Functional Programming for Logicians - Lecture 6

Beyond Haskell

Malvin Gattinger

21 January 2021

Lecture 6: Beyond Haskell

Haskell extras

User interfaces

Other Functional Languages

Functional Style

Haskell extras

Language Extensions

```haskell
{-# LANGUAGE
InstanceSigs,
BangPatterns,
ForeignFunctionInterface,
OverloadedStrings,
TemplateHaskell
#-}

module L6 where

import           Data.FileEmbed
import qualified Data.Text as T
import qualified Data.Text.Encoding as E
import qualified Data.Text.Lazy as TL
import           Foreign.C
import           Foreign.Ptr (Ptr,nullPtr)
import           Web.Scotty
```

https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/exts.html

Language Extension: InstanceSigs

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
  deriving (Eq,Ord,Show)
```

Remember that we had to put these in comments:

```
instance Functor Tree where
  -- fmap :: (a -> b) -> Tree a -> Tree b
  fmap f (Leaf x)            = Leaf (f x)
  fmap f (Branch left right) = Branch (fmap f left)
                                      (fmap f right)
```

Language Extension: InstanceSigs

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
  deriving (Eq,Ord,Show)
```

Remember that we had to put these in comments:

```
instance Functor Tree where
  -- fmap :: (a -> b) -> Tree a -> Tree b
  fmap f (Leaf x)              = Leaf (f x)
  fmap f (Branch left right) = Branch (fmap f left)
                                       (fmap f right)
```

The `InstanceSigs` extension allows this:

```
instance Functor Tree where
  fmap :: (a -> b) -> Tree a -> Tree b
  fmap f (Leaf x)              = Leaf (f x)
  fmap f (Branch left right) = Branch (fmap f left)
                                       (fmap f right)
```

Language Extension: BangPatterns

Recall that Haskell by default is lazy:

```
lazyAnd :: Bool -> Bool -> Bool
lazyAnd p q = p && q
```

```
λ> lazyAnd False undefined
False
```

Language Extension: `BangPatterns`

Recall that Haskell by default is lazy:

```
lazyAnd :: Bool -> Bool -> Bool
lazyAnd p q = p && q
```

```
λ> lazyAnd False undefined
False
```

A *bang pattern* makes a function *strict* in this argument:

```
strictAnd :: Bool -> Bool -> Bool
strictAnd !p !q = p && q
```

```
λ> strictAnd False undefined
*** Exception: Prelude.undefined
```

Language Extension: `BangPatterns` (continued)

NOTE: The '`!`' only evaluates to "weak head normal form".

For lists, this does not mean that we compute all elements!

Language Extension: BangPatterns (continued)

NOTE: The '!' only evaluates to "weak head normal form".

For lists, this does not mean that we compute all elements!

```haskell
myNumbers :: [Integer]
myNumbers = [1..]

myf :: [Integer] -> [Integer]
myf !xs = filter odd xs
```

Language Extension: BangPatterns (continued)

NOTE: The '!' only evaluates to "weak head normal form".

For lists, this does not mean that we compute all elements!

```
myNumbers :: [Integer]
myNumbers = [1..]

myf :: [Integer] -> [Integer]
myf !xs = filter odd xs
```

```
λ> take 10 (myf myNumbers)
[1,3,5,7,9,11,13,15,17,19]
```

Language Extension: BangPatterns (continued)

NOTE: The ‘!’ only evaluates to “weak head normal form”.

For lists, this does not mean that we compute all elements!

```
myNumbers :: [Integer]
myNumbers = [1..]

myf :: [Integer] -> [Integer]
myf !xs = filter odd xs
```

```
λ> take 10 (myf myNumbers)
[1,3,5,7,9,11,13,15,17,19]
```

See https://wiki.haskell.org/Weak_head_normal_form

FFI: Foreign Function Interface

We can call C functions from Haskell!

```haskell
-- pure function
foreign import ccall "sin" c_sin :: CDouble -> CDouble
sine :: Double -> Double
sine d = realToFrac (c_sin (realToFrac d))

-- impure function
foreign import ccall "time" c_time :: Ptr a -> IO CTime
getTime :: IO CTime
getTime = c_time nullPtr
```

