GEAVANCEERDE PROGRAMMEERTECHNIEKEN

Les 2: Types, Classes, Functies en Pattern Matching

Kris Luyten
21 Februari 2025

Types, Classes, Functies en Pattern Matching in Haskell

Kris Luyten

Doelstellingen Les 2

- Begrijpen en gebruiken van Haskell’s type systeem
- Type Definities voor Functies neerschrijven
- Pattern matching en guards

Haskell is Sterk in Types!

- Haskell is zowel "strongly typed" als "statically typed"
- *Strong*: Je kan geen types omzetten!
- *Statically*: Alle types worden at compile time geverifiëerd
- *Type inference*: Haskell kan de types vaak zelf afleiden
- In Haskell hoef je dus geen types aan te geven, maar het wordt wel *heel sterk* aangeraden, en verwacht
- Python en Java hebben ook sterke typering, maar niet dezelfde krachtige type inferentie, type polymorfisme en uitgebreide type definitie systeem

Static vs Dynamic Typing

Static Typing (Haskell)

```haskell
-- Types controleren at compile time
"123" + 456  -- Type error!
length 42    -- Type error!

-- Geen impliciete type conversie!
"123" ++ 456  -- Type error!
show 123 ++ "456"  -- OK: explicit conversion
```

Dynamic Typing (Python)

```python
# Types controleren at runtime
"123" + 456    # TypeError at runtime
len(42)        # TypeError at runtime

# Dynamische & sterke typering
x = "123"
x = 456      # OK: variable kan van type veranderen
"123" + 456  # TypeError: conversie van type is wel niet mogelijk
```

Basic Types in Haskell

Numeric Types

```haskell
Int        -- Fixed-precision integer
Integer    -- Arbitrary-precision integer
Float      -- Single-precision floating point
Double     -- Double-precision floating point
Rational   -- Arbitrary-precision rational numbers
```

Character and String Types

```haskell
Char       -- Unicode character
String     -- List of characters ([Char])
```

Andere Basis Data Types

```haskell
Bool       -- Boolean type (True/False)
Ordering   -- Comparison result (LT, EQ, GT)
```

Container Types

Lists

```
[]          -- Lege lijst type
[a]         -- Lijst van elementen van type a. Elk element is van hetzelfde type!
```

Tuples

```
(,)         -- 2-tuple (pair)
(,,)        -- 3-tuple
(,,,)       -- 4-tuple
```

Tot wel tuples met 15 elementen!

`type` declaraties

- Met het keyword `type` kunnen we types een andere naam geven
- of een specieke naam voor een samengesteld type definieren

```
1   type String = [Char]
2   type triplet = (Int, Char, Float)
3   type multiplet = triplet -> triplet -> triplet
```

`data` type constructie

Het keyword `data` laat toe nieuwe types te maken

```haskell
data MyBool = False | True

data Shape = Circle Float | Rect Float Float

data Grades = A | B | C | D | E | Unknown

gradeList :: [Grades]
gradeList = [A,B,C,Unknown,D,A,A,Unknown,B,C,C]

nrTopScores :: [Grades] -> Int
nrTopScores gradeList = countA gradeList

-- Pattern matching: drie mogelijke opties
countA [] = 0
countA (A:xs) = 1 + countA xs
countA (_:xs) = countA xs -- "_" maakt niet uit
```

Het Maybe Type

Wat is Maybe?

```haskell
data Maybe a = Nothing | Just a -- gedefinieerd in Prelude
```

Een `Maybe` is een container die:

- Ofwel een waarde bevat (`Just a`)
- Ofwel leeg is (`Nothing`)
- `Just` is een functie (een soort van constructor) die een waarde van een type a teruggeeft na een succesvolle berekening
- `Just` is vaak onderbelicht in Haskell boeken en tutorials, maar het zorgt dat er duidelijkheid is of er "iets van resultaat is" (`Just resultaat`) of "niets is" (`Nothing`).

Het Maybe Type

Wanneer gebruik je Maybe?

