Multirec

Johan Jeuring

Utrecht University

9 June, 2015

Before the break ...

- Regular: a library in which you can use the recursive structure in *a single* datatype.

Now...

Multirec

- A DGP library (`multirec` on Hackage)
- Uses a type-indexed representation type
- Based on sum-of-products fixed-point view
- Supports recursion over a family of datatypes

Problem

Recall:

```
data Fix f = In {out :: f (Fix f)}
```

Question

How do we represent the following datatypes in a fixed-point view?

```
data Expr  = Lit Int
           | Add Expr Expr
           | Mul Expr Expr
           | Var Name
           | Let Decl Expr
data Decl  = Name := Expr
           | Seq Decl Decl
type Name = String
```

Higher-Kinded Fix

- The datatypes Expr and Decl are mutually recursive.
- Fix takes a functor f with a single type of recursion.

  data Fix f = In {out :: f (Fix f)}

- We need a fixed-point representation with **two recursive types**.

  data Fix_2 f g = In_2 {out_2 :: f (Fix_2 f g) (Fix_2 g f)}

Mutually Recursive Datatypes With Fix_2

$$
\begin{aligned}
\textbf{data}\ \mathsf{ExprF}_2\ r_E\ r_D &= \mathsf{LitF\ Int} \\
&\mid \mathsf{AddF}\ r_E\ r_E \\
&\mid \mathsf{MulF}\ r_E\ r_E \\
&\mid \mathsf{VarF\ Name} \\
&\mid \mathsf{LetF}\ r_D\ r_E \\
\textbf{data}\ \mathsf{DeclF}_2\ r_D\ r_E &= \mathsf{Name} ::= r_E \\
&\mid \mathsf{SeqF}\ r_D\ r_D
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{type}\ \mathsf{Expr}' = \mathsf{Fix}_2\ \mathsf{ExprF}_2\ \mathsf{DeclF}_2 \\
&\textbf{type}\ \mathsf{Decl}' = \mathsf{Fix}_2\ \mathsf{DeclF}_2\ \mathsf{ExprF}_2
\end{aligned}
$$

And then we also need a $\mathsf{Functor}_2$ type class...

Fix_1

Fix_1 can have one recursive type:

$$\mathrm{Fix}_1 :: (* \to *) \to *$$

Fix_2

Fix_2 can have two recursive types:

$$
\begin{aligned}
&\mathsf{Fix}_1 :: (* \to *) && \to * \\
&\mathsf{Fix}_2 :: (* \to * \to *) \to (* \to * \to *) && \to *
\end{aligned}
$$

Fix_3

Fix_3 can have three recursive types:

$$
\begin{aligned}
&\mathsf{Fix}_1 :: (* \to *) && && \to * \\
&\mathsf{Fix}_2 :: (* \to * \to *) && \to (* \to * \to *) && \to * \\
&\mathsf{Fix}_3 :: (* \to * \to * \to *) && \to (* \to * \to * \to *) \to (* \to * \to * \to *) && \to *
\end{aligned}
$$

Fix_n

Question

How do we generalize to arbitrary arities?

$$
\begin{aligned}
&\mathsf{Fix}_1 :: (* \to *) && && && \to * \\
&\mathsf{Fix}_2 :: (* \to * \to *) && \to (* \to * \to *) && && \to * \\
&\mathsf{Fix}_3 :: (* \to * \to * \to *) && \to (* \to * \to * \to *) && \to (* \to * \to * \to *) && \to * \\
&\mathsf{Fix}_n :: ?
\end{aligned}
$$

Uncurry

Let's uncurry the functional kinds.

$$
\begin{aligned}
\mathsf{Fix}_1 &:: (* \to *) && && \to * \\
&:: (* \to *) && && \to * \\
\mathsf{Fix}_2 &:: (* \to * \to *) && \to (* \to * \to *) && \to * \\
&:: (* \times * \to *) && \times (* \times * \to *) && \to * \\
\mathsf{Fix}_3 &:: (* \to * \to * \to *) && \to (* \to * \to * \to *) \to (* \to * \to * \to *) && \to * \\
&:: (* \times * \times * \to *) && \times (* \times * \times * \to *) \times (* \times * \times * \to *) && \to *
\end{aligned}
$$

Exponents

Then, we can reduce the product kinds to exponential kinds.

