MC SYLLABUS 21

J.W. DE BAKKER (red.)

COLLOQUIUM PROGRAMMACORRECTHEID

MATHEMATISCH CENTRUM AMSTERDAM 1975

AMS(MOS) Subject classification scheme (1970): 68A05, 68A10, 68A20, 68A30

ACM -Computing Reviews- categories: 5.22, 4.22, 5.23, 5.25

ISBN 90 6196 103 3

INHOUD

MATHEMATICAL CENTRE SYLLABUS 21, 1975, I, 1-17.

INLEIDING BEWIJSMETHODEN

J.W. de BAKKER

1. INLEIDING

Wat bedoelen we als we zeggen dat een programma "correct" is? Welke technieken zijn beschikbaar om de correctheid van een gegeven programma te bewijzen, dan wel om een correct programma te construeren? Welke zijn de theoretische eigenschappen van deze technieken, en welke consequenties hebben zij voor de praktijk van het programmeren?
Met deze en dergelijke vragen zullen wij ons in dit colloquium bezighouden. Gestreefd zal worden naar een zeker evenwicht tussen meer theoretische beschouwingen enerzijds en praktische toepassingen anderzijds.
In deze eerste bijdrage wordt een literatuuroverzicht gegeven van de technieken ontwikkeld voor correctheidsbewijzen. Hiervan is de methode van de "inductieve asserties" thans de meest verbreide. Het zal blijken dat de essentie van deze techniek reeds in 1949 door TURING [27] is voorgesteld. Het formalisme van HOARE [10,11,12], hetwelk kan worden opgevat als een variant van de inductieve assertiemethode, wordt vervolgens gebruikt om een "moderne" behandeling te geven van een door TURING gegeven voorbeeld.

2. LITERATUUROVERZICHT

Vooraf zij opgemerkt dat onze bespreking van de diverse methoden in deze sectie noodzakelijkerwijze globaal blijft. De belangrijkste methoden komen echter in het vervolg van het colloquium meer in detail nogmaals aan de orde.
Reeds in een van de eerste publicaties over het gebruik van de computer, n.l. in het bekende artikel van GOLDSTINE en VON NEUMANN *Planning and coding problems for an electronic computer instrument* [8]*) komt een passage voor die de kiem voor een methodologie van programmacorrectheidsbewijzen bevat. Er is daar namelijk sprake (pag. 92) van het gebruik van "assertion boxes" in een blokdiagram op een manier die (enige) overeenstemming vertoont met het hedendaagse gebruik van inductieve asserties.
Veel duidelijker is de vraag naar verificatie van programma's gesteld door TURING in zijn artikel *On checking a large routine* [27]*). Wegens het

*) Referenties [8] en [27] zijn ontleend aan LONDON [16].

grote belang van dit artikel zullen we het in sectie 3 uitgebreid behandelen.

Na Turing's publicatie is er een aantal jaren weinig te melden, totdat rond 1960 enkele artikelen van McCARTHY verschijnen [22,23], die baanbrekend zijn geweest voor de huidige programmeertheorie. In deze publicaties is o.a. de basis gelegd voor verscheidene van de later ontwikkelde methoden voor de formele definitie van programeertalen, en voor axiomatische karakterisering van een aantal van de belangrijkste programmeerconcepten. Verder wordt een bewijsmethode voor eigenschappen van recursieve procedures geintroduceerd, evenals het begrip "toestandsvector", welk laatste fundamenteel is voor de thans gangbare opvatting van de correctheid van een programma, waarover nu iets meer.

Wat is een zinvolle opvatting van programmacorrectheid? Eerst een tot niet veel leidende beschouwing: ieder programma is per definitie correct, d.i., doet wat de programmeur ervan verwacht; immers, indien deze iets anders bedoeld had, dan had hij dat wel opgeschreven. Met deze benadering kunnen we weinig doen. Meer aanknopingspunten biedt de volgende gedachtengang: een programma heeft tot taak een probleem op te lossen, en zo'n oplossing is vrijwel steeds te karakteriseren als een transformatie van een gegeven begintoestand, gekenschetst door een aantal eigenschappen, naar een gewenste eindtoestand, evenzeer gekenschetst door een aantal eigenschappen. Zo kan dus als formulering van de correctheid van programma P dienen: Zij σ_1 de begintoestand, waarvoor de eigenschap p (meestal een aantal deeleigenschappen samenvattend) waar is. Zij $\sigma_2 = P(\sigma_1)$ de eindtoestand, bewerkstelligd door toepassing van P op σ_1. Dan voldoet σ_2 aan de eigenschap q. Ofwel:

(1) Voor alle σ_1,σ_2, als $p(\sigma_1)$ en $\sigma_2 = P(\sigma_1)$ dan $q(\sigma_2)$.

Deze definitie biedt enerzijds voldoende vrijheid, omdat zij nadere verfijning toelaat: bij verdere analyse zullen P, p en q in componenten moeten worden gesplitst, en de bewering (1) dienovereenkomstig in een aantal deelbeweringen ontleed. Anderzijds, is zij voldoende ruim om als kader te dienen voor veel van de voorgestelde concrete systemen, waarover nu verder.

De volgende belangrijke stap, na McCarthy's artikelen, is de methode van FLOYD [7], nu meestal aangeduid als de inductieve assertiemethode. Van zijn

artikel zijn vooral de eerste en laatste bladzijden van betekenis: De eerste pagina introduceert de methode, die overigens ongeveer in dezelfde tijd werd gepubliceerd door NAUR, en, zoals gezegd, in feite terug gaat tot TURING. Zij behelst het volgende: Stel we willen een bewering van het type (1) bewijzen over een programma P, waarvan wij nu gemakshalve aannemen dat het de vorm van een blokdiagram heeft. We introduceren daartoe naast de genoemde begin- en eindbeweringen p en q ook een aantal -geschikt gekozen, en voor deze keuze is i.h.a. een grondig inzicht in de werking van het programma vereist- tussenbeweringen $(p=p_0,)p_1,p_2,\ldots,p_{n-1}(,p_n=q)$, die we laten corresponderen met plaatsen in het blokdiagram π_0(start box),π_1, $\ldots,\pi_{n-1},\pi_n$(halt box). We moeten hierover dan het volgende bewijzen: Voor iedere i, i=0,...,n-1, als P_i het programmadeel is, doorlopen tussen plaats π_i en π_{i+1}, dan:

(1_i) Voor alle σ_1,σ_2, als $p_i(\sigma_1)$ en $\sigma_2 = P_i(\sigma_1)$, dan $p_{i+1}(\sigma_2)$.

Volgens de inductieve assertiemethode kan nu uit het bewijs van (1_i), i=0,...,n-1, tot dat van (1) geconcludeerd worden. Voor programma's zonder loops is de geldigheid van deze methode geen probleem; zijn er wel loops, dan moet een inductief argument worden toegepast naar het aantal malen dat de loops in kwestie worden doorlopen. Een precieze formulering en bewijs hiervan voert ons thans te ver; wellicht komen we er later in het colloquium nog op terug. Opgemerkt zij nog dat voor het bewijs van de beweringen (1_i) het natuurlijk nodig is om het effect van de deelprogramma's P_i te kennen; m.a.w., bij deze bewijzen zal de semantiek van de taal gebruikt om P_i te schrijven, een rol spelen. Anderzijds kan, zo men wil, het postuleren van geschikt gekozen beweringen (1_i) als middel van semantiekdefinitie gezien worden.
Aan het eind van Floyd's artikel komt de vraag naar *terminatie* van een programma aan de orde. De lezer heeft wellicht opgemerkt dat een bewering als (1) hierover geen uitspraak doet, omdat slechts iets wordt gezegd, *als* er een σ_2 is met $\sigma_2 = P(\sigma_1)$. Nodig is dus, in voorkomende gevallen, een apart bewijs van het bestaan van zo'n σ_2, indien nodig beperkt tot gevallen waarin $p(\sigma_1)$ waar is. Als methode voor zo'n bewijs stelt FLOYD voor: Associeer met de plaatsen π_i in het programma, behalve de p_i ook functies $\phi_i(\sigma)$, met

waarden in een wel-geordende verzameling W met welordening zeg "<". Bewijs nu:

Voor alle $\sigma_1,\sigma_2,w_1,w_2 \in W$,

als $p_i(\sigma_1)$ en $\phi_i(\sigma_1) = w_1$ en $\sigma_2 = P_i(\sigma_1)$ en $\phi_{i+1}(\sigma_2) = w_2$, dan $w_2 < w_1$.

De geldigheid van de methode kan weer worden gerechtvaardigd met een inductief element naar het aantal malen dat de loops in het programma worden doorlopen.

De ideeën van FLOYD zijn in verschillende richtingen uitgewerkt. In een reeks artikelen heeft MANNA laten zien hoe inductieve asserties behandeld kunnen worden binnen de eerste en tweede orde predicatenlogica (bv. in [17,18]). Een belangrijke bewering is, ruwweg, dat een programma dan en slechts dan "partieel correct" is (overeenstemmend met uitspraak (1)) als dit via de inductieve assertiemethode bewijsbaar is. Hierbij valt wel op te merken dat een grondiger bewijs van deze bewering dan door MANNA gegeven, ons wenselijk voorkomt. Verder introduceert MANNA naast partiële, ook totale correctheid, te formuleren als: P is totaal correct m.b.t. p en q, als

Voor alle σ_1: Als $p(\sigma_1)$ dan bestaat er een σ_2, met $\sigma_2=P(\sigma_1)$ en $q(\sigma_2)$.

Een methode wordt vervolgens aangegeven om bewijzen van totale correctheid tot die van partiële correctheid te herleiden. Hiertoe dient wel eerst een verfijnde versie van de formulering van partiële correctheid als in (1), te worden gegeven.

Een groot aantal toepassingen van Floyd's methode, ingekleed in een eigen formalisme, is gegeven door HOARE (bv. [10,11]). We komen hierop terug in sectie 3.

Andere voorbeelden zijn te vinden bij LONDON (bv. [15]). We vestigen verder in het bijzonder de aandacht op London's recente overzichtsartikel *The current state of proving programs correct* [16], waarin naast een opsomming van allerlei programma's bewezen met de assertiemethode, ook melding wordt gemaakt van het groeiend aantal interactieve "programmabewijssystemen", waarbij de computer wordt ingeschakeld als hulpmiddel bij correctheidsbewijzen. Een van de eerste dergelijke systemen was de *Verifying Compiler* van KING [13]. Het ligt in de bedoeling in een aparte bijdrage de methoden van computerhulp bij bewijzen te bespreken.

In het werk van DIJKSTRA [4] ligt de nadruk niet zozeer op het bewijzen van de correctheid van een gegeven programma, maar op een methodologie voor het ontwerpen van correct werkende programma's. Bij zijn "*top-down*" methode voor programmaontwerp speelt bij de argumentatie van de correctheid van iedere volgende verfijningsstap in het bijzonder Hoare's formalisme een rol.

Voor het bewijzen van eigenschappen van recursieve procedures was sinds McCARTHY [22] de methode der recursie-inductie beschikbaar. De inductieve assertiemethode is hier evenzeer toepasbaar gebleken, maar de precieze status van de methode, wanneer toegepast bij recursie, bleef onduidelijk -getuige bv. het ontbreken van een bevredigende behandeling bij HOARE of DIJKSTRA- totdat SCOTT via de kleinste dekpunt karakterizering van recursie (teruggaand tot KLEENE [14], p.348) zijn inductieregel afleidde (voor het eerst gepubliceerd in SCOTT & DE BAKKER [26]), die een zeer belangrijk hulpmiddel is gebleken bij het bewijzen van *eigenschappen van recursie* (zie bv. [2,3,19,20]). Hierover later zeker meer.

We vermelden aan het eind van dit overzicht nog enkele andere publicaties ter verdere introductie in de lectuur over programmacorrectheid. Het overzichtsartikel van ELSPAS e.a. [5], waarin naast een uitgebreide behandeling van de eerste periode in Manna's werk (voordat Scott's inductieregel een rol ging spelen) ook aandacht wordt geschonken aan de toepasbaarheid van theorem-proving technieken op verificatie van asserties. Verder de Proceedings van de *ACM Conference on Proving Assertions about Programs* McCARTHY [24], waarin een aantal recentere ontwikkelingen beschreven is (voor het bestuderen waarvan i.h.a. wel enige inleidende kennis van zaken nodig is). Recentelijk verscheen een boek over *Program Test Methods* HETZEL (ed.) [9] waarde vooral gelegen is in de vele -maar nogal lukraak samengebrachte- literatuuropgaven.

Voor wat meer algemene literatuur over programmeertheorie en semantiek, waarbij programmacorrectheid als onderdeel aan de orde komt, verwijzen we naar het overzichtsartikel *Semantics of Programming Languages* DE BAKKER [1], waarin de stand van zaken tot 1968 is beschreven, met ook veel gegevens over formele definitie van programmeertalen; het boek *Symposium on the Semantics of Algorithmic Languages*, onder redactie van ENGELER [6], en *Formal Semantics of Programming Languages*, redactie RUSTIN [25]. Aangekondigd is ook de verschijning van een boek door MANNA: *Introduction to the Mathematical Theory of Computation*, bij McGraw-Hill.

