GEAVANCEERDE PROGRAMMEERTECHNIEKEN

Les 1: Inleiding Functioneel Programmeren in Haskell

Kris Luyten
14 Februari 2025

Inleiding tot Functioneel Programmeren in Haskell

Kris Luyten

Doelstellingen Les 1

- Begrijpen en Gebruiken van het functioneel programmeerparadigma
- Haskell syntax en concepten
- Ontwikkelen van basis functionele programmeervaardigheden
- Toepassen van functionele concepten voor real world toepassingen

Een eerste verkenningstocht

dadadum.hs

```
f []     = []
f (x:xs) = f ys ++ [x] ++ f zs
           where
              ys = [a | a <- xs, a <= x]
              zs = [b | b <- xs, b > x]
```

Magie!

```
$ ghci
ghci> :load dadadum.hs
ghci> f
ghci> :quit
```

```
$ ghci
ghci> :load dadadum.hs
ghci> f []
ghci> :quit
```

```
$ ghci dadadum.hs
ghci> f [1,2,3]
ghci> f [2,2,3,18,0,16]
ghci> f [2,2,3,18,0,-16]
ghci> f [2,2,3,18,0,16,b,a]
ghci> :quit
```

1. Een functie f met een lege lijst $[]$ als parameter, geeft een lege lijst terug

```
f []     = []
```

2. Een functie f met een lijst $(x : xs)$ als parameter,

waarbij x het eerste element is van die lijst (**head**), en xs de rest van de lijst (**tail**)

geeft een samenvoeging terug van een lijst `f ys`, het element `x` en een lijst `f zs`

```
f (x:xs) = f ys ++ [x] ++ f zs
```

Dit is een voorbeeld van **list comprehension**

```
ys = [a | a <- xs, a <= x]
```

Dat lijkt heel erg op een definitie van een verzameling (en is het ook)!

$$ys = \{a \mid a \in xs \land a \leq x\}$$

Wat lees je?

Programmeerparadigma’s

Imperatief & OO

- **Imperatief Programmeren**
  - Focus op hoe uit te voeren
  - Stap-voor-stap instructies
  - Voorbeelden: C, Fortran, Cobol
- **Object-Georiënteerd**
  - Focus op objecten en interacties
  - Data en gedrag gecombineerd
  - Voorbeelden: Java, C++, C#

Functioneel & Logisch

- **Functioneel Programmeren**
  - Focus op functietoepassing
  - Wiskundige benadering
  - Voorbeelden: Haskell, ML
- **Logisch Programmeren**
  - Focus op declaratieve regels
  - Gebaseerd op formele logica
  - Voorbeelden: Prolog

Haskel, een puur functionele programmeertaal

1. Alles is een expressie

In Haskell is **alles een expressie**, inclusief:

- Getallen: `42`
- Wiskundige berekeningen: `2 + 3 * 4`
- Functies: `f x = x + 1`
- `if`-expressies: `if x > 0 then x else -x`
- `let`-bindingen: `let y = 5 in y * 2`
- `case`-expressies: `case xs of [] -> "leeg"; _ -> "niet leeg"`

Een expressie is iets wat een waarde heeft en geëavalueerd kan worden.

Haskel, een puur functionele programmeertaal

2. Zuivere (Pure) Functies

Een functie in Haskell is per definitie **puur**, wat betekent dat ze voldoet aan twee cruciale eigenschappen:

Referentiële transparantie

- Een functieaanroep kan altijd vervangen worden door zijn resultaatwaarde
- Voorbeeld:

```
dubbel x = x * 2

-- Deze expressies zijn equivalent
dubbel (dubbel 2)
dubbel 4
8
```

Haskel, een puur functionele programmeertaal

2. Zuivere (Pure) Functies

Geen side-effects

- Functies kunnen alleen waarden berekenen, niet de programmastaat wijzigen
- Geen globale variabelen
- Geen I/O binnen pure functies
- Geen mutatie van datastructuren

Haskel, een puur functionele programmeertaal

3. Idempotentie

Idempotentie garandeert dat herhaalde uitvoering met dezelfde input altijd dezelfde output produceert:

```
f :: a -> b   -- Type signature
-- Voor elke x, geldt:
f x = f x     -- Altijd waar in Haskell
```

Dit maakt functies:

- Voorspelbaar
- Testbaar (verifieerbaar gedrag)
- Makkelijker om over te redeneren
- Geschikt voor parallelle uitvoering

