

AUTEUR:

Pieter Edelman

TEKST:

Tussen de verhitte discussies over de beste manier om multicore systemen te programmeren, valt van tijd tot tijd de term 'functioneel programmeren'. Bij veel programmeurs roept dit gelijk negatieve reacties op, een ander deel - grotendeels academici - zal hier enthousiast op reageren. De meeste mensen zullen echter geen idee hebben waar het over gaat.

De methode is volgens sommigen de silver bullet voor het probleem dat de prestaties niet lineair schalen met het aantal kernen in een multicore systeem.

Wat functioneel programmeren precies inhoudt, is niet makkelijk uit te leggen. Standaard programmeertalen zoals C, C++ en Java heten imperatief te zijn. Ze gaan uit van data in een bepaalde toestand, die door statements veranderd kunnen worden. De programmeur kan statements samenbundelen tot een procedure, functie, methode of welke naam het ook mag hebben. Imperatief programmeren is gebaseerd op de werking van de hardware: er is een toestand die bestaat uit geheugen en registers, en instructies voor de processor manipuleren deze toestand. De volgorde van commando's is essentieel voor de werking.

Functioneel programmeren daarentegen laat het model van de hardware volledig los. Het uitvoeren van een programma is gebaseerd op het evalueren van wiskundige functies, er is geen toestand en veranderlijke data en er zijn geen neveneffecten, wat betekent dat de functies elke keer hetzelfde resultaat opleveren als ze worden uitgevoerd. Alleen de parameters zijn van invloed op de uitkomst, niet de toestand van het programma.

Er zijn vele functionele talen zoals Haskell, Clean, F#, Scheme, Erlang en Caml. De notatie van al deze talen is sterk gebaseerd op de wiskundige notatie van functies. Het programma bestaat uit het evalueren van een enkele hoofdfunctie. De waarden die daarin gebruikt worden, kunnen bepaald worden door andere functies en die kunnen op hun beurt weer andere functies aanroepen, en zo voorts. Een functie in een FP-benadering is iets subtiel anders dan in een imperatieve taal. In beide is een functie een brokje code dat parameters in kan nemen en een resultaat uit kan spugen. In C, Java, et cetera is een functie een verzameling commando's, in functionele talen is het een expressie. Nogmaals, imperatief specificereert hoe iets moet gebeuren, functioneel op wát.

'In zekere zin is functioneel programmeren een toepassing van 'in beperking toont zich de meester'', zegt Marko van Eekelen, hoogleraar Softwaretechnologie aan de Open Universiteit en wetenschappelijk directeur van de Nijmeegse LaQuSo-vestiging bij de Radboud Universiteit Nijmegen. Van Eekelen was, naast Rinus Plasmeijer, een van de drijvende krachten achter het ontstaan van de functionele programmeertaal Clean. 'De beperking is heel radicaal in dat je geen zijeffecten toestaat. Iedereen kent wel functies van de middelbare school, zoals  $f(x)=x^2+2$ , daarin wordt  $x$  onderweg niet even gewijzigd. Elke keer dat je  $f(x)$  uitvoert, blijft het resultaat hetzelfde. Het resultaat wordt alleen bepaald door de argumenten en niet door de context.' In een imperatieve taal zou je een  $f$  kunnen definiëren die de waarde van een globale variabele  $x$  wijzigt (een neveneffect) zodat er elke keer dat je  $f$  aanroept een ander resultaat uitkomt.

Functioneel

$f\ x = x^2+2$

```
[ f 3? 11! ]  
[ f 3? 11! ]
```

Imperatief

$f = x := x^2+2$

```
[ x:= 3 ]  
[ f;   ]
```

```
[ x? 11! ]
[ f;      ]
[ x? 123!]
```

Dat is wel even slikken voor iemand die alleen de imperatieve stijl gewend is. 'Die beperking voel je hard in het begin. Even wennen, maar als je eraan gewend bent is het gewoon een andere stijl van programmeren', zegt Van Eekelen.

Een beetje vreemd is dat wel, want veel dingen zoals bijvoorbeeld I/O zijn juist gebaseerd op neveneffecten. 'Maar er zijn wel technieken om bijvoorbeeld bestanden te lezen. Je moet er dan wel voor zorgen dat mensen niet op de een of andere manier dat bestand kunnen wijzigen terwijl jij er mee bezig bent, dat moet je in je taal inbouwen', legt Van Eekelen uit.

