An Introduction to Haskell Typeclasses

Lucian Mogosanu

02.07.2014

Haskell Types: a recap (1)

Algebraic Data Types

- Sum types:

```haskell
data SumType a b = C1 a | C2 b -- Either a b
```

- Product types:

```haskell
data ProductType a b = MkP a b -- (a, b)
```

Haskell Types: a recap (1)

Algebraic Data Types

- Sum types:

```
data SumType a b = C1 a | C2 b -- Either a b
```

- Product types:

```
data ProductType a b = MkP a b -- (a, b)
```

- Any combination between the two

```
data MyNonsensicalDataType =
    C1 String Int Bool
  | C2 Int Char (Double, Double) (Maybe Bool) String
```

Haskell Types: a recap (2)

- Commonly used types:

```
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
-- syntactic sugar ahead
data [] a = [] | a : ([] a) -- [a]
data (,) a b = (,) a b -- (a, b)
```

Typeclasses: constraints

```
(+) :: Num a => a -> a -> a
(==) :: Eq a => a -> a -> Bool
show :: Show a => a -> String
(>) :: Ord a => a -> a -> Bool
```

Typeclasses: definition

- Interfaces for types sharing a common property
- Example: `Float` and `Int` are both `Num`s
- Example: `Show`

```
class Show a where
    show :: a -> String
```

Typeclasses: definition (2)

- Each type has its own implementation
- **Ad-hoc polymorphism**

```
instance Show MyType where
    ...
```

Ad-hoc Polymorphism: defining instances

- Want to convert type to `String`
- `toString` in Java

```
class Show a where
    show :: a -> String

data Color = Red | Green | Blue deriving Show

data Container a = Empty | Holding a

instance Show (Container Int) where
    show (Holding x) = "[ " ++ show x ++ " ]"
    show _ = "[ ]"
```

Common Typeclasses: Num

```
class Num a where
    (+), (-), (*) :: a -> a -> a
    negate :: a -> a
    abs :: a -> a
    signum :: a -> a
    fromInteger :: Integer -> a

    x - y = x + negate y
    negate x = 0 - x
```

- Define all but one of `(-)` and `negate`

Common Typeclasses: Ord

```
class Eq a => Ord a where
    -- minimal complete definition:
    -- compare or <=
    compare :: a -> a -> Ordering
    (<), (<=), (>), (>=) :: a -> a -> Bool
    max, min :: a -> a -> a

data Ordering = LT | EQ | GT
```

- Example: sorting

```
sort :: Ord a => [a] -> [a]
```

Hands-on

```haskell
data Person = P
  { pnc :: String -- Int?
  , name :: String
  , age :: Int
  , married :: Bool
  } deriving Show
```

- Eq instance for Person

```haskell
instance Eq Person where
  p1 == p2 = pnc p1 == pnc p2
```

Hands-on (2)

- Parameterize pnc

```
data Person2 a = P2
  { pnc2 :: a
  , name2 :: String
  , age2 :: Int
  , married2 :: Bool
  } deriving Show
```

- Eq instance for `Person2`

```
instance Eq (Person2 a) where
  p1 == p2 = pnc2 p1 == pnc2 p2
```

Hands-on (3)

```
instance Eq a => Eq (Person2 a) where
  p1 == p2 = pnc2 p1 == pnc2 p2
```

Digression

Types have types!

- **kind**: the type of a type

```
> :k Int
Int :: *
> :k Maybe
Maybe :: * -> *
> :k Maybe Int
Maybe Int :: *
```

- Useful for describing parametric types
- Haskell code should have a well-formed type
  - i.e. having the kind $*$

Typeclasses for parametric types

- Parametric types can be polymorphic too
- Example: What is the type of **all** "containers"?

Typeclasses for parametric types

- Parametric types can be polymorphic too
- Example: What is the type of **all** “containers”?

```
container :: c a
```

Typeclasses for parametric types (2)

- We want a `map`-like operation on trees
- We want a `map`-like operation on containers
- We want a function `f :: (a -> b) -> c a -> c b` where `c` is a container-type
  - Note: `c` has kind `* -> *`

Functors

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b

instance Functor [] where
    fmap = map
```

Functors (2)

- Functors are mathematical objects!

```
fmap :: (a -> b) -> (f a -> f b)
```

- `fmap` "lifts" a function to the (parametric) type `f`

Hands-on Functors

```haskell
data Container a = Empty | Holding a
  deriving (Show, Eq)
```

- Define `fmap` for `Container`
- Definition is identical for `Maybe`

Custom typeclasses

- Source: `learnyouahaskell.com`

```
class YesNo a where
  yesno :: a -> Bool
```

- Define `yesno` for the following types:
  - `Bool`
  - `Int`
  - `[a]`
  - `Maybe a`

Monads (0)

"a monad is a monoid in the category of endofunctors, what's the problem?"

Monads (1)

- Functors map functions to containers
- Monads generalize computations over containers

Monads (2)

```haskell
class Monad m where
    (>>=)    :: m a -> (a -> m b) -> m b
    return   :: a -> m a
```

- `(>>=)` is also called **bind**
- `return` is called **return** (for some reason)

Maybes are Monads (1)

- Getting some intuition:

```
return :: a -> Maybe a
(>>=)  :: Maybe a -> (a -> Maybe b) -> Maybe b
```

- Definition(s) for `return`
- Definition(s) for `(>>=)`

Maybes are Monads (2)

- Getting some intuition:

```
return :: a -> Maybe a
(>>=)  :: Maybe a -> (a -> Maybe b) -> Maybe b
```

- `return`: put a value in the container
- `(>>=)`:
  - extract value from container, `Maybe a`
  - perform computation, `a -> Maybe b`
  - return result, `Maybe b`

Maybes are Monads: hands-on (1)

```
> let f x = Just $ 2 + x
> Just 3 >>= f
> Nothing >>= f
```

- Advantage: can easily chain computations

```
> Just 3 >>= (Just . (2 +)) >>= (Just . (* 5))
```

Maybes are Monads: hands-on (2)

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

- Can rewrite previous chain:

```
> return 3 >>= return . (2 +) >>= return . (* 5)
```

Maybes are Monads: hands-on (3)

- Binding looks similar to imperative programing
- Do-notation:

```
computation :: Num a => Maybe a -> Maybe a
computation m = do
  x <- m -- extract value, if it exists
  y <- return $ 2 + x
  z <- return $ y * 5
  return z
```

- Note: `return` doesn't “end” the function!

Maybes are Monads: hands-on (4)

- Cleaner style:

```haskell
computation :: Num a => Maybe a -> Maybe a
computation m = do
  x <- m -- extract value, if it exists
  let y = 2 + x -- m is the only possible failure
  return $ y * 5
```

Other Monads

- Lists
- Functions
- Global state/“the outside world”
  - `State`
  - `IO`

Extra: Limitations of Haskell's Type System

- Type inference automatically finds/checks the right type for us
- . . . because all computations on types terminate,
- . . . so Haskell types are not first-class values.

Extra: Limitations of Haskell's Type System (2)

- Example 1: prove that `map` preserves list length
- Example 2: restrict a type to a subset of its values (remember `Nat`)
- Alternative: **dependent types** (Agda, Idris)
  - Types that depend on values
  - Trade-off: they make type checking more difficult

Extra: Type Systems vs. Real World

- Strongly-typed programs, **sound** type system
- Translated into **weakly-typed** machine code
- Type erasure: deleting types **preserves** program semantics
  - Assuming the compiler is correct