Extensible and Modular Generics for the Masses

Sean Leather, Johan Jeuring

Utrecht University

20 September, 2012
Previously...

You learned about:

- Datatypes and Kinds
- Lightweight Implementation of Generics and Dynamics (LIGD)
This time...

We’re going to talk about the library Extensible and Modular Generics for the Masses (EMGM).

- Define an example generic function
- Introduce the run-time type representation
- Add datatype-generic support
- Demonstrate support for ad-hoc cases
- Change the representation to be extensible and modular
- Define other generic functions
  - Producer functions
  - Higher-kindred datatypes
  - Abstracting over more than one type
Defining an Example: Equality (1)

Defining a generic function in EMGM involves several steps. First, let’s decide what the “ideal” type signature should look like.

\[ \text{geq :: } a \rightarrow a \rightarrow \text{Bool} \]
Next, we need to define a `newtype` for the generic function.

```haskell
newtype Geq a = Geq { selEq :: a → a → Bool }
```

This is similar the use of `newtype` in LIGD.
Defining an Example: Equality (3)

Now, we implement the structural components of our generic function.

\[
\begin{align*}
\text{geq}_{\text{unit}}: \text{Unit} \rightarrow \text{Unit} & = \text{True} \\
\text{geq}_{\text{int}}: \text{i} \rightarrow \text{j} & = \text{i} \equiv \text{j} \\
\text{geq}_{\text{char}}: \text{c} \rightarrow \text{d} & = \text{c} \equiv \text{d} \\
\text{geq}_{\text{sum}}: \text{r} \rightarrow \text{L a}_1 \rightarrow \text{L a}_2 & = \text{selEq r a}_1 \text{ a}_2 \\
\text{geq}_{\text{sum}}: \text{r} \rightarrow \text{R b}_1 \rightarrow \text{R b}_2 & = \text{selEq r b}_1 \text{ b}_2 \\
\text{geq}_{\text{sum}}: \text{r} \rightarrow \text{a} \rightarrow _ & = \text{False} \\
\text{geq}_{\text{prod}}: \text{r} \rightarrow \text{a} : \times : \text{b}_1 \rightarrow \text{a} : \times : \text{b}_2 & = \text{selEq r a}_1 \text{ a}_2 \land \\
& \quad \text{selEq r b}_1 \text{ b}_2
\end{align*}
\]
Defining an Example: Equality (4)

That should look familiar. Here’s \texttt{geq} in LIGD.

\[
\begin{align*}
\text{geq (RUnit )} & \quad \text{Unit} \quad \text{Unit} \quad = \quad \text{True} \\
\text{geq (RInt )} & \quad \text{i} \quad \text{j} \quad = \quad \text{i} \equiv \text{j} \\
\text{geq (RChar)} & \quad \text{c} \quad \text{d} \quad = \quad \text{c} \equiv \text{d} \\
\text{geq (RSum r a r b)} & \quad \text{(L a}_1\text{)} \quad \text{(L a}_2\text{)} \quad = \quad \text{geq r}_a \text{ a}_1 \text{ a}_2 \\
\text{geq (RSum r a r b)} & \quad \text{(R b}_1\text{)} \quad \text{(R b}_2\text{)} \quad = \quad \text{geq r}_b \text{ b}_1 \text{ b}_2 \\
\text{geq (RSum r a r b)} & \quad \_ \quad \_ \quad = \quad \text{False} \\
\text{geq (RProd r a r b)} & \quad \text{(a}_1 \times \text{ b}_1\text{)} \quad \text{(a}_2 \times \text{ b}_2\text{)} \quad = \quad \text{geq r}_a \text{ a}_1 \text{ a}_2 \land \\
& \quad \quad \quad \quad \quad \text{geq r}_b \text{ b}_1 \text{ b}_2
\end{align*}
\]
Defining an Example: Equality (5)

Next, we create an instance of the `Generic` type class using our generic functions.

```haskell
instance Generic Geq where
  runit      = Geq geq_unit
  rint       = Geq geq_int
  rchar      = Geq geq_char
  rsum r a r b = Geq (geq_sum r a r b)
  rprod r a r b = Geq (geq_prod r a r b)
```

How does this tie the recursive knot with `selEq`?
Defining an Example: Equality (6)

At this point, our generic function is (partially) usable.

\[
\text{selEq (rprod rchar rint) ('Q' :×: 42) ('Q' :×: 42) ≡ True}
\]

But that’s not good enough...
We want to hide the type representation argument...

\[
\text{geq :: (Rep a) } \Rightarrow \ a \rightarrow a \rightarrow \text{Bool} \\
\text{geq = selEq rep}
\]

... to make it \textbf{implicit}:

\[
\text{geq ('Q' :\times: 42) ('Q' :\times: 42) } \equiv \text{True}
\]
The Mechanics: Run-time Type Representation (1)

Now, let’s talk about the run-time type representation machinery that allows us to define functions such as \( \text{geq} \).