Klinkt nog steeds vaag? FP-kenners trekken vaak de vergelijking met spreadsheets. De cellen in een spreadsheet kunnen allemaal verschillende formules die naar de resultaten van berekeningen uit andere cellen verwijzen. De gebruiker bemoeit zich niet met de volgorde waarin die geëvalueerd worden, laat staan met het geheugenbeheer; hij gaat ervan uit dat het programma dat voor hem uitvoert. En alleen de waarden bepalen de uitkomst.

Daarmee houdt de overeenkomst ook wel weer op. Gegevens in een spreadsheet worden doorgaans niet als draaiende applicatie gebruikt. Maar die twee dingen, onafhankelijkheid van een toestand en onafhankelijkheid van de volgorde, zijn twee essentiële ingrediënten van functioneel programmeren en de belangrijkste reden dat de aanpak geschikt is voor multicore, waarin de onderdelen van een programma parallel uitgevoerd moeten worden om de meerdere kernen te kunnen gebruiken. 'Een van de problemen met parallelisme is dat de verschillende onderdelen aan dezelfde variabelen zitten. Dat moet je blokken met semaforen eromheen. Om die synchronisatie goed te krijgen moet je wachten en dat is menselijk moeilijk te overzien', legt Van Eekelen uit. In de FP-aanpak zijn er geen veranderlijke variabelen, dus hoeft er ook niks geblokt te worden.

Bovendien hoeft de programmeur zich niet te bemoeien met de manier waarop het programma wordt uitgevoerd. Dat is aan de compiler of de runtime-omgeving. 'De compiler heeft met een functionele taal een makkelijkere taak om parallelle code te genereren', zegt Van Eekelen daar over. 'Er zijn geen zij-effecten dus het is heel duidelijk waar de afhankelijkheden zitten. De plaatsen waar je dingen parallel kan doen, zijn dus makkelijker te analyseren.' Niet voor niets schaalde de uitvoeringssnelheid van veel functionele programma's dan ook goed met het aantal rekenkernen. 'In de beste gevallen hebben we de prestaties met een factor 8,8 weten te verbeteren ten opzichte van een sequentieel programma op een acht-core-machine', vertelt Kevin Hammond, hoogleraar computerwetenschappen van de Schotse St. Andrews-universiteit. Hij werkt aan de Hume-taal, die specifiek gericht is op embedded systemen met beperkte hardware.

[...]

Laten we er een voorbeeldje bij nemen, het uitrekenen van de faculteit van een getal. Een C-aanpak zou er als volgt kunnen uitzien:

```
[int faculteit(int n) {
    int x = 1;

    for (int i = n; i >= 0) {
        x = x * i; i = i - 1
    }

    return x;
}]
```

Terwijl een equivalente FP-aanpak iets zou schrijven in de trant van:

```
faculteit n
| n <= 0    = 1
| otherwise = n * faculteit (n-1)
```

Twee wiskundige functies die die meer zeggen over de aard van de berekening dan

over de uitvoering ervan. Het is aan de software om een functieaanroep tot een goed einde te brengen. Vaak betekent dit het recursief uitvoeren van de functie. Een belangrijke troef is dan ook dat functies net zo gemakkelijk als waarden gebruikt kunnen worden als getallen of leestekens, waardoor ze ook als argumenten voor functieaanroepen te gebruiken zijn. In imperatieve talen kan dit ook, bijvoorbeeld pointers. Het zijn echter doorgaans geen first class citizens zoals in functionele talen.

Dit voorbeeld van de faculteit berekenen is niet willekeurig gekozen, het is zo'n beetje de hello world waarmee de FP-wereld het verschil met 'traditioneel' imperatief programmeren illustreert. Een toepassing die simpelweg 'hello world' op het scherm weergeeft wordt meestal niet getoond, want dat is vaak minder elegant.

