

Comparing Libraries for Generic Programming in Haskell

Alexey Rodriguez

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-2008-010

www.cs.uu.nl

ISSN: 0924-3275

Comparing Libraries for Generic Programming in Haskell

Alexey Rodriguez

Utrecht University, The Netherlands
alexey@cs.uu.nl

Johan Jeuring

Utrecht University, The Netherlands
johanj@cs.uu.nl

Patrik Jansson

Chalmers University of Technology &
University of Gothenburg, Sweden
patrikj@chalmers.se

Alex Gerdes

Open University, The Netherlands
alex.gerdes@ou.nl

Oleg Kiselyov

FNMOC, USA
oleg@pobox.com

Bruno C. d. S. Oliveira

Oxford University, UK
bruno.oliveira@comlab.ox.ac.uk

Abstract

Datatype-generic programming is defining functions that depend on the structure, or “shape”, of datatypes. It has been around for more than 10 years, and a lot of progress has been made, in particular in the lazy functional programming language Haskell. There are more than 10 proposals for generic programming libraries or language extensions for Haskell. To compare and characterise the many generic programming libraries in a typed functional language, we introduce a set of criteria and develop a generic programming benchmark: a set of characteristic examples testing various facets of datatype-generic programming. We have implemented the benchmark for nine existing Haskell generic programming libraries and present the evaluation of the libraries. The comparison is useful for reaching a common standard for generic programming, but also for a programmer who has to choose a particular approach for datatype-generic programming.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism

General Terms Design, Measurement

Keywords datatype-generic programming, libraries comparison

1. Introduction

Software development often consists of designing a datatype to which functionality is added. Some functionality is datatype specific. Other functionality is defined on almost all datatypes, and only depends on the structure of the datatype; this is called datatype-generic functionality. Examples of such functionality are comparing two values for equality, searching a value of a datatype for occurrences of a particular string or other value, editing a value, pretty-printing a value, etc. Larger examples include XML tools, testing frameworks, debuggers, and data-conversion tools.

Datatype-generic programming has been around for more than 10 years now. A lot of progress has been made in the last decade, in particular with generic programming in Haskell. There are more than 10 proposals for generic programming libraries or language extensions for Haskell. Such libraries and extensions are also starting to appear for other programming languages, such as ML.

Although generic programming has been used in several applications, it has few users for real-life projects. This is understand-

able. Developing a large application takes a couple of years, and choosing a particular approach to generic programming for such a project involves a risk. Few approaches that have been developed over the last decade are still supported, and there is a high risk that the chosen approach will not be supported anymore, or that it will change in a backwards-incompatible way in a couple of years time.

The Haskell Refactorer HaRe [Li et al., 2003] is an exception, and provides an example of a real-life project in which a generic-programming technique (Strafunski [Lämmel and Visser, 2002]) is used to implement traversals over a large abstract syntax tree. However, this project contains several other components that could have been implemented using generic-programming techniques, such as rewriting, unification, and pretty-printing modules. These components are much harder to implement than traversals over abstract-syntax trees. Had these components been implemented generically, we claim that, for example, the recent work of the HaRe team to adapt the refactoring framework to the Erlang language [Derick and Thompson, 2005] would have been easier. Other projects that use generic programming are the Haskell Application Server (HAppS), which uses the extensible variant of Scrap Your Boilerplate, and the Catch and Reach tools [Mitchell and Runciman, 2007a, Naylor and Runciman, 2007], which use the Uniplate library to implement traversals.

It is often not immediately clear which generic programming approach is best suited for a particular project. There are generic functions that are difficult or impossible to define in certain approaches. The datatypes to which a generic function can be applied, and the amount of work a programmer has to do per datatype and/or generic function varies among different approaches.

The current status of generic programming in Haskell is comparable to the lazy Tower of Babel preceding the birth of Haskell in the eighties [Hudak et al., 2007]. We have many single-site languages or libraries, each individually lacking critical mass in terms of language/library-design effort, implementations, and users.

How can we decrease the risk in using generic programming? Our eventual goal is to design a *common generic programming library* for Haskell. To increase the chances of continuing support, we would develop this library in an international committee. The rationale for developing a library for generic programming instead of a language extension is that Haskell is powerful enough to write generic programs that previously needed the support of language extensions such as PolyP [Jansson and Jeuring, 1997] or Generic Haskell [Löh et al., 2003]. Furthermore, compared with a

language extension, a library is much easier to ship, support, and maintain. The library might be accompanied by tools that depend on non-standard language extensions, for example for generating embedding-projection pairs, as long as the core is standard Haskell. The standard that the library design should target is Haskell 98 and widely-accepted extensions (such as existential types and multi-parameter type classes) that are likely to be included in the next Haskell standard [Haskell Prime list, 2006]. The library should support the most common generic programming scenarios, so that programmers can define the generic functions that they want and use them with the datatypes they want.

To design a common generic programming library, we first have to evaluate existing libraries to find out differences and commonalities, and to be able to make well-motivated decisions about including and excluding features. In this paper we take the first step towards our goal. We design a framework to compare generic programming libraries in an expressive functional programming language, and apply this framework to Haskell. We will evaluate and compare the following libraries:

- Lightweight Implementation of Generics and Dynamics (LIGD) [Cheney and Hinze, 2002]
- Polytypic programming in Haskell (PolyLib) [Norell and Jansson, 2004]
- Scrap your boilerplate (SYB) [Lämmel and Peyton Jones, 2003, 2004]
- Scrap your boilerplate, extensible variant using typeclasses (SYB3) [Lämmel and Peyton Jones, 2005]
- Scrap your boilerplate, spine view variant (Spine) [Hinze et al., 2006, Hinze and 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 and Runciman, 2007b]

Note that this list does not contain generic programming language extensions such as PolyP, Generic Haskell, or Template Haskell [Lynagh, 2003], and no pre-processing approaches to generic programming such as DrIFT [Winstanley and Meacham, 2006], and Data.Derive. We strictly limit ourselves to library approaches, which, however, might be based on particular compiler extensions. The SYB and Strafanski [Lämmel and Visser, 2003] approaches are very similar, and therefore we only take the SYB approach into account in this evaluation. The functionality of the Compos library [Bringert and Ranta, 2006] is subsumed by Uniplate, and hence we only evaluate the latter.

We evaluate existing libraries by means of a set of criteria. Papers about generic programming usually give desirable criteria for generic programs. Examples of such criteria are: can a generic function be extended with special behaviour on a particular datatype, and are generic functions first-class, that is, can they take a generic function as argument. We develop a set of criteria based on our own ideas about generic programming, and ideas from papers about generic programming. For most criteria, we have a generic test function that determines whether or not the criterion is satisfied. These test functions together form a benchmark which we try to implement for the different approaches.

We are aware of three existing comparisons of support for generic programming in programming languages. Garcia et al. [2007] and Bernardy et al. [2008] compare the support for *property-based* 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. Hinze et al. [2007] compare various approaches to datatype-generic programming in Haskell. However, most of the covered approaches are language extensions, and many of the recent library approaches have not been included.

This paper has the following contributions:

- It gives an extensive set of *criteria for comparing libraries for generic programming in Haskell*. The criteria might be viewed as a characterisation of generic programming in Haskell.
- It develops a *generic programming benchmark*: a set of characteristic examples with which we can test the criteria for generic programming libraries.
- It compares nine existing library approaches to generic programming in Haskell with respect to the criteria, using the implementation of the benchmark in the different libraries.
- The benchmark itself is a contribution. It can be seen as a cookbook that illustrates how different generic programming tasks are achieved using the different approaches. Furthermore, its availability makes it easier to compare the expressiveness of future generic programming libraries. The benchmark suite can be obtained following the instructions at <http://haskell.org/haskellwiki/GPBench>.

The outcome of this evaluation is not necessarily restricted to the context of Haskell. We think this comparison will be relevant for other programming languages as well. This paper will be useful for a programmer that develops a generic programming library in Haskell, and for a programmer with knowledge of the concepts behind generic programming that wants to select a library for a particular purpose, which requires generic programming techniques. We assume the reader is familiar with generic programming.

This paper is organised as follows. Section 2 introduces datatype-generic programming concepts and terminology. Section 3 shows the design and contents of the benchmark suite. Section 4 introduces and discusses the criteria we use for comparing libraries for generic programming in Haskell. Section 5 summarises the evaluation of the different libraries with respect to the criteria, using the benchmark. Section 6 presents the evaluation in full detail. Section 7 concludes.

2. Generic programming: concepts and terminology

This section introduces and illustrates generic programming using a simplified form of the datatype-generic programming library LIGD. We use LIGD because the encoding mechanisms of this library are simple and easier to understand than those of other more advanced libraries. The original LIGD paper [Cheney and Hinze, 2002] gives a more detailed explanation of this approach.

In polymorphic lambda calculus it is impossible to write one parametrically polymorphic equality function that works on all datatypes [Wadler, 1989]. That is why the definition of equality in Haskell uses type classes, and ML uses equality types. The Eq type class provides the equality operator `==`, which is overloaded for a family of types. To add a newly defined datatype to this family, a programmer defines an instance of equality for it. Thus, a programmer manually writes definitions of equality for every new datatype that is defined. For equality, type class deriving automates this process. However, this mechanism can only be used with a small number of type classes because it is hardwired into the language, making it impossible to extend or change by the programmer.

With generic programming, we can define equality once and use it on a large family of datatypes. Such functions are called

```

geq :: Rep a → a → a → Bool
geq (RUnit)      Unit      Unit      = True
geq (RSum ra rb) (Inl a1) (Inl a2) = geq ra a1 a2
geq (RSum ra rb) (Inr b1) (Inr b2) = geq rb b1 b2
geq (RSum ra rb) _      _      = False
geq (RProd ra rb) (Prod a1 b1) (Prod a2 b2)
  = geq ra a1 a2 ∧ geq rb b1 b2

```

Figure 1. Type-indexed equality function in the LIGD library

```

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

```

Figure 2. Unit, sum and product datatypes

generic functions. The introduction of a new datatype does not require redefinition or extension of an existing generic function. We merely need to describe the new datatype to the library, and all existing and future generic functions will be able to handle it.

Below we give a brief introduction to generic programming and the terminology that we use throughout this paper.

A *type-indexed function* (TIF) is a function that is defined on every type of a family of types. We say that the types in this family index the TIF, and we call the type family a universe. A TIF is defined by case analysis on types: each type is assigned a function that acts on values of that type. As a familiar example, consider the TIF equality implemented using Haskell type classes. The universe consists of the types that are instances of the Eq type class. Equality is given by the == method of the corresponding instance. And the case analysis on types is provided by instance selection.

Haskell type classes are only one of the possible implementations of TIFs. In this section we use LIGD with Generalised Algebraic Datatypes (GADTs) [Peyton Jones et al., 2006] to implement TIFs. We start with the equality TIF which is indexed by a universe consisting of units, sums and products. Figures 1 and 2 show the definitions. Note that in Haskell type variables appearing in type signatures are implicitly universally quantified. The first argument of the function is a *type representation*, which describes the type of the values that are to be compared (second and third arguments). Haskell does not allow functions to depend on types, so here types are represented by a GADT. This has the advantage that case analysis on types can be implemented by pattern matching, a familiar construct to functional programmers. The GADT represents the types of the universe consisting of units, sums and products:

```

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)

```

geq has three *type-indexed function cases*, one for each of the base types of the universe.

TIF *instantiation* is the process by which we make a TIF specific to some type t, so that we can apply the resulting function to t values. In LIGD the instantiation process is straightforward: geq performs a fold over Rep t using pattern matching, and builds an equality function that can be used on t values. In other approaches, instantiation uses, for example, the type class system.

Now we want to instantiate equality on lists. Since a generic function can only be instantiated on types in the universe, we extend our universe to lists. There are two ways to do this. The first is *non-generic extension*, we extend our case analysis on types so that lists are handled by equality. In LIGD, this translates into the following: extend Rep with an RList constructor that represents lists, and extend equality with a case for RList:

```
geq (RList ra) xs ys = ...
```

The second way to implement extension is *generic extension*: we describe the structure of the list datatype in terms of types inside the universe. The consequence is that instantiation to lists does not need a special case for lists, but reuses the existing cases for sums, products and units. To make the idea more concrete let us have a look at how type structure is represented in LIGD.

In LIGD, the structure of a datatype b is represented by the following Rep constructor.

```
RType :: Rep c → EP b c → Rep b
```

The type c is the *structure representation* type of b, where b can be embedded in c. The embedding is witnessed by a pair of embedding and projecting functions translating between b and c values.

```
data EP b c = EP { from :: (b → c), to :: (c → b) }
```

In LIGD, constructors are represented by nested sum types and constructor arguments are represented by nested product types. The structure representation type for lists is Sum Unit (Prod a [a]), and the embedding and projection for lists are as follows:

```

fromList :: [a] → Sum Unit (Prod a [a])
fromList []      = Inl Unit
fromList (a : as) = Inr (Prod a as)
toList :: Sum Unit (Prod a [a]) → [a]
toList (Inl Unit) = []
toList (Inr (Prod a as)) = a : as

```

To extend the universe to lists, we use RType:

```

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

```

Generic equality is still missing a case to handle datatypes that are represented by RType. The definition of this case is given below. It takes two values, transforms them to their structure representations and recursively applies equality.

```
geq (RType ra ep) t1 t2 = geq ra (from ep t1) (from ep t2)
```

In summary, there are two ways to extend a universe to a type T. Non-generic extension requires type-specific, ad-hoc cases for T in type-indexed functions, and generic-extension requires a structure representation of T but no additional function cases. This is a distinguishing feature between type-indexed functions and generic functions. The latter include a case for RType, which allows them to exploit the structure of a datatype to apply generic uniform behaviour to values of that datatype; while the former do not have a case for RType, and rely exclusively on non-generic extension.