3. TURING EN HOARE

3.1. *Hoare's formalisme*

De inductieve assertiemethode werd door FLOYD geïntroduceerd aan de hand van programma's in blokschemavorm. Het is de verdienste van HOARE geweest om de methode aan te passen voor toepassing op ALGOL-achtige programma's. Hij bedient zich van een aantal regels die hij "axioma's" noemt, waarvan hier die voor *assignment, compositie, conditionele statements en while statements*, behandeld worden.
Hoare's regels zijn speciale gevallen van uitspraken als in (1) gegeven

$$\text{Voor alle } \sigma_1,\sigma_2, \text{ als } p(\sigma_1) \text{ en } \sigma_2 = P(\sigma_1), \text{ dan } q(\sigma_2)$$

waarvoor hij schrijft $\{p\}P\{q\}$.
Om de vorm die deze regel aanneemt als P een eenvoudige assignment statement is, te verklaren bekijken we wat verder wat een "toestand" σ, zoals getransformeerd door P, eigenlijk is. Het is voldoende deze te zien als een "momentopname" van het geheugen, preciezer als de afbeelding die -op een gegeven moment- bestaat tussen adressen in het geheugen en bijhorende waarden. Als we de in het programma voorkomende variabelen, zeg $x_1,x_2,\ldots,x_n$ identificeren met de ermee geassocieerde adressen (deze simplificatie is voor ons doel hier niet bezwaarlijk), dan is een "toestand" een uitdrukking van de vorm $\begin{pmatrix} x_1 & x_2 & \cdots & x_n \\ a_1 & a_2 & \cdots & a_n \end{pmatrix}$, waarbij $a_1,a_2,\ldots,a_n$ de momentane waarden van de variabelen $x_1,x_2,\ldots,x_n$ voorstellen. Toepassing van de assignment statement $x_i := f(x_1,x_2,\ldots,x_n)$, waarbij voor f iedere voorkomende functie (optelling, sinus, of wat men maar wil) op $\sigma_1 = \begin{pmatrix} x_1 & x_2 & \cdots & x_n \\ a_1 & a_2 & \cdots & a_n \end{pmatrix}$ levert voor σ_2 op: $\sigma_2 = \begin{pmatrix} x_1 & \cdots & x_i & \cdots & x_n \\ a_1 & \cdots & f(a_1,\ldots,a_n) & \cdots & a_n \end{pmatrix}$.
We passen dit nu als volgt toe: Zij, in verkorte notatie, $\sigma_1 = \begin{pmatrix} x \\ a \end{pmatrix}$, $\sigma_2 = \begin{pmatrix} x \\ f(a) \end{pmatrix}$ als gevolg van $x := f(x)$, en stel $p(\sigma_2)$, d.w.z. p(x) met x refererend aan f(a), geldt na afloop van de assignment. Dan geldt dus p(f(a)), en dit was evenzeer waar *voor* de assignment, die a niet veranderd heeft. Daaruit volgt dat voor de assignment p(f(x)) ($\Longleftrightarrow$ p(f(a)) gold. Hiermee hebben we Hoare's axioma

H_a : $\{p(f(x))\}\ \ x := f(x)\{p(x)\}$

verklaard. Specialisatie van p en f geeft bijvoorbeeld
$\{x+1 > 0\}\ \ x := x+1\{x > 0\}$, of $\{3 = 3 \wedge y > 5\}\ \ x := 3\{x=3 \wedge y > 5\}$. Het tweede voorbeeld geeft aan hoe we formeel tot "x=3" na afloop van x := 3 kunnen concluderen.
Opgemerkt zij nog dat H_a eigenlijk een precieze omschrijving van de vormen die p kan aannemen, nodig maakt, omdat pas dan het substitutieproces voldoende streng gedefinieerd kan worden. Van de gevallen waartoe wij ons in deze inleiding beperken kan dergelijke precizering wel achterwege blijven.
Hoare's compositie axioma is direct duidelijk uit de interpretatie van $\{p\}P\{q\}$ als (1):

H_c: Als $\{p\}P_1\{q\}$ en $\{q\}P_2\{r\}$, dan $\{p\}P_1;P_2\{r\}$.

Als axioma voor de conditionele statement hebben we

H_s: Als $\{p \wedge q\}S_1\{r\}$ en $\{p \wedge \neg q\}S_2\{r\}$, dan
$\{p\}$ <u>if</u> q <u>then</u> S_1 <u>else</u> $S_2\{r\}$

welke regel voor zichzelf spreekt.
Wat meer toelichting heeft het while statement axioma nodig. Een while statement heeft de vorm <u>while</u> p <u>do</u> S, en betekent, zoals bekend: Voer S uit zolang p waar is (inclusief het geval dat p aan het begin reeds onwaar is, in welk geval <u>while</u> p <u>do</u> S gelijkwaardig is met de dummy statement).Duidelijk is dat na afloop van de while statement in ieder geval p onwaar is. Verder is ook in te zien dat indien voor S geldt dat deze een bepaalde eigenschap q invariant laat ($\{q\}S\{q\}$ geldt), dit ook voor <u>while</u> p <u>do</u> S geldt. Zelfs geldt een sterkere bewering:

H_w: Als $\{p \wedge q\}S\{q\}$, dan $\{q\}$ <u>while</u> p <u>do</u> $S\{\neg p \wedge q\}$.

Immers, in while p do S vindt iteratie van S slechts plaats onder de voorwaarde dat p waar is, en het invariant laten van q door S hoeft dan ook slechts onder die voorwaarde te worden aangenomen.
(Vooruitlopend op een verwachte latere bijdrage kunnen wij niet nalaten te vermelden dat H_w een deel is van de algemenere bewering: {q} while p do S{r} als *en slechts als* er een s bestaat met $q \supset s$, $\{p \wedge s\}S\{s\}$, en $\{\neg p \wedge s\}I\{r\}$, met I de dummy statement. Deze bewering is een speciaal geval van Manna's hoofdstelling over partiële correctheid, en tevens van een -aanzienlijk gecompliceerdere- bewering over recursieve procedures. Voor de liefhebbers voegen we hier nog aan toe dat zelfs het volgende geldt: Zij p,S willekeurig gegeven. Zij T een statement die voldoet aan: Voor alle q,r; $\{q\}T\{r\}$ als en slechts als $q \supset s$, $\{p \wedge s\}S\{s\}$, $\{\neg p \wedge s\}I\{r\}$. Dan valt T noodzakelijkerwijze samen met while p do S. M.a.w., met behulp van inductieve asserties kan een while statement volledig worden gekarakterizeerd; hiermede is dan een alternatief gevonden voor de, in een latere bijdrage te behandelen, kleinste dekpunt karakterizering.)
We geven nu een voorbeeld van toepassing van H_a, H_c en H_w, ontleend aan MANNA [21]: Beschouw het programma P:

$(y_1,y_2,y_3) := (0,1,1);$
while $y_2 \leq x$ do $(y_1,y_2,y_3) := (y_1+1,y_2+y_3+2,y_3+2);$
$z := y_1$

waarin een voor zichzelf sprekende "parallele" assignmentnotatie is toegepast. We zullen bewijzen dat $\{p\}P\{q\}$, met voor p:

$$p(\sigma) = p(x,y_1,y_2,y_3,z) : x \geq 0, \quad \text{en}$$

$$q(\sigma) = q(x,y_1,y_2,y_3,z) : z^2 \leq x < (z+1)^2.$$

We gebruiken als "tussen"-bewering -het vinden waarvan overigens in het algemeen de kern van het inductief assertiebewijs uitmaakt-

$r(\sigma) = r(x,y_1,y_2,y_3,z) : y_1^2 \leq x \wedge y_2 = (y_1+1)^2 \wedge y_3 = 2y_1 + 1$ (omdat r niet van z afhangt wordt deze parameter verder weggelaten). Duidelijk is dat $x \geq 0 \supset r(x,0,1,1)$. Toepassing van H_a geeft dan

(2) $\quad \{x \geq 0\}\ (y_1,y_2,y_3) := (0,1,1)\ \{r(x,y_1,y_2,y_3)\}.$

Omdat $r(x,y_1,y_2,y_3) \wedge y_2 \leq x \supset r(x,y_1+1,y_2+y_3+2,y_3+2)$ krijgen we, wederom H_a toepassend,

$$r(x,y_1,y_2,y_3) \wedge y_2 \leq x\}\ (y_1,y_2,y_3) := (y_1+1,y_2+y_3+2,y_3+2)\ \{r(x,y_1,y_2,y_3)\}.$$

We kunnen nu H_w toepassen en krijgen

(3) $\{r(x,y_1,y_2,y_3)\}$ <u>while</u> $y_2 \leq x$ <u>do</u> $(y_1,y_2,y_3) := (y_1+1,y_2+y_3+2,y_3+2)$

$$\{r(x,y_1,y_2,y_3) \wedge y_2 > x\}.$$

Toepassing van de compositieregel op (2) en (3) geeft

(4) $\{x \geq 0\}\ (y_1,y_2,y_3) := 0;$

<u>while</u> $y_2 \leq x$ <u>do</u> $(y_1,y_2,y_3) := (y_1+1,y_2+y_3+2,y_3+2)$

$$\{r(x,y_1,y_2,y_3) \wedge y_2 > x\}.$$

Omdat $r(x,y_1,y_2,y_3) \wedge y_2 > x \supset y_1^2 \leq x < (y_1+1)^2$ krijgen we door hernieuwd toepassen van de assignmentregel:

(5) $\quad \{r(x,y_1,y_2,y_3) \wedge y_2 > x\}\ z := y_1\{z^2 \leq x < (z+1)^2\}.$

Compositie van (4) en (5) levert tenslotte

(6) $\quad \{x \geq 0\}P\{z^2 \leq x < (z+1)^2\}$

hetgeen te bewijzen was.

Dit voorbeeld geeft aanleiding tot een tweetal belangrijke opmerkingen:

1. Het moeilijkste gedeelte van het bewijs was kennelijk het vinden van de bewering $r(x,y_1,y_2,y_3)$. Is dit eenmaal gelukt, dan is het duidelijk dat:
2. Bij het formeel manipuleren met de asserties op basis van H_a, H_c, H_w etc. kan de computer nuttig werk doen.

Op opmerking 2 zijn diverse interactieve programmabewijssystemen gebaseerd.

3.2. *Turing's voorbeeld*

In een artikel verschenen in het verslag van de eerste in Engeland gehouden computerconferentie: *Report on a Conference on High Speed Automatic Calculating Machines* [27], stelt TURING een bewijsmethode voor programmacorrectheid voor, die de essentie van de inductieve assertiemethode bevat. We zullen dit niet voor iedereen makkelijk vindbare artikel hier nogal uitgebreid citeren. TURING begint met

> "How can one check a large routine in the sense of making sure that it is right? In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows.
> ...
> This principle can be applied to the process of checking a large routine but we will illustrate the method by means of a small routine, viz. one to obtain n! without the use of a multiplier, multiplication being carried out by repeated addition."

Het te bewijzen programma wordt beschreven met het volgende blokdiagram (Turing's weergave hiervan zo getrouw mogelijk volgend):

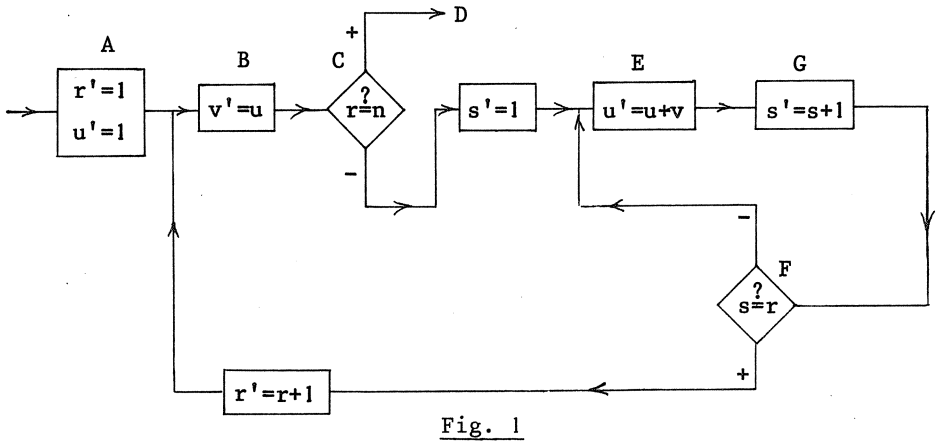


Fig. 1

Hierbij zij opgemerkt dat

1. Met r' = 1, u' = 1 etc. assignment statements bedoeld zijn.
2. Figuur 1 (en de dadelijk volgende table 1) een aantal onnauwkeurigheden bevat die in onze versie in sectie 3.3 hersteld zijn. Zo moet s $\stackrel{?}{=}$ r worden s $\stackrel{?}{=}$ r+1.

TURING vervolgt nu

"In order to assist the checker, the programmer should make assertions about the various states that the machine can reach. These assertions may be tabulated as in table 1.

STORAGE LOCATION	A k=6	B k=5	C k=4	D k=0	E k=3	F k=1	G k=2
27 (s)					s	s+1	s
28 (r)		r	r		r	r	r
29 (n)	n	n	n	n	r	n	n
30 (u)		r!	r!		s·r!	(s+1)·r!	(s+1)·r!
31 (v)			r!	n!	r!	r!	r!
	to B with r'=1 s'=1	to C	to D if r=n to E if r<n		to G	to B with r'=r+1 if s≠r; to E with s'=s+1 if s<r	to F

Table 1

"Assertions are only made for the states when certain particular quantities are in control, corresponding to the capital letters in the flow diagram. One column of the table is used for each such situation of the control. Other quantities are also needed to specify the condition of the machine completely: In our case it is sufficient to give r and s. The upper part of the table gives the various contents of the store lines in the various conditions of the machine, and restrictions on the quantities s,r (which we may call inductive variables). The lower part tells us which of the conditions will be the next to occur.
The checker has to verify that the columns corresponding to the initial condition and the stopped condition agree with the claims that are made for the routine as a whole. In this case the claim is that if we start with control in condition A and with n in line 29, we shall find a quantity in line 31 when the machine stops which is n! (provided this is less than 2^{40}, but this condition has been ignored.)
He has also to verify that each of the assertions in the lower half of the table is correct. In doing this the columns may be taken in any order and quite independently. Thus for column B the checker would argue: From the flow diagram we see that after B the box v' = u applies. From the upper part of the column for B we have u = r'. Hence v' = r, i.e. the entry for v, i.e. for line 31, in C should be r. The other entries are then the same as in B.
..."

TURING geeft tenslotte nog aan hoe terminatie te bewijzen: Zij w een voldoend groot getal (bv. $w = 2^{40}$). Dan zal iedere stap van het programma het getal $(n-r)w^2 + (r-s)w + k$ met tenminste één verkleinen, hetgeen terminatie van het programma impliceert.
Zo zien we uit deze citaten hoe TURING, die met behulp van de naar hem genoemde machines reeds in 1936 de grenzen van het begrip "berekenbaar" afbakende nog voor er sprake was van (niet-menselijke) computers, ook een be-

wijsmethode voor correctheid heeft voorgesteld waarvan het belang pas de laatste jaren is ingezien.

