Haskell full of Buzzwords

Martin Heuschober

metalab.at/wiki/Lambdaheads

24. Mai 2013

Inhalt

Haskell

- functional
- lazy
- statically, strongly typed
- abstract ;-)

a bit of syntax

```haskell
import Blah as Blubb hiding (foo, Foo(..))
type MyFoo = MyFoo1 |MyFoo2 | MyFoo3 {t :: MyFoo, g :: MyFoo}
newtype Bar = Bar {runBar :: b -> (a,b)}
```

a and b are type variables and a main ingredient in Haskell code, e.g. when writing type annotations to functions

```haskell
foo1 :: a -> [a]
foo1 x = [x] -- the list with a single element namely x
```

using a point-free style which is ubiquitous in Haskell code we can formulate this equivalently as

```haskell
foo2 :: a -> [a]
foo2 = (:[])
```

here ‘$(:)$’ is the cons operator and ‘$[]$’ denotes the empty-list

Polymorphism

Haskells way to express polymorphism is via type-classes

```
class Fooable a where -- not the Java kind of classes
    foo :: [a] → (a → a) → a
```

but similar to Java interfaces (or so I've heard). If something wants to be '$Fooable$' it has to implement this function 'foo', which takes a list of 'as' and a function 'f' and generates something of type 'a'.

Polymorphism

We can create instances for type-classes, unless there is already one defined (we will see that for the '*Applicative*' type-class for lists later on).

```
instance Foo [] where
    foo [] _ = undefined -- should never happen ;-)
    foo (x : xs) f = f x
```

Here *undefined* is a special function, which always type-checks but generates a, run-time error if invoked, useful for getting a structure. Another prominent technique can be seen in the last line - where the list is decomposed in *head* and *tail*.

Functors - Applicative Functors - Monads

Beginning from this section, all code presented is executable with your favourite Haskell compiler, feel free to load this file into $ghci$ and make your own experiments.

Abstraction

As one of Haskell's features we listed abstraction, which is only one side of the medal it is also really hard to get your head around.
In Haskell we have a great type system and one feature of it is type constructors which take a more basic type and produce a new type. Most prominent the list type $[a]$, but there are also $Trees$, $Vectors$, $Matrices$, $Tries$ and much more (including the kitchen sink). And one thing is we want to modify the values in such a "container". This lead to the discovery of $Functors$, originating from the rather obscure mathematical branch of category theory.

More Abstraction

Another problem was how to do chain stateful or even worse actions with side effects together. The first idea was to use continuation-passing style, but being involved with category theory already, Phil Wadler came up with the term of $Monads$. Later on concepts like $ApplicativeFunctors$, $Arrows$ and much more were added

Setup

```
import Prelude hiding (Functor, Monad, Maybe(..), fmap, (>>=),(
```

We start with importing the standard library *Prelude*, where we hide all operators, which we will define later on ourselves.

Functors

The first solution we see is the class *Functor*

```
class Functor f where
    fmap :: (a → b) → f a → f b
```

And though Haskell cannot enforce it, every instance of *Functor* should satisfy these two laws:

$$\mathrm{fmap}(\mathrm{id}) = \mathrm{id}$$
$$\mathrm{fmap}(g) \circ \mathrm{fmap}(h) = \mathrm{fmap}(g \circ h)$$

Functor - examples

The first and most obvious Functor we have is the list type

```
instance Functor [] where
    fmap = map
```

Another example is the type of $Maybe$, which indicates a state of failure or success

```
data Maybe a = Nothing | Just a
```

and we make it an instance of $Functor$ by

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

profing the functor laws for Maybe

TODO

Intro

A more recent development (than functors and monads) is the class of $ApplicativeFunctors$ or short $Applicatives$. Which came from the need of applying functions from inside a "container". Let's say we have a list of functions $[f_1, ..., f_9]$ and want to apply these to another list of $[1...3]$. If we only have plain old $Functor$ no way we can do that - so we define:

Applicative Functors

```haskell
class (Functor f) => Applicative f where
    pure  :: a -> f a
    (<*>) :: f (a -> b) -> (f a -> f b)
```

Note that the first function could also have the names return, singleton, unit point. We will name the second function apply, which gave this type-class its name. As we have the class constraint of f being a $Functor$ we introduce the symbol of

```haskell
(<$>) :: (Applicative f) => (a -> b) -> (f a -> f b)
(<$>) = fmap
```

which leads to a more readable code, if you have gotten used to it.

Again examples

```haskell
instance Applicative [] where
    pure x = [x]
    fs <*> xs = [f x | f <- fs, x <- xs]
```

And for the *Maybe* type

```haskell
instance Applicative Maybe where
    pure a = Just a
    (Just f) <*> (Just a) = Just (f a)
```

another list instance

To prevent clashes from happening we define:

```
newtype ZipList a = ZipList { getZipList : : [a] }
```

this is a "tabula rasa" version of the type $[a]$, all associated instances are forgotten. So we have to make $ZipList$ an instance of $Functor$ by

```
instance Functor ZipList where
    fmap f (ZipList zs) = ZipList (map f zs)
```

and an instance of $Applicative$

```
instance Applicative ZipList where
    pure z = ZipList (repeat z)
    ZipList fs <*> ZipList zs = ZipList (zipWith ($) fs zs)
```

Applicative Law

Again we have a law for the Applicative type-class

$$f < \$ > xs = (\text{pure}\, f) < * > xs$$

and of course we have all the previous laws for functors.

