Löh *et al.* 2008), or pre-processors such as DrIFT (Winstanley & Meacham 2006) and Data.Derive (Mitchell 2009) providing support for DGP. Such libraries and language variants are also appearing in ML (Karvonen 2007; Yallop 2007) and Scala (Moors *et al.* 2006; Oliveira & Gibbons 2010).

The abundance of generic programming libraries in Haskell makes it difficult to choose the most appropriate library for a particular project. The libraries do differ, not only in performance, but also in expressiveness: some generic functions – such as *gmap*, which requires abstraction over type constructors – are difficult or downright impossible to define in some libraries; some libraries do not support datatypes with higher-rank parameters; and adding new generic functions and custom datatypes is easier in some libraries than others. These differences have not been systematically documented before; there does not exist a comprehensive test suite for comparing datatype-generic libraries. The lack of a comprehensive test suite containing testable requirements for determining (the absence of) support for generic programming, makes it harder to develop a new library or to unify existing ones.

The goal of this paper is to provide a detailed comparison of ten DGP Haskell libraries, which we believe are representative of various approaches to DGP, against a set of criteria of relevant features for DGP. The set of libraries under analysis is:

- Lightweight impl. of generics and dynamics (LIGD) (Cheney & Hinze 2002)
- Polytypic programming in Haskell (PolyLib) (Norell & Jansson 2004)
- Scrap your boilerplate (SYB) (Lämmel & Peyton Jones 2003, 2004)
- Scrap your boilerplate with class (SYB3) (Lämmel & Peyton Jones 2005)
- The spine view variant of SYB (Spine) (Hinze *et al.* 2006; Hinze & Löh 2006)
- Extensible and Modular Generics for the Masses (EMGM) (Oliveira *et al.* 2006) based on (Hinze 2006)
- RepLib: a library for derivable type classes (Weirich 2006)
- Smash your boilerplate (Smash) (Kiselyov 2006)
- Uniplate (Mitchell & Runciman 2007)
- MultiRec: Generic programming with fixed points for mutually recursive datatypes (Rodriguez Yakushev *et al.* 2009).

Since SYB and Strafunski (Lämmel & Visser 2003) are very similar, we only use SYB in this evaluation. The Compos library (Bringert & Ranta 2006) is subsumed by Uniplate, hence we evaluate only the latter.

For concreteness, this paper is limited to Haskell DGP *library*-based approaches. Our goal is to identify the desirable features for DGP in Haskell libraries, test each library in terms of support for these features and provide a detailed evaluation of our results. These results will be useful for potential users of DGP libraries by providing those users with detailed information about each library as well as their advantages and limitations. Furthermore, this work is also relevant for designers of DGP libraries and tools by identifying a large set of desirable features in a DGP library and providing a test suite which new libraries can use to verify whether or not some feature is supported. Finally, while some of the components of our test suite uses advanced Haskell features like GADTs, type classes and type constructor polymorphism (see Figure 9), our baseline and the evaluation extends, we feel, beyond Haskell. After all, Clean, Scala and now OCaml support or emulate many of the above features. Therefore, the results presented in this paper are also useful to generic programmers using other languages.

Others have already performed good, high-level, comparisons of support for generic programming in various languages, and there are also more general studies comparing various approaches to DGP (tools, extensions or libraries) in Haskell. Garcia *et al.* (2007) and Bernardy *et al.* (2010) compare the support for *property-based* generic programming (which is broader than datatype-generic programming) across different programming languages. Haskell type classes support all the eight criteria of Garcia *et al.* We use more fine-grained criteria to distinguish the Haskell libraries' support for *datatype*-generic programming. Oliveira & Gibbons (2010) compare Scala and Haskell in terms of the language mechanisms available in the two languages for defining DGP libraries (type classes and GADTs in Haskell, and the Scala object system), but they do not compare the libraries themselves. We share with Hinze *et al.* (2007) the task of comparing approaches to datatype-generic programming in Haskell. We differ in wider coverage, concentrating on libraries rather than language dialects and including several new library approaches such as RepLib, Smash, Uniplate and MultiRec. Finally, Hinze & Löh (2009) focus on comparing the different concepts that play a role in datatype-generic programming, whereas we aim at comparing existing library implementations.

In summary, this paper makes the following contributions:

- It gives an extensive set of *criteria for comparing datatype-generic libraries* (Section 4). The criteria might be viewed as a characterisation of all common uses of datatype-generic programming in Haskell[1].
- It develops a *generic programming test suite*: a set of characteristic examples with which we can test the criteria for generic programming libraries (Section 3). The test suite can be seen as a cookbook that illustrates how different generic programming tasks are achieved using the different approaches. Furthermore, its availability makes

[1] We omit generic programming over existential and higher-rank datatypes, for example, as such cases are uncommon and rarely implemented, see Section 3.

it easier to compare the expressiveness of future generic programming libraries. (The test suite homepage is `http://haskell.org/haskellwiki/GPBench`.)

- It *compares ten existing library approaches* to generic programming in Haskell with respect to the criteria, using the implementation of the test suite in the different libraries (Section 5). The complete code from the comparison (including the test suite) is freely available from `http://code.haskell.org/generics/comparison/`.

We assume familiarity with generic programming but, for ease of reference, we overview the terminology in Section 2.

The present paper is a rewritten and significantly extended version of (Rodriguez Yakushev *et al.* 2008). We have added the detailed evaluation, in Sections 5, 6, and 7, and extended the set of evaluated libraries. Since the publication of the original paper, our terminology, criteria and tests have been used in three new generic programming libraries: MultiRec (Rodriguez Yakushev *et al.* 2009), Alloy (Brown & Sampson 2009) and "Instant Generics" (Chakravarty *et al.* 2009). We include the detailed evaluation of MultiRec in the present extended version.

2 Generic programming: a brief overview

In this section we briefly describe datatype-generic programming, mainly to introduce terminology; see (Gibbons 2007) for a detailed discussion.

Generic functions, in a broad sense, are functions that apply to classes of values of several types. We call the set, or family, of types in the domain of a generic function *the universe*. We distinguish *datatype-generic* functions, the subject of the paper, from mere overloading or parametric polymorphism. We use generic pretty-printing and generic equality to illustrate the differences. Haskell provides overloaded equality via `==`, and pretty-printing via *show*. If we want to compare and show values of a newly defined datatype, we have to add a new instance for the type class `Eq` and for the type class *Show*. For each existing overloaded generic function that we wish to use with the new datatype, we have to add the corresponding type class instance. In contrast, the datatype-generic libraries evaluated in this paper require us to describe a newly defined datatype to the library only once. Any generic function of the library will then work with our datatype, with no further additions or modifications.

Datatype-generic functions are parametrized by the *shape* of a datatype, which is a type functor or a type function, generally of a higher kind (Gibbons 2007). Datatype-generic functions could be parametric (that is, act uniformly on all shapes) (Gibbons & Paterson 2009) — or they could do case analysis on the shape parameter. We restrict our attention to the latter category. In fact, generic equality or pretty-printing, for example, cannot be defined as parametrically polymorphic functions (Wadler 1989) and must do case analysis.

The example below makes shape parametrization concrete. This example will help to illustrate our test suite in the next section. We use the example to describe the difference between generic and ad hoc universe extensions. We have picked one of the simplest libraries, LIGD (Cheney & Hinze 2002) to describe our example. LIGD originally relied on emulated Generalized Algebraic Datatypes (GADTs); in our presentation we use GADTs supported by GHC (Peyton Jones *et al.* 2006).

```
data Unit      = Unit
data Sum a b = Inl a | Inr b
data Prod a b = Prod a b
```

Fig. 1. Unit, sum and product datatypes

```
data Rep t where
  RUnit  :: Rep Unit
  RSum   :: Rep a → Rep b → Rep (Sum a b)
  RProd  :: Rep a → Rep b → Rep (Prod a b)
  RType  :: Rep a → EP b a → Rep b
```

Fig. 2. Type representation Rep, or universe, of the LIGD library.

LIGD represents the structure, or shape, of a datatype using sum-, product-, and base types, see Figure 1 (we show only one base type Unit). For example, the shape of a pair of booleans is represented by the type Prod (Sum Unit Unit) (Sum Unit Unit).

The *structure representation* of type a as a nested sum-of-products type b is witnessed by an embedding-projection pair

```
data EP a b = EP {from :: a → b, to :: b → a}
```

converting between a and b values. For the pair of booleans,

```
fromTwoBool :: (Bool, Bool) → Prod (Sum Unit Unit) (Sum Unit Unit)
fromTwoBool (True, True)   = Prod (Inr Unit) (Inr Unit)
 ...
fromTwoBool (False, False) = Prod (Inl Unit) (Inl Unit)

toTwoBool :: Prod (Sum Unit Unit) (Sum Unit Unit) → (Bool, Bool)
toTwoBool (Prod (Inr Unit) (Inr Unit)) = (True, True)
 ...
```

Datatype-generic functions in LIGD receive their shape parameter as the first argument. Since we cannot pass types as function arguments in Haskell, we pass values that encode, or represent the types. Our representation Rep is the GADT defined in Figure 2. Values of Rep t are *type representations*, representing types that are either sums-of-products or can be represented as sums-of-products via embedding-projection. In other words, Rep t encodes the universe of LIGD. By abuse of terminology, we call Rep t itself the universe. The type representation for our pair of booleans is

```
rTwoBool = RType (RProd (RSum RUnit RUnit) (RSum RUnit RUnit))
                 (EP fromTwoBool toTwoBool)
```

We use generic equality *geq* as an example generic function, see Figure 3. Its first argument is the Rep a value encoding the shape of the type of the values to compare, which are passed as the next two arguments. Our *geq*, as other datatype-generic functions in this

```
geq :: Rep a → a → a → Bool
geq (RUnit)          Unit          Unit          = True
geq (RSum r_a r_b)   (Inl a_1)     (Inl a_2)     = geq r_a a_1 a_2
geq (RSum r_a r_b)   (Inr b_1)     (Inr b_2)     = geq r_b b_1 b_2
geq (RSum r_a r_b)   _             _             = False
geq (RProd r_a r_b)  (Prod a_1 b_1) (Prod a_2 b_2) = geq r_a a_1 a_2 ∧ geq r_b b_1 b_2
geq (RType r_a ep) t_1 t_2                       = geq r_a (from ep t_1) (from ep t_2)
```

Fig. 3. Type-indexed equality function in the LIGD library

paper, is defined by case analysis on the shape representation. Before a datatype-generic function can be applied to a value, the function must receive the parameter describing the shape of that value. In other words, the function must be *instantiated* for a particular shape. In LIGD, instantiation is passing the generic function such as *geq* the value Rep a as the first argument.