$\mathsf{Fix}_1 :: (*^{1} \to *)^{1} \to *$

$\mathsf{Fix}_2 :: (*^{2} \to *)^{2} \to *$

$\mathsf{Fix}_3 :: (*^{3} \to *)^{3} \to *$

As a result, we can use `n` as the kind exponent.

$\mathsf{Fix}_{\mathsf{n}} :: (*^{\mathsf{n}} \to *)^{\mathsf{n}} \to *$

n Functors

Given:

$\mathsf{Fix}_\mathsf{n} :: (*^\mathsf{n} \rightarrow *)^\mathsf{n} \rightarrow *$

Fix_n is used for n functors:

$\mathsf{Fix}_\mathsf{n} :: ((*^\mathsf{n} \rightarrow *)^\mathsf{n} \rightarrow *)^\mathsf{n}$

Aside: Laws of Exponents (1)

Function types and functional kinds are exponential.

$t \to u \equiv u^t$

Aside: Laws of Exponents (2)

Note:

- 1 means the unit type, isomorphic to `()`, a type with 1 constructor
- $2 \equiv 1 + 1$, isomorphic to `Bool`, a type with 2 constructors
- etc.

The function type $2 \to 2$ can be written as $\mathsf{Bool} \to \mathsf{Bool}$. For example:

```
not True  = False
not False = True
```

The type is isomorphic to 2^2. Note that for each argument constructor to `not` (`True` or `False`), you have 2 possible output constructors. Thus, you can define $2 * 2$ or 2^2 different functions.
Try other experiments to convince yourself.

Aside: Laws of Exponents (3)

The standard laws for exponents also hold on types.

$$
\begin{aligned}
& 1 \to u \\
& \quad \equiv \\
& u^{1} \\
& \quad \equiv \\
& u
\end{aligned}
$$

$$
\begin{aligned}
& t + u \to v \\
& \quad \equiv \\
& v^{t+u} \\
& \quad \equiv \\
& v^{t} * v^{u} \\
& \quad \equiv \\
& (t \to v) * (u \to v)
\end{aligned}
$$

$$
\begin{aligned}
& t * u \to v \\
& \quad \equiv \\
& v^{t*u} \\
& \quad \equiv \\
& v^{u*t} \\
& \quad \equiv \\
& (v^{u})^{t} \\
& \quad \equiv \\
& t \to v^{u} \\
& \quad \equiv \\
& t \to (u \to v)
\end{aligned}
$$

Rearranging

We can apply the laws of exponents to $\mathsf{Fix_n}$:

$$\begin{aligned}\mathsf{Fix_n} &:: ((*^{\mathsf{n}} \rightarrow *)^{\mathsf{n}} \rightarrow *)^{\mathsf{n}} \\ &:: \mathsf{n} \rightarrow (\mathsf{n} \rightarrow ((\mathsf{n} \rightarrow *) \rightarrow *)) \rightarrow *\end{aligned}$$

Rearranging a bit:

$$\mathsf{Fix_n} :: ((\mathsf{n} \rightarrow *) \rightarrow (\mathsf{n} \rightarrow *)) \rightarrow (\mathsf{n} \rightarrow *)$$

n as a Type or Kind

Given:

$$\mathsf{Fix}_{n} :: ((n \to *) \to (n \to *)) \to (n \to *)$$

We can express it in Agda with a dependent type:

$$\mathsf{Fix} : (n : \mathsf{Nat}) \to ((\mathsf{Fin}\ n \to *) \to (\mathsf{Fin}\ n \to *)) \to \mathsf{Fin}\ n \to *$$

