Law and Order in Algorithmics

Maarten Fokkinga
Vakgroep SETI, fac INF, Universiteit Twente

Versie van May 25, 1993

Algoritmiek (engels: algorithmics) is de theorie en praktijk van het algebraïsch redeneren over programma's. Een elementaire stap in een algebraïsche redenering over programma's wordt wet genoemd (engels: law). Het systematisch ontdekken en gebruiken van wetten voor programma's is het onderwerp van mijn promotie-onderzoek geweest. Dit verhaal geeft een overzicht van dat onderzoek.

1 Inleiding. In de periode juli 1988 – juli 1991 ben ik gedetacheerd geweest naar het CWI (Centrum voor Wiskunde en Informatica) in Amsterdam, om onderzoek te doen op het gebied van algoritmiek. Dit heeft geleid tot verscheidene publicaties en mijn proefschrift *Law and Order in Algorithmics* (Universiteit Twente, februari 1992). Dit verhaal geeft een beschrijving van dat onderzoek, alsmede van de behaalde resultaten.

2 Waar gaat het over? Algoritmiek betreft een aspect van computerprogrammeren. Er zijn veel (en ook heel belangrijke) aspecten van programmeren die buiten algoritmiek vallen. Laten we ter verkoming van misverstanden eerst de grenzen goed afbakenen.

De weg van klant naar computerprogramma ziet er zeer vereenvoudigd in grote lijnen als volgt uit:

$$\text{klant} \;\Rightarrow\; \text{behoefte} \;\Rightarrow\; \text{specificatie} \;\boxed{\Rightarrow}\; \text{programma} \;\Rightarrow\; \text{gebruik}.$$

Het werven van potentiële klanten valt buiten algoritmiek, evenals het achterhalen (of aanpraten) van de behoefte aan programmatuur. Ook de uiterst moeilijke stap die daarna volgt, komt niet aan de orde: het precies specificeren van de informele wensen van de klant. Het is de

constructie van programma's uit een gegeven specificatie

dat het onderwerp van algoritmiek is. Het gebruik van de programma's, en het aanpassen en "onderhoud" van programmatuur, valt er weer buiten.

Algoritmiek: wat is dat?

Een eerste, grove, omschrijving van algoritmiek is: het redeneren over programma's; een preciese omschrijving komt verderop. Als je in staat bent om goed over programma's te *redeneren*, dan kan dat van hulp zijn bij het *maken* van programma's uit een gegeven specificatie (en dát is de motivatie voor het onderzoek). Laat me dit eerst uitleggen.

3 Het gebruik van algoritmiek. Vaak kunnen de wensen van een opdrachtgever gedeeltelijk beschreven worden in de vorm van een programma. Dus een deel van de specificatie *is* dan al een programma. Een programma dat als specificatie dienst doet, geeft het gewenste invoer-uitvoer gedrag overduidelijk weer, maar is —juist daarom— vaak hopeloos inefficiënt en, soms, alleen op een geïdealiseerde computer te verwerken en alleen onder geïdealiseerde omstandigheden met *oneindig* veel opslagruimte en rekentijd.

Het maken van het uiteindelijke, commerciële programma kan nu gebeuren door het specificerende programma geleidelijk om te vormen, te *transformeren*, tot een efficiënter programma; een programma dat nog steeds het gewenste invoer-uitvoer gedrag vertoont, maar wel efficiënt is, en ook eventuele andere eisen uit de specificatie vervult. Aldus is 'het maken van programma's uit een gegeven specificatie' teruggebracht tot 'het transformeren van een gegeven programma' waarbij de efficiëntie uiteindelijk wordt verhoogd en het invoer-uitvoer gedrag in stand wordt gehouden. Deze manier van programmeren heet *transformationeel programmeren*. Hierbij moet je weten welke programmadelen door welke andere vervangen kunnen worden, zonder dat het invoer-uitvoer gedrag verandert. Anders gezegd, je moet kunnen redeneren over de *gelijkheid qua invoer-uitvoer gedrag*. Het is dit soort van redeneren over programma's dat een hulp kan zijn bij het programmeren, en dat het onderwerp is van algoritmiek.