Example from https://wiki.haskell.org/FFI_complete_examples

FFI: Foreign Function Interface

We can call C functions from Haskell!

```
-- pure function
foreign import ccall "sin" c_sin :: CDouble -> CDouble
sine :: Double -> Double
sine d = realToFrac (c_sin (realToFrac d))

-- impure function
foreign import ccall "time" c_time :: Ptr a -> IO CTime
getTime :: IO CTime
getTime = c_time nullPtr
```

Example from https://wiki.haskell.org/FFI_complete_examples

More complex example: https://github.com/m4lvin/HasCacBDD

Overloaded Strings

The standard definition

```
type String = [Char]
```

is not very efficient for large amounts of (unicode) text.

Better types and functions are provided by:

- Data.Text
- Data.Text.Lazy

We can `pack` and `unpack` to convert between `String` and `Text`.

Overloaded Strings

The standard definition

```haskell
type String = [Char]
```

is not very efficient for large amounts of (unicode) text.

Better types and functions are provided by:

- Data.Text
- Data.Text.Lazy

We can `pack` and `unpack` to convert between `String` and `Text`.

With the `OverloadedStrings` language extension we can still easily write values of type `Text`:

```haskell
myText :: T.Text
myText = "justSomethingInQuotationMarks"
```

Template Haskell

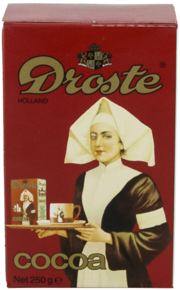
Imagine you want to write many similar functions.

```
plusOne :: Int -> Int
plusOne x = x + 1

plusTwo :: Int -> Int
plusTwo x = x + 2

plusThree :: Int -> Int
plusThree x = x +3
```

Template Haskell

Imagine you want to write many similar functions.

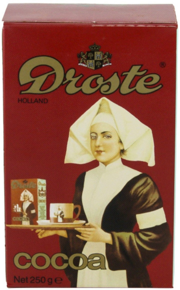
```
plusOne :: Int -> Int
plusOne x = x + 1

plusTwo :: Int -> Int
plusTwo x = x + 2

plusThree :: Int -> Int
plusThree x = x +3
```



Template Haskell: *write Haskell code to generate Haskell code.*

Template Haskell

Imagine you want to write many similar functions.




```
plusOne :: Int -> Int
plusOne x = x + 1

plusTwo :: Int -> Int
plusTwo x = x + 2

plusThree :: Int -> Int
plusThree x = x +3
```

Template Haskell: *write Haskell code to generate Haskell code.*

Concrete example of TH: include a file at compile-time:

```
thisFileContent :: T.Text
thisFileContent = E.decodeUtf8 $(embedFile "L6.lhs")
```

User interfaces

Lexing and Parsing

The problem: our users want to enter

```
(p -> q) & !(p2 <-> q23)
```

instead of

```
Conj (Impl (P "p") (P "q")) (Neg (BiImpl (P "p2") (P "q23")))
```

Lexing and Parsing

The problem: our users want to enter

```
(p -> q) & !(p2 <-> q23)
```

instead of

```
Conj (Impl (P "p") (P "q")) (Neg (BiImpl (P "p2") (P "q23")))
```

We want:

- a lexer to translate the string to a list of tokens
- a parser to translate tokens to something of type `Form`

Lexing and Parsing

The problem: our users want to enter

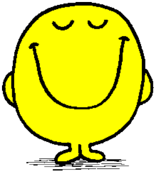
`(p -> q) & !(p2 <-> q23)`

instead of

`Conj (Impl (P "p") (P "q")) (Neg (BiImpl (P "p2") (P "q23")))`

We want:

- a lexer to translate the string to a list of tokens
- a parser to translate tokens to something of type `Form`



The standard Haskell tools for this are *Happy* and *Alex*.

Easy example: https://github.com/da-x/happy-alex-example
Longer example: Lex.x Parse.y from SMCDEL

Web interfaces

One of the easiest ways to make applications in Haskell usable by non-Haskellers and non-programmers is to add a web interface.

