Lightweight Implementation of Generics and Dynamics

Johan Jeuring

Utrecht University

21 May, 2014

Libraries for Generic Programming in Haskell

- Haskell is powerful enough to support most generic programming concepts by means of a library.
- Compared with a language extension (PolyP, Generic Haskell), a library is much easier to ship, support, and maintain.
- Compared with a preprocessing tool like DrIFT or Template Haskell, a library gives you much more support, such as types. Of course, a library might be accompanied by tools.

GP Libraries in Haskell (1)

- Lightweight Implementation of Generics and Dynamics (2002)
- Strafunski (2002)
- Scrap Your Boilerplate (SYB) (2003,2004,2005)
- Prototyping Generic Programming using Template Haskell (2004)
- Generics for the Masses (2004)
- SYB Reloaded, Revolutions (2006)
- Generic programming, now! (2006)
- RepLib (2006)
- Smash Your Boilerplate (2006)
- Almost Compositional Functions (2006)
- Extensible and Modular Generics for the Masses (2006)
- Uniplate (2007)

GP Libraries in Haskell (2)

- Alloy (2008)
- Instant Generics (2009)
- Multirec (2009)
- Regular (2009)
- Pointless Haskell (2010)
- Generic Deriving (2010)
- Multiplate (2010)
- Yoko (2012)
- Shapely-data (2013)
- ...

Essential Concepts In Libraries

There are three essential concepts related to generic programming which we will discuss for each library:

- Run-time type representation
- Generic view on data
- Support for overloading

Equality

Equality is a classic generic programming example.

$$
\begin{array}{llll}
eq_{String} :: String \to String \to Bool \\
eq_{String}\ [\,] & [\,] & = True \\
eq_{String}\ [\,] & \_ & = False \\
eq_{String}\ \_ & [\,] & = False \\
eq_{String}\ (a : as) & (b : bs) & = a \equiv b \wedge eq_{String}\ as\ bs
\end{array}
$$

The algorithm is simple:

- Check whether two values are in the same alternative.
- If not, they are not equal.
- Otherwise, they are equal if all arguments are equal.

LIGD

A Lightweight Implementation of Generics and Dynamics ("LIGD"):

- An approach to embedding generic functions and dynamic values into Haskell 98 augmented with existential types.
- Reflect the type argument onto the value level so we can do ordinary pattern matching on types.
- Developed by James Cheney and Ralf Hinze.
- We describe a variant of LIGD that uses GADTs.
- We do not discuss the "dynamics" feature of the library.

Example: Equality

```
geq :: Rep a →      a →          a →             Bool
geq    RUnit        Unit         Unit          = True
geq    RInt         i            j             = i ≡ j
geq    RChar        c            d             = c ≡ d
geq    (RSum ra _)  (L a1)       (L a2)        = geq ra a1 a2
geq    (RSum _ rb)  (R b1)       (R b2)        = geq rb b1 b2
geq    (RSum _ _)   _            _             = False
geq    (RProd ra rb) (a1 :×: b1) (a2 :×: b2)   = geq ra a1 a2 ∧ geq rb b1 b2
```

This is called a type-indexed function.

Structure types

These types represent some of the basic structural elements of datatypes.

Question

What are equivalent *standard* types?

data Unit = Unit	**data** () = ()
data a :+: b = L a | R b	**data** Either a b = Left a | Right b
data a :×: b = a :×: b	**data** (a, b) = (a, b)

Run-Time Type Representation

LIGD uses a GADT for representing the structure of a type at run-time:

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Rep} :: * \to * \ \textbf{where} \\
\quad \begin{array}{lll}
\mathsf{RUnit} & :: & \mathsf{Rep\ Unit} \\
\mathsf{RInt} & :: & \mathsf{Rep\ Int} \\
\mathsf{RChar} & :: & \mathsf{Rep\ Char} \\
\mathsf{RSum} & :: \mathsf{Rep\ a} \to \mathsf{Rep\ b} \to & \mathsf{Rep\ (a :+: b)} \\
\mathsf{RProd} & :: \mathsf{Rep\ a} \to \mathsf{Rep\ b} \to & \mathsf{Rep\ (a :\times: b)}
\end{array}
\end{array}
$$

`Rep t` is the type representation of `t`.

Question

What purpose do the type indexes serve here?

Going Generic: Universe Extension

The structure of a user-defined datatype `t` is represented by the following `Rep` constructor.

$$\textbf{data}\ \mathrm{Rep} :: * \to *\ \textbf{where}$$
$$\ldots$$
$$\mathrm{RType} :: \mathrm{EP}\ t\ r \to \mathrm{Rep}\ r \to \mathrm{Rep}\ t$$

The type `r` is the structure representation type of `t`, where `r` is a type isomorphic to `t`.

