Extensible and Modular Generics for the Masses

Sean Leather

Utrecht University

26 September, 2011

Previously...

You learned about:

- Datatypes and Kinds
- Lightweight Implementation of Generics and Dynamics (LIGD)
- Scrap Your Boilerplace (SYB)

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-kinded 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.

$$\mathsf{geq} :: \mathsf{a} \rightarrow \mathsf{a} \rightarrow \mathsf{Bool}$$

Defining an Example: Equality (2)

Next, we need to define a **newtype** for the generic function.

$\textbf{newtype}\ \mathsf{Geq}\ \mathsf{a} = \mathsf{Geq}\ \{\mathsf{selEq} :: \mathsf{a} \rightarrow \mathsf{a} \rightarrow \mathsf{Bool}\}$

This is similar the use of **newtype** in LIGD.

Defining an Example: Equality (3)

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

$$
\begin{array}{llll}
\mathsf{geq\_unit} & \mathsf{Unit} & \mathsf{Unit} & = \mathsf{True} \\
\mathsf{geq\_int} & \mathsf{i} & \mathsf{j} & = \mathsf{i} \equiv \mathsf{j} \\
\mathsf{geq\_char} & \mathsf{c} & \mathsf{d} & = \mathsf{c} \equiv \mathsf{d} \\
\mathsf{geq\_plus}\ \mathsf{r_a}\ \mathsf{r_b} & (\mathsf{L}\ \mathsf{a}_1) & (\mathsf{L}\ \mathsf{a}_2) & = \mathsf{selEq}\ \mathsf{r_a}\ \mathsf{a}_1\ \mathsf{a}_2 \\
\mathsf{geq\_plus}\ \mathsf{r_a}\ \mathsf{r_b} & (\mathsf{R}\ \mathsf{b}_1) & (\mathsf{R}\ \mathsf{b}_2) & = \mathsf{selEq}\ \mathsf{r_b}\ \mathsf{b}_1\ \mathsf{b}_2 \\
\mathsf{geq\_plus}\ \mathsf{r_a}\ \mathsf{r_b} & \_ & \_ & = \mathsf{False} \\
\mathsf{geq\_prod}\ \mathsf{r_a}\ \mathsf{r_b} & (\mathsf{a}_1 :\times: \mathsf{b}_1) & (\mathsf{a}_2 :\times: \mathsf{b}_2) & = \mathsf{selEq}\ \mathsf{r_a}\ \mathsf{a}_1\ \mathsf{a}_2 \wedge \\
 & & & \quad \mathsf{selEq}\ \mathsf{r_b}\ \mathsf{b}_1\ \mathsf{b}_2
\end{array}
$$

Defining an Example: Equality (4)

That should look familiar. Here's `geq` in LIGD.

$$
\begin{array}{lllll}
\mathsf{geq}\ (\mathsf{RUnit}) & \mathsf{Unit} & \mathsf{Unit} & = & \mathsf{True} \\
\mathsf{geq}\ (\mathsf{RInt}\ \ ) & \mathsf{i} & \mathsf{j} & = & \mathsf{i} \equiv \mathsf{j} \\
\mathsf{geq}\ (\mathsf{RChar}) & \mathsf{c} & \mathsf{d} & = & \mathsf{c} \equiv \mathsf{d} \\
\mathsf{geq}\ (\mathsf{RSum}\ \mathsf{r_a}\ \mathsf{r_b}) & (\mathsf{L}\ \mathsf{a}_1) & (\mathsf{L}\ \mathsf{a}_2) & = & \mathsf{geq}\ \mathsf{r_a}\ \mathsf{a}_1\ \mathsf{a}_2 \\
\mathsf{geq}\ (\mathsf{RSum}\ \mathsf{r_a}\ \mathsf{r_b}) & (\mathsf{R}\ \mathsf{b}_1) & (\mathsf{R}\ \mathsf{b}_2) & = & \mathsf{geq}\ \mathsf{r_b}\ \mathsf{b}_1\ \mathsf{b}_2 \\
\mathsf{geq}\ (\mathsf{RSum}\ \mathsf{r_a}\ \mathsf{r_b}) & \_ & \_ & = & \mathsf{False} \\
\mathsf{geq}\ (\mathsf{RProd}\ \mathsf{r_a}\ \mathsf{r_b}) & (\mathsf{a}_1 :\times: \mathsf{b}_1) & (\mathsf{a}_2 :\times: \mathsf{b}_2) & = & \mathsf{geq}\ \mathsf{r_a}\ \mathsf{a}_1\ \mathsf{a}_2 \wedge \\
 & & & & \mathsf{geq}\ \mathsf{r_b}\ \mathsf{b}_1\ \mathsf{b}_2
\end{array}
$$

