Comparing **Generic Deriving** and **Multiplate** to other Generic Programming Libraries in Haskell

Beerend Lauwers Augusto Martins
Frank Wijmans
{B.Lauwers, A.PassalaquaMartins, F.S.Wijmans}@students.uu.nl
Utrecht University, The Netherlands

November 11, 2011

Abstract

Research on datatype-generic programming in Haskell has been ongoing for over ten years, and new generic programming libraries continue to appear. We used an updated version of the benchmark suite by Rodriguez et al. to review and compare two new libraries: Multiplate and Generic Deriving. Furthermore, we extended the comparison with a review of MultiRec provided by the PhD thesis of Rodriguez. Multiplate is a combination of the Uniplate and Compos libraries, which allows features of both to be used alongside each other. Generic Deriving uses compiler support to automatically derive instances for its representation type and minimize boilerplate code. We compared these two libraries against MultiRec, Scrap Your Boilerplate, Extensible Modular Generics for the Masses and Uniplate. Multiplate requires little boilerplate code compared to other libraries, and defining functions is quite elegant. Unfortunately, its feature set is very limited. This is not the case with Generic Deriving, which passes the majority of the tests. However, its universe size is limited by the fact that it only abstracts over a single type parameter. Further development of the Generic Deriving library appears to be promising, and Multiplate may be of interest for fast traversal of AST-like structures.

1 Introduction

Since the beginning of this millennium, there has been a flood of proposals for generic programming libraries in Haskell. As the number of libraries grew, so has the need for an answer to the following question: *What is the best library for a given problem?*

Several articles have been published to address this question, and it continues to be relevant today. In this article, we are comparing Multiplate and Generic Deriving on top of the comparison done by Rodriguez et al. [3]. We will follow their style and add these libraries to the comparison to further complete the overview. Rodriguez et al. [3] already compared the following libraries:

- Lightweight Implementation of Generics and Dynamics (Cheney and Hinze [4])
- Polytypic programming in Haskell (Norell and Jansson [5])
- Scrap your boilerplate (Lämmel and Peyton Jones [6])

- Scrap your boilerplate, extensible variant using typeclasses (Lämmel and Peyton Jones [7])
- Scrap your boilerplate, spine view variant (Hinze and Löh [8])
- Extensible and Modular Generics for the Masses (Oliveira et al. [9])
- RepLib: a library for derivable type classes (Weirich [10])
- Smash your boilerplate (Kiselyov [11])
- Uniplate (Mitchell and Runciman [12])

We are comparing libraries that have similar workings to the new libraries. The reader is invited to read the article by our predecessors to see the comparison on the other libraries [1]. We will compare the following libraries:

- Multiplate (O'Connor [13])
- Generic Deriving (Magalhães et al. [14])
- MultiRec (Rodriguez et al. [15])
- Scrap your boilerplate (SYB) (Lämmel and Peyton Jones [6])
- Extensible and Modular Generics for the Masses (EMGM) (Oliveira et al. [9])
- Uniplate (Mitchell and Runciman [12])

SYB was chosen because it is included with GHC, and boasts a large feature set. EMGM is similar to Generic Deriving in terms of its feature set and some internal mechanisms, and Uniplate is essentially isomorphic to Multiplate, so it is an excellent candidate for comparison.

MultiRec was previously reviewed in the PhD thesis of Rodriguez [2], of which the fourth chapter builds upon the comparison of Rodriguez et al. [1]. We shall use this chapter for our evaluation of MultiRec.

We will extend the existing comparison by adding the Multiplate and Generic Deriving libraries. Furthermore, we will update the existing benchmark suite so that it will compile and run our reviewed libraries using GHC 7.2. We leave out MultiRec's benchmark code, as it has been recently reviewed by Rodriguez [2]. The code can be found in https://subversion.cs.uu.nl/repos/project.dgp-haskell.libraries/multirec/branches/Erik/examples/gpbench/.

