Inleiding tot Functioneel Programmeren – les 3

Kris Luyten, Jo Vermeulen
{kris.luyten,jo.vermeulen}@uhasselt.be

Expertisecentrum voor Digitale Media
Universiteit Hasselt

Currying

$$f: (X \times Y) \longrightarrow Z$$

Currying: een functie met meerdere parameters omzetten naar verschillende functies met telkens 1 parameter

$$f: X \longrightarrow (Y \longrightarrow Z)$$

```
f x y = z
(f x) y = z
((f x) y) = z
```

```
f x = \y -> z
f = \x -> \y -> z
```

Currying

- Som :: (Int,Int)->Int kan Som :: Int->Int->Int worden
  - "Som 5 6" is een functie, maar "Som 5" kan evengoed gebruikt worden als argument voor een andere functie (bijv. de *map* functie)
- Functies worden beter herbruikbaar
  - "Partiële" functies als argument
  - Functies samenstellen (f en g wordt f.g)

Leg uit/vul aan…

```
mysteryf = foldl (flip (:)) []
```

Extreme Haskell…

twisted_fac.hs

```
fac :: Int -> Int
fac n = foldr (*) 1 [1..n]

main = print (map (fac) [3,4,5,6,7,8])
```

Extreme Haskell…

lazy_primes.hs

```
primes = sieve [2..]
sieve (h:t) = h : sieve [x |
                         x <- t, x `mod` h /= 0]

main = do
    putStrLn "Geef een geheel getal:"
    nummer <- getLine
    print (take (read nummer) primes)
```

Extreme Haskell…

lazy_fibo.hs

```
lfibo :: [Int]
lfibo = 0 : 1 : zipWith (+) lfibo (tail lfibo)

main = do
  putStrLn "Enter a number: "
  n <- getLine
  print (take (read n) lfibo)
```

Evaluatiestrategiëen

Evaluatie Strategie

- Op welke manier wordt een expressie geëvalueerd?
  - De manier waarop argumenten van een functie worden geëvalueerd bij een functie-oproep.
- Strikte versus niet-strikte evaluatie
  - Strikt: parameters functie wordt geëvalueerd voor uitvoering functie
  - Niet-strikt: parameters functie worden niet geëvalueerd tenzij deze gebruikt worden. Evaluatie van de parameters gebeurt dan niet noodzakelijk voor de uitvoering van de functie

Strikte evaluatie

- Call-by-Value: parameter expressies worden geëvalueerd en toegekend aan een variabele naam (copy!) voor uitvoering functie.

Functie aanroep:

```
int i=3;
bereken(i);
```

Functie def:

```
int bereken(int j){
   j = j + j*2;
   return j;
};
```

Java, ML, Scheme, C, Pascal,…

Strikte evaluatie

- Call-by-Reference: : parameter expressies worden geëvalueerd en een variabelenaam verwijst naar de waarde (referentie naar geheugen!) voor uitvoering van de functie.

Functie aanroep:

```
int i=3;
bereken(i);
```

Functie def:

```
int bereken(int& j){
   j = j + j*2;
   return j;
}
```

Functie aanroep:

```
int i=3;
bereken(&i);
```

Functie def:

```
int bereken(int* j){
   *j = *j + (*j)*2;
   return *j;
}
```

Niet-strikte evaluatie

- Call-by-Name: iedere verwijzing naar de expressie wordt vervangen door de expressie zelf. De expressie wordt opnieuw geëvalueerd bij elk gebruik ervan.

Functie aanroep:

```
a[0]=3; a[1]=14;
int i =0;
bereken(a[i],&i);
```

Functie def:

```
int bereken(int l, int i){
    l = l+1;
    i = i+1;
    l = l-1
    return l;
}
```

Niet-strikte evaluatie

- Call-by-Name: iedere verwijzing naar de expressie wordt vervangen door de expressie zelf. De expressie wordt opnieuw geëvalueerd bij elk gebruik ervan.

Functie aanroep:

```
a[0]=3; a[1]=14;
int i =0;
bereken(a[i],&i);
```

Functie def:

```
int bereken(int a[i], int i){
    a[i] = a[i]+1;
    i = i+1;
    a[i] = a[i]-1
    return a[i];
}
```

Zelden of nooit ondersteund in de meeste programmeertalen.
Algol60 is de bekendste taal die call-by-name implementeert.

