Een verklaring voor de eenvoud en gemak van Bird's stijl

Maarten Fokkinga, 15 april 1987

In vorige verhaaltjes heb ik laten zien hoe ogenschijnlijk ingewikkelde algoritmen heel eenvoudig, duidelijk en begrijpelijk in de stijl van Bird kunnen worden uitgedrukt. Steeds heb ik gezegd dat ~~de~~ het programmeren op functie-niveau (dwz. abstract, in de stijl van Bird) misschien wel meer denkwerk vereist dan het programmeren op object-niveau (dwz. direkt in termen van de individuele data-elementen en met expliciete recursie), maar dat het denkwerk allerzins de moeite waard is gezien de elegantie (kwaliteit, begrijpelijkheid etc) van de resulterende programma's. Ik krijg nu zo langzaamaan de overtuiging dat het programmeren in de stijl van Bird niet méér denkwerk vereist, maar integendeel juist minder. In dit verhaal zal ik de redenen daarvoor proberen te expliciteren.

Redenen voor het gemak van programmeren à la Bird.

1. Lijstoperaties. De lijstoperaties die Bird presenteert in Hoofdstuk 3 zijn uitstekend gekozen: veel lijstmanipulaties zijn er mee uit te drukken (zonder zélf operaties te bouwen m.b.v. recursie!) en, wat zeker

zo belangrijk is, ze sluiten goed aan bij informeel gestelde manipulaties met lijsten. Operaties zoals takewhile, dropwhile, map, segments en dergelijke zijn precies de operaties die je intuitief zou willen gebruiken als "elementaire" operatie.

2. Abstractie. (Onder abstractie versta ik: het weglaten (niet in beschouwing nemen) van sommige aspecten. 'Abstract' is dus geen synoniem voor 'moeilijk'.) In Bird's stijl --althans in de eerste hoofdstukken-- wordt het aspect van machinale evaluatie volkomen achterwege gelaten; (dat komt pas in Hoofdstuk 5 aan bod en ook daar nog in abstracte vorm!). Bijgevolg is er geen reden om "grote bewerkingen" uit te drukken in "heel kleine stapjes op de individuele data-elementen". Juist dit laatste speelt zo'n grote rol in Pascal (en nog grotere rol in Assembler, en iets minder in Functioneel-op-object-niveau) en levert zo veel moeilijkheden op bij beginnende programmeurs. Voor een beginnend programmeur (en voor een non-programmeur) kan een algoritme in grote lijnen duidelijk zijn, terwijl juist het uitdrukken (realiseren) daarvan of in elementaire machine-stapjes het grote struikelblok vormt.

3. Functie-compositie. Een constructieve beschrijving van een gewenste uitkomst wordt vaak gegeven als een aaneenrijging van, hoe kan het anders, 'constructies'. Dus zoiets als: "pas deze constructie toe, daarna die, dan die, enzovoorts." [Let wel, de imperatieve en tijds- en sequentie-aspecten in zo'n samengestelde constructie zijn in feite alleen maar data-afhankelijkheden; eigenlijk luidt de samengestelde constructie: "pas deze constructie toe en op ~~het~~ de uitkomst daarvan die, en daarop weer die, enzovoorts."] Functie-compositie is het middel bij uitstek om zo'n aaneenrijging uit te drukken, en vervult dezelfde essentiele rol als de punt-komma in beschrijvingen zoals:

itereer (eerst) die en die functie;
neem (dan) zo en zo beginstuk van de lijst;
pas (vervolgens) functie split toe;
druk (tenslotte) de uitkomst af.

[Inderdaad, we zouden $(h \cdot g \cdot f)\ x$ net zo goed kunnen noteren met: x subjectto $(f; g; h)$.] Kortom, functie-compositie is niet iets moeilijks dat pas "in tweede ronde" onderwezen kan worden, maar het komt van nature voor in de formulering van constructies. Bird introduceert functie-compositie dan ook al in Hoofdstuk 1 (Introduction).

4. Afwezigheid recursie. Overbodigheid in het algemeen houdt eenvoud en gemak niet ten goede. Dat geldt ook voor overbodige recursie. Het blijkt dat veel algoritmen waarvan ik vroeger dacht dat recursie ervoor nodig is, nu zonder recursie kunnen worden uitgedrukt. Dat komt omdat de nodige recursie dan al verborgen zit in wat ik verder als "elementaire" operaties beschouw, zoals foldl, foldr, map, iterate.

5. Efficiëntie-beschouwingen.*) Een belangrijk aspect waarin programmeren van constructieve wiskunde verschilt is de aandacht en zorg voor de efficiëntie van de constructies. Ook met "Bird's stijl" zijn efficiëntie-beschouwingen mogelijk. Vanwege het "functie-nivo" der algoritmen zijn er slechts globalere (i.e. grootte-orde) uitspraken mogelijk dan wanneer de algoritmen zijn uitgedrukt in bewerkingen op de elementaire data-elementen. Bijvoorbeeld, omdat takewhile als elementair wordt genomen, kan en hoef je de recursieve definitie ervan niet te inspecteren, maar kun je volstaan met de uitspraak dat de bewerkingsduur hooguits evenredig is met de grootte van de uitkomst. Het lijkt mij waarschijnlijk dat iemand die op object-nivo z'n algoritme uitdrukt,

*) Meer HOOP dan FEIT?-- de wens is de vader van de gedachte

ook op object-nivo zin efficiëntie-beschouwingen geeft. Bird's stijl maakt dit laatste haast onmogelijk; wat overblijft is een algemenere en globalere efficiëntie-beschouwing -- dus eenvoudiger en gemakkelijker.

Een bekentenis.

Toen ik een kleine maand geleden Bird's boek voor het eerst onder ogen kreeg, dacht ik dat zijn stijl van programmeren te moeilijk zou zijn. Het leek mij toe dat ervaring op "object-nivo" nodig is vóór je zinvol op "functie-nivo" kunt denken. Door diverse probeersels ben ik intussen van gedachte veranderd. Het leek mij nuttig (ook en mede in het kader van de HOP activiteiten) om de redenen daarvoor zo expliciet mogelijk te formuleren. Vandaar bovenstaande verhaal.

* * *

Alhoewel "Bird's stijl" het programmeren vergemakkelijkt, beweer ik niet dat Bird's Boek makkelijk is. Naast het uitdrukken van algoritmen houdt hij zich ook bezig met correctheid (zie bijv. Hoofdstuk 4),

systematische afleiding van algoritmen, en programmatransformaties met het oog op efficiëntie-aspecten (de Duality Theorems voor foldl en foldr). Je kunt onmogelijk verwachten dat het behandelen ervan als gemakkelijker wordt ervaren dan het simpelweg niet behandelen ervan. Dat laatste gebeurt in het huidige programmeeronderwijs.