Functies gaan het maken!

Dr. ir. M.J. Plasmeijer

Katholieke *Universiteit* Nijmegen

Functies gaan het maken!

Rede
in verkorte vorm uitgesproken
bij de aanvaarding van het ambt
van bijzonder hoogleraar in Declaratieve Systemen
in het bijzonder Berekeningsmodellen en hun Implementaties
vanwege de Stichting International Academic Centre for Informatics (IACI)
aan de Katholieke Universiteit Nijmegen
op donderdag 15 juni 1995
door

Dr. ir. M.J. Plasmeijer

Katholieke *Universiteit* Nijmegen

met dank aan mijn ouders

Katholieke *Universiteit* Nijmegen

MIJNHEER DE RECTOR MAGNIFICUS,

ZEER GEWAARDEERDE TOEHOORDERS,

Computers en computerprogramma's

Dagelijks horen we via de radio en de TV dat "het de programma's zijn die het hem doen". Daar zit iets in. Alhoewel de technische mogelijkheden worden bepaald door de *state-of-the-art* in audio en video apparatuur (al of niet stereo, Dolby surround, AC-3, HDTV etc.), het gebruikersgenot hangt toch uiteindelijk af van de mate waarin kwalitatief goede programma's worden aangeboden.

Hetzelfde geldt voor computerapparatuur (de *hardware*) en computerprogrammatuur (de *software*). Ook hier bepaalt de apparatuur wat technisch kan. En technisch kan er steeds méér. Men heeft reeds vaak gedacht dat we aan de grens van het technisch haalbare zaten, maar de computerindustrie is er tot nu toe toch steeds in geslaagd de reken- en geheugen-capaciteit te verdubbelen. Computers worden steeds krachtiger, kosten steeds minder en zijn inmiddels net zo gemeen goed als radio's en TV's.

Maar of je met een computer ook daadwerkelijk iets nuttigs kunt doen hangt af van het aanbod en de kwaliteit van de computerprogramma's. Door de toenemende kracht kan door een computer steeds méér werk worden verricht in minder tijd. Daardoor zijn nieuwe toepassingen mogelijk geworden, zoals multi-media programma's met bewegende beelden en geluid. Men kan daarbij thuis via het internet of CD-Rom een keuze maken uit een gigantisch aanbod. Maar ook de traditionele programma's worden steeds krachtiger. Met een tekstverwerker bewerkt men niet alleen teksten, maar men kan hele pagina's opmaken met de keuze uit willekeurige lettertypes en lettergroottes, men kan daarbij kleur gebruiken, foto's en tekeningen toevoegen, de grammaticale correctheid van de zinnen laten testen terwijl eenvoudige tikfouten tijdens het intikken automatisch worden gecorrigeerd. Sommige snufjes zijn inderdaad erg prachtig en handig, om anderen zit men minder te springen. In ieder geval kan de fabrikant door het toevoegen van snufjes weer een nieuwe versie van zijn programma aan de man brengen. En om met dit alles geriefelijk te kunnen werken bent U uiteindelijk wel verplicht om de nieuwste krachtigste computer aan te schaffen. En zo houden we de economie draaiende.

De *software* crisis

Doordat computers steeds meer kunnen loopt men ook eerder tegen het probleem aan dat de programma's het helaas vaak niet zo doen als wel zou moeten. Alle handelingen die een computer moet verrichten zal men immers moeten programmeren in de *software*. De computerprogramma's worden derhalve steeds groter en daardoor ook steeds ingewikkelder. En menig programma is inmiddels zó groot en zó complex geworden dat niemand meer begrijpt hoe het programma nu precies werkt. Het zoeken van een fout of het aanbrengen van een verandering is dan bijna niet meer te doen.

Het is bijna onmogelijk een fout te vinden in de 63 miljoen (!) programmaregels van de besturings-*software* van een telefooncentrale. Nou ja, het is natuurlijk geen ramp als U een keer verkeerd verbonden wordt. Het is bijzonder lastig om 25.000 programma's, waarvan men niet precies meer weet hoe ze werken en hoe ze van elkaar afhangen, over te dragen op de nieuwe computer waarop die oude truc niet meer werkt. Nou ja, zolang de vergissingen van de bank maar in uw voordeel zijn valt er best mee te leven. Niet een piloot bestuurt de airbus maar een complex stuk *software* met, naar men zegt, een onvindbare fout die gelukkig niet dagelijks voorkomt. Gelukkig zit U niet dagelijks in het vliegtuig.

Dit zijn slechts enkele voorbeelden van de concrete *software*-problemen waar men in het bedrijfsleven momenteel mee worstelt en waarvan het onduidelijk is of ze wel op te lossen zijn zonder helemaal opnieuw te beginnen. Dat het maken van goede *software* niet eenvoudig is merkt U ook aan de programma's die U en ik dagelijks gebruiken. Eigenlijk zit er wel in bijna alle *software* fouten. We zijn dit zelfs doodgewoon gaan vinden. Dit in tegenstelling tot bijvoorbeeld een fout in de *hardware*. De fout laatst gevonden in de Intel Pentium processor haalde wereldwijd de voorpagina's en de fabrikant was genoodzaakt de processor kosteloos te vervangen. Ook bij alle andere gebruiksartikelen vinden we het vanzelfsprekend dat de fabrikant een eventuele fout gratis herstelt. Bij een fout in computer-*software* wachten we echter keurig op de nieuwe versie (met nieuwe fouten) en we zijn bereid daar nog grif voor te betalen ook. Dit is misschien wel een reden voor de fabrikant om zich over niet al te erge fouten niet al te druk te maken.

