Farewell

Functional Programming 2018/19

Alejandro Serrano

Final menu

- Presentations about libraries and structures
- Q&A session
  - Ask me more in the break
- Closing remarks

Participation on research

We would like to gather your DomJudge assignments to perform research on programming education

- You do not need to do anything else than allowing us to look at the assignments
- Assignments are anonymized before anybody looks at it

Participation is **completely optional**

- During the exam I will give you a *formulier toestemming*, which you need to sign if you agree to help us

Presentations

- Parallelism with `monad-par`
- Semirings

Q&A session

Things with symbols: $, <$>, <*>, <|>, >>=

```
($) :: (a -> b) -> a -> b
```

Rule 1: writing $ is like writing a parentheses until the end of the expression

Rule 2: nested uses of $ create nested parentheses

```
f lst = map (+1) (filter even (map read lst))
-- Too many parentheses!
f lst = map (+1) $ filter even $ map read lst
-- In this case, better use composition
f     = map (+1) . filter even . map read
```

Things with symbols: `<$>`, `<*>`

```haskell
-- From Functor and Applicative
fmap  ::   (a -> b) -> f a -> f b
(<$>) ::   (a -> b) -> f a -> f b
(<*>) :: f (a -> b) -> f a -> f b
```

If we have a bunch of arguments inside a context/functor/monad

```haskell
x :: m a, y :: m b, z :: m c ...
```

and we want to apply a *pure* function

```haskell
f :: a -> b -> ... -> r
```

we need to *lift* the function using a combination of them

```haskell
f <$> x <*> y <*> z <*> ... :: m r
```

Things with symbols: <$>, <*>

```
mapM :: (a -> m b) -> [a] -> m [b]
mapM _ []     = return []
mapM f (x:xs) = (:) <$> f x <*> mapM f xs
             --         ---     ---------
             --         m b       m [b]
             --
             -- (:) :: b -> [b] -> [b]
```

Things with symbols: <|>

`<|>` or `mplus` model the idea of *trying different possibilities*

- In the case of `[]`, concatenate all solutions

```
(<|>) = (++)
```

- In the case of `Maybe`, get the first one which doesn't fail
  - That is, obtain the first `Just`

```
Just x  <|> _ = Just x
Nothing <|> y = y
```

Things with symbols: >>=

`(>>=)` is the *bind* operation of a monad

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

- If you give me an `a` inside a monad
- And tell me how to continue if I “unwrap” the `a` for you
- Then I can apply the continuation to the value for you

```
do x <- thing    ===    thing >>= \x ->
   continue                 continue
```

Monoids

Monoid is the generalization of the properties exhibited by $+$, $\times$, list concatenation, $\vee$, $\wedge$, …

- A binary operation `mappend` or `(<>)`
- Which is associative

```
(x <> y) <> z  ===  x <> (y <> z)
```

- And has a neutral element `mempty`

```
x <> mempty = x
mempty <> x = x
```

  - 0 for $+$, 1 for $\times$, `[]` for `(++)`, `False` for $\vee$, `True` for $\wedge$

Monoids and monads

Are monoids and monads related in any way?

Yes, they are deeply connected.

Monoids and monads

Are monoids and monads related in any way?

Yes, they are deeply connected.

But we don't have time to explain how now.

Monoids and monads

Are monoids and monads related in any way?

Yes, they are deeply connected.

But we don't have time to explain how now.

You need to look at *category theory*, a branch of mathematics (and computer science).

```
return :: a -> m a          -- is like `mempty`
join   :: m (m a) -> m a    -- is like `mappend`
```

Lecture 10, exercise 1

Finish the proof of `reverse . reverse = id`

We need the following lemmas:

```
-- Distributivity of (++) over reverse
reverse (xs ++ ys) = reverse ys ++ reverse xs
-- Reverse on singleton lists
reverse [x]        = [x]
```

The second one is simple equational reasoning.

Lecture 10, exercise 1

```
reverse (xs ++ ys) = reverse ys ++ reverse xs
```

By induction on `xs`:

Lecture 10, exercise 1

```
reverse (xs ++ ys) = reverse ys ++ reverse xs
```

By induction on `xs`:

- Case `xs = []`

```
reverse ([] ++ ys)
= {- defn of (++) -}
reverse ys

reverse ys ++ reverse []
= {- defn of reverse -}
reverse ys ++ []
= {- WE ARE STUCK AGAIN!!! -}
```