Defining an Example: Equality (5)

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

```
instance Generic Geq where
  runit         = Geq geq_unit
  rint          = Geq geq_int
  rchar         = Geq geq_char
  rsum  r_a r_b = Geq (geq_plus  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.

selEq (rprod rchar rint) ('Q' $:\times:$ 42) ('Q' $:\times:$ 42) $\equiv$ True

But that's not good enough...

Defining an Example: Equality (7)

We want to hide the **type representation** argument...

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

... to make it **implicit**:

$\text{geq}\ (\texttt{'Q'} :\times: 42)\ (\texttt{'Q'} :\times: 42) \equiv \text{True}$

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 `geq`.

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

$$
\begin{aligned}
&\mathbf{data}\ \mathsf{Unit} &&= \mathsf{Unit} \\
&\mathbf{data}\ \mathsf{a} :+: \mathsf{b} &&= \mathsf{L}\ \mathsf{a} \mid \mathsf{R}\ \mathsf{b} \\
&\mathbf{data}\ \mathsf{a} :\times: \mathsf{b} &&= \mathsf{a} :\times: \mathsf{b}
\end{aligned}
$$

The Mechanics: Run-time Type Representation (2)

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

```
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.

The Mechanics: Run-time Type Representation (3)

To make the representation value implicit, we use the `Rep` class.

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

This allows us to substitute `rep` for any instance of `Generic`.

The Mechanics: Run-time Type Representation (4)

The instances of `Rep` include all representable types. We start with the universe of base and structure types.

instance Rep Unit **where**
  rep = runit

instance Rep Int **where**
  rep = rint

instance Rep Char **where**
  rep = rchar

instance (Rep a, Rep b) $\Rightarrow$ Rep (a :+: b) **where**
  rep = rsum rep rep

instance (Rep a, Rep b) $\Rightarrow$ 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.

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

Recall the analogous LIGD constructor:

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

Recall the embedding-projection pair datatype.

```
data EP d r = EP {from :: (d → r), to :: (r → d)}
```

Expanding the Universe (2)

The representation for List is:

```
rList :: (Generic g) ⇒ g a → g (List a)
rList r_a = rtype (EP fromList toList)
                  (rsum runit (rprod r_a (rList r_a)))
```

Again, notice the similarity to LIGD:

```
rList r_a = RType (EP fromList toList)
                  (RSum RUnit (RProd r_a (rList r_a)))
```

Expanding the Universe (3)

To add `rList` as another implicit representation, we define an instance of `Rep` for `List`.

```
instance (Rep a) ⇒ Rep (List a) where
  rep = rList rep
```

Expanding the Universe (4)

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

$$\mathsf{geq\_dt\ ep\ r_a\ a_1\ a_2 = selEq\ r_a\ (from\ ep\ a_1)\ (from\ ep\ a_2)}$$

instance Generic Geq **where**

...

$\mathsf{rtype\ ep\ r_a = Geq\ (geq\_dt\ ep\ r_a)}$

Overloaded and Ad-hoc (1)

Let's write a generic `show` function. Think: **deriving** Show.

```
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.

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

`rcon` is a wrapper around other structure types.

Overloaded and Ad-hoc (2)

We then add `rcon` to wrap each alternative in `rsum` with the name of the constructor.

```
rList :: (Generic g) ⇒ g a → g (List a)
rList r_a = rtype (EP fromList toList)
                  (rsum (rcon "Nil" runit)
                        (rcon "Cons" (rprod r_a (rList r_a))))
```

Overloaded and Ad-hoc (3)

Now, we can implement the cases of `gshow`. Most of the entries are exactly as you would expect (see lecture notes).