Nu moet erkend worden dat het ook niet eenvoudig is *software* foutloos op te leveren. Het maken van *software* vergt denkwerk en niet iedereen is daar even goed in. Men heeft soms te maken met moeilijke berekeningen met veel gevalsonderscheidingen en het is niet verwonderlijk dat men dan wel eens een steekje laat vallen. *Software* is arbeidsintensief handwerk en het

ontwikkelen van *software* is daarom een kostbare aangelegenheid terwijl men vanwege de goedkope *hardware* en de gewoonte om illegale kopieën te maken er niet al te veel meer voor kan vragen. En er is altijd een behoorlijke tijdsdruk. En goede informatici zijn schaars. En het mooie van *software* is dat het voor de gebruiker een zwarte doos is. Niemand ziet hoe het werkt. En wie kan het schelen hoe het werkt als het maar ongeveer werkt. Het moet er vooral van buiten mooi uitzien. U begrijpt het al waar de prioriteit bij het ontwikkelen van *software* ligt.

Maar ook wanneer een computerprogramma oorspronkelijk deugdelijk is opgezet zijn problemen op den duur moeilijk te voorkomen.

Het ontwikkelen van een programma vertoont grote overeenkomsten met het bouwen van een huis waarbij men in het algemeen te werk gaat in een aantal stappen. Eerst bedenkt men ruwweg wat voor een soort huis men graag zou willen, zet dit vervolgens om in een aantal wensen op papier, men gaat hiermee naar een architect om een ontwerp te laten maken en stapt vervolgens met dit ontwerp naar een aannemer om het huis te laten bouwen.

Ook een deugdelijk computerprogramma ontstaat op soortgelijke wijze al komt het hierbij regelmatig voor dat één team verantwoordelijk is voor alle stappen. Eerst probeert men zich een goed beeld te scheppen van wat het programma zou moeten kunnen. Dat is best een moeilijke stap omdat er zoveel mogelijkheden zijn. De vage ideeën moeten worden omgezet in een specificatie van wat het programma precies zou moeten doen. Vervolgens wordt er een ontwerp gemaakt dat uiteindelijk gebruikt wordt als blauwdruk voor de realisatie (de *implementatie*) van het programma.

Nu is het zo dat men een huis later nog wel eens gaat verbouwen, maar bij een computerprogramma gebeurt dat bijna continu: men blijft een programma maar uitbreiden en veranderen. Het kan zijn dat men door het gebruik van een programma nieuwe inzichten en wensen krijgt. Het kan ook zijn dat de omgeving waarin het programma wordt gebruikt verandert en de *software* moet dan noodgedwongen worden aangepast aan die veranderde omstandigheden. En natuurlijk moet men uitbreiden om meer te kunnen aanbieden dan de concurent. Gelukkig is *software* bijzonder flexibel en men blijft dus breien aan het oorspronkelijke programma. Nu kan men bij het ontwerp van dat oorspronkelijke programma onmogelijk rekening houden met alle veranderingen en uitbreidingen die men later bedenkt. Eigenlijk zou men vaker een programma opnieuw of gedeeltelijk opnieuw moeten opzetten, maar ja... Langzaam maar zeker verandert zo zelfs een oorspronkelijk goed doordacht programma in een onbeheersbare draak. In het

dagelijkse leven zien we dit soort van draken ook ontstaan, denk bijvoorbeeld aan de belastingwetgeving.

Programmeren in een imperatieve stijl

Van mijn vader heb ik geleerd dat goed gereedschap het halve werk is. Het is ook in ons vak essentieel om goed gereedschap te hebben waarmee men programma's snel kan ontwerpen, implementeren, veranderen en uitbreiden zonder dat het overzicht verloren gaat.

Gelet op de zo juist genoemde problemen is het niet zo gek nog eens kritisch te kijken naar het gereedschap dat men gebruikt. Een belangrijk stuk gereedschap is de programmeertaal waarin we de computer programmeren. Hoewel er talloze programmeertalen bestaan worden de meeste programma's toch geschreven in programmeertalen zoals C, C++, Fortran, Cobol en Pascal. Deze talen zijn directe afstammelingen van de allereerste programmeertalen uit de jaren vijftig. De eerste computers werden geprogrammeerd in machinetaal, een notatie direct afgeleid van de instructies die een processor begrijpt. In machinetaal beschrijft men stap voor stap in een reeks bevelen welke handelingen de computer achtereenvolgens moet verrichten. De machinetaal werd opgevolgd door "hoogniveau" programmeertalen waarmee het mogelijk werd computers te programmeren in een notatie die onafhankelijk was van de gebruikte processor. Hetzelfde programma kon nu gebruikt worden op verschillende type computers mits men maar een vertaalprogramma had (een *vertaler*) om een programma in hoogniveau taal om te zetten naar machinetaal. Langzaam maar zeker begreep men dat het belangrijk was dat programma's goed leesbaar waren voor de *mens* en dat men over de werking en de correctheid van programma's moest kunnen redeneren. Men zag het belang in van abstractie en van een modulaire en gestructureerde opbouw van de programma's. Wilde sprongen in een programma werden afgeraden terwijl procedures, functies, recursie, objecten en modules geïntroduceerd werden als abstractiemethoden. Maar ondanks alle geïntroduceerde verbeteringen is er klaarblijkelijk toch nog steeds iets flink mis. Een belangrijke reden zou wel eens kunnen zijn dat computerprogramma's nog steeds te *imperatief* zijn, dat wil zeggen: bij het programmeren ligt de nadruk nog steeds teveel op het uitschrijven van een reeks bevelen die de computer achter elkaar moet uitvoeren.

```
procedure quicksort (l,r: integer);
var i,j,v,t: integer;
begin
       if r>l then
            begin
                 v := a[r];
```

```
                i := l-1;
                j := r;
                repeat
                      repeat
                        i := i+1
                      until a[i]>=v;
                      repeat
                        j := j-1
                      until a[j]<=v;
                      t := a[i];
                      a[i] := a[j];
                      a[j] := t;
                until j <= i;
                a[j] := a[i];
                a[i] := a[r];
                a[r] := t;
                quicksort (l,i-1);
                quicksort (i+1,r)
            end
end;
```

Voorbeeld 1: Sorteren in een imperatieve stijl.