Web interfaces

One of the easiest ways to make applications in Haskell usable by non-Haskellers and non-programmers is to add a web interface.

There exist multiple libraries providing different levels of abstraction:

- the easiest: **Scotty**

```
myScotty :: IO ()
myScotty = scotty 3000 $
 get "/" $ do
   html $ mconcat
    [ "<h1>Hello world!</h1>"
    , TL.pack (show $ take 20 $ myf myNumbers) ]
```

More complex example: SMCDEL web interface (source)

Web interfaces

One of the easiest ways to make applications in Haskell usable by non-Haskellers and non-programmers is to add a web interface.

There exist multiple libraries providing different levels of abstraction:

- the easiest: **Scotty**

```
myScotty :: IO ()
myScotty = scotty 3000 $
 get "/" $ do
   html $ mconcat
    [ "<h1>Hello world!</h1>"
    , TL.pack (show $ take 20 $ myf myNumbers) ]
```

More complex example: SMCDEL web interface (source)

- more complex, fairly established: **Yesod**
  https://www.yesodweb.com/
- new and fancy: **IHP: Integrated Haskell Platform**
  https://ihp.digitallyinduced.com/

Other Functional Languages

Dependent Types

This is *not* valid Haskell:

```
repeater :: Int -> a -> ???
repeater 1 x = x
repeater 2 x = (x,x)
repeater 3 x = (x,x,x)
```

The result *type* is not allowed to depend on the input *value*!

Dependent Types

This is *not* valid Haskell:

```
repeater :: Int -> a -> ???
repeater 1 x = x
repeater 2 x = (x,x)
repeater 3 x = (x,x,x)
```

The result *type* is not allowed to depend on the input *value*!

Sometimes polymorphism might look like it allows this, but all polymorphism is resolved at compile-time!

Dependent Types

This is *not* valid Haskell:

```
repeater :: Int -> a -> ???
repeater 1 x = x
repeater 2 x = (x,x)
repeater 3 x = (x,x,x)
```

The result *type* is not allowed to depend on the input *value*!

Sometimes polymorphism might look like it allows this, but all polymorphism is resolved at compile-time!

The most common dependent types are $(\Sigma x : a, bx)$ and $(\Pi x : a, bx)$.

Dependent Types

This is *not* valid Haskell:

```
repeater :: Int -> a -> ???
repeater 1 x = x
repeater 2 x = (x,x)
repeater 3 x = (x,x,x)
```

The result *type* is not allowed to depend on the input *value*!

Sometimes polymorphism might look like it allows this, but all polymorphism is resolved at compile-time!

The most common dependent types are $(\Sigma x : a, bx)$ and $(\Pi x : a, bx)$.

For the current state of "adding dependent types to Haskell", follow Stephanie Weirich:

- Talk "Dependent Types in Haskell" at Strange Loop 2017

  https://youtu.be/wNa3MMbhwS4

- Episode 015 of the CoRecursive podcast (13 June 2018)

  https://corecursive.com/015-dependant-types-in-haskell-with-stephanie-weirich/

Lean

```
structure kripkeModel (W : Type) : Type :=
  (val : W → char → Prop)
  (rel : W → W → Prop)

def evaluate {W : Type} : kripkeModel W → W → formula → Prop
| M w bot          := false
| M w (P c)        := M.val w c
| M w (~ phi)      := not (evaluate M w phi)
| M w (phi ∧ psi) := evaluate M w phi ∧ evaluate M w psi
| M w ([] phi)     := ∀ v : W, (M.rel w v → evaluate M v phi)
```

Lean

```
structure kripkeModel (W : Type) : Type :=
  (val : W → char → Prop)
  (rel : W → W → Prop)

def evaluate {W : Type} : kripkeModel W → W → formula → Prop
| M w bot          := false
| M w (P c)        := M.val w c
| M w (~ phi)      := not (evaluate M w phi)
| M w (phi ∧ psi) := evaluate M w phi ∧ evaluate M w psi
| M w ([] phi)     := ∀ v : W, (M.rel w v → evaluate M v phi)
```

- proof assistant (with a comunity of actual mathematicians using it)
- based on "Propositions as Types": proving is programming is proving!
- includes dependent types