We need to prove a separate lemma: `ts ++ [] = ts`

Lecture 10, exercise 1

- Case `xs = z:zs`
  - IH: `reverse (zs++ys) = reverse ys ++ reverse zs`

```
reverse ((z:zs) ++ ys)
= {- defn of (++) -}
reverse (z : (zs ++ ys))
= {- defn of reverse -}
reverse (zs ++ ys) ++ [z]

reverse ys ++ reverse (z:zs)
= {- defn of reverse -}
reverse ys ++ (reverse zs ++ [z])
= {- associativity of (++) -}
(reverse ys ++ reverse zs) ++ [z]
= {- IH -}
reverse (zs ++ ys) ++ [z]
```

Lecture 10, exercise 1

```
ts ++ [] = ts
```

By induction on `ts`:

Lecture 10, exercise 1

```
ts ++ [] = ts
```

By induction on `ts`:

- Case `ts = []`:

```
[] ++ []
= {- defn of (++) -}
[]
```

- Case `ts = p:ps`
  - IH: `ps ++ [] = ps`

```
(p:ps) ++ []
= {- defn of (++) -}
p : (ps ++ [])
= {- IH -}
p : ps
```

Lectures 12/13, exercise 2

Define a function:

```
tuple :: Monad m => m a -> m b -> m (a, b)
```

using explicit (>>=), `do`-notation and applicative operators.

Lectures 12/13, exercise 2

```haskell
tuple :: Monad m => m a -> m b -> m (a, b)
```

Using do-notation

```haskell
tuple x y = do x' <- x
               y' <- y
               return (x', y')
```

Lectures 12/13, exercise 2

```
tuple :: Monad m => m a -> m b -> m (a, b)
```

Using do-notation

```
tuple x y = do x' <- x
               y' <- y
               return (x', y')
```

Using (>>=)

```
tuple x y = x >>= \x' ->
            y >>= \y' ->
            return (x', y')
```

Lectures 12/13, exercise 2

```haskell
tuple :: Monad m => m a -> m b -> m (a, b)
```

Using applicative operators

We need to find a function `a -> b -> (a, b)` and lift it

```haskell
pair x y = (x, y)            -- define it yourself
(,) :: a -> b -> (a, b)      -- constructor for pairs
```

To lift it, use a combination of `<$>` and `<*>`

```haskell
tuple x y = (,) <$> x <*> y
```

Functor-applicative-monad hierarchy

Functor which is not applicatives?

Applicative which is not monad?

Functor-applicative-monad hierarchy

Functor which is not applicatives?

Applicative which is not monad?

Thanks, StackOverflow!

Functor which is not applicative

```haskell
data Pair a b = Pair a b  -- like a tuple

instance Functor (Pair a) where
  fmap f (Pair x y) = Pair x (f y)

instance Applicative (Pair a) where
  pure y = ({- what here?? -}, y)

-- You can fix it, but not for every "a"
instance Monoid a => Applicative (Pair a) where
  pure y = (mempty, y)
  (<*>)  = ...
```

Applicative which is not monad

```haskell
data ZipList a = ZL [a]

instance Applicative ZipList where
  pure x = ZL (repeat x)  -- infinite list of "x"s
  ZL fs <*> ZL xs = ZL (zipWith (\f x -> f x) fs xs)
  -- This obbeys all the laws!

-- You can use the "Monad" from lists
-- But then the following does not hold:
f <*> x  ===  do f' <- f
                 x' <- x
                 return (f' x')
```

Closing remarks

Goals for the course

- Learn the **functional** paradigm and **style**
  - You can apply FP techniques everywhere!
  - Every (serious) language has H-O functions
- Experience a **strong static type system**
- **Reason** about programs
  - Correct software is our ultimate goal

Courses about or using FP at UU

- *Functioneel Programmeren*
- Talen en Compilers: year 3, period 2
  - Haskell applied to compiler writing
- Software Testing en Verificatie: year 3, period 4
  - More reasoning about programs

If you want to know more

More Haskell?

- *Pearls of Functional Algorithm Design*, by Bird
  - Puzzles with a nice functional solution
- *the fun of programming*, by Gibbons and de Moor
  - Even more niceties in a functional style
- *Haskell from First Principles*, by Allen and Moronuki
  - Covers additional topics, like transformers
- *Beginning Haskell*, by, ehmmm… me
  - Which happens to be an intermediate book

If you want to know more

Learn other functional languages

- *F#* for the .NET platform
  - *Beginning F# 4.0* and *Expert F# 4.0*
- *Kotlin* and *Scala* for the Java platform
  - *Functional Kotlin*
  - *Functional Programming in Scala*
- *Swift* for iOS development
  - *Functional Swift*

If you want to know more

Or just drop by my office

Success with your exams!