Niet-strikte evaluatie

- Call-by-Need: werkt zoals call-by-name maar na de eerste evaluatie wordt die waarde verder gebruikt. In pure functionele programmeertalen is het effect hetzelfde als call-by-name en wordt vaak “Lazy Evaluation” genoemd.

```
plus14 x y = x+x
plus14 (2*3) (4-2)
```

Haskell implementeert call-by-need. Verschillende andere talen ondersteunen deze semantiek ook, naast call-by-value: D, Scheme, Nemerle, Python,…

Data Types

Types

- Karakters en Strings
- Numerieke types: Integer, Float,…
- Boolean: True en False
  - Logische en: &&, logische of: || en negatie: not
- Polymorfe types *“a”*
  - Worden afgeleid door Haskell zelf
- Types makkelijk uitbreidbaar dankzij type classess

:t vraagt het type op
in de interactieve shell

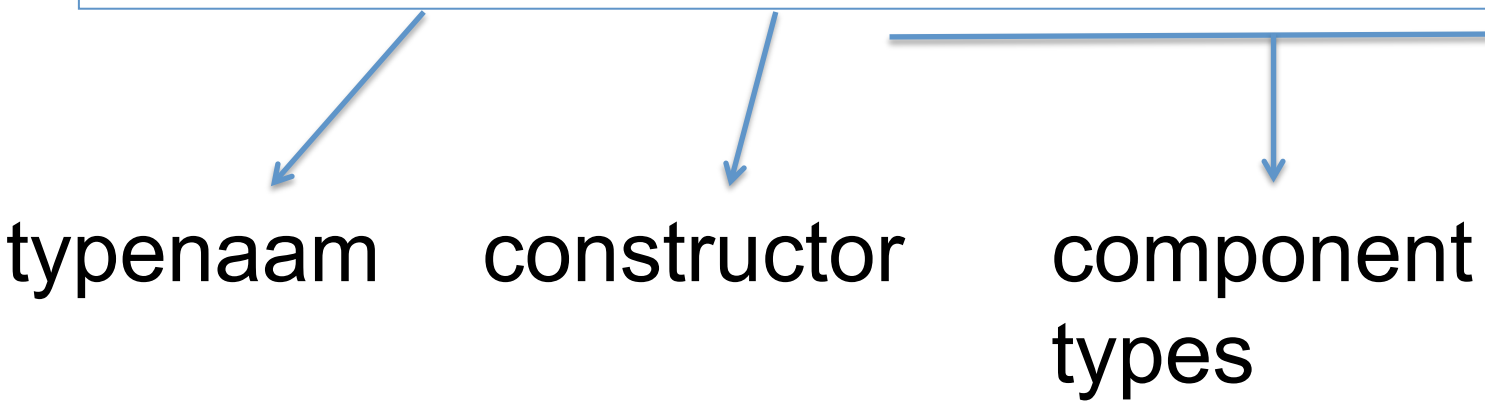
console

```
$> ghci
Prelude> :t 42
42 :: (Num t) => t
Prelude> :t "Hallo"
"Hallo" :: [Char]
Prelude> :t 3.1415
3.1415 :: (Fractional t) => t
Prelude> :t x
Prelude> :t [1,2,45]
[1,2,45] :: (Num t) => [t]
Prelude> :t [1,2,45.4]
[1,2,45.4] :: (Fractional t) => [t]
Prelude> :t ["hallo", "h"]
["hallo", "h"] :: [[Char]]
Prelude> :t ['h','a','l','l','o']
['h','a','l','l','o'] :: [Char]
```

Data Types

- Haskell laat user defined datatypes toe
- Bron: *Real World Haskell* van Bryan O'Sullivan, Don Stewart en John Goerzen - http://book.realworldhaskell.org/read/defining-types-streamlining-functions.html