Haskel, een puur functionele programmeertaal

4. Lambda Calculus

Haskell is gebaseerd op de lambda calculus, een formeel systeem in de mathematische logica:

```
-- Lambda expressie
\x -> x * 2

-- Is equivalent aan de functiedefinitie
f x = x * 2
```

Haskel, een puur functionele programmeertaal

Vergelijking met Imperatieve Programmering

Java

```java
int sum = 0;
for(int i = 0; i < arr.length; i++) {
    sum += arr[i];  // Mutatie van state
}
```

Haskell

```haskell
sum :: Num a => [a] -> a
sum = foldr (+) 0   -- Declaratieve berekening
```

Praktische Implicaties

1. Datastructuren zijn Immutable

```haskell
-- Een lijst aanpassen creëert een nieuwe lijst
lijst = [1,2,3]
nieuweLijst = 4 : lijst  -- Originele lijst blijft ongewijzigd
```

2. Recursie in plaats van Iteratie

```haskell
-- Recursieve implementatie van faculteit
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n-1)
```

3. Pattern Matching

```haskell
data Boom a = Leeg | Knoop a (Boom a) (Boom a)

diepte :: Boom a -> Int
diepte Leeg = 0
diepte (Knoop _ links rechts) = 1 + max (diepte links) (diepte rechts)
```

Voordelen voor Softwareontwikkeling

1. **Betrouwbaarheid**
   - Geen runtime state-mutaties
   - Gegarandeerde uitvoeringsresultaten
2. **Modulariteit**
   - Functies zijn volledig geïsoleerd
   - Eenvoudige compositie van functies
3. **Parallellisatie**
   - Geen gedeelde state
   - Veilige concurrent execution
4. **Formele Verificatie**
   - Direct verband met mathematische logica
   - Makkelijker te bewijzen correctheid

Interessant lesmateriaal

- Learn You a Haskell (LYAH) Voor deze les, **lees: hoofdstukken 1 & 2** (Intro en Starting Out)
- Real World Haskell
- Programming in Haskell, de slides van dit boek **lees: slides van hoofdstukken 1, 2, 4 en 5**, en de Youtube lectures
- Ninety-Nine Haskell Problems
- Effective Haskell

Aan de slag met Haskell

Benodigde Tools

1. **GHC (Glasgow Haskell Compiler)**
   - Industriestandaard compiler
   - Te downloaden van haskell.org
2. **GHCi (Interactieve Omgeving)**
   - Read-Eval-Print Loop (REPL) voor Haskell
   - Ideaal voor experimenteren

Basis Gebruik

```
$ ghci
GHCi> :load dadadum.hs
GHCi> :reload
GHCi> :quit
```

Alles is een Expressie!

In Haskell is alles een expressie die evalueert naar een waarde

```haskell
-- Simpele expressies
42
True
"Hallo"

-- Samengestelde expressies
2 + 3 * 4
if x > 0 then "Positief" else "Niet zo positief"
```

Basis Types en Expressies

Numerieke Types

```
-- Gehele getallen
42 :: Int
-17 :: Int

-- Kommagetallen
3.14 :: Float
-2.718 :: Double
```

Boolean & String Types

```
-- Booleans
True :: Bool
False :: Bool

-- Strings
"Hallo, Wereld!" :: String
```

Operatoren

Infix vs Prefix

```haskell
-- Infix notatie
3 + 4
x `div` y
1 : 2 : []

-- Prefix notatie
(+) 3 4
div x y
```

Functies in Haskell

Basis Syntax en gebruik

`functienaam` `parameter1 parameter2 parameter3...`

```haskell
-- Prefix notatie
succ 5        -- Geeft 6
min 3 4       -- Geeft 3

-- Standaard Prelude functies
abs (-3)      -- Geeft 3
sqrt 16       -- Geeft 4.0
```

Functies in Haskell

Functiedefinitie

`functienaam` `parameter1 parameter2 parameter3...` = `functie-body`

```
dubbel x = x + x
viervoud x = dubbel (dubbel x)

ghci> dubbel 3
6

ghci> viervoud 2
8
```

Functies in Haskell

Lambda(\)-expressie

- Wordt als volgt geschreven: `\arg1 arg2 ... argN` → `expressie`
- anonieme functie "ter plekke" gedefinieerd en gebruikt
- Kan je dadelijk argumenten meegeven

```
ghci> (\x y -> x + y) 3 5
8

ghci> (\x y -> x*x/y+y*y/x) 3 5
10.133333333333335
```

