Multirec

Johan Jeuring

Utrecht University

20 February, 2014

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

```
data ExprF2 r_E r_D = LitF Int
                    | AddF r_E r_E
                    | MulF r_E r_E
                    | VarF Name
                    | LetF r_D r_E
data DeclF2 r_D r_E = Name ::= r_E
                    | SeqF r_D r_D
```

```
type Expr' = Fix_2 ExprF2 DeclF2
type Decl' = Fix_2 DeclF2 ExprF2
```

And then we also need a Functor_2 type class...

Fix_1

Fix_1 can have one recursive type:

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

Fix_2

Fix_2 can have two recursive types:

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

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

Fix_3

Fix_3 can have three recursive types:

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

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

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

Fix_n

Question

How do we generalize to arbitrary arities?

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

$$\text{Fix\_2} :: (* \to * \to *) \to (* \to * \to *) \to *$$

$$\text{Fix\_3} :: (* \to * \to * \to *) \to (* \to * \to * \to *) \to (* \to * \to * \to *) \to *$$

$$\text{Fix\_n} :: ?$$

Uncurry

Let's uncurry the functional kinds.

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

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.

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

n Functors

Given:

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

Fix_n is used for n functors:

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

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 `Fix_n` :

$$\begin{aligned}\mathsf{Fix\_n} &:: ((*^{\mathsf{n}} \to *)^{\mathsf{n}} \to *)^{\mathsf{n}} \\ &:: \mathsf{n} \to (\mathsf{n} \to ((\mathsf{n} \to *) \to *)) \to *\end{aligned}$$

Rearranging a bit:

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

n as a Type or Kind

Given:

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

We can express it in Agda with a dependent type:

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

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."

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

Back to Expr and Decl

Recall:

```
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
```

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.

```
data PF_AST :: (* → *) → * → * where
  LitF  :: Int                → PF_AST r Expr
  AddF  :: r Expr → r Expr    → PF_AST r Expr
  MulF  :: r Expr → r Expr    → PF_AST r Expr
  VarF  :: r Name             → PF_AST r Expr
  LetF  :: r Decl → r Expr    → PF_AST r Expr
  BindF :: r Name → r Expr    → PF_AST r Decl
  SeqF  :: r Decl → r Decl    → PF_AST r Decl
  NameF :: String             → PF_AST r Name
```

Tying the Knot with HFix

We can recover the recursion with `HFix`:

```
type AST' = HFix PF_AST
```

Note how the index `ix` is used in `HFix`.

```
data HFix f ix = HIn {hout :: f (HFix f) ix}

data PF_AST r ix where
  AddF  :: r Expr → r Expr  → PF_AST r Expr
  BindF :: r Name → r Expr → PF_AST r Decl
    ...
```

Pattern Functor in Regular

Question

Why can't we use Regular?

```
data U       r = U
data (f :+: g) r = L (f r) | R (g r)
data (f :×: g) r = f r :×: g r
newtype I    r = I r
newtype K a r = K a
```

Pattern Functor With Indexes

We need the index of the type in the family.

```
data U       (r :: * → *) ix = U
data (f :+: g) (r :: * → *) ix = L (f r ix) | R (g r ix)
data (f :×: g) (r :: * → *) ix = f r ix :×: g r ix
data I xi    (r :: * → *) ix = I (r xi)
data K a     (r :: * → *) ix = K a
```

Question

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

We tag the constructor with its index.

```
data f :>: ix :: (* → *) → * → * where
  Tag :: f r ix → (f :>: ix) r ix
```

Pattern Functor of AST

Here's the pattern functor type for AST:

```
type PF_AST = K Int                    :>: Expr :+:  --  Lit
            | (I Expr   :×: I Expr)    :>: Expr :+:  --  Add
            | (I Expr   :×: I Expr)    :>: Expr :+:  --  Mul
            | I Name                   :>: Expr :+:  --  Var
            | (I Decl   :×: I Expr)    :>: Expr :+:  --  Let
            | (I Name   :×: I Expr)    :>: Decl :+:  --  :=
            | (I Decl   :×: I Decl)    :>: Decl :+:  --  Seq
            | K String                 :>: Name      --  Name
```

Families of Datatypes

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

Question

What is the index of PF?

type family PF ...