In LIGD, sums, products, and units are used to represent the structure of a datatype. Other choices are possible. For example, PolyLib includes the datatype Fix in its universe, to represent the recursive structure of datatypes. We refer to these representation choices as *generic views* [Holdermans et al., 2006]. Informally, a view consists of base (or view) types for the universe (for example Sum and Prod) and a convention to represent structure, for example, the fact that constructors are represented by nested sums. The choice of a view often has an impact on the expressiveness of a li-

brary, that is, which generic function definitions are supported and what are the set of datatypes on which generic extension is possible.

3. Design of the benchmark suite

Most previous work on datatype-generic programming focuses on either increasing the number of scenarios in which generic programming can be applied, or on obtaining the same number of scenarios using fewer or no programming language extensions. For example, Hinze’s work on “Polymorphic values possess polykinded types” [Hinze, 2002] shows how to define generic functions that work on types of arbitrary kinds, instead of on types of a particular kind, and “Generics for the Masses” [Hinze, 2006] shows how to do a lot of generic programming without using Haskell extensions. Both goals are achieved by either inventing a new generic programming approach altogether, or by extending an existing approach.

We have collected a number of typical generic programming scenarios from the literature. These are used as a guide to design our benchmark suite. The intuition is that the evaluation of a library should give an accurate idea of how well the library supports the generic programming scenarios. We list the scenarios below:

- Generic versions of Haskell type class functionality such as equality (Eq), comparison (Ord) and enumeration (Enum) [Jansson and Jeuring, 1998].
- Serialisation and deserialisation functions such as *read* and *show* in Haskell [Jansson and Jeuring, 2002].
- Traversals to query and modify information in datatypes [Lämmel and Peyton Jones, 2003].
- Functions like map, crush, and transpose, which manipulate elements of a parametrised datatype such as lists [Jansson and Jeuring, 1998, Norell and Jansson, 2004].
- Data conversion [Jansson and Jeuring, 2002, Atanassow and Jeuring, 2004].
- Test data generation [Koopman et al., 2003, Lämmel and Peyton Jones, 2005].

We have identified the features that are needed from a generic library to implement the scenarios above. These features are used as criteria to characterise generic programming from a user’s point of view, where a user is a programmer who *writes* generic programs. There are also users who only *use* generic programs (such as people that use *deriving* in Haskell), but the set of features needed by the latter kind of users is a subset of that needed by the former. Generic programming scenarios are not the only source of criteria, we also use the following sources:

- new features introduced to existing approaches such as Hinze [2002],
- Comparing approaches to generic programming in Haskell [Hinze et al., 2007],
- the Haskell generics wiki page [Haskell Generic Library list, 2008],
- our own ideas, based on several years experience with different approaches to generic programming.

We test whether the criteria are fulfilled with a benchmark suite. Each function in the suite tests whether or not an approach satisfies a particular criterion. For example, generic map cannot be implemented if the library does not support “abstraction over type constructors”. Hence, if a library cannot be used to implement a function, it means that it does not support the criterion that the function is testing. Each function in the benchmark suite is a simplified version of one of the above programming scenarios.

```

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
  RSalary :: Rep Salary
  RWTree :: Rep a → Rep w → Rep (WTree a w)

```

Figure 3. Definition of Rep. The two last constructors are not part of the LIGD library.

```

rCompany :: Rep Company
rDept    :: Rep Dept
rBinTree :: Rep a → Rep (BinTree a)
rWTree   :: Rep a → Rep w → Rep (WTree a w)
rGRose   :: (∀ a . Rep a → Rep (f a)) →
            Rep a → Rep (GRose f a)

```

Figure 4. Type signatures of some type representations.

Before introducing the functions used in the benchmark suite, we describe the datatypes on which they are used, and the related structure representation machinery.

3.1 Datatypes

The datatype construct in Haskell combines many aspects: type abstraction and application, recursion, records, local polymorphism, etc. In this section we introduce a number of datatypes, that cover many of these aspects. A generic programming library that can apply generic functions to one of these datatypes is said to support the aspects that the datatype requires in its definition.

Aspects that we test for are: parametrised types (type constructors, using type abstraction and application), simple and nested recursion, higher-kinded datatypes (with a parameter of kind $\star \rightarrow \star$) and constructor name information (to implement generic show).

Aspects that we do not test in this paper are higher-rank constructors (explicit forall in the datatype declaration), existential types, GADTs, and parsing related information, namely record label names, constructor fixity, and precedence. The first three aspects are not tested because they are hardly supported by any of the libraries that we evaluate. The last aspect, parsing-related information, can be incorporated using the same mechanisms as for providing constructor names, and therefore we do not add datatypes that test for this aspect.

Following each datatype definition we must also provide the machinery that allows universe extension for the particular library we are using. For LIGD each datatype T must map to a structure representation type T’ and back, with functions *fromT* and *toT*. The type representation for T is *rT*, it is written using *RType* like the list representation (*rList*) in Section 2. However, two of the datatypes presented below, namely Salary and WTree, are used in non-generic extension tests. For this reason the definition of Rep in Figure 3 includes the constructors *RSalary* and *RWTree*.

The company datatype. The Company datatype [Lämmel and Peyton Jones, 2003] represents the organisational structure of a company.

```

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

```

To define the representation of `Company` we must also define the representation of the supporting datatypes `Dept`, `DUnit`, etc.

```

rCompany = RType (rList rDept)
              (EP fromCompany toCompany)
rDept = ...

```

Because `Salary` is used with non-generic extension, the representation uses `RSalary` directly:

```
rSalary = RSalary
```

Binary trees. The recursive `BinTree` datatype abstracts over the type of its elements stored in the leaves.

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

Like lists, the representation depends on the representation of `a`:

```

rBinTree :: Rep a → Rep (BinTree a)
rBinTree ra = let r = rBinTree ra in
  RType (RSum ra (RProd r r)) (EP fromBinT toBinT)

```

Trees with weights. We adapt the type of binary trees such that we can assign a weight, whose type is abstracted, to a (sub)tree.

```

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

```

Some of the generic function tests treat weights differently from elements, even if their types are the same. The representation of `WTree` and the remaining datatypes are omitted because they follow the same pattern as `rBinTree` defined just above. However, Section 3.2.3 uses a different structure representation of `WTree` to define specialised behaviour for constructors.

Generalised rose trees. Rose trees are (non-empty) trees whose internal nodes have a list of children instead of just two.

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

We can generalise `Rose` by abstracting from the list datatype:

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

The interesting aspect that `GRose` tests is higher-kindedness: it takes a type constructor argument `f` of kind $\star \rightarrow \star$.

Perfect trees. The datatype `Perfect` is used to model perfect binary trees: binary trees that have exactly 2^n elements, where n is the depth of the binary tree.

```

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

```

The depth of a perfect binary tree is the Peano number represented by its constructors. The datatype `Perfect` is a so-called *nested datatype* [Bird and Meertens, 1998], because the type argument changes from `a` to `Fork a` in the recursion.

Nested generalised rose trees. The `NGRose` datatype is a variation on `GRose` that combines nesting with higher-kinded arguments: at every recursive call `f` is passed composed with itself:

```

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

```

Non-generic representations for `Salary` and `WTree`. Two of the datatypes introduced above are used in tests that check whether a library supports non-generic extension. Because non-generic extension is not supported by `LIGD`, we assume that `Rep` includes representation constructors for those datatypes in order to be able to describe the extension tests. The full definition of `Rep` and the signatures of some representations are shown in Figures 3 and 4. Note that post-hoc addition of constructors to the `Rep` datatype is a suboptimal idea that will break existing code. Concretely, the definition of `geq` in this paper is for the first four constructors (`RUnit`, `RSum`, `RProd`, `RType`) of `Rep`, thus any use of `geq` on `RSalary` or `RWTree` will fail. We return to this problem in Section 5 where we discuss support for ad-hoc definitions in `LIGD`.

3.2 Functions

Inspired by the generic programming scenarios given at the beginning of this section, we describe a number of generic functions for our benchmark suite.

It is not necessary to include all functions arising from the generic programming scenarios. If two functions use the same set of features from a generic programming library, it follows that if one of them can be implemented, the other can be implemented too. For example, the test case generator, generic read, and generic enumeration functions rely on library support for writing producer functions. So, it is enough to test that feature with one function, and hence we omit the last two functions from the benchmark suite.

3.2.1 Generic variants of type class functionality: Equality

Generic equality (in `LIGD`) takes a type representation argument `Rep a` and produces the equality function for `a`-values.

```
geq :: Rep a → a → a → Bool
```

Two values are equal if and only if they have the same constructor and the arguments of the constructors are pairwise equal. `LIGD` encodes constructors as nested sum types, so two constructors are the same only if they have the same sum-constructor (`Inl` or `Inr`). Constructor arguments are encoded as nested products, hence product equality requires the equality of corresponding components, see Fig. 1. `LIGD` ignores constructor names — only positions of constructors in the “constructor list” and positions of arguments in the “argument list” are taken into account.

The generic version of the `Ord` method, `compare`, would have type `Rep a → a → a → Ordering`. Like equality it takes two arguments and consumes them. Approaches that can implement equality can also implement comparison if constructor information is available (see the corresponding criterion in Section 4).

3.2.2 Serialisation and deserialisation: Show

The `show` function takes a value of a datatype as input and returns its representation as a string. It can be viewed as the implementation of **deriving** `Show` in Haskell. Its type is as follows:

```
gshow :: Rep a → a → String
```

The function `gshow` is used to test the ability of generic libraries to provide constructor names for arbitrary datatypes. For the sake of simplicity this function is not a full replacement of Haskell’s `show`:

- The generic show function treats lists in the same way as other algebraic datatypes. (Note that in the examples that follow we use \rightsquigarrow to indicate reductions of expressions.)

```
gshow [1, 2]  $\rightsquigarrow$  "(:) 1 ((:) 2 [])"
```

Note, however, that *gshow* is extended in one of the tests to print lists using Haskell notation. This is a separate test that is called *gshowExt*.

- It also treats strings just as lists of characters:

```
gshow "GH"  $\rightsquigarrow$  "(:) 'G' ((:) 'H' [])"
```

- Other features that are not supported are constructor fixity, precedence and record labels.

3.2.3 Querying and transformation traversals

A typical use of generic functions is to collect all occurrences of elements of a particular constant type in a datatype. For example, we might want to collect all *Salary* values that appear in a datatype:

```
selectSalary :: Rep a  $\rightarrow$  a  $\rightarrow$  [Salary]
```

We can instantiate this function to *Company*:

```
selectSalary rCompany :: Company  $\rightarrow$  [Salary]
```

Collecting values is an instance of a more general pattern: querying traversals. The function above can be implemented using (1) a general function (which happens to be generic) that performs the traversal of a datatype, and (2) a specific case that actually collects the *Salary* values. Such an implementation of *selectSalary* requires two features from a generic programming library:

- A generic function can have an ad-hoc (non-uniform) definition for some type. For example, *salaryCase* returns a singleton list of its argument if applied to a *Salary* value. Otherwise it returns the empty list.

```
salaryCase :: Rep a  $\rightarrow$  a  $\rightarrow$  [Salary]
salaryCase RSalary sal = [sal]
salaryCase rep      -  = []
```

The LIGD library does not support this feature, but we extended the *Rep* type in Fig. 3 to be able to show what it would look like.

- A generic function can take another generic function as argument. Consider for example (the LIGD version of) the *gmapQ* function from the first SYB paper,

```
gmapQ :: ( $\forall a$ . Rep a  $\rightarrow$  a  $\rightarrow$  r)  $\rightarrow$  Rep b  $\rightarrow$  b  $\rightarrow$  [r]
gmapQ f rT (K a1 ... an)  $\rightsquigarrow$  [f rT1 a1, ..., f rTn an]
```

This function takes three arguments: a generic function *f*, a type representation and a value of that type. If the value is a constructor *K* applied to a number of arguments, *gmapQ* returns a list of *f* applied to each of the arguments.

Using *salaryCase* as argument it gives:

```
gmapQ salaryCase (rList rSalary) (S 1.0 : [S 2.0])
 $\rightsquigarrow$  [salaryCase rSalary (S 1.0)
      , salaryCase (rList rSalary) [S 2.0]]
 $\rightsquigarrow$  [[S 1.0], []]
```

It is not a good idea to test for both features with one single test case in our suite: if a library does not support one of them the other will remain untested. For this reason we test these two features separately, using the functions *selectSalary* and *gmapQ*:

```
selectSalary :: Rep a  $\rightarrow$  a  $\rightarrow$  [Salary]
gmapQ :: ( $\forall a$ . Rep a  $\rightarrow$  a  $\rightarrow$  r)  $\rightarrow$  Rep a  $\rightarrow$  a  $\rightarrow$  [r]
```

Transformation traversals. An obvious variation on queries are transformation traversals. A typical example of such a traversal consists of transforming some nodes while performing a bottom-up traversal. Function *updateSalary* increases all occurrences of *Salary* by some factor in a value of an arbitrary datatype.

```
updateSalary :: Float  $\rightarrow$  Rep a  $\rightarrow$  a  $\rightarrow$  a
updateSalary 0.1 (rList rSalary) [S 1000.0, S 2000.0]
 $\rightsquigarrow$  [S 1100.0, S 2200.0]
```

Transformations on constructors. The *updateSalary* function traverses datatypes other than *Salary* generically, in other words the traversal is performed on the structure representation using the cases for products, sums and units. It follows that it is unnecessary to supply ad-hoc traversal cases for such datatypes.

The ad-hoc behaviour in *updateSalary* targets a particular datatype. Constructor cases [Clarke and Löh, 2003], a refinement of this idea, introduce ad-hoc behaviour that instead targets a particular constructor. Suppose we want to apply an optimisation rule $x + 0 \mapsto x$ to values of a datatype that consists of a large number of constructors. Ideally, we want a rewrite function that has an ad-hoc case for sums, and traverses other constructors generically.