Lean

```
structure kripkeModel (W : Type) : Type :=
  (val : W → char → Prop)
  (rel : W → W → Prop)

def evaluate {W : Type} : kripkeModel W → W → formula → Prop
| M w bot           := false
| M w (P c)         := M.val w c
| M w (~ phi)       := not (evaluate M w phi)
| M w (phi ∧ psi)  := evaluate M w phi ∧ evaluate M w psi
| M w ([] phi)      := ∀ v : W, (M.rel w v → evaluate M v phi)
```

- proof assistant (with a comunity of actual mathematicians using it)
- based on "Propositions as Types": proving is programming is proving!
- includes dependent types

Side note: There seems surprisingly little **Logic** in mathlib?

If you are interested: Malvin is currently trying to translate this old proof of unknown status to Lean: https://malv.in/2020/borzechowski-pdl/.

Elm

```
update : Msg -> Model -> Model
update msg model =
  case msg of
    Increment ->
      model + 1
    Decrement ->
      model - 1
```

Elm

```
update : Msg -> Model -> Model
update msg model =
  case msg of
    Increment ->
      model + 1
    Decrement ->
      model - 1
```

- compiles to JavaScript
- based on “functional reactive programming”
- established the “elm architecture”
- goal: 0 runtime errors ⇒ even more strict than Haskell!

See https://elm-lang.org/, above is the “Buttons” example.

Larger example: https://github.com/RamonMeffert/elm-gossip

Many more languages

- Agda
- Idris
- Scala
- C#
- F#
- ...

Functional Style

Functional Style

- Avoid global variables (and thus global state)!
- Try to write pure functions whenever possible!
- Use data structures that can be `map`ped over etc.

Example: Python

Python has `lambda`, `map` and `filter` too!

Example: Python

Python has `lambda`, `map` and `filter` too!

Example with `lambda`:

```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)
mytripler = myfunc(3)

print(mydoubler(11))
print(mytripler(11))
```

from https://www.w3schools.com/python/python_lambda.asp

Example: C — the horror

Any function may do whatever it wants!

```
int square(int n) {
  // format hard drive here?!
  return n * n;
}
```

Example: C — some good stuff

C has types, including sums and products:

```
type Thing = Either Int String
```

```
data Animal = Cat | Horse | Koala
```

```
typedef union Thing {
  int   myInt;
  char* myCharP;
} Thing;

typedef enum Animal {
  Cat,
  Horse,
  Koala
} Animal;
```

Example: C — some good stuff

C has types, including sums and products:

```
type Thing = Either Int String
```

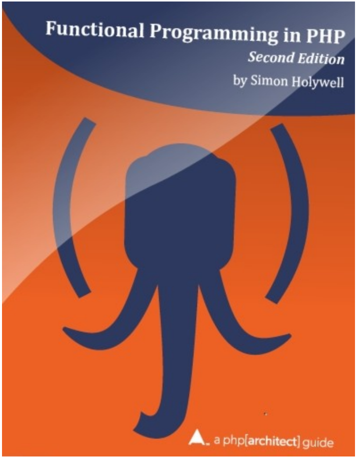
```
data Animal = Cat | Horse | Koala
```

```
typedef union Thing {
  int   myInt;
  char* myCharP;
} Thing;

typedef enum Animal {
  Cat,
  Horse,
  Koala
} Animal;
```

Note: depending on the compiler `unions` are not actually checked!

You might thus interpret something as `int` that is actually `char*`.

Example: PHP



https://www.functionalphp.com/

QuickCheck conquering the world

Property-based testing is the main idea behind QuickCheck:

1. define properties that should hold,
2. define "recipes" for generating random values,
3. run the tests!

QuickCheck conquering the world

Property-based testing is the main idea behind QuickCheck:

1. define properties that should hold,
2. define "recipes" for generating random values,
3. run the tests!

By now this has spread to many other languages:

- Python: https://hypothesis.works/
- JavaScript: https://github.com/jsverify/jsverify
- Go: https://pkg.go.dev/testing/quick
- ...

Thank you for listening, and I am curious to see your projects!

Next and & last meeting: presentations on 28 January