This paper places emphasis on Multiplate and Generic Deriving. The other libraries will not be discussed in-depth, as these discussions can be found in the previous comparison articles.

The structuring of this article is as follows. In Section 2, we will shortly explain the concept of Generic Programming (GP) libraries, and introduce Multiplate and Generic Deriving. After that, in Section 3, we shall explain the details about the benchmark as created by Rodriguez et al. [1]. Section 4 explains the criteria which we will use to evaluate the libraries in Section 5. Section 6 concludes and suggests future work.

2 Concepts and terminology of Generic programming

The goal of generic programming is to create functions that can be applied to more than one datatype. This is done by defining a function, for example equality, once, and then be able to use it on a larger family of datatypes.

A function, defined for a family of types, is called a *type-indexed function* (TIF). These can be used to create a generic function. The types in a family index the TIF, while a family of types are called a *universe*. When adding

new types to a universe of datatypes for which generic equality is defined, the programmer should be able to extend the universe without rewriting the generic equality function. So, to be able to extend a universe with new datatypes, these datatypes should be able to be described using the existing types. This is called a *type representation*. Defining the representation of a datatype consists of two things. First, we have the structure of the datatype, called a *structure representation*, defined from (if we use the sum-of-products view) units, sums, products, etc. Secondly, we have two dual functions that are often called a *projection pair*. These functions are usually called *from* and *to*, either creating a representation *from* a datatype, or converting a representation *to* a datatype. The choice of a generic view (that is, the way the structure representation is represented) often influences the expressiveness of a library. Some libraries use GADTs for assigning representations to datatypes. Others use the sum-of-products view, where representations can consist of units, constants, sums, products and other types, such as composition and recursion representations.

There are many ways to define generic functions. For example, one way is creating a class and defining instances for each of the structure representation types.

To test a given library on the relevant aspects of generic programming, we need to specify what the usual features of such a library should be. The article by Rodriguez et al. [1] proposes a list of recommended features of a generic programming library. We give a short summary and advise the reader to read the aforementioned article for a more detailed overview.

Before we proceed to Section 3, we will elaborate on the generic views of Generic Deriving and Multiplate in Section 2.1 and 2.2. For a description of the other libraries, the reader is invited to read section 4 of the PhD thesis by Rodriguez [2] .

2.1 Generic Deriving

The Generic Deriving library uses a new feature in GHC 7.2 to automatically derive instances for the *Generic* class, after which the datatype can be used in generic functions supported by Generic Deriving. It supports many datatypes, with only nested higher-kinded datatypes, datatypes with a context, existentially-quantified datatypes and GADTs being unsupported.

To define a generic function, one creates a class and defines instances for each of the representation types of the universe.

```
class GEq' f where
  geq' :: f a → f a → Bool
instance GEq' U1 where
  geq' _ _ = True
instance (GEq c) ⇒ GEq' (K1 i c) where
  geq' (K1 a) (K1 b) = geq a b

  --Etc
```

Another class is then created that houses a default method which takes a value of a type and transforms it into a representation and passes it to the generic function:

```
class GEq a where
  geq :: a → a → Bool
  default geq :: (Generic a, GEq' (Rep a))
      ⇒ a → a → Bool
  geq x y = geq' (from x) (from y)
```

To define ad-hoc cases for a generic function, one simply overrides the default method:

```
-- Not overridden
instance (GEq a) ⇒ GEq (Maybe a)
instance (GEq a) ⇒ GEq [a]

-- Overridden
instance GEq Char
  where geq = (==)
instance GEq Int
  where geq = (==)
instance GEq Float
  where geq = (==)
```

2.2 Multiplate

The Multiplate library is a combination of the Uniplate and the Compos libraries. O'Connor

[13] shows that the core datatypes of Uniplate and Compos are isomorphic, allowing features of both Uniplate and Compos to be used and even combined. The only compiler extension the library requires is rank-3 polymorphism. The library natively supports multi-type traversals for mutually recursive datatypes. Uniplate requires multi-parameter type classes for this functionality. Using Multiplate is also more intuitive compared to Uniplate when dealing with mutually recursive datatypes.