In de taal Haskell:

```
[main = putStrLn "Hello World"]
```

In Erlang:

```
[-module(hello).  
-export([hello/0]).  
hello() ->  
    io:format("Hello World!~n", []).]
```

Nog steeds vaag? Dat is begrijpelijk. Functioneel programmeren is vooral een manier van denken die misschien het beste te leren is door te doen. Je laat het idee van sequentiële opdrachten los en vervangt het door het evalueren van functies. Niet voor niets wemelt het op het web van 'Haskell-tutorials voor C-programmeurs' en gelijkaardige hulpjes die de pijn voor imperatief programmeurs moeten verzachten.

Volgens FP-voorstanders is de aanpak veel eleganter dan een traditionele imperatieve methode. Uiteraard is dat grotendeels een kwestie van smaak, en daarover valt slecht te twisten. Functioneel programmeren is niet nieuw. De geschiedenis gaat terug tot de jaren vijftig van de vorige eeuw, met de grootste ontwikkelingen in de jaren zeventig. Haskell, waarschijnlijk de meest populaire algemeen programmeerbare functionele taal, stamt uit de jaren tachtig. Ondanks deze lange voorgeschiedenis is de kans om een toepassing geschreven volgens FP-methoden in het wild aan te treffen klein. Niet dat er geen bekende toepassingen zijn. De bekende Emacs-IDE leunt bijvoorbeeld voor een groot deel van zijn basisfunctionaliteit en alle uitbreidingen op de functionele Lisp-taal, net als Autocad. Twitter is grotendeels geschreven in de relatief jonge Scala-taal, die sterk beïnvloed is door FP en in de Java-runtimeomgeving draait. Wie op internet zoekt, komt verschillende websites tegen die de praktische toepassingen opsommen. Daar staan toepassingen met grote impact bij aanzienlijke bedrijven tussen. Maar het feit dat ze op een website verzameld kunnen worden, zegt veel over het volume. Tot nog toe zijn de voordelen van FP blijkbaar nog niet groot genoeg geweest om de imperatieve aanpak aan de kant te schuiven.

In de embedded-wereld geldt dit nog sterker. Echt goede voorbeelden zijn nauwelijks te vinden, op één uitzondering na: Ericsson ontwikkelde in de jaren negentig zijn eigen functionele taal om zijn gedistribueerde telecomsystemen te kunnen programmeren: Erlang. Dit bleek een schot in de roos. Vandaag de dag heeft Ericsson tientallen projecten gerealiseerd in Erlang.

Maar verder? 'Er zijn commerciële technieken gebaseerd op 'synchrone dataflow' zoals Lustre, die geëvolueerd zijn tot de Scade-ontwikkeltool van Esterel Technologies. Deze talen zijn sinds de jaren tachtig ontwikkeld en worden toch wel veel gebruikt in bepaalde gebieden zoals veiligheidscritieke realtime systemen. Deze notaties zijn echter behoorlijk beperkt en relatief laag niveau vergeleken met algemene talen zoals Haskell of Ocaml', zegt Kevin Hammond, hoogleraar aan de University of St. Andrews in Groot-Brittannië.

Dat doet vermoeden dat functioneel programmeren geen goede aanpak is voor embedded systemen. Er wordt bijvoorbeeld wel beweerd dat functionele talen minder efficiënt met de hardwarebronnen omgaan, in het embedded wereldje natuurlijk een doodzonde. Dat is niet terecht, meent Van Eekelen: 'performance is geen belangrijk punt, er zijn tegenwoordig hele goede compilers. Je hebt shoot-outs op internet en daar doen FP-talen het goed. Ze gebruiken meestal wel wat meer geheugen.' Ook Hammond denkt er zo over: 'Optimaliserende compilers

kunnen tegenwoordig goede prestaties leveren, zeker beter dan Java voor zowel tijd als ruimte en in sommige gevallen dicht tegen C aan'.

Toch zijn er wel kanttekeningen te plaatsen, met name op het gebied van voorspelbaarheid. Veel functionele talen gebruiken bijvoorbeeld garbage collection, waardoor van te voren geen duidelijk beeld ontstaat van het geheugengebruik. Ook prestaties kunnen moeilijk te voorspellen zijn met de functionele aanpak. 'Omdat functionele talen op zo'n hoog niveau werken, kan de compilatieroute nogal complex zijn en kunnen de prestaties van de geproduceerde machinecode nogal verschillen van de originele broncode', zegt Hammond. Zeker voor realtime systemen diskwalificeert dit een functionele aanpak.