If we want to apply *geq* to lists or other (newly introduced) datatypes, we have to *extend* the universe to include the new datatype. We have two choices: either to introduce lists as a new shape, or to express lists in terms of the existing shapes of our representation. The first choice is an ad hoc, *non-generic extension*. In LIGD, we modify the definition of Rep to add a new data constructor *RList*. The change in Rep definition forces us to extend *geq* and all other generic functions, adding a case for *RList*. The second choice is *generic extension*. In LIGD, we merely need to establish the correspondence between lists and their sums-of-products representation by defining the embedding-projection pair, *fromList* and *toList*. We define the type representation for lists (parametrized by the representation for the list elements) as

```
rList    :: Rep a → Rep [a]
rList r_a = RType (RSum RUnit (RProd r_a (rList r_a))) (EP fromList toList)
```

Since list is a recursive datatype, its type representation is also recursive. We have not changed the definition of Rep nor of any generic function. Our old *geq* can be used as is to compare lists; we merely have to pass *rList* r_a as the first argument to *geq*.

The ad hoc extension of the universe is unappealing – in LIGD – since it breaks existing code. The ad hoc extension can be useful, to let a generic function process some datatypes in particular ways. For example, with the ad hoc extension, *geq* could compare lists directly, without first converting them, using *fromList*, to the sum-of-product representation and wasting time and memory. Other libraries support ad hoc extensions better.

LIGD's sum-of-product representation of datatypes defines a *generic view* (Holdermans *et al.* 2006) of the datatypes, which determines the set of generic functions and datatypes supported by the library. The LIGD view is not the only one possible; for example, PolyLib adds Fix to explicitly represent the recursive structure of datatypes. A library may support more than one view.

3 Design of the test suite

This section motivates and describes our test suite — a set of example datatypes and datatype-generic functions. The test suite is designed to test how a particular library sup-

ports criteria summarised in Section 4. Testing consists of attempting to implement test cases using a particular library and observing for each test case, if its implementation is even possible, if and how much the library core has to be extended, how many new definitions have to be introduced, and how fast the test case runs. Overall, the test suite is intended as the yardstick against which to compare the expressivity and performance of datatype-generic libraries. We describe the test suite before the criteria to build the intuition for them.

Our test suite is based on typical generic programming scenarios found in the literature; examples used in related generic programming comparison studies, in particular (Hinze *et al.* 2007); the Haskell generics wiki page and the generics mailing list; and our own extensive experience with generic programming.

3.1 Datatypes

Our test suite is designed to represent the wide range of datatypes found in typical generic programming scenarios: algebraic and nested datatypes; datatypes with simple, mutual, and nested recursion; datatypes parametrised by one or several simple types (of kind $\star$); and higher-kinded datatypes (with a type parameter of kind $\star \to \star$). Access to the names of the data constructors is also need for generic show, etc. In this section, we define our example datatypes and describe why we choose them.

Our test suite omits higher-rank data constructors (explicit $\forall$ in the datatype declaration), existential types and GADTs because hardly any library under evaluation can deal with these features (Spine is the only datatype-generic library supporting GADTs and, partially, existential types.) We do not consider record label names, constructor fixity, and precedence in choosing datatypes; there are no generic functions in our suite that use these features.

The company datatype. The Company datatype representing the organisational structure of a company is the motivating example for the generic programming library SYB (Lämmel & Peyton Jones 2003); the datatype together with the generic function *updateSalary* (shown later) has become a popular example of datatype-generic programming, often referred to as the "paradise benchmark".

```
data Company  = C [Dept]
data Dept     = D Name Manager [DUnit]
data DUnit    = PU Employee | DU Dept
data Employee = E Person Salary
data Person   = P Name Address
data Salary   = S Float
type Manager  = Employee
type Name     = String
type Address  = String
```

Company is a group of mutually recursive monomorphic datatypes, and is quite characteristic: the datatypes comprising the group make use of products (Employee, Person), sums (DUnit), lists, and mutual recursion (Dept and DUnit). Furthermore, Name and Address

(which are strings and so technically lists in Haskell) would most of the time be treated differently from the lists appearing in Companys or Depts.

To use Company with LIGD as described in Section 2, we represent the datatypes of the group as sums-of-products, and define type representations:

```
rCompany :: Rep Company
rCompany = RType (rList rDept) (EP fromCompany toCompany)
rDept    :: Rep Dept
rDept    = ...
```

However, since we intend to process Salary in an ad hoc way, we extend the universe non-generically, defining Salary as a datatype shape of its own. In LIGD, we have to modify the definition of Rep and add a new data constructor:

```
data Rep t where
  ...
  RSalary :: Rep Salary
```

Binary trees. We choose the datatype BinTree as a representative polymorphic recursive algebraic datatype. It is parametrized by the type of the values stored in the leaves:

```
data BinTree a = Leaf a | Bin (BinTree a) (BinTree a)
```

To use BinTree with LIGD, we define a parametrized type representation:

```
rBinTree :: Rep a → Rep (BinTree a)
```

similar to that of lists in Section 2.

Trees with weights. We regard the datatype WTree as BinTree with an extra variant, to assign the weight (of type w) to a (sub)tree. The datatype now has two type parameters.

```
data WTree a w = WLeaf a
               | WBin (WTree a w) (WTree a w)
               | WithWeight (WTree a w) w
```

The motivation for WTree is the implementation of scenarios of generic traversals and transformations with ad hoc processing for particular nodes, like *WithWeight*. We have a test case that treats weights differently from elements, even when their types are the same.

Generalised rose trees. Generalised rose trees are an example of a higher-kinded datatype. Starting with ordinary rose trees (non-empty trees whose internal nodes may have any number of children collected in a list),

```
data Rose a = Node a [Rose a]
```

we abstract over the type (constructor) of the collection:

```
data GRose f a = GNode a (f (GRose f a))
```

The datatype GRose has two parameters, one of which is higher kinded: $f :: \star \to \star$.

The type representation of GRose in LIGD uses higher-rank polymorphism:

$$rGRose :: (\forall a\,.\,\mathsf{Rep}\ a \to \mathsf{Rep}\ (f\ a)) \to \mathsf{Rep}\ a \to \mathsf{Rep}\ (\mathsf{GRose}\ f\ a)$$

Perfect trees. The datatype Perfect is an example of a *nested datatype* (Bird & Meertens 1998):

```
data Perfect a = Zero a | Succ (Perfect (Fork a))
data Fork a    = Fork a a
```

The datatype models perfect depth-n binary trees, which have exactly 2^n leaves. The depth is encoded in the number of *Succ* constructors. The datatype is non-regular: the type parameter changes from a parent node (of type Perfect a) to a children node (of type Perfect (Fork a)).

Nested generalised rose trees. The most advanced datatype in our suite combines nesting with higher-kinded arguments:

```
data NGRose f a = NGNode a (f (NGRose (Comp f f) a))
newtype Comp f g a = Comp (f (g a))
```

We use this datatype to test a generic library's support for operations on type constructors, such as composition.

3.2 Generic functions

In this section we describe example generic functions, chosen to represent typical datatype-generic programming scenarios. By checking how well a library implements our example functions we infer how well the library supports the scenarios. To describe the intended signatures and behaviour for our example generic functions, we use their implementation in LIGD.

3.2.1 Serialisation and deserialisation: Show

Data conversion (Jansson & Jeuring 2002) is one of the most common applications of generic programming: the first thing a programmer wants after defining a datatype is to inspect and enter its values. The Haskell Prelude provides overloaded generic serialisation and deserialisation functions, *read* and *show*, and a mechanism to automatically derive instances of these functions for new datatypes. Alas, the deriving mechanism is limited, for example, it is impossible to define ad hoc serialisation for a particular data constructor; GHC cannot derive *Show* instances for GRose; and manual construction of *Show* and *Read* instances involves a lot of boilerplate.

We choose the datatype-generic version of Haskell's *show* for our test suite (*read* is an instance of a generic producer, for which we have a test case *gfulltree* below). In LIGD, generic show has this signature:

$$gshow :: \mathsf{Rep}\ a \to a \to \mathsf{String}$$

To implement *gshow*, a generic view supported by a library should provide the names of data constructors. To simplify the test case, our *gshow* does not support record labels, fixity and precedence. Moreover, *gshow* should print strings as any other lists, for example:

gshow rep $[1,2] \rightsquigarrow$ `"(:) 1 ((:) 2 [])"`
gshow rep `"GH"` $\rightsquigarrow$ `"(:) 'G' ((:) 'H' [])"`

We use $\rightsquigarrow$ to indicate the result of evaluating an expression. We will often elide the shape representation argument, writing *rep* as the placeholder.

A separate example generic function, *gshowExt*, should print lists in Haskell notation:

gshowExt rep $(1,$ `"GH"`$) \rightsquigarrow$ `"(1,['G','H'])"`

3.2.2 Generic Equality

Structurally comparing two values is another typical generic programming scenario. Our test suite includes the structural generic equality (see also Figure 3):

$geq :: \mathsf{Rep\ a} \to \mathsf{a} \to \mathsf{a} \to \mathsf{Bool}$

The function returns *True* if the two arguments have the same constructor and their arguments are pairwise equal. To implement *geq*, a generic view does not have to provide the names of data constructors; only the order of the data constructors in the shape description matters.

3.2.3 Querying and transformation traversals

Querying traversals collect all occurrences of values of a particular fixed type in a datatype. For example,

$selectSalary :: \mathsf{Rep\ a} \to \mathsf{a} \to [\mathsf{Salary}]$

lists all Salary values within a datatype. Such traversals are implemented as generic traversals with an ad hoc case for the selected datatype. Here is the implementation in LIGD:

$$
\begin{array}{lll}
selectSalary\ RUnit & i & = [\,] \\
selectSalary\ (RSum\ r_{\mathsf{a}}\ r_{\mathsf{b}})\ (Inl\ a) & & = selectSalary\ r_{\mathsf{a}}\ a \\
selectSalary\ (RSum\ r_{\mathsf{a}}\ r_{\mathsf{b}})\ (Inr\ b) & & = selectSalary\ r_{\mathsf{a}}\ b \\
selectSalary\ (RProd\ r_{\mathsf{a}}\ r_{\mathsf{b}})\ (Prod\ a\ b) & & = selectSalary\ r_{\mathsf{a}}\ a +\!\!+ selectSalary\ r_{\mathsf{b}}\ b \\
selectSalary\ (RType\ r_{\mathsf{a}}\ ep)\ t & & = selectSalary\ r_{\mathsf{a}}\ (from\ ep\ t) \\
selectSalary\ RSalary & sal & = [sal]
\end{array}
$$

All clauses but the last implement a generic tree fold. The last, ad hoc, clause relies on the ad hoc extension of the Rep universe with *RSalary*.

Higher-order traversals. The generic traversal part of a querying traversal can be factored out into a generic function of its own. That function takes a selector argument, and applies it to the values encountered during the traversal. The selector is a generic function with an ad hoc clause. Thus the generic traversal function is higher-order, taking a generic function as an argument.