```
newtype Gshow a = Gshow {selShow :: a → String}
gshow_unit            Unit = ""
 ...
gshow_dt       ep r_a a     = selShow r_a (from ep a)
gshow_constr s    r_a a     = "(" ++ s ++
                              " " ++ selShow r_a a ++
                              ")"
instance Generic Gshow where
   runit = Gshow gshow_unit
     ...
```

Overloaded and Ad-hoc (4)

The final generic show function looks like this:

$$\mathsf{gshow} :: (\mathsf{Rep\ a}) \Rightarrow \mathsf{a} \rightarrow \mathsf{String}$$
$$\mathsf{gshow} = \mathsf{selShow\ rep}$$

And it works like this:

$\mathsf{gshow}\ (\mathsf{Cons}\ 4\ (\mathsf{Cons}\ 2\ \mathsf{Nil})) \equiv$ `"(Cons 4 (Cons 2 (Nil )))"`

But the output is ugly! We need to fix it...

Overloaded and Ad-hoc (5)

We want something specific for `List`. Instead of the general `rList` representation based on `rtype`, we can add a special list case to `Generic`.

```
class Generic g where
  ...
  list :: g a → g (List a)
```

We also need to register `list` as a representable type.

```
instance (Rep a) ⇒ Rep (List a) where
  rep = list rep
```

Overloaded and Ad-hoc (6)

We extend `gshow` for lists...

```
gshow_list r_a Nil          = "[]"
gshow_list r_a (Cons a as) = selShow r_a a ++ ":" ++
                             selShow (list r_a) as

instance Generic Gshow where
  ...
  list r_a = Gshow (gshow_list r_a)
```

... arriving at a more concise output:

```
gshow (Cons 4 (Cons 2 Nil)) ≡ "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 *EM*GM.

Let's try a hierarchy of classes.

```
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?

```
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`:

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

What happens if we follow GHC's advise?

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

Becoming Modular and Extensible (3)

Instead, let's not assume that `g` is always an instance of `Generic`. We abstract over the type constructor in `Rep`.

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

Becoming Modular and Extensible (4)

We rewrite the instances from before:

```
instance (Generic g) ⇒ Rep g Unit where
  rep = runit
instance (Generic g, Rep g a, Rep g b) ⇒ Rep g (a :+: b)
  where rep = rsum rep rep
...
```

And use `GenericList` in the context for the `List` instance instead of `Generic`.

```
instance (GenericList g, Rep g a) ⇒ Rep g (List a) where
  rep = rlist rep
```

Becoming Modular and Extensible (5)

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.

A Simple Producer: Empty

Here is a simple generic producer function in its entirety.

$$
\begin{array}{l}
\textbf{newtype}\ \mathsf{Gempty\ a} = \mathsf{Gempty}\ \{\mathsf{selEmpty} :: \mathsf{a}\} \\
\textbf{instance}\ \mathsf{Generic\ Gempty}\ \textbf{where} \\
\quad \begin{array}{lll}
\mathsf{runit} & & = \mathsf{Gempty\ Unit} \\
\mathsf{rint} & & = \mathsf{Gempty}\ 0 \\
\mathsf{rchar} & & = \mathsf{Gempty}\ \texttt{'\textbackslash NUL'} \\
\mathsf{rsum} & \mathsf{r_a\ r_b} & = \mathsf{Gempty\ (L\ (selEmpty\ r_a))} \\
\mathsf{rprod} & \mathsf{r_a\ r_b} & = \mathsf{Gempty\ (selEmpty\ r_a}\ {:}{\times}{:}\ \mathsf{selEmpty\ r_b)} \\
\mathsf{rtype\ ep} & \mathsf{r_a} & = \mathsf{Gempty\ (to\ ep\ (selEmpty\ r_a))} \\
\mathsf{rcon\ s} & \mathsf{r_a} & = \mathsf{Gempty\ (selEmpty\ r_a)}
\end{array} \\
\mathsf{gempty} :: \mathsf{(Rep\ Gempty\ a)} \Rightarrow \mathsf{a} \\
\mathsf{gempty} = \mathsf{selEmpty\ rep}
\end{array}
$$

Higher Kinds: Crush (1)

We have dealt with types of kind $*$ up to this point. How do we deal with kind $* \to *$? These include the "container" datatypes: List a, Tree a, etc.

We use a generic crush function as an example.