Bekijken we eens een klassiek imperatief sorteer programma om een rij elementen te sorteren (zie voorbeeld 1), het zogenaamde quicksort algorithme, genomen uit een standaardboek waarin dit soort van veel gebruikte programma's worden beschreven [Sedgewick, 1983]. Wat een informaticus direct opvalt zijn de zogenaamde *toestandsvariabelen* (zoals `i`, `j`, `v` en `t`) die in het programma steeds een andere waarde toegewezen krijgen (`i := i+1`, `j := j-1`). In dit geval worden toestandsvariabelen gebruikt als index waarmee men naar steeds weer andere elementen in de rij kan wijzen (`a[i]`, `a[j]`, `a[r]`). Op een gegeven moment worden een aantal elementen in de rij met elkaar verwisseld (`t := a[i]; a[i] := a[j]; a[j] := t;`) naar we aannemen met het doel uiteindelijk een gesorteerde rij over te houden.

Men vraagt zich direct af naar welk element in de rij nu weer wordt gewezen en of er niet een element te veel of te weinig wordt verwisseld. Eindigt het programma wel en is het resultaat wel een gesorteerde rij? Waar staat die rij eigenlijk en om wat voor soort rij gaat het? Zelfs voor een ervaren informaticus is het niet eenvoudig in te zien wat het programma zou moeten doen laat staan of dat ook wel altijd goed gebeurt. Om een klein beetje te begrijpen wat zo een programma doet zit er niets anders op dan de bevelen in het programma net als een computer na te spelen. Wil men echt de correctheid aantonen dan moet men zich zelfs speciale bewijsmethodes eigen maken. Er zijn maar weinig programmeurs die dat kunnen of die de moeite nemen en dat is ook wel begrijpelijk omdat een dergelijk correctheidsbewijs al snel veel ingewikkelder wordt dan het programma zelf. En dan gaat het hier nog wel om een bijzonder klein en eenvoudig programma dat nog relatief netjes is geprogrammeerd. Men kan zich nu misschien

voorstellen welke problemen zich zullen voordoen bij de echt grote programma's.

Programmeren in een functionele stijl

Bij de imperatieve stijl van programmeren moet de programmeur zich met *teveel* onbelangrijke details bezig houden. Een andere stijl van programmeren is de zogenaamde *declaratieve stijl*. Hierbij geeft de programmeur in principe aan *wat* er moet gebeuren en wordt aan het vertaalprogramma overgelaten te bedenken *hoe* de computer dat dan precies zou moeten doen. En zoals U weet: het is makkelijker gezegd dan gedaan. Voor een dergelijke stijl van programmeren heb je een heel ander soort programmeertaal nodig, bijvoorbeeld een *functionele programmeertaal*.

Een functionele programmeertaal is gebaseerd op het mathematische begrip functie. In een functioneel programma staan functiedefinities centraal. De gebruikte notatie is nauw verwant aan notaties zoals gebruikelijk zijn in de wiskunde.

Het quicksort algorithme ziet er heel anders uit in een functionele stijl (zie voorbeeld 2).

```
quicksort :: [Getal] -> [Getal]
quicksort []          = []
quicksort [e:rest]    = quicksort [x \\ x <- rest | x < e]
                        ++ [e] ++
                        quicksort [x \\ x <- rest | x >= e]
```

Voorbeeld 2: Sorteren in een functionele stijl.

Wat direct opvalt is dat het programma nogal wat korter is dan het vorige. De eerste regel definieert het *type* van de functie: het geeft aan op welke soort van argumenten de functie werkt (het *domein* van de functie, in casu een lijst van getallen) en wat de functie voor soort resultaat oplevert (het *bereik* van de functie, in casu ook weer een lijst van de hopelijk gesorteerde getallen). De tweede regel geeft aan dat het sorteren van een lege rij wederom een lege rij oplevert. De derde regel geeft aan dat voor het sorteren van een niet lege rij, waarvan we het eerste element van de rij `e` noemen en de rest van de rij `rest`, we `e` moeten plaatsen tussen twee nieuwe lijsten. Het element `e` wordt voorafgegaan door de gesorteerde lijst van alle elementen die kleiner zijn dan `e` en gevolgd door de gesorteerde lijst van alle elementen die groter danwel gelijk zijn aan `e`.

Het programma is niet alleen korter maar vooral veel begrijpelijker. Geen informaticus zal er ook maar een moment aan twijfelen dat deze functie `quicksort` inderdaad een gesorteerde lijst getallen oplevert. Voor het formeel

bewijzen van de correctheid heeft men niet meer nodig dan traditionele wiskunde. Het is in dit geval zelfs een fluitje van een cent.

Dat functionele programma's veel inzichtelijker zijn komt doordat het resultaat van een functie uitsluitend bepaald wordt door de waarde van de argumenten waarmee de functie wordt toegepast. Variabelen waaraan men verschillende waardes kan toekennen zijn er niet. Het resultaat van een functie is dan ook nimmer van buiten af te beïnvloeden. Men kan een functietoepassing daardoor onder alle omstandigheden vervangen door het bijbehorende functieresultaat, zoals men `2+3` altijd kan vervangen door `5`. Om te begrijpen wat een functie doet hoeft men alleen maar te kijken naar zijn definitie. In die definitie kunnen weer andere functies gebruikt worden. Maar begrijpt men de werking van die functies dan begrijpt men ook de werking van het geheel. Functies zijn modulaire bouwsteentjes, een soort van *Lego* blokjes waarmee men op inzichtelijke wijze nieuwe ingewikkelder bouwsteentjes (functies) kan maken totdat men uiteindelijk de laatste bouwsteen, het complete programma heeft gerealiseerd.