First, you should recall these structure representation types. They are the same as those in LIGD.

```haskell
data Unit = Unit
data a :+: b = L a | R b
data a :×: b = a :×: b
```
The Mechanics: Run-time Type Representation (2)

The `Generic` class has a method for each representation type.

```haskell
class Generic g where
  runit :: g Unit
  rint  :: g Int
  rchar :: g Char
  rsum :: g a -> g b -> g (a :+: b)
  rprod :: g a -> g b -> g (a :×: b)
```

An instance of `Generic` defines a type-indexed function.
To make the representation value implicit, we use the `Rep` class.

```haskell
class Rep a where
    rep :: (Generic g) ⇒ g a
```

This allows us to substitute `rep` for any instance of `Generic`.
The instances of `Rep` include all representable types. We start with the universe of base and structure types.

```haskell
instance Rep Unit where
  rep = runit

instance Rep Int where
  rep = rint

instance Rep Char where
  rep = rchar

instance (Rep a, Rep b) ⇒ Rep (a :+: b) where
  rep = rsum rep rep

instance (Rep a, Rep b) ⇒ Rep (a :×: b) where
  rep = rprod rep rep
```
Expanding the Universe (1)

To make our functions truly generic, we need to expand our universe to include user-defined datatypes.

```haskell
class Generic g where
    ... 
    rtype :: EP b a → g a → g b
```

Recall the analogous LIGD constructor:

```haskell
RType :: EP b a → Rep a → Rep b
```

Recall the embedding-projection pair datatype.

```haskell
data EP d r = EP {from :: (d → r), to :: (r → d)}
```
Expanding the Universe (2)

The representation for `List` is:

\[
\text{rList :: (Generic g) } \Rightarrow \text{ g a } \rightarrow \text{ g (List a)} \\
\text{rList } r_a = \text{ rtype (EP fromList toList)} \\
\quad (\text{rsum runit (rprod } r_a \text{ (rList } r_a))
\]

Again, notice the similarity to LIGD:

\[
\text{rList } r_a = \text{ RType (EP fromList toList)} \\
\quad (\text{RSum RUnit (RProd } r_a \text{ (rList } r_a))
\]
To add \texttt{rList} as another implicit representation, we define an instance of \texttt{Rep} for \texttt{List}.

\begin{verbatim}
instance (Rep a) \Rightarrow Rep (List a) where
  rep = rList rep
\end{verbatim}
Expanding the Universe (4)

To make `geq` a generic function that supports user-defined datatypes, we add another case.

```haskell
geq_{\text{type}} \text{ep } r_a\ a_1\ a_2 = \text{selEq } r_a\ (\text{from ep } a_1)\ (\text{from ep } a_2)
```

```haskell
\text{instance } \text{Generic Geq where }

... 

\text{rtype ep } r_a = \text{Geq } (\text{geq}_{\text{type}}\ \text{ep } r_a)
```
Let’s write a generic `show` function. Think: `deriving Show`.

```haskell
gshow :: a → String
```

But we don’t have access to the constructor names. For that, we can add another case to our generic function signature.

```haskell
class Generic g where
  ...
  rcon :: String → g a → g a
```

`rcon` is a wrapper around other structure types.
We then add \texttt{rcon} to wrap each alternative in \texttt{rsum} with the name of the constructor.