Higher Kinds: Crush (2)

Recall the standard `foldr` function:

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

It generalizes to `crushr`:

$$\text{crushr} :: (a \to b \to b) \to b \to f\ a \to b$$

- $(a \to b \to b)$ — A "combining" function
- b — A "zero" value
- $f\ a$ — A container

Higher Kinds: Crush (3)

The type-indexed function is straightforward.

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

crushr_unit         _            e = e
 ...
crushr_plus   r_a r_b (L a)      e = selCrush r_a a e
crushr_plus   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_dt ep  r_a     a          e = selCrush r_a (from ep a) e

instance Generic (Crush b) where
  runit = Crush crushr_unit
    ...
```

Higher Kinds: Crush (4)

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 $* \to *$.

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)
```

Higher Kinds: Crush (5)

Translating an instance from `Rep`

```
instance (Generic g, Rep g a) ⇒ Rep g (List a) where
  rep = rList rep
```

to `FRep`

```
instance (Generic g) ⇒ FRep g List where
  frep = rList
```

requires removing all references to the type variables for the container's element.

Higher Kinds: Crush (6)

How do we come up with ...

$\text{crushr} :: (\ldots) \Rightarrow (\text{a} \rightarrow \text{b} \rightarrow \text{b}) \rightarrow \text{b} \rightarrow \text{f a} \rightarrow \text{b}$

... given this, ...

$\text{selCrush} :: \text{Crush b a} \rightarrow \text{a} \rightarrow \text{b} \rightarrow \text{b}$

... this, ...

$\text{frep} :: (\text{FRep g f}) \Rightarrow \text{g a} \rightarrow \text{g (f a)}$

... and this?

$\text{Crush} :: (\text{a} \rightarrow \text{b} \rightarrow \text{b}) \rightarrow \text{Crush b a}$

Higher Kinds: Crush (7)

Let's assemble this type jigsaw puzzle:

$$\mathsf{selCrush} :: \mathsf{Crush}\ b\ a \rightarrow a \rightarrow b \rightarrow b$$

$$\mathsf{frep} :: (\mathsf{FRep}\ g\ f) \Rightarrow g\ a \rightarrow g\ (f\ a)$$

$$\mathsf{Crush} :: (a \rightarrow b \rightarrow b) \rightarrow \mathsf{Crush}\ b\ a$$

First, $\mathsf{frep} \circ \mathsf{Crush}$:

$$\mathsf{frep} \circ \mathsf{Crush} ::$$
$$\quad (\mathsf{FRep}\ (\mathsf{Crush}\ b)\ f) \Rightarrow (a \rightarrow b \rightarrow b) \rightarrow \mathsf{Crush}\ b\ (f\ a)$$

Higher Kinds: Crush (8)

Let's assemble this type jigsaw puzzle:

$$\mathsf{selCrush} :: \mathsf{Crush}\ \mathsf{b}\ \mathsf{a} \rightarrow \mathsf{a} \rightarrow \mathsf{b} \rightarrow \mathsf{b}$$

$$\mathsf{frep} :: (\mathsf{FRep}\ \mathsf{g}\ \mathsf{f}) \Rightarrow \mathsf{g}\ \mathsf{a} \rightarrow \mathsf{g}\ (\mathsf{f}\ \mathsf{a})$$

$$\mathsf{Crush} :: (\mathsf{a} \rightarrow \mathsf{b} \rightarrow \mathsf{b}) \rightarrow \mathsf{Crush}\ \mathsf{b}\ \mathsf{a}$$

Then, $\mathsf{selCrush} \circ \mathsf{frep} \circ \mathsf{Crush}$:

$$\mathsf{selCrush} \circ \mathsf{frep} \circ \mathsf{Crush} ::$$
$$\quad (\mathsf{FRep}\ (\mathsf{Crush}\ \mathsf{b})\ \mathsf{f}) \Rightarrow (\mathsf{a} \rightarrow \mathsf{b} \rightarrow \mathsf{b}) \rightarrow \mathsf{f}\ \mathsf{a} \rightarrow \mathsf{b} \rightarrow \mathsf{b}$$

Higher Kinds: Crush (9)

Finally, we can define `crushr`.

```
crushr :: (FRep (Crush b) f) ⇒ (a → b → b) → b → f a → b
crushr f z x = selCrush (frep (Crush f)) x z
```