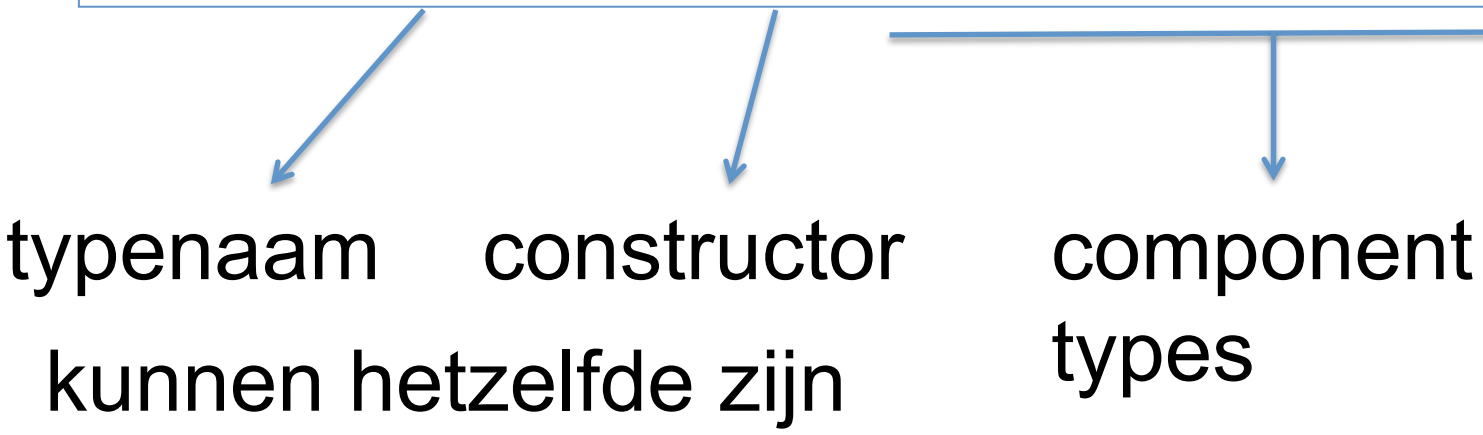
```
data BookInfo = Book Int String [String]
```



Data Types

- Haskell laat user defined datatypes toe
- Bron: *Real World Haskell* van Bryan O'Sullivan, Don Stewart en John Goerzen - http://book.realworldhaskell.org/read/defining-types-streamlining-functions.html

```
data Book       = Book Int String [String]
```



Data Types

```
data BookInfo      = Book      Int String [String]
```

```
data MagazineInfo = Magazine Int String [String]
```

```
Book 42 “The Hitchhikers Guide to the
Galaxy” “Douglas Adams”
```

Algebraic Data Types

```
data Bool = True | False
```

Meerdere constructors

Alternatives of *cases*

Algebraic Data Types

billinginfo.hs

```
type CustomerID = Int
type CardHolder = String
type CardNumber = String
type Address = [String]
data BillingInfo =
   CreditCard CardNumber CardHolder Address
   | CashOnDelivery
   | Invoice CustomerID
    deriving (Show)
```

`type` definieert synoniem

Currying and Algebraic Data Types

```
Prelude> :load billinginfo.hs
[1 of 1] Compiling Main
( billinginfo.hs, interpreted )
Ok, modules loaded: Main.
*Main> :type CreditCard
CreditCard :: CardNumber -> CardHolder ->
Address -> BillingInfo
```

CreditCard Cardnumber
(CreditCard Cardnumber) CardHolder
((CreditCard Cardnumber) CardHolder) Address
((CreditCard Cardnumber) CardHolder) Address = BillingInfo

