

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 understandable. 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 [Derrick 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 sup-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Haskell'08, September 25, 2008, Victoria, BC, Canada.
Copyright © 2008 ACM 978-1-60558-064-7/08/09...\$5.00.

port, 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]
- Polytropic 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 Strafinski [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 evaluates the different libraries with respect to the criteria, using the benchmark. For reasons of space, the evaluation cannot be presented in full detail, a complete evaluation can be found in the technical report version of this paper [Rodriguez et al., 2008]. Section 6 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

```

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

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 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 library, 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 “Polytypic 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

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.

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.

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.

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
flattenBinTree :: BinTree a  $\rightarrow$  [a]
flattenBinTree (Bin (Leaf 2) (Leaf 1))  $\rightsquigarrow$  [2,1]
```

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

```
crushRight :: Rep' f  $\rightarrow$  (a  $\rightarrow$  b  $\rightarrow$  b)  $\rightarrow$  f a  $\rightarrow$  b  $\rightarrow$  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:

$$\begin{aligned} \text{sumList} \quad \quad \quad \text{xs} &= \text{crushRight } rList' (+) \text{ xs } 0 \\ \text{flattenBinTree } bt &= \text{crushRight } rTree' (:) \text{ bt } [] \end{aligned}$$

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.

$$\begin{aligned} \text{flattenWTWeights} &:: \text{WTree } a \text{ w} \rightarrow [\text{w}] \\ \text{flattenWTWeights } (WBin \ (WithWeight \ (WLeaf \ 1) \ 2) \\ &\quad \quad \quad (WithWeight \ (WLeaf \ 3) \ 4)) \\ &\rightsquigarrow [2, 4] \end{aligned}$$

This is just an instance of *crushRight*:

$$\text{flattenWTWeights } tree = \text{crushRight } rWTree' (:) \text{ tree } []$$

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:

$$\begin{aligned} \text{selectInt } (WBin \ (WithWeight \ (WLeaf \ 1) \ 2) \\ &\quad \quad \quad (WithWeight \ (WLeaf \ 3) \ 4)) \\ &\rightsquigarrow [1, 2, 3, 4] \end{aligned}$$

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*:

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

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

$$\text{gmap } rBinTree' :: (\text{a} \rightarrow \text{b}) \rightarrow \text{BinTree } \text{a} \rightarrow \text{BinTree } \text{b}$$

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 $\text{T} \rightarrow \text{T}'$: it converts *T* values into *T'* values. Jansson and Jeurig [2002] and Atanassow and Jeur-

ing [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:

$$\text{gtranspose} :: \text{Rep}' \text{ f} \rightarrow \text{Rep}' \text{ g} \rightarrow \text{f } (\text{g } \text{a}) \rightarrow \text{g } (\text{f } \text{a})$$

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

$$\begin{aligned} \text{transpose} &= \text{gtranspose } rList' \text{ rList}' \\ \text{transpose } [[1, 2, 3], [4, 5, 6]] &\rightsquigarrow [[1, 4], [2, 5], [3, 6]] \end{aligned}$$

It can also be instantiated to other datatypes:

$$\begin{aligned} \text{gtranspose } rList' \text{ rBinTree}' [Leaf \ 1, Leaf \ 2, Leaf \ 3] \\ &\rightsquigarrow Leaf \ [1, 2, 3] \end{aligned}$$

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:

$$\begin{aligned} \text{gfulltree} &:: \text{Rep } \text{a} \rightarrow \text{Int} \rightarrow [\text{a}] \\ \text{gfulltree } (rList \text{ rInt}) \ 4 &\rightsquigarrow [[], [0], [0, 0], [0, 0, 0]] \\ \text{gfulltree } (rBinTree \text{ rInt}) \ 4 &\rightsquigarrow \\ &[Leaf \ 0 \\ &, Bin \ (Leaf \ 0) \quad \quad \quad (Leaf \ 0) \\ &, Bin \ (Leaf \ 0) \quad \quad \quad (Bin \ (Leaf \ 0) \ (Leaf \ 0)) \\ &, Bin \ (Bin \ (Leaf \ 0) \ (Leaf \ 0)) \ (Leaf \ 0) \\ &, Bin \ (Bin \ (Leaf \ 0) \ (Leaf \ 0)) \ (Bin \ (Leaf \ 0) \ (Leaf \ 0))] \end{aligned}$$

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

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.

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 5. Criteria overview

- Usability: How convenient a library is to use, efficiency, quality of library distribution, portability.

Figure 5 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 from BinT to BinT)
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 6 shows the coverage of testable criteria. The rows represent testable criteria and the columns represent the means of testing them. The first group of columns stand for the 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 6. 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 ○.

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 7 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 can be found in the technical report version of this paper.

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 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 *SYB*, *SYB3*, and *Uniplate* libraries do not support this criterion. However, it is possible to define *gmap* in *SYB* using the *unsafeCoerce* extension [Reinke, 2008]. As such generic function definitions break type safety, we do not consider them in the comparison.

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*.

	<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 6. Functions and datatypes set out against criteria.

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, 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 arity¹. The number of representations in

¹ Informally, the arity of a generic function is the number of type constructor occurrences in the signature of the function. For example, the arities of *crushRight* and *gmap* are 1 and 2 respectively.

Smash is even higher due to the amount of relatively specialised representations. More information can be found in the technical report. 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

	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 7. Evaluation of generic programming approaches

such approach is EMGM, which uses arity-based representations. More information can be found in the technical report.

6. 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, 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.
- 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. Polytypic programming in Haskell. In *IFL'03*, LNCS 3145, pages 168–184, 2004.
- B. C. d. S. Oliveira, R. Hinze, and A. Löh. 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.
- A. Rodriguez, J. Jeuring, P. Jansson, A. Gerdes, O. Kiselyov, and B. C. d. S. Oliveira. Comparing libraries for generic programming in haskell. Technical report, Utrecht University, 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/>.