As a generic traversal we choose the shallow traversal function *gmapQ* from (Lämmel & Peyton Jones 2003), of the following LIGD signature and schematic reduction.

$$gmapQ :: (\forall \mathsf{a} \,.\, \mathsf{Rep\ a} \to \mathsf{a} \to \mathsf{r}) \to \mathsf{Rep\ b} \to \mathsf{b} \to [\mathsf{r}]$$
$$gmapQ\ sel\ rT\ (K\ a_1 \ldots a_n) \rightsquigarrow [sel\ rT_1\ a_1, \ldots, sel\ rT_n\ a_n]$$

If the last argument to *gmapQ* has the form $K\ a_1 \ldots a_n$ for some data constructor K, then the shape descriptor rT must contain the shape descriptors $rT_1 \ \ldots \ rT_n$ for all arguments of the constructor. The function *gmapQ* should know how to extract these descriptors in order to instantiate the selector function *sel*. Here is an example application of *gmapQ* to *selectSalary*:

$$\begin{aligned}
&gmapQ\ selectSalary\ (rList\ (RSum\ RUnit\ RSalary))\ (Inl\ Unit : [Inr\ (S\ 2.0)]) \\
&\quad \rightsquigarrow [selectSalary\ (RSum\ RUnit\ RSalary) \qquad (Inl\ Unit) \\
&\qquad , selectSalary\ (rList\ (RSum\ RUnit\ RSalary))\ [Inr\ (S\ 2.0)]] \\
&\quad \rightsquigarrow [[\,], [S\ 2.0]]
\end{aligned}$$

Transformation traversals. A transformation traversal is also a generic traversal with an ad hoc clause. Rather than collecting values of some fixed datatype, a transformation traversal returns a new value of the same type as the original one but with the updated values of the fixed datatype. Our test suite includes *updateSalary p*, which increases (by a proportion p) all occurrences of Salary in a value of an arbitrary datatype. The function is part of the "paradise benchmark".

$$updateSalary :: \mathsf{Float} \to \mathsf{Rep\ a} \to \mathsf{a} \to \mathsf{a}$$
$$updateSalary\ 0.1\ (rList\ RSalary)\ [S\ 1000.0, S\ 2000.0] \rightsquigarrow [S\ 1100.0, S\ 2200.0]$$

Transformations on constructors. The ad hoc behaviour in *updateSalary* targets a particular datatype as a whole. A refinement, constructor cases (Clarke & Löh 2003), targets a particular data constructor within the fixed datatype. The rest of the data constructors should be handled uniformly. Moreover, we should not even mention the rest of the data constructors when writing the generic function. The motivation comes from applying an optimisation $Plus\ x\ 0 \mapsto x$ to values of a datatype with a large number of constructors. Our optimisation function should only mention *Plus* by name.

In our test suite, a test case for constructor cases is a function *rmWeights*, which removes the constructors *WithWeight* from a WTree:

$$\begin{aligned}
&rmWeights\ (RWTree\ RInt\ RInt)\ (WBin\ (WithWeight\ (WLeaf\ 17)\ 1) \\
&\qquad\qquad (WBin\ (WLeaf\ 23)\ (WithWeight\ (WLeaf\ 38)\ 2))) \\
&\qquad \rightsquigarrow WBin\ (WLeaf\ 17)\ (WBin\ (WLeaf\ 23)\ (WLeaf\ 38))
\end{aligned}$$

The transformation should be defined so as to mention only *WithWeight* explicitly:

$$\begin{aligned}
&rmWeights :: \mathsf{Rep\ a} \to \mathsf{a} \to \mathsf{a} \\
&rmWeights\ r@(RWTree\ ra\ rw)\ t = \mathbf{case}\ t\ \mathbf{of} \\
&\quad WithWeight\ t'\ w \to rmWeights\ r\ t' \\
&\quad t' \qquad\qquad\quad \to \ldots \text{handle generically} \ldots \\
&\ldots \text{rest of definition omitted} \ldots
\end{aligned}$$

The second branch of the **case** traverses the structure representation of t' generically, rather than matching *WBin* and *WLeaf* explicitly.

3.2.4 Abstraction over type constructors: crush and map

Haskell's Data.*Foldable.foldr* and *fmap* are overloaded functions that generalise *foldr* and *fmap* to act on any collection rather than a list. Our test suite includes datatype-generic versions of these functions, called *crushRight* and *gmap* for historical reasons. The function *crushRight* (Meertens 1996) is a generic foldr, whose typical instances are summing up all integers in a list, or flattening a tree into a list of elements.

```
sumList :: [Int] → Int
sumList [2,3,5,7] ⇝ 17
flattenBinTree :: BinTree a → [a]
flattenBinTree (Bin (Leaf 2) (Leaf 1)) ⇝ [2,1]
```

The generic version of these functions abstracts over the type of the collection:

```
crushRight :: Rep' f → (a → b → b) → b → f a → b
```

The type variable f is of the kind $\star \to \star$; therefore, in LIGD we need a different type representation, Rep′ (see Figure 4), which is parametrised by type constructors rather than ordinary types. (Actually, instead of Rep′, the LIGD library uses more complex arity-based type representations Hinze (2000, 2006).)

It is instructive to contrast the type representation *rList* r_a :: Rep [a] representing the shape of lists of a specific element type a, with *rList′* :: Rep′ [], which represents the shape of lists regardless of the element type. The functions *sumList* and *flattenBinTree* are obtained by instantiating *crushRight* to lists or trees:

```
sumList       = crushRight rList' (+) 0
flattenBinTree = crushRight rTree' (:) []
```

```
data Unit' c      = Unit'
data Sum' a b c   = Inl' (a c) | Inr' (b c)
data Prod' a b c  = Prod' (a c) (b c)
data Id' c        = Id' c
data Rep' (f :: ⋆ → ⋆) where
  RUnit'  :: Rep' Unit'
  RSum'   :: Rep' a → Rep' b → Rep' (Sum' a b)
  RProd'  :: Rep' a → Rep' b → Rep' (Prod' a b)
  RType'  :: Rep' a → (∀c. EP (b c) (a c)) → Rep' b
  RId'    :: Rep' Id'
rList'    :: Rep' []   -- Note that the list type constructor is used here
rList'    = RType' (RSum' RUnit' (RProd' RId' rList')) (EP fromList' toList')
fromList' :: [a] → Sum' Unit' (Prod' Id' []) a
toList'   :: Sum' Unit' (Prod' Id' []) a → [a]
```

Fig. 4. Representations of types of kind $\star \to \star$ in our version of LIGD.

There is a subtle but important difference between *crushRight* and traversal queries. Suppose we want to collect weights from a WTree:

```
flattenWTWeights :: WTree a w → [w]
flattenWTWeights (WBin (WithWeight (WLeaf 17) 1) (WithWeight (WLeaf 38) 2)))
  ⇝ [1,2]
```

With *crushRight*, we merely need to instantiate it to the type constructor $\mathsf{WTree\ a} :: \star \to \star$.

```
flattenWTWeights = crushRight rWTree' (:) []
```

We may also attempt the traversal query *selectInt* (analogous to *selectSalary*). However, it cannot distinguish Int-weights from Int-elements in the tree and gives an incorrect result for our example:

```
selectInt (WBin (WithWeight (WLeaf 17) 1) (WithWeight (WLeaf 38) 2)))
  ⇝ [17,1,38,2]
```

To use the generic traversal correctly, we need an ad hoc constructor case for *WithWeight* to extract the weight value. The *crushRight* approach avoids such ad hoc cases.

Map. Generic map is to transformation traversals what *crushRight* is to query traversals:

```
gmap :: Rep' f → (a → b) → f a → f b
```

It is a datatype-generic version of Haskell's *fmap*. Our test case uses *gmap* for binary trees:

```
gmap rBinTree' :: (a → b) → BinTree a → BinTree b
```

3.2.5 Test data generation: Fulltree

We have not yet covered producing values of a particular datatype. Haskell's *read* is the best example of an overloaded generic producer; other examples are *minBound* and *maxBound*. For our test suite we select a generalisation of *minBound*, *gfulltree*, a simple value enumerator for datatypes. The function *gfulltree* takes a shape description as input and returns all possible values of the represented datatype up to the given depth. (The depth argument only makes sense with a recursive datatype). Using such systematically generated values for testing is characteristic of SmallCheck (Runciman *et al.* 2008).

```
gfulltree :: Rep a → Int → [a]
gfulltree (rList RUnit) 4 ⇝ [[],[Unit],[Unit,Unit],[Unit,Unit,Unit]]
gfulltree (rBinTree RUnit) 4 ⇝
  [Leaf Unit
  ,Bin (Leaf Unit)                    (Leaf Unit)
  ,Bin (Leaf Unit)                    (Bin (Leaf Unit) (Leaf Unit))
  ,Bin (Bin (Leaf Unit) (Leaf Unit)) (Leaf Unit)
  ,Bin (Bin (Leaf Unit) (Leaf Unit)) (Bin (Leaf Unit) (Leaf Unit))]
```

A slight variation of *gfulltree* could be used to generate the (usually infinite) list of all (finite) values of type a.

Types

- Universe size
- Subuniverses
- Views

Expressiveness

- First-class gen. functions
- Abstraction over type constructors

Expressiveness (continued)

- Ad hoc definitions for datatypes
- Ad hoc definitions for constructors
- Extensibility
- Multiple arguments
- Constructor names
- Consumers, transformers, and producers

Usability

- Separate compilation
- Performance
- Portability
- Overhead of library use
- Practical aspects
- Ease of use and learning
- Implementation mechanisms

Fig. 5. Criteria overview

4 Criteria

While describing our test suite in the previous section, we discussed the typical generic programming scenarios the test cases represent and the generic programming features the test cases require. This section distills these features and presents them as criteria against which to evaluate datatype-generic programming libraries. We have grouped the criteria around three aspects:

- Types: the classes of datatypes that can be used with a library
- Expressiveness: the classes of generic functions that can be written
- Usability: ease of use, portability, maintenance, performance

Figure 5 summarises the criteria and the organisation.

Our criteria characterise generic programming from the point of view of a user — a programmer who *writes* generic functions. There are also users who only apply generic functions that have already been developed, and so use a subset of generic programming features.

4.1 Types

Universe size. Generic programming libraries differ in the set of the datatypes the library can represent and hence lets us write generic functions for. Typically a library supports a small set of 'standard' datatypes like integers, pairs, lists; more datatypes can be added. Although user-defined regular algebraic datatypes can be added to any library under evaluation, nested and higher-kinded datatypes can be added only to some libraries.

We estimate the universe size of a library by testing if a library's universe contains or can be extended with a number of specially chosen datatypes given in Section 3.1.

Subuniverses. Is it possible to restrict the use of a generic function to a particular set of datatypes (e.g., non-nested lists)? Will the compiler report a type error if a restricted generic function is used on datatypes outside its subuniverse?