But in Haskell, we are restricted to simple kinds:

$$\mathsf{HFix} :: ((* \to *) \to (* \to *)) \to (* \to *)$$

We would like to say that $*$ should only be instantiated by a suitable index type. Really, we use a bit of "dynamic kinding."

$$\textbf{data}\ \mathsf{HFix}\ (f :: (* \to *) \to (* \to *))\ (ix :: *) = \mathsf{HIn}\ \{\mathsf{hout} :: f\ (\mathsf{HFix}\ f)\ ix\}$$

Back to Expr and Decl

Recall:

$$
\begin{aligned}
\textbf{data}\ \mathrm{Expr}\ &= \mathrm{Lit\ Int} \\
&\mid \mathrm{Add\ Expr\ Expr} \\
&\mid \mathrm{Mul\ Expr\ Expr} \\
&\mid \mathrm{Var\ Name} \\
&\mid \mathrm{Let\ Decl\ Expr} \\
\textbf{data}\ \mathrm{Decl}\ &= \mathrm{Name} := \mathrm{Expr} \\
&\mid \mathrm{Seq\ Decl\ Decl} \\
\textbf{type}\ \mathrm{Name} &= \mathrm{String}
\end{aligned}
$$

Question

How do we represent this in a fixed-point view?

First: A GADT

Instead of three separate types, we could define it as a GADT.

```
data AST :: * → * where
  Lit  :: Int                        → AST Expr
  Add  :: AST Expr → AST Expr        → AST Expr
  Mul  :: AST Expr → AST Expr        → AST Expr
  Var  :: AST Name                   → AST Expr
  Let  :: AST Decl → AST Expr        → AST Expr

  Bind :: AST Name → AST Expr        → AST Decl
  Seq  :: AST Decl → AST Decl        → AST Decl

  Name :: String                     → AST Name
```

`Expr`, `Decl`, and `Name` are empty datatype declarations.

Next: A GADT Pattern Functor

Lift the recursion to a parameter.

$$
\begin{array}{lll}
\textbf{data}\ \mathrm{PF}_{\mathrm{AST}} :: (* \to *) \to * \to *\ \textbf{where} & & \\
\quad \mathrm{LitF} & :: \mathrm{Int} & \to \mathrm{PF}_{\mathrm{AST}}\ \mathrm{r}\ \mathrm{Expr} \\
\quad \mathrm{AddF} & :: \mathrm{r}\ \mathrm{Expr} \to \mathrm{r}\ \mathrm{Expr} & \to \mathrm{PF}_{\mathrm{AST}}\ \mathrm{r}\ \mathrm{Expr} \\
\quad \mathrm{MulF} & :: \mathrm{r}\ \mathrm{Expr} \to \mathrm{r}\ \mathrm{Expr} & \to \mathrm{PF}_{\mathrm{AST}}\ \mathrm{r}\ \mathrm{Expr} \\
\quad \mathrm{VarF} & :: \mathrm{r}\ \mathrm{Name} & \to \mathrm{PF}_{\mathrm{AST}}\ \mathrm{r}\ \mathrm{Expr} \\
\quad \mathrm{LetF} & :: \mathrm{r}\ \mathrm{Decl} \to \mathrm{r}\ \mathrm{Expr} & \to \mathrm{PF}_{\mathrm{AST}}\ \mathrm{r}\ \mathrm{Expr} \\
\quad \mathrm{BindF} & :: \mathrm{r}\ \mathrm{Name} \to \mathrm{r}\ \mathrm{Expr} & \to \mathrm{PF}_{\mathrm{AST}}\ \mathrm{r}\ \mathrm{Decl} \\
\quad \mathrm{SeqF} & :: \mathrm{r}\ \mathrm{Decl} \to \mathrm{r}\ \mathrm{Decl} & \to \mathrm{PF}_{\mathrm{AST}}\ \mathrm{r}\ \mathrm{Decl} \\
\quad \mathrm{NameF} & :: \mathrm{String} & \to \mathrm{PF}_{\mathrm{AST}}\ \mathrm{r}\ \mathrm{Name}
\end{array}
$$