3.3. *Turing's voorbeeld in Hoare's formalisme*

We geven een ALGOL-achtige versie van figuur 1, waarbij we, naast de while statement, ook de "*repeat statement*" repeat S until p gebruiken, die gedefinieerd kan worden door

repeat S until p = S; while ¬p do S

We krijgen dan van figuur 1

```
S0 : begin r := 1; u := 1; v := u;
     S1 : while r < n do
          S2 : begin s := 1
               S3 : repeat
                    S4 : begin u := u+v; s := s+1 end
                         until s > r;
                    r := r+1; v := u
               end
     end
```

Merk op dat eigenlijk een bewijs gegeven zou moeten worden dat dit programma equivalent is met (een geamendeerde versie van) het blokschema. Een precies bewijs hiervan zou een formele behandeling van de executie van een blokschema nodig maken, wat een enigszins moeizame en voor ons doel niet speciaal vruchtbare aangelegenheid is, reden waarom dit achterwege blijft.
We zullen bewijzen: $\{n \geq 1\}S_0\{v = n!\}$
De eerste stappen van het bewijs gaan als volgt:

(1)	$\{1 \leq n,\ 1 = 1\}$	, aanname
	r := 1;	(+ tautologie)
(2)	$\{r \leq n,\ r = 1\}$	, (1), H_a
(3)	$\{r \leq n,\ r = 1,\ 1 = 1\}$	, (2)
	u := 1	
(4)	$\{r \leq n,\ r = 1,\ u = 1,\ 1 = 1\}$	, (3), H_a
	v := u	

(5) $\{r \leq n,\ r = 1,\ u = 1,\ v = 1\}$, (4), H_a

(6) $\{1 \leq r \leq n,\ u = r!,\ v = r!\}$, (5), def.!,eig.≤

(een volledige formalisering, zoals bv. voor computerbehandeling nodig, zou ook de te gebruiken eigenschappen van ≤ moeten weergeven.)
We passen nu op (6) en de while statement S_1 het axioma H_w toe. Daartoe moeten we bewijzen dat

(7) $\{\overbrace{r < n}^{q},\ \overbrace{1 \leq r \leq n,\ u = r!,\ v = r!}^{p}\}\ S_1\ \{\overbrace{1 \leq r \leq n,\ u = r!,\ v = r!}^{p}\}$

om dan te kunnen concluderen dat na afloop van S_1 zowel $r \geq n$ ($\neg q$) als $\{1 \leq r \leq n,\ u = r!,\ v = r!\}$ waar zijn. Is (7) bewezen, dan is het gezochte resultaat inderdaad verkregen; immers, $\{r \geq n,\ 1 \leq r \leq n,\ u = r!,\ v = r!\}$ geeft $\{r = n,\ v = r!\}$ waaruit $\{v = n!\}$ volgt. Rest dus het bewijs van (7).

(8) $\{r < n,\ 1 \leq r \leq n,\ u = r!,\ v = r!\}$, aanname

(9) $\{1 \leq 1 \leq r+1 \leq n,\ v = r!,\ u = 1 \times r!\}$, (8)

$s := 1;$

(10) $\{1 \leq s \leq r+1 \leq n,\ v = r!,\ u = s \times r!\}$, (9), H_a

We stuiten nu op de repeat statement S_3, waarvoor we nog geen axioma geformuleerd hebben. We hebben hier voldoende aan

H_r : Als $\{\neg p \wedge q\}S\{q\}$, dan

$\{\neg p \wedge q\}$ <u>repeat</u> S <u>until</u> $p\{p \wedge q\}$.

Passen we dit toe met voor p : $s > r$, en voor q : (10), dan is het voldoende te bewijzen:

(11) $\{s \leq r,\ 1 \leq s \leq r+1 \leq n,\ v = r!,\ u = s \times r!\}\ S_4\ \{1 \leq s \leq r+1 \leq n,\ v = r!,\ u = s \times r!\}$

waarvan het bewijs als volgt gaat

(12) $\{s \leq r,\ 1 \leq s \leq r+1 \leq n,\ v = r!,\ u = s \times r!\}$, aanname

(13) $\{1 \leq s+1 \leq r+1 \leq n,\ v = r!,\ u = s \times r!\}$, (12)

$u := u+v;$

(14) $\{1 \leq s+1 \leq r+1 \leq n,\ v = r!,\ u = (s+1) \times r!\}$, (13), H_a

$s := s+1;$

(15) $\{1 \leq s \leq r+1 \leq n,\ v = r!,\ u = s \times r!\}$, (14), H_a

Uit (12) t/m (15) en H_r volgt dan dat

$$\{1 \leq s \leq r+1 \leq n,\ v = r!,\ u = s \times r!\}\ S_3\ \{s > r,\ 1 \leq s \leq r+1 \leq n,\ v = r!,\ u = s \times r!\}$$

waaruit m.b.v. (10) volgt

(16) $\{s > r,\ 1 \leq s \leq r+1,\ v = r!,\ u = s \times r!\}$, (10), (11), H_r

(17) $\{1 \leq s = r+1 \leq n,\ v = r!,\ u = s \times r!\}$, (16)

(18) $\{1 \leq r+1 \leq n,\ v = r!,\ u = (r+1)!\}$, (17)

$r := r+1;$

(19) $\{1 \leq r \leq n,\ v = (r-1)!,\ u = r!\}$, (18), H_a

$v := u$

(20) $\{1 \leq r \leq n,\ v = r!,\ u = r!\}$, (19), H_a

en we zien dat het bewijs van (7), en hiermede van Turing's voorbeeld, voltooid is.

Een tweetal opmerkingen tot slot van onze behandeling van Hoare's formalisme:

1. In samenwerking met WIRTH heeft HOARE een axiomatische karakterisering gegeven van de programmeertaal PASCAL [12].
2. Mocht de lezer wellicht al te zeer onder de indruk zijn geraakt van de bewijskracht van Hoare's systeem, dan nodigen we hem uit het bewijs van equivalenties zoals

   while p **do** S = **if** p **then begin** S; **while** p **do** S **end**
   while p **do while** p **do** S = **while** p **do** S

   of, iets ingewikkelder,

- repeat S until $(p_1 \vee p_2)$ =
  repeat (repeat S until p_1) until p_2

op Hoare's axioma's te baseren. Hij zal daar niet in slagen.

LITERATUUR

[1] BAKKER, J.W. DE, *Semantics of programming languages*, in *Advances in information systems science, Vol. II*, J.T. TOU (ed.), Plenum Press, New York, 1965, pp.173-227.

[2] BAKKER, J.W. DE, *Recursive procedures*, Mathematical Centre Tracts 24, Mathematisch Centrum, Amsterdam, 1971.

[3] BAKKER, J.W. DE & W.P. DE ROEVER, *A calculus for recursive program schemes*, in *Automata, languages and programming*, M. NIVAT (ed.), North Holland Publ. Cy., Amsterdam, 1973, pp.167-156.

[4] DIJKSTRA, E.W., *Notes on structured programming*, in *Structured programming*, DAHL, E.W. DIJKSTRA & C.A.R. HOARE, Academic Press, New York, 1972, pp.1-82.

[5] ELSPAS, B., K.N. LEVITT , R.J. WALDINGER & A. WAKSMAN, An assessment of techniques for proving program correctness, *Comput Surveys*, 4 (1972) pp.97-147.

[6] ENGELER, E. (ed.), *Symposium on the semantics of algorithmic languages*, Lecture Notes in Mathematics, Vol. 188, Springer Verlag, Berlin etc., 1971.

[7] FLOYD, R.W., *Assigning meanings to programs*, in Proc. of a Symposium in Applied Mathematics, Vol. 19: *Mathematical aspects of computer science*, J.T. SCHWARTZ (ed.), 1967, pp.19-32.

[8] GOLDSTINE, H.H. & J. VON NEUMANN, *Planning and coding problems for an electronic computer instrument*, in *John von Neumann collected works, vol. 5*, A.M. TAUB (ed.), Pergamon Press, 1963, pp.80-235.

[9] HETZEL, W.C. (ed.), *Program test methods*, Prentice Hall, 1973.

[10] HOARE, C.A.R., An axiomatic basis for computer programming, *Comm. ACM*, 12 (1969) pp.576-580.

[11] HOARE, C.A.R., Proof of a program: FIND, *Comm. ACM*, 14 (1971) pp.39-45.

[12] HOARE, C.A.R. & N. WIRTH, *An axiomatic definition of the programming language Pascal*, E.T.H. Zürich Berichte, 1972.

[13] KING, J., *A program verifier*, thesis, Carnegie-Mellon University, 1969.

[14] KLEENE, S.C. *Introduction to Metamathematics*, North-Holland Publ. Cy., Amsterdam, 1952.

[15] LONDON, R.L., Proof of algorithms: A new kind of certification (Certification of Algorithm 245 Treesort 3), *Comm. ACM*, 13 (1970) pp.371-373.

[16] LONDON, R.L., *The current state of proving programs correct*, Proc. ACM 25th National Conference,

[17] MANNA, Z., Properties of programs and the first order predicate calculus, *J. Assoc. Comput. Mach.*, 16 (1969) pp.244-255.

[18] MANNA, Z., The correctness of programs, *J. Comput. System, Sci.*, 3 (1969) pp.119-127.

[19] MANNA, Z., S. NESS & J. VUILLEMIN, Inductive methods for proving properties of programs, *Comm. ACM*, 16 (1973) pp.491-502.

[20] MANNA, Z. & J. VUILLEMIN, Fixpoint approach to the theory of computation, *Comm. ACM*, 15 (1973) pp.528-536.

[21] MANNA, Z., *Introduction to the mathematical theory of computation*, McGraw-Hill, to appear.

[22] McCARTHY, J., *A basis for a mathematical theory of computation*, in *Computer programming and formal systems*, P. BRAFFORT & D. HIRSCHBERG (eds.), North-Holland Publ. Cy., Amsterdam, 1963, pp.33-69.

[23] McCARTHY, J., *Towards a mathematical science of computation*, in *Proc. IFIP Congress 1962*, C.M. POPPLEWELL (ed.), North-Holland Publ. Cy., 1963, pp.21-28.

[24] McCARTHY, J., Proc. ACM Conference on Proving Assertions about Programs, *SIGPLAN Notices*, 7 (1972), no. 1 (January).

[25] RUSTIN, R. (ed.), *Formal semantics of programming languages*, Prentice-Hall, 1972.

[26] SCOTT, D. & J.W. DE BAKKER, *A theory of programs*, Notes of an IBM-Vienna Seminar, unpublished, 1969.

[27] TURING, A.M., *On checking a large routine*, in *Report of a Conference on high-speed automatic calculating machines*, University Mathematical Laboratory, Cambridge, 1949, pp.67-69.

MATHEMATICAL CENTRE SYLLABUS 21, 1975, II, 19-46.

OVER GOTO'S EN PROGRAMMACORRECTHEID

R.P. van de RIET
Vrije Universiteit, Amsterdam

1. INLEIDING

In een colloquium over programmacorrectheid mag een discussie over *goto-statements*, ook wel *sprongen* of *jumps* genoemd, maar door ons verder aangeduid met "*goto's*", niet ontbreken.
Zowel theoretische als praktische aspecten zullen worden belicht, met een nadruk op de laatste.
We zullen zien dat men enerzijds goto's niet kan missen, dat ze anderzijds echter overbodig zijn.
De conclusie zal luiden dat er in hogere programmeertalen geen plaats is voor goto's, mits deze zijn voorzien van meer adequate middelen om besturing van programma's te beschrijven, zoals conditionele statements, repetitie statements met meervoudige "exit" mogelijkheden, recursie en blockstructuur.

2. GOTO'S

Bij de eerste automatische rekenmachine, rekenautomaat, kwamen geen goto's voor; wel voorzag de ontwerper CHARLES BABBAGE deze machine, die hij "analyse machine" noemde, van de volgende mogelijkheden om de loop van de berekeningen op dynamische wijze te regelen [2]:

a) de machine kon de stapel ponskaarten, waarop het programma was vastgelegd, herhaald uitvoeren,
b) er was een hefboom, bestuurd door het al of niet omslaan van het teken van een variabele in het geheugen, die ervoor kon zorgen dat de machine de uitvoering van de ene stapel ponskaarten stopte en overging op de uitvoering van een andere stapel ponskaarten.

BABBAGE had hiermee de "do-loop" en de "conditionele statement" uitgevonden.
De uitvinding van de goto kan dus niet op naam van BABBAGE geschreven wor-

den. Zij moet vermoedelijk meer in de richting van TURING gezocht worden die als instructies van zijn machine goto's met zij-effecten kiest. Elke Turing-machine instructie bepaalt immers de nieuwe status van de machine. Moderne electronische computers zijn, als afstammelingen van de VON NEUMAN computer, alle voorzien van goto's en wel in diverse soorten. Ook hier wordt met een goto expliciet de toestand van de machine veranderd door de instructieteller abrupt een andere waarde te geven.
In de hogere programmeertalen zijn goto's overgenomen; in het bijzonder moet hier FORTRAN genoemd worden met goto's, computed goto's en if statements. In dit verband spreekt LEAVENWORTH [12] van een "FORTRAN II IF SYNDROM".

De uitvoering van een programma zonder goto's is eenvoudig gedefinieerd als het uitvoeren van de instructies te beginnen met de eerste, vervolgens de tweede, tot de laatste.
Voor vrijwel alle algoritmen is deze gang van zaken te eenvoudig. Het is bijna altijd nodig een herhaling aan te kunnen geven of om aan te kunnen geven dat een bepaalde reeks van instructies onder zekere omstandigheden overgeslagen moet worden.
De doeltreffende oplossing was de goto instructie, die namelijk tot effect heeft dat een willekeurige opvolgende instructie wordt aangewezen. De herhaling en het conditioneel stellen van uitvoering van een serie instructies is namelijk met goto's te realiseren niet alleen theoretisch maar ook in de praktijk.
LEAVENWORTH stelt: "The FORTRAN IF SYNDROM - IF (expr) n1,n2,n3 - is a prime example of the power of language to influence program organization, and probably corrupted a generation of programmers".

Het was E.W. DIJKSTRA die, in een geruchtmakende "letter to the editor" in de *Communications of the ACM* van maart 1968 [4], op de kwalijke gevolgen van het gebruik van goto's wijst. Zijn betoog, dat een ieder beter zelf kan lezen, komt ongeveer op het volgende neer: programmeurs schrijven programma's die processen beschrijven uitgevoerd door de computer. Het gaat erom dat het proces correct verloopt, daartoe is nodig dat de programmeur zich een goed beeld vormt van dat proces, daarvoor moet zijn programma hulp bieden. Hoe meer het programma (structuur, symbolen, namen) hem een idee geeft over het proces des te meer hulp wordt geboden. Uiteraard speelt daarbij de taal waarin het programma geschreven is een essentiële rol.

Stel, het programma is slechts een rij eenvoudige instructies (bijvoorbeeld assignment statements), dan is de invoering van een virtuele *tekstwijzer* voldoende voor de vertaling van het programma naar het proces. (Men kan met de vinger aanwijzen waar het proces "zit".)
Invoering van conditionele statements (if B then A1 else A2) of van keuze statements (case i of (A1,A2,...,An)), brengt hierin geen verandering: nog steeds kunnen we met één vinger aanwijzen waar het proces "zit". Bij invoering van repetities (zoals while B repeat A) wordt het nodig bij te houden in een *teller* hoe vaak de repetitie reeds heeft plaats gevonden. Omdat repetities genest voorkomen moeten we i.h.a. meerdere tellers bijhouden, echter niet meer dan er geneste repetities voorkomen. Dit aantal ligt dus door de vorm van het programma vast.
Laten we bovendien procedures, ook in recursieve vorm, toe, dan is het nodig meerdere tekstwijzers te gebruiken om aan te geven waar het proces "zit".
Tekstwijzers en tellers zijn nodig en voldoende om zich een beeld te vormen van de loop van het proces; het zijn als het ware onafhankelijke coördinaten waarin het proces zich laat beschrijven zonder dat er nog enige sprake is van een machine.
Hoe anders wordt de situatie wanneer we ook goto's invoeren; immers de goto wijst een willekeurige opvolger aan, geheel afhankelijk van de loop van het proces.
Het is niet meer mogelijk met tekstwijzers en tellers te volstaan, het is daarentegen noodzakelijk om er een *instructieteller* bij te halen die bijvoorbeeld het aantal instructies telt sinds de start van het proces. Voor de begrijpelijkheid van het proces heeft deze instructieteller echter geen enkele waarde; er is immers geen duidelijke relatie tussen de waarde van de teller en de plaats waar het proces "zit".