To implement a generic function, one first defines a *Plate* record type for the mutually recursive datatype:

```
-- Simple mutually recursive language
data Expr = Con Int
          | Add Expr Expr
          | EVar Var
          | Let Decl Expr
          deriving (Eq, Show)

data Decl = Var := Expr
          | Seq Decl Decl
          deriving (Eq, Show)

type Var = String

-- Plate record type
data Plate f = Plate
           { expr :: Expr → f Expr
           , decl :: Decl → f Decl
           }
```

Then, an instance of *Multiplate* is defined:

```
instance Multiplate Plate where
multiplate child = Plate buildExpr
    buildDecl
  where
    buildExpr (Add e1 e2) = Add <$> expr
        child e1 <*> expr child e2
    buildExpr (Let d e) = Let <$> decl
        child d <*> expr child e
    buildExpr e = pure e
    buildDecl (v := e) = (:=) <$> pure v
        <*> expr child e
    buildDecl (Seq d1 d2) = Seq <$> decl
        child d1 <*> decl child d2
mkPlate build = Plate (build expr) (build
    decl)
```

After this, users can use the traversal functions of Multiplate to quickly write generic functions.

```
-- Collect all variable names
varPlate :: Plate (Constant [Var])
varPlate = purePlate { expr = exprVars }
 where
  exprVars (EVar v) = Constant [v]
  exprVars x = pure x

collectVars :: Expr → [Var]
collectVars = foldFor expr $ preorderFold
    $ varPlate
```

3 Benchmark suite

We used the same benchmark suite as Rodriguez et al. [1], bringing it up to date to work with our reviewed libraries in GHC 7.2. We will summarize the test cases of the benchmark suite. For details, please refer to the earlier comparison article. First, we will talk about the scenarios they proposed. Following that, we will elaborate on what is needed to test these scenarios.

3.1 Scenarios

As Rodriguez et al. [1] stated in their article, the tested libraries should support common generic programming scenarios. The following scenarios were tested in the benchmark:

- Generic equality
- Generic Show
- Querying / Traversals
- Transformation functions
- Consumer functions
- Producer functions

From these scenarios, we further specified requirements for the details of the benchmark. Firstly, we required several classes of datatypes to work with. Secondly, we needed functions that work on these datatypes. From these two elements, we drew our conclusions and compared the libraries.

3.2 Datatypes

The datatypes used in the benchmark are realistic and have been well-chosen to complement the scenarios to be tested.

Datatypes may contain the following properties:

- Parametrised types
- Simple and nested recursion
- Higher-kinded datatypes

Along with these classes of datatypes, there were other classes and datatype information that were not tested:

- Higher-rank constructors
- Existential types
- GADT's
- Parsing related information (record label names, constructor fixity and precedence)

The first three are hardly supported in any library, while the latter can be incorporated with constructor names.

3.2.1 Defined datatypes

The following datatypes were implemented to allow testing. Listed first is the datatype description, second is the name as implemented in the benchmark.

- Company mutually recursive datatype, *Company*
- Binary trees, *BinTree*
- Trees with weight, *WTree*
- Generalised rose trees, *GRose*
- Nested perfect trees, *Perfect*
- Nested generalised rose trees, *NGRose*

3.3 Functions

To test the scenarios, we needed a set of functions that exhibit the required properties according to a scenario. The following functions were implemented to test all scenarios. Please note that constraints and type representation arguments have been left out for simplicity's sake.

