Elo en co presenteren: Functionele en Logische Programmeertalen 2015-2016

Prelude

Referential transparency: Dit houdt in dat f(x) op elke plaats in een programma kan vervangen worden door zijn output. M.a.w. dat de uitkomst van f(x) steeds gelijk blijft.

Lambda calculus

`\` is de code voor het lambda symbool

Uitdrukkingen worden omschreven a.d.h.v. volgende zaken:

- Variabelen: `x`
- abstracties: `(\x . M)`
- toepassing: `(M N)`

Abstracties komen overeen met substituties: `(\x . f x) i == f i` Variabelen kunnen gebonden `(\x . f x)` of vrij `(\y . f x)` zijn.

Haskell doet aan *lazy evaluation*, dit houdt in dat een waarde slechts 1 keer wordt berekend, en slechts wanneer deze wordt opgevraagd.

Voor voorbeelden: zie slides

Haskell

Lists: bestaan uit head+tail **of** init+last Concatenatie: `[1, 2] ++ [3, 4]` **of** `'U' : "Gent"` Index: `[1, 2, 3] !! 1` Ordening: ordening van lijsten gebeurd lexicografisch Take en Drop (spreekt voor zich)

Haskell heeft listcomprehensions en ranges (`[1..2]`) Pattern matching op lists: (`x:xs`) `:t` geeft het type van zijn parameter terug.

Types & functions

Type signaturen: `Int -> Int` : neemt 1 `Int` als argument en geeft er 1 terug

Type classes

Komt overeen met interfaces in andere talen.

- Eq : (==) en (/=)
- Ord : groter/kleiner
- Enum : types die overlopen kunnen worden (met o.a. succ)
- Bounded : types met een boven-en onderlimiet hebben (zoals int)
- Number : types die als getal gebruikt kunnen worden (int, float,...)
- Integral: types die een geheel getal voorstellen
- Show : types waarvan de waarde als string kan worden voorgesteld
- Read : types waarvan de waarde uit een string kan gehaald worden

Pattern matching

Volgorde is belangrijk bij het definiëren van een functie per case! Wildcards (`\_`) kunnen aangeven dat een parameter niet van belang is. `all@(x:xs)` houdt een referentie naar de gehele lijst bij in *all*

Pattern matching kan ook door gebruik te maken van case of, of binnen een ander functie gedefinieerd in een where clausule, of door gebruik te maken van guards. Let-in constructie lijkt zeer op where, zonder de *in* is hetgeen gedefinieerd in de Let echter overal zichtbaar.

Common operations

```
map     :: (a -> b) -> [a] -> [b]
filter  :: (a -> Bool) -> [a] -> [a]
zip     :: [a] -> [b] -> [(a, b)]
unzip   :: [(a, b)] -> ([a], [b])
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

Folding

```
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
```

Magic with Algebraic Data Types

```
data Car = Car String String Int deriving (Show)
ghci> Car "Ford" "Mustang" 1967
Car "Ford" "Mustang" 1967

data Car = Car {company :: String, model :: String, year :: Int} deriving (Show)
ghci> Car {company="Ford", model="Mustang", year=1967}
Car {company = "Ford", model = "Mustang", year = 1967}
```

De parameters in record syntax moeten niet in de juiste volgorde staan.

Maybe

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

Record syntax

Voorbeeld:

```
data Person = Person String String Int Float String String deriving (Show)
firstName :: Person -> String
firstName (Person firstname _ _ _ _ _ ) = firstname
```

Wordt:

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     , height :: Float
                     , phoneNumber :: String
                     , flavor :: String
                     } deriving (Show)
```

Functor

Iets waar over gemapt kan worden.

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

```
fmap :: Functor f => (a -> b) -> f a -> f b
fmap f (Maybe a) === f <$> (Maybe a)
```

Neemt geen concrete type zoals `Int`, `Bool`, `Maybe String`,... Het neemt een type constructor met 1 parameter, zoals `Maybe`. `fmap` vb: neem een functie `Int -> Bool`, en een `Maybe Int`. fmap geeft een `Maybe String` terug.