Some extra functions

The module *Control.Applicative* provides the following helper functions: A variant of $<*>$ with the arguments reversed.

```
(<**>) :: Applicative f ⇒ f a → f (a → b) → f b
(<**>) = liftA2 (flip ($))
```

Lift a function, this function may be used as a value for fmap in a *Functor* instance.

```
liftA :: Applicative f ⇒ (a → b) → f a → f b
liftA f a = pure f <*> a
```

Lift a binary function to actions.

```
liftA2 :: Applicative f ⇒ (a→b→c)→f a→f b→f c
liftA2 f a b = f <$> a <*> b
```

and furthermore `liftA3` and `optional`.

And now for the juicy part

Functor and Applicative are clear so far ??

Intro

Monads were brought to solve the problem of IO in Haskell - though before that people used a continuation-passing style to chain actions after one another. Btw it turns out there is a Monad called *Cont*, the continuation monad, which has some universal property, but unfortunately I had no time to investigate that - maybe more the next time about that.
From a compiler's point of view a monad is nothing more than a type class, but as monads are ubiquitous in Haskell code, almost every program has some part in the *IO* Monad, there is some syntactic sugar provided - the so called *do* notation. Which we will meet in a few slides.

Monads

The most ~~famous~~ dreaded concept when learning Haskell

```
class Monad m where
    return :: a -> m a
    (>>=)  :: m a -> (a -> m b) -> m b -- bind
    (>>)   :: m a -> m b -> m b
    f >> b = f >>= (\_ -> b)
    fail :: String -> m a
```

I will show that every monad is an applicative, which could be included in the class definition as a constraint, but the use of monads pre-dates the use of both functors and applicatives, so one came to the definition above. Note that the definition of the function $(>>)$ is optional as it can be implemented by using $(>>=)$ it can be thought of as follows: the side effects of f are executed whilst the result is thrown away.

An example - Please

The *Maybe* Monad:

```
instance Monad Maybe where
    return = Just -- point free style
    Nothing >>= f = Nothing
    Just x >>= f = f x
    fail _ = Nothing
```

and applied

```
Just 3 >>= (λx→Just(x+3)) >>= (λx → Just (y*3))
> Just 12
Nothing >>= (λx→Just(x+3)) >>= (λx → Just (y*3))
> Nothing
Just 3 >>= (λx→Just(x+3)) >>= (λx→ Just (x*3) >> return x)
>  Just 6
```

Do-Notation

As one can see the examples above are a bit to work your head around - so to make Haskell a bit more beginner-friendly the Do-Notation was introduced, this is Especially useful within the IO monad.

```
iofunction  : :  IO ()
iofunction = do putStrLn "enter your name"
                a ← getLine
                let caps = a++"!"
                putStrLn ("Hello my friend " ++ caps)
```

Do-Notation

and we have the basic transformations

$$
\begin{aligned}
\mathrm{do}\, e &\longrightarrow e \\
\mathrm{do}\{e; stmts\} &\longrightarrow e >> \mathrm{do}\{stmts\} \\
\mathrm{do}\{v \leftarrow e; stmts\} &\longrightarrow e >>= \lambda v \rightarrow \mathrm{do}\{stmts\} \\
\mathrm{do}\{\mathrm{let}\, decls; stmts\} &\longrightarrow \mathrm{let}\, decls \,\mathrm{in}\, \mathrm{do}\{stmts\}
\end{aligned}
$$

Do-Notation

so the function above could be alternatively written in the form

```
iofunction' :: IO ()
iofunction' = putStrLn "enter your name" >>
              getLine >>= λa →
              let caps = a++"!" in
              putStrLn ("Hello my friend " ++ caps)
```

Note: actually this code is not working - remember we have hidden (») and (»=)

State - example

All of the code in this example will be available seperately

```
newtype State s a = State {runState :: s -> (a, s)}
```

and we can make *State* an instance of *Monad*

```
instance Monad (State s) where
    return x = State $ \s -> (x,s)
    (State h) >>= f = \s -> let (a, newState) = h s
                                (State g)     = f a
                            in  g newState
```

from `learnyouahaskell.com` we take the example. Of a Stack State

State - example

and

```
type Stack = [Int]
```

for such a stack we can define operations

```
pop :: State Stack Int
pop = state $ λ(x : xs) → (x, xs)

push :: Int → State Stack ()
push a = state $ λxs → ((),a : xs)
```

State - example

and

```
stackManip :: State Stack Int
stackManip = do push 3
                pop
                pop
```

or equivalently

```
stackManip' :: State Stack Int
stackManip' = push 3 >> pop >> pop
```

Monad laws

Monad has the following laws

$$return\ a >>= f = fa$$
$$m >>= return = m$$
$$m >>= (\lambda x \to kx >>= h) = (m >>= k) >>= h$$
$$fmap\ f\ xs = xs >>= return \circ f = liftM\ f\ xs$$

Monad laws

or with a helper function

```
(>=>) :: (Monad m) ⇒ (a → m b) → (b → m c) → a → m c
(f >=> g) x = return x >>= g >>= f
```

$$return\ a >=> f = f$$

$$f >=> return = f$$

$$(f >=> g) >=> h = f >=> (g >=> h)$$

Arrows

we will see next(?) time

Bibliography

- `www.learnyouahaskell.com`
- Typeclassopedia - Brent Yorgey (in TheMonadReader 13)
- www.haskell.org