- **Equality**
  $geq :: a \to a \to Bool$
- **Show**
  $gshow :: a \to String$
- **Querying / Traversals**
  $selectSalary :: a \to [Salary]$
  $gmapQ :: (\forall a.a \to r) \to b \to [r]$
- **Transformation Functions**
  $updateSalary :: Float \to a \to a$
  $rmWeights :: a \to a$
  $gmap :: (a \to b) \to fa \to fb$
- **Consumer Functions**
  $crushRight :: (a \to b \to b) \to fa \to b \to b$
- **Producer Functions**
  $fulltree :: Int \to [a]$

4 Criteria

In this section, we give an overview of the criteria that a generic programming library is subjected to. We have grouped the criteria in the same manner as Rodriguez et al. [1] for easy comparison:

4.1 Types

When looking at types, we make a distinction between the universe and subuniverses.

Universe A generic function is a function that can be applied to a certain family of types. The more types a function can be used on,

the larger the universe size of that function. It is inevitable that universes differ between libraries.

Subuniverses A library supports subuniverses if it is possible to restrict a generic function to some group of datatypes. Types outside the group should throw a compile-time error when used on the function.

4.2 Expressiveness

This group discusses the library's capabilities of defining several types of generic functions, as well as its flexibility in doing so.

First-class generic functions Can generic functions take a generic function as an argument?

Abstraction over type constructors Is it possible to apply a generic function to an arbitrary container type? In other words, can generic map and generic fold be defined?

Separate compilation Does the library have to be recompiled for universe extension?

Ad-hoc definitions Can we define specific behaviour for particular datatypes or constructors?

Extensibility Can the programmer non-generically extend the universe of a generic function in a different module?

Multiple arguments Can a generic function have more than one generic argument?

Constructor names Can the library provide constructor metadata?

Consumers, transformers and producers Is the approach capable of defining generic functions that are of signature $(a \to T)$, $(a \to a)$, $(a \to b)$ and $(T \to a)$?

4.3 Usability

When trying to find out what generic programming library one wants to use, usability will be a very important criterion. Consequently, we have included several criteria that gauge the overall usability of a library. We have left out the performance criterion, as we believe an in-depth comparison would be out of the scope of this article, while a shallow comparison does not accurately portray the strengths and weaknesses of each library, performance-wise.

Portability How many compiler extensions does a library require, and how common are they?

Overhead of use How much time must a user invest before he is able to make use of the library's features? This criterion is subdivided into the following subcriteria:

- *Automatic generation of representations*: Can type representations be generated automatically?
- *Number of structure representations*: Having many structure representations may influence the amount of work a user must do to create a generic function.
- *Work to instantiate a generic function*: How much code must be written in order to use a generic function?
- *Work to define a generic function*: How much code must be written to define a new generic function?

Practical aspects How well is the library and its documentation maintained?

Ease of learning and use How difficult is it to understand the library's functionalities and mechanisms?

4.4 Design Choices

Depending on how a library is designed, some functionalities or features may require extra work to implement, or even be impossible to express in the library. Hence, analysing a library's internal workings also yields valuable information.

Implementation mechanisms How are types represented in the library, and what limitations does that impose?

Views What views does the library support? As an example, the sum-of-products view is an often-supported view in generic programming libraries.

5 Evaluation Summary

We have ordered the benchmark results in the same manner as Rodriguez et al. [1] to facilitate further comparison. Annex table 1 shows an overview of the comparison results. Furthermore, we explain in detail the evaluation of the different libraries and compare them in the rest of this section.

5.1 Universe Size

It was unclear as to how higher-kinded datatypes and nested datatypes could be defined in Multiplate, even though Uniplate supports nested datatypes, and higher-kinded types with extra programming effort. For that reason, Multiplate scores sufficient.

Generic Deriving's universe is about the same size as SYB's and Uniplate's: it is able to represent nearly all tested datatypes. Only nested higher-kinded datatypes cannot be represented, as the library's representation only abstracts over a parameter of kind $\star$ (Magalhães et al. [14]). This also means that datatypes with multiple parameters require extra work to be implemented, and the available generic functions will be less useful for these datatypes. Generic Deriving also scores sufficient.

