More polymorphism!

Osλo Haskell, April 2015

Alexander Berntsen, plaimi, 2015

Introduction

More polymorphism!

- Two main kinds of polymorphism used in Haskell.
- Parametric & ad-hoc polymorphism.
- Generic ADTs & functions "solve" parametric polymorphism.
- Typeclasses are a form of constrained polymorphism which in Haskell "solve" ad-hoc polymorphism.

More polymorphism!

Osλo Haskell, April 2015

- Parametric polymorphism: Operate on values independently of their type.

```
length          :: [α] → Int
length []       = 0
length (_:xs)   = 1 + length xs
```

- length only cares about the shape! There are plenty of functions like this in Prelude – try looking through it & identifying some of them!

More polymorphism!

- Ad-hoc polymorphism: Operate on values of different types.
- The perhaps most common need for ad-hoc polymorphism: avoiding the need for separate addInt & addFloat functions.
- Solved by typeclasses in Haskell.

More polymorphism!

Osλo Haskell, April 2015

- Typeclasses is a form of constrained polymorphism (AKA bounded qualification).

- class Num α where
  (+), (-), (*) :: Num α => α → α → α

- The functions are constrained to work on any type α which has a Num instance.

More polymorphism!

Osλo Haskell, April 2015

Bit bigger example of where ad-hoc polymorphism is useful...

- class Visible α where
  render :: α → Picture

- instance Visible Board where
  render (Board bs) = Pictures (map render bs)

- instance Visible Brick where
  render b = Color (mixColors 1.0 (health b) (colour b) white
  $ uncurry Translate (centre b)
  $ rectangleSolid (width b) (height b)

Semigroups & Monoids

Semigroups & Monoids

- Two extremely simple typeclasses.
- Semigroups are a set of stuff with a binary operation that combines the stuff.
- A Monoid is a Semigroup with an element for which their binary operation is id.

Semigroups

- class Semigroup α where
  (<>) :: α → α → α
- Associative binary operation.
- Examples: addition & multiplication of numbers, conjuction & disjunctions of booleans

Monoids

- Simple API. Just the identity, and the binary operation for combining things.
- class Monoid α where

```
  mempty  :: α              -- Identity
  mappend :: α → α → α  -- Binary operation
```

Monoids

- Some really simple examples:

- Numbers under addition: 42 + 0 = 42
- Numbers under multiplication: 42 * 1 = 42

- [4] ++ [2] is the same as mappend [4] [2]
- [42] ++ [] is the same as mappend [42] mempty

Monoids

- Let's say you have some ByteStrings which you want to turn into Text.

- ```
  f :: (T.Text → T.Text) → T.Text → T.Text → T.Text
  f g x y = g x `T.append` g y    -- annoying to "port"!
  ```

- ```
  f :: Monoid m => (m → m) → m → m → m
  f g x y = g x <> g y             -- no "porting" necessary!
  ```

-> polymorphism

-> polymorphism

- So far we've seen * polymorphism.
- By *, we mean "term level" values.
- We've seen functions that operate on data of different types. length doesn't care if you have a list of integers or a list of wibbles.

-> polymorphism

- Now we're going to take a look at things that don't even care if it's a list of things!
- Lists are after all just a shape. What if you have a tree? Sometimes you don't care if you have a list of wibbles or a tree of wibbles, or indeed a wobble of wibbles!

-> polymorphism

- But first – what does * and *->* really mean?? * is a concrete type, whilst *->* is incomplete.
- Here are some example of things of kind *:
- 42 :: Int :: *
- [42] :: [Int] :: *
- Just [42] :: Maybe [Int] :: *

-> polymorphism

- Now let's see some things that are kind *->*:

- Maybe :: *->*
- [] :: *->*

- Aha... Interesting... Hmm... So how do we use this?

- Allow me to demonstrate.

Functors, Applicatives, & Monads

Functor

- Arguably the most ubiquitous typeclass in Haskell.
- Represents a computational context & a way to operate on whatever data is in this context.
- Basically used for any type that can be "mapped over".

Functor

- The API is very simple!
- class Functor φ where
  fmap :: (α → β) → φ α → φ β
- Just a way to apply α → β on φ α and get φ β.