Views. Which views are supported by the library? Examples of views are the sum-of-products view, the fixed-point view, and the spine view.

4.2 Expressiveness

First-class generic functions. Can a generic function take a generic function as argument? This is tested by the function *gmapQ* from Section 3.2.3.

Abstraction over type constructors. Does the library support parametrization over type constructors of kind $\star \to \star$ or higher? We test this by checking if the library can implement *gmap* and *crushRight* from Section 3.2.4.

Ad hoc definitions for datatypes. Does the library support ad hoc behaviour for some particular datatype? In other words, can the library implement *selectSalary* from Section 3.2.3?

Ad hoc definitions for constructors. Does the library support constructor cases, Section 3.2.3? We test this by implementing *rmWeights*.

Extensibility. Can the programmer non-generically extend the universe of a generic function in a different module? This criterion only makes sense for libraries that support ad hoc definitions. We test for this criterion by extending *gshow* with an ad hoc case for printing lists using Haskell notation:

```
module ExtendedGShow where
import GShow    -- import definition of gshow
  -- ad hoc extension
gshow (RList r_a) xs = ...
```

Multiple arguments. Does the library support generic functions that take more than one generic argument, such as generic equality *geq*?

Constructor names. Does the library support a generic view providing the names of data constructors (so that *gshow* from Section 3.2.1 is implementable)?

Consumers, transformers, and producers. Does the library support the following classes of generic functions?

- consumers ($\mathsf{a} \to \mathsf{T}$): *gshow* and *selectSalary*
- transformers ($\mathsf{a} \to \mathsf{a}$ or $\mathsf{a} \to \mathsf{b}$): *updateSalary* and *gmap*
- producers ($\mathsf{T} \to \mathsf{a}$): *gfulltree*

4.3 Usability

Separate compilation. Is the universe extension modular? That is, can a datatype defined in one module be used with a generic function and type representation defined in another module without the need to modify or recompile the original module? This criterion is tested by applying generic equality to BinTree, which is defined in a different module than the equality and the library itself.

```
module BinTreeEq where
import LIGD   -- import LIGD representations
import GEq    -- and geq
data BinTree a = ...
rBinTree ra = RType (...) (EP fromBinT toBinT)
eqBinTree = geq (rBinTree RInt) (Leaf 2) (Bin (Leaf 1) (Leaf 3))
```

Performance. How fast is the library on typical generic programming operations?

Portability. Most of the libraries rely on various extension to the Haskell 2010 standard such as multi-parameter type classes with functional dependencies, GADTs, etc. Some of these extensions are GHC (-version)-specific, which precludes using the library with different Haskell compilers.

Overhead of library use. How difficult is it to use the library? We are interested in (1) support for automatic generation of structure representations, (2) the number of structure representations needed per datatype, (3) the amount of work to instantiate a generic function, and (4) the amount of work to define a generic function.

Practical aspects. How well is the library documented and maintained?

Ease of learning and use. How easy is it to understand the implementation of the library, should it have to be modified?

4.4 Coverage of testable criteria

Criteria like the quality of documentation or maintenance cannot be tested with a test suite. The other criteria are tested by trying to implement a particular test case of the suite and evaluating the success of the implementation. Figure 6 shows the coverage of testable criteria. The rows represent testable criteria and the columns represent the means of testing them. The first group of columns stand for the example functions introduced in Section 3. The second group of columns stand for datatypes that test generic universe extension. Those tests check whether *geq* can be instantiated and applied to values of the corresponding types.

Some test cases require a library's support of several criteria. For example, the generic extension test on the GRose datatype requires separate compilation and higher-kinded datatypes. If the library fails one of these criteria, the test case cannot be completely implemented. Therefore, we cannot check if the library supports the other criterion. In some cases we can modify the test case to exclude its reliance on one of the criteria, such as separate compilation. That is, we check if a simpler version of the test case can still be implemented by the library. We mark the criteria for which a simplification is possible with ○.

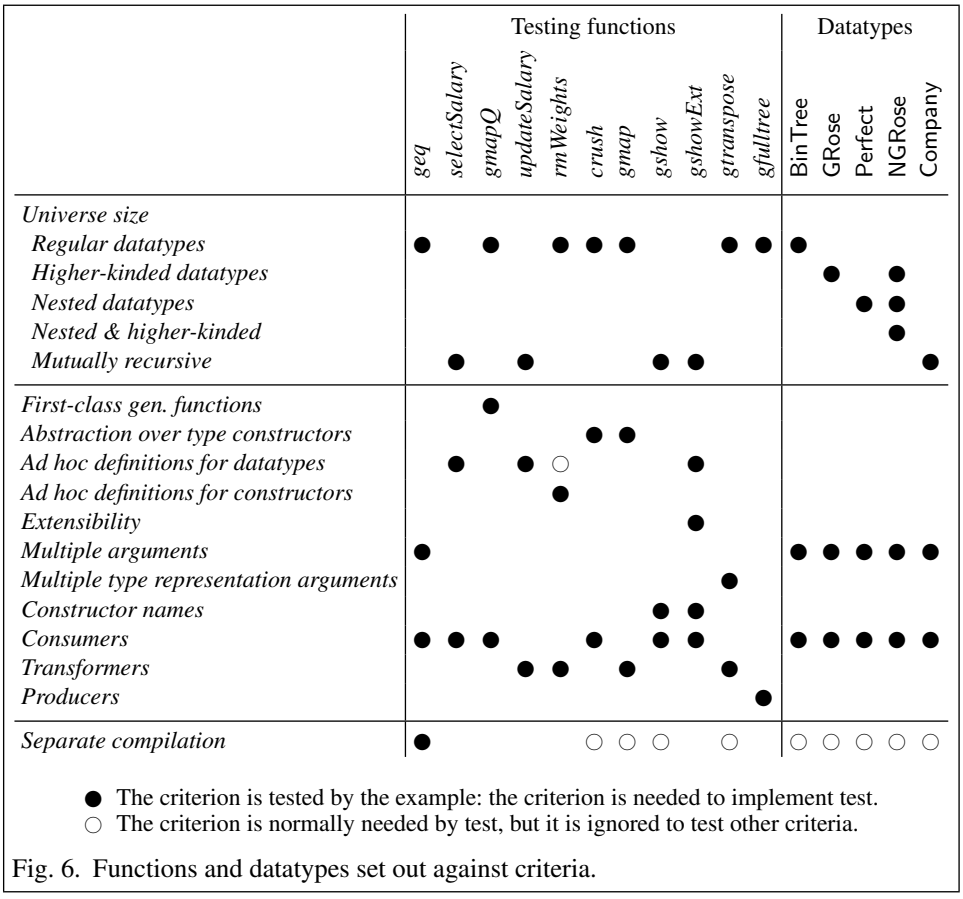

Fig. 6. Functions and datatypes set out against criteria.

4.5 Design choices

The criteria have been presented from a user's point of view. They inform the user which generic programs can or cannot be written using a given library. It is also illustrative to compare the design choices behind the libraries, which can give insight into the expressiveness of the library or the lack of it. For example, if generic functions are type class methods, it becomes quite difficult to pass them as first-class values (since that requires higher-rank polymorphism and many type annotations).

One design choice is what **type representation** to use. How are types and their structure represented at runtime? Are these representations handled explicitly (as arguments that can be pattern-matched) or implicitly (as type class contexts)? Are they abstract (higher-order, including functions) or concrete (first order syntax for types).

A second design choice is whether generic functions are instantiated by compile time specialisation or by interpretation of type representations at runtime. We do not discuss this design choice further, because all evaluated libraries use interpretation. The approaches which use explicit type representations clearly use run-time interpretation. The type-class–based approaches also perform interpretation, with the help of run-time dictionary values.

(Inlining may shift some, or all, of that run-time interpretation into compile-time (Magalhães *et al.* 2010).)

5 Evaluation: Types

We have implemented the test suite for each evaluated generic programming library. We discuss the results in the following three sections and Figure 7 presents the summary. The criteria supported by a generic programming library are marked with a black circle (● = *'good'*), not supported criteria are marked with a white circle (○ = *'bad'*). If a criterion is partially supported, or if it requires unusual programming effort, it is marked with a half-black circle (◐ = *'sufficient'*). The complete code for all our implementations is freely available from `http://code.haskell.org/generics/comparison/`.

5.1 Universe size

On which types can we use generic functions? That is, which datatypes can we add to library's universe? We answer this question by trying to extend a library's universe with each of our example datatypes. A library scores 'good' on regular, higher-kinded datatypes, nested datatypes, higher-kinded and nested datatypes, and mutually recursive datatypes, if it can generically extend the universe to BinTree, GRose, Perfect, NGRose, and Company respectively, and use the generic equality on them. The library scores 'bad' otherwise.

Figure 7 shows the scores. The libraries that depend on Haskell's Typeable or Data instances (e.g., SYB) have trouble with the higher-kinded datatypes GRose and NGRose: the necessary instances cannot be derived automatically. However, the programmer can write the Typeable and Data instances for GRose manually, relying on GHC's support for cyclic instances. We count that as support for GRose. In contrast, Typeable instances for NGRose cannot be written even manually. SYB3 library supports BinTree only if the necessary SYB3's Data instance is manually written; the automatically derived one is buggy. EMGM scores sufficient on NGRose because the GRep instance for that datatype cannot be defined, impairing the library. Smash supports all datatypes, yet adding Perfect, NGRose, and Company requires extra effort, as documented in our code.

5.2 Subuniverses

Is it possible to restrict the use of a generic function to a particular set of datatypes? A library scores 'good' if using the restricted generic function on datatypes outside of the subuniverse are flagged as compile-time errors.

In RepLib or PolyLib, the set of types to which a generic function can be instantiated is controlled by instance declarations. For example, if generic equality is to be used on lists, the programmer is expected to write the following RepLib instance (or an instance containing an ad hoc definition):

instance Geq $a \Rightarrow$ Geq $[a]$

Omitting such instances would result in a type checking error when applying equality to lists.

