

1 Inleiding in Functioneel Programmeren

door Elroy Jumpertz

1.1 Inleiding

Aangezien Informatica een populaire minor is voor wiskundestudenten, leek het mij nuttig om een stukje te schrijven over een onderwerp dat voor wiskundigen interessant is, namelijk functioneel programmeren. Voor dit artikel beperk ik me tot de taal Clean, die door de faculteit Informatica is ontwikkeld.

Vrijwel iedereen heeft al eens wat geprogrammeerd, bijvoorbeeld in Java, PHP of Matlab, maar waar je misschien niet van op de hoogte bent is dat deze talen allemaal behoren tot de klasse der *imperatieve* talen. Deze programmeerstijl wordt gekenmerkt door de mogelijkheid om waarden aan variabelen toe te kennen, en de programmacode wordt van boven naar beneden uitgevoerd, statement voor statement. Daarnaast bestaat er een verzameling *functionele* talen. Functionele talen hebben geen variabelen en geen assignments; er bestaan uitsluitend functies die aan elkaar worden geregen.

Een ander belangrijk kenmerk van functionele talen is dat ze doorgaans statisch getypeerd zijn. Dat betekent dat de types van de gebruikte functies in compile-time bekend moeten zijn en op elkaar aan moeten sluiten. Compilers beschikken dan ook over een type-afleidingsmechanisme.

1.2 Recursie

Doordat je enkel met functies kunt werken en geen loops en globale variabelen hebt, kom je al snel terecht bij recursie. Een recursieve functie is simpelweg een functie die zichzelf aanroept. Als klassiek voorbeeld neem ik de faculteitsfunctie. Een recursieve wiskundige definitie is:

$$\text{fac} : \mathbb{N} \rightarrow \mathbb{N} : n \mapsto \begin{cases} 1 & n \leq 1 \\ n \cdot \text{fac}(n-1) & \text{anders} \end{cases}$$

In Clean zou je het als volgt opschrijven:

```
fac :: Int -> Int
fac n
| n <= 1 = 1
| otherwise = n * fac(n-1)
```

Het is meteen duidelijk hoe dicht de Clean-syntax bij de wiskunde ligt. De recursieve aanroep (`fac(n-1)`) zorgt ervoor dat het “probleemgebied” iets kleiner

wordt gemaakt (in dit geval een kleinere n) en dat dezelfde functie wordt gebruikt om een uitkomst te berekenen. De functie termineert zogauw n gelijk wordt aan 1; dan worden alle tussenresultaten vermenigvuldigd om de uiteindelijke uitkomst op te leveren.

1.3 Lijsten

In een functionele taal is het erg handig om met lijsten te werken. Een lijst is een rijtje van data van hetzelfde type, ongeveer zoals een array in andere talen. Ook lijsten zijn recursief opgebouwd: ze bevatten een kop en een staart, die op zijn beurt weer een lijst is. De lijst van Integers `[1, 2, 3, 4]` wordt dus intern gerepresenteerd als een lijst met een Integer 1 en een lijst `[2, 3, 4]`. Deze lijst bevat een 2 en de lijst `[3, 4]`, enzovoort. Notatie: `[1: [2: [3: [4: []]]]]`. Het symbool `[]` staat voor de lege lijst. Omdat lijsten van zichzelf een recursieve structuur hebben, lenen ze zich goed voor recursieve algoritmen. De volgende functie draait een lijst om:

```
reverse :: [a] -> [a]
reverse []      = []
reverse [x:xs] = reverse xs ++ [x]
```

Hier worden een aantal nieuwe concepten geïntroduceerd die nadere toelichting vereisen. Het type van de functie is `[a] -> [a]`. De `a` hierin is een *type-variabele* (een soort jokerteken), wat betekent dat de functie werkt op lijsten van elk willekeurig type. Dit mogen dus lijsten van Integers zijn, of van Reals, of van Bools, etc.

Daarnaast wordt er gebruik gemaakt van *pattern matching*. Ik definieer meerdere “versies” van `reverse`, afhankelijk van de data die wordt aangeboden. Als `reverse` wordt aangeroepen met een lege lijst, wordt er simpelweg een lege lijst teruggegeven. De tweede regel komt in actie als er een lijst met inhoud wordt aangeboden. De lijst “matcht” dan op `[x:xs]`. Dit is meteen de enige manier om in Clean met lijsten te werken: je hebt de kop (de `x`) en de rest (`xs`). Merk op dat `x` dus van het type `a` is en `xs` van het type `[a]`.