Een groot voordeel van functionele talen is dat men bijzonder *universele* bouwsteentjes kan maken: men kan namelijk zogenaamde hogere orde functies gebruiken, d.w.z. functies kunnen functies als argument krijgen en functies als resultaat opleveren.

```
quicksort :: (a a -> Bool) [a] -> [a]
quicksort kd []       = []
quicksort kd [e:rest] = quicksort kd [x \\ x <- rest | kd x e]
                        ++ [e] ++
                        quicksort kd [x \\ x <- rest | not (kd x e)]
```

Voorbeeld 3: Een universele sorteerfunctie m.b.v. een hogere orde functie.

Dit stelt ons in staat om bijvoorbeeld een functie `quicksort` te maken die niet alleen een rij getallen kan sorteren maar een rij van *willekeurige* dingen (zie voorbeeld 3), bijvoorbeeld de gegevens uit een databestand. Zo een functie `quicksort` krijgt dan naast de te sorteren rij ook een functie als argument, de functie 'kleiner dan' (`kd`), waarmee de objecten uit die rij geordend kunnen worden. Weet men hoe men elementen ten opzichte van elkaar kan ordenen, dan kan men ze ook sorteren.

Dit kan men nog iets mooier opschrijven (zie voorbeeld 4). Men staat het toe functienamen (zoals `<`, `<=`, `>`, `>=`) als synoniem te gebruiken voor *verschillende* functies die *gelijksoortig* werk doen zij het op *verschillende* objecten (*overloading*). De vertaler moet dan voor een synoniem de juiste functie invullen afhankelijk van de manier waarop het synoniem wordt gebruikt. Dergelijke functies hoeven nu niet meer expliciet te worden mee-

gegeven als argument. Dankzij dit mechanisme kan de oorspronkelijke beschrijving van `quicksort` nu tevens dienen als *universele* beschrijving.

```
quicksort :: [a] -> [a] | Ord a
quicksort []          = []
quicksort [e:rest]    = quicksort [x \\ x <- rest | x < e]
                        ++ [e] ++
                        quicksort [x \\ x <- rest | x >= e]
```

Voorbeeld 4: Een universele sorteerfunctie m.b.v. overloading.

De functie heeft nu wel een algemener type: de functie kan nu immers een lijst van *willekeurige* objecten sorteren mits maar ergens in het programma de bijbehorende functies voor het ordenen (de klasse `Ord` bevat `<`, `<=`, `>`, `>=`) gedefinieerd zijn.

Het is natuurlijk heel handig om zoveel mogelijk van dergelijke universele bouwsteentjes klaar te hebben staan. De programmeur hoeft dan niet steeds opnieuw het wiel uit te vinden. Op die manier bespaart men veel werk, programma's worden korter en bevatten mede daardoor minder fouten. Daarbij zorgt het typeringssysteem ervoor dat de bouwstenen alleen maar op elkaar kunnen worden gestapeld als ze perfect passen zodat uiteindelijk het hele programma staat als een huis.

```
quicksort :: [a] -> [a] | Ord a
quicksort []          = []
quicksort [e:rest]    = quicksort kleiner
                        ++ [e] ++
                        quicksort groter
where
       (kleiner, groter) = splits rest ([],[])

       splits []     (klein,groot) = (klein,groot)
       splits [x:xs] (klein,groot)
            | x < e      = splits xs ([x] ++ klein,groot)
            | otherwise  = splits xs (klein,[x] ++ groot)
```

Voorbeeld 5: Een efficiëntere versie van de universele sorteerfunctie.

Het kan best zijn dat men bij het bouwen van het huis merkt dat bepaalde werkzaamheden veel te lang duren. Het kan om efficiëntie redenen noodzakelijk blijken dat bepaalde functies vervangen moeten worden door andere functies die hetzelfde werk doen, maar dan sneller.

Bijvoorbeeld, het functioneel sorteer algorithme kan men versnellen (zie voorbeeld 5) door een hulpfunctie te maken die de originele lijst in een keer kan splitsen in twee lijsten, een met de elementen die kleiner zijn dan `e` en een met de elementen die groter zijn dan `e`. Gebruikers van de sorteer functie zien geen verschil, behalve verschil in snelheid. In principe kan men zo een efficiëntie bereiken die vergelijkbaar is met de imperatieve methode.

De modulariteit en compactheid van functies maakt een stijl van programmeren mogelijk waar men zich op de eerste plaats richt op het verkrijgen van een begrijpelijk en correct programma. Blijkt het programma niet snel genoeg dan kan men zich later buigen over de inefficiënte functies en die vervangen door efficiënte.

Het gebruik van een functionele taal hoeft men niet te beperken tot de realisatiestap. Men kan dit gereedschap ook gebruiken voor de specificatie van de gewenste eigenschappen, voor het maken van een ontwerp, voor het bouwen van een prototype. Door het toevoegen van meer detail kan men zo geleidelijk overstappen van de ene fase naar de andere fase van het *software*-ontwikkelingsproces. Op deze wijze verhoogt men de algehele consistentie. Ook het later aanbrengen van veranderingen wordt eenvoudiger.

Functies hebben ook nog andere opmerkelijke eigenschappen. Omdat functies op zichzelf staan mag men functieapplicaties in willekeurige volgorde uitrekenen, het eindresultaat zal er niet door kunnen veranderen. Men kan dus vrij de volgorde kiezen waarin men de bouwsteentjes plaatst.

Zo kan men besluiten een bouwsteen alleen te gaan kopen op het moment dat je het nodig hebt bij het bouwen. Bij deze luie handelswijze (*lazy evaluation*) zit je nooit opgescheept met onderdelen die achteraf toch niet nodig waren. Voor de programmeur geeft dit het gemak dat hij zich bij het schrijven van een programma niet druk hoeft te maken over de volgorde waarin de steentjes nodig zijn. Een functie wordt pas uitgerekend op het moment dat het functieresultaat nodig blijkt. In het dagelijks leven kan deze manier van werken ook handig zijn. Heb je bijvoorbeeld een hekel aan de vaat doen dan kun je jezelf werk besparen door alleen maar datgene af te wassen wat je op dat moment wilt gaan gebruiken. Soms is deze werkwijze onhandig. Wanneer je op deze manier bij stukjes en beetjes toch de hele vaat doet had je veel beter toch in één keer de hele handel kunnen afwassen.