4 Algoritme. Een programma heeft vele aspecten. Hier zijn een paar voorbeelden:

- Het invoer-uitvoer gedrag: wat is de uitkomst bij welke invoer?
- De interactie met de omgeving: hoe wordt de invoer gepresenteerd, en hoe de uitvoer? Via toetsenbord en beeldscherm, of via menu's en muis-klikken?
- De snelheid: het ene programma produceert de uitkomst heel snel, terwijl een ander programma met hetzelfde invoer-uitvoer gedrag heel traag kan zijn.
- De zogenaamde tijds-complexiteit: het aantal elementaire stappen dat nodig is om de uitvoer te produceren bij gegeven invoer. De tijds-complexiteit is een abstractie van de snelheid.
- De benodigde computercapaciteit: draait het programma op een PC, of heb je er een supercomputer voor nodig?
- De computertaal waarin het programma geschreven is.

Het is ondoenlijk om met ál die aspecten rekening te houden, wanneer je een theorie wil opzetten over het redeneren over programma's. (Trouwens, ook voor de programmeur zelf is het verstandig om niet alles tegelijk te willen doen; hij kan zich beter eerst om de correctheid bekommeren, en daarna pas om de syntactische eigenaardigheden van de programmeertaal.) Wanneer je bij een programma afziet (abstraheert) van alle aspecten behalve het invoer-uitvoer gedrag, dan houd je een wiskundige *functie* over. Laat je echter ook de tijds-complexiteit nog een rol spelen dan houd je een *algoritme* over. Dus, een algoritme is een programma waarbij alleen het invoer-uitvoer gedrag en de tijds-complexiteit telt; een functie is een programma waarbij alleen het invoer-uitvoer gedrag telt.

We kunnen nu een nauwkeuriger omschrijving geven van 'algoritmiek': het redeneren over de *gelijkheid van programma's* waarbij er wordt afgezien van een aantal aspecten:

> **algoritmiek** *is de theorie en praktijk van het algebraïsch redeneren over algoritmen.*

Een uitleg van 'algebraïsch' volgt zo dadelijk.

Mijn onderzoek ging voornamelijk over functies: de tijds-complexiteit komt alleen informeel aan bod. De resultaten van het onderzoek zijn direct toepasbaar in *functioneel programmeren*, zoals programmeren in Miranda.

5 Wat is algebra? Kort gezegd is algebra: *het rekenen met formules*. Verreweg het bekendst is het rekenen met formules die getallen voorstellen. Bij dat rekenen (transformeren) worden de stappen gerechtvaardigd met rekenregels ofwel **wetten** (engels: **law**, zoals in de titel van mijn proefschrift). Hier is een voorbeeld van zo'n wet voor getallen:

$$x^2 - y^2 \quad = \quad (x + y) \times (x - y)\,.$$

Hier is een wet voor functies:

$$(\!|x|\!) \vartriangle (\!|y|\!) \quad = \quad (\!|x \times y \circ \mathit{abide}|\!)\,. \qquad \text{Banana Split}$$

In de eerste wet staan x en y voor willekeurige getallen. De wet geeft aan dat linker- en rechterlid gelijke getallen aanduiden. In de tweede wet staan x en y voor willekeurige programma's, en is *abide* een bekend programma, zoals π en e bekende getallen zijn. De wet geeft aan dat linker- en rechterlid programma's zijn met gelijk invoer-uitvoer gedrag. Maar de twee programma's hebben een verschillende vorm, en daarom mogelijk een verschillende tijds-complexiteit (dus efficiëntie en snelheid). De haakjes $(\!|\ |\!)$ noemen we bananen-haakjes, en het symbool $\vartriangle$ wordt uitgesproken als 'split'; deze wet heeft dan ook de naam Banana Split gekregen.

> *Om over te slaan: uitleg van de wet* Banana Split.
> De wet formuleert een bekende programmeertechniek héél algemeen: twee inductief gedefinieerde functies kunnen gecombineerd worden tot één inductief gedefinieerde functie (de "paring" van de twee functies). Ik zal de formules hier iets toelichten.