Tying the Knot with HFix

We can recover the recursion with HFix:

$$\mathbf{type}\ \mathsf{AST}' = \mathsf{HFix}\ \mathsf{PF}_{\mathsf{AST}}$$

Note how the index ix is used in HFix.

$$\mathbf{data}\ \mathsf{HFix}\ \mathsf{f}\ \mathsf{ix} = \mathsf{HIn}\ \{\mathsf{hout} :: \mathsf{f}\ (\mathsf{HFix}\ \mathsf{f})\ \mathsf{ix}\}$$

$$\begin{aligned}
&\mathbf{data}\ \mathsf{PF}_{\mathsf{AST}}\ \mathsf{r}\ \mathsf{ix}\ \mathbf{where} \\
&\quad \mathsf{AddF} :: \mathsf{r}\ \mathsf{Expr} \to \mathsf{r}\ \mathsf{Expr} \to \mathsf{PF}_{\mathsf{AST}}\ \mathsf{r}\ \mathsf{Expr} \\
&\quad \mathsf{BindF} :: \mathsf{r}\ \mathsf{Name} \to \mathsf{r}\ \mathsf{Expr} \to \mathsf{PF}_{\mathsf{AST}}\ \mathsf{r}\ \mathsf{Decl} \\
&\qquad \ldots
\end{aligned}$$

Pattern Functor in Regular

Question

Why can't we use Regular?

data U $\quad\quad$ r = U

data (f :+: g) r = L (f r) | R (g r)

data (f :×: g) r = f r :×: g r

newtype I $\quad$ r = I r

newtype K a r = K a

Pattern Functor With Indexes

We need the index of the type in the family.

$$
\begin{array}{ll}
\textbf{data}\ \mathsf{U} & (r :: * \to *)\ ix = \mathsf{U} \\
\textbf{data}\ (f :+: g) & (r :: * \to *)\ ix = \mathsf{L}\ (f\ r\ ix) \mid \mathsf{R}\ (g\ r\ ix) \\
\textbf{data}\ (f :\times: g) & (r :: * \to *)\ ix = f\ r\ ix :\times: g\ r\ ix \\
\textbf{data}\ \mathsf{I}\ xi & (r :: * \to *)\ ix = \mathsf{I}\ (r\ xi) \\
\textbf{data}\ \mathsf{K}\ a & (r :: * \to *)\ ix = \mathsf{K}\ a
\end{array}
$$

Question

How do we represent the difference between an `Expr` constructor and a `Decl` constructor?

We tag the constructor with its index.

$$
\begin{array}{l}
\textbf{data}\ f :\triangleright: ix :: (* \to *) \to * \to * \to *\ \textbf{where} \\
\quad \mathsf{Tag} :: f\ r\ ix \to (f :\triangleright: ix)\ r\ ix
\end{array}
$$

Pattern Functor of AST

Here's the pattern functor type for AST:

$$
\begin{array}{lllll}
\textbf{type}\ \mathsf{PF}_{\mathsf{AST}} = & \mathsf{K\ Int} & :\triangleright\ \mathsf{Expr}\ :+: & \text{-- } & \mathsf{Lit} \\
 & (\mathsf{I\ Expr}\ :\times:\ \mathsf{I\ Expr}) & :\triangleright\ \mathsf{Expr}\ :+: & \text{-- } & \mathsf{Add} \\
 & (\mathsf{I\ Expr}\ :\times:\ \mathsf{I\ Expr}) & :\triangleright\ \mathsf{Expr}\ :+: & \text{-- } & \mathsf{Mul} \\
 & \mathsf{I\ Name} & :\triangleright\ \mathsf{Expr}\ :+: & \text{-- } & \mathsf{Var} \\
 & (\mathsf{I\ Decl}\ :\times:\ \mathsf{I\ Expr}) & :\triangleright\ \mathsf{Expr}\ :+: & \text{-- } & \mathsf{Let} \\
 & (\mathsf{I\ Name}\ :\times:\ \mathsf{I\ Expr}) & :\triangleright\ \mathsf{Decl}\ :+: & \text{-- } & := \\
 & (\mathsf{I\ Decl}\ :\times:\ \mathsf{I\ Decl}) & :\triangleright\ \mathsf{Decl}\ :+: & \text{-- } & \mathsf{Seq} \\
 & \mathsf{K\ String} & :\triangleright\ \mathsf{Name} & \text{-- } & \mathsf{Name}
\end{array}
$$

