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

Beerend Lauwers
Augusto Martin
Frank Wijmans

{B.Lauwers, A.PassalaquaMartins, F.S.Wijmans}@students.uu.nl

Utrecht University, The Netherlands

June 25, 2012

Outline

Our main motivation for this project was:
How well do **Generic-Deriving** and **Multiplate** compare to other Generic Programming Libraries?

Libraries

The GP Bench discussed on Rodriguez et al. [2008] already presents comparison of the following libraries:

- **Lightweight Impl. of Generics and Dynamics** (LIGD)
- **Polytypic programming in Haskell** (PolyLib)
- **Scrap your boilerplate** (SYB)
- **SYB, spine view variant** (Spine)
- **SYB, extensible variant using typeclasses** (SYB3)
- **Extensible and Modular Generics for the Masses** (EMGM)
- **RepLib**
- **Smash your boilerplate** (Smash)
- **Uniplate**

Also, there is similar comparison information available about MultiRec in the PhD thesis of Rodriguez [2009].

Project implementation

To answer the question, we added Multiplate and Generic–Deriving to the GP Bench code used by Rodriguez et al. [2008].
We also updated the benchmark code and a subset of the libraries to work with **GHC 7.2.1**.
Namely, we updated the following libraries:

- **Scrap your boilerplate** (SYB)
  - Included with GHC
- **Extensible and Modular Generics for the Masses** (EMGM)
  - Similar to generic-deriving
- **Uniplate**
  - Similar to Multiplate

Multiplate

- Combination of Uniplate and Compos
- Native support for multi-type traversals for mutually recursive datatypes
- Only requires rank-3 polymorphism extension

Multiplate - Example

```haskell
-- 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
```

Define a *Plate* record type:

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

Multiplate - Example

Define an instance of *Multiplate* :

```
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)
```

Boilerplate's done! Example traversal function:

```
-- 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
```

Generic–Deriving

- Uses *DeriveGeneric* and *DefaultSignatures* introduced in GHC 7.2
- Very little boilerplate code required
- Drawback: only abstracts over single type parameter

Generic Deriving - Example

Create a class that acts upon the structure representation:

```haskell
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

instance (GEq' a) => GEq' (M1 i c a) where
  geq' (M1 a) (M1 b) = geq' a b

instance (GEq' a, GEq' b) => GEq' (a :+: b) where
  geq' (L1 a) (L1 b) = geq' a b
  geq' (R1 a) (R1 b) = geq' a b
  geq' _      _      = False

instance (GEq' a, GEq' b) => GEq' (a :*: b) where
  geq' (a1 :*: b1) (a2 :*: b2) = geq' a1 a2 && geq' b1 b2
```

Generic Deriving - Example

Create another class that acts as a mediator between the actual type and 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)
```

Ad-hoc cases:

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

Default cases:

```
instance (GEq a) => GEq (Maybe a)
instance (GEq a) => GEq [a]
```

Evaluation criteria

We evaluated the following criteria:

Types
- Universe
  - *Which datatype classes are supported?*
- Subuniverses
  - *Can generic functions be limited to certain datatypes?*

Expressiveness
- First-class functions
  - *Can a generic function be passed to another generic function?*
- Abstraction over type constructors
  - *Is it possible to define generic map and generic fold functions?*
- Separate compilation
  - *Do you have to recompile the library to extend the universe?*

Evaluation criteria (Cont.)

We evaluated the following criteria:

Expressiveness

- Ad-hoc definitions for constructors/datatypes
  - *Is datatype-specific (non-generic) behaviour possible in a generic function?*
- Extensibility
  - *Can you (non-generically, so ad-hoc required) extend a generic function in a different module?*
- Multiple arguments
  - *Can a generic function have multiple generic arguments?*
- Constructor names
  - *Can you add metadata to a type representation?*
- Consumers, transformers and producers
  - *Can you define functions of types $(a \to T)$, $(a \to a)$, $(a \to b)$ and , $(T \to a)$?*

Evaluation criteria (Cont.)

We evaluated the following criteria:

Usability

- Portability
  - *How many compiler extensions does the library require and how common are they?*
- Overhead of library use
  - *How much code do you have to write to get a type representation / generic function / generic function instance?*
- Practical aspects
  - *How well is the library and its documentation maintained?*
- Ease of learning/use.
  - *How difficult is it to understand the library's functionalities and mechanisms?*

Design Choices

- Implementation mechanisms
  - *How are types represented at runtime?*
- Views
  - *Which views does the library support (e.g. sum-of-products)?*

Multiplate results

(Only criteria with noteworthy results are discussed.)

Types
- **Universe**: Defining nested datatypes was unclear

Expressiveness
- **Separate compilation**: Supported through defining a *Plate* record type
- **Ad-hoc definitions for constructors and datatypes**: Support via *purePlate*
- **Consumers, transformers and producers**: Does not support producer functions

Usability
- **Portability**: Only rank-3 polymorphism required (could be brought down to rank-2 polymorphism)
- **Overhead of library use**
  - **Automatic generation of representations**: Unsupported, but possible through TH
  - **Work to instantiate a generic function**: Little compared to other libraries
- **Practical aspects**: No longer actively maintained

Generic-Deriving results

(Only criteria with noteworthy results are discussed.)

Types
- **Universe**: Abstracts over a single type parameter

Expressiveness
- **First-class functions**: Currently unsupported, research ongoing
- **Abstraction over type constructors**: *crushRight* instance for composition limited functionality, nested trees didn't terminate

Usability
- **Portability**: Only multi-parameter type classes
- **Overhead of library use**
  - **Automatic generation of representations**: *Generic* is derived by compiler, *Generic*1 possible through TH
  - **Work to instantiate a generic function**: Default class methods minimized overhead

Comparison table

	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

Table: Evaluation of generic programming libraries

Conclusions

To conclude what we have done:

- Updated a existing benchmark to compile under **GHC 7.2.1**
- Added code for Generic Deriving and Multiplate to the benchmark
- Compared the results for EMGM, SYB, Uniplate, Multiplate, Generic Deriving and MultiRec

Our conclusions:

- Multiplate could serve as a starting point for specialized library
  - Minimal boilerplate code
  - Relatively easy to use
  - Possibly much faster compared to MultiRec
- Generic–Deriving looks promising
  - Automatic deriving limits boilerplate code and complexity
  - Supports many datatypes
  - Author mentioned further enhancements (Magalhães et al. [2010])

Future work

Performance testing Define criteria for performance testing and test and compare all libraries accordingly

Cabal package Create a cabal package from the benchmark code to make it easier for others to (re)use it

References

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

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

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.