The isomorphism is witnessed by an embedding projection pair:

$$\textbf{data}\ \mathrm{EP}\ t\ r = \mathrm{EP}\ \{\mathrm{from} :: (t \to r), \mathrm{to} :: (r \to t)\}$$

Structure of Lists (1)

The structure representation type of `List`:

```
data List  a = Nil    | Cons a     (List a)
type Listₛ a = Unit :+:      a :×: List a
```

We define two functions to transform between the Haskell type and its LIGD representation.

```
fromList :: List a → Listₛ a
```

```
fromList Nil          = L Unit
fromList (Cons a as) = R (a :×: as)
```

```
toList :: Listₛ a → List a
```

```
toList (L Unit)        = Nil
toList (R (a :×: as)) = Cons a as
```

Structure of Lists (2)

We define an embedding-projection pair to implement the transformation between the Haskell type and its LIGD representation.

$$
\begin{array}{l}
\mathsf{rList} :: \mathsf{Rep}\ \mathsf{a} \rightarrow \mathsf{Rep}\ (\mathsf{List}\ \mathsf{a}) \\
\mathsf{rList}\ \mathsf{r_a} = \mathsf{RType}\ (\mathsf{EP}\ \mathsf{fromList}\ \mathsf{toList}) \\
\qquad\qquad\qquad (\mathsf{RSum}\ \mathsf{RUnit}\ (\mathsf{RProd}\ \mathsf{r_a}\ (\mathsf{rList}\ \mathsf{r_a})))
\end{array}
$$

Generic Equality

geq is turned into a generic function by adding the following case to its definition.

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

...

$$\mathsf{geq}\ (\mathsf{RType}\ \mathsf{ep}\ r_a)\ \mathsf{t1}\ \mathsf{t2} = \mathsf{geq}\ r_a\ (\mathsf{from}\ \mathsf{ep}\ \mathsf{t1})\ (\mathsf{from}\ \mathsf{ep}\ \mathsf{t2})$$

Notes on geq

- geq can be viewed as an implementation of **deriving** Eq in Haskell.
- Similarly, we can define functions that implement the methods of the other classes that can be derived in Haskell: Show, Read, Ord, Enum, and Bounded.

Generic Show

We will implement a generic `show` function to illustrate how a library deals with:

- Constructor names
- Ad-hoc cases for particular datatypes: `"abc"` should not be printed as `Cons 'a' (Cons 'b' (Cons 'c' Nil))`

Constructor Names

To deal with constructor names, we add a constructor to Rep.

```
data Rep :: * → * where
  ...
  RCon :: String → Rep a → Rep a
```

Using this constructor, the representation of List becomes:

```
rList :: Rep a → Rep (List a)
rList r_a = RType (EP fromList toList)
                  (RSum (RCon "Nil" RUnit)
                        (RCon "Cons" (RProd r_a (rList r_a))))
```

Generic Show: First Attempt

```
gshow :: Rep t → t → String
gshow RInt              t        = show t
gshow RChar             t        = show t
gshow RUnit             t        = ""
gshow (RSum r_a _)      (L a)    = gshow r_a a
gshow (RSum _ r_b)      (R b)    = gshow r_b b
gshow (RProd r_a r_b)   (a :×: b) = gshow r_a a ++ " " ++ gshow r_b b
gshow (RType ep r_a)    t        = gshow r_a (from ep t)
gshow (RCon s RUnit)    t        = s
gshow (RCon s r_a)      t        = "(" ++ s ++ " " ++ gshow r_a t ++ ")"
```

Question

What will `gp` look like?

```
gp = gshow (rList RChar) (Cons 'g' (Cons 'p' Nil))
```

Support for Overloading (1)

- This definition shows strings and Haskell's lists using constructor names.
- We want `gshow` to use the standard Haskell `"string"` format instead.
- We want `gshow` to behave in a *specialized*, *non-generic* way for strings.
- Solution: Extend `Rep` with a case for strings:

$$
\begin{array}{l}
\textbf{data}\ \mathrm{Rep} :: * \rightarrow * \ \textbf{where} \\
\quad \ldots \\
\quad \mathrm{RString} :: \mathrm{Rep}\ \mathrm{String}
\end{array}
$$

Support for Overloading (2)

Now we can add the following line to the generic show function to obtain type-specific behavior for the type String.

$$\begin{array}{l} \text{gshow} :: \text{Rep } t \to t \to \text{String} \\ \ldots \\ \text{gshow RString } s = s \end{array}$$

Support for Overloading Is Weak

- We have to adapt the type representation type `Rep` to obtain type-specific behavior in the `gshow` function.
- It is undesirable to adapt a library for the purpose of obtaining special behavior of a single generic function on a particular datatype.
- Unfortunately, this is unavoidable in the LIGD library.
- This implies that many users will construct their own variant of the LIGD library, making both the library and the generic functions written using it less portable and reusable.