	LIGD	PolyLib	SYB	SYB3	Spine	EMGM	RepLib	Smash	Uniplate	MultiRec
Universe size										
Regular datatypes	●	●	●	●	●	●	●	●	●	●
Higher-kinded datatypes	●	○	●	●	●	●	○	●	●	○
Nested datatypes	●	○	●	○	●	●	●	◐	●	○
Nested & higher-kinded	●	○	○	○	●	◐	○	◐	○	○
Mutually recursive	●	○	●	●	●	●	●	◐	●	●
Subuniverses	○	●	○	○	○	◐[1]	●	●	○	○
First-class gen. functions	●	○	●	●	●	◐	●	◐	○	○
Abstraction over type constructors	●	◐	◐	○	◐	●	●	●	○	○
Ad hoc definitions for datatypes	○	○	●	●	○	●	●	●	●	◐
Ad hoc definitions for constructors	◐	●	●	●	●	●	●	●	●	●
Extensibility	○	○	○	●	○	●	●	●	○	○
Multiple arguments	●	●	◐	◐	●	●	●	◐	○	●
Constructor names	●	●	●	●	●	●	●	●	○	●
Consumers	●	●	●	●	●	●	●	●	●	●
Transformers	●	●	●	●	●	●	●	◐	●	●
Producers	●	●	◐	◐	◐	●	●	◐	○	●
Separate compilation	●	●	●	●	○	●	●	●	●	●
Performance (overhead—low is good)	13	8	51	20	7	5	7	5	16	8
Portability	●	○	○	○	○	●	○	○	●	○
Overhead of library use										
Automatic generation of representations	○	○	●	◐	○	●	◐	○	●	◐
Number of structure representations	4	1	2	2	3	4	4	8	1	1
Work to instantiate a generic function	●	●	●	●	●	●[1]	◐	●	●	●
Work to define a generic function	◐	●	●	◐	●	●	◐	●	●	●
Practical aspects	○	◐	●	◐	○	●	●	○	●	●
Ease of learning and use	◐	●	○	○	●	◐	○	○	●	○

● Supported criterion
○ Unsupported criterion
◐ Partially supported criterion or unusual programming effort required

Fig. 7. Evaluation of generic programming approaches — the main results of this paper. Notes: 1) These scores reflect the EMGM variant dicussed in Jeuring *et al.* (2009), which was used in the evaluation. In the original EMGM (Oliveira *et al.* 2006) proposal the two scores would be swapped.

In EMGM proposal by Oliveira *et al.* (2006), subuniverses can be controlled very much like in RepLib. For example this is how we would declare that generic equality can be used on lists:

```
instance GenericList Geq
```

Note, however, that in RepLib and this version of EMGM we have to write such trivial instances for each combination of a generic function and datatype. This lead Jeuring *et al.* (2009) to suggest that it may be desirable to trade some flexibility for a reduced number of instance declarations (which increases the score of the approach in the work to instantiate generic functions criteria). Therefore Jeuring *et al.* propose a slight variation of the EMGM library, which still allows the universe to grow by adding new instances, but it enforces that all generic functions must be defined for that universe. While in theory this approach is not flexible enough to capture subuniverses in their general form, it is quite convenient and it works well in practice. In the evaluation we used this later approach, which explains why EMGM only scores sufficient on the table.

Finally, Smash lets the programmer specify which datatypes are not to be handled by a generic function. Therefore, Smash supports subuniverses "by exclusion".

5.3 Views

Which views does a generic library support? That is, how are datatypes encoded in structure representations and what are the view types used in them? This subsection tries to answer these questions and Figure 9 shows a summary.

The LIGD and EMGM libraries encode datatypes as sums of products, where the sums encode the choice of a constructor and the products encode the fields used in them. This view is usually referred to as the sum-of-products view. The higher-kinded representations for sums of products can be used to abstract over type constructors.

PolyLib represents the structure of regular datatypes using a fixed-point view. This view uses sums and products to encode constructors and their arguments, but additionally a type-level fixed-point constructor to make the recursive occurrences of the datatype explicit. MultiRec extends the fixed-point view to represent systems of mutually recursive datatypes.

The Spine approach uses the Spine datatype to make the application of a constructor to its arguments observable to a generic function. As observed by the authors of this view, the *gfoldl* combinator used in SYB and SYB3 is the catamorphism of the Spine datatype, and hence these approaches also use the spine view. The SYB, SYB3, and Spine approaches also provide a type-spine view, which is used to write producer generic functions. Unlike SYB and SYB3, Spine supports abstraction over $\star \to \star$-types using an additional view called the lifted Spine view.

In the RepLib library datatypes are represented as a list of constructor representations, which are a heterogeneous list of constructor arguments. As in LIGD and EMGM, the structure representation can be adapted to support higher arities for abstraction over type constructors.

In the Uniplate library, the traversal combinators in the core of this library are based on the *uniplate* operation (the other combinators are based on *biplate*). The operation *uniplate*

takes an argument of some type t, and returns the list of children that have the same type t, and a function to reconstruct the argument with new children. This view is similar to the fixed-point view, but fixed-points are not explicitly marked.

It is difficult to point to specific views in the Smash library. Although there are three basic strategies (rewriting, reduction, and lock-step reduction), the rest of the strategies are not defined using any of the more fundamental ones. Hence every traversal strategy can be considered as a way to view the structure of a datatype.

6 Evaluation: Expressiveness

6.1 First-class generic functions

Can a generic function take a generic function as an argument? A library scores 'good' if it can implement *gmapQ* and apply it to *gshow*.

In LIGD, SYB, and Spine, a generic function is a polymorphic Haskell function, so it is a first-class value in Haskell implementations that support rank-2 polymorphism.

EMGM scores 'sufficient' because although EMGM supports first-class generic functions, they are implemented in a rather complicated way. We are forced to go from a relatively simple (but wrong) signature for GMapQ:

$$\textbf{data}\ \mathsf{GMapQ\ a} = \mathit{GMapQ}\ \{\mathit{gmapQ} :: (\ldots \to \mathsf{r}) \to \mathsf{a} \to [\mathsf{r}]\}$$

to a type signature that allows to track calls to the generic function argument. The new signature below abstracts over a type g, the signature of the function argument, and *garg*, which is the generic function argument itself.

$$\begin{aligned}&\textbf{data}\ \mathsf{GMapQ\ g\ a} = \mathit{GMapQ}\ \{\\ &\quad \mathit{garg} \quad :: \mathsf{g\ a},\\ &\quad \mathit{gmapQ} :: (\forall \mathsf{b}\,.\,\mathsf{g\ b} \to \mathsf{b} \to \mathsf{r}) \to \mathsf{a} \to [\mathsf{r}]\}\end{aligned}$$

This makes the definition of *gmapQ*, significantly more complex than other functions, such as generic equality.

In Uniplate, traversal combinators are parametrised over monomorphic functions and not over other generic functions, as is the case in SYB. It is not evident how *gmapQ* would be implemented in Uniplate, hence it scores 'bad.'

In Smash, *gmapQ* is implemented using the *TL_red_shallow* reduction strategy. However, having a new strategy altogether, in place of using an existing one, imposes the burden of one more structure representation for the user. Therefore this library only scores 'sufficient.'

To implement *gmapQ* in PolyP and MultiRec, we need a form of abstraction over type classes. It may be possible to adapt the abstraction technique introduced in SYB3.

6.2 Abstraction over type constructors

Is a generic library able to define the generic functions *gmap* and *crushRight*? If a library can define both functions which can then be used on BinTree and WTree, respectively:

$$\begin{aligned}&\mathit{mapBinTree} :: (\mathsf{a} \to \mathsf{b}) \to \mathsf{BinTree\ a} \to \mathsf{BinTree\ b}\\ &\mathit{flattenWTree} :: \mathsf{WTree\ a\ w} \to [\mathsf{a}]\end{aligned}$$

then the approach scores 'good.' If only one of the definitions is supported, the score is 'sufficient'.

PolyLib includes the definition of *gmap* and *crushRight*. However these functions can be instantiated only to regular datatypes with kind $\star \rightarrow \star$, and so *flattenWTree* is not expressible. Therefore PolyLib scores 'sufficient.'

In Smash, the definition of *gmap* and *crushRight* are supported. Generic map is implemented by means of the rewriting traversal strategy *TL_recon*. This strategy supports ad hoc cases that can change the type of elements, so *gmap* can be instantiated to *mapBinTree*. The definition of *crushRight* uses two special purpose reduction strategies, one for $\star \rightarrow \star$-types and the other for $\star \rightarrow \star \rightarrow \star$-types.

Recent work by (Reinke 2008) and (Kiselyov 2008) shows that SYB supports the definition of *gmap* and *crushRight*. However, SYB scores 'sufficient' only because of complexity in the definitions. For example, the definition of *gmap* uses direct manipulation of type representations, runtime casts, and the *gunfold* combinator. Furthermore, the programmer must take additional steps to ensure the totality of *gmap*. Indeed, the type of the function is too flexible and it can cause runtime failure if instantiated with the wrong types. Hence, the programmer must provide a wrapper that suitably restricts *gmap*'s type.

6.3 Ad hoc definitions for datatypes

Can a generic function have an ad hoc case for a particular datatype? The addition of ad hoc cases should not require the recompilation of the existing code (which happens if the datatype definitions as Rep in LIGD have to be modified). A library scores 'good' if *selectSalary* can be implemented by a using an ad hoc case for the Salary datatype.

We give MultiRec the score 'sufficient' because ad hoc cases are inconvenient for polymorphic datatypes: each polymorphic instance requires a different, redundant type representation datatype.

6.4 Ad hoc definitions for constructors

Can we give an ad hoc definition for a particular constructor, and let the remaining constructors be handled generically? We take the function *rmWeights* as our test. If a library allows the implementation of this function such that an explicit case for the *WithWeight* constructor is given and the remaining constructors are handled generically, then the library scores 'good' on this criterion.

The six approaches that support ad hoc definitions for datatypes, also support ad hoc definitions for constructors. Although LIGD, Spine, and PolyLib strictly speaking do not support ad hoc definitions for datatypes, they can support ad hoc definitions for constructors if we relax the test case, overlooking the need for recompilation when adding ad hoc definitions. LIGD still scores only 'sufficient' because not only recompilation is required but the Rep datatype has to be modified, as we explained in Section 3.2.3.

6.5 Extensibility

Can a programmer extend the universe of a generic function in a different module than that of the definition without the need for recompilation? Libraries that let the generic *gshow*

be extended with a case for printing lists score 'good.' Extensibility is not possible for approaches that do not support ad hoc cases.

To make a generic function extensible, one may try to cheat and give the generic function an extra argument that receives the ad hoc case (or cases). Such a trick is possible with SYB, for example. However, we do not accept it for two reasons. First, this would impose a burden on the user: the generic function has to be "closed" by the programmer before use. Second, functionality that is implemented on top of such an extensible generic function would have to expose the extension argument in its interface. An example of such functionality is discussed by Lämmel & Peyton Jones (2005) in their QuickCheck case study. QuickCheck implements shrinking of test data by using a *shrink* generic function, which should be extensible. If we cheat on the extensibility, the high-level *quickCheck* function would have to include the extension arguments for *shrink*.

6.6 Multiple arguments

Can a generic programming library support a generic function definition that consumes more than one generic argument, such as the generic equality function? If generic equality is definable then the approach scores 'good.' If the definition is more involved than those of other consumer functions (*gshow* and *selectSalary*), then the approach scores 'sufficient.'