Describing a Family

```
type family PF (phi :: * → *) :: (* → *) → * → *
```

- 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 `phi`:
  - 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 phi where
  from :: phi ix → ix → PF phi I0 ix
  to   :: phi ix → PF phi I0 ix → ix
```

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

Representation Instances

The representation instances are straightforward, if tedious.

```
type instance PF AST = PF_AST
```

```
i_0 = I ∘ I0

instance Fam AST where
  from Expr  (Lit i)       = L $ Tag $                    K i
  from Expr  (Add e_1 e_2) = R $ L $ Tag $                i_0 e_1 :×: i_0 e_2
  ...
  from Decl  (v := e)      = R $ R $ R $ R $ R $ L $ Tag $ i_0 v :×: i_0 e
  ...
  from Name n              = R $ R $ R $ R $ R $ R $ R $ Tag $ K n
  ...
```

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 = $\mathsf{PF}_{\mathsf{AST}}$

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.

```
left :: (El phi ix, Fam phi, Left phi (PF phi)) ⇒ phi ix → ix
left w = to w $ fromJust $ gleft w (I0 ∘ left)
```

Defining a Generic Function: Left

```
class Left phi f where
  gleft :: phi ix → r ix → Maybe (f r ix)
```

```
instance Left phi U where
  gleft _ _ = Just U
```

```
instance (Left phi a, Left phi b) ⇒ Left phi (a :+: b) where
  gleft w r = fmap L (gleft w r)
```

```
instance (Left phi a, Left phi b) ⇒ Left phi (a :×: b) where
  gleft w r = (:×:) ⟨$⟩ gleft w r ⟨⋆⟩ gleft w r
```

Constant Types

For constant types, we use an additional type class:

```
instance (Left_K a) ⇒ Left phi (K a) where
  gleft _ _ = Just (K left_K)
```

```
class Left_K a where
  left_K :: a

instance Left_K Char where
  left_K = minBound
```

Recursion

For recursion, we use the argument:

```
instance Left phi (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 phi ix where
  proof :: phi 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 phi f where
  gleft :: phi ix → (∀xi.El phi xi ⇒ phi xi → r xi) → Maybe (f r ix)
```

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

```
instance (El phi xi) ⇒ Left phi (I xi) where
  gleft _ r = Just (I (r proof))
```

Tags

Recall:

```
data f :>: ix :: (* → *) → * → * where
  Tag :: f r ix → (f :>: ix) r ix
```

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

```
instance (TEq phi, El phi ix, Left phi f) ⇒ Left phi (f :>: ix) where
  gleft w r = do
    Refl ← teq (proof :: phi ix) w
    fmap Tag (gleft w r)
```

Type Equality

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

```
data (:=:) :: * → * → * where
  Refl :: a :=: a

class TEq f where
  teq :: f a → f b → Maybe (a :=: b)
```

HFunctor

Here is a generalization of `fmap`:

```
class HFunctor phi f where
  hmapA :: (Applicative a)
        ⇒ (∀xi.phi xi → r xi → a (s xi))
        → phi ix → f r ix → a (f s ix)
```

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`:

```
hmap :: (HFunctor phi f)
     ⇒ (∀xi.phi xi → r xi → s xi)
     → phi ix → f r ix → f s ix
hmap f p x = unI0 (hmapA (λix x → I0 (f ix x)) p x)
```

fold

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

```
type Algebra phi r = ∀ix.phi ix → (PF phi) r ix → r ix

fold :: (Fam phi, HFunctor phi (PF phi))
     ⇒ Algebra phi r → phi ix → ix → r ix
fold f p = f p ∘ hmap (λp (I0 x) → fold f p x) p ∘ from p
```

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.