Functioneel programmeren

1. Berekenbaarheid: twee modellen In de wiskunde definieer je begrippen (bijvoorbeeld getallen, meetkundige figuren), je rekent ermee (kwadrateren, bepaling van de oppervlakte of inhoud) en je bewijst er dingen over (ieder getal is de som van vier kwadraten, de som van de hoeken van een driehoek in het platte vlak is altijd 180 graden). De vraag hoe je precies een definitie opschrijft en een waterdicht bewijs levert was reeds in de oudheid door de Griekse filosoof Aristoteles gesteld. Een grondig inzicht in dat definiëren en bewijzen werd gegeven door de Duitse logicus Frege aan het einde van de 19-de eeuw. Vreemd genoeg werd het begrip berekenbaarheid pas in de jaren 1930 diepgaand geanalyseerd, terwijl het rekenen al bekend was in de Egyptische, Chinese en Babyloonse wiskunde duizenden jaren geleden en tot grote hoogte werd verheven in de 19-de eeuw (o.a. door de Duitse wiskundigen Gauss en Jacobi).

Rond 1900 ontbrak het volgende aan inzicht in het berekenen. Zodra men iets kon berekenen, schreef men een recept (een zogenaamd algoritme) hiervoor op. Feit is echter dat het soms niet lukte om een probleem op te lossen door rekenen. De vraag is dan of dat komt omdat we niet goed genoeg gezocht hebben, of dat sommige problemen nooit en te nimmer door een algoritme kunnen worden opgelost. Misschien is het zo dat alle nauwkeurig geformuleerde problemen wél door een berekening opgelost kunnen worden? Deze vraag, voor het eerst gesteld rond 1685 door de Duitse wiskundige Leibniz, kon men nog niet oplossen. Een voorbeeld van een goed geformuleerd probleem is het volgende. Gegeven het natuurlijke getal n,

$$\text{zijn er natuurlijke getallen } x, y, z > 0 \text{ zodat } 4xyz = n(xy + yz + zx)? \tag{1}$$

Om dit soort vragen ontkennend te kunnen beantwoorden is het nodig het intuïtive begrip berekenbaarheid precies te kennen.

In het midden van de jaren 1930 werden er maar liefst twee voorstellen gedaan wat berekenbaarheid precies inhoudt. De Amerikaanse logicus Church stelde voor dat berekenbaarheid 'gevangen' kan worden met een door hem geïntroduceerde formele taal van functies, de *lambda calculus*. De Engelse logicus Turing stelde voor dat berekenbaarheid gevangen wordt door een heel eenvoudig mechanisch principe. Zijn voorstel kwam iets later dan dat van Church. Bovendien liet Turing zien, dat beide voorstellen op hetzelfde neerkomen. Church en Turing lieten het niet hierbij; zij gaven beiden een voorbeeld van een goed gesteld probleem dat geen berekenbare oplossing heeft, een zogenaamd onoplosbaar probleem. Die problemen waren gesteld in hun eigen formalisme (van functies, respectievelijk van machines). Later kon men ook van bepaalde wiskundige problemen aantonen dat ze onoplosbaar zijn. Bijvoorbeeld liet de Russische wiskundige Matijasevic in 1970 zien dat er Diophantische problemen zijn van de vorm (1), welke onoplosbaar zijn.

De definitie van berekenbaarheid van Turing heeft aanleiding gegeven tot de programmeerbare computer en de zogenaamde imperatieve programmeertalen, zoals C en Java(script). Hiermee schrijf je programma's waarbij de belangrijkste opdracht bestaat uit het veranderen van een waarde in een bepaalde geheugenplaats. Bijvoorbeeld

$$\texttt{x} := 7 \text{ of } \texttt{x} := \texttt{x} + 1.$$

Het model van Church is heel anders. Het werkt met functies, eenvoudig of complex, en hun samenstellingen. Dit heeft aanleiding gegeven tot functioneel programmeren.