Producer Function: Generic Empty

- Both `geq` and `gshow` are generic consumer functions. They take generic values as arguments and produce non-generic results.
- We can also define generic producer functions that do the reverse.
- Simple example: associate an "empty" value with every type.

```
gempty :: Rep a → a
gempty RUnit          = Unit
gempty RInt           = 0
gempty RChar          = '\NUL'
gempty (RSum ra _)    = L (gempty ra)
gempty (RProd ra rb)  = gempty ra :×: gempty rb
gempty (RType ep ra)  = to ep (gempty ra)
gempty (RCon s ra)    = gempty ra
```

Generic Flatten (1)

- Many datatypes can be considered container datatypes: those used to store and structure values.
- Examples are the datatypes `List a`, `Maybe a`, etc.
- A function `flatten` takes a value of a container datatype and returns a list containing all values that it contains.
- `Data.Tree` has `flatten :: Tree a → [a]`.
- The `Prelude` has a related function for lists of lists, `concat :: [[a]] → [a]`.

Generic Flatten (2)

Question

What is the type of the generic flatten function?

Attempt 1:

gflatten :: Rep g $\to$ g a $\to$ [a]

Attempt 2:

gflatten :: Rep (g a) $\to$ g a $\to$ [a]

Attempt 3:

gflatten :: Rep1 ... a $\to$ b $\to$ [a]

A New Representation Type

```
data Rep1 :: (* → *) → * → * where
  RChar1 ::                                Rep1 g Char
  RInt1  ::                                Rep1 g Int
  RUnit1 ::                                Rep1 g Unit
  RSum1  :: Rep1 g a → Rep1 g b →          Rep1 g (a :+: b)
  RProd1 :: Rep1 g a → Rep1 g b →          Rep1 g (a :×: b)
  RCon1  :: String → Rep1 g a →            Rep1 g a
  RType1 :: EP b a → Rep1 g a →            Rep1 g b
  RVar1  :: g a →                          Rep1 g a
```

We added a new constructor `RVar1` for the type of the container elements.

A New List Representation

$$
\begin{aligned}
&\mathsf{rList1} :: \mathsf{Rep1}\ \mathsf{g}\ \mathsf{a} \rightarrow \mathsf{Rep1}\ \mathsf{g}\ (\mathsf{List}\ \mathsf{a})\\
&\mathsf{rList1}\ \mathsf{r_a} = \mathsf{RType1}\ (\mathsf{EP}\ \mathsf{fromList}\ \mathsf{toList})\\
&\qquad\qquad\qquad (\mathsf{RSum1}\ \mathsf{RUnit1}\ (\mathsf{RProd1}\ \mathsf{r_a}\ (\mathsf{rList1}\ \mathsf{r_a})))
\end{aligned}
$$

Defining GFlatten (1)

$$\textbf{newtype}\ \mathrm{GFlatten}\ b\ a = \mathrm{GFlatten}\ \{\mathrm{selFlatten} :: a \rightarrow [b]\}$$

$$
\begin{array}{llll}
\mathrm{gflatten}'\ \mathrm{RUnit1} & \mathrm{Unit} & = [\,] \\
\mathrm{gflatten}'\ (\mathrm{RSum1}\ r_a\ \_) & (\mathrm{L}\ a) & = \mathrm{gflatten}'\ r_a\ a \\
\mathrm{gflatten}'\ (\mathrm{RSum1}\ \_\ r_b) & (\mathrm{R}\ b) & = \mathrm{gflatten}'\ r_b\ b \\
\mathrm{gflatten}'\ (\mathrm{RProd1}\ r_a\ r_b) & (a :\times: b) & = \mathrm{gflatten}'\ r_a\ a +\!\!+ \mathrm{gflatten}'\ r_b\ b \\
\mathrm{gflatten}'\ \mathrm{RInt1} & i & = [\,] \\
\mathrm{gflatten}'\ \mathrm{RChar1} & c & = [\,] \\
\mathrm{gflatten}'\ (\mathrm{RCon1}\ \_\ r_a) & x & = \mathrm{gflatten}'\ r_a\ x \\
\mathrm{gflatten}'\ (\mathrm{RType1}\ ep\ r_a) & x & = \mathrm{gflatten}'\ r_a\ (\mathrm{from}\ ep\ x) \\
\mathrm{gflatten}'\ (\mathrm{RVar1}\ f) & x & = \mathrm{selFlatten}\ f\ x
\end{array}
$$