MultiRec cannot represent higher-kinded datatypes as well as nested datatypes, but supports all other datatypes properly. It scores sufficient as well.

5.2 Subuniverses

Uniplate, Multiplate and MultiRec do not support subuniverses as proposed in the benchmark of the comparison paper.

Like EMGM, the Generic Deriving library supports subuniverses through generic functions that are only applicable to types that instantiate the class of the generic function. So, it gets the same score as EMGM.

5.3 First-class generic functions

In Multiplate, like in Uniplate, it is not clear how *gmapQ* could be implemented and for the moment it is marked as unsupported in the results.

MultiRec also does not support first-class generic functions.

The authors of Generic Deriving noted that $gfoldl$ cannot be expressed in their library yet, which made defining *gmapQ* unclear. They have declared that they will continue to investigate in order to support $gfoldl$ and other generic functions. Therefore, first-class generic functions are not marked as supported.

5.4 Abstraction over type constructors

Multiplate only represents types of kind $\star$. Hence, it does not support *gmap* or *crushRight* and has a bad score.

Generic Deriving includes the definition of *gmap*, and defining *crushRight* was mainly trivial, with only composition being harder to define. In the end, composition was left out,

as it narrowed down the possible functions that could work with *crushRight*. The library scores well.

MultiRec does not support this criterion.

5.5 Separate Compilation

Just like Uniplate, Multiplate supports generic universe extension. Multiplate does so by defining a *Plate* for the datatype, which can then be used in the generic traversal functions.

In Generic Deriving, it is possible to extend the universe using the *Generic* and *Generic*1 classes.

MultiRec uses the same mechanism.

5.6 Ad-hoc definitions for datatypes

Multiplate easily supports ad-hoc cases for certain datatypes through the function *purePlate*, which is used to build a *Plate* that does nothing. This *Plate* can then be updated on a per-record basis:

```
purePlate :: (Multiplate p, Applicative f)
    ⇒ p f
purePlate = mkPlate (\_ → pure)

updatedPlate = purePlate
 { expr = (\ x → Constant [x]) }
```

The library scores well.

As in EMGM, Generic Deriving defines ad-hoc cases in instance declarations which allow for extensible generic functions.

MultiRec does case-analysis on the type representation to achieve ad-hoc behaviour, but for polymorphic types, multiple redundant datatypes are required. Thus, it only scores sufficient.

5.7 Ad-hoc definitions for constructors

In all six compared libraries, it is possible to implement *rmWeights*, which is the test case for the ad-hoc definition for constructors.

5.8 Extensibility

Multiplate, like Uniplate, does not support exensible generic functions, nor does MultiRec.

The Generic Deriving library supplies an ad-hoc case of the *gshow* generic function for lists, and thus is marked as extensible.

5.9 Multiple Arguments

It was unclear how to write a generic equality function for Multiplate. Thus, it scores bad.

Generic Deriving and MultiRec support the generic equality function.

5.10 Constructor Names

All libraries except for Multiplate and Uniplate support constructor metadata in their representations.

5.11 Consumers, transformers, and producers.

The Multiplate library does not contain any means for writing producer functions. It is, however, possible to write transformer and consumer functions in it.

The Generic Deriving and MultiRec libraries support functions in all three categories.

5.12 Portability

Multiplate only requires rank-3 polymorphism, but the author has declared that it might be possible to bring it down to rank-2 polymorphism (O'Connor [13]). We have marked it as portable.

Uniplate is able to be defined in Haskell 98, but requires multi-parameter type classes to implement multi-type traversals.

Generic Deriving only requires multi-parameter type classes, but automatic deriving of the *Generic* type is supported by the GHC compiler as of 7.2. Using Generic Deriving

in other compilers (or compiler versions) may be possible, but will result in extra work. MultiRec makes extensive use of GADTs and also uses type families to map datatypes to their corresponding representations. Overall, Uniplate, EMGM and Generic Deriving are the most portable.