Het bijbehorende algoritme invertteert middels recursie eerst de staart van de aangeboden lijst (`reverse xs`) en plakt daar later het eerste element (`x`) achter. Omdat de `++` operator, die twee lijsten aan elkaar lijmt, van het type `[a] [a] -> [a]` is, moeten we twee lijsten aanbieden. `reverse xs` levert al een lijst op (kijk maar naar de functiedefinitie), maar van `x` moeten we even een singleton-lijst `[x]` maken; dat is een lijst met maar één element erin.

1.4 Hogere-orde functies

Het volgende onderdeel waarin een functionele taal verschilt van zijn imperatieve broeders, is hogere-orde functies. Functies die werken op argumenten van een bepaald primitief type (Ints, Chars) zijn eerste-orde functies. Functies die andere functies als argument hebben of die een functie opleveren, heten hogere-orde functies. Het favoriete voorbeeld van elke functioneel programmeur is de functie `map`, die een andere functie toepast op alle elementen van een lijst:

```
map :: (a -> b) [a] -> [b]
map f []      = []
map f [x:xs] = [f x: map f xs]
```

Hier wordt duidelijk dat `map` twee argumenten wil hebben: een functie van type `a -> b` en een lijst `[a]`. Het resultaat is een lijst `[b]`. In de functie met type `a -> b` worden twee verschillende typevariabelen gebruikt om aan te geven dat het een functie mag zijn die een parameter van een willekeurig type accepteert, en een ander willekeurig type oplevert. Deze types hoeven dus niet hetzelfde te zijn (maar het mag wel!).

In het algoritme wordt de functie `f` recursief op de lijst `[x:xs]` uitgevoerd door eerst `f` los te laten op `x`, en de staart van de op te leveren lijst recursief af te handelen. Merk op dat een functie-aanroep geen haakjes vereist. `f x` betekent dus: pas `f` toe op `x`. Zo is `map f xs` een aanroep van `map` met als argumenten `f` en `xs`. Aan het einde van het liedje is de hele oorspronkelijke lijst getransformeerd naar iets van het type `[b]`.

Een voorbeeld is hier wellicht op zijn plaats. Als input-lijst neem ik `[1, 2, 3, 4, 5]` en als functie `isEven`. Deze functie is van het type `Int -> Bool` en bepaalt van een Integer of het een even getal is. Voorbeeld: `isEven 3 = False`. Nu geldt:

```
map isEven [1, 2, 3, 4, 5] = [False, True, False, True, False]
```

Ik heb dus een functie van type `Int -> Bool` gegeven, plus een lijst van `Int`, en ik krijg een lijst van `Bool`. Handig toch?

1.5 Termreductie

De manier waarop de Clean-compiler programma's evalueert is door middel van termreductie. Dit kan het beste worden uitgelegd met een voorbeeld. De expressie `1 + 2 * 3` wordt gereduceerd naar `1 + 6` en dat wordt gereduceerd naar `7`. In dit geval is er maar één manier waarop er gereduceerd kan worden, maar vaak kan het op meerdere manieren. Stel we hebben een functie die een kwadraat berekent:

```
square :: Int -> Int
square x = x * x
```

Dan kan de expressie `square (1 + 2)` als volgt worden gereduceerd:

```
square (1 + 2)    (definitie van +)
  =====
= square 3        (definitie van square)
  =====
= 3 * 3          (definitie van *)
  =====
= 9
```

Ik onderstreep telkens het deel van de expressie dat ik ga reduceren. Deze reduceerbare expressie wordt heel creatief een *redex* genoemd. Naast elke reductiestap noteer ik welke regel ik toepas. De uitkomst is 9, en omdat deze expressie niet verder gereduceerd kan worden heet dit een *normaalvorm*. Maar we kunnen onze expressie ook anders reduceren:

```
square (1 + 2)    (definitie van square)
  =====
= (1 + 2) * (1 + 2) (definitie van +)
  =====
= 3 * (1 + 2)     (definitie van +)
  =====
= 3 * 3          (definitie van *)
  =====
= 9
```

Zoals er vele wegen naar Rome leiden, zijn er meerdere reductiepaden die tot een normaalvorm leiden. Clean kent een eigenschap die ervoor zorgt dat *als* een expressie een normaalvorm heeft, deze altijd hetzelfde is, ongeacht het gekozen reductiepad.