And we can use it, too.

```
gflatten :: (FRep (Crush [a]) f) ⇒ f a → [a]
gflatten = crushr (:) []
```

gflatten (Cons 4 (Cons 2 Nil)) $\equiv [4, 2]$

Higher Abstraction: Map (1)

The standard `map` function is a very handy function. We often want to apply a function to all elements in a list.

$$map :: (a \to b) \to [a] \to [b]$$

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

$$gmap :: (a \to b) \to f\ a \to f\ b$$

Higher Abstraction: Map (2)

The type-indexed function.

$$
\begin{array}{llll}
\textbf{newtype}\ \mathsf{Gmap\ a\ b} = \mathsf{Gmap}\ \{\mathsf{selMap} :: \mathsf{a} \rightarrow \mathsf{b}\} \\
\mathsf{gmap\_unit} & & \mathsf{x} & = \mathsf{x} \\
\ldots \\
\mathsf{gmap\_plus} & \mathsf{r_a}\ \mathsf{r_b} & (\mathsf{L\ a}) & = \mathsf{L}\ (\mathsf{selMap}\ \mathsf{r_a}\ \mathsf{a}) \\
\mathsf{gmap\_plus} & \mathsf{r_a}\ \mathsf{r_b} & (\mathsf{R\ b}) & = \mathsf{R}\ (\mathsf{selMap}\ \mathsf{r_b}\ \mathsf{b}) \\
\mathsf{gmap\_prod} & \mathsf{r_a}\ \mathsf{r_b} & (\mathsf{a} :\!\times\!: \mathsf{b}) & = \mathsf{selMap}\ \mathsf{r_a}\ \mathsf{a} :\!\times\!: \mathsf{selMap}\ \mathsf{r_b}\ \mathsf{b}) \\
\mathsf{gmap\_dt}\ \mathsf{ep}_1\ \mathsf{ep}_2 & \mathsf{r_a} & \mathsf{a} & = (\mathsf{to}\ \mathsf{ep}_2 \circ \mathsf{selMap}\ \mathsf{r_a} \circ \mathsf{from}\ \mathsf{ep}_1)\ \mathsf{a}
\end{array}
$$

Higher Abstraction: Map (3)

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

$$
\begin{array}{l}
\textbf{class}\ \mathrm{Generic2}\ g\ \textbf{where} \\
\quad \mathrm{runit2} :: g\ \mathrm{Unit}\ \mathrm{Unit} \\
\quad \mathrm{rint2} :: g\ \mathrm{Int}\ \mathrm{Int} \\
\quad \mathrm{rchar2} :: g\ \mathrm{Char}\ \mathrm{Char} \\
\quad \mathrm{rsum2} :: g\ a_1\ a_2 \rightarrow g\ b_1\ b_2 \rightarrow g\ (a_1 :+: b_1)\ (a_2 :+: b_2) \\
\quad \mathrm{rprod2} :: g\ a_1\ a_2 \rightarrow g\ b_1\ b_2 \rightarrow g\ (a_1 :\times: b_1)\ (a_2 :\times: b_2) \\
\quad \mathrm{rtype2} :: \mathrm{EP}\ a_2\ a_1 \rightarrow \mathrm{EP}\ b_2\ b_1 \rightarrow g\ a_1\ b_1 \rightarrow g\ a_2\ b_2
\end{array}
$$

Higher Abstraction: Map (4)

We define our instance of `Generic2`.

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

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

```
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)))
```

Higher Abstraction: Map (5)

We can immediately use the list representation to implement the standard `map` on `List` containers.

$$\text{mapList} :: (a \rightarrow b) \rightarrow \text{List } a \rightarrow \text{List } b$$
$$\text{mapList } f = \text{selMap } (\text{rList2 } (\text{Gmap } f))$$

But our ultimate goal (as always) is to generalise...

Higher Abstraction: Map (6)

We can't use the `FRep` class. Why?

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

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

```
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`.

Higher Abstraction: Map (7)

We can now define `gmap` with a method similar to how we defined `crushr`.

$$\mathsf{gmap} :: (\mathsf{FRep2\ Gmap\ f}) \Rightarrow (\mathsf{a} \to \mathsf{b}) \to \mathsf{f\ a} \to \mathsf{f\ b}$$
$$\mathsf{gmap\ f} = \mathsf{selMap\ (frep2\ (Gmap\ f))}$$

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

Next time...

- Projects
- Deriving Generics