\begin{verbatim}
\begin{verbatim}
\texttt{rList :: (Generic g) \Rightarrow g a \rightarrow g (List a)}
\texttt{rList r} = \texttt{rtype (EP fromList toList)}
\texttt{(rsum (rcon "Nil" runit)}
\texttt{(rcon "Cons" (rprod r a (rList r a))))}
\end{verbatim}
\end{verbatim}
Now, we can implement the cases of `gshow`. Most of the entries are exactly as you would expect (see lecture notes).

```haskell
newtype Gshow a = Gshow { selShow :: a -> String }
gshow_unit Unit = ""
...
gshow_type ep r a a = selShow r a (from ep a)
gshow_con s r a a = "(" ++ s ++
                   " " ++ selShow r a ++
                   ")"

instance Generic Gshow where
    runit = Gshow gshow_unit
    ...
```
The final generic show function looks like this:

\[
gshow :: (\text{Rep } a) \Rightarrow a \rightarrow \text{String}
\]
\[
gshow = \text{selShow } \text{rep}
\]

And it works like this:

\[
gshow (\text{Cons 4 } (\text{Cons 2 } \text{Nil})) \equiv "(\text{Cons 4 } (\text{Cons 2 } (\text{Nil }))")"
\]

But the output is ugly! We need to fix it...
We want something specific for \texttt{List}. Instead of the general \texttt{rList} representation based on \texttt{rtype}, we can add a special list case to \texttt{Generic}.

\begin{verbatim}
class Generic g where

  ...

  list :: g a → g (List a)

\end{verbatim}

We also need to register \texttt{list} as a representable type.

\begin{verbatim}
instance (Rep a) ⇒ Rep (List a) where

  rep = list rep

\end{verbatim}
We extend `gshow` for lists...

\[
\begin{align*}
gshow_{\text{list}} \, ra \, \text{Nil} &= \ "\ [] \" \\
gshow_{\text{list}} \, ra \, (\text{Cons a as}) &= \text{selShow} \, ra \, a \, \# "\ : \" \ \# \\
& \quad \text{selShow} \, (\text{list} \, ra) \, \text{as}
\end{align*}
\]

\textbf{instance} Generic Gshow \textbf{where}

\[
\begin{align*}
& \quad \ldots \\
& \quad \text{list} \, ra = \text{Gshow} \, (gshow_{\text{list}} \, ra)
\end{align*}
\]

\ldots arriving at a more concise output:

\[
gshow \, (\text{Cons} \, 4 \, (\text{Cons} \, 2 \, \text{Nil})) \equiv \ "4:2:[\]"
\]
Becoming Modular and Extensible (1)

Modifying the *Generic* class for every type is bad. The process is not modular and reduces the reusability of a library. (Just like LIGD.) We can change this with *EMGM*. Let’s try a hierarchy of classes.

```haskell
class (Generic g) ⇒ GenericList g where
  rlist :: g a → g (List a)
  rlist = rList

instance GenericList Gshow where
  rlist r_a = Gshow (gshow_list r_a)
```

We can now use `selShow`.

```
selShow (rlist rint) (Cons 2 Nil) ≡ "2: []"
```
Becoming Modular and Extensible (2)

What happens when we define the following instance?

```haskell
instance (Rep a) ⇒ Rep (List a) where
  rep = rlist rep
```

GHC complains:

```
Could not deduce (GenericList g)
  from the context (Rep (List a), Rep a, Generic g)
  arising from a use of `rlist` at ...
```

Possible fix:
```
add (GenericList g) to the context of
  the type signature for `rep` ...
```
The current type signature for rep:

\[ \text{rep :: (Generic } g, \text{ Rep } a) \Rightarrow g \, a \]

What happens if we follow GHC’s advise?

Possible fix:

add \((\text{GenericList } g)\) to the context of the type signature for ‘rep’ ...
Instead, let’s not assume that \( g \) is always an instance of \( \text{Generic} \). We abstract over the type constructor in \( \text{Rep} \).

```haskell
class Rep g a where
    rep :: g a
```
We rewrite the instances from before:

\[
\text{instance } (\text{Generic } g) \Rightarrow \text{Rep } g \text{ Unit where }
\]
\[
\text{rep } = \text{runit}
\]

\[
\text{instance } (\text{Generic } g, \text{Rep } g \text{ a, Rep } g \text{ b}) \Rightarrow \text{Rep } g \text{ (a :+: b) where rep } = \text{rsum rep rep}
\]

... 