Overhead of library use

- *Automatic generation of representations:* While Multiplate currently does not contain any means of automatic generation, it should be trivial to generate the *Plate* record type for a datatype using Template Haskell. But for the moment, it scores bad.

  For Generic Deriving, the *Generic* type can be automatically derived by the GHC compiler as of version 7.2, and the *Generic*1 type could also be generated through Template Haskell, but this is not yet included. It scores sufficient.

  MultiRec is equipped with automatic generation of representations, but generation fails for polymorphic types, so it scores sufficient.

- *Number of structure representations:* The Multiplate library only requires a single representation, namely the *Plate* representation. MultiRec also only requires a single representation.

  Generic Deriving has two representations: *Generic* and *Generic*1. *Generic* is currently automatically derived.

- *Work to instantiate a generic function:* The only boilerplate code that must be written in Multiplate is the *Plate* record type for a datatype and an instance of that datatype for the *Multiplate* class. Generic Deriving uses an implicit representation, and, like EMGM, requires an instance declaration per function per datatype. Due to default class methods, this is hardly extra overhead, so the library scores well on this criterion. MultiRec also scores well, as it allows direct use of a generic function on a type.

- *Work to define a generic function:* The Multiplate library scores well on this criterion, as only a *Plate* record with functions has to be defined.

  In Generic Deriving, the amount of work required to write a generic function is nearly equivalent to EMGM's, and MultiRec also requires a low amount of "encoding work" to be done, so these libraries do well in this criterion.

5.13 Practical aspects

All libraries are available online via Hackage. The Multiplate library is in a very early state and appears to be no longer actively maintained, as the last update is dated nearly a year ago. It has a bad score on this aspect.

On the other hand, Generic Deriving is still actively developed and maintained, and has several pre-defined generic functions available. It receives a good score for this criterion.

The MultiRec library also scores well here, as it is well maintained and documentation is available online.

5.14 Implementation mechanisms

Multiplate uses a novel way of encoding as datatype, namely via a *Plate* record type. It also uses this record type to define update functions on the datatype, as well as collection functions.

Like EMGM, Generic Deriving uses type classes to define its functionalities.

MultiRec uses type families to map types to structure representations. These representations are then associated to individual

datatypes by means of multi-parameter type classes.

5.15 View

In the same spirit as Uniplate, the core of the Multiplate library is the *multiplate* function, which, given a *Plate* of functions over some applicative functor f, creates a new *Plate* that applies those functions to the children of each datatype in the plate.

Generic Deriving uses the often-employed sum-of-products view, and because it abstracts over a single type parameter, it can handle composition in a more effective manner.

MultiRec extends PolyP's fixed point view to represent systems of mutually recursive datatypes.

5.16 Ease of learning and use

Because Multiplate is currently a very small library, there are not many functions available. However, the basic concept of converting a datatype into a *Plate* record type is relatively painless, especially so for mutually recursive datatypes.

Although Generic Deriving uses type classes, it keeps complexity down through automated deriving and default class signatures.

Like SYB, MultiRec suffers from a large number of implementation mechanisms, which may confuse new users as to how they all work together.

6 Conclusions and future work

Both Multiplate and Generic Deriving are very recent libraries, both of them originating at around the end of 2010. Because Multiplate was a side project of its creator, it is not actively maintained at the moment. It could serve instead as an interesting starting point for a library specialized in mutually recursive datatypes, with ease of use and a minimal amount of boilerplate code as its main strengths. Generic Deriving is being maintained by its creator, and already boasts an impressive feature set. Unfortunately, it is limited to a single abstract type parameter, which limits its universe size. The author has mentioned that he will continue to enhance the library to support even more generic functions and datatypes while keeping complexity at a minimum.

MultiRec's feature set is larger than that of Multiplate's, but it is also more complex. Additionally, MultiRec may be significantly slower than Multiplate if we extrapolate Uniplate's performance to Multiplate. Test results of all three libraries performing on a large AST will most certainly prove interesting.