Lijsten

Lijst Creatie

```
-- Lijst met elementen
[1, 2, 3, 4, 5]
['a', 'b', 'c']
["een", "twee", "drie"]

-- Bereik notatie
[1..10]
[2,4..20]
```

Lijsten

De `:` (Cons) notatie

- `:` voegt een element toe aan de voorkant van een lijst
- is de basis-operatie voor lijstcreatie in Haskell
- is een recursieve functie
- infix operator

```
1 : [2,3,4]   -- Geeft [1,2,3,4]

-- "h" wordt toegevoegd aan "allo"
'h' : "allo"  -- Geeft "hallo"

ghci> 0 : 1 : 2 : [] - recursieve functie
[0,1,2]
```

Lijsten

Lijst Operaties

```haskell
-- Basisfuncties, werken ook voor strings!
head [1,2,3]    -- Geeft 1
tail [1,2,3]    -- Geeft [2,3]
length [1,2,3]  -- Geeft 3
take 2 [1,2,3]  -- Geeft [1,2]
drop 2 [1,2,3]  -- Geeft [3]
init (1:[2,3,4,5]) -- Geeft [1,2,3,4]
dropWhile (< 3) [1,2,3,4,5,1,2,3] -- Geeft [3,4,5,1,2,3]
takeWhile (< 3) [1,2,3,4,5,1,2,3] -- Geeft [1,2]
filter (< 3) [1,2,3,4,5,1,2,3] -- Geeft [1,2,1,2]
reverse [1,2,1,2] -- Geeft [2,1,2,1]
null [] -- Geeft True
null (1:2:[2,3]) -- Geeft False
```

Werken ook op strings!

```haskell
ghci> tail "Haskell"
"askell"
ghci> reverse "Haskell"
"lleksaH"
```

Tuples

Basis Gebruik

```haskell
-- Paar
(1, "een")

-- Drietal
(True, 42, "hallo")

-- functies op elementen van tuples
fst (1, 2)     -- Geeft 1
snd (1, 2)     -- Geeft 2
```

Geneste Structuren

```haskell
-- Lijsten van tuples
[(1,"een"), (2,"twee")]

-- Tuples van lijsten
([1,2,3], [4,5,6])

-- Complexe nestings
[([1,2], "a"), ([3,4], "b")]
```

Lijsten & Tuples

De `zip` functie in Haskell

```
zip lijst1 lijst2
```

De functie `zip` combineert **twee lijsten** in **paren**.

- De eerste elementen worden samen een paar.
- De tweede elementen worden een paar.
- Dit gaat zo door totdat **één van de lijsten op is**.
- **Geen expliciete loops nodig!**

Lijsten & Tuples

De `zip` functie in Haskell

Voorbeeld:

```
ghci> zip [1,2,3] ['a', 'b', 'c']
[(1,'a'), (2,'b'), (3,'c')]
ghci> zip [1,2] ['a', 'b', 'c']
[(1,'a'), (2,'b')]
ghci> zip [1,2,3,4] ['x','y']
[(1,'x'), (2,'y')]
```

Resultaat: een lijst van tuples

Lijsten & Tuples

De `zip` functie in Haskell

Indexeren van een lijst door `lazy execution` te gebruiken

```
ghci> zip [0..] ["a", "b", "c", "d", "e"]
[(0,"a"),(1,"b"),(2,"c"),(3,"d"),(4,"e")]

ghci> zip [0..] ['a'..'z']
[(0,'a'),(1,'b'),(2,'c'),(3,'d'),(4,'e'),(5,'f'),(6,'g'),(7,'h'),(8,'i'),(9,'j'),(10,'k'),
(11,'l'),(12,'m'),(13,'n'),(14,'o'),(15,'p'),(16,'q'),(17,'r'),(18,'s'),(19,'t'),(20,'u'),
(21,'v'),(22,'w'),(23,'x'),(24,'y'),(25,'z')]

ghci> zip (tail [1,2,3,4]) [1,2,3,4]
[(2,1),(3,2),(4,3)]
```

Lijsten & Tuples

De `map` functie in Haskell

`map functie lijst`

- map` is een **functie op lijsten** die **elke waarde transformeert**.
- Het past een **functie** toe op **elk element** in een lijst.
- **Geen expliciete loops nodig!**

```
ghci> map (*2) [1,2,3,4]
[2,4,6,8]

ghci> map toUpper "haskell"
"HASKELL"

ghci> map (^2) [1..5]
[1,4,9,16,25]

ghci> map (\(x,y) -> x+y) [(1,2), (3,4), (7,7)]
[3,7,14]
```