The type:

$$\mathrm{gflatten}' :: \mathrm{Rep1}\ (\mathrm{GFlatten}\ b)\ a \rightarrow a \rightarrow [b]$$

Defining GFlatten (2)

To simplify the types, we define a type synonym:

type GFlattenRep a b = Rep1 (GFlatten b) a

gflatten′ :: GFlattenRep a b → a → [b]

Defining GFlatten (3)

We need to describe:

1. A `GFlattenRep` for containers
2. What to do with the elements of the container

Recall the representation for lists:

```
rList1 :: Rep1 g a → Rep1 g (List a)
```

We use a function as a representation for containers...

```
gflatten :: (GFlattenRep a a → GFlattenRep b c) → b → [c]
gflatten repContainer = gflatten′ (repContainer repVar)
```

... and we tell it to insert every element into a singleton list.

```
where repVar :: GFlattenRep a a
      repVar = RVar1 (GFlatten (:[]))
```

Defining GFlatten (4)

To obtain an instance of the generic function $\mathrm{gflatten}'$ on List, we write:

$$\mathrm{flattenList} :: \mathrm{List}\ a \rightarrow [a]$$
$$\mathrm{flattenList} = \mathrm{gflatten}\ \mathrm{rList1}$$

Generic Map (1)

The generic map function `gmap`:

- A generalisation of the `map` on lists
- Takes a function of type $a \to b$ and a value of a datatype containing `a`-type elements and applies the function to all the elements in the value.
- `gmap` can be viewed as the implementation of **deriving** for the `Functor` type class in Haskell.
- As with `gflatten`, `gmap` needs to know where the occurrences of the type argument of the datatype appear in a constructor.

Generic Map (2)

- Suppose we use the representation type `Rep1` to implement the generic map function.
- The action on variables then has type $\mathrm{g\ a} \rightarrow \mathrm{Rep1\ g\ a}$, with `g` instantiated to a **newtype** GMap.
- The argument function of `gmap` can only return a value of a type that depends on `a`, or a constant type, but *not* a value of a type `b`.
- To pass a function of type $\mathrm{a} \rightarrow \mathrm{b}$, we need an extra type variable in the `RVar1` constructor.

Another Type Representation

$$
\begin{array}{lll}
\textbf{data}\ \text{Rep2} :: (* \to * \to *) \to * \to * \to *\ \textbf{where} & & \\
\quad \text{RChar2} & :: & \text{Rep2 g Char Char} \\
\quad \text{RInt2} & :: & \text{Rep2 g Int Int} \\
\quad \text{RUnit2} & :: & \text{Rep2 g Unit Unit} \\
\quad \text{RSum2} & :: \text{Rep2 g a b} \to \text{Rep2 g c d} \to & \text{Rep2 g (a :+: c) (b :+: d)} \\
\quad \text{RProd2} & :: \text{Rep2 g a b} \to \text{Rep2 g c d} \to & \text{Rep2 g (a :}\times\text{: c) (b :}\times\text{: d)} \\
\quad \text{RCon2} & :: \text{String} \to \text{Rep2 g a b} \to & \text{Rep2 g a b} \\
\quad \text{RType2} & :: \text{EP b a} \to \text{EP d c} \to \text{Rep2 g a c} \to & \text{Rep2 g b d} \\
\quad \text{RVar2} & :: \text{g a b} \to & \text{Rep2 g a b}
\end{array}
$$

RVar2 represents the two type variables.