De operaties $\triangle$, $\times$ stellen functies samen tot nieuwe functies, zoals operaties div en $\times$ getallen samenstellen tot nieuwe getallen. Zij zijn als volgt gedefinieerd.

$$\begin{array}{lcll} f \triangle g & = & \text{de functie} \quad x \mapsto (fx,\, gx) & \text{“}f \text{ gepaard met } g\text{”} \\ f \times g & = & \text{de functie} \quad (x,y) \mapsto (fx,\, gy) & \text{“}f \text{ naast } g\text{”}\,. \end{array}$$

Een preciese definitie van de haakjes $(\!|\ |\!)$ is hier nog niet mogelijk. Dat komt door de verregaande abstractie en algemeenheid van $(\!|\ |\!)$. Een correcte, maar misschien nogal vage omschrijving is de volgende:

$$\begin{array}{lcl} (\!|f|\!) & = & \text{de inductief gedefinieerde functie} \\ & & \text{waarvan de basis en de recursiestap bepaald zijn door } f\,. \end{array}$$

Dus het linkerlid van Banana Split is de paring van twee inductief gedefinieerde functies (bepaald door x en y), en het rechterlid is één inductief gedefinieerde functie (bepaald door $x \times y$ en de bekend veronderstelde *abide*). De wet zegt dat beide functies gelijk zijn.

Hier is een toepassing van de wet. Laat *sum* en *prod* functies zijn die de som en het product van een rij getallen opleveren. Deze functies kunnen ieder met inductie naar de opbouw van rijen gedefinieerd worden. Volgens Banana Split kan ook $sum \triangle prod$ met inductie gedefinieerd worden. De functie $sum \triangle prod$ levert bij iedere rij het paar (s,p) van de som s en het product p van de rij.

6 Over de notatie van functies. Bij algebra, het rekenen met formules, wordt steeds een formule ontleed in zijn onderdelen, en wordt uit die onderdelen een nieuwe formule samengesteld op ándere wijze (maar zónder de betekenis van de formule te veranderen). Het gaat hierbij niet om de *syntactische* onderdelen (louter de tekens en symbolen), maar om de *semantische* onderdelen, dat wil zeggen die onderdelen die op zichzelf betekenis dragen en bepalend zijn voor de betekenis van het geheel. Bij algoritmiek is het daarom belangrijk dat de notatie van functies (algoritmen, programma's) de semantische onderdelen goed weergeeft. De traditionele notaties zoals in Modula-2 en Miranda doen dit niet altijd. Bijvoorbeeld, beschouw de functie *sum* die de som van een rij getallen oplevert:

$$\begin{array}{lcl} sum\,(\text{lege-rij}) & = & 0 \\ sum\,(x \text{ gevolgd-door } xs) & = & plus(x, sum\,(xs))\,. \end{array}$$

In de notatie van mijn proefschrift luidt de definitie van *sum*:

$$sum \quad = \quad (\!|\underline{0} \triangledown plus|\!)\,.$$

In het rechterlid van deze laatste definitie zijn de semantische onderdelen duidelijker te onderscheiden dan in de traditionele notatie: het recursie-patroon wordt genoteerd met $(\!|\ |\!)$, en de overige semantische onderdelen zijn de konstante 0 en de optel-operatie *plus*. Miranda-programmeurs zullen hier de haakjes $(\!|\ |\!)$ als *foldr* herkennen:

$$sum \quad = \quad foldr\ plus\ 0\,.$$

Functie *foldr* is specifiek voor rijen, terwijl de haakjes $(\!|\ |\!)$ heel algemeen zijn voor willekeurig 'inductief' datatype. (In feite hoort $(\!|\ |\!)$ nog een parameter te hebben die de inductieve structuur van het datatype aangeeft.)

De parameters van de functie *sum* worden in het geheel niet meer genoteerd. De ontleding in de semantische onderdelen is in het algemeen gemakkelijker wanneer we *dummy variabelen* (zoals formele parameters van functie-definities) en functie-toepassing vermijden, en in plaats daarvan *functie-compositie* (genoteerd met $\circ$) en andere samenstellingen van functies (zoals $\vartriangle$, $\times$, en $\triangledown$ hierboven) gebruiken.