Men kan ook besluiten om een aantal steentjes *tegelijkertijd* te plaatsen. Zo kan men bijvoorbeeld de beide deelrijen die aan `e` geplakt moeten worden tegelijkertijd sorteren in de hoop dat het dan sneller gaat. Men heeft dan wel een computer nodig met meer dan één processor. Het parallelliseren van een functioneel programma is vergeleken met het parallelliseren van een imperatief programma kinderspel maar het is in praktijk toch nog vrij lastig om een programma zo op te delen dat ook daadwerkelijk snelheidswinst wordt geboekt. Ook dit is een ervaring uit het dagelijks leven. De *overhead* die gepaard gaat met het uitbesteden van werk is vaak zo groot dat je het werk beter zelf had kunnen doen.

Functies in theorie en praktijk

Een functionele programmeerstijl levert dus relatief korte, leesbare en betrouwbare programma's op. Omdat men zich beter kan concentreren op de echte problematiek wordt het voor de programmeur ook leuker om te doen. Iedere universitaire informatica opleiding leert haar studenten dan ook tegenwoordig programmeren in een functionele taal. Toch worden functionele talen door het bedrijfsleven nauwelijks gebruikt. Daar zijn diverse redenen voor aan te wijzen.

Pas sinds kort weet men hoe men een programma geschreven in een functionele taal kan vertalen in een redelijk efficiënt werkend programma in machinetaal. Een beetje merkwaardig is dat wel omdat het concept functie voor informaticabegrippen al "eeuwenoud" is.

In het begin van deze eeuw, dus lang voordat er computers bestonden, hebben de wiskundigen zich al afgevraagd wat nu precies een functie was en wat je ermee kon berekenen. In de jaren dertig werd het begrip functie door Alfonzo Church geformaliseerd in de λ-calculus [Church 1932, 1933]. Toevallig werd ongeveer tegelijkertijd door Alan Turing een ander wiskundig model bedacht, de Turing "machine" [Turing 1937]. In 1936 toonde Church aan dat beide modellen conceptueel dezelfde berekeningskracht hadden. Maar er zijn ook belangrijke verschillen. Het model van Turing is een vrij simpel model en een computer gebaseerd op dit principe was dan ook vrij eenvoudig te maken. En nog steeds zijn de computers die wij dagelijks gebruiken en de imperatieve talen waarmee we deze computers programmeren direct gebaseerd op het model van Turing. De λ-calculus, het model waar functionele talen op zijn gebaseerd, is daarentegen een relatief ingewikkeld model. Het heeft daarom vrij lang geduurd voordat men begreep hoe men berekeningen gebaseerd op dit model efficiënt kon laten uitvoeren door een computer. Vergeet daarbij niet dat de computers die in de winkel liggen nu eenmaal werken volgens het model van Turing en niet volgens die van Church.

In het begin van de jaren zestig bedacht John McCarthy de eerste functionele taal, Lisp [McCarthy, 1960]. Lisp werd behoorlijk populair niet in het minst vanwege zijn aantrekkelijke programmeeromgeving. Maar Lisp is geen *pure* functionele taal. Ook McCarthy zag niet in hoe men zonder de variabelen die van waarde kunnen veranderen de noodzakelijke efficientie kon bereiken.

Pas eind jaren zeventig komen functionele talen opnieuw in de schijnwerpers. John Backus, de ontwerper van Fortran, de allereerste hoogniveau programmeertaal uit de jaren vijftig die nog steeds alom wordt gebruikt,

kreeg voor zijn pionierswerk de Turing Award (genoemd naar Alan Turing). Het is bijzonder imponerend dat juist een man als John Backus een warm pleidooi hield voor het gebruik van functionele talen (FP) [Backus, 1978]. Ook het werk van David Turner mag niet ongenoemd blijven. De notatie gebruikt in de λ-calculus is voor programmeurs lastig te lezen. Turner ontwierp daarom een aantal prachtige functionele talen (Sasl, Miranda) [Turner 1979, 1985]. Tevens liet hij een elegante zij het even inefficiënte implementatietechniek zien gebaseerd op de combinatorische logica. Het werk van John Backus en David Turner gaf een wereldwijde stimulans nog eens goed te kijken naar functionele talen en verder te zoeken naar betere implementatietechnieken.

Die implementatietechnieken gingen er de daarop volgende jaren inderdaad flink op vooruit met name dankzij werk op de Universiteiten van Göteborg (Augustsson en Johnsson), Glasgow (Peyton-Jones), Yale (Hudak) en Nijmegen. In Nijmegen is werk verricht op zowel theoretisch als praktisch gebied. Samen met de Universiteit van East-Anglia (Glauert, Kennaway en Sleep) werd als een alternatief berekeningsmodel voor de λ-calculus graafherschrijfsystemen ontwikkeld, waarmee bepaalde implementatieaspekten nu ook op theoretisch niveau goed konden worden onderzocht. Dankzij al dit onderzoek is het verschil in efficiëntie met imperatieve talen steeds kleiner geworden. En sinds kort kan men dan ook eindelijk efficiënte grote programma's schrijven in een functionele taal.