Dit pleidooi van DIJKSTRA tegen goto's heeft menigeen aan het werk en denken gezet en, zoals het gaat, zijn er voor- en tegenstanders gekomen. Alvorens hier nader op in te gaan, lijkt het ons goed de argumenten van DIJKSTRA te illustreren met een voorbeeld. Op het examen Wetenschappelijk Rekenen A werd, teneinde ALGOL 60 kennis te testen, het volgende gevraagd:

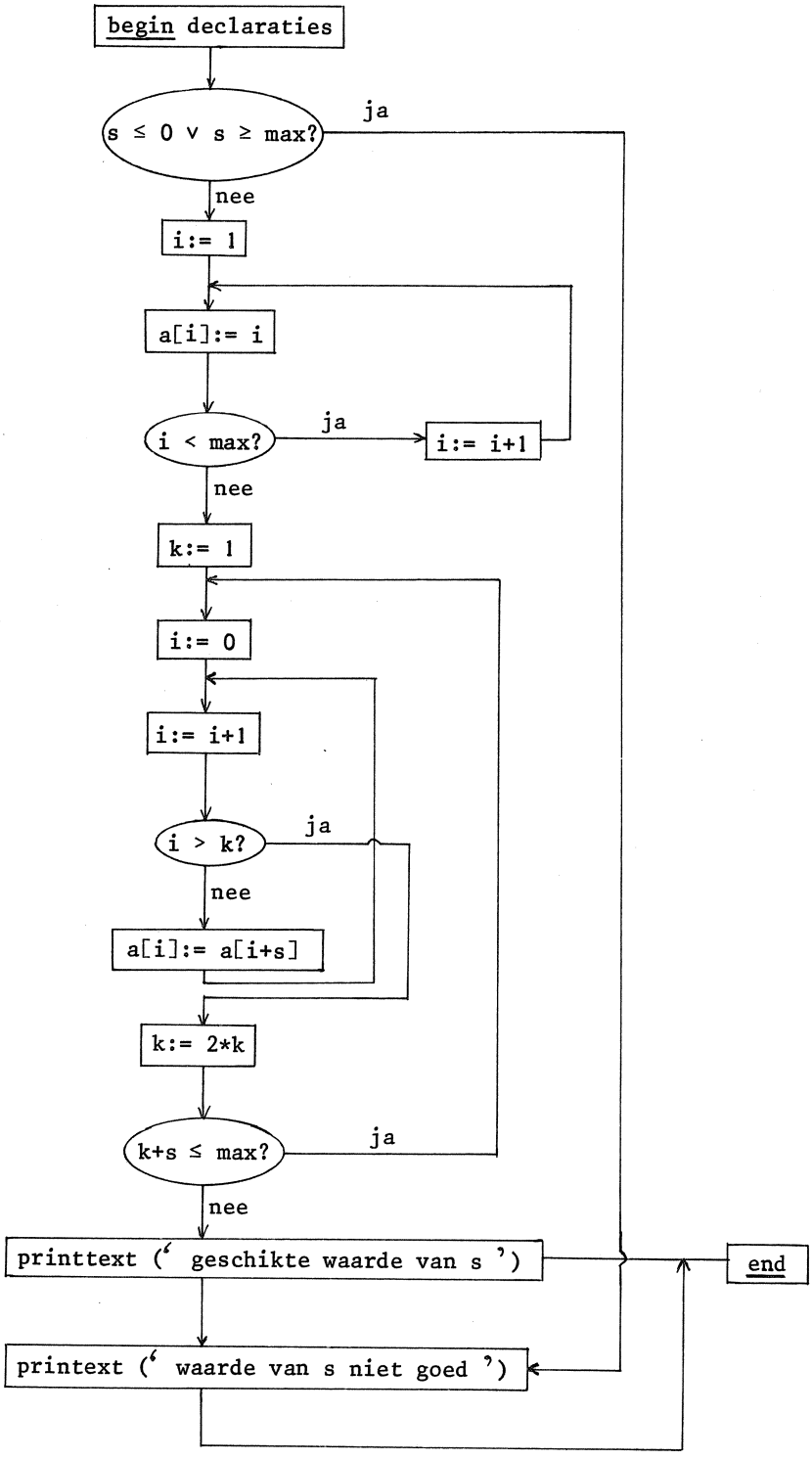
> Vervang het volgende ALGOL 60 block door een equivalent block, waarin zo min mogelijk labels voorkomen.
> s en max zijn niet-locale integers.

```
begin integer i, k;
      integer array a[1:max];
      if s ≤ 0 ∨ s ≥ max then go to P;
      i:= 1;
  L : a[i]:= i; if i < max then
      begin i:= i+1; go to L end;
      k:= 1;
  M : i:= 0;
  N : i:= i+1; if i > k then go to O;
      a[i]:= a[i+s];
      go to N;
  O : k:= 2 × k;
      if k+s ≤ max then go to M;
      printtext (' geschikte waarde van s ');
      go to Q;
  P : printtext (' waarde van s niet goed ');
  Q :
end
```

Het equivalente block heeft de volgende vorm:

```
begin integer i,k; integer array a[1:max];
      if s > 0 ∧ s < max then
      begin i:= 1;
            for a[i]:= i while i < max do i:= i+1;
            for k:= 1,2*k while k+s ≤ max do
            for i:= 1 step 1 until k do a[i]:= a[i+s];
            printtext (' geschikte waarde van s ')
      end else printtext (' waarde van s niet goed ')
end
```

Terwijl het tweede programma ogenblikkelijk te begrijpen is, moet men behoorlijk veel moeite doen om het eerste te vatten. Het is eigenlijk nodig om van het eerste programma een stroomdiagram te maken, zodat de structuur doorzichtiger wordt:

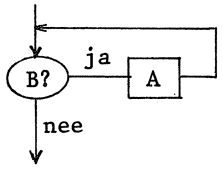

begin declaraties
s ≤ 0 ∨ s ≥ max?
ja
nee
i:= 1
a[i]:= i
i < max?
ja
i:= i+1
nee
k:= 1
i:= 0
i:= i+1
i > k?
ja
nee
a[i]:= a[i+s]
k:= 2*k
k+s ≤ max?
ja
nee
printtext (' geschikte waarde van s ')
end
printext (' waarde van s niet goed ')

Is het een wonder dat het programmeren met goto's ook wel "spaghetti programmeren" wordt genoemd HOPKINS [9].

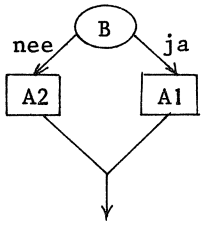
3. ITERATIE, CONDITIONELE STATEMENT EN BLOCK-STRUCTUUR

In veel gevallen laat de programmastructuur zich uitdrukken met behulp van iteraties, waarvoor we de door HOARE ingevoerde while statement als voorbeeld nemen, met behulp van de conditionele statement, met als voorbeeld de if-then-else statement en de case statement en met behulp van block-structuur.
In stroom-diagramstijl zien deze hulpmiddelen er als volgt uit:

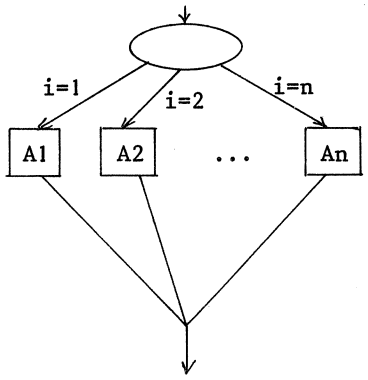
while B do A end



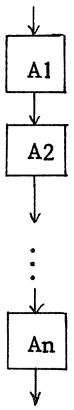
if B then A1 else A2



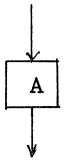
case i of (A1,A2,...,An)



begin A1;A2;...,An end



conceptioneel hetzelfde als:



4. GOTO'S ZIJN OVERBODIG

Op een eenvoudige manier kunnen we goto's uit een programma verwijderen en omzetten in één while gecombineerd met een case statement.
Aan de hand van het voorbeeld zullen we zien hoe dit in zijn werk gaat. Een automatische processor die dit uitvoert lijkt niet moeilijk te maken.

We voeren een variabele "state" in, voorts knippen we het programma bij de labels in tweeën (we nemen aan dat het programma geen block of compound statement bevat met labels). Elk van de stukken krijgt een integer waarde van 0 tot n toegevoegd. De stukken worden, gescheiden door komma's, als een case statement gestructureerd. Goto's worden vervangen door: "state:= i", waarbij i het nummer is van een door de label aangewezen stuk. Tenslotte wordt het geheel aan een "while state ≤ n do" gehangen.
Het resultaat van deze transformatie voor ons voorbeeld is:

```
begin integer i,k; integer array a[1:max];
      state:= 0;
      while state ≤ 6 do case state of
      (if s ≤ 0 ∨ s ≥ max then state:= 5 else
       begin i:= 1; state:= 1 end,
1: L: a[i]:= i; if i < max then begin i:= i+1; state:= 1 end
      else begin k:= 1; state:= 2 end,
2: M: i:= 0; state:= 3,
3: N: i:= i+1; if i > k then state:= 4 else
      begin a[i]:= a[i+s]; state:= 3 end,
4: O: k:= 2*k; if k+s ≤ max then state:= 2 else
      begin printtext (' geschikte waarde van s '); state:= 6 end,
5: P: printtext (' waarde van s niet goed '); state:= 6,
6: Q: state:= 7
     )
end
```

Uiteraard is bovenstaand programma even onduidelijk en ongestructureerd als het oorspronkelijke. Mogelijk is het zelfs slechter leesbaar geworden.
Een andere mogelijkheid om goto's kwijt te raken is door gebruik te maken van procedures op de volgende wijze KNUTH & FLOYD [10]: Label elk statement. Beschouw de rij "L1:S;L2:". Maak een procedure:

```
procedure L1; begin S; L2 end
```

Doe dit voor elk statement (het laatste statement uiteraard iets anders). Verander "goto L" in "L" en voer expliciet het eerste statement uit door een procedure-aanroep.
Deze techniek, reeds eerder door VAN WIJNGAARDEN [22] beschreven, is uiteraard slechts van theoretisch belang.

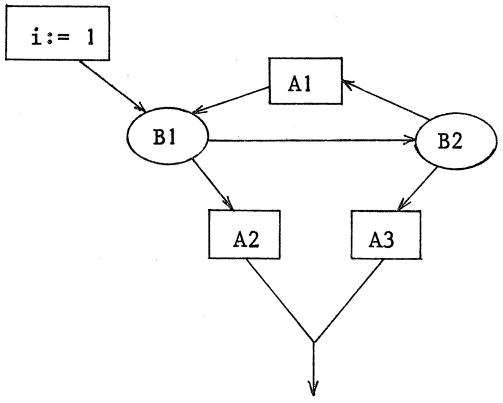
5. GOTO'S ZIJN NODIG

Bij het overboord zetten van goto's hebben we extra dingen gebruikt: een variabele en procedures. De vraag rijst of we het zonder deze extra's kunnen stellen. Het antwoord is, zoals te verwachten was, ontkennend. In [10] laten KNUTH en FLOYD zien dat het volgende programma, waarin een veel voorkomende constructie wordt gebruikt, de zogenaamde *exit constructie*, niet in eenvoudige repetities met condities is om te schrijven zonder invoering van een extra variabele en extra operaties (zoals bijv. de ∧ operatie):

```
for i:= 1 step 1 until n do
    if A[i] = x then goto found;
not found: n:= i; A[i]:= x; B[i]:= 1; goto end;
found: B[i]:= B[i]+1;
end:
```

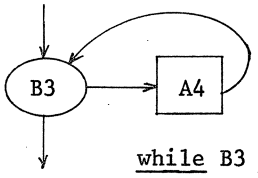
(N.B. Hier wordt gebruik gemaakt van de officieel ongedefinieerde waarde van de controlled variable i).
In navolging van PETERSON et al. [13] maken we van dit programma een stroomdiagram.

P:

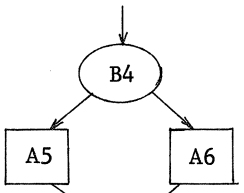


A1 ≡ i:= i+1
B1 ≡ i < n?
B2 ≡ A[i] = x?
A2 ≡ n:= i; A[i]:= x; B[i]:= 1
A3 ≡ B[i]:= B[i]+1

Schrijven we nu de stroomdiagrammen voor een while en een if statement op:



while B3 do A4



if B4 then A5 else A6

De vraag: "is programma P om te schrijven in een ander programma Q dat equivalent is met P maar geen goto's bevat?" leidt ogenblikkelijk tot de vraag wat we onder "equivalent" verstaan. Wel, we zullen hieronder verstaan dat bij elke weg volgens P van begin tot eind, die gerepresenteerd kan worden als een rij van elementaire acties $a_1\ a_2\ \ldots$, een weg door Q hoort van begin tot eind bestaande uit dezelfde rij elementaire acties, en omgekeerd. Deze elementaire acties betreffen statements s_i en tests t_i. Zo zijn A1, B1, B2 en A3 elementaire acties, terwijl A2 bestaat uit 3 elementaire acties.
Stel nu dat we een programma Q hadden gemaakt slechts opgebouwd uit while- en if statements. Neem die while of if die zelf geen while of if bevat. Stel dit is de while statement gerepresenteerd in ons stroomdiagram. Aangenomen dat dit statement uitgevoerd kan worden (anders laten we het weg), dan zullen door Q wegen lopen waarin de rij "... B3 A4 B3 A4 B3 A4 ..." voorkomt. Bij A4 mag gedacht worden aan een rij van s_i's. B3 moet echter één elementaire test zijn (dus ofwel "i < n?" of "A[i] = x?"). Deze wegen hebben dus de gedaante: "... $t_i\ s_p\ s_q\ t_j\ s_r\ t_k\ s_u\ s_v$ ...". Echter, de wegen in P hebben alle de gedaante: "... B1,B2,A1,B1,B2,A1,...", d.w.z. "... $t_a\ t_b\ s_c\ t_d\ t_e\ s_f\ t_g\ t_h\ s_j\ s_k$ ...". Conclusie: de aanname is incorrect.
Stel dat de diepste statement een if statement is, die dus noch while noch if statements bevat, dan concluderen we dat er wegen zijn waar "... B4,A5,C ..." in voorkomt en andere wegen waar "... B4,A6,C ..." in voorkomt; als deze wegen namelijk niet voorkomen kunnen we de if statement vervangen door een gewoon statement. Echter, hoe we B4 ook kiezen, het is niet mogelijk twee wegen in P aan te wijzen met de structuur "$t_i\ s_a\ s_b\ \ldots\ s_c$ C" en "$t_i\ s_x\ s_y\ \ldots\ s_z$ C", met $s_a\ s_b\ \ldots\ s_c \neq s_x\ s_y\ \ldots\ s_t$.
Conclusie: ook in dit geval leidt de aanname tot tegenspraak, zodat we moeten concluderen dat een equivalent programma in de vorm van while- en if statements niet bestaat.