And use \text{GenericList} in the context for the \text{List} instance instead of \text{Generic}.

\[
\text{instance } (\text{GenericList } g, \text{Rep } g \text{ a}) \Rightarrow \text{Rep } g \text{ (List a) where }
\]
\[
\text{rep } = \text{rlist rep}
\]
Lastly, we rewrite the generic show...

gshow :: (Rep Gshow a) ⇒ a → String
gshow = selShow rep

... by explicitly filling in the newtype Gshow for the \( g \) parameter. Let’s move on to some other examples. Some of them challenge the approaches we’ve shown so far.
Here is a simple generic producer function in its entirety.

```haskell
newtype Gempty a = Gempty { selEmpty :: a }

instance Generic Gempty where
  runit = Gempty Unit
  rint = Gempty 0
  rchar = Gempty '\NUL'
  rsum r a r b = Gempty (L (selEmpty r a))
  rprod r a r b = Gempty (selEmpty r a :×: selEmpty r b)
  rtype ep r a = Gempty (to ep (selEmpty r a))
  rcon s r a = Gempty (selEmpty r a)

gempty :: (Rep Gempty a) ⇒ a

gempty = selEmpty rep
```
We have dealt with types of kind \( \ast \) up to this point. How do we deal with kind \( \ast \to \ast \)? These include the “container” datatypes: \( \text{List} \ a \), \( \text{Tree} \ a \), etc.
We use a generic \texttt{crush} function as an example.
Higher Kinds: Crush (2)

Recall the standard \texttt{foldr} function:

\[
\texttt{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b
\]

It generalizes to \texttt{crushr}:

\[
\texttt{crushr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow f\ a \rightarrow b
\]

- \((a \rightarrow b \rightarrow b)\) — A “combining” function
- \(b\) — A “zero” value
- \(f\ a\) — A container
Higher Kinds: Crush (3)

The type-indexed function is straightforward.

```haskell
newtype Crush b a = Crush { selCrush :: a → b → b }

crushr_unit _ e = e

... 

crushr_sum r_a r_b (L a) e = selCrush r_a a e

crushr_sum r_a r_b (R b) e = selCrush r_b b e

crushr_prod r_a r_b (a :×: b) e = selCrush r_a a (selCrush r_b b e)

crushr_type ep r_a a e = selCrush r_a (from ep a) e

instance Generic (Crush b) where
  runit = Crush crushr_unit
  ...
```
We have `selCrush`, so how do we write `crushr`? Recall `rep` again.

```
class Rep g a where
    rep :: g a
```

The type variable `a` has kind `∗`. We want to abstract over container types of the form `f a` where `f` has kind `∗ → ∗`.

Key: *The type of the representation function reflects the kind of the represented type.*

```
class FRep g f where
    frep :: g a → g (f a)
```
Translating an instance from $\text{Rep}$

\begin{verbatim}
instance (Generic g, Rep g a) ⇒ Rep g (List a) where
  rep = rList rep
\end{verbatim}

to $\text{FRep}$

\begin{verbatim}
instance (Generic g) ⇒ FRep g List where
  frep = rList
\end{verbatim}

requires removing all references to the type variables for the container’s element.
How do we come up with ...

\[
\text{crushr :: (...) \Rightarrow (a \to b \to b) \to b \to f\ a \to b}
\]

... given this, ...

\[
\text{selCrush :: Crush b a \to a \to b \to b}
\]

... this, ...

\[
\text{frep :: (FRep g f) \Rightarrow g\ a \to g\ (f\ a)}
\]

... and this?

\[
\text{Crush :: (a \to b \to b) \to Crush b a}
\]
Let’s assemble this type jigsaw puzzle:

selCrush :: Crush b a → a → b → b
frep :: (FRep g f) ⇒ g a → g (f a)
Crush :: (a → b → b) → Crush b a

First, frep ◦ Crush :

frep ◦ Crush ::
(FRep (Crush b) f) ⇒ (a → b → b) → Crush b (f a)
Higher Kinds: Crush (8)

Let’s assemble this type jigsaw puzzle:

\[
\text{selCrush} :: \text{Crush } b \ a \to a \to b \to b
\]

\[
\text{frep} :: (\text{FRep } g \ f) \Rightarrow g \ a \to g \ (f \ a)
\]

\[
\text{Crush} :: (a \to b \to b) \to \text{Crush } b \ a
\]

Then, \(\text{selCrush} \circ \text{frep} \circ \text{Crush} ::\)

\[
\text{selCrush} \circ \text{frep} \circ \text{Crush} :: (\text{FRep } (\text{Crush } b) \ f) \Rightarrow (a \to b \to b) \to f \ a \to b \to b
\]
Finally, we can define \( \text{crushr} \).