Maar die problemen spelen niet bij alle talen. 'In Hume zijn we met een schone lei begonnen en zorgen we ervoor dat bij elk construct zowel eenvoudig de kosten te bepalen zijn als dat de vertaling naar machinecode duidelijk te zien is in de broncode en geen onverwachte verrassingen toevoegt.'

Een belangrijke overweging is ook of de taal strict of 'lui' is. Luie talen stellen een evaluatie pas tot het allerlaatste moment uit. Dat maakt programma's zeer flexibel. Het is bijvoorbeeld mogelijk om met oneindig grote lijsten te werken. Pas op het moment dat een deelverzameling uit de lijst daadwerkelijk wordt opgevraagd, worden de berekeningen hiervoor uitgevoerd. Het nadeel is dat het benodigde geheugen pas tijdens het draaien gealloceerd wordt, wat het lastig maakt om van te voren over het geheugengebruik te redeneren. In een stricte taal treedt dat probleem echter niet op.

Kortom, in principe is er geen reden om FP niet in te zetten in systemen met beperkte hardwarebronnen en voor realtime toepassingen, zolang je maar goed let op welke taal je gebruikt. Hammond heeft bijvoorbeeld responstijden van 1 ms gehaald en zijn Hume-code in 4 kb Ram weten te persen. Bovendien is een programma vrij eenvoudig te tunen, meent Hammond: 'in tegenstelling tot imperatieve talen is het doorgaans eenvoudig om vrij radicale veranderingen door te voeren zonder grote hoeveelheden code weg te gooien. Dat betekent dat de gevolgen van kleine ontwerpfouten veel kleiner zijn en dat er meer ruimte is voor code-evolutie en -hergebruik.'

'Als je echt per se registers wil beïnvloeden, kan je dat natuurlijk wel beter in een taal schrijven die daarvoor bedoeld is. Een geboren en getogen FP-programmeur zal dat misschien ook wel kunnen, maar het gaat misschien toch niet zo handig als in een imperatieve taal', zegt Van Eekelen.

Dan is er nog een voordeel aan de functionele aanpak, wat juist van toepassing is op veel embedded systemen: omdat de programma's eigenlijk in wiskundige notatie staan, is het in veel gevallen mogelijk om de correctheid van de implementatie mathematisch te bewijzen, de heilige graal voor safety-critical systemen. Buffer overflows, deadlocks en race conditions spelen geen rol.

Blijft de vraag staan waarom we dan zo weinig functioneel programmatuur zien in embedded systemen. Dat lijkt vooral een gevolg te zijn van 'onbekend maakt onbemind'. Bovendien is het nog niet zo lang dat functionele talen de prestaties en voorspelbaarheid van imperatieve implementaties benaderen.

Hoe ziet de toekomst eruit voor FP? Van Eekelen constateert een 'heel langzame' groei. Volgend Hammond zijn bedrijven enthousiast over Hume, ondanks dat het nog in de ontwikkelfase verkeert. Van Eekelen denkt dat de FP-aanpak op termijn misschien wel meer acceptatie zal krijgen vanuit de theorem prover-aanpak: 'academisch gebeurt daar nu veel aan. Je kan zeggen dat theorem provers, zoals PVS, Coq en ACL2, gebaseerd zijn op een functioneel principe waarin je eigenschappen kunt bewijzen en ook kunt programmeren. Dan zou je daarin kunnen programmeren, de gewenste eigenschappen bewijzen en van daaruit gelijk kunnen compileren. De kracht ervan is dat je bewijs door de computer volautomatisch gecontroleerd kan worden zodat de hoogst denkbare betrouwbaarheid hiermee kan worden verkregen. Zo hebben we bijvoorbeeld voor een recente LaQuSo opdracht van Rijkswaterstaat de kern van de code van de Maeslantkering in een theorem prover gemodelleerd en bewezen dat de code conform de specificatie is.' Maar voordat de aanpak doorbreekt voor meer generiek programmeren, zal hij toch minder onbekend moeten worden. Wellicht dat multicore hiervoor kan zorgen; eindelijk een killer app voor functioneel programmeren.