Another List Representation

$$
\begin{aligned}
&\mathsf{rList2} :: \mathsf{Rep2}\ \mathsf{g}\ \mathsf{a}\ \mathsf{b} \to \mathsf{Rep2}\ \mathsf{g}\ (\mathsf{List}\ \mathsf{a})\ (\mathsf{List}\ \mathsf{b}) \\
&\mathsf{rList2}\ \mathsf{r_a} = \mathsf{RType2}\ (\mathsf{EP}\ \mathsf{fromList}\ \mathsf{toList}) \\
&\qquad\qquad\qquad\quad (\mathsf{EP}\ \mathsf{fromList}\ \mathsf{toList}) \\
&\qquad\qquad\qquad\quad (\mathsf{RSum2}\ \mathsf{RUnit2}\ (\mathsf{RProd2}\ \mathsf{r_a}\ (\mathsf{rList2}\ \mathsf{r_a})))
\end{aligned}
$$

Defining GMap (1)

$$\textbf{newtype}\ \mathsf{GMap\ a\ b} = \mathsf{GMap}\ \{\mathsf{selMap} :: \mathsf{a} \to \mathsf{b}\}$$
$$\textbf{type}\ \mathsf{GMapRep\ a\ b} = \mathsf{Rep2\ GMap\ a\ b}$$

$$\mathsf{gmap}' :: \mathsf{GMapRep\ a\ b} \to \mathsf{a} \to \mathsf{b}$$

$$\begin{array}{lll}
\mathsf{gmap}'\ \mathsf{RUnit2} & \mathsf{Unit} & = \mathsf{Unit} \\
\mathsf{gmap}'\ (\mathsf{RSum2}\ \mathsf{r_a}\ _) & (\mathsf{L\ a}) & = \mathsf{L}\ (\mathsf{gmap}'\ \mathsf{r_a}\ \mathsf{a}) \\
\mathsf{gmap}'\ (\mathsf{RSum2}\ _\ \mathsf{r_b}) & (\mathsf{R\ b}) & = \mathsf{R}\ (\mathsf{gmap}'\ \mathsf{r_b}\ \mathsf{b}) \\
\mathsf{gmap}'\ (\mathsf{RProd2}\ \mathsf{r_a}\ \mathsf{r_b}) & (\mathsf{a} :\times: \mathsf{b}) & = \mathsf{gmap}'\ \mathsf{r_a}\ \mathsf{a} :\times: \mathsf{gmap}'\ \mathsf{r_b}\ \mathsf{b} \\
\mathsf{gmap}'\ \mathsf{RInt2} & \mathsf{i} & = \mathsf{i} \\
\mathsf{gmap}'\ \mathsf{RChar2} & \mathsf{c} & = \mathsf{c} \\
\mathsf{gmap}'\ (\mathsf{RCon2}\ _\ \mathsf{r_a}) & \mathsf{x} & = \mathsf{gmap}'\ \mathsf{r_a}\ \mathsf{x} \\
\mathsf{gmap}'\ (\mathsf{RType2}\ \mathsf{ep_1}\ \mathsf{ep_2}\ \mathsf{r_a}) & \mathsf{x} & = (\mathsf{to}\ \mathsf{ep_2} \circ \mathsf{gmap}'\ \mathsf{r_a} \circ \mathsf{from}\ \mathsf{ep_1})\ \mathsf{x} \\
\mathsf{gmap}'\ (\mathsf{RVar2}\ \mathsf{f}) & \mathsf{x} & = \mathsf{selMap}\ \mathsf{f}\ \mathsf{x}
\end{array}$$

Defining GMap (2)

gmap is defined similarly to gflatten.

$$
\begin{aligned}
&\mathsf{gmap} :: (\mathsf{GMapRep}\ a\ b \to \mathsf{GMapRep}\ c\ d) \to (a \to b) \to c \to d\\
&\mathsf{gmap}\ \mathsf{repContainer}\ f = \mathsf{gmap}'\ (\mathsf{repContainer}\ \mathsf{repVar})\\
&\quad \mathbf{where}\ \mathsf{repVar} = \mathsf{RVar2}\ (\mathsf{GMap}\ f)
\end{aligned}
$$

Unlike gflatten, we use a parameter for the function f on the elements.

Defining GMap (3)

Recall the List representation:

```
rList2 :: Rep2 g a b → Rep2 g (List a) (List b)
```

with the instance:

```
rList2 :: GMapRep a b → GMapRep (List a) (List b)
```

To obtain an instance of gmap on the datatype List a we write:

```
gmapList :: (a → b) → List a → List b
gmapList = gmap rList2
```

Notes on GMap

Question

Do we need to introduce a new representation type for every generic function we define?

- For all practical purposes, it appears that we need at most 3 type variables.
- We could use the datatype `Rep3` for all of our generic functions, but that would introduce many type variables which are never used.

Conclusions

We have covered:

- A library for generic programming in Haskell: LIGD
- The important aspects, strengths, and weaknesses of this library.

LIGD:

- Is just one of many libraries for generic programming.
- Uses one of a number of views on datatype structure.
- May be one of the easiest to implement and understand.
- May be one of the least modular and extensible.