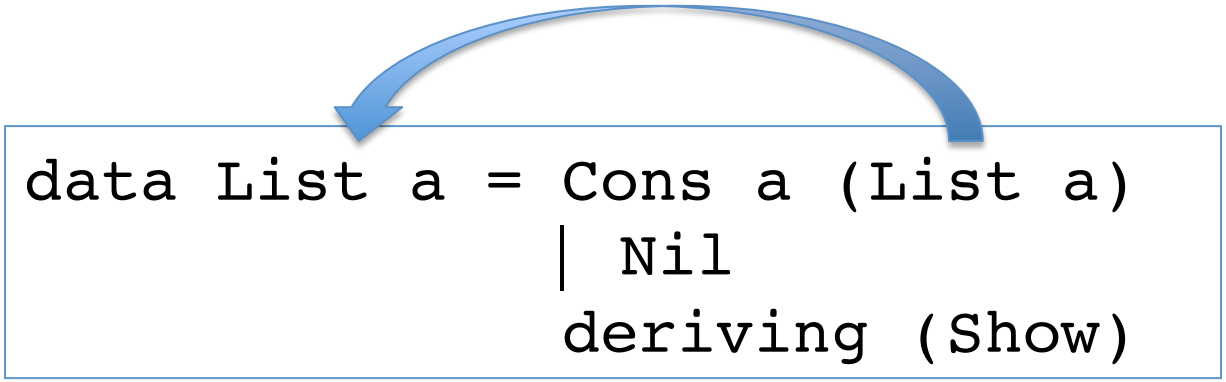
Parameterised Types

- Zelf lezen: http://book.realworldhaskell.org/read/defining-types-streamlining-functions.html#deftypes.paramtypes
- Leg uit hoe `Maybe` polymorfisme ondersteunt voor zelfgedefinieerde datatypes

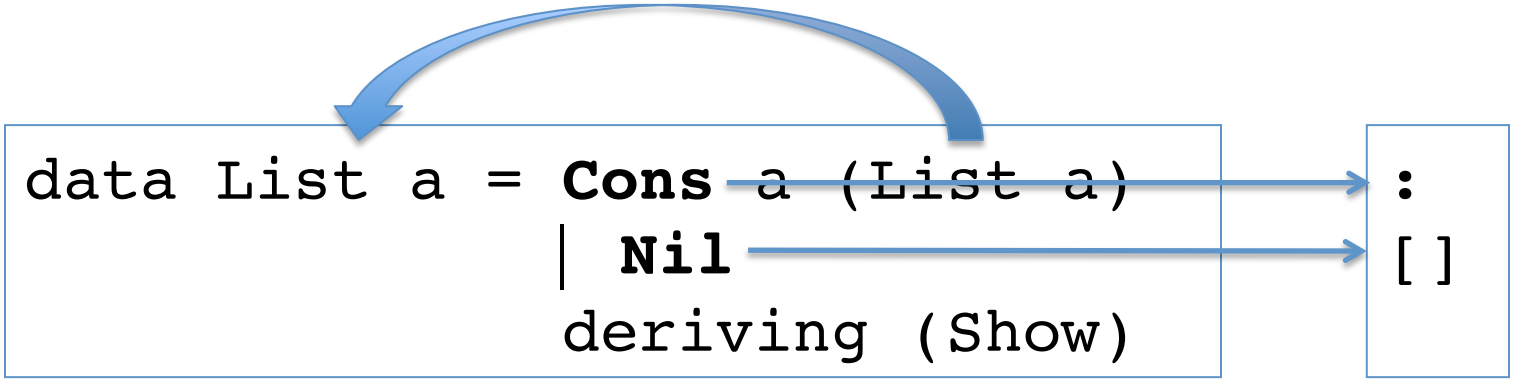
Recursive Types

```
data List a = Cons a (List a)
            | Nil
            deriving (Show)
```

Recursive Types



Recursive Types



Recursive Types

```
data List a = Cons a (List a)
            | Nil
              deriving (Show)

llzip Nil _ = Nil
llzip _ Nil = Nil
llzip (Cons x xs) (Cons y ys) =
                Cons (x,y) (llzip xs ys)

llconvert (x:xs) = Cons x (llconvert xs)
llconvert [] = Nil

main = print ( llzip (llconvert [2,3,7,5])
                     (llconvert [1,0,4,8,37]))
```

Oefeningen

- (Her)schrijf de map en filter functies voor List
- Herneem de volgende oefeningen met deze datastructuur: rotate, draaiom, fibonacci

Recursive Types: Trees

```
data Tree a = Node a (Tree a) (Tree a)
            | Empty
           deriving (Show)
```

Oefeningen

- Schrijf een programma dat enkele binaire bomen definieert (handmatig)
- Schrijf een functie die het aantal nodes telt in de boom
- Schrijf een programma die opzoekt of een element in de boom aanwezig is of niet
- Schrijf een programma dat de hoogte van de boom berekent (lengte grootste pad root->blad)

Oefeningen (ietsje moeilijker)

- Schrijf een functie die checkt of de boom gebalanceerd is
- Kan je een programma schrijven dat alle mogelijke subtrees van de tree teruggeeft?