N.B. ASHCROFT en MANNA [1] toonden met een iets algemenere definitie van "equivalentie" hetzelfde aan, maar dan wel gebruik makend van een ingewikkeld voorbeeld en een gecompliceerd bewijs.

6. MULTIPLE EXIT STATEMENTS

De eenvoudige while- en if statements bleken onvoldoende te zijn om goto's overbodig te maken.

Voor het voorbeeld in sectie 5 kunnen we een *exit statement* invoeren van de vorm:

```
L: begin ... exit L; ... end
```

met het effect:

```
begin ... goto exit; ... exit: end
```

Het voorbeeld krijgt nu de vorm:

```
L: begin for i:= 1 step 1 until n do
        if A[i] = x then begin B[i]:= B[i]+1; exit L end;
        n:= i; A[i]:= x; B[i]:= 1
   end;
```

Bij BOCHMANN [3] krijgt het voorbeeld de vorm:

```
for i:= 1 step 1 until n do
    if A[i] = x then exitloop found;
found: B[i]:= B[i]+ 1;
ended: n:= i; A[i]:= x; B[i]:= 1
```

Het idee van BOCHMANN is een multiple exit te creëren door de label te gebruiken om diverse uitgangen te maken, terwijl er één uitgang (ended) is voor het geval de loop uitgeput is. Hoewel dit idee aardig lijkt moet het om een aantal redenen worden afgewezen: de eerste is de gecompliceerdheid van de constructie, de tweede is dat dit soort exit statements niet voldoende mogelijkheden biedt omdat de exit uit geneste loops met slechts één stap tegelijk kan. PETERSON, KASAMI & TOKURA [13] bewezen namelijk een aantal interessante en verhelderende stellingen.
Als iteratie- en exit statement gebruiken zij, min of meer analoog aan KNUTH & FLOYD [10]:

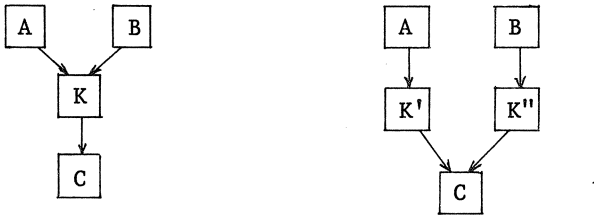
```
L: it ... exit L; ... ti      (notatie van mij)
```

met het effect:

```
while true do begin ... goto exit; ... end; exit:
```

De eerste stelling kennen we reeds: er zijn stroomdiagrammen die niet in while-if programma's vertaald kunnen worden zonder de lengte te vergroten en/of executievolgorde te veranderen. De tweede stelling zegt hetzelfde als de eerste ook in het geval dat multiple exit statements gebruikt mogen wor-

den. Voor de derde stelling moeten we het begrip *node splitting* of *knoopsplitsing* invoeren. Dit is een techniek om, door het maken van een kopie, een knoop met meerdere ingangen te veranderen in meerdere knopen elk met één ingang.



De derde stelling zegt nu dat er stroomdiagrammen zijn die niet in programma's met if statements, iteratie statements en met enkelvoudige exit statements vertaald kunnen worden, zelfs niet als knoopsplitsen is toegestaan. Dit laatste betekent dus dat we eventueel stukken programma mogen dupliceren.

Dat knoopsplitsen desalniettemin een machtig wapen is bewijst stelling 4 die zegt dat elk stroomdiagram omgezet kan worden in een welgevormd stroomdiagram door middel van knoopsplitsing.

De *welgevormde stroomdiagrammen* (op de precieze definitie, die nogal ingewikkeld is, kunnen we hier niet ingaan) zijn intuïtief precies die programma's die met de bovenstaande iteraties en multiple exit statements gemaakt kunnen worden. Elke loop heeft slechts één ingang en eventueel meerdere uitgangen. In een vijfde stelling tonen ze aan dat een bepaald, precies beschreven, algoritme elk welgevormd stroomdiagram vertaalt in een *welgevormd programma* (wgp).

Een wgp is een tamelijk willekeurig programma waarin if statements, repeat statements en multiple exit statements gebruikt kunnen worden. Er mogen ook goto's in voorkomen, maar hun gebruik is beperkt door de eis dat loops geheel binnen elkaar moeten liggen én binnen een if statement én binnen iteratie statements. Het is dan ook niet verwonderlijk dat voor wgp's goto's zonder meer geëlimineerd kunnen worden en omgezet in if en repeat statements.

Naar het oordeel van PETERSON ET AL. kenmerken de leesbare, goed gestructureerde, programma's zich hierdoor dat ze welgevormd zijn.

Dit wordt bevestigd door WULF [20,21] die de systeemprogrammeertaal BLISS zo ontwierp dat elk programma welgevormd is; goto's zijn in deze taal name-

lijk niet toegestaan.
Trouwens ook de ervaringen van schrijver dezes zijn dat de grotere programma's welgevormd zijn. Als voorbeeld van programmeren zonder goto's kan de MIXAL assembler van G. TEN VELDEN [17] genoemd worden die, nota bene in ALGOL 60 geprogrammeerd, slechts één goto bevat.
Als belangrijk voordeel van het verbieden van goto's noemt WULF [21] de mogelijkheid om de compiler de code te laten optimaliseren op veel eenvoudiger, en dus betere, manier dan voor een taal met goto's. Bovendien bestaat er geen gevaar voor de voor de compiler en voor het vertaalde programma zo akelige goto's uit een block.

Voor wat betreft bewijzen van correctheid van programma's met en zonder goto's verwijzen we naar andere sprekers in dit colloquium. We vestigen de aandacht op een artikel van CLINT & HOARE [6] waarin een iets andere goto wordt ingevoerd: de *label statement*. De label wordt als naam van een procedure opgevat die echter na executie niet terugspringt naar de plaats van aanroep maar naar het einde van het block. Dus: "begin label L: begin S1;...;Sn end; A; goto L; B; end" heeft tot effect: "begin A; S1;...;Sn; goto exit; B; exit: end". Deze constructie is in principe equivalent met de multiple exit statement. Het blijkt dat voor dit soort label statements eenvoudige bewijsregels op te stellen zijn. CLINT en HOARE illustreren dit met een correctheidsbewijs voor een logaritmische tabel-opzoek-algoritme.

7. GOTO'S EN RECURSIE

Zowel KNUTH & FLOYD [10] als WIRTH [19] kiezen als voorbeeld van programma's met goto's die niet zonder meer in while-if programma's vertaald kunnen worden, programma's voor in wezen *recursieve* processen.
Beschouw als voorbeeld (zie VAN DE RIET [14]) het in alfabetische volgorde printen van de inhoud van een binaire boom.
De knooppunten van de boom worden gerepresenteerd in een integer array "cel" en wel als volgt: als k een knooppunt aanwijst, dan wijst cel[k] de linker tak aan, mits ongelijk nul en cel[k-1] de rechter tak, mits ongelijk aan nul, terwijl cel[k+1],...,cel[k+n] de inhoud van het knooppunt bevatten. Overigens nemen wij aan dat n = 1.
Het printproces verloopt ongeveer als volgt: Zij k het knooppunt van de top van de boom; zij a het knooppunt van de top van de linker tak van k als deze er is, anders is a = 0; zij b het overeenkomstig knooppunt van de

rechter tak van k. Print eerst alle knooppunten van de boom met a als top; print het knooppunt k; print tenslotte alle knooppunten van de boom met b als top.
Om dit printproces te realiseren wordt een stapel gebruikt, die er als volgt uitziet:

```
integer array st[1:1000]; integer ptr;
```

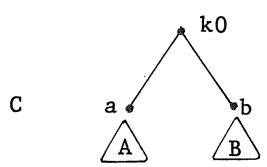
De procedure die de boom in alfabetische volgorde print is de volgende:

```
procedure print alf(k), value k; integer k;
begin integer ℓ; ptr:= ptr0; comment ptr0 heeft de waarde 0;
links: ℓ := cel[k];
       if ℓ>0 then begin ptr:= ptr+1; st[ptr]:= k; k:= ℓ; goto links end;
midden: print(cel[k+1]); comment de informatie wordt geprint;
rechts: k:= cel[k-1]; if k>0 then goto links;
        if ptr > ptr0 then begin k:= st[ptr]; ptr:= ptr-1; goto midden end
end
```

Nu zal dit programma niet iedereen even duidelijk zijn; integendeel, slechts weinigen zullen het ogenblikkelijk helder vinden. Het proces verloopt als volgt:

- ga zo ver mogelijk naar links, elke keer k op de stapel zettend;
- print de inhoud van k;
- ga dan naar rechts als het kan en ga ogenblikkelijk herhalen met het naar links gaan en anders, haal van de stapel een nieuw knooppunt en herhaal het spelletje.

Teneinde overtuigd te raken van de juistheid van de algoritme wordt in [14] zelfs overgegaan tot een bewijsvoering die ongeveer als volgt verloopt:
We beweren dat het programma: a) de knooppunten in "alfabetische" volgorde print; b) de waarde van ptr na afloop gelijk houdt aan die bij het begin (dus ptr0).
Volledige inductie wordt als volgt toegepast: neem aan de bewering is juist voor bomen met hoogstens n knopen. Zij C een boom met n+1 knopen. Zij A de linker tak met knoop a, B de rechter tak met knoop b en k0 het knooppunt van de top, dan hebben zowel A als C minder dan n+1 knopen zodat, wanneer A of C aangeboden was aan "print alf" de deelbomen alfabetisch geprint zouden zijn. Bovendien zou de



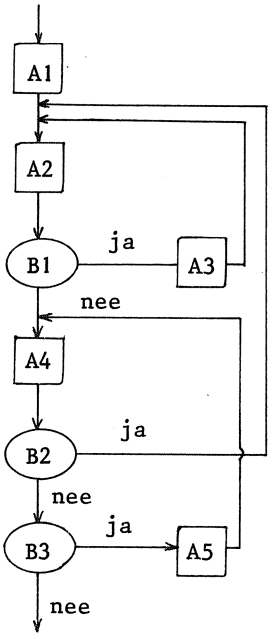
waarde van ptr voor en na afloop van het proces hetzelfde zijn.
We analyseren nu de aanroep "print alf (k0)":

1. ptr:= ptr0 (die nul was).
2. ℓ:= cel[k0], **dus** ℓ = a.
   2.1. Stel a > 0 dan wordt ptr = ptr0+1, st[ptr0+1] = k0, k:= a en we gaan naar de label "links".
   Precies hetzelfde effect hadden we verkregen door "print alf(a)" aan te roepen, maar dan met de waarde van ptr0 één hoger. De inductieaanname zegt nu dat dit fictieve proces zou aflopen met ptr = ptr0+1. Als gevolg van de directe sprong naar "links" zou dit fictieve proces aflopen als ptr0 één groter was geweest; hij is echter niet één groter dus zal als laatste activiteit van het werkelijke proces de statement:

   if ptr > ptr0 then begin k:= st[ptr]; ptr:= ptr-1; goto midden end

   met ptr = ptr0+1 in de conditie uitgevoerd worden.
   Gevolg is: k = k0; ptr = ptr0 en een sprong naar de label "midden".
   2.2. Stel a = 0 dan zijn we ook bij de label "midden" aangeland en ook nu geldt k = k0 en ptr = ptr0.
3. De inhoud van knooppunt k0 wordt geprint.
4. k:= cel[k0-1], dus k = b.
   4.1. Stel b > 0 en we gaan naar label links.
   Ook nu passen we de inductieaanname toe op het fictieve proces "print alf(b)" dat netjes de elementen van B in alfabetische volgorde print en de waarde van ptr voor en na het proces ongewijzigd laat. Voor het werkelijke proces was die waarde ptr0, zodat de beginsituatie van "print alf(b)" hieraan gelijk is, waaruit volgt dat de knopen B inderdaad netjes geprint worden, terwijl de waarde van ptr nog steeds ptr0 is.
   4.2. Als b = 0, springen we niet naar "links" maar bekijken we de waarde van ptr, die gelijk aan ptr0 is, zodat de laatste statement ook is uitgevoerd.
5. Het programma is uitgevoerd en heeft gedaan wat we vroegen, waarmee de inductiestap bewezen is.

Mogelijk is de bewijsvoering niet geheel overtuigend; één ding is in ieder geval wel duidelijk, namelijk dat het programma gecompliceerd is. Wellicht kunnen we iets meer duidelijkheid verkrijgen door gebruik te maken van if- en while statements.
Het programma in stroomdiagramstijl ziet er als volgt uit:



A1 ≡ ptr:= ptr0;

A2 ≡ 1:= cel[k];

B1 ≡ 1 > 0?

A3 ≡ ptr:= ptr+1; st[ptr]:= k; k:= 1;

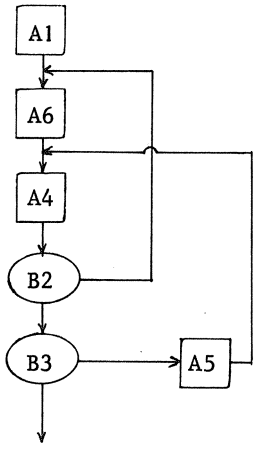
A4 ≡ print(cel[k+1]); k:= cel[k-1];

B2 ≡ k > 0?

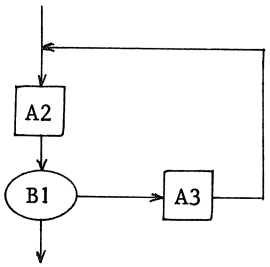
B3 ≡ ptr > ptr0?