Lijsten & Tuples

De `map` functie in Haskell

```
ghci>map (\woord -> woord ++ "!!!") ["hallo", "wereld", "haskell"]
["hallo!!!","wereld!!!","haskell!!!"]

ghci> map (\f -> f 5) [(+1), (*2), (^2)]
[6,10,25]

caesar n string = map (\c -> if c >= 'a' && c <= 'z'
                      then toEnum (((fromEnum c - fromEnum 'a' + n) `mod` 26) + fromEnum 'a')
                      else c) string
ghci> caesar 14 "SecretMessage" -- laat hoofdletters ongemoeid
"SsqfshMsggous"
ghci> caesar (-14) "SsqfshMsggous"
"SecretMessage"
```

Lijsten & Tuples

De `zipWith` functie in Haskell

- `zipWith functie lijst1 lijst2`
- `zip` on steroids!
- `zip` en een functie met exact twee argumenten

```
ghci> zipWith (+) [1,2,3] [4,5,6]
[5,7,9]

ghci> zipWith (\x i -> x^i) [0..10] [0..]
[1,1,4,27,256,3125,46656,823543,16777216,387420489,10000000000]

ghci> zipWith (\x y -> (x, y)) [1,2,3] ["a", "b", "c"]
[(1,"a"), (2,"b"), (3,"c")]

ghci>fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
ghci>take 10 fibs
[0,1,1,2,3,5,8,13,21,34]
```

List Comprehension in Haskell

Syntax:

```
[expressie | generator, voorwaarde1, voorwaarde2, ...]
```

Waar:

- `expressie` is wat je in je resultaatlijst wilt
- `generator` heeft de vorm `variabele <- lijst` en vormt de $input$ voor je expressie
- `voorwaarden` zijn optionele Boolean expressies die je op de input kan toepassen

Basis List Comprehension

Verdubbel alle getallen

```
x * 2 | x ← [1..5]
```

Resultaat: `[2,4,6,8,10]`

Filter met voorwaarde (guard)

```haskell
[x * 2 | x <- [1..10], even x]
```

Resultaat: `[4,8,12,16,20]`

Mathematische notatie

$$\{2x \mid x \in \{1..10\} \land even(x)\}$$

Complexere Voorbeelden

Combineren van lijsten

```
colors = ["rood", "blauw"]
sizes = ["S", "M", "L"]
[kleur ++ " " ++ maat | kleur <- colors, maat <- sizes]
```

Resultaat: `["rood S","rood M","rood L","blauw S","blauw M","blauw L"]`

Merk op dat deze expressie alle mogelijk combinaties uit de twee lijsten genereert!

Werken met strings

```
-- Vind alle klinkers in een string
[c | c <- "Hello World", c `elem` "aeiou"]
```

Resultaat: `"eoo"`

Complexere Voorbeelden

Pythagorese drietallen

Pythagorean triples: drie positieve getallen x, y en z waarvoor geldt dat $x^2 + y^2 = z^2$

```
[(x,y,z) | x <- [1..10],
           y <- [x..10],
           z <- [y..10],
           x*x + y*y == z*z]
```

Resultaat: `[(3,4,5),(6,8,10)]`

Mathematische notatie

$$\{(x, y, z) \mid x, y, z \in \{1..10\} \land x \leq y \leq z \land x^2 + y^2 = z^2\}$$

Praktische Toepassingen

Matrix operaties

```haskell
-- Transpose een 2D matrix
transpose matrix = [[row !! i | row <- matrix] | i <- [0..n]]
  where n = length (head matrix) - 1
```

`[lijst] !! i` haalt het element op index `i` uit `[lijst]`
`[10, 20, 30, 40] !! 2` geeft `30`

Quick Sort implementatie

```haskell
quicksort [] = []
quicksort (x:xs) = quicksort smaller ++ [x] ++ quicksort larger
  where
    smaller = [a | a <- xs, a <= x]
    larger  = [b | b <- xs, b > x]
```

De Zeef van Eratosthenes in Haskell

Wat gaan we bouwen?

- Een implementatie van de **Zeef van Eratosthenes** in Haskell.
- Dit algoritme genereert een lijst van priemgetallen tot een bepaalde limiet.
- We gebruiken: **list comprehension, lijst operaties en recursie**

Wat is de Zeef van Eratosthenes?

- Een efficiënt algoritme om alle priemgetallen tot een bepaald getal te vinden.
- Werkt door herhaaldelijk de veelvouden van elk priemgetal te elimineren.
- Efficiënter dan brute-force testen van getallen.