The benchmark suite includes the function *rmWeights* to test ad-hoc behaviour for constructors. This function removes the weight constructors from a *WTree*:

```
rmWeights (RWTree RInt RInt)
  (WBin (WithWeight (WLeaf 42) 1)
        (WithWeight (WLeaf 88) 2))
 $\rightsquigarrow$  (WBin (WLeaf 42) (WLeaf 88))
```

The definition of the transformation handles the *WithWeight* constructor and lets the remaining constructors be handled by the generic machinery.

```
rmWeights :: Rep a  $\rightarrow$  a  $\rightarrow$  a
rmWeights r@(RWTree ra rw) t =
  case t of
    WithWeight t' w  $\rightarrow$  rmWeights r t'
    t'                 $\rightarrow$  ... handle generically ...
  ... rest of definition omitted ...
```

The second arm of the case traverses the structure representation of *t'* generically rather than matching *WBin* and *WLeaf* explicitly. The full code of the function is shown in Fig. 5.

The last line of the definition uses *rWTree* to traverse the structure representation of *t'*. Because it is essential that remaining *WithWeight* constructors in *t'* are removed, the definition of *rWTree* has to be altered for this function. The recursive occurrences of *WTree* have to be represented by *RWTree* rather than *rWTree* as is usually done in other structure representations. In this way traversals of the subtrees will again be handled by the ad-hoc case (see Fig. 5).

3.2.4 Abstraction over type constructors: crush and map

The function *crushRight* [Meertens, 1996] is a generic fold-like function. Typical instances are summing all integers in a list, or flattening a tree into a list of elements.

```
sumList :: [Int]  $\rightarrow$  Int
sumList [2, 3, 5, 7]  $\rightsquigarrow$  17
```

```

rmWeights :: Rep a → a → a
rmWeights RUnit Unit = Unit
rmWeights (RSum ra rb) (Inl x)
  = Inl (rmWeights ra x)
rmWeights (RSum ra rb) (Inr x)
  = Inr (rmWeights rb x)
rmWeights (RProd ra rb) (Prod a b)
  = Prod (rmWeights ra a) (rmWeights rb b)
rmWeights (RType ra ep) t
  = to ep (rmWeights ra (from ep t))
rmWeights (RWTree ra rw) t
  = case t of
    WithWeight t' w → rmWeights (RWTree ra rw) t'
    t'                → rmWeights (rWTree ra rw) t'

```

```

rWTree :: Rep a → Rep w → Rep (WTree a w)
rWTree ra rw = let r = RWTree ra rw in
  RType (RSum ra (RSum (RProd r r) (RProd r rw)))
    (EP fromWTree toWTree)

```

Figure 5. Generically remove weights from a WTree.

```

flattenBinTree :: BinTree a → [a]
flattenBinTree (Bin (Leaf 2) (Leaf 1)) ∼ [2,1]

```

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

```

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

```

The function *crushRight* traverses the *f a* structure accumulating a value of type *b*, which is updated by combining it with every *a*-value that is encountered during the traversal.

So far, generic functions use a type representation that encodes types of kind \star . Lists are not an exception: *rList r_a* represents fully applied list types. To define *crushRight* we switch to a type representation that encodes types of kind $\star \rightarrow \star$. This is why we use *Rep'* instead of *Rep* (and below *rList'* instead of *rList*). This is a common situation: to increase expressiveness of a generic library the type representation is adjusted. This is unfortunate because different type and structure representations are usually incompatible.

Functions *sumList* and *flattenBinTree* are obtained by instantiating *crushRight* on lists or trees with the appropriate arguments:

```

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

```

How are generic queries different from *crushRight*? We could for example define a function *selectInt* to flatten a *BinTree Int* into a list of *Int* values. There are two differences. First, if the *BinTree* elements were booleans instead of integers, we would need a different querying function: *selectBool*. With *flattenBinTree* we do not have this problem because it is parametrically polymorphic in the elements of the datatype.

The second difference is about the type signature of the querying function. Suppose now that we want to flatten *WTree Int Int* into a list of weights.

```

flattenWTWeights :: WTree a w → [w]
flattenWTWeights (WBin (WithWeight (WLeaf 1) 2)
  (WithWeight (WLeaf 3) 4))
  ∼ [2,4]

```

This is just an instance of *crushRight*:

```

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

```

where *rWTree'* represents *WTree a* for any *a*. In contrast, for *selectInt*, there is no difference between *Int*-weights and *Int*-elements in the tree. So it gives the following incorrect result:

```

selectInt (WBin (WithWeight (WLeaf 1) 2)
  (WithWeight (WLeaf 3) 4))
  ∼ [1,2,3,4]

```

Alternatively, we could use ad-hoc cases to solve this problem with queries. For example, we could wrap the *w*-elements in a **newtype**-type and give an ad-hoc case for it. We could also give an ad-hoc case for *WTree* where the second argument of *WithWeight* is extracted. This highlights the difference with *crushRight*: *crushRight* does not need ad-hoc cases, while traversal queries do.

To sum up, the difference with queries is that *crushRight* views the datatype as an application of a type constructor *f* to an element type *a*, and processes only *a*-values. In contrast, traversal queries do not make such element discrimination based on the type structure of the datatype.

Map. Generic map is to transformation traversals what *crushRight* is to query traversals. The *gmap* function takes a function and a structure of elements, and applies the function argument to all elements in that structure. The type signature of *gmap* uses the same representation as *crushRight*:

```

gmap :: Rep' f → (a → b) → f a → f b

```

The best known instance is the map function on lists, but we also have instances like

```

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

```

In general, *gmap* can be viewed as the implementation of **deriving** for the Functor type class in Haskell.

Another function, generic transpose, is representative not only of abstraction over type constructors but also of data conversion functions. We discuss it next.

3.3 Data conversion: transpose

A data conversion function has type $T \rightarrow T'$: it converts *T* values into *T'* values. Jansson and Jeuring [2002] and Atanassow and Jeuring [2004] discuss generic approaches to build conversion functions. It turns out that there is no need to include the conversion functions from these sources, because the conversion functions are built out of simpler generic functions which are already accounted for in our scenarios. The former paper uses a combination of serialisation, deserialisation, and abstraction over type constructors. The latter paper composes serialisation and deserialisation functions that exploit isomorphisms in the intermediate structures.

A more sophisticated version of data conversion is the generic transpose function, described in Norell and Jansson [2004]. Although this function can be implemented using a combination of serialisation, deserialisation, and abstraction over type constructors, it is an interesting challenge for library approaches because it abstracts over two type constructors.

The generic transpose function is a generalisation of the function *transpose* that is defined in the Haskell standard library. Since this function abstracts over two type constructors it takes two type representations:

```

gtranspose :: Rep' f → Rep' g → f (g a) → g (f a)

```

Instantiating both *f* and *g* to the list type we obtain *transpose* again.

```

transpose = gtranspose rList' rList'

```

$transpose \ [[1, 2, 3], [4, 5, 6]] \rightsquigarrow \ [[1, 4], [2, 5], [3, 6]]$

It can also be instantiated to other datatypes:

$gtranspose \ rList' \ rBinTree' \ [Leaf \ 1, Leaf \ 2, Leaf \ 3]$
 $\rightsquigarrow \ Leaf \ [1, 2, 3]$

3.3.1 Test data generation: Fulltree

Testing is used to gain confidence in a program. QuickCheck [Claessen and Hughes, 2000] is a popular tool that supports automatic testing of properties of programs. A user-defined datatype can be used in an automated test, provided it is an instance of the *Arbitrary* class.

To implement the test data generation scenario, a library should be able to *produce* values. The test data generation scenario is represented by the *gfulltree* function.

The *gfulltree* function takes a representation of a container datatype as input and returns all possible values of the represented datatype up to a given depth. Note that the depth argument only makes sense with a *recursive* datatype. At each recursion every constructor is used again to generate a value. Here is the type signature of *gfulltree* and some examples of its usage:

$gfulltree :: Rep \ a \rightarrow Int \rightarrow [a]$
 $gfulltree \ (rList \ rInt) \ 4 \rightsquigarrow \ [[], [0], [0, 0], [0, 0, 0]]$
 $gfulltree \ (rBinTree \ rInt) \ 4 \rightsquigarrow$
 $\ [Leaf \ 0$
 $\ , Bin \ (Leaf \ 0) \quad (Leaf \ 0)$
 $\ , Bin \ (Leaf \ 0) \quad (Bin \ (Leaf \ 0) \ (Leaf \ 0))$
 $\ , Bin \ (Bin \ (Leaf \ 0) \ (Leaf \ 0)) \ (Leaf \ 0)$
 $\ , Bin \ (Bin \ (Leaf \ 0) \ (Leaf \ 0)) \ (Bin \ (Leaf \ 0) \ (Leaf \ 0))]$

Function *gfulltree* can also be used to generate large values that are used in performance tests.

3.3.2 More general representations of type constructors

The generic functions above use type representations over types of kind \star :

$geq :: Rep \ a \rightarrow a \rightarrow a \rightarrow Bool$

and over types of kind $\star \rightarrow \star$

$gmap :: Rep' \ f \rightarrow (a \rightarrow b) \rightarrow f \ a \rightarrow f \ b$

What if we want to apply *gmap* to transform *both* the payload and weights of a *WTree*? Should we define Rep'' to represent types of kind $\star \rightarrow \star \rightarrow \star$? It is not good practice to define a new representation every time we come across a kind that was not previously representable.

Below we discuss briefly how some libraries generalise over types of arbitrary kinds. While those solutions are more general (and maybe more useful) than having one representation per kind, the full discussion of these solutions would complicate the presentation. We continue to use Rep and Rep' in this paper to keep the presentation of *gmap* and *crushRight* simple.

Representations and arities. The work of Hinze [2002] introduces a technique to allow generic functions to be applied to type constructors of arbitrary kinds. The technique consists in generalising the type signature of a generic function so that it becomes generic in more than one type variable. Let us examine this technique in a library setting, here we use the representations introduced by the GM approach [Hinze, 2006].

The idea is that GM type representations, in addition to abstract over a type a :

data $Rep2 \ sig \ a \ b \ where$

$RUnit2 :: Rep2 \ sig \ Unit \ Unit$
 $RSum2 :: Rep2 \ sig \ a \ b \rightarrow Rep2 \ sig \ c \ d$
 $\rightarrow Rep2 \ sig \ (Sum \ a \ c) \ (Sum \ b \ d)$
 $RProd2 :: Rep2 \ sig \ a \ b \rightarrow Rep2 \ sig \ c \ d$
 $\rightarrow Rep2 \ sig \ (Prod \ a \ c) \ (Prod \ b \ d)$
 $RType2 :: Rep2 \ sig \ a \ d \rightarrow EP \ b \ a \rightarrow EP \ e \ d$
 $\rightarrow Rep2 \ sig \ b \ e$
 $RVar2 :: sig \ a \ b \rightarrow Rep2 \ sig \ a \ b$
 $rWTree2 :: Rep2 \ sig \ a \ b \rightarrow Rep2 \ sig \ c \ d$
 $\rightarrow Rep2 \ sig \ (WTree \ a \ c) \ (WTree \ b \ d)$

Figure 6. Type representation for generic functions of arity two

data $Rep \ a = \dots$

they also abstract over the signature of a generic function:

data $Rep1 \ (sig :: \star \rightarrow \star) \ (a :: \star) = \dots$

Here sig could be used, for example, with the signature of equality (**newtype** $Eq \ a = Eq \ (a \rightarrow a \rightarrow Bool)$) or generic show (**newtype** $Show \ a = Show \ (a \rightarrow String)$). Generic map, however, cannot be defined using $Rep1$ because it needs to change the type of its argument. So the type signature needs to abstract over *two* types rather than one: **newtype** $Map \ a \ b = Map \ (a \rightarrow b)$. This prompts for a new representation:

data $Rep2 \ (sig :: \star \rightarrow \star \rightarrow \star) \ (a :: \star) \ (b :: \star) = \dots$

and a new definition:

$gmap :: Rep2 \ Map \ a \ b \rightarrow a \rightarrow b$
 $gmap = \dots$

Now, what happens if we want to define generic zip? The signature abstracts over three variables: **newtype** $Zip \ a \ b \ c = Zip \ (a \rightarrow b \rightarrow c)$, which needs a new representation:

data $Rep3 \ (sig :: \star \rightarrow \star \rightarrow \star \rightarrow \star) \ (a :: \star) \ (b :: \star) \ (c :: \star) = \dots$

The number of variables over which a generic function signature abstracts is called the arity of the function. Generic equality has arity one, while generic map has arity two. Unfortunately, arities are a source of redundancy: every arity requires a new representation. It is possible to define generic functions of arity one and two using $Rep3$, but that solution is rather inelegant.

Let us examine the increased generality of these new representations. The definition of $Rep2$, which is given in Figure 6, reveals strong similarities to Rep : there are representations for units, sums and products. The novelty lies in the *Var2* constructor. This constructor stores a function having type $sig \ a \ b$. This is the key that enables abstraction over type constructors of arbitrary kind. To see why, let us try to specialize *gmap* to *WTree*, which has kind $\star \rightarrow \star \rightarrow \star$.

$mapWTree :: (a \rightarrow b) \rightarrow (w \rightarrow t) \rightarrow WTree \ a \ w \rightarrow WTree \ b \ t$
 $mapWTree \ f \ g = let \ mkR \ x = RVar2 \ (Map \ x)$
 $\quad in \ gmap \ (rWTree2 \ (mkR \ f) \ (mkR \ g))$

The functional arguments which perform the transformation on the tree data are stored inside the representation, so that *gmap* can apply them when handling the *RVar2* case.

New representations for differently-kinded types are not required, because the instantiation looks very similar to what we saw above in *mapWTree*. Hence it is more general than kind-specific representations like Rep' .