7 Voorbeeld. Ter illustratie van algebraïsch redeneren, volgen hier nog twee voorbeelden van het rekenen met formules. Allereerst een bewijs van de wet $a^2 - b^2 = (a+b) \times (a-b)$. Dit moet voor iedere lezer te volgen zijn. We beginnen met het rechterlid, omdat dat het ingewikkeldst is, en transformeren dat stap voor stap tot het linkerlid.

$$
\begin{array}{ll}
 & (a+b) \times (a-b) \\
= & \quad \text{distributie van } \times \text{ over } - \\
 & (a+b) \times a - (a+b) \times b \\
= & \quad \text{distributie over } + \text{ van } \times \\
 & (a \times a + b \times a) - (a \times b + b \times b) \\
= & \quad \text{commutativiteit van } \times \text{, associativiteit van } + \\
 & a \times a + a \times b - a \times b - b \times b \\
= & \quad \text{definitie van kwadrateren, associativiteit van } + \\
 & a^2 - b^2.
\end{array}
$$

Nu het tweede voorbeeld: een deel van een bewijs van de Banana Split wet. Doe niet te veel moeite het te begrijpen; het dient alleen ter illustratie van 'het rekenen met formules (die functies voorstellen)'. [Ter informatie: F geeft de inductieve structuur aan van het datatype, en α staat voor de constructor(s) van het datatype; deze twee zijn in feite parameters van $(\!|\ |\!)$, maar hebben we voor de leesbaarheid nergens expliciet in de notatie vermeld.] Formule $(\!|x|\!) \vartriangle (\!|y|\!)$ wordt afgekort tot Δ.

$$
\begin{array}{ll}
 & x \times y \ \circ \ abide \ \circ \ F\Delta \\
= & \quad \text{definitie: } abide = Fexl \vartriangle Fexr \\
 & x \times y \ \circ \ Fexl \vartriangle Fexr \ \circ \ F\Delta \\
= & \quad \text{distributie eigenschap: } f \times g \ \circ \ h \vartriangle j \ \circ \ k = (f \circ h \circ k) \vartriangle (g \circ j \circ k) \\
 & (x \circ Fexl \circ F\Delta) \quad \vartriangle \quad (y \circ Fexr \circ F\Delta) \\
= & \quad \text{functor eigenschap: } F(f \circ g) = Ff \circ Fg \\
 & (x \circ F(exl \circ \Delta)) \quad \vartriangle \quad (y \circ F(exr \circ \Delta)) \\
= & \quad \text{afkorting: } \Delta = (\!|x|\!) \vartriangle (\!|y|\!) \\
 & \quad \text{eigenschap: } exl \circ f \vartriangle g = f \text{ en: } exr \circ f \vartriangle g = g
\end{array}
$$

$$
\begin{array}{ll}
 & (x \circ F(\!|x|\!)) \quad \vartriangle \quad (y \circ F(\!|y|\!)) \\
= & \quad \text{definierende eigenschap van } (\!|\,|\!)\colon\ z \circ F(\!|z|\!) = (\!|z|\!) \circ \alpha \\
 & ((\!|x|\!) \circ \alpha) \quad \vartriangle \quad ((\!|y|\!) \circ \alpha) \\
= & \quad \text{distributie eigenschap: } (h \circ k) \vartriangle (j \circ k) = h \vartriangle j \circ k \\
 & (\!|x|\!) \vartriangle (\!|y|\!) \circ \alpha.
\end{array}
$$

De begrippen algoritmiek, algebra en wet zijn nu hopelijk duidelijk. Het wordt tijd om het onderzoek zelf te beschrijven.

Verslag van het onderzoek

8 Hoofddoel. Het hoofdoel van mijn onderzoek in algoritmiek luidde:

het systematisch ontdekken en gebruiken van
wetten
voor functies en dus programma's.