Werking:

1. Begin met een lijst van getallen vanaf 2.
2. Selecteer het eerste getal als priemgetal.
3. Verwijder alle veelvouden van dat getal.
4. Herhaal dit proces met het volgende niet-gemarkeerde getal.

De Zeef

```
zeef [] = []
zeef (x:xs) = x : zeef [y | y <- xs, y `mod` x /= 0]
```

- `zeef [] = []` : De zeef toepassen op een lege lijst, geeft een lege lijst
- `zeef (x:xs) = x : [...` : De eerste waarde van de lijst is altijd een priemgetal, dat hangen we vooraan de lijst vast met `x:`
- `[y | y <- xs, y mod x /= 0]` : We filteren de rest van de lijst `y <- xs` en verwijderen alle veelvouden van x door de voorwaarde `y mod x /= 0`.
- `zeef [y | y <- xs, y` mod `x /= 0]` : Het algoritme wordt recursief toegepast op de gefilterde lijst.

Stap 3 - Gebruik van de Zeef

```
ghci> priemgetallen n = zeef [2..n]
ghci> priemgetallen 100
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97]
```

Grootste Palindroomgetal vinden

Een palindroomgetal leest hetzelfde van links naar rechts als van rechts naar links.
Bijvoorbeeld: $\ldots, 60006, 69596, 80008, 89598, 220022, 52525, 57475, 98889, 201102, \ldots$
We gaan het grootste palindroomgetal vinden dat gevormd wordt door het product van twee 3-cijferige getallen.

```
isPalindroom n = let s = show n in s == reverse s -- show converteert een getal naar een string
maximum [x * y | x <- [100..999], y <- [100..999], isPalindroom (x * y)]
```

Probeer eens met ghci, welk getal krijg je dan? 906609

- `isPalindroom n = n == reverse n`: eerst checken of een getal een palindroom is.
- `let s = show n`: reverse werkt niet op getallen, met `show` zetten we het getal `n` om naar een string en "binden" we dat aan de variabele `s`.
- `[x * y | x <- [100..999], y <- [100..999]`: alle vermenigvuldigen met 3-cijferige getallen...
- `isPalindroom (x * y)`: ...die een palindroom zijn
- `maximum [...`: en neem daarvan het maximum

Functie Bibliotheken en Prelude.hs

Functies Importeren

```
import Data.List
import Data.Char

-- Nu kun je functies gebruiken zoals:
sort [3,1,4,1,5]
```

Vele bibliotheken standaard beschikbaar: Data.Map, Data.Set, Data.Sequence, Data.Vector, Data.Text, Data.Aeson (JSON), Network.Socket,...

Prelude (en andere bibliotheken)

- Haskell Hackage Repository van bibliotheken en documentatie
- De bibliotheek Prelude wordt standaard geimporteerd. Bestudeer/doorzoek de beschikbare functies voordat je aan een implementatie begint

Uitsmijter: Haskell is lui!

Lazy execution: evalueert enkel expressie als deze nodig zijn (by-need). Kan dus probleemloos werken met oneindige lussen, en oneindige structuren - maar wees toch voorzichtig!

```
ghci> take 7 [5..]
[5,6,7,8,9,10,11]

ghci> take 20 [7..]
[7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26]

ghci> fibs = 0 : 1 : zipWith (+) fibs (tail fibs) --opgepast: breinbreker!
ghci> take 10 fibs
[0,1,1,2,3,5,8,13,21,34]

ghci> infiniteChar a = [a] : infiniteChar a
ghci> take 12 (zipWith (++) (infiniteChar 'a') (infiniteChar 'b'))
["ab","ab","ab","ab","ab","ab","ab","ab","ab","ab","ab","ab"]

ghci> take 120 [a ++ b ++ c| a <- (infiniteChar 'a'), b <- (infiniteChar 'b'), c <- (infiniteChar 'c')]

ghci> take 100 (zeef [2..])

ghci> take 2 [(x,y,z)| x<-[1..],y<-[x..],z<-[y..],x*x+y*y==z*z] -- zooo goed werkt het dus niet...
```

Volgende les

Functies, Types en Classes

Deze stof verwerken?

- Lees de code, kopieer ze, voer ze uit, begrijp de code
- Maak de oefeningen op Dodona
- Zoek je meer uitdaging? Kijk zeker eens naar Project Euler of naar de opgaves van de vorige Vlaamse Programmeerwedstrijden