Types	Expressiveness (continued)	Usability
<ul style="list-style-type: none"> • Universe Size • Subuniverses 	<ul style="list-style-type: none"> • Ad-hoc definitions for datatypes • Ad-hoc definitions for constructors • Extensibility • Multiple arguments • Multiple type representation arguments • Constructor names • Consumers, transformers, and producers 	<ul style="list-style-type: none"> • Performance • Portability • Overhead of library use • Practical aspects • Ease of use and learning
Expressiveness <ul style="list-style-type: none"> • First-class generic functions • Abstraction over type constructors • Separate compilation 		

Figure 7. Criteria overview

4. Criteria

This section describes the criteria used to evaluate the generic programming libraries. We have grouped criteria around three aspects:

- **Types:** To which datatypes generic functions can be applied, and the signatures of generic functions.
- **Expressiveness:** The kind of generic programs that can be written.
- **Usability:** How convenient a library is to use, efficiency, quality of library distribution, portability.

Figure 7 summarises the criteria and the organisation. In this section, we describe the evaluation criteria and, when possible, we illustrate them with code.

Types

- **Universe Size.** What are the types that a generic function can be used on? The more types a generic function can be used on, the bigger the universe size for that library. Different approaches implement generic universe extension in different ways, hence the sizes of their universes can differ.

Ideally, we would like to know whether a given library supports generic extension to nested and higher-kinded datatypes. But the claim that universe extension applies to, for example, nested datatypes is impractical to verify. It would require a rigorous proof that covers all nested datatypes.

Instead, we take a less ambitious alternative to estimate the size of the universe. We test whether a given approach supports extension to a number of datatypes, each of which demonstrates a particular datatype property. We test universe extension on lists, `BinTree` and `WTree` (regular datatypes), `GRose` (higher-kinded), `Perfect` (nested), `NGRose` (higher-kinded and nested), and `Company` (mutually recursive).

- **Subuniverses.** Is it possible to restrict the use of a generic function to a particular set of datatypes, or to a subset of all datatypes? Will the compiler flag uses on datatypes outside that subuniverse as a type error?

Expressiveness

- **First-class generic functions.** Can a generic function take a generic function as an argument? This is tested by `gmapQ`, the function that applies a generic function argument to all constructor arguments:

```
gmapQ (rList RInt) gshow (1 : [2])
  ~> [gshow RInt 1, gshow (rList RInt) [2]]
  ~> ["1", "(:" 2 []"]
```

Here `gshow` is applied to the two fields of the list constructor `(:)`, each having a different type, hence `gshow` must be instantiated to different types.

- **Abstraction over type constructors.** The equality function can usually be defined in an approach to generic programming, but a generalisation of the map function on lists to arbitrary container types cannot be defined in all proposals. This criterion is tested by the `gmap` and `crushRight` generic functions.
- **Separate compilation.** Is generic universe extension modular? That is, can a datatype defined in one module be used with a generic function and type representation defined in other modules without the need to modify or recompile them? This criterion is tested by applying generic equality to `BinTree`, which is defined in a different module than equality and the library itself.

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

- **Ad-hoc definitions for datatypes.** Can a generic function contain specific behaviour for a particular datatype, and let the remaining datatypes be handled generically? In this situation, ad-hoc, datatype-specific definitions are used instead of uniformly generic behaviour. This is tested by the `selectSalary` function, which consists of cases that perform a traversal over a datatype, accumulating the values collected by the `Salary` ad-hoc case (traversal code omitted for brevity):

```
selectSalary :: Rep a -> a -> [Salary]
selectSalary RSalary (S x) = [S x]
...
```

- **Ad-hoc definitions for constructors.** Can we give an ad-hoc definition for a particular constructor, and let the remaining constructors be handled generically? This is tested by the `rmWeights` function, which should have an explicit case to remove `WithWeight` constructors and the remaining constructors should be handled generically.
- **Extensibility.** Can the programmer non-generically extend the universe of a generic function in a different module? Because the extension meant here is non-generic, this criterion makes sense only if ad-hoc cases are possible. This criterion is tested by extending `gshow` with an ad-hoc case that prints lists using Haskell notation:

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

- **Multiple arguments.** Consumer functions such as `gshow` and `selectSalary` have one argument that is generic. Can the approach define a function that consumes more than one generic argument, such as the generic equality function?
- **Multiple type representation arguments.** Can a function be generic in more than one type? That is, can a generic function,

such as the generic transpose function, receive two or more type representations? The evaluation of this criterion is work in progress, so we do not include it in this paper.

- **Constructor names.** Can the approach provide the names of the constructors to which a generic function is applied? This is tested by the *gshow* generic function.
- **Consumers, transformers, and producers.** Is the approach capable of defining generic functions that are:
 - consumers ($a \rightarrow T$): *gshow* and *selectSalary*
 - transformers ($a \rightarrow a$ or $a \rightarrow b$): *updateSalary* and *gmap*
 - producers ($T \rightarrow a$): *gfulltree*

Usability

- **Performance.** Some proposals use many higher-order functions to implement generic functions, others use conversions between datatypes and structure types. We have compared running times for some of the test functions for the different libraries.
- **Portability.** Few proposals use only the Haskell98 standard for implementing generic functions, most use (sometimes unimplemented) extensions to Haskell98, such as recursive type synonyms, multi-parameter type classes with functional dependencies, GADTs, etc. A proposal that uses few or no extensions is easier to port across different Haskell compilers.
- **Overhead of library use.** How much additional programming effort is required from the programmer when using a generic programming library? We are interested in (1) support for automatic generation of structure representations, (2) 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.** Is there an implementation? Is it maintained? Is it documented?
- **Ease of learning and use.** Some generic programming libraries use implementation mechanisms that make their use or learning more difficult.

4.1 Coverage of testable criteria

Criteria can be divided into testable and non-testable groups. Testable criteria are the ones that can be tested by means of a generic function in the benchmark suite. Figure 8 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 testing functions introduced in Section 3. The criteria that a generic function tests are marked with ●.

The second group of columns stand for datatypes that test generic universe extension. These tests check whether *geq* can be instantiated and applied to values of those types.

Some testing functions unavoidably require support of two criteria from a library. For example, the generic extension test on the *GRose* datatype requires separate compilation and higher-kinded datatypes. This brings up the problem that lack of support for the first criterion will cause failure of the test, which, according to our procedure (described in Section 3), means failure for the second criterion too. As a result, despite the fact that the second criterion remains untested, the criterion will be assumed non-supported by the library. This test would fail on *Spine* because it does not support separate compilation, but from the failure of the test it can erroneously be concluded that higher-kinded datatypes are not supported by *Spine*.

For this reason we have tried to avoid requiring more than one criterion to implement a testing function, but this is not always possible. In such a situation we cheat a little: we ignore the issue of separate compilation and test it separately. This is shown in

Figure 8. The criteria that are normally needed but are ignored for the particular test (because of the more than one criterion per test issue) are marked with ○.

4.2 Design choices

The criteria that we have seen so far are interesting from a user's point of view. They inform the user on what generic programs can and cannot be written using the libraries. However, it is also illustrative to see the design choices that have been taken by the designers of these libraries, because a particular design choice may improve or hinder the support of an expressiveness criterion. For example, the use of type classes is essential to libraries that support extensibility, but they can also make the use of generic functions as first-class values more difficult.

In this paper, we look at two design choices:

- **Implementation mechanisms.** 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).
- **Views.** What are the views that the generic library supports? Examples of views are the sum of products view, the fixed point view, and the spine view. A library typically includes a type representation per view.

A third possible design choice is whether generic functions are instantiated by compile time specialisation or by interpretation of type representations at runtime. Here we do not include this design choice, because all evaluated libraries use interpretation. Approaches that encode type and structure representations as datatypes are clearly doing interpretation of the representation values. Type class based approaches also perform interpretation: dictionary values are used at runtime. Of course, some specialisation may take place if the compiler performs inlining in the generic program.

Why is this design decision important? If an approach would implement instantiation by compile-time specialisation, that approach would most likely not support higher-order generic functions. This is because higher-orderness requires specialising the generic function argument at *runtime*, as opposed to compiled time. Yet, it is interesting to see that some type class based libraries, namely *EMGM*, have difficulties supporting higher order functions.

5. Evaluation summary

We have tried to implement the benchmark in each of the generic programming libraries. This section gives a summary of the results. Figure 9 presents the results in a table. The criteria that a generic programming library supports are marked with ●. The ones that are not supported are marked with ○. If a criterion is partially supported, or if it requires unusual programming effort, it is marked with ◐ (in the text we say it scores *sufficient*). A more detailed evaluation and a discussion on the design choices can be found in Section 5. That section also discusses the design choices behind each of the evaluated libraries.

Universe Size. The *PolyLib* library is limited to regular datatypes (with one parameter). In *RepLib*, the datatypes with higher-kinded arguments (*GRose* and *NGRose*) are not supported. Approaches such as *SYB*, *SYB3*, *Uniplate*, and *EMGM*, which are based on type classes, have trouble supporting *NGRose*; the three first do not support it at all, while *EMGM* supports it but loses some functionality. Furthermore, the *SYB3* library does not support *Perfect* and it has an additional complication: *BinTree* is supported only if the instance is manually written, but not with the generated instance; we return to this problem when evaluating the generation

	<i>geq</i>	<i>selectSalary</i>	<i>gmapQ</i>	<i>updateSalary</i>	<i>rmWeights</i>	<i>crush</i>	<i>gmap</i>	<i>gshow</i>	<i>gshowExt</i>	<i>gtranspose</i>	<i>gfulltree</i>	BinTree	GRose	Perfect	NGRose	Company	
Universe Size	●		●		●	●	●				●	●					
Regular datatypes	●		●		●	●	●				●	●					
Higher-kinded datatypes													●		●		
Nested datatypes														●	●		
Nested & higher-kinded															●		
Mutually recursive		●		●				●	●								●
First-class generic functions			●														
Abstraction over type constructors						●	●										
Separate compilation	●					○	○	○		○		○	○	○	○	○	○
Ad-hoc definitions for datatypes		●		●	○				●								
Ad-hoc definitions for constructors					●												
Extensibility									●								
Multiple arguments	●											●	●	●	●	●	●
Multiple type representation arguments											●						
Constructor names								●	●								
Consumers	●	●	●			●		●	●			●	●	●	●	●	●
Transformers				●	●		●				●						
Producers											●						

● The criterion is tested by the example: the criterion is needed to implement test.
○ The criterion is normally needed by test, but it is circumvented to test other criteria.

Figure 8. Functions and datatypes set out against criteria.

of representations. LIGD and Spine have the advantage of a large universe size: they support all datatypes in this test. Smash also supports all datatypes, but there are datatypes that require unusual effort to allow generic extension: Perfect, NGRose, and Company.

Subuniverses. The PolyLib, EMGM, RepLib, and Smash libraries support subuniverses.

First-class generic functions. In LIGD, SYB, and Spine a generic function is a polymorphic Haskell function, so it is a first-class value. The PolyLib and Uniplate libraries do not support this criterion. The SYB3, EMGM, and RepLib libraries support this criterion, but in the second there is additional complexity so it only scores sufficient. Smash requires a new structure representation for this test so it only scores sufficient.

Abstraction over type constructors. The LIGD, PolyLib, EMGM, RepLib, Spine, and Smash libraries support abstraction over type constructors. However, PolyLib, and Spine only support abstraction over type constructors of kind $\star \rightarrow \star$, so the support of these approaches for this criterion is only sufficient. The SYB3, and Uniplate libraries do not support this criterion. Surprisingly, recent work [Kiselyov, 2008] shows that SYB supports the definition of functions such as *gmap* and *crushRight*.

Separate compilation. LIGD, PolyLib, SYB, SYB3, EMGM, RepLib, Smash, and Uniplate support separate compilation. The only evaluated approach that does not support this criterion is Spine: generic universe extension requires recompilation.

Ad-hoc definitions for datatypes. This criterion is supported by SYB, SYB3, EMGM, RepLib, Smash, and Uniplate, but not by LIGD and Spine. PolyLib supports ad-hoc cases for regular datatypes, which excludes the Company datatype, so it fails the test.

Ad-hoc definitions for constructors. Ad-hoc definitions for constructors are supported by LIGD, SYB, SYB3, Spine, EMGM,

RepLib, Smash, and Uniplate. However, in LIGD the structure representation has to be adapted for this criterion to work, and in PolyP *rmWeights* is not flexible enough to be applied to other types than WTree.

Extensibility. This criterion is supported by SYB3, EMGM, RepLib, and Smash. It is partially supported by PolyLib, because it works only for regular datatypes, and hence it fails for this test.

Multiple arguments. Multiple argument functions are supported by almost all approaches, however, in some approaches, such as SYB and SYB3, the definitions can be rather complex, and therefore they score sufficient. In Smash the definition is not complex, but it requires a separate structure representation. The only library that fails to support multiple arguments is Uniplate.

Constructor names. Constructor names are supported by all evaluated approaches except Uniplate.

Consumers, transformers, and producers. Almost all libraries support definitions of functions in the three categories. However, there are libraries that use different structure representations for consumers and producers such as SYB, SYB3, Spine, and Smash. Smash in addition uses a different structure representation for transformations. Uniplate does not support producer functions.

Performance. We have used some of the test functions for a performance benchmark comparing running times for larger inputs. The results are very sensitive to small code differences and compiler optimisations so firm conclusions are difficult to draw, but the best overall performance score is shared between EMGM, Smash, and Uniplate.