Maar er was nog een ander probleem dat het commercieel gebruik van functionele talen in de weg stond. Een programma geschreven in een functionele taal moet harmonieus kunnen samenwerken met de andere programma's en systemen die op de computer draaien. En de bestaande systemen werken nu eenmaal volgens het principe van Turing. Ook in een functioneel programma wil je de mogelijkheid hebben om bijvoorbeeld de gegevens in een databestand te veranderen. Hoe beschrijf je dat in een puur functionele taal waarin men immers geen toestandsvariabelen kent die van waarde mogen veranderen? Inmiddels is ook een goede oplossing gevonden voor dit probleem. Wanneer men op een of andere manier kan garanderen dat een functie privé toegang heeft tot zijn argument (b.v. het databestand) dan is het toegestaan dit argument te gebruiken voor het maken van het functieresultaat (b.v. het veranderde databestand). Het oude bestand is toch niet meer nodig omdat daar geen enkele andere functie meer naar kan verwijzen. Op die manier kan men dus toch in een puur functionele wereld een destructieve toewijzing realiseren. Men kan de privé toegang regelen door ervoor te zorgen dat het te veranderen object verborgen blijft voor de programmeur (de monadische benadering). In Nijmegen is dankzij een goede

samenwerking tussen theoretisch en praktisch georiënteerde onderzoekers een andere manier bedacht. Men kan in een speciaal typeringssysteem (uniciteitstypering) aangeven dat een functie privé toegang eist tot een bepaald object. Het typeringssysteem staat nu alleen maar programma's toe die deze eis respecteren.

Daarmee kan men een functioneel programma naadloos inpassen in de niet functioneel werkende computerwereld. Interactieve toepassingen zoals tekstverwerkers, databaseprogramma's en spreadheets kunnen nu met een pure functie beschreven worden met alle voordelen van dien. Zo kan men een interactief functioneel programma op eenvoudige wijze splitsen en verspreiden over een netwerk van computers. De onderdelen kunnen boodschappen met elkaar uitwisselen waarbij de vertaler garandeert dat de juiste boodschap aankomt bij de juiste ontvanger.

Al met al zijn er thans geen onoverkomelijke technische problemen meer die het commercieel gebruik van functionele talen in de weg staan. Met functies kan men nu maken wat men wil. De vraag blijft: gaan functies het nu ook *maken*? Pakt de industrie de draad op? Dat is nog maar de vraag. Een goed idee hoeft lang niet altijd aan te slaan. Is het niet al te laat? De hele wereld is aan de traditionele manier van programmeren gewend en zoiets laat zich zomaar niet veranderen. Denk bijvoorbeeld aan het QWERTY toetsenbord. De indeling is indertijd zo gemaakt om ervoor te zorgen dat mensen op een typemachine niet te snel tikken om te voorkomen dat de letterhamertjes vast komen te zitten. Dat slaat nu natuurlijk nergens meer op, maar de indeling is niet meer te veranderen omdat iedereen het nu eenmaal zo is gewend. Een ander probleem is dat voor industrieel gebruik van puur functionele talen betrouwbare vertaalprogramma's nodig zijn. De vertaalprogramma's zijn echter zo specialistisch dat men ze op dit moment alleen nog maar tegenkomt op universiteiten. Een bedrijf maakt zich niet graag afhankelijk van een dergelijke leverancier. Commerciële *software* fabrikanten zullen vertalers pas als serieus produkt aanbieden wanneer er voldoende vraag naar is. Vraag uit het bedrijfsleven is er niet of nauwelijks omdat men doorgaans geen weet heeft dat er functionele talen bestaan en wat men ermee zou kunnen doen. En zou men het weten, tja, er worden zoveel wondermiddeltjes beloofd en er zijn natuurlijk ook andere wegen die naar Rome leiden. Het is moeilijk voor bedrijven het kaf van het koren te scheiden. En men wil niet het risico lopen te gokken op het verkeerde paard want dat kan men zich vaak domweg niet veroorloven.

Toch zijn er een aantal lichtpuntjes. Dat men niet meer op de huidige voet door kan gaan is een aantal bedrijven inmiddels op de harde manier duidelijk geworden. Er ontstaat langzaam maar zeker een klimaat waarin een

nieuwe aanpak die bouwt op een gedegen wiskundige ondergrond wordt geapprecieerd. Een aantal grote bedrijven (zoals het Zweedse Ericsson en de Duitse Software AG) hebben inmiddels hun eigen functionele taal ontwikkeld. De talen zijn relatief eenvoudig maar toch is men al zeer tevreden over de behaalde resultaten. Ericsson, de trotse eigenaar van de al eerder genoemde 63 miljoen regels code denkt door het herschrijven in een functionele taal het aantal regels tot 6 miljoen terug te brengen (overigens nog steeds een belachelijk groot aantal).

Universiteit en bedrijfsleven

Geachte toehoorders, de primaire doelstelling van de universiteit is het verrichten van academisch onderzoek en het geven van academisch onderwijs. Daar is geld voor nodig en studenten en aan beide hebben we een groot gebrek.

Het valt niet mee om meer studenten te bewegen exacte vakken te gaan studeren. Exact is moeilijk en klaarblijkelijk even niet populair in Nederland. De universiteiten stellen zich steeds meer als concurenten van elkaar op om het schaarse goed, studenten, binnen te halen. Ik vind dat wel eens te ver gaan. Een goede inhoudelijke voorlichting maakt plaats voor werving met slogans die doen denken aan de wasmiddelenreclame. Door het tekort dat zal ontstaan aan academici met een exacte opleiding zullen er uiteindelijk wel weer meer studenten komen. Tot die tijd moeten we zien te overleven.

De maatschappij is van mening dat de universiteit zich meer moet richten op de noden van het bedrijfsleven. Daar is niets mis mee als er ook maar voor gezorgd wordt dat er voldoende ruimte blijft voor fundamenteel onderzoek ook als niet direct duidelijk is of het ooit zou kunnen worden toegepast.