- Als een functie mogelijk geen resultaat heeft
- Om null-pointer exceptions te voorkomen

```
1  veiligeDelingM :: Int -> Int -> Maybe Int
2  veiligeDelingM _ 0 = Nothing  -- Division by zero is not allowed
3  veiligeDelingM x y = Just (x `div` y)
4
5  ghci> veiligeDelingM 10 2
6  Just 5
7  ghci> veiligeDelingM 10 0
8  Nothing
```

Voorbeelden van Maybe

Veilige Lijst Operaties

```haskell
-- Onveilige head functie
head [1,2,3]      -- Geeft 1
head []           -- Crash!

-- Veilige versie met Maybe
safeHead :: [a] -> Maybe a
safeHead []     = Nothing
safeHead (x:_)  = Just x

ghci> safeHead [1,2,3]
Just 1
ghci> safeHead []
Nothing
```

Het Either Type

Wat is Either?

```haskell
data Either a b = Left a | Right b
```

Een `Either` is een container die:

- Ofwel een waarde van type `a` bevat (`Left a`)
- Ofwel een waarde van type `b` bevat (`Right b`)

Conventie

- `Left` wordt meestal gebruikt voor fouten
- `Right` wordt meestal gebruikt voor succesvolle resultaten
- "Right is right" (Right is correct)

Voorbeelden van Either

Foutafhandeling

Nu met duidelijkere foutafhandeling dan met Maybe.

```
veiligeDeling :: Int -> Int -> Either String Int
veiligeDeling _ 0 = Left "Kan niet delen door nul!"
veiligeDeling x y = Right (x `div` y)

ghci> veiligeDeling 10 2
Right 5
ghci> veiligeDeling 10 0
Left "Kan niet delen door nul!"
```

Praktisch Gebruik van Either

Foutmeldingen met Context

```haskell
-- Definieer een nieuw type ``Fout'' dat afgeleid wordt van Show
data Fout =
    OngeldigBedrag Double
  | OntoereikendSaldo Double
  deriving Show

bankTransactie :: Double -> Double -> Either Fout Double
bankTransactie saldo bedrag
  | bedrag <= 0 = Left (OngeldigBedrag bedrag)
  | bedrag > saldo = Left (OntoereikendSaldo saldo)
  | otherwise = Right (saldo - bedrag)
```

```
ghci> bankTransactie 100 50
Right 50.0
ghci> bankTransactie 100 150
Left (OntoereikendSaldo 100.0)
ghci> bankTransactie 100 (-10)
Left (OngeldigeBedrag -10.0)
```

Maybe vs Either

Wanneer gebruik je wat?

Maybe

- Als je alleen wil weten **of** er een waarde is
- Bij simpele ja/nee situaties
- Als je geen extra informatie nodig hebt bij falen

Either

- Als je wilt weten **waarom** iets faalde
- Bij complexe foutafhandeling
- Als je verschillende soorten fouten wilt onderscheiden

Common Type Classes

Basic Classes

```
Eq          -- Types met "equality" support
Ord         -- Types die kunnen geordend worden
Show        -- Types die naar een String kunnen worden omgezet
Read        -- Types die uit een String kunnen gehaald worden
```

Numeric Classes

```
Num         -- Numeric types
Real        -- Real numbers
Integral    -- Integral numbers
Fractional  -- Fractional numbers
Floating    -- Floating point numbers
```

Numerieke Type Classes in Haskell

Num

- Basis numerieke type class
- Voor alle getallen (geheel én met komma)
- Ondersteunt basisbewerkingen: `+`, `-`, `*`
- Voorbeelden: `Int`, `Integer`, `Float`, `Double`

```
ghci> :info Num
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
```

Real en Integral

Real

- Voor alle reëele getallen
- Getallen die je kunt ordenen
- Omvat gehele en kommagetallen

Integral

- Specifiek voor gehele getallen
- Ondersteunt gehele deling (`div`) en modulo (`mod`)
- Voorbeelden: `Int` en `Integer`
- Functies specifiek voor gehele getallen