Portability. The three most portable approaches are LIGD, EMGM, and Uniplate. The first approach relies on existential types and the other two on multi-parameter type classes, both extensions are very likely to be included in the next Haskell standard. Furthermore,

	LIGD	PolyLib	SYB	SYB3	Spine	EMGM	RepLib	Smash	Uniplate
Universe Size	●	●	●	●	●	●	●	●	●
Regular datatypes	●	●	●	●	●	●	○	●	●
Higher-kinded datatypes	●	○	●	○	●	●	●	●	●
Nested datatypes	●	○	●	○	●	●	●	●	●
Nested & higher-kinded	●	○	○	○	●	●	○	●	○
Mutually recursive	●	○	●	●	●	●	●	●	●
Subuniverses	○	●	○	○	○	●	●	●	○
First-class generic functions	●	○	●	●	●	●	●	●	○
Abstraction over type constructors	●	●	●	○	●	●	●	●	○
Separate compilation	●	●	●	●	○	●	●	●	●
Ad-hoc definitions for datatypes	○	●	●	●	○	●	●	●	●
Ad-hoc definitions for constructors	●	●	●	●	●	●	●	●	●
Extensibility	○	●	○	●	○	●	●	●	○
Multiple arguments	●	●	●	●	●	●	●	●	○
Constructor names	●	●	●	●	●	●	●	●	○
Consumers	●	●	●	●	●	●	●	●	●
Transformers	●	●	●	●	●	●	●	●	●
Producers	●	●	●	●	●	●	●	●	○
Performance	●	●	○	○	●	●	●	●	●
Portability	●	○	○	○	○	●	○	○	●
Overhead of library use									
Automatic generation of representations	○	○	●	●	○	○	●	○	●
Number of structure representations	4	1	2	2	3	4	4	8	1
Work to instantiate a generic function	●	●	●	●	●	●	●	●	●
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

Figure 9. Evaluation of generic programming approaches

multi-parameter type classes in EMGM are used in a non-essential way: the functionality of EMGM would only be slightly affected in their absence. The other approaches rely on non-portable Haskell extensions.

Overhead of library use. The SYB, SYB3, RepLib, and Uniplate libraries are equipped with automatic generation of representations. However, automatic generation in RepLib fails for type synonyms. In SYB3, the generated Data instance for BinTree causes non-termination when used with generic equality.

The number of structure representations is high for libraries such as LIGD, EMGM, and RepLib. The reason is that type constructor abstraction in these approaches requires one representation per generic function arity. The number of representations in Smash is even higher due to the amount of relatively specialised representations. More information can be found in Section 5. The SYB, SYB3, and Spine approaches have one representation for consumers and another for producers. In addition, Spine has a representation to abstract over type constructors. PolyLib and Uniplate have only one representation.

The instantiation of a generic function is easier (for the programmer) in libraries that support implicit type representations, such as PolyLib, SYB, SYB3, Uniplate, Smash, EMGM, and RepLib. However, the last two libraries require additional effort to enable instantiation. Therefore PolyLib, SYB, SYB3, Smash, and Uniplate are the libraries that require the least effort to instantiate a generic function.

The work required to define a generic function is higher, in the sense that more implementation machinery is required, in LIGD, SYB3, and RepLib.

Practical aspects. The SYB, RepLib, and Uniplate libraries have well-maintained and documented distributions. PolyLib has an official distribution, but it is not maintained anymore. The SYB3 library has two distributions: one does not compile under some versions of GHC (6.6, 6.8.1, 6.8.2) and the other does not have a number of useful combinators. Smash has an online distribution, but its interface is not as structured as, for example, SYB3. The remaining approaches, LIGD, Spine, and EMGM, do not have a well-maintained distribution.

Ease of learning and use. It is hard to determine how easy it is to learn how to use a library. We approximate this criterion by looking at the mechanisms used in the implementation of the libraries. We consider an approach easier if its implementation mechanisms are relatively simple such as for PolyLib and Uniplate (type classes), and Spine (GADTs). An approach is relatively difficult if it uses sophisticated implementation mechanisms, for example rank-2 typed combinators and abstraction over type classes as in SYB3. Intermediate approaches use advanced mechanisms only occasionally. One such approach is EMGM, which uses arity-based representations. More information can be found in Section 5.

6. Evaluation

A generic programming library provides an interface to achieve generic programming behaviour and uses certain mechanisms to implement it. These interfaces and mechanisms correspond to the concepts that we introduced in Section 2. In particular we can usually identify mechanisms corresponding to type representations, structure representations and functions that act on them. So two libraries may implement generic behaviour in very different ways,

by providing different ways to encode structure representations, for example.

Before proceeding with the detailed evaluation we introduce below each of the compared libraries and relate them against the concepts introduced in Section 2. We focus in particular on how they implement case analysis on types and structure representation of datatypes.

6.1 Lightweight implementation of Generics and Dynamics

The Lightweight implementation of Generics and Dynamics (LIGD) library was introduced by Cheney and Hinze [2002]. The presentation in the current paper largely follows the original presentation of LIGD. The difference is that the original Rep is not a GADT but a normal datatype. This datatype encodes the GADT by including conversion functions in the datatype constructors and by the use of existential types.

The generic view that LIGD uses is the sum of products view. The original LIGD paper does not include a view to abstract over type constructors, but it is well known how to do so: Hinze and Löh [2007] present a variant of LIGD called dictionary-passing style that abstracts over type constructors.

In this library case analysis on types is performed by means of pattern matching. The type structure of datatypes is represented by the *RType* constructor.

6.2 PolyLib

The pre-processor-based language extension PolyP [Jansson and Jeurig, 1997, 1998] was later packaged up as a more lightweight library [Norell and Jansson, 2004] and this library is what we compare in this paper. The library is limited to regular datatypes (with one parameter) so the supported universe is relatively small. But the smaller universe makes it possible to express a wider range of generic functions — the library contains definitions of folds and unfolds, traversals and even functions generic in two type parameters such as $transpose :: \dots \Rightarrow d (e a) \rightarrow e (d a)$.

The limited universe means that PolyLib is not suitable as a general generic library — it is included here as a “classic reference” and because of its expressiveness.

PolyLib uses a combination of the fixed-point view and the sum of products view. Each TIF is defined as a type (constructor) class with one instance for each universe building block (unit, sum, product, composition, function, const, parameter, and recursion).

6.3 Scrap your boilerplate

In the Scrap your boilerplate (SYB) library [Lämmel and Peyton Jones, 2003, 2004] generic functions are not programmed by pattern matching on the structural representation of a value, but rather by means of combinators. There are combinators for doing case analysis on types and for inspecting the structure of values.

Case analysis combinators exist in several variants: query combinators, transformation combinators, and monadic transformation combinators amongst others. Let us consider the combinators for queries: *mkQ* and *extQ*.

$$\begin{aligned} mkQ &:: (\text{Typeable } a, \text{Typeable } b) \\ &\Rightarrow r \rightarrow (b \rightarrow r) \rightarrow a \rightarrow r \\ extQ &:: (\text{Typeable } a, \text{Typeable } b) \\ &\Rightarrow (a \rightarrow r) \rightarrow (b \rightarrow r) \rightarrow (a \rightarrow r) \end{aligned}$$

The *mkQ* combinator takes an ad-hoc case specific to *b* values and creates a polymorphic function that can be applied to any value *a* that is an instance of *Typeable*. If it is applied to a *b* value —that is, if at runtime it is determined that *a* and *b* are the same type— the function with type $b \rightarrow r$ is applied to it, otherwise the argument of type *r* is returned. The *extQ* combinator extends a polymorphic query built with *mkQ* with yet another ad-hoc case.

These combinators are implemented by means of type-safe casting, which ultimately relies on the function *unsafeCoerce*, an unsafe Haskell extension that converts a value from one type to any other type. They also rely on the *Typeable* type class, which provides runtime representations of types, so that the equality of two types can be established at runtime.

The structure of datatypes is represented by the higher-order combinators *gfoldl* and *gunfold*. These are used to write consumer and producer functions respectively. Datatypes whose structure is represented are instances of the *Data* type class.

$$\begin{aligned} \text{class Typeable } a \Rightarrow \text{Data } a \text{ where} \\ gfoldl &:: \text{Data } a \Rightarrow (\forall a b. \text{Data } b \Rightarrow w (a \rightarrow b) \\ &\quad \rightarrow a \rightarrow w b) \\ &\quad \rightarrow (\forall a. a \rightarrow w a) \\ &\quad \rightarrow a \rightarrow w a \end{aligned}$$

The instance for lists below makes *gfoldl* clearer.

$$\begin{aligned} \text{instance Data } a \Rightarrow \text{Data } [a] \text{ where} \\ gfoldl \ k \ z \ [] &= z \ [] \\ gfoldl \ k \ z \ (x : xs) &= (z \ (\cdot) \ 'k' \ x) \ 'k' \ xs \end{aligned}$$

The *gfoldl* function applies *z* to the constructor and applies the result to the arguments of the constructor using *k*. In essence, *gfoldl* exposes the constructor and the arguments to the *k* and *z* functions. Further explanations can be found in the original SYB paper and in the description of the Spine approach below.

The SYB library provides two structure representations of data, one through *gfoldl* for consumer functions, and another through *gunfold* for producer functions. Because of their types these functions need support for rank-2 polymorphism.

As an example, this is how *selectSalary* is implemented in SYB:

$$\begin{aligned} selectSalary &:: \text{Data } a \Rightarrow a \rightarrow [\text{Salary}] \\ selectSalary \ x &= ([\ 'mkQ' \ salaryCase \] \ x : \\ &\quad \text{concat } (gmapQ \ selectSalary \ x)) \\ \text{where } salaryCase \ (sal :: \text{Salary}) &= [sal] \end{aligned}$$

The *mkQ* expression performs case analysis on types, *x* is added to the result if it is a *Salary* but not otherwise. Then *gmapQ* applies *selectSalary* to the children of *x*.

$$gmapQ :: \text{Data } a \Rightarrow (\forall a. \text{Data } a \Rightarrow a \rightarrow u) \rightarrow a \rightarrow [u]$$

6.4 Scrap your boilerplate, extensible with typeclasses

A serious drawback of Scrap your boilerplate is that it is not extensible: once a function (such as *selectSalary*) is defined, it cannot be extended with an ad-hoc case. This problem is solved by the extensible variant of Scrap your boilerplate (SYB3) [Lämmel and Peyton Jones, 2005]. The extended approach is still combinator based, indeed generic functions are written using combinators such as *gfoldl* and *gmapQ*.

$$\begin{aligned} \text{class (Typeable } a, \text{Sat } (ctx \ a)) \Rightarrow \text{Data } ctx \ a \text{ where} \\ gfoldl &:: \text{Proxy } ctx \\ &\quad \rightarrow (\forall b \ c. \text{Data } ctx \ b \Rightarrow w (b \rightarrow c) \rightarrow b \rightarrow w \ c) \\ &\quad \rightarrow (\forall g. g \rightarrow w \ g) \\ &\quad \rightarrow a \rightarrow w \ a \\ gmapQ &:: \text{Proxy } ctx \\ &\quad \rightarrow (\forall a. \text{Data } ctx \ a \Rightarrow a \rightarrow r) \rightarrow a \rightarrow [r] \end{aligned}$$

There is an additional type argument *ctx* to *Data*. This is the essential ingredient that allows extension. Both functions also take an additional *Proxy* argument, which is merely a way to inform the type checker what *ctx* type is used and the actual argument value is not important. We shall explain more about *ctx* shortly.

However, case analysis on types is no longer based on combinators such as *mkQ* and *extQ*. Instead, generic functions are defined as a type class and ad-hoc cases are given as instances. Consider *selectSalary*, for example:

```
class SelectSalary a where
  selectSalary :: a → [Salary]
```

The generic function is the method of a type class. It follows that case analysis on types is performed by the type class system. We show below how to write the generic and the Salary-specific cases:

```
-- case for Salary
instance SelectSalary Salary where
  selectSalary sal = [sal]

-- generic case, not complete yet
instance Data SelectSalary a ⇒ SelectSalary a where
  selectSalary x = gmapQ someProxy
    (...) x
```

The idea of the generic case is that we want to apply *selectSalary* recursively to the top-level sub-trees in *x*. Since *gmapQ* abstracts over the generic function that is applied, it follows that it has to abstract over the type class as well. However, Haskell does not support abstraction over type classes, so in this approach abstraction over type classes is emulated by means of dictionaries.

The first step is to define a dictionary datatype that represents *SelectSalary* instances.

```
data SelectSalaryD a
  = SelectSalaryD { selectSalaryD :: a → [Salary] }
```

Next, every generic function definition must include a *Sat* instance declaration. The *Sat* type class is used to enable SYB3 combinators to construct dictionaries of generic functions.

```
class Sat a where { dict :: a }
instance SelectSalary a ⇒ Sat (SelectSalaryD a) where
  dict = SelectSalaryD selectSalary
```

Finally, combinators that take generic functions as arguments, such as *gmapQ*, include *Sat* in their context to abstract over the dictionary argument. This becomes clearer in the definition of *gmapQ* for lists:

```
instance (Sat (ctx [a]), Data ctx a) ⇒ Data ctx [a] where
  gmapQ _f [] = []
  gmapQ _f (x : xs) = [f x, f xs]
```

The generic case of *selectSalary* looks as follows:

```
instance Data SelectSalaryD a ⇒ SelectSalary a where
  selectSalary x = concat (gmapQ selectSalaryProxy
    (selectSalaryD dict) x)

selectSalaryProxy :: Proxy SelectSalaryD
selectSalaryProxy = undefined
```

This approach uses the same structure representation as SYB. However case analysis on types is implemented using the type class system, and hence it no longer uses type safe casts. However, casts are still used in the library, for example in the definition of the generic equality function.

6.5 Scrap your boilerplate, spine view variant

The Scrap your Boilerplate variant introduced in Hinze et al. [2006] replaces the combinator based approach of SYB by a tangible

representation of the structure of values, which is embodied by the *Spine* datatype:

```
data Spine :: * → * where
  Con :: a → Spine a
  (: $) :: Spine (a → b) → Typed a → Spine b
```

where the *Typed* representation is given by:

```
data Typed a = (:>) { typeOf :: Type a, val :: a }
data Type :: * → * where
  IntR :: Type Int
  ListR :: Type a → Type [a]
  ...
```

This approach represents the structure of datatype values by making the application of a constructor to its arguments explicit. For example, the list $[1, 2]$ can be represented by $Con (:) :\$ (IntR :> 1) :\$ (ListR IntR :> [2])$.