A5 ≡ k:= st[ptr]; ptr:= ptr-1;

We zien ogenblikkelijk dat er een simpele loop in zit, namelijk A2 B1 A3. Voeren we A6 in als aangegeven, dan krijgen we de meest eenvoudige vorm van de algoritme:



met A6:



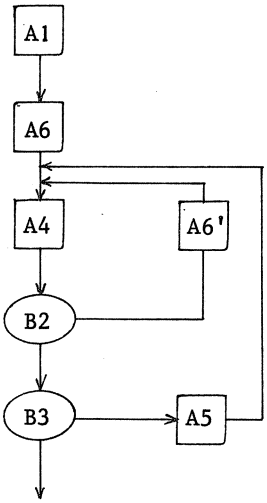
A6 kunnen we schrijven als:

```
L: it ℓ:= cel[k]; if ¬ℓ> 0 then exit L;
      ptr:= ptr+1; st[ptr]:= k; k:=ℓ ;
   ti
```

In ALGOL 60 kunnen we (toevalligerwijs) A6 ook beschrijven met een while:

```
for ℓ:= cel[k] while ℓ > 0 do
begin ptr:= ptr+1; st[ptr]:= k; k:= ℓ end
```

Zoals gesuggereerd door PETERSON [13] en KNUTH [10] passen we knoopsplitsing toe op de hoofdalgoritme. PETERSON geeft een redenering waaruit blijkt dat we zonder knoopsplitsing geen equivalent if-while programma kunnen maken (stelling 3).
Knoopsplitsing is niet moeilijk, kennelijk moeten we A6 herhalen:



In de vorm van if- en iteratie statements:

```
   A1; A6;
L1: it L2: it A4; if  ¬ B2 then exit L2; A6' ti;
       if  ¬ B3 then exit L1; A5
   ti
```

Uitgeschreven heeft de procedure

```
procedure print alf2 (k); value k; integer k;
begin integer ℓ;  ptr := ptro;
          L0: it ℓ := cel[k]; if ¬ ℓ > 0 then exit L0;
                 ptr := ptr+1; st[ptr] := k; k := ℓ
              ti;
```

```
L1: it L2: it print(cel[k+1]); k:= cel[k-1];
                if ¬k > 0 then exit L2;
                L3: it ℓ:= cel[k]; if ¬ℓ > 0 then exit L3;
                       ptr:= ptr+1; st[ptr]:= k; k:= ℓ
                    ti;
            ti;
        if ¬ptr > ptr0 then exit L1;
        k:= st[ptr]; ptr:= ptr-1
    ti
end
```

Inderdaad, het programma is gemaakt zonder goto's en zonder een extra toestandsvariabele (sectie 4), echter is het duidelijker? Integendeel!

De vraag rijst: hoe duidelijk en helder kan het programma maximaal zijn? Om deze vraag te beantwoorden gaan we terug naar het recursieve proces zelf; uiteindelijk moet dit proces honderd procent duidelijk zijn:

```
procedure print alf rec(k); value k; integer k;
begin if cel[k] > 0 then print alf rec(cel[k]);
      print(cel[k+1]);
      if cel[k-1] > 0 then print alf rec(cel[k-1])
end
```

Om van deze recursieve versie naar de goto-versie te komen is het nodig het invoeren van de stack te expliceren. Neem de volgende recursieve, parameterloze, procedure:

```
procedure P;
begin A; P; B; P; C end
```

N.B. Het is de taak van de niet nader gespecificeerde A, B en C het proces te beëindigen.

Het uitvoeren van P betekent: voer A uit; voer dan P weer uit en als je klaar bent voer dan B uit; voer dan P weer uit en als je klaar bent C. Het uitvoeren van A is geen probleem; maar als we deze procedure niet-recursief maken, hoe weten we dan dat na de eerste uitvoering van P ook B, vervolgens weer P en daarna C, nog moeten worden uitgevoerd? Simpelweg door een geheugen in te voeren in de vorm van een stapel.
Nu is een stapel een LIFO instrument, zodat we de volgorde van het op de

stapel zetten zo moeten kiezen dat het er afhalen, hetgeen tevens uitvoering betekent, in de juiste volgorde gebeurt. (Wellicht zou het conceptueel eenvoudiger zijn om met een FIFO geheugen in de vorm van een "queue" (KNUTH [11]) te werken, maar we zijn nu eenmaal gewend geraakt aan stapels.)
Voor de procedure P betekent dit dat we na uitvoering van A op de stapel moeten zetten dat, na uitvoering van P nog gedaan moeten worden B, P en C. Daarna kunnen we aan P beginnen. Een nog eenvoudiger voorstelling van zaken krijgen we door de uitvoering van de procedure P slechts hieruit te laten bestaan, dat op de stapel wordt gezet dat A, P, B, P en C in deze volgorde nog moeten worden uitgevoerd.
Op de stapel moet nu onderscheid worden gemaakt tussen A, B, C en P processen. Daartoe geven we elk van deze processen een identificatienummer case van 1 tot 4.
De stapel wordt bespeeld met de volgende procedures:

```
integer procedure stack(case,inf); value case, inf; integer case, inf;
```

die case en inf op de stapel zet en de plaats ervan aanwijst; en

```
Boolean procedure unstack(case,inf); integer case, inf;
```

die false wordt als de stapel leeg is en anders true wordt, terwijl in dat geval aan de actuele parameters case en inf de stapelwaarde wordt meegegeven.
De procedure P kan nu eenvoudig als volgt geïmplementeerd worden:

```
stack(4,...);
while unstack(proces,informatie) do
if proces = 1 then A else
if proces = 2 then B else
if proces = 3 then C else
begin stack(3,...); stack(4,...); stack(2,...); stack(4,...);
      stack(1,...)
end
```

Wanneer er parameters aan P zitten, kunnen deze op de plaats van "..." komen. Wanneer er locale variabelen zijn, is de toestand gecompliceerder; het kan dan noodzakelijk zijn dat proces A informatie achterlaat voor P, terwijl P weer voor informatie t.b.v. B moet zorgen. We kunnen dit ook op eenvoudige wijze inbouwen door aan het begin van het P proces voldoende ruimte

op de stapel te reserveren en de wijzer naar die ruimte mee te geven aan de verschillende processen.
De rechtstreekse vertaling van bovenstaande gedachten naar het alfabetisch print proces is:

```
stack(2,k);
while unstack(proces,k) do
if proces = 2 then
begin if cel[k-1] > 0 then stack(2,cel[k-1]);
      stack(1,k);
      if cel[k] > 0 then stack(2,cel[k])
end else print(cel[k+1]);
```

De vertaling was recht-toe recht-aan, zodat we voor de juistheid van het programma nog in kunnen staan.
Nu echter gaan we enkele typische programmeertrucjes toepassen, ten behoeve van "efficiëntie" (zodat de kans dat we later het verkregen programma niet meer zullen begrijpen groot is).
We constateren namelijk dat het einde van proces 2, namelijk "stack(2,cel[k])", ogenblikkelijk gevolgd wordt door "unstack ( )". Dit kan kortgesloten worden:

```
stack(2,k);
while unstack(proces,k) do
      if proces = 2 then
      begin L: if cel[k-1] > 0 then stack(2,cel[k-1]);
               stack(1,k);
               if cel[k] > 0 then begin k:= cel[k]; goto L end
      end else print(cel[k+1])
```

Verder constateren we dat de conditionele stapeloperatie stack(2,...) absoluut gemaakt kan worden door het proces 2 te veranderen in:

```
if proces = 2 then
begin if k > 0 then
      begin L: stack(2,cel[k-1]); stack(1,k);
               if cel[k] > 0 then begin k:= cel[k]; goto L end
      end
end
```

Door i.p.v. cel[k-1], de waarde van k op de stapel te zetten krijgen we:

```
if proces = 2 then
begin k:= cel[k-1]; if k > 0 then
      begin L: stack(2,k); stack(1,k);
               if cel[k] > 0 then begin k:= cel[k]; goto L end
      end
end
```

Uiteraard moet de allereerste stackoperatie ook wat veranderen, maar we zien dadelijk wel hoe.
We constateren nu dat de stapel, behalve het eerste element, nu altijd de gedaante: $(2,k_1)(1,k_1)(2,k_2)(1,k_2)\ldots(2,k_n),(1,k_n)$ zal hebben. Dit houdt in dat bij het afhalen van $(1,k_i)$ van de stapel de volgende altijd $(2,k_i)$ zal zijn. Dus na uitvoering van het proces 1 (het printen) volgt altijd proces 2 met dezelfde k. Of, anders gezegd, net voorafgaand aan proces 2 gaat altijd proces 1 (afgezien van misschien de eerste keer). Het ligt dus voor de hand beide processen te combineren tot:

```
while unstack(k) do
begin print(cel[k+1]);
      k:= cel[k-1]; if k > 0 then
      begin L: stack(k);
               if cel[k] > 0 then begin k:= cel[k]; goto L end
      end
end
```

Wanneer de start van het proces gegeven is door "stack(a)", moeten we bedenken dat eerst de inhoud van a wordt geprint; daarna wordt met cel[a-1], dus de rechter tak van a, verder gegaan.
Als de te printen boom gegeven is door de top k, kunnen we een hulpknoop a invoeren met cel[a-1] = k. Als verder cel[a+1] = 0, wordt de boom geprint, voorafgegaan door het getal 0.
Indien we geen hulpknoop willen invoeren, zijn we verplicht óf een goto uit te voeren midden in de while statement, namelijk naar de statement: "if k > 0 then ...", óf de laatste statement te herhalen:

```
if k > 0 then
begin L: stack(k); if cel[k] > 0 then begin k:= cel[k]; goto L end end;
while unstack(k) do
begin print(cel[k+1]); k:= cel[k-1]; if k > 0 then
      begin L1: stack(k); if cel[k] > 0 then
                begin k:= cel[k]; goto L1 end
      end
end;
```

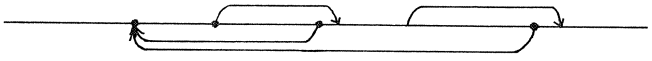
Vergelijken we nu dit resultaat met de procedure print alf 2, dan blijkt dat er een frappante overeenkomst is.
Hiermee zien we ook hoe duidelijk deze procedure, en daarmee de oorspronkelijke procedure print alf is, namelijk net zo duidelijk als

de duidelijkheid van het recursieve proces;
de duidelijkheid van het stapelmechanisme;
de duidelijkheid van de "smerige" programmeertrucjes.

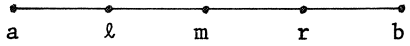
Is het een wonder dat het programma zonder goto's zo onduidelijk is?

8. EEN ANDER VOORBEELD VAN GOTO'S EN RECURSIE

In het boekje *Computerwiskunde* [16] beschrijft VAN DER SLUIS een integratiemethode met "glijdende" stapgrootte. Het resulterende programma bevat drie labels en vier goto's en heeft globaal de volgende structuur:



Naar mijn overtuiging is dit precies het soort programma's dat in een inleiding van de informatica niet mag voorkomen, aangezien het voor de eenvoudige geest te gecompliceerd is en voor de briljante geest een uitnodiging om zelf ook van deze "getructe" programma's te gaan maken.
Hoe het naar onze mening wel zou moeten, schetsen we hieronder [15].
Het programma, in de vorm van een recursieve procedure "int", moet de functie f(x) integreren voor $a \leq x \leq b$, terwijl een fout ter grootte "eps" is toegestaan.



Het interval wordt in 4 gelijke stukken verdeeld met steunpunten a, ℓ, m, r en b.

Nu leert een numeriek wiskundige beschouwing dat de integraal van a naar b benaderd wordt door de uitdrukking:

$$I = (f(a) + 4(f(\ell) + f(r)) + 2f(m) + f(b)) * (b-a)/12 + T$$

waarbij T de vermoedelijke fout is. T is gegeven door:

$$T = (4(f(\ell) + f(r)) - 6f(m) - (f(a) + f(b))) * (b-a)/180.$$

(De formules ontstaan door 2 maal Simpson toe te passen en te extrapoleren; T heet de Richardson-correctie).
In het vervolg nemen we aan dat T de echte fout aangeeft. De meest eenvoudige algoritme is nu de volgende:

> als $|T| < eps$, dan is de integraal gelijk aan I, anders is de integraal de som van de integralen van a naar m en van m naar b, met dien verstande dat de toegelaten fout voor beide integralen eps/2 is.

In ALGOL 60:

```
real procedure intl(f,a,b,eps); value a,b,eps; real a,b,eps;
     real procedure f;
begin real ℓ,m,r,fa,fl,fm,fr,fb;
      m:= (a+b)/2; ℓ:= (a+m)/2; r:= (m+b)/2;
      fa:= f(a); fℓ:= f(ℓ);fm:= f(m); fr:= f(r); fb:= f(b);
      T:= (4*(fℓ+fr)-6*fm-(fa+fb)) * (b-a)/180;
      intl:= if abs(T) > eps then intl(f,a,m,eps/2)+intl(f,m,b,eps/2)
             else (fa+4*(fℓ+fr)+2*fm+fb) * (b-a)/12 + T
end
```

Wederom uit overwegingen van efficientie maken we enkele veranderingen:

1e. de parameter eps is niet nodig aangezien het halveren van eps gepaard gaat met het halveren van het interval.
Als de allereerste eps, a en b gegeven zijn door eps0, a0 en b0, dan kan een willekeurige eps bepaald worden uit:

$$eps = eps0 * |(b-a)/(b0-a0)|.$$

Als we nu de test "abs(T) > eps" vervangen door

"abs(T') * |b-a|/180 > eps0 * |(b-a)/(b0-a0)|"

hetgeen equivalent is met:

"abs(T') > eps0 * 180/|b0-a0|"

waarin T' = 4 * (fℓ+fr) - 6 * fm - (fa+fb), dan kunnen we inderdaad eps uit de parameterlijst weglaten.

2e. Nadat fa, fℓ, fm, fr, fb en m berekend zijn, worden ze, als T te groot is, weer opnieuw berekend bij intl(f,a,m,eps/2) en intl(f,m,b,eps/2). Liever voeren we vier nieuwe parameters in: fa, fm, fb en m, waarin we aan de nieuwe aanroepen van intl de waarden van f aan de uiteinden en in het midden kunnen meegeven omdat we die reeds eerder berekend hadden.

