Regular

Johan Jeuring

Utrecht University

20 February, 2014

Regular

- A DGP library (`regular` on Hackage)
- Uses a type-indexed representation type
- Based on sum-of-products fixed-point view
- Supports the generic fold

Generic Deriving (1)

The Generic Deriving structure representation:

$$\begin{array}{ll}\textbf{data}\ \mathsf{U\_1} & \mathsf{p} = \mathsf{U\_1} \\ \textbf{data}\ (\mathsf{f} :+: \mathsf{g}) & \mathsf{p} = \mathsf{L\_1}\ (\mathsf{f}\ \mathsf{p}) \mid \mathsf{R\_1}\ (\mathsf{g}\ \mathsf{p}) \\ \textbf{data}\ (\mathsf{f} :\times: \mathsf{g}) & \mathsf{p} = \mathsf{f}\ \mathsf{p} :\times: \mathsf{g}\ \mathsf{p} \\ \textbf{newtype}\ \mathsf{Par}_1 & \mathsf{p} = \mathsf{Par}_1\ \mathsf{p} \\ \textbf{newtype}\ \mathsf{Rec}_1\ \mathsf{f} & \mathsf{p} = \mathsf{Rec}_1\ (\mathsf{f}\ \mathsf{p})\end{array}$$

- Supports parameterized types with sum-of-products view
- The type parameter `p` represents the type parameter of the Haskell datatype
- Recursion is indicated by a value of the Haskell datatype `f` applied to the parameter

Generic Deriving (2)

The List type representation in Generic Deriving:

```
data List a = Nil | Cons a (List a)
instance Generic1 List where
  type Rep1 List = U_1 :+: Par1 :×: Rec1 List
```

- "Recursion" here is really only a tag
- We could change the "name" of the tag to represent a different type:

```
data Two a = Zero | OneOrTwo a (Maybe a)
instance Generic1 Two where
  type Rep1 Two = U_1 :+: Par1 :×: Rec1 Maybe
```

- Given Rep_1 Two or Rep_1 List, we don't know where the recursive positions are

Recursion

- Recursion is very important in FP
- Explicit recursion is the use of a function in its definition

$\text{fac n} = \textbf{if } \text{n} \leqslant 0 \textbf{ then } 1 \textbf{ else } \text{n} * \text{fac (pred n)}$

- Explicit recursion can be difficult to do correctly
  - Avoid/ensure nontermination
  - Laziness and strictness
  - Pass appropriate arguments to recursive calls
- There are schemes that describe variants of recursion:
  - catamorphism: fold, "natural" recursion
  - anamorphism: unfold, dual of catamorphism
  - hylomorphism: composition of catamorphism and anamorphism
  - ...
- Functions for these schemes avoid problems with explicit recursion

$\text{fac}' \text{ n} = \text{foldl}' \ (*) \ 1 \ [1 \mathinner{..} \text{n}]$

Folds for Datatypes

Recall Fix:

```
data Fix f        = In {out :: f (Fix f)}
data List_F a r = Nil_F | Cons_F a r
type List a       = Fix (List_F a)
```

- We raise the recursive reference to a parameter
- We **manually** recreate the structure of the datatype
- Now, we can systematically represent the structure **and** the recursive reference

Enter Regular

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
```

Generic Deriving:

```
data U_1              p = U_1
data (f :+: g)        p = L_1 (f p) | R_1 (g p)
data (f :×: g)        p = f p :×: g p
newtype Rec1 f p = Rec1 (f p)
```

- The parameters for unit, sum, and product are used in the same way
- For recursion:
  - I uses the parameter directly
  - Rec_1 uses the parameter to encode a saturated functor f
  - r can be one type per representation
  - f can be any provided type at each Rec_1
- In other words, I actually encodes recursion and Rec_1 does not

Representing Lists

Now, we can represent the same `List_F` type...

```
data List_F a r = Nil_F | Cons_F a r
```

... with some help for constant types (which are like a unit with a value) ...

```
newtype K a r = K a
```

... as the following:

```
type List_F′ a = U :+: K a :×: I
type List′   a = Fix (List_F′ a)
```

Pattern Functor

In Regular and other libraries that use this view, we call the representation a pattern functor.

type family PF a :: $* \to *$

PF is a type-indexed type encoding a parameterized representation.

type instance PF (List a) = U :+: K a :×: I

We also require Functor instances for the representation.

instance Functor U **where** ...
instance (Functor f, Functor g) ⇒ Functor (f :+: g) **where** ...
...

Isomorphism with Representation

We instantiate a type class to define the embedding-projection pair.

class Regular a **where**
  from :: a $\rightarrow$ PF a a
  to   :: PF a a $\rightarrow$ a

instance Regular (List a) **where** ...

Question

Why is the parameter of the pattern functor duplicated?

Defining Generic Equality (1)

Generic functions, such as equality:

```
class Geq f where
```

- Operate on the parameterized representation

```
geq :: ... → f r → f r → Bool
```

- Use a function argument for recursion

```
geq :: (r → r → Bool) → f r → f r → Bool
```

Defining Generic Equality (2)

The type cases for generic equality:

```
instance Geq U where
  geq _ U U = True
instance (Geq f, Geq g) ⇒ Geq (f :×: g) where
  geq f (x_1 :×: y_1) (x_2 :×: y_2) = geq f x_1 x_2 ∧ geq f y_1 y_2
...
```

Constant types must support non-generic equality:

```
instance (Eq a) ⇒ Geq (K a) where
  geq _ (K x) (K y) = x ≡ y
```

Recursion uses the function argument:

```
instance Geq I where
  geq f (I x) (I y) = f x y
```

Defining Generic Equality (3)

The final generic function uses explicit recursion:

$$\text{eq} :: (\text{Regular a}, \text{Geq (PF a)}) \Rightarrow \text{a} \rightarrow \text{a} \rightarrow \text{Bool}$$
$$\text{eq x y} = \text{geq eq (from x) (from y)}$$

Why the Fixed-Point View?

There are a large number of applications that use the recursive structure of datatypes:

- Fold and its variants (Malcolm, Meijer et al)
- Accumulations on trees (Bird, Gibbons)
- Unification, and matching (Jansson, Jeuring)
- Rewriting (Jansson, Jeuring, van Noort et al)
- Pattern matching (Jeuring)
- Design patterns (Gibbons)
- The zipper and its variants (McBride, Hinze, Jeuring, Löh)
- Subterm selection (Van Steenbergen et al)
- Generating arbitrary elements (for QuickCheck; Hesselink, Jeuring, Löh, Magalhães)

Folds and Algebras (1)

In Regular, implementing the generic fold requires two components:

- The algebra
- Recursion

An algebra, specifically an `F`-algebra, is defined according to the structure of the functor `F`.

Folds and Algebras (2)

We define a type-indexed type for algebras that is indexed by the functor type `f`:

```
type family Alg (f :: * → *) r
```

The type `Alg f` indicates some structure that will “extract” an element from the functor `f`. For example:

```
type instance Alg Maybe r = (r, r → r)
applyMaybeAlg :: Alg Maybe r → Maybe r → r
applyMaybeAlg (n, _) Nothing = n
applyMaybeAlg (_, j ) (Just x) = j x
```

Folds and Algebras (3)

Application of the algebra is also defined according to the structure of the functor. As usual, we use a type class:

```
class Apply f where
  apply :: Alg f r → f r → r
```

Then, we can define instances for the representation types:

```
type instance Alg U r = r

instance Apply U where
  apply f U = f
```

The binary sum requires a pair of algebras, one for each alternative.

```
type instance Alg (f :+: g) r = (Alg f r, Alg g r)

instance (Apply f, Apply g) ⇒ Apply (f :+: g) where
  apply (f, _) (L x) = apply f x
  apply (_, g) (R y) = apply g y
```

Folds and Algebras (4)

Constant and recursive algebras are simple functions.

type instance Alg (K a) r = a → r

instance Apply (K a) **where**
  apply f (K x) = f x

type instance Alg I r = r → r

instance Apply I **where**
  apply f (I x) = f x

Folds and Algebras (5)

The binary product algebra is the composition of algebras. We simplify the composition to define only the product cases that we expect to find.

```
type instance Alg (K a :×: g) r = a → Alg g r
instance (Apply g) ⇒ Apply (K a :×: g) where
  apply f (K x :×: y) = apply (f x) y
```

```
type instance Alg (I :×: g) r = r → Alg g r
instance (Apply g) ⇒ Apply (I :×: g) where
  apply f (I x :×: y) = apply (f x) y
```

Note that this implies a right-nested representation.

Folds and Algebras (6)

In the `fold`, the algebra is applied recursively to the functorial representation.

$$\text{fold} :: (\text{Regular a}, \text{Apply (PF a)}, \text{Functor (PF a)}) \Rightarrow \text{Alg (PF a) r} \rightarrow \text{a} \rightarrow \text{r}$$
$$\text{fold alg} = \text{apply alg} \circ \text{fmap (fold alg)} \circ \text{from}$$

Folds and Algebras (7)

Using the `fold` only requires defining an algebra:

$$\text{listMaxAlg} :: \text{Alg}\ (\text{PF}\ (\text{List Int}))\ \text{Int}$$
$$\text{listMaxAlg} = (\text{minBound}, \text{max})$$

$$\text{listMax} :: \text{List Int} \rightarrow \text{Int}$$
$$\text{listMax} = \text{fold listMaxAlg}$$

Question

What is the dual "top-down" recursion scheme? What can we do with it?

Resources

- Thomas van Noort, Alexey Rodriguez, Stefan Holdermans, Johan Jeuring, Bastiaan Heeren. A Lightweight Approach to Datatype-Generic Rewriting. WGP 2008.