Real life vb: `map` is een implementatie van `fmap`, waar `f` altijd `List` is.

```haskell
map :: (a -> b) -> [a] -> [b]

-- We gebruiken hier geen [a] maar []!
instance Functor [] where
    fmap = map
```

Nu kunnen we een functie partially applyen zodat ze maar 1 type parameter neemt zoals gedefinieerd in de `Functor` typeclass.

```haskell
instance Functor (Either a) where
    fmap f (Right x) = Right (f x)
    fmap f (Left x) = Left x -- Hier niet toepassen omdat a en b in (Either a b) ander
                             -- gelijk zouden moeten zijn
```

De type signature kan je dan zien als:

```haskell
(b -> c) -> Either a b -> Either a c
```

`*` is het type van een concrete type zoals `Int` of `Maybe Int`, maar dus niet `Maybe`.

```haskell
data Barry t k p = Barry { yabba :: p, dabba :: t k }
```

Heeft dus het kind `(* -> *) -> * -> * -> *`.

Om dit `Functor` te laten instancen, moeten we te eerste 2 parameters applyen.

```haskell
instance Functor (Barry a b) where
    fmap f (Barry {yabba = x, dabba = y}) = Barry {yabba = f x, dabba = y}
```

`IO` is een ook een `Functor` dus kunnen we erover fmappen

```haskell
do line <- fmap reverse getLine
```

```
If you ever find yourself binding the result of an I/O action to a name, only to apply
```

Functor laws

Een `Functor` moet aan twee regels voldoen die niet enforced zijn door Haskell.

- Als we de `id` functie over een functor mappen, moeten we dezelfde functor terug krijgen (`fmap`

`id = id`)

- Mappen van twee composed functies is hetzelfde als mappen over de twee functies apart (`fmap (f . g) = fmap f . fmap g`)

Applicative

Elke `Applicative` is ook een `Functor`. `Applicative` neemt opnieuw een type constructor en geen concreet type.

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

`pure` neemt een waarde en geeft een minimale context met die waarde erin terug

`Applicative` kan gebruikt worden om een functie op > 1 functors toe te passen: `pure f <*> x <*> y <*> ...` is hetzelde dan `fmap f x <*> y <*> ...` en dit is hetzelfde dan `f <$> x <*> y <*> z`.

Real life vb:

```
instance Applicative [] where
    pure x = [x]
--  (<*>) :: [a -> b] -> [a] -> [b]
    fs <*> xs = [f x | f <- fs, x <- xs]
```

We gebruiken een list comprehension omdat `fs` 0 of meer functies kan bevatten. We passen elke functie uit `fs` toe op elke waarde uit `xs`.

Nog een vb:

```
instance Applicative IO where
    pure = return
--  (<*>) :: IO (a -> b) -> IO a -> IO b
    a <*> b = do
        f <- a
        x <- b
        return (f x)

myAction :: IO String
myAction = do
   a <- getLine
   b <- getLine
   return $ a ++ b

-- Hetzelfde als
```

```haskell
myAction :: IO String
myAction = (++) <$> getLine <*> getLine
```

Newtype

Newtype kan je gebruiken om een nieuw type te maken van een bestaand type. Het kan maar 1 value constructor hebben dat 1 veld heeft. Met `data` kan je er meerdere hebben.

```haskell
newtype Pair b a = Pair { getPair :: (a,b) }
```

Een `newtype` moet niet gaan boxen en unboxen, als resultaat kan je het volgende doen.

```haskell
newtype CoolBool = CoolBool { getCoolBool :: Bool }

helloMe :: CoolBool -> String
helloMe (CoolBool _) = "hello"

ghci> helloMe undefined
"hello"
```

Dit werkt niet met `data` omdat de `undefined` moet geevalueerd worden (aangezien we meerdere value constructors kunnen hebben en we dus moeten kunnen pattern matchen) en crashed dus. `newtype` moeten we echter niet evalueren (maar 1 value constructor) en dat werkt dus gewoon.