De vorm wordt nu:

```
real procedure int2(f,a0,b0,eps0); value a0,b0,eps0; real a0,b0,eps0;
          real procedure f;
begin real delta;
      real procedure int3(fa,fm,fb,a,m,b); value fa,fm,fb,a,m,b;
                real fa,fm,fb,a,m,b;
      begin real ℓ,r,fℓ,fr; ℓ:= (a+m)/2; r:= (m+b)/2;
            fℓ:= f(ℓ); fr:= f(r);
            T:= 4 * (fℓ+fr) - 6 * fm - (fa+fb);
            int3:= if abs(T) > delta then
                       int3(fa,fℓ,fm,a,ℓ,m) + int3(fm,fr,fb,m,r,b)
                   else (fa+4*(fℓ+fr)+2*fm+fb+T/15) * (b-a)
      end;
      delta:= eps0 * 180/abs(b0-a0);
      int2:= int3(f(a0),f((a0+b0)/2),f(b0),a0,(a0+b0)/2,b0)/12
end
```

Nu passen we het stapelmechanisme toe om de procedures te "derecursiveren":

```
delta:= eps0 * 180/abs(b0-a0); I:= 0;
stack(f(a0),f((a0+b0)/2),f(b0),a0,(a0+b0)/2,b0);
while unstack(fa,fm,fb,a,m,b) do
begin ℓ:= (a+m)/2; r:= (m+b)/2;
      fℓ:= f(ℓ); fr:= f(r);
      T:= 4 * (fℓ+fr) - 6 * fm - (fa+fb);
      if abs(T) > delta then
      begin stack(fm,fr,fb,m,r,b); stack(fa,fℓ,fm,a,ℓ,m) end
      else I:= I + (fa+4*(fℓ+fr)+2*fm+fb+T/15) * (b-a)
end;
int2:= I/12;
```

We kunnen nu weer enkele programmeertrucjes gebruiken zoals:

1. de integraal wordt van links naar rechts "opgerold", zodat het niet nodig is de waarden aan de linkerkant op de stapel te zetten omdat deze gelijk zijn aan de waarden aan de rechterkant van het vorig berekende interval. Dus fa en a worden expliciet aan het eind van een interval gelijk gesteld aan fb en b.
2. De tweede aanroep van stack kan vervangen worden door: "fb:= fm; fm:= fℓ; b:= m; m:= ℓ; goto L", waarin L een label is die aan "ℓ:= (a+m)/2" geplakt wordt.
3. De test "unstack(...)" niet aan het begin, maar aan het eind.

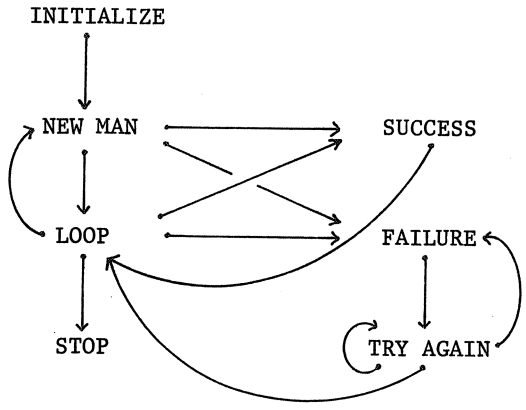
Het is nu niet moeilijk meer in te zien dat bovenstaand correct programma equivalent is met het programma van VAN DER SLUIS dat we voor de volledigheid hier afdrukken:

```
        delta:= eps0 * 180/abs(b0-a0); I:= 0; ptr:= 0;
        a:= a0; b:= b0; m:= (a0+b0)/2; fa:= f(a); fm:= f(m); fb:= f(b);
cycle:  ℓ:= (a+m)/2; p:= (m+b)/2; fℓ:= f(ℓ); fr:= f(r);
        T:= 4 * (fℓ+fr) - 6 * fm - (fa+fb);
        if abs(T) ≤ delta then goto vervuld;
        st[i+1]:= fr; st[i+2]:= fb; st[i+3]:= r; st[i+4]:= b; i:= i+4;
        fb:= fm; fm:= fℓ; b:= m; m:= ℓ; goto cycle;
```

```
vervuld: I:= I + (b-a) * (fa+4*(fℓ+fr)+2*fm+fb+T/15);
         if i = 0 then goto klaar
         a:= b; fa:= fb;
         i:= i-4; fr:= st[i+1]; fb:= st[i+2]; r:= st[i+3]; b:= st[i+4];
         goto cycle;
klaar:   int2:= I/12;
```

9. CONCLUSIE

Programma's die in essentie recursief zijn, kunnen zeker iteratief worden gemaakt zonder dat de helderheid verloren gaat door het expliciet maken van het stapelmechanisme. Gaan we deze programma's uit efficientie-overwegingen verder uitkleden dan hebben we daarvoor goto's nodig. Het resultaat is dan echter ondoorzichtig en dus vermoedelijk incorrect geworden.
Slechts aan de hand van het derecursiveringsproces kan de correctheid behouden blijven. Van de gevaren van programmeertrucjes moet men zich echter bewust zijn.
In het bovenstaande zijn twee recursieve programma's tamelijk grondig bekeken. Dit soort programma's komen we herhaaldelijk tegen, veelal in de onleesbare goto stijl. Zo wordt door GRIES in zijn boek over compilerbouw [7, p.89] een algoritme gegeven dat een gegeven tekst analyseert of hij een zin is van een zeer algemene context-vrije grammatica. Hoewel schrijver dezes enkele uren besteedde om de algoritme te doorgronden is het tot op heden niet anders gelukt dan door zelf een recursief proces te definiëren. Deze algoritme heeft de volgende structuur:



Grafentheoretici en automatentheoretici zullen dit vermoedelijk een kolfje naar hun hand vinden.

Slecht gestructureerde programma's gemaakt met goto's brachten Dijkstra er toe de kwaliteit van een programmeur omgekeerd evenredig te schatten met de hoeveelheid goto's die hij gebruikt.

MARTIN HOPKINS presteert het nota bene in een pleidooi [9] voor de goto, deze te verdedigen met het argument dat ze zo handig zijn om even snel een "fout te verbeteren". Letterlijk zegt hij:

> "Part of the reason for retaining the goto is that the world is not always a very elegant place and sometimes a goto is a useful, if ugly, tool to handle an awkward situation. Algorithms are often messy. Sometimes this may be due to inherent complexity. I suspect, however, that most of the time it is because not enough time or intelligence has been applied. Where time and intelligence are lacking a goto may do the job. Every program will not be published. Many may be used only once. I tend to sympathise with the programmer who fixes up a one-time program at 3.00 a.m. with a goto ..."

Een dergelijke houding is de oorzaak dat computers op supersnelle wijze en vermoedelijk ook op zeer efficiente wijze zullen doorgaan met het produceren van foute resultaten.

AUERBACH zegt in the *Skyline of information processing* [23, p.11]:

> "Unfortunately, in information systems this type of efficiency (speed of computation, number of machine instructions, memory) is often bought at too high a price: namely lack of structural modularity".

We zien hier iemand, een computerman van het eerste uur en leider van een groot software-house, die "structural modularity" dus minstens even belangrijk vindt als efficientie.

LITERATUUR

[1] ASHCROFT, E. & Z. MANNA, *The translation of 'g to' programs to 'while' programs*, in: *Information Processing*, C.V. FREIMAN (ed.), Proceedings IFIP Conference 1971 (Ljubljana), TA-2, North-Holland Publ. Cy., Amsterdam, 1972, pp.147-152.

[2] BABBAGE, Charles, in: *Charles Babbage and his calculating engines*, Philip MORRISON & Emily MORRISON (eds.), Dover Publications, New York, 1961.

[3] BOCHMANN, G.V., Multiple exits from a loop without the goto, *Comm. ACM*, 16 (1973) 443-444.

[4] DIJKSTRA, E.W., Goto statement considered harmful, *Comm. ACM*, 11 (1968) 147-148.

[5] DIJKSTRA, E.W., A constructive approach to the problem of program correctness, *BIT*, 8 (1968) 174-186.

[6] CLINT, M. & C.A.R. HOARE, Jumps and functions, *Acta Informat.*, 1 (1972) 214-224.

[7] GRIES, D., *Compiler construction for digital computers*, Wiley, New York, 1971.

[8] HOARE, C.A.R., A note on the for statement, *BIT*, 12 (1972) 334-341.

[9] HOPKINS, M., A case for the goto, *Proceedings ACM*, 1972, pp.787-790.

[10] KNUTH, D.E. & R.W. FLOYD, Notes on avoiding "goto" statements, *Information processing letters*, 1971, pp.23-31.

[11] KNUTH, D.E., *The art of computer programming 1, Fundamental algorithms*, Addison-Wesley, Reading (Mass.), 1968.

[12] LEAVENWORTH, B.M., Programming with(out) the goto, *Proceedings ACM*, 1972, pp.782-287 (bevat 90 referenties).

[13] PETERSON, W.W., T. KASAMI & N. TOKURA, On the capabilities of while, repeat and exit statements, *Comm. ACM*, 16 (1973) 503-512.

[14] RIET, R.P. van de, *Simulatietechnieken*, collegedictaat Vrije Universiteit.

[15] RIET, R.P. van de, *Basiscursus informatica*, collegedictaat Vrije Universiteit.

[16] SEIDEL, J.J. (red.), *Computerwiskunde*, Aula Boeken 407, Het Spectrum, Utrecht, 1969.

[17] VELDEN, G. ten, *Mixal assembly*, Mathematisch Centrum Rapport IW 3/73, Amsterdam, 1973.

[18] WIRTH, N. & C.A.R. HOARE, A contribution to the development of ALGOL, *Comm. ACM*, 9 (1966) 413-432.

[19] WIRTH, N., *Programming and programming languages*, Proceedings International Computing Symposium, Bonn, 1970.

[20] WULF, W.W., *Programming without the goto*, in: *Information Processing*, C.V. FREIMAN (ed.), Proceedings IFIP Conference 1971 (Ljubljana, TA-3, North-Holland Publ. Cy., Amsterdam, 1972, pp.84-88.

[21] WULF, W.W., A case against the goto, *Proceedings ACM*, 1972, pp.791-797.

[22] WIJNGAARDEN, A. van, *Recursive definition of syntax and semantics*, in: *Formal language description languages for Computer Programming*, T.B. STEEL, Jr. (ed.), North-Holland Publ. Cy., Amsterdam, 1966.

[23] ZEMANEK, H. (ed.), *The skyline of information processing*, North-Holland Publ. Cy., Amsterdam, 1973.

MATHEMATICAL CENTRE SYLLABUS 21, 1975, III, 47-67.

GESTRUCTUREERD PROGRAMMEREN

L.J.M. GEURTS

1. INLEIDING

In de praktijk van het programmeren speelt het bewijzen van de correctheid van een programma slechts een bescheiden rol. Meestal wordt in het geheel geen correctheidsbewijs gegeven, bv. om een van de volgende redenen.

- Het leveren van zo'n bewijs blijkt in het algemeen een moeizame zaak, en maar weinig programma's hebben de importantie die zo'n inspanning rechtvaardigt.
- Voor veel programma's is het niet goed mogelijk nauwkeurig de eisen vast te stellen waaraan de resultaten moeten voldoen (is een damprogramma correct als het voor elke bordstand een legitieme zet aflevert?).
- Als een programmeur de laatste hand aan zijn programma heeft gelegd zal hij ervan overtuigd zijn dat het doet wat hij wil, en zal hij menen dat vergissingen door testen tijdig aan het licht zullen komen.
- De programmeur is niet altijd bij machte de correctheid van een correct programma ook daadwerkelijk aan te tonen: het bewijs is vaak moeilijker te bedenken dan het programma.

Maar ook als de omstandigheden er wel aanleiding toe geven heeft de programmeur heel wat meer te doen dan het leveren van een correctheidsbewijs: hij moet eerst een correct programma creëren.
Het is juist deze activiteit van het ontwikkelen van een programma die het onderwerp van deze beschouwing uitmaakt. Het zal blijken dat het geheel van ideeën en technieken dat door de term "gestructureerd programmeren" wordt aangeduid, de programmeur tot richtsnoer kan dienen om tot een goed programma te komen, maar hem tegelijk de aanzet tot een correctheidsbewijs in handen geeft.

2. VERDEEL EN HEERS

Centraal in het gestructureerd programmeren staat het strategische adagium "divide et impera". Voor het beheersen van iets dat te omvangrijk of te complex is om in zijn geheel aangepakt te worden, staat ons geen ander

middel ten dienste dan het te splitsen in delen die elk wel te overzien zijn, of op dezelfde manier verdeeld kunnen worden. Toegespitst op het ontwikkelen van een programma: de te beschrijven taak kan beschreven worden in termen van enige minder complexe taken, die op hun beurt beschreven moeten worden. De resultaten van deze disciplinering in het programmeren komen tegemoet aan vele eisen die aan programma's gesteld mogen worden:

1. *Betrouwbaarheid.* Als steeds een klein overzienbaar probleem wordt aangepakt, worden niet zo gemakkelijk fouten gemaakt.
2. *Bewijsbaarheid.* Het bewijs van de correctheid van het gehele programma valt uiteen in een aantal weinig complexe bewijzen, berustend op asserties die bij het uiteenrafelen van het programma bedacht zijn.
3. *Aanpasbaarheid.* Het aanpassen van het programma aan nieuwe eisen kan beperkt blijven tot veranderingen in een zo klein mogelijk deel van het programma.
4. *Testbaarheid.* Het testen van het programma op de aanwezigheid van fouten en foutjes van allerlei soort kan per sub-algoritme gebeuren, zodat een fout gemakkelijk gelokaliseerd kan worden. (Ten overvloede: testen moet gebeuren met testgevallen die, blijkens de tekst van een sub-algoritme, relevant zijn voor die sub-algoritme; maar ook dan kan testen slechts in triviale gevallen een correctheidsbewijs vervangen.)
5. *Documentatie.* De tekst van het uiteindelijke programma zal in het algemeen te omvangrijk of te ingewikkeld zijn om gemakkelijk begrijpbaar te zijn; de tekst van de algoritmen die hoger in de hiërarchische ontwikkeling staan vormt een ideale documentatie.
6. *Mogelijkheid tot taakverdeling.* Een aantal programmeurs kan onafhankelijk van elkaar sub-algoritmen ontwikkelen.

Het is duidelijk dat er nauwe relaties tussen de zes genoemde punten bestaan: testbaarheid is pas nuttig als het programma gemakkelijk aanpasbaar is als er een fout gevonden is; aanpasbaarheid vereist documentatie (het is ondoenlijk een rij bits aan te passen); betrouwbaarheid en bewijsbaarheid gaan hand in hand: een onbewijsbaar programma boezemt weinig vertrouwen in, een programma waarvan het bewijs triviaal lijkt is ook betrouwbaar.
Wat de onder punt 2 genoemde asserties betreft: gestructureerd programmeren komt pas goed tot zijn recht in samenhang met asserties à la (FLOYD [10], DE BAKKER [1]) zoals die gelden voor en na elke sub-algoritme. Andersom is het moeilijk in een ongestructureerd programma de voor een correctheidsbewijs nodige asserties te plaatsen.

3. EEN VOORBEELD

Hoewel de aangeduide techniek van stapsgewijze verfijning pas volledig tot zijn recht komt bij het ontwikkelen van een omvangrijk programma, zullen we hier, terwille van de beknoptheid waartoe de bescheiden omvang van deze beschouwing en het beperkte geduld van de lezer uitnodigen, een vrij eenvoudig voorbeeld behandelen.
We stellen ons een lijst voor waarop we in de loop van een aantal jaren de namen van enige honderden mensen hebben opgeschreven. De namen staan niet in alfabetische volgorde, en het is dan ook wel voorgekomen dat een naam die niet snel vindbaar was nogmaals is opgenomen. We willen nu een nieuwe, alfabetische lijst van deze namen aanleggen, waarin elke naam natuurlijk maar één keer mag voorkomen. We zullen hiervoor een programma schrijven dat door een menselijke uitvoerder kan worden gevolgd die gewapend is met gum (alles is met potlood opgeschreven), een potlood, papier en een vinger. Andere veronderstellingen blijven impliciet, of worden bij het ontwikkelen van het programma genoemd.
In eerste benadering kan het programma er zo uitzien:

A0:
Leg in *nieuwe lijst* een alfabetische lijst aan van de namen die in *oude lijst* voorkomen, zonder dezelfde naam meer dan eens op te nemen.

