Polytypic Programming in Haskell

Ulf Norell and Patrik Jansson
Chalmers University of Technology

Goals

- Polytypic functions
  - On regular datatypes of kind $* \to *$
  - In Haskell
- How?
  - Type classes!

```
> pmap (+1) [1,2,3]
[2,3,4]
> pmap (+1) (Branch (Leaf 1) (Leaf 2))
Branch (Leaf 2) (Leaf 3)
```

Overview

- What is the structure of a datatype?
- How to define polytypic functions
- Example functions (pmap, cata, transpose)
- Dealing with non–regular datatypes
- Preprocessor support

Pattern Functors

- Pattern functors model the structure of a datatype

```
data [a]     =   []      |    a    :   [a]
type ListF   = Empty :+: Par :*: Rec
```

- Note: ListF :: $* \rightarrow * \rightarrow *$

  We abstract over the recursive call

```
data     (f :+: g) a r = InL (f a r) | InR (g a r)
data     (f :*: g) a r = f a r :*: g a r
data     Empty     a r = Empty
newtype Par        a r = Par a
newtype Rec        a r = Rec r
```

FunctorOf

- The FunctorOf class connects a datatype and its pattern functor
- inn/out folds/unfolds the **top–level** structure
- Only one pattern functor per datatype

```
class FunctorOf f d | d → f where
    inn :: f a (d a) → d a
    out :: d a → f a (d a)
```

```
instance FunctorOf ListF [] where
    inn (InL Empty)                = []
    inn (InR (Par x :*: Rec xs)) = x : xs
    out []         = InL Empty
    out (x : xs) = InR (Par x :*: Rec xs)
```

Note: Recursive calls are untouched

Defining polytypic function
–pattern functor level

- Polytypic function = pattern functor class + instances

```
class P_fmap2 f where
   fmap2 :: (a → c) → (b → d) → f a b → f c d

instance (P_fmap2 f, P_fmap2 g) ⇒ P_fmap2 (f :*: g) where
   fmap2 p r (x :*: y) = fmap2 p r x :*: fmap2 p r y
...
```

- Only polytypic functions over pattern functors!
  We want functions over regular datatypes.

Defining polytypic functions
–datatype level

- Using inn and out together with the pattern functor classes, we can define polytypic functions over regular datatypes

```
pmap :: (FunctorOf f d, P_fmap2 f) ⇒ (a → b) → (d a → d b)
pmap f = inn . fmap2 f (pmap f) . out

cata :: (FunctorOf f d, P_fmap2 f) ⇒ (f a b → b) → (d a → b)
cata φ = φ . fmap2 id (cata φ) . out

ana :: (FunctorOf f d, P_fmap2 f) ⇒ (b → f a b) → (b → d a)
ana ψ = inn . fmap2 id (ana ψ) . ψ
```

A more complex function

- transpose (or zip) takes (for instance) a list of trees to a tree of lists

```
pzipWith :: .. ⇒ (a → b → c) → d a → d b → d c
separate :: .. ⇒ d a → (d (), [a])
combine  :: .. ⇒ d () → [a] → d a

listTranspose :: .. ⇒ [d a] → d [a]
listTranspose (x : []) = pmap singleton x
listTranspose (x : xs) = pzipWith (:) x (listTranspose xs)

transpose :: .. ⇒ d (e a) → e (d a)
transpose x = pmap (combine s) (listTranspose l)
   where (s,l) = separate x
```

Why the dots?

```
tranpose ::
    ( FunctorOf f d
    , FunctorOf g e
    , P_fmap2 f
    , P_fmap2M f
    , P_fflatten f
    , P_fmap2 g
    , P_fprop g
    , P_fzip g
    ) ⇒ d (e a) → e (d a)
transpose = ...
```

Non–regular datatypes

- Nested types

```
data Bin a = Leaf a | Branch (Bin (a,a))

type BinF  = Par :+: Rec :*: Rec
instance FunctorOf BinF Bin where ...
```

Note: inn is a partial function

- Abstract datatypes

```
type Stack a -- abstract

push  :: a → Stack a → Stack a
pop   :: Stack a → Maybe (a,Stack a)
empty :: Stack a

type StackF = Empty :+: Par :*: Rec
instance FunctorOf StackF Stack where ...
```

Preprocessor support

- Three tiresome things:
  - FunctorOf instances
  - Pattern functor classes
  - Class constraints
- Solution: The PolyP preprocessor
  - Generates FunctorOf instances
  - Provides convenient syntax for pattern functor classes
  - Infers class constraints

Pattern functor class syntax

```
polytypic fmap2 :: (a → c) → (b → d) → (f a b → f c d)
  = λp r → case f of
      g :*: h → λ(x :*: y) → fmap2 p r x :*: fmap2 p r y
      Empty   → λEmpty → Empty
      ...
```

```
class P_fmap2 f where
   fmap2 :: (a → c) → (b → d) → (f a b → f c d)

instance (P_fmap2 f, P_fmap2 g) ⇒ P_fmap2 (f :+: g) where
   fmap2 p r = λ(x :*: y) → fmap2 p r x :*: fmap2 p r y

instance P_fmap2 Empty where
   fmap2 p r = λEmpty → Empty
...
```

Related work

- Generic Haskell (Jeuring, Löh, et.al)
  - All datatypes (of all kinds)
  - Requires preprocessing
- Strafunski (Lämmel, Visser)
  - Only kind $*$
- GHC Generics (Lämmel, Peyton Jones)
  - Fully integrated in GHC
  - Only kind $*$

Conclusions

- Polytypic functions directly in Haskell
  - Examples: pmap, cata, transpose
  - Using the class system
  - Only for regular (and some non–regular) datatypes
- Possible future work
  - Mutually recursive datatypes
  - Implement PolyP as a Template Haskell library