De afgelopen tien jaar heb ik met mijn groep onderzoek verricht op het gebied van functionele talen. Ook daarvan was niet bij voorbaat duidelijk of het wel iets praktisch bruikbaars zou opleveren. Dat wilden we juist onderzoeken. Om dit onderzoek goed te kunnen doen heb je een team nodig van onderzoekers dat in staat is een probleem zowel van de theoretische als van de praktische kant te benaderen. Het onderzoek heeft een zware theoretisch component welke nauw verwant is met de grondslagen van de informatica. Om te laten zien dat de ideeën om te zetten zijn in een praktisch bruikbaar stuk gereedschap moet echter ook zeer veel werk gestopt worden in de realisatie. Een bruikbaar stuk gereedschap moet namelijk werken, ook voor die gevallen die academisch gezien triviaal zijn. Er is weinig waardering voor implementatiewerk op een universiteit: slechts het

aantal publicaties telt en niet wat je er voor hebt moeten doen. Toch vind ik het belangrijk dat je laat zien dat een idee ook echt werkt in de praktijk. Met een nieuw stuk gereedschap krijg je bovendien een uniek stuk infrastructuur dat je in staat stelt geheel nieuwe onderzoekswegen te bewandelen. Verder hebben we dikwijls ervaren dat de combinatie van theoretisch en praktisch onderzoek zeer vruchtbaar is. Wanneer je alleen maar theoretisch bezig bent heb je namelijk te snel de neiging vereenvoudigingen aan te brengen teneinde je bewijs rond te krijgen. Waneer je alleen praktisch bezig bent heb je de neiging te snel iets in elkaar te flansen dat lijkt te werken. Wil je iets praktisch bruikbaars bouwen op een degelijk fundament dan ben je verplicht in de theorie ook de lastig te bewijzen maar praktisch essentiële gevallen mee te nemen. Ook is gebleken dat een experimenteel bouwwerk het theoretisch inzicht vergroot en een nieuwe richting kan geven aan de theorie.

Het moge duidelijk zijn dat voor dergelijk onderzoek veel mankracht nodig is, zeker wanneer je op internationaal niveau wilt meetellen. Het leeuwedeel van het onderzoek moet men dan ook financieren via externe geldbronnen. Voor universitair onderzoek is echter ook extern steeds minder geld beschikbaar terwijl dat geld ook nog eens verdeeld wordt over steeds meer potjes. Onderzoekleiders moeten daarom steeds meer tijd besteden aan het werven van geld terwijl door het groot aantal gegadigden de kans op succes steeds kleiner wordt. Sommigen collega's specialiseren zich daarom liever in slim politiek handelen dan in het verrichten van goed onderzoek. Is het geld eenmaal binnen moet men er overigens ook nog eens voor waken dat niet teveel van het binnenstromende geld wordt afgeroomd want tekorten zijn er overal.

Het is dus niet zo verwonderlijk dat de universiteiten de banden met het bedrijfsleven willen aanhalen. Ik vind dat die liefde niet alleen uit financiële overwegingen moet opbloeien. Bedrijfsleven en universiteit hebben elkaar nodig en kunnen elkaar versterken. Intensievere samenwerking leert ons beter begrijpen welke problemen binnen het bedrijfsleven spelen en met die wetenschap kunnen we ons onderzoek en onderwijs verbeteren. Het bedrijfsleven kan natuurlijk baat hebben van de expertise die op de universiteit aanwezig is.

De samenwerking die ik probeer tot stand te brengen is speciaal in die zin dat het in dit geval gaat om een *technology push* vanuit de universiteit naar het bedrijfsleven. Functionele talen en de onderliggende λ-calculus hebben al hun plaats verworven in het academisch informaticaonderwijs en zelfs ook al in het hogerberoepsonderwijs (de Hogere Informatica Opleidingen). Wij willen nu aan het bedrijfsleven laten zien dat men door ons nieuw gereedschap te gebruiken aanzienlijk kan besparen terwijl de produkten veel

betrouwbaarder worden. Een dergelijke *technology push* is echter niet mogelijk zonder dat het bedrijfsleven daar zelf ook in investeert. De enorme kloof in kennis tussen bedrijfsleven en universiteit moet worden overbrugd. Ons gereedschap moet worden omgezet van academisch speeltje naar professioneel gereedschap dat kan worden ingezet in de concrete bedrijfsomgeving.

Ik heb het afgelopen jaar ervaren dat het bedrijfsleven wel degelijk open staat voor nieuwe ideeën en bedrijven gevonden die met functionele talen aan de slag willen gaan. Het zou fijn zijn wanneer we samen de belofte hard kunnen maken. Het zou nog mooier zijn wanneer we hiermee de achterstand van de Nederlandse *software* industrie zouden kunnen omzetten in een voorsprong en de kloof tussen universiteit en bedrijfsleven zouden kunnen verkleinen.

Dankwoord

MIJNHEER DE RECTOR,

ZEER GEACHTE DAMES EN HEREN,

Aan het einde van mijn rede gekomen maak ik graag van de gelegenheid gebruik een aantal personen en instellingen te bedanken zonder wie ik hier niet zou staan.

Zeer geacht Bestuur van de stichting International Academic Centre for Informatics

U heeft zich de afgelopen jaren ingezet voor de Nijmeegse Informatica. Mede dankzij de door U georganiseerde *masters* opleiding heeft U ervoor gezorgd dat de banden tussen bedrijfsleven en universiteit verder konden worden aangetrokken. Door diverse omstandigheden heeft U helaas deze activiteiten moeten staken. Uw doelstellingen moeten nu verder via deze leerstoel gestalte krijgen. Ik dank U voor het vertrouwen dat U mij heeft gegeven. Ik zal mijn uiterste best doen en daarbij Uw grote inzet en geduld als voorbeeld nemen.

In het bijzonder wil ik de heer Van de Heuvel bedanken, directeur van Kropman B.V. (Installatietechniek, Nijmegen) en sponsor van deze leerstoel. Uiteraard wil ik de hoop uitspreken dat U ons ook in de toekomst wilt blijven steunen. U bent van mening dat een bedrijf indien mogelijk ook een zekere bijdrage moet leveren aan de infrastructuur van de omgeving. Een dergelijke opvatting en handelswijze is helaas niet alledaags meer in onze maatschappij. Ik dank U voor dit goede voorbeeld en hoop dat anderen het zullen volgen.

Zeer geachte leden van het College van Bestuur van deze Universiteit

Het academisch bedrijf heeft soms wel iets weg van een merkwaardig circus waarbij een wel zeer bonte verzameling van nogal eigengereide maar zeer knappe artiesten werken. Geen circus zonder clowns, maar men moet er niet teveel van hebben rondlopen. Ik dank het College, en met name de heer Stoelinga voor de professionele wijze waarop men haar taak als directie van een dergelijk moeilijk bedrijf gestalte geeft.