We noticed that neither previous articles (Rodriguez et al. [1] and Rodriguez [2]) did an in-depth performance comparison of all libraries. As the feature sets of generic programming libraries continue to grow, performance may become an important discerning criterion. We leave such a comparison as future work.

Growth of the libraries' feature sets also implies a need for greater granularity of comparison criteria. Hence, for future benchmarking, it may be worthwhile to expand the criteria, as it is certain that generic programming research opportunities are long from exhausted.

7 Acknowledgements

We would like to thank Sean Leather for his guidance in the quickly-evolving world of generic programming and helping us decide which libraries to review. We also thank José Pedro Magalhães for his help and suggestions in implementing the benchmark code for the Generic Deriving library, and Sjoerd Visscher's Multiplate code was a great help for imple-

menting the benchmark code for that library.

References

[1] Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C. d. S. Oliveira. Comparing libraries for generic programming in Haskell. ACM SIGPLAN Haskell Symposium 2008, 2008.

[2] Alexey Rodriguez. *Towards Getting Generic Programming Ready for Prime Time*. PhD thesis, Utrecht University, May 2009. ISBN 978-90-393-5053-9.

[3] Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C. d. S. Oliveira. Comparing libraries for generic programming in Haskell. Technical Report UU-CS-2008-010, Utrecht University, 2008.

[4] James Cheney and Ralf Hinze. Lightweight Implementation of Generics and Dynamics (LIGD). Haskell'02, ACM, 2002. DOI: 10.1145/581690.581698.

[5] Ulf Norell and Patrik Jansson. Polytypic programming in Haskell (PolyLib). *IFL'03*, volume 3145 of LNCS, pages 168-184, 2004.

[6] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. ACM SIGPLAN TLDI'03, 2003.

[7] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. ACM SIGPLAN ICFP'05, pages 204-215, 2005.

[8] Ralf Hinze and Andres Löh. "Scrap your boilerplate" revolutions. *MPC'06*, volume 4014 of LNCS, pages 180-208, 2006.

[9] Bruno C. d. S. Oliveira, Ralf Hinze, and Andres Löh. Extensible and Modular Generics for the Masses. *Trends in Functional Programming*, pages 168–184, 2006.

[10] Stephanie Weirich. RepLib: a library for derivable type classes. Haskell'06, ACM, 2006. DOI: 10.1145/1159842.1159844.

[11] Oleg Kiselyov. Smash your boilerplate without class and typeable. http://article.gmane.org/gmane.comp.lang.haskell.general/14086, 2006.

[12] Neil Mitchell and Colin Runciman. Uniform boilerplate and list processing. Haskell'07, ACM, 2007.

[13] Russell O'Connor. Functor is to Lens as Applicative is to Biplate: Introducing Multiplate. ACM SIGPLAN WGP'11, 2011.

[14] José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. A generic deriving mechanism for Haskell. ACM SIGPLAN Haskell Symposium 2010, 2010.

[15] Alexey Rodriguez, Stefan Holdermans, Andres Löh, and Johan Jeuring. Generic programming with fixed points for mutually recursive datatypes. Technical Report UU-CS-2008-019, Utrecht University, 2008.

	SYB	EMGM	Uniplate	Multiplate	Multirec	Generic Deriving
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	□	●	○	○	●	●
Portability	○	●	●	●	○	●
Overhead of library use						
Automatic generation of representations	●	○	●	○	□	□
Number of structure representations	2	4	1	1	1	2
Work to instantiate a generic function	●	□	●	●	●	●
Work to define a generic function	●	●	●	●	●	●
Practical aspects	●	○	●	○	●	●
Ease of learning and use	○	□	●	●	○	●
	SYB	EMGM	Uniplate	Multiplate	Multirec	Generic Deriving

● - Library supports this criterion.
□ - Library partially supports this criterion or requires extra programming effort.
○ - Library does not support this criterion.

Table 1: Evaluation of generic programming libraries