Fractional en Floating

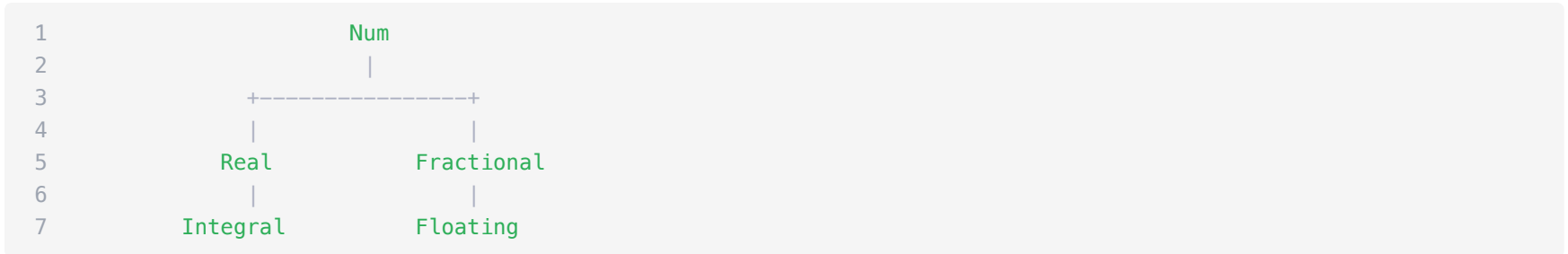
Fractional

- Voor getallen met komma’s
- Ondersteunt deling (`/`)
- Voorbeelden: `Float` , `Double` , `Rational`

Floating

- Voor kommagetallen
- Ondersteunt geavanceerde wiskundige operaties
- Bijvoorbeeld: `sin` , `cos` , `tan` , `sqrt`
- Types: `Float` en `Double`

Hiërarchie van Numerieke Types



Voorbeeld:

```haskell
-- Werkt met alle Num types
optellen :: Num a => a -> a -> a
optellen x y = x + y

-- Alleen voor gehele getallen
isEven :: Integral a => a -> Bool
isEven x = x `mod` 2 == 0

-- Alleen voor kommagetallen
wortel :: Floating a => a -> a
wortel = sqrt
```

Function Type Definitions

De notatie `naam :: a -> b ...` specifieert de types waarop de functie werkt.

- `naam` is de naam van de expressie (of functie)
- `::` is de *type operator*, kan je lezen als "`naam` heeft type `a -> b -> c`"
- `->` is een *function type constructor* , en is rechts asociatief. `a -> b -> c -> d` is dus hetzelfde als `a -> (b -> (c -> d))`

```haskell
succ :: Int -> Int              -- Functie succ van Int naar Int

convert :: Char -> Int -> Int   -- Functie convert die eerst een Char accepteert,
                                -- dan een Int, en een Int teruggeeft

listgen :: (Int,Int) -> [Int]   -- Functie listgen die een tuple van twee Ints accepteert
                                -- en een lijst van Ints teruggeeft

factorial :: Integer -> Integer
factorial n = product [1..n]

map :: (a -> b) -> [a] -> [b]   -- hogere orde functie type
```

Function Type Definitions voor Polymorfe Functies

- Polymorfe functies kunnen op meerdere types werken (vergelijkbaar met generics)
- Type vaak als een letter, waarbij het polymorfe type verduidelijkt wordt voor `=>`

```haskell
(+) :: Num a => a -> a -> a -- Polymorfe functie die twee getallen mapt naar een getal

fibs :: Num a => [a]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)


tf :: Num a => Int -> [a]
tf x = take x fibs
```

Function Type Definitions en Curried Functions

```
1   driePlus :: Int -> Int -> Int -> Int
2   driePlus x y z = x + y + z
3
4   ghci> driePlus 1 2 7
5   10
6   ghci> (driePlus 1) 2 7
7   10
8   ghci> ((driePlus 1) 2) 7
9   10
10  ghci> driePlus (1 2 7)
11    <interactive>:6:11: error:
```

- `(driePlus 1)` is een expressie van het type `:: Int -> Int -> Int`
- `((driePlus 1) 2)` is een expressie van het type `:: Int -> Int`
- *Curried functions* : Elke functie in Haskell gecombineerd met één element, wordt het dan een nieuwe functie
- `driePlus` is een functie met drie parameteres, maar `(driePlus 1)` is ook een functie die twee parameters neemt, en `((driePlus 1) 2)` is ook een functie die één parameter neemt

Nog een voorbeeldje van Curried functies

```
1   add :: Int -> Int -> Int
2   add x y = x + y
3
4   ghci> add 1 2
5   3
6   ghci> (add 1) 2
7   3
8
9   ghci> plus1 = add 1
10  ghci> plus1 2
11  3
```