Type

Met `type` kan je een alias definieren voor een type.

```haskell
type IntList = [Int]
```

Monoid

Eender welk concreet type kan een instance van `Monoid` zijn.

```haskell
class Monoid m where
    mempty :: m
    mappend :: m -> m -> m
    mconcat :: [m] -> m
    mconcat = foldr mappend mempty

instance Monoid [a] where
    mempty = []
    mappend = (++)
```

Enkele eigenschappen:

- Ze nemen twee parameters
- De parameters en teruggeven waarden hebben hetzelfde type
- Er is een neutraal element

```
A monoid is a pair of an operator (@@) and a value u,
where the operator has the value as identity and is associative.
Examples: (+) and 0; (++) and []; (||) and False; (>>) and done
```

```
u@@x       = x
x@@u       = x
(x@@y)@@z = x@@(y@@z)
```

Design Patterns for Functional Programming (Monads)

Monad

Als we een waarde met context (Zoals `Maybe`) en een functie hebben dat een gewone waarde neemt maar een waarde met context returned, hoe krijgen we dan een waarde met context in die functie?

Monadic value = waarde met context

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

    (>>)   :: m a -> m b -> m b
    x >> y = x >>= \_ -> y

    fail   :: String -> m a
    fail msg = error msg

instance Monad Maybe where
    return x       = Just x
    Nothing >>= f = Nothing
    Just x  >>= f = f x
    fail _         = Nothing
```

Er is geen class constraint om historische redenen maar, elke monad is wel degelijk een applicative functor.

`return` is hetzelfde als `pure`, het geeft een waarde een minimale context die de warde bevat.

`bind` laat ons toe van het resultaat van een vorige berekening constant door te geven aan de volgende. `bind` neemt een functie en past deze toe op alle interne waarden.

`>>` (lees als: then) past bind toe op de eerste parameter en een functie die de tweede parameter teruggeeft (zijn input dus negeert). `Nothing >> Just 3` geeft `Nothing` omdat `Nothing` binden met iets anders altijd `Nothing` is. `Just 3 >> Just 4` geeft `Just 4` omdat de input genegeerd wordt. `Just 4 >> Nothing` geeft `Nothing` omdat de functie gewoon `Nothing` returned.

```
putChar :: Char -> IO ()
```

Geeft een commando terug dat een IO actie uitvoert wanneer uitgevoerd.

Do notatie

`<-` unwrapped een type, `IO Int` wordt dus `Int`.

Real life vb:

```
foo :: Maybe String
foo = Just 3 >>= (\x ->
      Just (show x ++ "!"))

foo :: Maybe String
foo = do
    x <- Just 3
    Just $ show x ++ "!"
```

Zo kan je zien dat de laatste expressie in een do notatie geen `<-` kan zijn. Als er in de do notatie iets `Nothing` is, dan zal do `Nothing` returnen.

Een monadic value, in de do notatie, zonder `<-` is hetzelfde als `>>` na de monadic value te plaatsen.

Nog een vb:

```
listOfTuples :: [(Int,Char)]
listOfTuples = do
    n <- [1,2]
    ch <- ['a','b']
    return (n,ch)
```

Dit returned een lijst van tuples van elke combinatie van 1,2 en a,b.

`return` wrapped een type, `Int` wordt dus `IO Int`.

`fail` wordt opgeroepen als pattern matching faalt in een do expressie.

Wetten

```
return a >>= f  = f a -- left identity
m >>= return    = m   -- Right identity
(m >>= f) >>= g = m >>= (\x -> f x >>= g) -- Associativity
```

```
do { x' <- return x;
     f x'
   }

-- Same as
do { f x }
```

```
do { x <- m;
     return x
   }

-- Same as
do { m }
```

```
do { y <- do { x <- m;
               f x
             }
     g y
   }

-- Same as
do { x <- m;
     do { y <- f x;
          g y
        }
   }

-- Same as
do { x <- m;
     y <- f x;
     g y
   }
```

IO mapping

Om een IO functie (zoals print) te mappen over een lijst moet worden gebruik gemaakt van `mapM` ipv

`map`.

```
mapM :: (Traversable t, Monad m) => (a -> m b) -> t a -> m (t b)
```

Een lijst overlopen gebeurt met `forM`.