Unlike in LIGD, there is no general purpose constructor like *RType* to support generic universe extension. Generic universe extension is achieved as follows: (1) the datatype must have a *Type* constructor that represents it, e.g. the *ListR* constructor for lists, and (2) the function *toSpine* that transforms a value to its structure representation must be extended to cover that type.

```
toSpine :: Type a → a → Spine a
toSpine (ListR t) [] = Con []
toSpine (ListR t) (x : xs) = Con (:) :\$ (t :> x)
  :\$ (ListR t :> xs)
  ...
```

In *Spine*, case analysis on types is done as in LIGD, by pattern matching on *Type* values.

Generic and non-generic universe extension in *Spine* require recompilation of type representations and generic functions. For this reason *Spine* cannot be used as a library, and so it is a design pattern rather than a library. The authors of *Spine* also describe an extensible variant of *Spine* that is based on type classes (and therefore can be used as a library), but we do not evaluate it in this paper. This variant uses techniques similar to those in SYB3, so we expect that both libraries have similar expressiveness.

Producer generic functions cannot be defined using *Spine*. To solve this deficiency the authors introduced a “type spine view” in Hinze and Löh [2006]. In the evaluation we refer to both approaches as *Spine*. Both views, *spine* and the type spine view, correspond to *gfoldl* and *gunfold* in SYB. As the authors of *Spine* note, *gfoldl* is a fold over *Spine* values.

6.6 Extensible and modular Generics for the masses

The EMGM library [Hinze, 2006, Oliveira et al., 2006] does not use a datatype like *Rep* to represent types. Instead the type representations are encoded in the type class *Generic*, where every represented type has a corresponding method:

```
class Generic g where
  unit :: g Unit
  bool :: g Bool
  plus :: g a → g b → g (Sum a b)
  prod :: g a → g b → g (Prod a b)
  view :: EP b a → g a → g b
```

The type class abstracts over the signature of a generic function, here represented by *g*. To define a generic function, the programmer defines a type for the signature and then the definition is given in the instance declaration for that type. Consider, for example, the equality function. The signature type is defined as follows:

```
newtype Geq a = Geq{geq :: a → a → Bool}
```

The definition of the generic function resides in the `Geq` instance declaration.

```
instance Generic Geq where
  unit      = Geq (λUnit Unit → True)
  bool     = Geq (λx y → eqBool x y)
  plus a b = Geq (λx y → case (x, y) of
    (Inl xl, Inl yl) → geq a xl yl
    (Inr xr, Inr yr) → geq b xr yr
    _                → False)
  prod a b = Geq (λ(Prod a1 b1) (Prod a2 b2) →
    geq a a1 a2 ∧ geq b b1 b2)
  view ep a = Geq (λx y → geq a (from ep x) (from ep y))
```

The equality function is now defined for the universe of types comprising units, sums, products, and datatypes that have their structure represented by `view`, which is similar to the use of the `RType` constructor in LIGD. It follows that in EMGM case analysis on types is encoded using the methods of `Generic`. This is how the structure of lists is represented in EMGM.

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

Extensibility of generic functions is achieved by means of defining sub-classes of `Generic`. For example, we define `GenericList` to enable ad-hoc definitions for lists:

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

The default implementation of `GenericList` uses the structure representation for lists. Therefore we can request generic behaviour for list equality with an empty instance declaration:

```
instance GenericList Geq
```

But we can also give a definition of equality specific to lists:

```
instance GenericList Geq where
  list geqa = Geq (λx y → ...)
```

Now let us see how to apply generic equality:

```
geq (list bool) [True, False] [True, True] ∼∼ False
```

It is possible to make the use of generic functions easier by making the type representations implicit. This is achieved by means of a type class:

```
class GRep g a where
  over :: g a
instance Generic g ⇒ GRep g Unit where
  over = unit
instance (Generic g, GRep g a, GRep g b)
  ⇒ GRep g (Prod a b) where
  over = prod over over
instance (GenericList g, GRep g a) ⇒ GRep g [a] where
  over = list over
```

Now, generic equality can be defined as follows:

```
gequal :: GRep Geq a ⇒ a → a → Bool
gequal x y = geq over
```

6.7 RepLib

The `RepLib` library [Weirich, 2006] uses an ingenious combination of GADTs and type classes to implement generic functions. A generic function in this approach is implemented as a type class. Ad-hoc cases are given as an instance of this class. We use the `gsum` function from the original paper as an example.

```
instance GSum IntSet where
  gsum (IntSet xs) = gsum (nub xs)
```

Here we give an ad-hoc case for sets of integers. This case eliminates duplicate elements and calls generic sum on the resulting list.

What makes `gsum` a generic function, and not-merely type-indexed, is the default implementation, which exploits the structure of datatypes:

```
class Rep1 GSumD a ⇒ GSum a where
  gsum :: a → Int
  gsum = gsumR1 rep1
```

The structure representation for `a` is generated by `rep1`, a method of the `Rep1` type class.

```
class Rep a ⇒ Rep1 c a where
  rep1 :: R1 c a
```

Now, `gsumR1` can use the representation produced by `rep1` to process its argument of type `a`.

But what happens if `gsumR1` needs to recursively apply `gsum` to a substructure inside `a`? In `RepLib` such recursive calls are allowed by parametrising `Rep1` over a dictionary type. In our example the dictionary is `GSumD`, which is defined as follows:

```
data GSumD a = GSumD{gsumD :: a → Int}
```

The representation produced by `rep1` contains `GSumD` dictionaries. These dictionaries package sum instances for a values. These instances are used when `sum` is applied recursively to the argument of a constructor, for example. To produce such dictionaries, the programmer defining the generic function is required to define a `Sat` instance for `GSumD`, it suffices to say that `Sat` is used in `Rep1` instances to produce dictionaries. We give the instance definition for `GSumD` below:

```
class Sat a where dict :: a
instance GSum a ⇒ Sat (GSumD a) where
  dict = GSumD gsum
```

Note that this instance uses `GSum` to produce dictionaries. This means that even `GSum` instances defined in other modules are used. This is what allows `RepLib` generic functions to be extensible. The technique of explicit dictionaries to abstract over type classes is the same as in “Scrap your boilerplate with class”.

The `gsumR1` function is the structure-based definition of generic sum.

```
gsumR1 :: R1 GSumD a → a → Int
gsumR1 Int1      x = ...
gsumR1 (Arrow r1 r2) f = ...
gsumR1 (Data1 dt cons) x
  = case (findCon cons x) of
    Val emb rec kids
      → foldl_1 (λca a b → (gsumD ca b) + a) 0 rec kids
  gsumR1 _      x = 0
```

The two first cases and the last one correspond to integers, functions and all other cases respectively. Without going into detail, the third case uses the structure representation of the datatype (stored in `Data1`) to (1) find the representation for the constructor at hand (using `findCon`), (2) convert the constructor arguments into

a heterogenous list, and (3) fold over this list applying generic sum to the constructor arguments (using *foldl.L*). Note that the last step involves a recursive application of the function. Remember that the representation stores GSumD dictionaries for the constructor arguments. Suppose that *b* (a constructor argument) has type *c* and *ca* has type GSumD *c*, then we can apply the dictionary function using *gsumD*.

In this approach the structure of datatypes is represented by GADTs such as R1. Case analysis over types is performed by the type class system, because generic functions are implemented as type classes (for example, GSum).

The RepLib library has an alternative way to implement case analysis on types. It provides SYB-style combinators such as *mkQ* and *extQ*, these are implemented using type safe casts like in SYB. But these combinators use RepLib representations, rather than the Typeable class.

6.8 Smash your boilerplate

The ‘Smash’ approach is conceptually closely related to SYB. The latter uses a ‘typecase’ operation based on the run-time type representation (Typeable). The Smash approach uses a *compile-time* typecase operation. In both approaches, the structure of a new datatype is presented to the library (added to the universe) by declaring an instance of a special class: Data in SYB, LDat in Smash. A generic function is made of two parts. First, there is a term traversal strategy, identified by a label. One strategy may be to ‘reduce’ a term using a supplied reducing function (cf. fold over a tree). Another strategy may rebuild a term. The second component of a generic function is *spec*, the list of ‘exceptions’, or ad-hoc redefinitions. Each component of *spec* is a function that tells how to transform a term of a specific type. Exceptions override the generic traversal.

As an example, consider how *selectSalary* is defined in Smash:

```
selectSalary :: Company → [Salary]
selectSalary x =
  gapp (TL_red concat) (salaryCase :+: HNil) x
  where
    salaryCase :: Salary → [Salary]
    salaryCase s = [s]
```

Here the library function *gapp* is applied to *TL_red concat*, which selects a bottom-up traversal (parametrised with *concat*) on *x*. This traversal applies one of the ad-hoc cases (second argument of *gapp*) to the nodes of *x* being traversed. When traversing a node, the results of traversing the children are merged using the *concat* function. Note that ad-hoc cases are encoded as a heterogeneous list of functions. In the above example the list contains only one element.

This library implements case analysis using extensible record operations [Kiselyov et al., 2004], due to the way that ad-hoc cases are encoded. The structure representation is given once per datatype and per traversal strategy. To implement the functions in the test suite the following strategies are used:

- A rewriting strategy (*TL_recon*) that is used to implement functions such as *gmap* and *updateSalary*.
- A reduction strategy (*TL_red*) that is used to implement *selectSalary*.
- A reduction strategy that also provides access to constructor names (*TL_red_con*). This strategy is used to implement *gshow*.
- A twin traversal strategy (*TL_red_lockstep*) that is used to implement functions with multiple arguments such as equality.

- A shallow reduction traversal (*TL_red_shallow*) that is used to implement *gmapQ*.
- A couple of reduction traversals that abstract over $\star \rightarrow \star$ -types (*TL_red_cr1*) and $\star \rightarrow \star \rightarrow \star$ -types (*TL_red_cr2*).
- a traversal strategy for producer functions.

6.9 Uniplate

The Uniplate library provides a form of generic programming based on traversal combinators. There are two sorts of traversals: single type and multi-type traversals. Unlike SYB, Uniplate combinators do not require a type system that supports rank-2 types. This is because traversals are customized by functions that are monomorphic rather than polymorphic, as in SYB. An example of a Uniplate combinator is the bottom-up transformation traversal (*transform*).

$$\text{transform} :: \text{Uniplate } a \Rightarrow (a \rightarrow a) \rightarrow a \rightarrow a$$

The *transform* traversal applies the function argument to every a-value that is contained within the a argument. The Uniplate type class is the analog of Data in SYB, it provides a set of common traversals on top of which more sophisticated traversals are defined. The Uniplate type class is equipped with 10 traversal methods. However, all of them can be defined in terms of the fundamental *uniplate* operation (the analog in SYB is *gfoldl*):

$$\text{uniplate} :: \text{Uniplate } a \Rightarrow a \rightarrow ([a], [a] \rightarrow a)$$

This function takes an argument of type *a* and returns a pair of (1) a list of maximal substructures with type *a*, and (2) a function that rebuilds the argument using new values for those substructures. For example, *uniplate (Bin (Leaf 1) (Leaf 2))* yields *([Leaf 1, Leaf 2], Bin)*. The *uniplate* function can be seen as the structure representation of a-values, because all combinator definitions ultimately rely on it.

In Uniplate, the arguments of traversals cannot perform case analysis on types, because they are monomorphic functions. Interestingly, this is not a limitation in practice, because in general the interesting case of a traversal is restricted to one type. This is also the case of functions *selectSalary* and *updateSalary* in our suite.

Uniplate also provides multi-type traversals using multi-parameter type classes. Consider the multi-type variant of *transform*:

$$\text{transformBi} :: \text{Biplate } b \ a \Rightarrow (a \rightarrow a) \rightarrow b \rightarrow b$$

This combinator applies the function argument to all a-values that are contained in the b argument. For example, this is how Uniplate implements *updateSalary*:

```
increase :: Float → Company → Company
increase k = transformBi (incS k)

incS :: Float → Salary → Salary
incS k (S s) = S (s * (1 + k))
```

Multi-type traversals are more flexible than single-type traversals, in that they allow the specification of an ad-hoc case on one type while doing the traversal on another.

6.10 Detailed evaluation

The evaluation is described criterion by criterion below and summarised in Fig. 9.

Universe Size. What are the types that a generic function can be used on? That is, what are the datatypes to which generic universe extension is possible? This question is answered separately for each of the sub-criteria of universe size.

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 apply generic equality to them. The library scores bad otherwise.

The LIGD approach can represent the structure of all datatypes in the universe size test, therefore it scores good on this criterion. The structure of a datatype T of kind \star is represented as a $\text{Rep } T$ value constructed with $RType$. For instance, the datatype `Company` is represented by $rCompany$, which has type $\text{Rep } Company$. Type constructors are represented by functions on representations. For instance, lists are represented by $rList$ which has type $\text{Rep } a \rightarrow \text{Rep } [a]$. The encoding can be generalised to higher-kinded parameters and nested datatypes: they are represented by higher-order functions with rank-2 types.

The PolyLib library is limited to regular datatypes (with one parameter) and cannot handle mutually recursive datatypes, so the set of datatypes (the universe) supported is relatively small.

The SYB library scores well on the universe size criteria, even for the `Perfect` datatype, which is nested. The `GRose` datatype presents difficulties because it has an type argument of kind $\star \rightarrow \star$, and so instances for `Data` and `Typeable` cannot be automatically derived. However, these instances can be written by the programmer, therefore generic universe extension to `GRose` is supported. Note that in the instance for `GRose`

```
instance (Typeable1 f, Typeable a, Data a
         , Data (f (GRose f a)))
  => Data (GRose f a) where ...
```

the instance head $(GRose\ f\ a)$ reappears in the context. This implies that cycle-aware constraint resolution [Lämmel and Peyton Jones, 2005] is required to type this program. In contrast, SYB does not support generic extension to `NGRose`. The reason is that in the `NGRose` instance

```
instance (Typeable1 f, Typeable a, Data a
         , Data (f (NGRose (Comp f f) a)))
  => Data (NGRose f a) where ...
```

the head $(NGRose\ f\ a)$ becomes bigger in the context, namely f becomes $\text{Comp } f\ f$, and therefore cycle-aware resolution is not enough to type check programs using this instance.