2. Hoe werkt FP? *Data.* Hoewel gehele getallen en floating-point ('wetenschappelijke') reële getallen gedefinieerd kunnen worden in de lambda calculus, worden deze ter wille van de efficiëntie ingevoerd als speciale constanten met primitieve functies voor de standaard operaties.

Herschrijven. In het functioneel programmeren worden uitdrukking stapsgewijs herschreven met behoud van betekenis. In dit proces van herschrijven zijn er doorgaans meerdere keuzes. Maar het is zo dat indien er een uitkomst komt, dan is deze uniek. Een eenvoudig voorbeeld: $(3+2)\times(5+7)$ kan op meerdere manieren systematisch herschreven worden tot 5×12 en dan tot het unieke antwoord 60. We noteren dit als volgt

$$\begin{array}{rcl}(3+2)\times(5+7) & \to & 5\times\ (5+7)\\ & \to & 5\times\ 12\\ & \to & 60\end{array}$$

of ook als $(3+2)\times(5+7)\ \twoheadrightarrow\ 60$

Toepassen. Een belangrijke eigenschap van functionele programma's is dat twee uitdrukkingen op elkaar *toegepast* kunnen worden. Een functioneel programma bestaat uit een uitdrukking, in principe van de vorm '`F A`'. Daarbij representeert `F` een functie en `A` het argument van die functie. We zeggen dat `F` wordt *toegepast* op `A`. Meer in het algemeen is zo'n uitdrukking van de vorm '$\mathtt{F\ A_1\cdots A_k}$', waarbij de functie `F` meerder argumenten heeft. Zowel `F` als de `A`'s hebben dezelfde status. Dat betekent dat functies ook op functies toegepast mogen worden. Dat zijn dan de zogenaamde hogere orde functies. Bijvoorbeeld, op de vijf elementen van de lijst $(1,2,3,4,5)$ kunnen we de functie `square` toepassen (je krijgt dan $(1,4,9,16,25)$), maar ook de functie `cube`, tot de derde macht verheffen. Je krijgt dan $(1,8,27,64,125)$. In functionele programmeer talen kun je de hogere orde functie `map` definiëren, met als herschrijf gedrag

$$\mathtt{map\ f}\ (\mathtt{a_1},\mathtt{a_2},\mathtt{a_3},\mathtt{a_4},\mathtt{a_5})\twoheadrightarrow(\mathtt{f\ a_1},\mathtt{f\ a_2},\mathtt{f\ a_3},\mathtt{f\ a_4},\mathtt{f\ a_5}).$$

Daarmee kun je de functies '`map square`' en '`map cube`' kunnen krijgen, welke voldoen aan

$$\begin{array}{rcl}\mathit{map\ square}\ (1,2,3,4,5) & \twoheadrightarrow & (1,4,9,16,25)\\ \mathit{map\ cube}\ (1,2,3,4,5) & \twoheadrightarrow & (1,8,27,64,125).\end{array}$$

Abstractie. Daarnaast is er abstractie. Dit geeft de mogelijkheid complexe functies in te voeren. Voor uitdrukkingen `F` en `G`, welke functies voorstellen, kun je vormen $F\circ G$ en $G\circ F\circ G$ met de herschrijf regels

$$\begin{array}{rcl}(F\circ G)\ a & \to & F(G\ a);\\ (G\circ F\circ G)\ a & \to & G(F(G\ a)).\end{array}$$

Universaliteit. In de door Church uitgevonden lambda calculus kan iedere computer taak op een dergelijke manier gerepresenteerd worden als uitdrukking die herschreven wordt met behoud van betekenis.