Functor

```
instance Functor [] where
  fmap _ []        = []
  fmap f (x:xs)    = f x : fmap f xs
-- Yes, this is just Prelude.map!

instance Functor Maybe where
  fmap _ Nothing   = Nothing
  fmap f (Just a)  = Just (f a)
```

Functor

Lists

```
fmap (+1) [1, 2, 3] -- [2, 3, 4]
fmap (+1) []        -- []
```

Maybes

```
fmap (+1) (Just 1)  -- Just 2
fmap (+1) Nothing   -- Nothing
```

Applicative Functor

- A bit more specialised than Functor.
- Useful for certain effective computations.
- Lets us apply a function in a computational context to values in the same context.

Applicative Functor

- Another simple API!

- class Functor φ => Applicative φ where
  pure :: α → φ α
  (<*>) :: φ (α → β) → φ α → φ β

- Apply φ (α → β) on φ α and get φ β.

Applicative Functor

- So for lists, the instance looks like this:
- instance Applicative [] where

```
    pure            = (:[])
    f:fs <*> xs     = fmap f xs ++ (fs <*> xs)
    [] <*> _        = []
```

Applicative Functor

```
pure 42 :: [Int]                               -- [42]

pure (*2)   <*> pure 21                        -- [42]
pure (*)    <*> [2, 20] <*> pure 21            -- [42, 420]

[(*2), (*20)] <*> pure 21                      -- [42, 420]
[(*2), (*20)] <*> [21, 42]                     -- [42, 84, 420, 840]

pure (*) <*> [2, 20] <*> [2, 20]               -- [42, 84, 420, 840]
```

Applicative Functor

- Combining the Applicative API with the Functor API, we get what is called "applicative style" programming.
- (<$>) = fmap -- Applicative style operator fmap
- f <$> a <*> b

Applicative Functor

- `(*) <$> [2, 20] <*> pure 21  -- [42, 420]`
- `(*) <$> [2, 20] <*> [21, 42]  -- [42, 84, 420, 840]`
- And so on.

Monad

- A bit more specialised than Applicative.
- Useful for computational contexts which may be composed sequentially.
- Lets us apply a function that takes a regular value, and returns a value in a context, to values in the same context.

Monad

- The API is, of course, very simple!

- class Applicative μ => Monad μ where

```
return   :: α → μ α
(>>=)    :: μ α → (α → μ β) → μ β
(>>)     :: μ α → μ β → μ β
m >> n   = m >>= λ_ → n
```

Monad

- instance Monad Maybe where

```
  return             = Just
  Just x    >>= f    = f x
  Nothing   >>= _    = Nothing
```

Monad

- `return 42 :: Maybe Int -- Just 42`
- ```
  doubleIfPos x
    | x > 0      = Just (x * 2)
    | otherwise  = Nothing
  ```
- `Just 21   >>= doubleIfPos -- Just 42`
- `Nothing   >>= doubleIfPos -- Nothing`
  ```

do-notation

- Haskell is the world's finest imperative programming language.
- do-notation is a convenient and nice way to program with monads.
- Monads are programmable semicolons!

do-notation

- a >>= λx → b x >>= λy → c y -- tedious to chain!
- do

```
  x ← a
  y ← b x
  c y
  -- Ahh... much nicer!
```

do-notation

- Side-by-side:

```
a    >>= λx →                      do  x ← a
b x  >>= λy →                          y ← b x
c y                                    c y
```

do-notation

- Bit more practical example:

- do

```
  x ← Just 4         -- x is now 4
  y ← Just 2         -- y is now 2
  let z = x+y        -- z = 6
  return (z*(z+1))   -- Just 42
```

do-notation

- Currently only for Monad.
- Applicative-do: coming soon to a GHC near you!

Monad comprehensions

- Another cute syntax sugar for monads.
- A lot like set comprehension from maths, if you're into that kind of stuff.
- Usually used for lists (called list comprehensions).

Monad comprehensions

- [z*(z+1)
  |x ← Just 4, y ← Just 2, let z = x+y] -- Just 42

- [x+y | x ← [1, 2], y ← [10, 100]] -- [11,101,12,102]

Monad comprehensions

- We can now make the list applicative a bit nicer:

```
(f:fs) <*> xs   = fmap f xs ++ (fs <*> xs)
fs <*> xs       = [f x | f ← fs, x ← xs]
```

Foldable & Traversable

Foldable

- Functor is for things which may be mapped over, Foldable is for things which may be folded up.
- Folding is also known as reducing, injecting, and various other things in other languages.
- But WTF is folding anyway? Allow me to explain...

Foldable

- Remember length?

- length :: [α] → Int
- length [] = 0
- length (_:xs) = 1 + length xs

- This is actually a variation of a super common pattern!

Foldable

Consider these:

```
sum           :: [Int] → Int
sum []        = 0
sum (x:xs)    = x + sum xs

reverse         :: [α] → [α]
reverse []      = []
reverse (x:xs)  = reverse xs ++ [x]
```

Foldable

```
(++)             :: [α] → [α] → [α]
[] ++ ys         = ys
(x:xs) ++ ys     = x : xs ++ ys

filter                :: (α → Bool) → [α] → [α]
filter p []           = []
filter p (x:xs)
    | p x             = x : filter p xs
    | otherwise       = filter p xs
```

Foldable

And what about these chaps?

- map :: (α → β) → [α] → [β]
- map _ [] = []
- map f (x:xs) = f x : map f xs

- id :: [α] -> [α]
- id x = x

Foldable

- These are all just variations on a theme encapsulated by what we call a fold.

- foldr :: (α → β → β) → β → [α] → β
- foldr _ z [] = z
- foldr f z (x:xs) = f x (foldr f z xs)

Foldable

```
length      = foldr (const (1 +)) 0
sum         = foldr (+) 0
reverse     = foldr (flip (++) . return) []
xs ++ ys    = foldr (:) ys xs
filter p    = foldr (λx xs →
                          if p x then x : xs else xs) []
map f       = foldr ((:) . f) []
id          = foldr (:) []
```

Foldable

- So that's how you fold lists.
- But, surely!, you won't be surprised to learn that lists aren't all that special.
- We can fold anything that has a Foldable instance!

Foldable

- The API looks a bit more complicated this time around.
- Don't be scared though! To make a Foldable, you only need to implement <u>one</u> method! You get the others for free.
- Your pick between foldMap and foldr.

Foldable

- class Foldable τ where

```
fold      :: Monoid μ => τ μ → μ
foldMap   :: Monoid μ => (α → μ) → τ α → μ
foldr     :: (α -> β -> β) -> β -> τ α -> α
foldl     :: (α -> β -> α) -> α -> τ β -> α
foldr1    :: (α -> α -> α) -> τ α -> α
foldl1    :: (α -> α -> α) -> τ α -> α
```

Foldable

```
instance Foldable Maybe where
  foldr _ z Nothing     = z
  foldr f z (Just x)    = f x z

instance Foldable ((,) a) where
  foldr f z (_, y) = f y z
```

Traversable

- Traversable represents data structures which can be traversed while preserving the shape.
- A Foldable Functor that lets us commute two functors.

Traversable

- Like Foldable, what initially looks to be a semi-complicated API.
- Like Foldable, only one function needs to be implemented. Pick between sequence or traverse.

Traversable

- class (Functor τ, Foldable τ) => Traversable τ where

```
traverse  :: Applicative φ  => (α → φ β) → τ α → φ (τ β)
sequenceA :: Applicative φ  => τ (φ α) → φ (τ α)
mapM      :: Monad μ        => (α → μ β) → τ α → μ (τ β)
sequence  :: Monad μ        => τ (μ α) → μ (τ α)
```

Traversable

```haskell
instance Traversable Maybe where
  traverse _ Nothing = pure Nothing
  traverse f (Just x) = Just <$> f x
```

```haskell
instance Traversable ((,) a) where
    traverse f (x, y) = (,) x <$> f y
```

Traversable

- sequence [Just 42] -- Just [42]
- sequence Just [42] -- [Just 42]

Traversable

```
doubleIfPos <$> [-5, 21]                    -- [Nothing, Just 42]
doubleIfPos <$> [21, 42]                    -- [Just 42, Just 84]

doubleIfPos `traverse` [-5, 21]             -- Nothing
doubleIfPos `traverse` [21, 42]             -- Just [42, 84]

sequence $ doubleIfPos <$> [-5, 21]         -- Nothing
sequence $ doubleIfPos <$> [21, 42]         -- Just [42, 84]
```