Een voorbeeld hiervan is de "ontdekking" van de Banana Split wet. Op zichzelf is die wet niet zo verrassend; de verdienste is veeleer dat die nu *formeel* bewezen is in een *heel algemene* vorm.

9 Methode van onderzoek. De vraag komt wellicht op hoe je zo'n onderzoek doet. Voor een deel is de volgende methode gevolgd. Bekijk een programma-ontwikkeling en probeer zo veel mogelijk te generaliseren: welke eigenschappen zijn echt gebruikt, welke waren overbodig, welke structuur is echt gebruikt, enzovoorts. Probeer dan de programma-ontwikkeling en met name de gebruikte wetten heel algemeen (dus abstract) te formuleren. Probeer ook overeenkomsten te ontdekken met reeds bekende transformaties en wetten: zijn die overeenkomsten toevallig of af te leiden uit een nog algemenere vorm? Aldus krijgt de onderzoeksactiviteit voor een deel het karakter van *patroonherkenning* (in de gebruikte formules) en pogingen tot generalistie en unificatie. Daarnaast zijn er nog allerlei vermoedens geuit door andere mensen die zich ook met dit onderwerp bezig houden, en die een formeel bewijs behoeven.

10 Hulpmiddelen — Categorie-theorie. De zojuist beschreven methode vereist wel dat er een formalisme is dat geschikt is voor dergelijke generalisaties, en dat er begrippen zijn waarmee je goed kunt abstraheren van de toevallige datatypen die in een voorbeeld-programma gebruikt worden. Zo'n formalisme en zo'n begrippenapparaat blijken aanwezig te zijn in een vrij nieuwe tak van wiskunde: *categorie-theorie*. Deze theorie is oorspronkelijk opgezet om verschillende takken van wiskunde op gelijke manier te beschrijven, zodat ook resultaten uit de ene tak van wiskunde naar een andere tak overgezet kunnen worden. De begrippen in categorie-theorie zijn dan ook verregaande abstracties en generalisaties van diverse wiskundige begrippen. Lange tijd heeft categorie-theorie de naam van "general abstract nonsense" gehad, maar tegenwoordig wordt het steeds meer als waardevol

erkend, ook binnen de informatica. Voor de algemeenheid van de resultaten in mijn onderzoek zijn de begrippen *functor*, *initialiteit*, *natuurlijkheid* en *dualiteit* onmisbaar gebleken. In Hoofdstuk 3 van mijn proefschrift worden datatypen met behulp van deze begrippen beschreven en onderzocht.

11 Resultaten — algemeen. Zoals al gezegd heeft het onderzoek geleid tot de ontdekking van een aantal wetten die op zich misschien niet zo verrassend zijn, maar waarvan de waarde vooral ligt in het feit dat ze zeer algemeen toepasbaar zijn én formeel bewezen. Dit staat in mijn proefschrift voornamelijk in Hoofdstukken 3 en 4. Die wetten hebben vooral betrekking op "inductieve datatypen" en de daarop gedefinieerde functies. (In categorie-theorie zijn deze datatypen de *initiële* algebra's.) Voorbeelden van dergelijke datatypen zijn de natuurlijke getallen, *eindige* rijen en *eindige* bomen. Voor dergelijke datatypen kan een functie gedefinieerd worden met *inductie naar de opbouw van het argument*; zoals de functie *sum* die al eerder is besproken. Maar ook anderssoortige datatypen komen aan bod (in categorie-theorie de *duale* van de vorige): datatypen waarbij een functie gedefinieerd kan worden met *inductie naar de afbraak van het resultaat*. Een voorbeeld hiervan is het datatype van *oneindige* rijen, en de functie *from* die bij argument n de oneindige rij $[n, n+1, n+2, n+3, \ldots]$ oplevert. Er geldt:

$$\begin{array}{lcll} \mathit{head}(\mathit{from}(n)) & = & n & \text{de kop van het resultaat} \\ \mathit{tail}(\mathit{from}(n)) & = & \mathit{from}(n+1) & \text{de staart van het resultaat}\,. \end{array}$$

Deze vergelijkingen definiëren *from* niet met inductie naar de opbouw van het argument n, maar met inductie naar de afbraak van het resultaat: de vergelijkingen beschrijven, met recursie, wat de kop en staart van het resultaat van $\mathit{from}(n)$ is.