```
forM :: (Traversable t, Monad m) => t a -> (a -> m b) -> m (t b)
```

Errors kunnen opgevangen worden door *catch*

State

```
s -> (a, s)
input -> (result, new\_state)
```

MonadPlus

```
guard :: (MonadPlus m) => Bool -> m ()
guard True = return ()
guard False = mzero

ghci> [1..50] >>= (\x -> guard ('7' `elem` show x) >> return x)
[7,17,27,37,47]omay
```

Monad Transformers

Zie uitgewerkte voorbeelden in slides. Monad transformers komt erop neer 2 monads te combineren, om de "kracht" van beide te hebben.

Lenses

Elke lens heeft een structuur en een focus, bv een persoon structuur met als focus het adres (`laddr :: LensR Person Address`)

```
type Lens s a = forall f . Functor f => (a -> f a) -> s -> f s


view :: Lens s a -> s -> a
view lns s = getConst (lns Const s)
```

```haskell
-- Const is een functie v -> Const v a
-- Dus lns heeft zijn a -> f a namelijk a -> Const a a

-- De haken returnen dus (Const a s)
-- En getConst daarop geeft de a

newtype Const v a = Const v
getConst :: Const v a -> v
getConst (Const x) = x
```

Equational Reasoning

Code bekijken en delen substitueren alsof het wiskundige expressies zijn. Dit wordt gebruikt om de correctheid van programma's aan te tonen

Parallel programeren

Semiexpliciet: (par en pseq) Om efficiënt parallel te programmeren moet rekening gehouden worden met de volgorde waarin delen worden berekend. Hiervoor kan worden gebruik gemaakt van pseq (`pseq :: a -> b -> b` evaluate a then evaluate b)

Expliciet:

- Spawn (threads)
- Blocking
- Non-Blocking (Maybe)

Software Transactional Memory: zie "Beautiful Concurrency" by Simon Peyton Jones. Blocking:

```haskell
type Account = TVar Int

limitedWithdraw :: Account -> Int -> STM ()
limitedWithdraw acc amount = do
    bal <- readTVar acc
    if amount > 0 && amount > bal
    then retry
    else writeTVar acc (bal - amount)

-- Hetzelfde als
limitedWithdraw :: Account -> Int -> STM ()
limitedWithdraw acc amount = do
    bal <- readTVar acc
    check (amount <= 0 || amount <= bal)
    writeTVar acc (bal - amount)
```

Choice:

```
limitedWithdraw2 :: Account -> Account -> Int -> STM ()
-- (limitedWithdraw2 acc1 acc2 amt) withdraws amt from acc1,
-- if acc1 has enough money, otherwise from acc2.
-- If neither has enough, it retries.
limitedWithdraw2 acc1 acc2 amt
  = orElse (limitedWithdraw acc1 amt) (limitedWithdraw acc2 amt)
```

Logisch programmeren

1. Omschrijf de wereld.
2. Stel vragen over deze wereld

De wereld wordt omschreven adhv feiten en regels (implicaties). In prolog is een implicatie voorgesteld door `:-`, een conjunctie (en) door `,` en een disjunctie (of) door `;`.

Stament = clause Predicaat = functie Atom = constant

`[x->mia]` is vervang `mia` door `x` in de uitdrukking `mia`

Unification if

- 2 atoms are the same
- if one of the terms a variable, T1 is instantiated as T2 and vice versa
- if both are complex terms with same functor/arity (# of params) and all the corresponding arguments unify and the variable instantions are compatible

Examenvragen

```
eval :: Term -> Env -> Value
-- b = body
eval (Lam n b ) env = Fun (\x -> eval b ((n,x) : env))
```

Dit evalueert de body van de lambda met de huidige environment, waar de parameter van de lambda aan is toegevoegd.

```
x >>= f = MaybeT $ do maybe_value <- runMaybeT x
                      case maybe_value of
                          Nothing    -> return Nothing
                          Just value -> runMaybeT $ f value

-- Of
```

```haskell
x >>= f = MaybeT $ runMaybeT x >>=  maybe (return Nothing) (runMaybeT . f)
```