Families of Datatypes

- As with Regular, we can use a type-indexed type for structure representation types such as PF_{AST} .
- But we're missing one thing.

Question

What is the index of PF ?

type family PF ...

Describing a Family

type family PF $(\phi :: * \to *) :: (* \to *) \to * \to *$

- The index of the PF indicates the **family** or **system** of types.
- All functions in Multirec operate on a system of types. ("System" being a better word to distinguish it from the type families in GHC.)
- The system type ϕ:
  - Enumerates all types in the system
  - Reveals the index type (of kind $*$)
- The system datatype is a GADT:

```
data AST :: * → * where
  Expr :: AST Expr
  Decl :: AST Decl
  Name :: AST Name
```

Representation Class

Also, as with Regular, we use a class to define the embedding-projection pair for translating between the Haskell type and its pattern functor representation.

```
class Fam φ where
   from :: φ ix → ix → PF φ I0 ix
   to   :: φ ix → PF φ I0 ix → ix
```

```
newtype I0 a = I0 a
```

Representation Instances

The representation instances are straightforward, if tedious.

$$\textbf{type instance}\ \mathsf{PF\ AST} = \mathsf{PF}_{\mathsf{AST}}$$

$$i_0 = \mathsf{I} \circ \mathsf{I0}$$

$$
\begin{array}{llll}
\multicolumn{4}{l}{\textbf{instance}\ \mathsf{Fam\ AST}\ \textbf{where}} \\
\quad \mathsf{from\ Expr} & (\mathsf{Lit}\ i) & = \mathsf{L}\ \$\ \mathsf{Tag}\ \$ & \mathsf{K}\ i \\
\quad \mathsf{from\ Expr} & (\mathsf{Add}\ e_1\ e_2) & = \mathsf{R}\ \$\ \mathsf{L}\ \$\ \mathsf{Tag}\ \$ & i_0\ e_1 :\times: i_0\ e_2 \\
\quad \ldots & & & \\
\quad \mathsf{from\ Decl} & (v := e) & = \mathsf{R}\ \$\ \mathsf{R}\ \$\ \mathsf{R}\ \$\ \mathsf{R}\ \$\ \mathsf{R}\ \$\ \mathsf{L}\ \$\ \mathsf{Tag}\ \$ & i_0\ v :\times: i_0\ e \\
\quad \ldots & & & \\
\quad \mathsf{from\ Name}\ n & & \multicolumn{2}{l}{= \mathsf{R}\ \$\ \mathsf{R}\ \$\ \mathsf{R}\ \$\ \mathsf{R}\ \$\ \mathsf{R}\ \$\ \mathsf{R}\ \$\ \mathsf{R}\ \$\ \mathsf{Tag}\ \$\ \mathsf{K}\ n} \\
\quad \ldots & & &
\end{array}
$$

Representation Via Template Haskell

Fortunately, you can generate these instances with Template Haskell:

```
$(deriveConstructors [''Expr, ''Decl, ''Name])
$(deriveFamily ''AST [''Expr, ''Decl, ''Name] "PF_AST")
type instance PF AST = PF_AST
```

Generic Left

As an example of Multirec, we will define the generic function that produces the leftmost value of a datatype. It's called "empty" in other GP libraries.