12 Resultaten — het begrip 'wet'. Beschouw nogmaals het datatype der eindige rijen, die we noteren met $[a_0, \ldots, a_{m-1}]$. We definieren:

$$\begin{array}{lcl} \mathit{nil} & = & [\,] \\ \mathit{tip}(a) & = & [a] \\ a \text{ cons } [a_0, \ldots, a_{m-1}] & = & [a, a_0, \ldots, a_{m-1}] \\ [a_0, \ldots, a_{m-1}] \text{ join } [b_0, \ldots, b_{n-1}] & = & [a_0, \ldots, a_{m-1}, b_0, \ldots, b_{n-1}]\,. \end{array}$$

(Wanneer een functie van twee argumenten als infix operator gebruikt wordt, noteren we die in het roman font: ' cons ' in plaats van ' *cons* '.) De *nil* en *cons* vormen samen een stel constructoren omdat iedere rij daarmee te schrijven is:

$$[a_0, \ldots, a_{m-1}] \quad = \quad a_0 \text{ cons } (a_1 \text{ cons } \ldots (a_{m-1} \text{ cons } \mathit{nil}))\,.$$

Maar ook *nil*, *tip* en *join* zijn constructoren:

$$[a_0, \ldots, a_{m-1}] \quad = \quad \mathit{nil}, \qquad\qquad \text{if } m = 0$$

$$= \quad tip(a_0) \,\mathrm{join}\, \ldots \,\mathrm{join}\, tip(a_{m-1}), \quad \text{if } m > 0\,.$$

Voor het stel constructoren nil, tip en $join$ gelden de volgende wetten:

$$\begin{array}{ll} nil \,\mathrm{join}\, xs \;=\; xs \;=\; xs \,\mathrm{join}\, nil & (\,nil \text{ is neutraal voor } join\,) \\ (xs \,\mathrm{join}\, ys) \,\mathrm{join}\, zs \;=\; xs \,\mathrm{join}\, (ys \,\mathrm{join}\, zs) & (\,join \text{ is associatief})\,. \end{array}$$

Dus, zo beschouwd, is het datatype der eindige rijen een datatype met constructoren die aan zekere wetten voldoen. Om heel algemeen een definitie te kunnen geven van het begrip 'datatype-met-wetten', is er een definitie nodig van het begrip 'wet'. Zoiets bestond er nog niet, althans niet in die abstracte en algemene vorm die past bij de manier waarop het begrip datatype geformaliseerd is. Ik heb zo'n definitie voorgesteld en het begrip 'wet' op zijn eigenschappen onderzocht. Dit gebeurt in Hoofdstuk 5 van mijn proefschrift. De gebruikelijke wetten blijken aan de definitie te voldoen. En veel (alle?) uitspraken over wetten waarvan je verwacht dat ze waar zouden moeten zijn, blijken ook op grond van die definitie *formeel* bewezen te kunnen worden.

Voorbeeld. Hier is een voorbeeld van een bewering over wetten in het algemeen: *optilling behoudt wetten.* Laat $+$ een binaire operatie zijn, denk aan optelling van getallen. Definieer nu een operatie $\hat{+}$ (de $+$ opgetild tot het nivo van functies) als volgt:

$$f \,\hat{+}\, g \quad = \quad \text{de functie } \; x \mapsto fx + gx\,.$$

Dan geldt dat $\hat{+}$ commutatief is als $+$ dat is, en omgekeerd:

$$(\forall x, y :: \quad x + y = y + x) \qquad \Leftrightarrow \qquad (\forall f, g :: \quad f \,\hat{+}\, g = g \,\hat{+}\, f)\,.$$