Berekeningen en processen. Bij berekeningen wordt de expressie aan het begin herschreven, totdat dit niet meer mogelijk is. Daarnaast is het ook mogelijk doorgaande *processen* functioneel te programmeren. Bij deze wordt de begin uitdrukking voortdurend herschreven, zonder dat dat ophoudt. Daarbij worden er voortdurend nuttige dingen gedaan. Denk aan een proces dat de werking van een fabriek volautomatisch bestuurt (waarbij het altijd goed is om op een noodstop te kunnen drukken indien dat wenselijk is), of aan het proces dat het besturingssysteem van een computer uitvoert.

Typen. Een belangrijk hulpmiddel voor de compacte duidelijke weergave vormen de zogenaamde *typen*. In de natuurkunde hebben constanten een 'dimensie'. Snelheid wordt bijvoorbeeld gemeten in km/h. Als we met snelheid $\mathtt{v}=\mathtt{12km/h}$ per uur fietsen en dit gedurende $\mathtt{t}=\mathtt{3h}$ doen, dan zijn we $\mathtt{vt}=\mathtt{12}\cdot\mathtt{3}=\mathtt{36km}$ verder. De dimensies verhinderen dat we bijvoorbeeld $\mathtt{vt}^2$ opschrijven om de afstand uit te rekenen.

Analoog hieraan hebben de uitdrukkingen in de meest gebruikte functionele programmeer taken een typeringssysteem, dat help om modules op correcte wijze samen te stellen. Een programma dat bijvoorbeeld een getal van type `Int` verwacht mag niet op een waarheidswaarde (`True`, `False`) van type `Bool` toegepast worden.

Als we een module `F` het type `A` toekennen, dan wordt dit genoteerd als `F : A`[1]. Typering wordt nu verkregen door aan de basis data typen toe te kennen, bijvoorbeeld `3 : Int`, `True : Bool`. Daarna krijgen functies met gedrag `G x = y` als type `A → B`, waar `x : A` en `y : B`. Een toepassing `F a` is slechts toegestaan indien de typen overeenkomen, bijvoorbeeld `F : A → B` kan op `a : A` toegepast worden.

3. Grote voordelen en kleine nadelen. *Compact en duidelijk.* De 'code' (programma's) voor berekeningen en processen kunnen als functioneel programma veel compacter weergegeven worden dan als imperatief programma. Dat komt door de hogere orde functies, zoals `map` boven gedefinieerd. Naast de compacte weergave is ook de begrijpelijkheid van de uitdrukkingen belangrijk. De typen spelen een belangrijke rol om de compacte modules begrijpelijk te houden en om ze op correcte manier samen te stellen tot een groter programma. In veel gevallen hoeft de programmeur de typen niet zelf aan te geven, maar kunnen die afgeleid worden.

Wiskundig denken. De compactheid van de code kan het beste geapprecieerd worden indien men gewend is om wiskundig te denken. Dat houdt in dat men situaties kan weergeven met weglating van overbodige details. Dit vergt een zekere aanleg en training.

Input/output. In toepassingen heeft men input/output mogelijkheden nodig om informatie die via de randapparatuur[2] beschikbaar is te kunnen manipuleren. Deze informatie noemen we de 'toestand' van de computer en zijn randapparatuur welke een programma draaien. Het effect kan zijn dat de inhoud van een bestand gelezen of uitgebreid wordt. In zuivere functionele programmeertalen moet men input/output op een speciale manier behandelen, terwijl men toch modulariteit wil handhaven. Men doet dat door aan de toestand een type toe te kennen dat 'State' genoemd wordt. Voor output verandert men de toestand ('schrijven') en voor input kan men een bepaalde waarde van de toestand beschikbaar stellen aan het hoofdprogramma ('lezen'). Het functionele programma heeft geen toegang tot de toestand zelf, alleen tot de acties die dit schrijven en lezen uitvoeren. Zou men wel toegang tot de toestand '`t`' hebben, dan kan men met het programma

$$(\texttt{schrijf 1 t}, \texttt{schrijf 2 t})$$

tegenstrijdige opdrachten geven, waarbij de uitkomst niet meer uniek is.