$$\mathsf{left} :: (\mathsf{El}\ \phi\ \mathsf{ix}, \mathsf{Fam}\ \phi, \mathsf{Left}\ \phi\ (\mathsf{PF}\ \phi)) \Rightarrow \phi\ \mathsf{ix} \to \mathsf{ix}$$
$$\mathsf{left}\ \mathsf{w} = \mathsf{to}\ \mathsf{w}\ \$\ \mathsf{fromJust}\ \$\ \mathsf{gleft}\ \mathsf{w}\ (\mathsf{I0} \circ \mathsf{left})$$

Defining a Generic Function: Left

class Left ϕ f **where**
  gleft :: ϕ ix $\to$ r ix $\to$ Maybe (f r ix)

instance Left ϕ U **where**
  gleft _ _ = Just U

instance (Left ϕ a, Left ϕ b) $\Rightarrow$ Left ϕ (a :+: b) **where**
  gleft w r = fmap L (gleft w r)

instance (Left ϕ a, Left ϕ b) $\Rightarrow$ Left ϕ (a :$\times$: b) **where**
  gleft w r = (:$\times$:) $\langle \$ \rangle$ gleft w r $\langle \star \rangle$ gleft w r

Constant Types

For constant types, we use an additional type class:

$$
\begin{aligned}
&\textbf{instance } (\mathrm{Left}_K\ a) \Rightarrow \mathrm{Left}\ \phi\ (K\ a)\ \textbf{where}\\
&\quad \mathrm{gleft}\ \_\ \_ = \mathrm{Just}\ (K\ \mathrm{left}_K)
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{class } \mathrm{Left}_K\ a\ \textbf{where}\\
&\quad \mathrm{left}_K :: a\\
&\textbf{instance } \mathrm{Left}_K\ \mathrm{Char}\ \textbf{where}\\
&\quad \mathrm{left}_K = \mathrm{minBound}
\end{aligned}
$$

Recursion

For recursion, we use the argument:

instance Left ϕ (I xi) **where**
gleft _ r = Just (I r)

Oops!

```
Couldn't match type ‘ix’ with ‘xi’
  ‘ix’ is a rigid type variable bound by
       the type signature for ‘gleft’
  ‘xi’ is a rigid type variable bound by
       the instance declaration
In the first argument of ‘I’, namely ‘r’
```

We Need a Witness!

- Of course, ix and xi are not necessarily going to be the same.
- We support recursion on multiple types.
- We need to identify which type is this particular recursive reference.
- To produce a witness, we use a type class.

class El ϕ ix **where**
  proof :: ϕ ix

We need instances for all the possible proofs, namely all types in the family.

instance El AST Expr **where** proof = Expr
instance El AST Decl **where** proof = Decl
instance El AST Name **where** proof = Name

Recursion (Actually)

To handle recursion properly, we need to redefine the Left class.

- Since recursive indexes will be different, this indicates that we need a rank-2 type.
- Since we need to get a witness of the recursive index, this indicates that we need a El constraint.

class Left ϕ f **where**
  gleft :: ϕ ix $\to$ ($\forall$xi.El ϕ xi $\Rightarrow$ ϕ xi $\to$ r xi) $\to$ Maybe (f r ix)

And so the recursive case can now be defined. (The rest is unchanged.)

instance (El ϕ xi) $\Rightarrow$ Left ϕ (I xi) **where**
  gleft _ r = Just (I (r proof))

Tags

Recall:

$$\textbf{data}\ f :\triangleright: ix :: (* \to *) \to * \to *\ \textbf{where}$$
$$\quad \mathsf{Tag} :: f\ r\ ix \to (f :\triangleright: ix)\ r\ ix$$

For tags, we need to prove that the tag index and the family index are equal.

$$\textbf{instance}\ (\mathsf{TEq}\ \phi, \mathsf{El}\ \phi\ ix, \mathsf{Left}\ \phi\ f) \Rightarrow \mathsf{Left}\ \phi\ (f :\triangleright: ix)\ \textbf{where}$$
$$\quad gleft\ w\ r = \textbf{do}$$
$$\qquad \mathsf{Refl} \leftarrow teq\ (proof :: \phi\ ix)\ w$$
$$\qquad fmap\ \mathsf{Tag}\ (gleft\ w\ r)$$