Deze concrete bewering is eenvoudig te bewijzen:

$$\begin{array}{ll} & \forall f, g :: \qquad f \,\hat{+}\, g = g \,\hat{+}\, f \\ \Leftrightarrow & \quad \text{gelijkheid van functies is gelijkheid voor alle argumenten} \\ & \forall f, g, z :: \quad (f \,\hat{+}\, g)(z) = (g \,\hat{+}\, f)(z) \\ \Leftrightarrow & \quad \text{definitie van } \hat{+} \\ & \forall f, g, z :: \quad fz + gz = gz + fz \\ \Leftrightarrow & \quad \text{voor } \Rightarrow \text{ kies, bij gegeven } x, y\,, \text{ functies } f, g \text{ zó dat } fz = x \text{ en } gz = y\,; \\ & \quad \text{voor } \Leftarrow \text{ kies, bij gegeven } f, g, z\,, \text{ de } x, y \text{ zó dat } x = fz \text{ en } y = gz \\ & \forall x, y :: \qquad x + y = y + x. \end{array}$$

Maar hoe bewijs je nou dat zoiets voor *alle* wetten geldt (niet alleen voor de commutativiteit), en voor alle operaties $+$ (en niet alleen voor binaire operaties)? Daarvoor moet je een formele karakterisering van 'wet' hebben, en die wordt in mijn proefschrift gegeven. Het bewijs van de stelling 'optilling behoudt wetten', dat wil zeggen:

$$\text{operatie } \hat{\varphi} \text{ vervult wet } W \text{ precies wanneer operatie } \varphi \text{ wet } W \text{ vervult}$$

is inmiddels geleverd door collega's in het onderzoeksproject. Deze stelling is een reden om een opgetilde operatie $\hat{+}$ exact hetzelfde te noteren als de onderliggende operatie $+$: de algebraïsche wetten zijn voor beide gelijk, en in het algebraïsch redeneren hoef je niet te weten of je met de opgetilde dan wel de onderliggende operatie te maken hebt.

13 Resultaten — 'Order'. In het overgrote deel van het onderzoek is van een vereenvoudigende veronderstelling uit gegaan. Namelijk, dat iedere functie *totaal* is, dat wil zeggen bij iedere invoer een uitkomst heeft. Dat is eigenlijk wel een redelijke veronderstelling, vind ik, want wat heb je nou aan functies die bij sommige invoer geen uitkomst hebben? Toch komen in de praktijk van het programmeren *partiële* functies veel voor, dat wil zeggen functies waarbij er voor sommige invoer geen uitkomst is. Beschouw bijvoorbeeld de volgende vergelijkingen:

$$\begin{array}{lcl} f(0) & = & 17 \\ f(n+1) & = & f(n+1)\,. \end{array}$$

In sommige programmeertalen kunnen zulke vergelijkingen als 'definitie' van een functie f gegeven worden. Maar ... iedere partiële of totale functie f die bij invoer 0 de uitkomst 17 geeft, voldoet aan de vergelijkingen. Functie f is dus niet eenduidig bepaald; en in die zin vormen de vergelijkingen dus géén definitie. Er is wél een "kleinste" (= meest partiële) functie f' die aan de vergelijkingen voldoet, namelijk:

$$\begin{array}{ll} f'(0) & = 17 \\ f'(n+1) & \text{heeft geen uitkomst.} \end{array}$$

We kunnen afspreken dat deze partiële functie f' degene is die door de vergelijkingen wordt 'gedefinieerd'. Wiskundig gezien is dit zinvol omdat de "kleinste" oplossing bestaat en eenduidig bepaald is. Dat is ook de functie die in de meeste programmeertalen door zo'n stel vergelijkingen 'gedefinieerd' wordt. Om dit wiskundig allemaal netjes te formaliseren en te bewijzen, wordt er een **ordening** gedefinieerd op de verzameling van functies, zodat het begrip 'kleinste' formele betekenis krijgt. Het engels voor *ordening* is **order**; dit verklaart dan het derde trefwoord uit de titel van het proefschrift.

Vergelijkingen van de vorm:

$$f(x) \quad = \quad \ldots f \ldots$$

heten wel 'recursieve definities'. Recursie wordt in de praktijk veel gebruikt, maar in een groot aantal gevallen is recursie louter *inductie naar de opbouw van het argument* of *inductie naar de afbraak van het resultaat*, zoals al in paragraaf 11 besproken is.