\[
\text{crushr} :: (\text{FRep} (\text{Crush} b) f) \Rightarrow (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow f a \rightarrow b \\
\text{crushr} f z x = \text{selCrush} (\text{frep} (\text{Crush} f)) \times z
\]

And we can use it, too.

\[
\text{gflatten} :: (\text{FRep} (\text{Crush} [a]) f) \Rightarrow f a \rightarrow [a] \\
\text{gflatten} = \text{crushr} (:) [] \\
\text{gflatten} (\text{Cons} 4 (\text{Cons} 2 \text{Nil})) \equiv [4, 2]
\]
The standard `map` function is a very handy function. We often want to apply a function to all elements in a list.

\[
\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
\]

Why don’t we generalise this to other datatypes as we generalised `foldr` to `crushr`?

\[
\text{gmap} :: (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b
\]
The type-indexed function.

\[
\textbf{newtype} \ Gmap \ a \ b = Gmap \{ \text{selMap} :: a \to b \} \\
\text{gmap}_{\text{unit}} \quad \times \quad = \times \\
\ldots \\
\text{gmap}_{\text{sum}} \quad r_a \ r_b \ (L \ a) \quad = \ L \ (\text{selMap} \ r_a \ a) \\
\text{gmap}_{\text{sum}} \quad r_a \ r_b \ (R \ b) \quad = \ R \ (\text{selMap} \ r_b \ b) \\
\text{gmap}_{\text{prod}} \quad r_a \ r_b \ (a \times b) \quad = \ \text{selMap} \ r_a \ a \times \text{selMap} \ r_b \ b \\
\text{gmap}_{\text{type}} \ ep_1 \ ep_2 \ r_a \ a \quad = \ (to \ ep_2 \circ \text{selMap} \ r_a \circ \text{from} \ ep_1) \ a
\]
Higher Abstraction: Map (3)

gmap is both a generic consumer and generic producer, so we must abstract over two types, input and output.

```haskell
class Generic2 g where
  runit2 :: g Unit Unit
  rint2  :: g Int Int
  rchar2 :: g Char Char
  rsum2 :: g a1 a2 → g b1 b2 → g (a1 :+: b1) (a2 :+: b2)
  rprod2 :: g a1 a2 → g b1 b2 → g (a1 :×: b1) (a2 :×: b2)
  rtype2 :: EP a2 a1 → EP b2 b1 → g a1 b1 → g a2 b2
```
We define our instance of `Generic2`.

```haskell
instance Generic2 Gmap where
  runit2        = Gmap gmap_unit
...
  rtype2 ep1 ep2 ra = Gmap (gmap_type ep1 ep2 ra)
```

Since we have this new `rtype2` method (rather than `rtype`), we need to redefine our list representation.

```haskell
rList2 :: (Generic2 g) ⇒ g a b → g (List a) (List b)
rList2 ra = rtype2 (EP fromList toList)
            (EP fromList toList)
            (rsum2 runit2 (rprod2 ra (rList2 ra)))
```
We can immediately use the list representation to implement the standard map on List containers.

```haskell
mapList :: (a -> b) -> List a -> List b
mapList f = selMap (rList2 (Gmap f))
```

But our ultimate goal (as always) is to generalise...
Higher Abstraction: Map (6)

We can’t use the `FRep` class. Why?

```haskell
class FRep g f where
  frep :: g a → g (f a)
```

We must extend it to support the higher-kindred `g (* → * → *)`, i.e. abstraction over two types.

```haskell
class FRep2 g f where
  frep2 :: g a b → g (f a) (f b)

instance (Generic2 g) ⇒ FRep2 g List where
  frep2 = rList2
```

Our instance for `List` is similar to the instance for `FRep`.
We can now define \texttt{gmap} with a method similar to how we defined \texttt{crushr}.

\[
gmap :: (\text{FRep2 \ Gmap \ f}) \Rightarrow (a \rightarrow b) \rightarrow f \ a \rightarrow f \ b
\]

\[
gmap \ f = \text{selMap \ (frep2 \ (Gmap \ f))}
\]
Conclusions

We have covered the following concepts of using generic functions in EMGM:

- Equality: basic
- Show: ad-hoc, extensible, and modular
- Empty: producer
- Crush: higher-kinded datatypes
- Map: abstraction over more than one type