Meer informatie over een type, class of functie

met `ghci>:info TYPENAAM` kan je gedetialleerde achtergrond informatie opvragen over type definities, classes en functies, waar ze gedefinieerd worden, welke constructor er kan gebruikt worden, en hoe ze in elkaar zitten.

```
ghci> :info Bool
type Bool :: *
data Bool = False | True
    -- Defined in ‘GHC.Types’
instance Bounded Bool -- Defined in ‘GHC.Enum’
instance Read Bool -- Defined in ‘GHC.Read’
instance Enum Bool -- Defined in ‘GHC.Enum’
instance Show Bool -- Defined in ‘GHC.Show’
instance Eq Bool -- Defined in ‘GHC.Classes’
instance Ord Bool -- Defined in ‘GHC.Classes’
```

Type Informatie opvragen van een expressie

Met `:type` kan je de type definitie opvragen.

In GHCi:

```
ghci> :type driePlus
driePlus :: Int -> Int -> Int -> Int
ghci> :type filter
filter :: (a -> Bool) -> [a] -> [a]
```

Nog enkel Voorbeelden met "::"

Complexere Types

```haskell
-- Een lijst van tuples
coordinaten :: [(Int, Int, Int)]
coordinaten = [(1,2,5), (3,4,5), (5,6,5)]

-- Een functie die met lijsten werkt
eersteElement :: [a] -> Maybe a
eersteElement [] = Nothing
eersteElement (x:_) = Just x
```

Type Classes

```haskell
-- Een getal omzetten naar string (werkt voor alle Show types)
naarString :: Show a => a -> String
naarString x = show x

-- Twee waarden vergelijken (werkt voor alle Eq types)
zijnGelijk :: Eq a => a -> a -> Bool
zijnGelijk x y = x == y
```

Samengestelde Types: Lists en Tuples

```haskell
-- Lists (homogeneous)
[1, 2, 3] :: [Int]
["a", "b", "c"] :: [String]

-- Tuples (heterogeneous)
(True, 42) :: (Bool, Int)
(1, "Hello", False) :: (Int, String, Bool)

-- Nested structures
[(1,2), (3,4)] :: [(Int, Int)]
([1,2], ['a','b']) :: ([Int], [Char])
```

Type Inference

Haskell kan zelf types afleiden

```
-- Type inference in action
ghci> double x = 2*x
ghci> :type double
double :: Num a => a -> a

ghci>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> :type caesar
caesar :: Int -> [Char] -> [Char]

ghci> upOne x = map (\x -> x + 1) [1,2,3]
ghci> :type upOne
upOne :: Num b => p -> [b]

ghci> :type map (\x -> x + 1) [1,2,3]
map (\x -> x + 1) [1,2,3] :: Num b => [b]

ghci> upOne xs =  map (\x -> x + 1) xs
ghci> :type upOne
upOne :: Num b => [b] -> [b]
```

Common Type Classes - werken allen polymorf

Gelijkheid tussen waardes van hetzelfde type (zoals de interface Comparable in Java)

```
1   (==) :: Eq a => a -> a -> Bool
```

Ordening tussen waardes van hetzelfde type:

```
1   (<) :: Ord a => a -> a -> Bool
```

Conversie naar een string van waardes van een type `a`

```
1   show :: Show a => a -> String
```

Optellen van twee waardes van hetzelfde numerieke type

```
1   (+) :: Num a => a -> a -> a
```

Conditional Expressions

Eindelijk, een if-then-else:

```
1   -- if-then-else
2   abs :: Int -> Int
3   abs n = if n >= 0 then n else -n
```

Je kan ze bovendien nesten (maar beperkt dat toch maar):

```
1   -- Nested conditions
2   signum :: Int -> Int
3   signum n = if n < 0 then -1 else
4              if n == 0 then 0 else 1
```

Guards

- Guards gebruiken het pipe-symbool (`|`) gevolgd door een booleaanse expressie en het resultaat wanneer die expressie waar is.
- Guards worden van boven naar beneden gecontroleerd, waarbij de eerste overeenkomende guard het resultaat bepaalt.
- De `otherwise` guard (gedefinieerd als `otherwise == True`) kan dienen als een vangnet aan het einde.