Ik wilde in het onderzoek ook algemene 'recursieve definities' toe laten. Een gevolg daarvan is dat partiële functies in plaats van totale functies het uitgangspunt worden. Daardoor zijn er sommige aannamen niet meer vervuld die wel bij totale functies vervuld zijn, en tot nu toe veelvuldig in de bewijsvoering zijn gebruikt. Bijvoorbeeld, door partiële functies toe te laten is er geen operatie $\triangledown$ met al die eigenschappen die nodig zijn om twee

functies f en g samen te vatten tot één, $f \triangledown g$, waaruit f en g weer gereconstrueerd kunnen worden; operatie $\triangledown$ kwam al te voorschijn in paragraaf 6: $sum = (\!|\underline{0} \triangledown plus|\!)$. (Voor de ingewijden: in categorie CPO bestaat er geen co-pruduct.)

Hoofdstuk 6 van mijn proefschrift beschrijft wat er van de mooie wetten en de systematiek overblijft wanneer je de aanname van totaliteit van functies laat vallen, en in plaats daarvan ook partiële functies in de beschouwingen een rol laat spelen. Het blijkt dat veel wetten onverminderd waar blijven behoudens wat extra toe te voegen condities (die 'strictheid' van sommige onderdelen eisen).

14 Resultaten — categorie-theorie. Zoals gezegd was het hoofddoel van mijn onderzoek het systematisch ontdekken (en gebruiken) van wetten voor functies en dus voor programma's. Om een voldoende algemeenheid (en systematiek!) te bereiken heb ik het formalisme van categorie-theorie gehanteerd. Met name 'initialiteit' speelt een grote rol in de behandeling van datatypen. Maar initialiteit is een begrip dat ook los van datatypen een grote rol speelt binnen de categorie-theorie. En voorts bleek dat de wetten voor datatypen voor een groot deel volgen uit initialiteit alleen. Die wetten en hun bewijzen kunnen dus algemener geformuleerd worden dan in de context van algoritmiek en datatypen. Dit heeft me er toe gebracht om binnen de categorie-theorie dezelfde werkwijze te volgen als ons voor ogen staat bij algoritmiek: algebraïsch redeneren. Eén resultaat hiervan is een andere manier voor het construeren en presenteren van categorische bewijzen: algebraïsche *berekeningen* in plaats van het in categorie-theorie gebruikelijke *diagrammen jagen* (een methode waarbij plaatjes een grote rol spelen). Een ander resultaat is een grotere systematiek in de wetten en het gebruik ervan binnen de categorie-theorie zelf. Dit staat beschreven in Hoofdstuk 2 van het proefschrift.

Wat mij persoonlijk betreft is pas door deze algebraïsche aanpak de categorie-theorie gaan leven. Niet zozeer dat ik de dingen beter *begrijp*, maar veeleer dat ik ermee kan *rekenen*. Net als met reële getallen: je kunt er mee rekenen, zonder de echte definitie ervan te begrijpen — denk aan Cauchy-rijen of Dedekind sneden. Ik denk dat deze ervaring voor veel meer mensen zal gelden.

Tot besluit

Het hier beschreven onderzoek vormt een onderdeel van een groter project. Binnen Nederland zijn er onderzoeksgroepen op het gebied van algoritmiek bezig in Amsterdam op het CWI (Meertens), in Utrecht aan de RUU (Swierstra) en in Eindhoven aan de TUE (Backhouse), en daarbuiten zijn er groepen in Oxford (Bird, de Moor) en Glasgow/Göteborg (Sheeran, Hutton). Er wordt gewerkt aan uitbreiding van de theorie met non-determinisme (zodat de uitvoer niet exact vastligt bij gegeven invoer), aan probleem-gerichte stellingen (in plaats van de datatype-gerichte wetten zoals in mijn onderzoek), en aan een specialisatie tot het "programmeren" van schakelingen. Er is nog veel te doen ...

Mijn dank gaat uit naar Klaas van den Berg en Nico van Diepen voor hun commentaar op een eerste versie van dit verhaal.