The definition of equality in Spine is no more complex than other consumer functions. Therefore Spine scores 'good.' It can be argued, however, that the Spine version is not entirely satisfactory because it ultimately relies on equality of constructor names. Therefore the user could make a mistake when providing a constructor name in the representation.

The SYB library only scores 'sufficient' because defining *geq*, Lämmel & Peyton Jones (2004), is not as direct as for other functions such as *gshow* and *selectSalary*. While the Spine definition *equalSpine* can inspect the two arguments to be compared, in SYB the *gfoldl* combinator forces to process one argument at a time. For this reason, the definition of generic equality has to perform the traversal of the arguments in two stages.

Smash supports multiple arguments to a generic function essentially through currying – a special traversal strategy that traverses several terms in lock-step. However, the fact that a special purpose traversal must be used for functions on multiple arguments imposes a burden on the user. The user has to write one more structure representation per datatype. Therefore Smash only scores 'sufficient.'

The traversal combinators of the Uniplate library are designed for single argument consumers. We have not been able to write a generic equality function for this approach, so Uniplate scores 'bad.'

6.7 Constructor names

Is the approach able to provide the names of the constructors to which the generic function is applied? Library approaches that support the definition of *gshow* score 'good.' With the exception of Uniplate, all generic programming libraries discussed in this paper provide support for constructor names in their structure representations.

	LIGD	PolyLib	SYB	SYB3	Spine	EMGM	RepLib	Smash	Uniplate	MultiRec
geq	28.19	5.97	52.31	86.00	15.03	1.44	7.39	4.00	-	6.50
selectInt	25.00	-	36.00	15.06	13.69	23.25	12.94	14.63	5.81	47.44
rmWeights	3.32	1.00	68.68	5.89	1.85	3.52	3.57	1.83	3.94	1.69
Geom. mean	13	8(2)	51	20	7	5	7	5	16(5)	8

Fig. 8. Preliminary performance experiments showing average overhead (so small numbers are good). The last row shows the rounded geometric mean of the three tests, where a failing test has been replaced by twice the time of the worst succeeding run (in parentheses is shown the geometric mean of the succeeding runs).

6.8 Consumers, transformers, and producers

Generic libraries that can define functions in the three categories: consumers, transformers and producers, score 'good.' This is the case for LIGD, PolyLib, EMGM, RepLib, and MultiRec. If a library uses a different structure representation or type representation for say consumer and producer functions, that library scores 'sufficient.' This is the case for SYB, SYB3, Smash, and Spine. Furthermore, Smash uses a different representation (traversal strategy) for transformers than for consumers, therefore it scores 'sufficient' as well on that criterion. The Uniplate library does not contain combinators to write producer functions, so it scores 'bad.'

7 Evaluation: Usability

7.1 Separate compilation

Is generic universe extension modular? Approaches that can instantiate generic equality to `BinTree` without modifying and recompiling the function definition and the type/structure representation score 'good.'

The Spine library scores 'bad' because the universe extension requires that the datatype, in this case `BinTree`, is represented by a constructor in the GADT that encodes types. Because this datatype is defined in a separate module, recompilation is required.

7.2 Performance

We have used some of the test functions for a performance benchmark but the results are very sensitive to small code differences and compiler optimisations so firm conclusions are difficult to draw. These tests were compiled using GHC 6.8.3 using the optimisation flag `-O2`. As an example, Figure 8 shows running times (in multiples of the running time of a hand-coded monomorphic version) for the *geq*, *selectInt* and *rmWeights* tests. In general, these results show that there is still a significant overhead involved in using these libraries, compared to hand-written code.

For the *geq* test, the compiler manages to produce very efficient code for EMGM, while the SYB family has problems with the two-argument traversal. Uniplate produces the best result for the *selectInt* test. However, for such results, the programmer is expected to manually define a large number of structure representations (as many as the number of datatypes squared). For *rmWeights*, there are several approaches that are within a factor of two of the hand written approach. PolyLib shows the most impressive result: the compiler

produces code as efficient as that of the hand-written version. Using the geometric mean for ranking best overall performance, the best libraries are Smash, EMGM, Spine, RepLib, PolyLib, MultiRec, followed by LIGD and Uniplate and finally SYB3 and SYB.

7.3 Portability

Figure 9 shows that the majority of approaches rely on compiler extensions provided by GHC to some extent. Approaches that are most portable rely on few, and less controversial extensions.

Of all generic programming libraries, LIGD, EMGM, and Uniplate are the most portable. LIGD only relies on one extension to Haskell 2010 – existential types, – which is widely supported and likely to be included in the upcoming Haskell standards. The use of rank-2 types in LIGD is confined to the representations of GRose and NGRose; therefore, rank-2 types are not an essential part of LIGD.

The EMGM library relies on multi-parameter type classes (also a widely supported extension since it is needed for the popular monad transformer libraries) to implement implicit type representations. Multi-parameter type classes make EMGM more convenient to use, but even without it, it would still be possible to do generic programming in EMGM.

In SYB3, overlapping and undecidable instances are needed for the implementation of extensibility and ad hoc cases. Overlapping instances arise because of the overlap between the generic case and the type-specific cases. Undecidable instances are required for the Sat type class, which is used to implement abstraction over type classes. This library and its predecessor (SYB) use rank-2 polymorphism to define the *gfoldl* and *gunfoldl* combinators. These two libraries, as well as RepLib, use *unsafeCoerce* to implement type-safe casts.

GADTs are an essential building block of Spine, RepLib and MultiRec. They are used to represent types and their structure. Moreover, the first two libraries also make use of existential types.

The SYB3, RepLib and MultiRec libraries provide automatic generation of representations, which is implemented using Template Haskell. The SYB library, on the other hand, relies on compiler support for deriving Data and Typeable instances.

The core of Uniplate can be defined in Haskell 98. However, in order to use multi-type traversals, multi-parameter type classes are needed. Uniplate can derive representations by either using a tool that uses Template Haskell, or by relying on compiler support to derive Data and Typeable. However, the use of these extensions is optional, because structure representations can be written by programmers directly.

Smash relies on various extensions such as multi-parameter type classes, undecidable instances, overlapping instances, and functional dependencies. These are needed to implement the techniques for handling ad hoc cases and traversal strategies.

The implementation of MultiRec also uses type families to map datatypes to their respective representations.

7.4 Overhead of library use

How much overhead is imposed on the programmer by using a generic programming library? We are interested in the following aspects:

	LIGD	PolyLib	SYB	SYB3	Spine	EMGM	RepLib	Smash	Uniplate	MultiRec
Implementation mechanisms										
Type representation is GADT					●		●			●
Type representation is D.T.	●									
Rank-2 struc. repr. combinator			●	●						
Type-safe cast			●	●			●			
Extensible rec. and type eval.								●		
Type classes		●	●	●		●	●	●	●	●
Type families										●
Abstraction over type classes				●			●			
Compiler extensions										
Undecidable instances		●		●				●		
Overlapping instances				●				●		
Multi-parameter type classes		●				●		●	●	●
Functional dependencies								●		
Type families										●
Rank-2 polymorphism			●	●						●
Existential types	●				●		●			
GADTs					●		●			●
unsafeCoerce			●	●			●			
Template Haskell				●			●		●	●
Derive Data and Typeable			●						●	
Views										
Fixed-point view		●								●
Sum-of-products	●					●				
Spine			●	●	●					
Lifted spine					●					
RepLib							●			
Uniplate									●	
Smash								●		

Fig. 9. Implementation mechanisms, required compiler extensions, and views.

7.4.1 Automatic generation of representations

The libraries that offer support for automatic generation of representations are SYB, SYB3, EMGM, RepLib, Uniplate and MultiRec. Note that automatic generation of representations in all approaches is limited to datatypes with no arguments of higher-kinds, hence GRose and NGRose are not supported.

PolyLib used to include support for generation of representations, but this functionality broke with version 2 of Template Haskell.

The SYB library relies on GHC to generate Typeable and Data instances for new datatypes. The SYB library scores 'good' on this criterion.

The SYB3 library also uses Template Haskell to generate representations. However, the generated representations for BinTree cause non-termination when type-safe casts are used on a BinTree value. This is a serious problem: regular datatypes and type-safe casting are very commonly used. Hence this approach does not score 'good' but 'sufficient.'

The EMGM library on Hackage (see `http://hackage.haskell.org/package/emgm`) supports automatic generation using Template Haskell.

The RepLib library includes Template Haskell code to generate structure representations for new datatypes in its distribution. However, RepLib does not support the generation of representations for arity two generic functions and does not include the machinery for such representations. Furthermore, automatic generation fails when a type synonym is used in a datatype declaration. RepLib scores 'sufficient' because of these problems.

Uniplate can use the Typeable and Data instances of GHC for automatic generation of representations. Furthermore, a separate tool, based on Template Haskell, is provided to derive instances. The Uniplate library scores 'good' on this criterion.

MultiRec also includes automatic generation of structure representations based on Template Haskell. Unfortunately, generation does not work for parametrised datatypes such as lists, and in those cases the user must manually define structure representations. MultiRec only scores 'sufficient' in this criterion.

7.4.2 Number of structure representations

PolyLib only supports regular datatypes of kind $\star \to \star$, thus it only has one representation. It would be straightforward to add a new representation for each kind, but it would still only support regular datatypes. MultiRec is equipped with one representation only.

The LIGD, EMGM, and RepLib libraries have two sorts of representations: (1) a representation for $\star$-types (for example Rep in Section 2), and (2) representations for type constructors, which are arity-based Hinze (2000, 2006). The latter consist of a number of arity-specific representations. For example, to write the *gmap* function we would have to use a representation of arity two. Which arities should be supported? Probably the best approach is to support those arities for which useful generic functions are known: *crush* (arity one), *gmap* (arity two), and generic zip (arity three). This makes a total of four representations needed for these libraries, one to represent $\star$-types, and three for all useful arities.

The functions of arity one can also be defined using a representation of arity three (Hinze 2006) by ignoring the extra type arguments, and hence we could remove representations

of arities one and two. The next step is to subsume the representation for $\star$-types with the arity three representation. This makes using some approaches less convenient — for instance, the EMGM approach loses the generic dispatcher. Although reducing the number of representations is possible, we do not do so in this comparison, because it is rather inelegant. Defining generic equality using a representation of arity three would leave a number of unused type variables that might make the definition confusing.

When a new datatype is used with SYB/SYB3, the structure representation is given in a Data instance. This instance has two methods *gfoldl* and *gunfold* which are used for consumer and transformer, and producer generic functions, respectively. Therefore every new datatype needs two representations to be used with SYB/SYB3 functions.

The Spine library needs three structure representations per datatype. The first, the spine representation, is used for consumer and transformer functions. The second, the type spine view, is used with producer functions. The third representation is used for write generic functions that abstract over type constructors of kind $\star \to \star$.