The universe of SYB3 is even smaller than that of SYB: `Perfect` is not supported. The instance that is automatically derived looks as follows:

```
instance (Data ctx (Perfect (Fork a)), ...) =>
  Data ctx (Perfect a) where
```

Note that this instance has the same problems as the `NGRose` instance in SYB. Programs that use it, will not type check. The `NGRose` datatype is not supported for the same reason. Universe extension for the `BinTree` datatype is supported, but we surprisingly have to manually write a `Data` instance for it. The reason is that `Derive`, the module that automatically generates representations, produces an erroneous `Data` instance. Indeed, the generated instance causes non-termination at runtime. The reason, we believe, is an erroneous `Typeable` dictionary at runtime, which causes looping when it is used to cast inside generic equality. We give `BinTree` a good score anyway, because this is a problem of `Derive`, rather than of support for regular datatypes.

The `Spine` approach has the advantage of a large universe size: it can handle all datatypes in the universe size test.

In EMGM, the structure of a datatype T is represented by a value of type `Generic g => g T`, which is built using the `view` method. Like in LIGD, type constructors are encoded as functions over representations, for example the type constructor `list` is encoded as `Generic g => g a -> g [a]`. Higher-kinded datatypes such as `GRose` and even `NGRose` can also be encoded in

EMGM. But `NGRose` cannot be used with implicit representations, because the type class that implements them (`GRep`), would need an instance that raises the same issues as the `Data` instance above. In summary, generic extension to `NGRose` is supported but at the cost of reduced functionality. Therefore EMGM scores good for all criteria, except for nested and higher kinded datatypes where it scores sufficient.

The `RepLib` library cannot represent datatypes with higher-kinded arguments. It follows that it satisfies the tests for regular and mutually recursive datatypes only.

The `Smash` library represents the structure of all datatypes in the universe size test. Nested datatypes such as `Perfect` and `NGRose` present problems similar to those in other type class based approaches. However, it is possible to represent them with some effort. The `Company` datatype caused looping during type checking. A workaround is possible, but at the moment we have not identified the exact cause of the problem. Because of the difficulties mentioned, `Smash` scores sufficient on the problematic sub-criteria.

The `Uniplate` library scores well on universe size, it can handle all datatypes except nested generalised rose trees. To support higher-kinded datatypes, the same instances for `Data` and `Typeable` are used as for SYB.

Subuniverses. Is it possible to restrict the use of a generic function to a particular set of datatypes? An approach scores good if uses of the generic function on datatypes outside of the set are flagged as compile-time errors.

The `RepLib` and EMGM approaches score good on this criterion. In both approaches 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 instance (or an instance containing an ad-hoc definition):

```
instance GenericList Geq
```

otherwise compilation fails with a type checking error when applying equality to lists.

`PolyLib` also supports subuniverses - a TIF is limited to the instances defined and this is compiler checked.

In `Smash`, it is possible to specify which datatypes are not to be handled by a generic function. Therefore, `Smash` supports subuniverses “by exclusion”.

First-class generic functions. Can a generic function take a generic function as an argument? If $gmapQ$ can be implemented in a library such that it can be passed a generic function (for example, `gshow`) as argument, the library scores good. If $gmapQ$ can be written but at the price of additional complexity the library scores sufficient. Otherwise, if $gmapQ$ cannot be implemented, the library scores bad.

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. Consider for example $gmapQ$ in LIGD:

```
gmapQ :: (forall a. Rep a -> a -> r) -> Rep b -> b -> [r]
```

Here the function argument is polymorphic, which allows $gmapQ$ to instantiate it to different types. In short, in LIGD, SYB, and `Spine` the generic function argument is just a normal functional argument, albeit a polymorphic one. It follows that LIGD, SYB, and `Spine` score good on this criterion.

EMGM scores sufficient because although EMGM supports first-class generic functions, they are implemented in a rather complicated way. The reason is that the type class system needs to track calls to generic functions. So we are forced to go from a relatively simple (but wrong) signature for `GMapQ`:

```

data GMapQ a = GMapQ{
  gmapQ :: (... → r) → a → [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.

```

data GMapQ g a = GMapQ{
  garg   :: g a,
  gmapQ  :: (∀a. g a → a → r) → a → [r]}

```

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

The test function $gmapQ$ can be defined with no difficulties in SYB3 and RepLib, which therefore score good.

Like in SYB, generic programming in Uniplate is combinator based. However, 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.

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 instantiated to $mapBinTree$ and $flattenWTree$, which have types:

```

mapBinTree :: (a → b) → BinTree a → BinTree b
flattenWTree :: WTree a w → [a]

```

then the approach scores good. If the library can only support the definition of one of the functions or none, it scores sufficient or bad, respectively.

The LIGD, EMGM, and RepLib libraries support the definitions of $gmap$ and $crushRight$ by means of arity-based type representations (Section 3.3.2), and their instantiations yield functions $mapBinTree$ and $flattenWTree$ as required. Hence these libraries score good.

PolyLib includes the definition of $gmap$ and $crushRight$. However these functions can be instantiated only to regular datatypes with kind $\star \rightarrow \star$. It follows that $flattenWTree$ cannot be obtained from $crushRight$ because $WTree$ has kind $\star \rightarrow \star \rightarrow \star$. 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.

The Uniplate and SYB3 libraries represent types with kind \star but they do not represent type constructors. It follows that they are unable to support the definitions of $gmap$ and $crushRight$.

Separate compilation. Is generic universe extension modular? Approaches that can instantiate generic equality to BinTree without recompiling the function definition or the type/structure representation score good.

The LIGD, EMGM, and RepLib libraries score good on this criterion. In LIGD and RepLib, representation types have a constructor to represent the structure of datatypes, namely *RType* and *Data1*. It follows that generic universe extension requires no extension of the representation datatypes and therefore no recompilation. In EMGM, datatype structure is represented by *view*, so a similar argument applies.

PolyLib uses instance declarations (of the *FunctorOf* class) for universe extension, so the score is good.

The SYB, Uniplate, and SYB3 libraries score good on this criterion. Generic universe extension is achieved by defining *Data* and *Typeable* instances for *BinTree*, which does not require recompilation of existing code in other modules. In Smash, the universe is likewise extended by defining instances (of *LDat*).

The Spine library scores bad on this criterion. The reason is that 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.

Ad-hoc definitions for datatypes. Can a generic function contain specific behaviour for a particular datatype, and let the remaining datatypes be handled generically? Moreover, the use of ad-hoc cases should not require recompilation of existing code (for instance the type representations). If the function *selectSalary* can be implemented by a library using an ad-hoc case for the *Salary* datatype, it scores good. Otherwise, it scores bad.

In LIGD and Spine, the ad-hoc case in the *selectSalary* function would have to be given by pattern matching on the type representation constructor that encodes *Salary*, namely *RSalary*. However, this requires the type representation datatype to be extended, and hence the recompilation of the module that contains it. For this reason both approaches score bad.

In PolyLib, the *Company* datatypes cannot be represented (only regular datatypes are supported), so the *selectSalary* test cannot be compiled. But ad-hoc cases is supported for regular datatypes, so PolyLib scores sufficient.

In SYB, ad-hoc cases for queries are supported by means of the *mkQ* and *extQ* combinators. Such combinators are also available for other traversals, for example transformations. The only requirement for ad-hoc cases is that the type being case-analysed should be an instance of the *Typeable* type class. The new instance does not require recompilation of other modules, so SYB scores good on this criterion.

The SYB3, EMGM, and RepLib libraries score good on this criterion. In SYB3 and RepLib, ad-hoc cases are given as an instance of the type class that implements the generic function. In EMGM, ad-hoc cases are given as an instance of *Generic* (or a subclass corresponding to the represented datatype). Because ad-hoc cases are given as type class instances, recompilation is not needed.

In Uniplate, it is possible to define datatype specific behaviour for a multi-type traversal. This is usually achieved by using a traversal combinator that is parametrised over (1) the type on which the traversal is performed, (2) the type for which the ad-hoc case is given, and (3) the ad-hoc case function. Function *transformBi* is such a combinator.

In Smash, a monomorphic ad-hoc definition is given as an element in the list of ad-hoc cases (a function of type $\text{ad_hoc_type} \rightarrow \text{String}$ in case of *gshow*). Smash performs case analysis on types using a type equality operation implemented as a type class, which was originally used to implement extensible records [Kiselyov et al., 2004]. Because no recompilation of the library modules is

needed to allow case analysis over a new type, this library scores good on this criterion.

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 that library scores good on this criterion.

The LIGD and Spine libraries do not support ad-hoc definitions for datatypes. It follows that trying to implement such definitions for constructors would require recompilation (because ad-hoc definitions would be needed). Should we then declare that this criterion is unsupported? We do not think so. The user might be interested in providing an ad-hoc constructor definition, and still be willing to pay the price of lack of support for ad-hoc definitions for datatypes. We make this explicit in Figure 8: it is allowed to cheat in the *rmWeights* test for the ad-hoc definitions for datatypes criterion.

The LIGD and Spine libraries are able to support the definition of *rmWeights* as required. The Spine library scores good, but LIGD scores sufficient because of additional complications. As explained in Section 3.2.3, LIGD needs a modified datatype representation that allows ad-hoc definitions. This gives a total of two representations for WTree, one for generic extension and the other for non-generic extension.

In PolyLib, *rmWeights* can be defined as required, that is, a specific definition for *WithWeight* is given and the other constructors are traversed generically. However, this function can only be applied to WTree values, so PolyLib does not fully support this criterion.

The six approaches that support ad-hoc definitions for datatypes, also support ad-hoc definitions for constructors, hence SYB, SYB3, Uniplate, EMGM, Smash, and RepLib score good on this criterion.

Extensibility. Can a programmer extend the universe of a generic function in a different module without the need for recompilation? Libraries that allow the extension of the generic *gshow* function with a case for printing lists score good. As mentioned before, extensibility is not possible for approaches that do not support ad-hoc cases. For this reason the LIGD and Spine approaches score bad.

Before proceeding to the evaluation, let us remark that a library that supports ad-hoc cases can be made extensible. The trick is to extend the generic function with an argument that receives the ad-hoc case (or cases) with which the generic function is extended. Such a trick would be possible with SYB, for example. However, this is unacceptable 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 and 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 function extensibility would be implemented as proposed in this paragraph, the high-level *quickCheck* function would have to include extension arguments for *shrink*. For this reason, we do not accept such implementations of extensibility in our evaluation.

PolyLib is extensible because it uses class instances to extend the universe. However, it scores sufficient because Company, the datatype that is used in this test, is not supported by PolyLib.

The SYB and Uniplate libraries support ad-hoc definitions, but do not support extensible generic functions. Therefore they score bad on this criterion.

In SYB3, EMGM, and RepLib, ad-hoc cases are given in instance declarations, which can reside in separate modules. Therefore these libraries support extensible generic functions.

In Smash, the traversal for a particular strategy can be overridden for a type. This overriding takes place in a separate module, so Smash supports extensible generic functions.

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 such as *gshow* and *selectSalary*, then the approach scores sufficient.

The LIGD, PolyLib, EMGM, and RepLib approaches support the definition of generic equality. Furthermore, equality is not more difficult to define than other consumer functions. For example, in LIGD, every case of equality matches a type representation and two structurally represented values that are to be compared. Because these two values have the same type, usual pattern matching is enough to give the definition. Defining generic equality is not any more difficult than defining *gshow*. Therefore, these approaches score good on this criterion.

In Spine, the generic equality function is defined as follows:

```
equal :: Typed a → Typed b → Bool
equal x1 x2 = equalSpines (toSpine x1) (toSpine x2)
```

It converts both values to their Spine representation and the real comparison is performed on the spine itself:

```
equalSpines :: Spine a → Spine b → Bool
equalSpines (Con c1) (Con c2) = name c1 == name c2
equalSpines (f1 :$ x1) (f2 :$ x2) = equalSpines f1 f2 ∧
                                     equal x1 x2
equalSpines _ _ = False
```

This function is not more complicated than other consumer functions in this approach. Therefore Spine scores good. It can be argued, however, that this function is not entirely satisfactory. Equality relies ultimately 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 on this criterion. The reason is that the definition 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. The definition can be found in Lämmel and Peyton Jones [2004]

Smash supports multiple arguments to a generic function essentially through currying – a special traversal strategy that traverses several terms in lockstep. 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.

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. The Uniplate library itself does not provide any means to access constructor names.

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, and RepLib.

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 of 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. Why is it a disadvantage to have several structure representations? Because this implies more work for the programmer when doing generic universe extension: more representations have to be written per datatype.

The Uniplate library does not contain combinators to write producer functions, so it scores bad.

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 optimizations so firm conclusions are difficult to draw. As a sample, Figure 10 shows running times (in multiples of the running time of a hand-coded monomorphic version) for the *geq*, *selectInt* and *rmWeights* tests.

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. The fact that the overhead for *rmWeights* starts around a factor of two is disappointing, but shows the improvement potential from using partial evaluation techniques (as used in C++ STL). The best overall performance score is shared between EMGM, Smash and Uniplate.

Portability. Figure 11 shows that the majority of approaches rely on compiler extensions provided by GHC to some extent. Approaches that are most portable rely on few extensions or on extensions that are likely to be included in Haskell prime [Haskell Prime list, 2006].

Of all generic programming libraries, LIGD, EMGM, and Uniplate are the most portable. The only compiler extension that LIGD relies on is existential types, and this extension is very likely to be included in the next Haskell standard [Haskell Prime list, 2006]. However, LIGD needs rank-2 types for the representations of *GRose* and *NGRose*, but not for other representations or functions. Hence rank-2 types are not an essential part of the LIGD approach.

The EMGM library relies on multi-parameter type classes (also likely to be included in the next standard) to implement implicit type representations. This type class makes EMGM more convenient to use, but, even without it, it would still be possible to do generic programming in EMGM, be in a less convenient way.

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

Rank-w polymorphism is required to type the *gfoldl* and *gunfold* combinators in SYB and SYB3.

GADTs are used by the Spine and RepLib libraries to represent types and their structure. Both libraries also use existential types in their representations.

The *unsafeCoerce* extension is used to implement type safe casts in SYB, SYB3, and RepLib.

The RepLib and SYB3 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.

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 extension 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. This is needed to implement the techniques for handling ad-hoc cases and traversal strategies.

Overhead of library use. How much overhead is imposed on the programmer for the use of a generic programming library? We are interested in the following aspects:

- **Automatic generation of representations.** The three libraries that offer support for automatic generation of representations are SYB, SYB3, Uniplate, and RepLib.

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

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 the problems mentioned.

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.

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.

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

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.

- **Number of structure representations.** PolyLib only supports regular datatypes of arity one, thus it only has one representation. It would be straightforward to add a new representation for each arity *a*, but it would still only support regular datatypes.

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 (Section 3.3.2). The latter consists 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 answer is to support the 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.

Test	LIGD	PolyLib	SYB	SYB3	Spine	EMGM	RepLib	Smash	Uniplate
<i>geq</i>	26	8	52	70	13	1.5	8	4	-
<i>selectInt</i>	3	-	5	2	2	3	2	2	0.8
<i>rmWeights</i>	3	-	78	7	2	4	4	2	4

Figure 10. Preliminary performance experiments.

Note, however, that functions of arity one can be defined using a representation of arity three. This means that we could remove representations of arities one and two. So, we could imagine a library needing only two representations. The next step is to subsume the representation for \star -types with the arity three representation. Although this makes some approaches more inconvenient – for instance, the EMGM approach would lose 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 with consumer and transformer functions. And the second, the type spine view, is used with producer functions. The third representation is used to write generic functions that abstract over type constructors of kind $\star \rightarrow \star$.

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 arbitrary number of custom traversal strategies. Although three strategies are fundamental – reconstruction, reduction, and parallel traversal – the simplified variations of these turn out 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 $\star \rightarrow \star$ and $\star \rightarrow \star \rightarrow \star$, and a traversal strategy for producer functions.

- *Work to instantiate a generic function.* The instantiation of a generic function requires a value representing the type to which it is instantiated. The representation can be explicit (the programmer has to supply it) or implicit. Approaches that use type classes can make representations implicit, making instantiation easier for the user. This is the case of SYB, SYB3, EMGM, RepLib, and Smash.

However, some type-class based approaches, such as EMGM and RepLib, require an instance declaration per function per datatype in order to allow instantiation of the function to that datatype. This requires additional work from the user, but on the other hand this enables control over the instantiation of a function (subuniverses).

The SYB, Smash, Uniplate, and SYB3 libraries have implicit representation arguments and do not need instance declarations per function per type, therefore these libraries score good.

- *Work to define a generic function.* Is it possible to quickly write a simple generic function? A library scores good if it requires roughly one definition per generic function case, with no need for additional artefacts.

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. The other libraries score good because they demand less “encoding work” when defining a generic function.

Practical aspects. For this criterion we consider aspects such as

- there is a library distribution available online,
- the library interface is well-documented,
- and other aspects such as an interface organized into modules, and common generic functions.

The LIGD, EMGM, and Spine libraries do not have distributions online. These libraries score bad.

PolyLib has an online 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` amongst others.

The RepLib library has an online distribution. The distribution includes a number of pre-defined generic functions, the structure representation generation machinery and Haddock documentation. The score of this library is good.

The Uniplate library and haddock documentation are available online (also via HackageDB). Just like SYB it can derive representations. This library scores good.

The Smash library is distributed as a standalone module that can be downloaded from <http://okmij.org/ftp/Haskell/generics.html#Smash>. There are a few example functions in

	LIGD	PolyLib	SYB	SYB3	Spine	EMGM	RepLib	Smash	Uniplate
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		●	●	●		●	●	●	●
Abstraction over type classes				●			●		
Compiler extensions									
Undecidable instances		●		●				●	
Overlapping instances				●				●	
Multi-parameter typeclasses		●				●		●	●
Functional dependencies								●	
Rank-2 polymorphism			●	●					
Existential types	●				●		●		
GADTs					●		●		
<i>unsafeCoerce</i>			●	●			●		
Template Haskell				●			●		●
Derive Data and Typeable			●						●
Views									
Fixed point view		●							
Sum of products	●					●			
Spine			●	●	●				
Lifted spine					●				
RepLib							●		
Uniplate									●
Smash								●	

Figure 11. Implementation mechanisms, required compiler extensions, and views.

that module that illustrate the use of the approach. However, the library is not as organized as SYB3, and therefore it scores bad.

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 converts a-values into b-values at runtime, where a and b are unknown statically.
- *Extensible records and type-level evaluation.* The work of Kiseiyov 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 to generic functions.

- *Abstraction over type classes.* Generic programming libraries that support extensible generic functions, do so by using type classes. Some of this approaches, however, require a form of abstraction over type classes, which can be encoded by a technique that uses explicit dictionaries, popularized by Lämmel and 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 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-level typecase. 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.

Views. What are the views that the generic library supports? That is, how are datatypes encoded in structure representations and what are the view types used in them?

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 sum of products. The representations for sums of products that are based on arities (Section 3.3.2) can be used to abstract over type constructors.

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 a catamorphism of a Spine value, and hence these approaches also use the spine view. The SYB, SYB3, and Spine approaches also provide a type-spine view that is used to write producer generic functions. Unlike SYB and SYB3, Spine supports abstraction over $\star \rightarrow \star$ -types due to an additional view called the lifted Spine view.

The RepLib library also has a different view, datatypes are represented as a list of constructor representations, which are a heterogeneous list of constructor arguments. Like in LIGD and EMGM, the structure representation can be adapted to be arity based, in order to support abstraction over type constructors.

The Uniplate library also has a view of its own. All traversal combinators in this library are based on the *uniplate* operation. This operation takes an argument of a particular datatype, and returns its children that have the same type as the argument, and a function to reconstruct the argument with new children.

It is difficult to point to specific views in the Smash library. Although there are three basic strategies (rewriting, reduction, and lockstep 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.

Ease of learning and use. It is hard to determine how easy it is to learn how to use a library. We approximate this criterion by looking at the mechanisms used in the implementation of the libraries. Below we describe the difficulties that arise with some of the implementation mechanisms:

- *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, by-passing 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.
- *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.
- *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 and it might confuse beginning generic programmers.

- *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 $\text{Rep2 } a \ b \rightarrow a \rightarrow b$, which does not bear a strong resemblance to the type of the BinTree instance of map: $(a \rightarrow b) \rightarrow \text{BinTree } a \rightarrow \text{BinTree } b$. 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 use of an approach is the use of a relatively large number of implementation mechanisms. This is the case of SYB3 and RepLib. While it is possible to work out how one of the many mechanisms, for example GADTs in RepLib, is used into 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 and RepLib.

7. Conclusions

We have introduced a set of criteria to compare libraries for generic programming in Haskell. These criteria can be viewed as a characterisation of generic programming in Haskell. Furthermore, we have designed a generic programming benchmark: a set of characteristic examples that check whether or not criteria are supported by generic programming libraries. Using the criteria and the benchmark, we have compared nine approaches to generic programming in Haskell.

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.

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. We now discuss which libraries are most suitable for implementing one of three typical generic programming scenarios.

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 parameterised by the actual transformations. Traversals also require little work to define a generic function, because users often define their own traversals. 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 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 from the user. For serialization, 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.

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. 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. Finally, Andres Löh provided useful comments and formatting tips.

References

- F. Atanassow and J. Jeuring. Inferring type isomorphisms generically. In *MPC'04*, LNCS 3125, pages 32–53, 2004.
- J.-P. Bernardy, P. Jansson, M. Zalewski, S. Schupp, and A. Priesnitz. A comparison of C++ concepts and Haskell type classes. In *ACM SIGPLAN Workshop on Generic Programming*, 2008.
- R. Bird and L. Meertens. Nested datatypes. In J. Jeuring, editor, *MPC'98*, LNCS 1422, pages 52–67, 1998.
- B. Bringert and A. Ranta. A pattern for almost compositional functions. In *ICFP'06*, pages 216–226, 2006.
- J. Cheney and R. Hinze. A lightweight implementation of generics and dynamics. In *Haskell'02*, pages 90–104, 2002.
- K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP'00*, pages 286–279, 2000.
- D. Clarke and A. Löh. Generic haskell, specifically. In *Proceedings of the IFIP TC2/WG2.1 Working Conference on Generic Programming*, pages 21–47. Kluwer, B.V., 2003.
- J. Derrick and S. Thompson. FORSE: Formally-Based Tool Support for Erlang Development. Project description, 2005. URL <http://www.cs.kent.ac.uk/projects/forse/>.
- R. Garcia, J. Järvi, A. Lumsdaine, J. Siek, and J. Willcock. An extended comparative study of language support for generic programming. *J. Funct. Program.*, 17(2):145–205, 2007.
- Haskell Generic Library list. Generic programming criteria template, 2008. Wiki page at haskell.org/haskellwiki/Applications_and_libraries/Generic_programming.
- T. Haskell Prime list. Haskell prime, 2006. Wiki page at <http://hackage.haskell.org/trac/haskell-prime>.
- R. Hinze. Generics for the masses. *Journal of Functional Programming*, 16:451–482, 2006.
- R. Hinze. Polytypic values possess polykinded types. *Science of Computer Programming*, 43(2–3):129–159, 2002.
- R. Hinze and A. Löh. “Scrap Your Boilerplate” revolutions. In *MPC'06*, LNCS 4014, pages 180–208, 2006.
- R. Hinze and A. Löh. Generic programming, now! In R. Backhouse et al., editors, *Datatype-Generic Programming*, LNCS, pages 150–208. 2007.
- R. Hinze, A. Löh, and B. C. d. S. Oliveira. “Scrap Your Boilerplate” reloaded. In P. Wadler and M. Hagiya, editors, *FLOPS'06*, LNCS 3945, 2006.
- R. Hinze, J. Jeuring, and A. Löh. Comparing approaches to generic programming in haskell. In *Datatype-Generic Programming*, LNCS 4719, pages 72–149. 2007.
- S. Holdermans, J. Jeuring, A. Löh, and A. Rodriguez. Generic views on data types. In T. Uustalu, editor, *MPC'06*, volume 4014 of *LNCS*, pages 209–234, 2006.
- P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A history of Haskell: being lazy with class. In *HOPL III*, pages 12–1–12–55, 2007.
- P. Jansson and J. Jeuring. PolyP — a polytypic programming language extension. In *POPL'97*, pages 470–482, 1997.
- P. Jansson and J. Jeuring. Polytypic data conversion programs. *Science of Computer Programming*, 43(1):35–75, 2002.
- P. Jansson and J. Jeuring. PolyLib – a polytypic function library. Workshop on Generic Programming, Marstrand, 1998.
- O. Kiselyov. Smash your boilerplate without class and typeable. <http://article.gmane.org/gmane.comp.lang.haskell.general/14086>, 2006.
- O. Kiselyov. Compositional gmap in sybl. <http://www.haskell.org/pipermail/generics/2008-July/000362.html>, 2008.
- O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107, New York, NY, USA, 2004. ACM Press. ISBN 1581138504. URL <http://portal.acm.org/citation.cfm?id=1017488>.
- P. Koopman, A. Alimarine, J. Tretmans, and R. Plasmeijer. Gast: Generic automated software testing. In R. Peña and T. Arts, editors, *IFL'02*, volume 2670 of *LNCS*, 2003.
- R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. In *TLDI*, pages 26–37, 2003.
- R. Lämmel and S. Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *ICFP*, pages 244–255, 2004.
- R. Lämmel and S. Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *ICFP*, pages 204–215, 2005.
- R. Lämmel and J. Visser. A Strafunski Application Letter. In *PADL'03*, LNCS 2562, pages 357–375, 2003.
- R. Lämmel and J. Visser. Typed combinators for generic traversal. In *PADL'02*, LNCS 2257, pages 137–154, 2002.
- H. Li, C. Reinke, and S. Thompson. Tool support for refactoring functional programs. In *Haskell'03*, pages 27–38, 2003.
- A. Löh, D. Clarke, and J. Jeuring. Dependency-style Generic Haskell. In *ICFP'03*, pages 141–152, 2003.
- I. Lynagh. Template Haskell: A report from the field. <http://www.comlab.ox.ac.uk/oucl/work/ian.lynagh/papers/>, 2003.
- L. Meertens. Calculate polytypically! In H. Kuchen and S. D. Swierstra, editors, *PLILP*, LNCS 1140, pages 1–16, 1996.
- N. Mitchell and C. Runciman. A static checker for safe pattern matching in Haskell. In *Trends in Functional Programming*, volume 6. Intellect, 2007a.

- N. Mitchell and C. Runciman. Uniform boilerplate and list processing. In *Haskell'07*, 2007b.
- M. Naylor and C. Runciman. Finding inputs that reach a target expression. In *SCAM'07*, pages 133–142, 2007.
- U. Norell and P. Jansson. Polymorphic programming in Haskell. In *IFL'03*, LNCS 3145, pages 168–184, 2004.
- B. C. d. S. Oliveira, R. Hinze, and A. Löb. Extensible and modular generics for the masses. In H. Nilsson, editor, *Trends in Functional Programming*, pages 199–216, 2006.
- S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *ICFP'06*, pages 50–61, 2006.
- C. Reinke. Traversable functor data, or: X marks the spot. <http://www.haskell.org/pipermail/generics/2008-June/000343.html>, 2008.
- P. Wadler. Theorems for free! In *FPCA'89*, pages 347–359. 1989.
- S. Weirich. RepLib: a library for derivable type classes. In *Haskell'06*, pages 1–12, 2006.
- N. Winstanley and J. Meacham. *DrIFT user guide*, 2006. <http://repetae.net/~john/computer/haskell/DrIFT/>.