Genoemde nadelen houden in dat het leren van functioneel programmeren iets lastiger is dan dat van imperatief programmeren. Heeft men het eenmaal geleerd, dan is het veel gemakkelijker en sneller om op de functionele manier complexe programma's op correcte wijze te schrijven.

4. Toekomst en perspectief *Parallellisme.* Zuivere functionele talen zijn beter uitgerust om multi-cores te programmeren dan de imperatieve talen. Dit komt omdat het resultaat berekening van een functie op een bepaalde waarde onafhankelijk is van de uitvoering ervan. Daarom kunnen modules in een functionele taal veilig in parallel uitgevoerd worden. Het uitbesteden van werk aan een andere processor kost tijd en dient daarom alleen te gebeuren bij substantiële verwerkingstaken. Men verwacht op deze manier computers met meerdere processoren goed te kunnen uitbuiten, zie [3].

Correctheid. Een aantal functionele talen zijn voorzien van een formalisme om uitspraken mee te formuleren. Daarin kan men een programma volledig specificeren. Wanneer men nu

[1] lees '`F` in `A`'.

[2] Bestanden, toetsenbord en muis voor de input en het scherm en wederom de bestanden als output.

ook kan bewijzen dat dat programma aan de specificaties voldoet en bovendien het bewijs door een programma dat dit kan verifieren laat 'nakijken', dan bereikt men de hoogst mogelijke betrouwbaarheid. Deze methode wordt reeds uitgebreid toegepast bij de constructie van hardware. Recentelijk heeft men ook voor een geoptimaliseerde versie van de kern van een C-compiler een dergelijke certificering weten te verkrijgen, zie [2].

Geschiedenis De eerste functionele taal was Lisp, [5], met als moderne variant Scheme, [1]. Het gebruikt geen typering en input/output wordt imperatief gedaan. De taal ML, [6], met als moderne varianten CaML, [7] and F#, [9], zijn eveneens niet zuiver functioneel, maar wel getypeerd. De taal Miranda, [10], was eebn van de eerste zuivere functional programmeer talen, met Clean, [8], and Haskell, [4] als moderne varianten. Haskell wordt veel gebruikt op universiteiten.

Zuiver functioneel programmeren is nog geen gemeengoed, ondanks de uitdrukkingskracht en betrouwbaarheid. Om gebruik te kunnen maken van deze kracht, moet de programmeur het type systeem begrijpen en de juiste abstracties kunnen maken. Heeft men dit geleerd, dan stellen functionele talen de programmeur in staat om toepassingen te schrijven in een fractie van de gebruikelijke tijd nodig voor het schrijven en corrigeren van de software.

Bibliografie

[1] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams IV, D. P. Friedman, E. Kohlbecker, G. L. Steele Jr., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, M. Wand, W. Clinger, and J. Rees. Revised Report on the Algorithmic Language Scheme. *ACM SIGPLAN Lisp Pointers*, IV(3):1–55, 1991.

[2] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[3] S. Marlow, S. L. Peyton Jones, and S. Singh. Runtime support for multicore Haskell. In G. Hutton and A. P. Tolmach, editors, *Proceeding of the 14th ACM SIGPLAN International Conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, pages 65–78. ACM, 2009.

[4] S. Peyton Jones, editor. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[5] J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin. *LISP 1.5 Programmer's Manual*. MIT Press, 1962.

[6] R. Milner, M. Tofte, R. Harper, and D. McQueen. *The Definition of Standard ML*. The MIT Press, 1997.

[7] Ocaml Development Team. *The Objective Caml system, release 3.12*, 2011. `http://caml.inria.fr/`.

[8] R. M. Plasmeijer and M. C. J. D. van Eekelen. *Clean Language Report*. University of Nijmegen, 2002. Software Technology Group.

[9] D. Syme, A. Granicz, and A. Cisternino. *Expert F#*. Apress, 2007.

[10] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *FPCA*, pages 1–16, 1985.