In Uniplate, the number of structure representations that are needed can range from one Uniplate instance per datatype, to $O(n^2)$ instances for a system of n datatypes, when maximum performance is wanted. For our comparison, we assume that one representation is needed.

Smash is specifically designed to allow an arbitrary number of custom traversal strategies. Although three strategies are fundamental — reconstruction, reduction, and parallel traversal — the simplified variations of these turn out to be more convenient and frequently used. However, this also means that the programmer defines more structure representations than in other libraries. The representations that are used in this evaluation are eight: a rewriting strategy, a reduction strategy, a reduction strategy with constructor names, a twin traversal strategy, a shallow reduction traversal, two traversals for abstracting over type constructors of kinds $\star \to \star$ and $\star \to \star \to \star$, and a traversal strategy for producer functions.

7.4.3 Work to instantiate a generic function

Ideally, having the definition of a generic function and the structure representation for a datatype should be sufficient for applying the generic function. In this case, the total work to instantiate a generic function amounts to defining the structure representation and the generic function.

The EMGM, LIGD, Spine and MultiRec approaches score ‘good’: all that is needed is to apply the generic function to the appropriate type representation. Type-class based approaches, such as SYB, Smash, Uniplate, and SYB3 require even less effort because the type representation passed to the generic function is implicit.

In contrast, the RepLib library requires an instance declaration that enables the use of a generic function on a datatype. In addition to defining generic functions and structure representation, the user must also spend additional effort defining instances for every use of a generic function on a datatype. On the other hand, this allows precise control over the domain of a generic function, which fulfils the subuniverses criterion. A similar trade-off happens in the original EMGM version proposed by Oliveira *et al.* (2006), as already discussed in Section 5.2.

7.4.4 Work to define a generic function

Is it easy to write a simple generic function? A library scores 'good' if it requires roughly one definition per generic function case, with no need for additional artifacts.

The LIGD, SYB3, and RepLib libraries score 'bad' on this criterion. In LIGD, the definitions of generic functions become more verbose because of the emulation of GADTs. However, this overhead does not arise in implementations of LIGD that use GADTs directly. In RepLib we need to define a dictionary datatype, and an instance of the Sat type class, in addition to the type class definition that implements the generic function. In SYB3 the definitions needed (besides a type class for the function) are the dictionary type, a Sat instance for the dictionary, and a dictionary proxy value. Therefore these libraries only score 'sufficient.'

7.5 Practical aspects

For this criterion we consider aspects such as

- there is a library distribution available on-line,
- the library interface is well-documented,
- and other aspects, such as a modular interface, definitions of common generic functions, etc.

The LIGD and Spine libraries do not have distributions on-line. These libraries score 'bad.' PolyLib has an on-line distribution (as part of PolyP version 2) but the library is not maintained anymore.

The SYB library is distributed with the GHC compiler. This distribution includes a number of traversal combinators for common generic programming tasks and Haddock documentation. The GHC compiler supports the automatic generation of Typeable and Data instances. This library scores 'good.'

The official SYB3 distribution failed to compile with the versions of GHC that we used in this comparison (6.6, 6.8.1, 6.8.2). This distribution can be downloaded from: `http://homepages.cwi.nl/~ralf/syb3/code.html`. There is an alternative distribution of this library which is available as a Darcs repository from: `http://happs.org/HAppS/syb-with-class`. This distribution is a cabal package that includes Haddock documentation for the functions that it provides. However, it does not contain many important combinators such as *gmapAccumQ* and *gzipWithQ*.

The EMGM, RepLib, Uniplate and MultiRec libraries have on-line distributions at the HackageDB web site. All libraries all well-structured, have a number of useful pre-defined functions, and come with Haddock documentation. All these libraries score 'good.'

The Smash library is distributed as a stand-alone module that can be downloaded from `http://okmij.org/ftp/Haskell/generics.html#Smash`. There are a few example functions in that module that illustrate the use of the approach. However, the library is not as well structured or documented as other generic programming libraries.

7.6 Ease of learning and use

It is hard to determine how easy it is to learn using a generic programming library. We approximate this criterion by looking at the complexity underlying the mechanisms used in the implementation of the libraries. Below we describe the difficulties that arise with some of the implementation mechanisms:

1. *Rank-2 structure representation combinators.* There are two problems with rank-2 structure representation combinators. First, rank-2 polymorphic types are difficult to understand for beginning users. This implies that defining a generic function from scratch — that is, using the type structure directly, bypassing any pre-defined combinators — presents more difficulties than in other approaches. Second, structure representation combinators such as those used in SYB can be used directly to define functions that consume one argument. But if two arguments are to be consumed instead (which is the case in generic equality), then the definition of the function becomes complicated.
2. *Extensible records and type-level evaluation.* The techniques to encode extensible records make advanced use of type classes and functional dependencies. This encoding can be troublesome to the beginning generic programmer in at least one way: type errors arising from such programs can be very difficult to debug.
3. *Abstraction over type classes.* Abstraction over type classes is emulated by means of explicit dictionaries that represent the class that is being abstracted. This is an advanced type class programming technique which might confuse beginning generic programmers.
4. *Arity based representations.* Arity based representations are used to represent type constructors. However, it is more difficult to relate the type signature of an arity-based generic function to that of an instance. For example, generic map has type $\mathsf{Rep2\ a\ b} \to \mathsf{a} \to \mathsf{b}$, which does not bear a strong resemblance to the type of the $\mathsf{BinTree}$ instance of map: $(\mathsf{a} \to \mathsf{b}) \to \mathsf{BinTree\ a} \to \mathsf{BinTree\ b}$. For this reason, programming with arity-based representations might be challenging to a beginner.

The approaches that suffer from the first difficulty are SYB and SYB3. The second difficulty affects Smash. The third difficulty affects SYB3 and RepLib. The fourth difficulty affects LIGD, EMGM, and RepLib. However, the first two libraries need such arities only occasionally, namely to define functions such as *gmap* and *crushRight*.

Another problem that can impede learning and using an approach is the use of a relatively large number of implementation mechanisms. This is the case for SYB3, RepLib and MultiRec. While it is possible to work out how one of the many mechanisms, for example GADTs in RepLib, is used when writing a generic function, it is much more difficult to understand the interactions of all the mechanisms in the same library. This, we believe, will make it more difficult for new users to learn and use SYB3, RepLib and MultiRec.

7.7 Implementation mechanisms

What are the mechanisms through which a library encodes a type or its representation? We have the following options:

- *Types and structure represented by GADTs.* Types are encoded by a representation GADT, where each type is represented by a constructor. The GADT may also have a constructor which encodes datatypes structurally (like *RType* in this paper).
- *Types and structure represented by datatypes.* When GADTs are not available, it is still possible to emulate them by embedding conversion functions in the datatype constructors. In this way a normal datatype can represent types and their structure.
- *Rank-2 structure representation combinators.* Yet another alternative is to represent the structure of a datatype using a rank-2 combinator such as *gfoldl* in SYB.
- *Type-safe cast.* Type-safe casts are used to implement type-specific functionality, or ad hoc cases. Type-safe casting attempts to convert `a`-values into `b`-values at runtime. Statically it may not be known that `a` and `b` are the same type, but if this is the case at runtime, the conversion succeeds.
- *Extensible records and type-level evaluation.* The work of Kiselyov *et al.* (2004) introduces various techniques to implement extensible records in Haskell. The techniques make advanced use of type classes to perform record lookup statically. This operation is an instance of a general design pattern: encoding type-level computations using multi-parameter type classes and functional dependencies.
- *Type classes.* Type classes may be used in a variety of ways: to represent types and their structure and perform case analysis on them, to overload structure representation combinators, and to provide extensibility of generic functions.
- *Type families.* Type families are used to map a datatype or a system of datatypes to their corresponding structure representation.
- *Abstraction over type classes.* Generic programming libraries that support extensible generic functions do so by using type classes. Some of these approaches, however, require a form of abstraction over type classes, which can be encoded by a technique that uses explicit dictionaries, popularised by Lämmel & Peyton Jones (2005).

The LIGD and Spine libraries represent types and structure as datatypes and GADTs respectively, while EMGM uses type classes to do so. In SYB and SYB3, datatypes are structurally represented by rank-2 combinators *gfoldl* and *gunfold*. Ad hoc cases in SYB are given using pre-defined combinators such as *extQ* and *mkQ*, which are implemented using type-safe casts.

In SYB3, case analysis over types is performed directly by the type class system, because generic functions are type classes. However, type-safe casts are still important because they are used to implement functions such as equality. Furthermore, this approach implements abstraction over type classes to support extensibility.

The RepLib library is an interesting combination. It has a datatype that represents types and their structure, and so generic functions are defined by pattern matching on those representation values. On the other hand, RepLib also uses type classes to allow the extension of a generic function with a new type-specific case. To allow extension, type classes must encode type class abstraction using the same technique as used in SYB3. Optionally, the RepLib library allows the programmer to use a programming style reminiscent of SYB, where ad hoc cases are aggregated using functions such as *extQ* and *mkQ*. These combinators are implemented using the GADT-based representations and type-safe casts.

The Smash library uses an extensible record-like list of functions to encode ad hoc cases. Case analysis on types is performed by a lookup operation, which in turn is implemented

by type-case. This library also uses type-level evaluation to determine the argument and return types of method *gapp*, based on the traversal strategy and the list of ad hoc cases.

MultiRec is the only approach that uses type families. Type families map a type that represents a system of datatypes onto a structure representation. This representation is built out of pre-defined GADT type constructors and associated to individual datatypes in the system by means of multi-parameter type classes.

8 Conclusions

We have introduced a set of criteria to compare libraries for datatype-generic programming in Haskell. These criteria can be viewed as a characterisation of generic programming in Haskell. Furthermore, we have designed a generic programming test suite: a set of characteristic examples that check whether or not criteria are supported by generic programming libraries. Using the criteria and the test suite, we have compared ten approaches to generic programming in Haskell. Since the publication of the original Haskell Symposium paper, our terminology and criteria have been used in three new generic programming libraries MultiRec (Rodriguez Yakushev *et al.* 2009), Alloy (Brown & Sampson 2009) and "Instant Generics" (Chakravarty *et al.* 2009). We have included one of these (MultiRec) in full detail in this extended version; for the other two libraries we refer to the corresponding papers.

Is it possible to combine the libraries into a single one that has a perfect score? Our comparison seems to suggest otherwise. A good score on one criterion generally causes problems in another. For example, approaches with extensible generic functions sometimes have problems that are absent in non-extensible ones. The SYB3 library is extensible but defining a generic function requires more boilerplate than in SYB. Furthermore, SYB3 has a smaller universe than SYB. And while the EMGM library provides extensible generic functions, defining higher-order generic functions is far from trivial. Another difficulty is that generic libraries use particular type and structure representations that may differ and are usually incompatible.