1.6 Correctheidsbewijzen

Tot slot wil ik het nog even hebben over correctheidsbewijzen. Het is af en toe wel fijn als je op formele wijze een bepaalde eigenschap van je programma kunt bewijzen; dan weet je immers zeker dat een functie of een programma doet wat het moet doen en heb je uiteindelijk minder bugs. In imperatieve talen is dat nogal een omslachtig proces maar in een functionele taal is dat een stuk eenvoudiger. De strikte typering en de recursieve opbouw van de code pleiten al snel voor inductie. Laten we een voorbeeld erbij nemen.

Stel dat we een nieuwe operator `o` introduceren (deze operator zit standaard al in Clean, maar we houden ons even van de domme). Dit is het bolletje dat je kent uit de wiskunde en dat wordt gebruikt voor functiecompositie. We definiëren onze operator als volgt (het type is even niet van belang):

$$(f \circ g) x = f (g x) \quad (1)$$

Ik nummer de definitie zodat ik er later naar kan verwijzen. Daarnaast nemen we nogmaals de definitie van `map`:

$$\text{map } f [] = [] \quad (2)$$

$$\text{map } f [x:xs] = [f x: \text{map } f xs] \quad (3)$$

We zouden nu de volgende eigenschap kunnen bewijzen:

$$\text{map } (f \circ g) xs = \text{map } f (\text{map } g xs)$$

voor alle eindige lijsten `xs` en alle functies `f` en `g`.

Het bewijs gaat met inductie over de lengte van `xs`. Allereerst bewijzen we de eigenschap voor de lege lijst (de inductiebasis). Daarna nemen we aan dat de eigenschap geldt voor lijsten van lengte `n` en concluderen daaruit dat de eigenschap ook geldt voor lijsten van lengte `n+1` (de inductiestap). We maken hierbij gebruik van de genummerde functiedefinities die hierboven staan. We gaan op zoek naar syntactische gelijkheid door expressies te reduceren.

Inductiebasis

Aanname: `xs = []`

Te bewijzen: `map (f o g) [] = map f (map g [])`

Bewijs:

$$\begin{aligned} & \text{map } (f \circ g) [] && \text{(definitie 2)} \\ & \text{=====} \\ & = [] && \text{(2) (van rechts naar links)} \\ & == \\ & = \text{map } f [] && \text{(2) } \leq \\ & == \\ & = \text{map } f (\text{map } g []) \end{aligned}$$

...en dat is wat we in deze stap moesten bewijzen. Vervolgens nemen we aan dat onze eigenschap al geldt voor een willekeurige lijst `xs` en benoemen dit tot de inductiehypothese:

$$\text{map } (f \circ g) xs = \text{map } f (\text{map } g xs) \quad (\text{I.H.})$$

Wat we in de inductiestap gaan doen, is het bewijs leveren voor de lijst `[x:xs]`; dat is de lijst `xs` met één element meer.

Inductiestap

Aanname: `xs = [x:xs]`

Te bewijzen: `map (f o g) [x:xs] = map f (map g [x:xs])`

Bewijs:

$$\text{map } (f \circ g) [x:xs] \quad (3)$$

```

=====
= [(f o g) x: map (f o g) xs] (I.H.)
=====
= [(f o g) x: map f (map g xs)] (1)
=====
= [f (g x): map f (map g xs)] (3) <=
=====
= map f [g x: map g xs] (3) <=
=====
= map f (map g [x:xs])

```

... waarmee we ook de inductiestap hebben bewezen. Per inductie hebben we nu aangetoond dat de eigenschap geldt voor alle eindige lijsten `xs`.

1.7 Het reclamepraatje

Ik heb mijn best gedaan om op zéér beknopte wijze enkele interessante eigenschappen van functioneel programmeren onder de aandacht te brengen. Nu rest mij enkel nog een schaamteloos reclamepraatje voor onze fijne informaticafaculteit. De volgende tekst moet worden beschouwd als een grote knipperende neonreclame...

De bachelorcursus Functioneel Programmeren wordt elk jaar aangeboden bij Informatica en kan prima deel uitmaken van je bijvakpakket naast de opleiding Wiskunde. Als je meer informatie wilt hebben, aarzel dan niet om mij een mailtje te sturen: ***.