Type Equality

To determine type equality on the indexes, we use a GADT and type class.

data (:=:) :: $* \to * \to *$ **where**
  Refl :: a :=: a
class TEq f **where**
  teq :: f a $\to$ f b $\to$ Maybe (a :=: b)

HFunctor

Here is a generalization of fmap :

$$
\begin{aligned}
&\textbf{class}\ \mathsf{HFunctor}\ \phi\ \mathsf{f}\ \textbf{where}\\
&\quad \mathsf{hmapA} :: (\mathsf{Applicative}\ \mathsf{a})\\
&\qquad\qquad \Rightarrow (\forall \mathsf{xi}.\phi\ \mathsf{xi} \to \mathsf{r}\ \mathsf{xi} \to \mathsf{a}\ (\mathsf{s}\ \mathsf{xi}))\\
&\qquad\qquad \to \phi\ \mathsf{ix} \to \mathsf{f}\ \mathsf{r}\ \mathsf{ix} \to \mathsf{a}\ (\mathsf{f}\ \mathsf{s}\ \mathsf{ix})
\end{aligned}
$$

Note:

- The recursive type changes.
- The function parameter for recursion is polymorphic on the index.

We won't look at the definition of HFunctor .

hmap

With hmapA, we can define hmap:

$$
\begin{aligned}
&\text{hmap} :: (\text{HFunctor}\ \phi\ \text{f}) \\
&\qquad \Rightarrow (\forall \text{xi}.\phi\ \text{xi} \rightarrow \text{r xi} \rightarrow \text{s xi}) \\
&\qquad \rightarrow \phi\ \text{ix} \rightarrow \text{f r ix} \rightarrow \text{f s ix} \\
&\text{hmap f p x} = \text{unI0 (hmapA}\ (\lambda \text{ix x} \rightarrow \text{I0 (f ix x)) p x)}
\end{aligned}
$$

fold

With `hmap`, we can define `fold`:

$$\textbf{type}\ \mathsf{Algebra}\ \phi\ \mathsf{r} = \forall \mathsf{ix}.\phi\ \mathsf{ix} \to (\mathsf{PF}\ \phi)\ \mathsf{r}\ \mathsf{ix} \to \mathsf{r}\ \mathsf{ix}$$

$$\begin{aligned}
&\mathsf{fold} :: (\mathsf{Fam}\ \phi, \mathsf{HFunctor}\ \phi\ (\mathsf{PF}\ \phi)) \\
&\qquad \Rightarrow \mathsf{Algebra}\ \phi\ \mathsf{r} \to \phi\ \mathsf{ix} \to \mathsf{ix} \to \mathsf{r}\ \mathsf{ix} \\
&\mathsf{fold}\ \mathsf{f}\ \mathsf{p} = \mathsf{f}\ \mathsf{p} \circ \mathsf{hmap}\ (\lambda \mathsf{p}\ (\mathsf{I0}\ \mathsf{x}) \to \mathsf{fold}\ \mathsf{f}\ \mathsf{p}\ \mathsf{x})\ \mathsf{p} \circ \mathsf{from}\ \mathsf{p}
\end{aligned}$$

But this isn't the only sort of `fold` that can be used with Multirec. See the library for more.

Other Multirec Features

Some features that we did not discuss:

- Metadata on constructors
- Type composition

Using Multirec

Using Multirec requires certain prerequisites:

- Define a GADT for the system of datatypes
- Define the pattern functor for the system (i.e. the PF instance)
- Define instances of Fam and El for the datatypes in the system.
- Define other instances for functions as necessary

Applications

There are numerous applications for Multirec:

- General purpose functions: eq, enum, show, size, etc.
- Recursion schemes such as folds, paramorphisms, etc.
- Rewriting (including unification)
- Generation of values
- The zipper

Conclusions

- This library solved an open generic programming problem in Haskell: how can we write generic functions that use the fixed-point structure of datatypes on mutually recursive datatypes?
- The library has quite a number of useful applications.