What is the best generic programming library? Since no library has good scores on all criteria, the answer depends on the scenario at hand. Some libraries, such as LIGD, PolyLib and Spine, score 'bad' on important criteria such as ad hoc cases, separate compilation and universe size (support for mutual recursion), and are unlikely candidates for practical use. The other libraries all have their particular application areas in which they shine. We now discuss which libraries are most suitable for implementing one of the three typical generic programming scenarios introduced in Section 3.

Consider the criteria required for transformation traversals. Implementing traversals over abstract syntax trees requires support for mutually recursive datatypes. Furthermore, traversals are higher-order generic functions since they are parametrised by the actual transformations. The libraries that best satisfy these criteria are SYB and Uniplate. Uniplate does not support higher-orderness, but Uniplate functions are monomorphic, so that criterion is not needed. If extensible traversals are needed and the additional work to define a generic function is not a problem, we can also use SYB3 or RepLib.

The criterion needed for operating over the elements of a container datatype is abstraction over type constructors. Ad hoc cases are also commonly needed to process a container

in a particular way. The libraries that best fit this scenario are EMGM and RepLib. Smash can also be used but the high number of representations may demand more effort.

For serialisation, one can use approaches that have good scores on constructor names, producers, ad hoc cases and universe size (mutual recursion), namely EMGM and RepLib. SYB, SYB3 and Smash can also be used, if one is willing to learn a different API for writing a producer function.

Pick your choice!

Acknowledgements. This research has been partially funded by: the Netherlands Organisation for Scientific Research (NWO), via the Real-life Datatype-Generic programming project, project nr. 612.063.613; the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST) / Korea Science and Engineering Foundation (KOSEF) grant number R11-2008-007-01002-0; and the Mid-career Researcher Program (2010-0022061) through NRF grant funded by the MEST. We thank J. Gibbons, S. Leather and J.P. Magalhães for their thoughtful comments and suggestions. We also thank the participants of the generics mailing list for the discussions and the code examples that sparked the work for this paper. In particular, S. Weirich and J. Cheney provided some of the code on which our test suite is based. Andres Löh provided useful comments and formatting tips. Finally, we thank the anonymous reviewers whose comments were very helpful in improving both the presentation and the contents of the paper.

References

Alimarine, Artem, & Plasmeijer, Marinus J. (2001). A generic programming extension for Clean. *Pages 168–185 of: IFL'01*. LNCS, vol. 2312. Springer.

Bernardy, Jean-Philippe, Jansson, Patrik, Zalewski, Marcin, & Schupp, Sibylle. (2010). Generic programming with c++ concepts and haskell type classes — a comparison. *JFP*, **20**(3–4), 271–302.

Bird, Richard, & Meertens, Lambert. (1998). Nested datatypes. *Pages 52–67 of:* Jeuring, J. (ed), *MPC'98*. LNCS, vol. 1422. Springer.

Bringert, Björn, & Ranta, Aarne. (2006). A pattern for almost compositional functions. *Pages 216–226 of: ICFP'06*.

Brown, Neil C.C., & Sampson, Adam T. (2009). Alloy: fast generic transformations for Haskell. *Pages 105–116 of: Haskell Symposium'09*.

Chakravarty, Manuel M. T., Ditu, Gabriel, & Leshchinskiy, Roman. (2009). *Instant generics*. Available from `www.cse.unsw.edu.au/~chak/papers/CDL09.html`.

Cheney, James, & Hinze, Ralf. (2002). A lightweight implementation of generics and dynamics. *Pages 90–104 of: Haskell Workshop'02*.

Clarke, Dave, & Löh, Andres. (2003). Generic haskell, specifically. *Pages 21–47 of: Proc. IFIP TC2/WG2.1 working conference on generic programming*. Kluwer.

Garcia, Ronald, Järvi, Jaakko, Lumsdaine, Andrew, Siek, Jeremy, & Willcock, Jeremiah. (2007). An extended comparative study of language support for generic programming. *JFP*, **17**(2), 145–205.

Gibbons, Jeremy. (2007). Datatype-generic programming. *Pages 1–71 of: Spring school on datatype-generic programming*. LNCS, vol. 4719. Springer.

Gibbons, Jeremy, & Paterson, Ross. (2009). Parametric datatype-genericity. *Pages 85–93 of: WGP '09*.

Hinze, Ralf. (2000). Polytypic values possess polykinded types. *Pages 2–27 of: MPC'00*. LNCS, vol. 1837. Springer.

Hinze, Ralf. (2006). Generics for the masses. *JFP*, **16**, 451–482.

Hinze, Ralf, & Löh, Andres. (2006). "Scrap Your Boilerplate" revolutions. *Pages 180–208 of: MPC'06*. LNCS, vol. 4014. Springer.

Hinze, Ralf, & Löh, Andres. (2009). Generic programming in 3D. *SCP*, **74**(8), 590–628.

Hinze, Ralf, Löh, Andres, & Oliveira, Bruno C. d. S. (2006). "Scrap Your Boilerplate" reloaded. *FLOPS'06*. LNCS, vol. 3945. Springer.

Hinze, Ralf, Jeuring, Johan, & Löh, Andres. (2007). Comparing approches to generic programming in Haskell. *Pages 72–149 of: Spring school on datatype-generic programming*. LNCS, vol. 4719. Springer.

Holdermans, Stefan, Jeuring, Johan, Löh, Andres, & Rodriguez, Alexey. (2006). Generic views on data types. *Pages 209–234 of: MPC'06*. LNCS, vol. 4014. Springer.

Jansson, Patrik, & Jeuring, Johan. (1997). PolyP — a polytypic programming language extension. *Pages 470–482 of: POPL'97*.

Jansson, Patrik, & Jeuring, Johan. (2002). Polytypic data conversion programs. *SCP*, **43**(1), 35–75.

Jeuring, Johan, Leather, Sean, Pedro MagalhÃ£es, JosÃ©, & Rodriguez Yakushev, Alexey. (2009). Libraries for generic programming in haskell. *Pages 165–229 of:* Koopman, Pieter, Plasmeijer, Rinus, & Swierstra, Doaitse (eds), *Advanced functional programming*. Lecture Notes in Computer Science, vol. 5832. Springer.

Karvonen, Vesa A.J. (2007). Generics for the working ML'er. *Pages 71–82 of: ML Workshop '07*.

Kiselyov, Oleg. (2006). *Smash your boilerplate without class and typeable*. `http://article.gmane.org/gmane.comp.lang.haskell.general/14086`.

Kiselyov, Oleg. (2008). *Compositional gMap in SYB1*. `http://www.haskell.org/pipermail/generics/2008-July/000362.html`.

Kiselyov, Oleg, Lämmel, Ralf, & Schupke, Keean. (2004). Strongly typed heterogeneous collections. *Pages 96–107 of: Haskell Workshop'04*.

Lämmel, Ralf, & Peyton Jones, Simon. (2003). Scrap your boilerplate: A practical design pattern for generic programming. *Pages 26–37 of: TLDI'03*.

Lämmel, Ralf, & Peyton Jones, Simon. (2004). Scrap more boilerplate: reflection, zips, and generalised casts. *Pages 244–255 of: ICFP'04*.

Lämmel, Ralf, & Peyton Jones, Simon. (2005). Scrap your boilerplate with class: extensible generic functions. *Pages 204–215 of: ICFP'05*.

Lämmel, Ralf, & Visser, Joost. (2003). A Strafunski application letter. *Pages 357–375 of: Proc. practical aspects of declarative programming, PADL 2003*. LNCS, vol. 2562.

Löh, Andres, Jeuring, Johan, Noort, Thomas van, Rodriguez, Alexey, Clarke, Dave, Hinze, Ralf, & de Wit, Jan. (2008). *The Generic Haskell user's guide, Version 1.80 - Emerald release*. Tech. rept. UU-CS-2008-011. Utrecht University.

Magalhães, José Pedro, Holdermans, Stefan, Jeuring, Johan, & Löh, Andres. (2010). Optimizing generics is easy! *Pages 33–42 of: PEPM '10*.

Meertens, Lambert. (1996). Calculate polytypically! *Pages 1–16 of:* Kuchen, H., & Swierstra, S. D. (eds), *PLILP*. LNCS, vol. 1140.

Mitchell, Neil. (2009). Deriving a relationship from a single example. *AAIP 2009: Proc. ACM SIGPLAN workshop on approaches and applications of inductive programming*. Available from `http://www.cogsys.wiai.uni-bamberg.de/aaip09/`.

Mitchell, Neil, & Runciman, Colin. (2007). Uniform boilerplate and list processing. *Pages 49–60 of: Haskell'07*.

Moors, Adriaan, Piessens, Frank, & Joosen, Wouter. (2006). An object-oriented approach to datatype-generic programming. *Pages 96–106 of: WGP'06*.

Norell, Ulf, & Jansson, Patrik. (2004). Polytypic programming in Haskell. *Pages 168–184 of: IFL'03*. LNCS, vol. 3145. Springer.

Oliveira, Bruno C. d. S., & Gibbons, Jeremy. (2010). Scala for generic programmers. *JFP*, **20**(Special Issue 3–4), 303–352.

Oliveira, Bruno C. d. S., Hinze, Ralf, & Löh, Andres. (2006). Extensible and modular generics for the masses. *Pages 199–216 of: Trends in functional programming*.

Peyton Jones, S., Vytiniotis, D., Weirich, S., & Washburn, G. (2006). Simple unification-based type inference for GADTs. *Pages 50–61 of: ICFP'06*.

Reinke, Claus. (2008). *Traversable functor data, or: X marks the spot*. `http://www.haskell.org/pipermail/generics/2008-June/000343.html`.

Rodriguez Yakushev, Alexey, Jeuring, Johan, Jansson, Patrik, Gerdes, Alex, Kiselyov, Oleg, & d. S. Oliveira, Bruno C. (2008). Comparing libraries for generic programming in Haskell. *Pages 111–122 of: Haskell symposium '08*.

Rodriguez Yakushev, Alexey, Holdermans, Stefan, Löh, Andres, & Jeuring, Johan. (2009). Generic programming with fixed points for mutually recursive datatypes. *Pages 233–244 of: ICFP'09*.

Runciman, Colin, Naylor, Matthew, & Lindblad, Fredrik. (2008). SmallCheck and Lazy SmallCheck: Automatic exhaustive testing for small values. *Pages 37–48 of:* Gill, Andy (ed), *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on haskell*.

Wadler, Philip. (1989). Theorems for free! *Pages 347–359 of: FPCA'89*.

Weirich, Stephanie. (2006). RepLib: a library for derivable type classes. *Pages 1–12 of: Haskell'06*.

Winstanley, Noel, & Meacham, John. (2006). *DrIFT user guide*. `http://repetae.net/~john/computer/haskell/DrIFT/`.

Yallop, Jeremy. (2007). Practical generic programming in OCaml. *Pages 83–94 of: ML '07: Proc. 2007 workshop on ML*.