Guards

```
1   abs n | n >= 0    = n
2         | otherwise = -n
```

Guards

Meerdere guards, zoals cases bij een switch statement, maar dan met krachtige patroon matching;

```haskell
grade :: Int -> String
grade n | n >= 90  = "A"
        | n >= 80  = "B"
        | n >= 70  = "C"
        | n >= 60  = "D"
        | otherwise = "F"
```

```haskell
describeMaybeNumber :: Maybe Int -> String
describeMaybeNumber Nothing = "No number provided"
describeMaybeNumber (Just n)
  | n < 0     = "Negative number: " ++ show n
  | n == 0    = "Zero"
  | n > 100   = "Large number: " ++ show n
  | otherwise = "Regular number: " ++ show n
```

Where Clauses

`where` *bindt een resultaat aan een variabele* binnen de scope van de functie, en wordt *lazy* geëvalueerd

```
1   initials :: String -> String -> String
2   initials firstname lastname = [f,l]
3       where f = head firstname
4             l = head lastname
```

Bijvoorbeeld: Bereken de nulpunten voor

$$ax^2 + bx + c = 0$$

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
1   roots :: Float -> Float -> Float -> (Float,Float)
2   roots a b c = ((-b + sqrt(disc))/2*a,
3                  (-b - sqrt(disc))/2*a)
4       where disc = b*b - 4*a*c
```

`let ... in`: zoals where, maar nog lokaler

- `let` is een expressie(!), `where` is louter een "binding" van een variabele aan een berekening
- `let` is enkel geldig op het gedeelte acher `in`
- `let` kan je dus eender waar in je code als shorthands gebruiken

```haskell
-- Using let
cylinder :: Float -> Float -> Float
cylinder r h = let area = pi * r^2
                   volume = area * h
               in volume

-- let in list comprehensions
[let sq x = x*x in sq n | n <- [1..5]]
```

Pattern Matching en `_`

- Haskell selecteert de juiste functie door pattern matching met de argumenten
- Als de waarde van een argument niet uitmaakt, kan je `_` gebruiken

```haskell
-- Pattern matching op waardes
not :: Bool -> Bool
not True  = False
not False = True

-- Pattern matching op lijsten
head :: [a] -> a
head (x:_) = x
head []    = error "Empty list"

-- Pattern matching op tuples
fst :: (a,b) -> a
fst (x,_) = x
```

Case Expressions

Case expressies ondersteunen dezelfde pattern matching mogelijkheden als functies

```
case expression of patroon1 -> expressie
                   patroon2 -> expressie
                   ...
```

```
head' :: [a] -> a
head' xs = case xs of
             []    -> error "Empty list"
             (x:_) -> x

-- Pattern matching with case
describeList :: [a] -> String
describeList xs = case xs of
                    [] -> "Empty"
                    [x] -> "Singleton"
                    xs -> "Longer list"
```

Case Expressions

Merk op dat met functie parameters pattern matching en where expressies met functie definities je hetzelfde kan doen:

```haskell
describeList :: [a] -> String
describeList xs = what xs
  where
    what [] = "empty"
    what [x] = "a singleton list"
    what xs = "a longer list"
```

Pattern Matching op Maybe

```haskell
toonEersteElement :: Show a => [a] -> String
toonEersteElement xs = case safeHead xs of
    Nothing -> "De lijst is leeg!"
    Just x  -> "Eerste element is: " ++ show x

-- Voorbeeld gebruik
ghci> toonEersteElement [1,2,3]
"Eerste element is: 1"
ghci> toonEersteElement []
"De lijst is leeg!"


applyIfJust :: Maybe a -> (a -> b) -> Maybe b
applyIfJust Nothing _ = Nothing
applyIfJust (Just x) f = Just (f x)
```

Reading

- Learn You a Haskell (LYAH) Hoofdstukken 2, 3 & 4
- Programming in Haskell (PIH) Slides Chapter 3 & 4

Suggested Reading

- Real World Haskell: Hoofdstukken 2, 3 en 4 over Types en Functies[1]

Volgende les

- Recursive Functions
- Higher Order Functions

---

1. Hoofdstuk 3 gaat al iets verder met het opbouwen van recursieve data types `->` Dat komt aan bod in les 4.
   Hoofdstuk 4 bespreekt ook de beruchte `foldl` en `foldr` functie `->` Die komen aan bod in de volgende les. ↩