Zeergeleerde Aarts, Beste Christ

Onder jouw directeurschap is hier de Faculteit Natuurwetenschappen letterlijk en figuurlijk van de grond gekomen. Dankzij jouw visie en inzet

kwam er in Nijmegen ook een studierichting informatica. Ik wil je hiervoor bedanken. Men beseft vaak te weinig hoeveel energie en jaren erin gaan zitten om iets op te bouwen, terwijl met één bezuinigingsmaatregel de zaak in *no time* is afgebroken. Ik ben er best een beetje trots op dat deze leerstoel één van de vele dingen is die mede dankzij jouw visie tot stand is gekomen. Heel erg bedankt.

Hooggeleerde Barendregt, Beste Henk

Wij kennen elkaar sinds 1984 toen jij me vroeg of ik met je wilde samenwerken in het Parallelle Reductie Machine project. Vanaf dat moment ontstond een vruchtbare samenwerking tussen ons en onze twee onderzoeksgroepen. We delen de liefde voor functionele talen en de grondslagen van de informatica, we delen de secretaresse, de gang, de koffie, geldpotjes, de zorg voor de vakgroep en dit alles zonder onderling ooit één conflict te hebben gehad. Zonder jouw steun, die meer energie heeft gekost dan men van iemand kan verlangen, zou ik hier niet staan. Mijn dank voor alles.

Geachte leden van de onderzoekslijn Functionele Programmeertalen

Ik ben jullie veel dank verschuldigd. Het is een bijzonder voorrecht leiding te mogen geven aan een zo getalenteerde en gemotiveerde groep medewerkers. Een projectleider kan helaas slechts weinig tijd besteden aan het eigenlijke onderzoek. Jullie moeten vaak het echte werk doen. Er zijn de afgelopen jaren door jullie prachtige resultaten geboekt. Dit is deels te danken aan de combinatie van theoretisch en praktisch georiënteerd onderzoek. Maar vooral ook doordat ieder van jullie altijd bereid was om een flink stuk extra werk te verrichten ten behoeve van het onderzoek van een ander teamlid. Er zijn inmiddels vele nieuwe vragen opgekomen die we nog moeten beantwoorden. Er is dus nog een zeer lange maar uitdagende weg te gaan. Ik hoop dat we in de gelegenheid blijven om het onderzoek op dezelfde prettige wijze voort te zetten.

Zeergeleerde Van Eekelen, Beste Marko

Wij zitten al meer dan tien jaar bij elkaar op de kamer en zijn inmiddels volkomen op elkaar ingespeeld. We worden beschouwd als een soort siamese tweeling, nou ja, tegenwoordig misschien meer als "dik en dun". De één kan te allen tijde invallen voor de ander, of het nu gaat om onderzoek, onderwijs of management. Stel dat nu de telefoon zou gaan en ik zou weg moeten voor iets belangrijks, dan weet ik zeker dat jij zonder blikken of blozen verder zou gaan waar ik gebleven was, zelfs al moest je het doen zonder papieren. Dat ik hier sta is dus in zekere zin toeval, jij had hier in alle opzichten net zo

goed kunnen staan. Ik ben je veel dank verschuldigd en hoop dat we nog vele jaren zullen samenwerken.

Lieve moeder

Uiteraard zou ik zonder jou hier niet staan. Ik weet dat je nu erg trots bent op je zoon. De eer komt echter jouw toe omdat jij me altijd op het hart hebt gedrukt te studeren. Bedankt hiervoor en bedankt voor mijn fijne jeugd.

Lieve Trees, Lieve Ruvani

Werken en hobby gaan bij mij in elkaar over en daardoor gaat er te vaak teveel tijd in zitten. Zoiets is niet vol te houden zonder de liefde en steun (en de koffie en knollen) van jullie tweetjes. Jullie laten me dagelijks zien wat echt belangrijk is in het leven. Bedankt.

Beste studenten

Het laatste woord is voor jullie en niet zonder reden: zonder studenten zijn wij nergens. Ik wil jullie vragen om bij de studie de theoretische vakken niet te schuwen. Die zijn misschien niet het makkelijkst maar onontbeerlijk voor een goed inzicht in ons vak. Je zult er later veel plezier van hebben en ontdekken dat het vak ook mooi kan zijn zonder *hacken*.

Het wordt nu de hoogste tijd voor een borrel en een hapje.

Ik dank U allen hartelijk voor Uw aanwezigheid en Uw aandacht.

Ik heb gezegd.

Referenties

Backus J. (1978). Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs. *Comm ACM,* **21**, pp. 613–641.

Church A. (1932). A set of postulates for the foundation of logic. *Annals of Mathematics,* **2**, (33), pp. 346–366.

Church A. (1933). A set of postulates for the foundation of logic. *Annals of Mathematics,* **2**, (34), pp. 839–864.

Church A. (1936). An unsolvable problem of elementary number theory. *American J Mathematics,* **58**, pp. 354–363.

McCarthy J. (1960). Recursive functions of symbolic expressions and their computation by machine. *Comm ACM,* **4**, pp. 184–195.

Sedgewick R. (1983). *Algorithms*. Addison-Wesley Publishing Company, Reading, Massachusetts.

Turing A.M. (1937). Computability and λ-definability. *J of Symbolic Logic,* **2**, pp. 153-163.

Turner D.A. (1979a). A new implementation technique for applicative languages. *Software Practice & Experience,* **9**, (1), pp. 31-49.

Turner D.A. (1985). Miranda: a non-strict functional language with polymorphic types. In *Proc. Conference on Functional Programming Languages and Computer Architecture,* Nancy, France (Jouannaud J.P., ed.), *LNCS*, **201**, pp. 1-16. Berlin: Springer-Verlag.