We nemen aan dat deze opdracht niet tot het repertoire van de uitvoerder behoort, zodat we nader moeten aangeven hoe deze algoritme moet worden uitgevoerd. We doen dat door de volgende verfijning van algoritme A0.

A1:
maak *nieuwe lijst* leeg;
while *oude lijst* is niet leeg
do *laagste*:= de alfabetisch eerste uit *oude lijst*;
    gum *laagste* uit *oude lijst* weg;
    if *laagste* komt niet in *nieuwe lijst* voor *then*
        voeg *laagste* aan *nieuwe lijst* toe
    fi
od

We nemen aan dat de opdrachten in A1 wel tot het repertoire van onze uitvoerder behoren, behalve

laagste:= de alfabetisch eerste uit *oude lijst*

en

gum *laagste* uit *oude lijst* weg.

Voor het verfijnen van deze laatste opdracht zijn drie strategieën denkbaar:

1. De *oude lijst* nog eens doorlopen totdat een naam wordt aangetroffen die gelijk is aan *laagste*, en deze dan uitgummen.
2. Bij het zoeken van *laagste* bijhouden waar deze wordt aangetroffen, en deze later, zonder noodzaak tot zoeken, uitgummen.
3. Bij het zoeken van *laagste* deze uitgummen zodra hij wordt aangetroffen.

Omdat mogelijkheid 1 veel werk eist van de uitvoerder, en we aannemen dat de uitvoerder voor mogelijkheid 2 niet is toegerust, zullen we proberen mogelijkheid 3 aan te grijpen: het uitgummen integreren in het zoekproces. Onmiddellijk duikt nu een moeilijkheid op: van een naam die kandidaat is om de alfabetisch eerste te zijn, kan niet onmiddellijk vastgesteld worden of hij inderdaad de eerste is. De oplossing is: we laten elke keer de nieuwe kandidaat-eerste uitgummen, en op de vrijgekomen plek de oude kandidaat-eerste neerschrijven. De laatste kandidaat-eerste (de echte eerste) wordt zo wel uitgegumd, maar niet weer opgeschreven, zodat hij inderdaad uit *oude lijst* verwijderd is.

De verfijning van het stukje algoritme

laagste:= de alfabetisch eerste uit *oude lijst;*
gum *laagste* uit *oude lijst* weg

uit A1 luidt nu:

A2.1:
zet *vinger* direct na eerste naam van *oude lijst;*
laagste:= naam voor *vinger;*
gum de naam voor *vinger* weg;
while er staat een naam na *vinger*
do deze naam:= naam na *vinger;*
  if deze naam komt alfabetisch eerder dan *laagste then*
    gum de naam na *vinger* weg en schrijf *laagste* op deze plaats;
    laagste:= deze naam
  fi;
  zet *vinger* direct na naam na *vinger*
od

In A1 kan nog de regel

if laagste komt niet in *nieuwe lijst* voor *then*

verfijnd worden tot:

A2.2:
if nieuwe lijst is leeg of anders
laagste ≠ laatste naam van *nieuwe lijst then*

Hiermee is A1 voldoende verfijnd, maar in A2.1 komt de regel

if deze naam komt alfabetisch eerder dan *laagste then*

voor. Hier wordt van de uitvoerder gevergd dat hij niet alleen van twee letters kan bepalen welke eerder komt in het alfabet, maar ook kan vaststellen welke van twee woorden alfabetisch, of liever lexicografisch, het eerste komt. Het is duidelijk dat we deze laatste taak kunnen beschrijven in termen van de eerste. We kunnen bovenstaande regel uit A2.1 verfijnen tot

A3.1:
woord1:= deze naam; woord2 := laagste;
letter 1:= eerste letter van *woord1;*
woord1:= woord1 minus de eerste letter;
letter 2:= eerste letter van *woord2;*
woord2:= woord2 minus de eerste letter;
while letter1 = letter2 en
zowel *woord1* als *woord2* bevat nog letters
do letter1:= eerste letter van *woord1;*
woord1:= woord1 minus de eerste letter;
letter2:= eerste letter van *woord2;*
woord2:= woord2 minus de eerste letter
od;
if letter1 alfabetisch eerder dan *letter2* of
woord1 heeft geen letters meer maar *woord2* wel *then*

In figuur 1 zien we het resultaat als we deze verfijning A3.1 inpassen in niveau 2, en niveau 2 weer in niveau 1, en het zo ontstane niveau 1 op zijn beurt in niveau 0.
Bij het doorlezen van dit resultaat zal menige lezer even de verdenking hebben dat hier meer gedaan is dan het eenvoudig "in elkaar schuiven" van

```
maak nieuwe lijst leeg;
while oude lijst is niet leeg
do zet vinger direct na eerste naam van oude lijst;
   gum de naam voor vinger weg;
   while er staat een naam na vinger
   do deze naam:= naam na vinger;
      woord1:= deze naam; woord2:= laagste;
      letter1:= eerste letter van woord1;
      woord1:= woord1 minus de eerste letter;
      letter2:= eerste letter van woord2;
      woord2:= woord2 minus de eerste letter;
      while letter1 = letter2 en
            zowel woord1 als woord2 bevat nog letters
      do letter1:= eerste letter van woord1;
         woord1:= woord1  minus de eerste letter;
         letter2:= eerste letter van woord2;
         woord2:= woord2 minus de eerste letter
      od;
      if letter1 alfabetisch eerder dan letter2 of
         woord1 heeft geen letters meer maar woord2 wel then
         gum de naam na vinger weg en schrijf laagste op deze plaats;
         laagste:= deze naam
      fi;
      zet vinger direct na naam na vinger
   od;
   if nieuwe lijst is leeg of anders
      laagste ≠ laatste naam van nieuwe lijst then
      voeg laagste aan nieuwe lijst toe
      fi
   od
```

Fig. 1. Resulterend programma voor het aanleggen van een enkelvoudige, alfabetische nieuwe lijst van namen die voorkomen in *oude lijst*.

de boom van sub-algoritmen die al ontwikkeld waren. Het lijkt alsof er nog heel wat zorg besteed is aan het op de juiste plek invoegen van de gewenste handeling en aan het vermijden van ongewenste overgangen. Maar dit systeem van programma-ontwikkeling is juist gevolgd ter vermijding van deze "nazorg", die bij het rechttoe-rechtaan programmeren zo op de voorgrond treedt. Het is bij deze aanpak niet nodig voortdurend de grote lijn in het oog te houden. Elk van de sub-algoritmen hoeft alleen de taak uit te voeren die de ermee corresponderende opdracht op hoger niveau voor zijn rekening nam. Hoe zo'n taak nauwkeurig gedefinieerd kan worden laat de volgende paragraaf zien.

4. ASSERTIES

De allereerste, triviale, stap in de ontwikkeling van het voorbeeld in paragraaf 3 luidde

A0:
Leg in *nieuwe lijst* een alfabetische lijst aan van de namen die in *oude lijst* voorkomen, zonder dezelfde naam meer dan eens op te nemen

Deze beschrijving is niet erg nauwkeurig. Er staat weinig meer dan los het probleem op. Hoe dit moet gebeuren wordt in de verfijningen aangegeven; in A0 zelf wordt alleen aangeduid wat er gedaan moet worden. Omdat dit een belangrijke kwestie is, loont het de moeite een manier aan te geven om systematisch te beschrijven wat een (stuk) algoritme moet doen.
Voor de algoritme A0 kan dat zo:

{*oude lijst* = oorspronkelijke lijst}
leg in *nieuwe lijst* een alfabetische lijst aan van de namen die in *oude lijst* voorkomen, zonder dezelfde naam meer dan eens te nemen
{alle namen uit de oorspronkelijke lijst komen voor in *nieuwe lijst* en vice versa, *nieuwe lijst* is alfabetisch geordend, geen namen dubbel in *nieuwe lijst*}

Tussen accolades zijn nu beweringen (asserties) geplaatst die voor resp. na uitvoering van de algoritme gelden. De algoritme ontleent nu zijn betekenis aan deze asserties: de verfijning is alleen correct als hij de eerste bewering in de tweede "overvoert".
We zullen A0 nog eenmaal herschrijven, en daarbij enige predicaten gebruiken als afkortingen van de hierboven gebruikte asserties:

VV (*verz*) = alle namen uit de oorspronkelijke lijst komen in de verzameling *verz* voor, en vice versa

ALFENK = *nieuwe lijst* is alfabetisch geordend en geen namen dubbel in *nieuwe lijst*.

We hebben nu

{VV (*oude lijst*)}
A0
{VV (*nieuwe lijst*), ALFENK }

In de algoritme A1, die een eerste verfijning is van A0, kunnen we nu ook asserties plaatsen die een precieze betekenis aan de opdrachten geven. We zullen daarbij de relatie alfabetisch eerder noteren met <. Deze relatie geldt tussen twee verzamelingen als elk woord uit de ene verzameling alfabetisch eerder komt dan alle woorden uit de andere verzameling. Als ook gelijke woorden in beide verzamelingen kunnen voorkomen schrijven we ≤.

A1:
{VV (*oude lijst*)}
maak *nieuwe lijst* leeg;
{VV (*nieuwe lijst* ∪ *oude lijst*), ALFENK, *nieuwe lijst* ≤ *oude lijst*}
while NOG OUD: *oude lijst* is niet leeg
do {VV (*nieuwe lijst* ∪ *oude lijst*), ALFENK, *nieuwe lijst* ≤ *oude lijst*,
NOG OUD}
laagste:= de alfabetisch eerste uit *oude lijst;*
gum *laagste* uit *oude lijst* weg;
{VV (*nieuwe lijst* ∪ *laagste* ∪ *oude lijst*), ALFENK,
nieuwe lijst ≤ *laagste* ≤ *oude lijst*}
if laagste komt niet voor in *nieuwe lijst* *then*
voeg *laagste* aan *nieuwe lijst* toe
fi
{VV (*nieuwe lijst* ∪ *oude lijst*), ALFENK, *nieuwe lijst* ≤ *oude lijst*}
od
{VV (*nieuwe lijst* ∪ *oude lijst*), ALFENK, *nieuwe lijst* ≤ *oude lijst*,
¬NOG OUD}
{VV (*nieuwe lijst*), ALFENK}

{VV* (*oude lijst*}, NOG OUD}
zet *vinger* direct na eerste naam van *oude lijst;*
{VV* (*oude lijst*), NOG OUD
oude lijst = namen voor *vinger* ∪ namen na *vinger*}
laagste:= naam voor *vinger;*
{VV* (*laagste* ∪ namen na *vinger*), NOG OUD,
oude lijst = namen voor *vinger* ∪ namen na *vinger*}
gum de naam voor *vinger* weg;
{VV* (*laagste* ∪ *oude lijst*), *laagste* ≤ namen voor *vinger*,
oude lijst = namen voor *vinger* ∪ namen na *vinger*}
while er staat een naam na *vinger*
do {VV* (*laagste* ∪ *oude lijst*), er staat een naam na *vinger*,
laagste ≤ namen voor *vinger*}
deze naam := naam na *vinger;*
{VV* (*laagste* ∪ *oude lijst*), deze naam staat na *vinger*,
laagste ≤ namen voor *vinger*}
if deze naam komt alfabetisch eerder dan *laagste then*
{VV* (*laagste* ∪ *oude lijst*), *deze naam* staat na *vinger*,
deze naam < *laagste* ≤ namen voor *vinger*}
gum de naam na *vinger* weg en schrijf *laagste* op deze plaats;
{VV* (*deze naam* ∪ *oude lijst*), *laagste* staat na *vinger*,
deze naam < *laagste* ≤ namen voor *vinger*}
laagste:= deze naam
{VV* (*laagste* ∪ *oude lijst*), er staat een naam na *vinger*,
laagste < naam na *vinger* ≤ namen voor *vinger*}
fi;
{VV* (*laagste* ∪ *oude lijst*), er staat een naam na *vinger*,
laagste ≤ naam na *vinger*, *laagste* ≤ namen voor *vinger*}
zet *vinger* direct na naam na *vinger*
{VV* (*laagste* ∪ *oude lijst*), *laagste* ≤ namen voor *vinger*}
od
{VV* (*laagste* ∪ *oude lijst*), *laagste* ≤ namen voor *vinger*,
er staat geen naam na *vinger*,
oude lijst = namen voor *vinger* ∪ namen na *vinger*}
{VV* (*laagste* ∪ *oude lijst*), *laagste* ≤ *oude lijst*}

Fig.2. Algoritme A2.1 met asserties.

Als elke opdracht in deze algoritme inderdaad de assertie ervoor overvoert in de assertie erna, dan volgt de correctheid van de totale algoritme direct uit Floyd's inductiestelling voor stroomdiagrammen (FLOYD [10]) en de daarop gebaseerde regel van Hoare voor de while-statement (DE BAKKER [1]). (De lezer ga dit na.) Doordat de begin- en de eind-assertie gelijk zijn aan die van A0, is A1 inderdaad een correcte verfijning van A0.

In figuur 2 is de verfijning A2.1 uit de vorige paragraaf met asserties aangevuld. Het hierin gebruikt predicaat VV* wordt gedefinieerd door

VV* (*verz*) = in de verzameling *verz* komen alle namen voor die in *oude lijst* voorkomen op het moment dat A2.1 uitgevoerd gaat worden, en vice versa

Merk op dat aan alle asserties binnen de do-loop eigenlijk nog moet worden toegevoegd:

oude lijst = namen voor *vinger* ∪ namen na *vinger*

Ook in A2.1 kan de lezer verifieren dat de asserties een consistent geheel vormen (b.v. rond *if* ... *fi* en *do* ... *od*) en dat uit de correctheid van de tussen-asserties inderdaad volgt dat de begin-assertie wordt overgevoerd in de eind-assertie:

{VV* (*oude lijst*), NOG OUD}

↓A2.1

{VV* (*laagste* ∪ *oude lijst*), *laagste* ≤ *oude lijst*}

We zien dat deze begin- en eindasserties niet gelijk zijn aan die voor en na de opdracht waarvan A2.1 een verfijning is. Toch kunnen we uit het feit dat A2.1 *nieuwe lijst* niet verandert (er zelfs niet naar kijkt) concluderen dat de overgang tussen de gewenste begin- en eindassertie bereikt wordt:

{VV (*nieuwe lijst* ∪ *oude lijst*), ALFENK, *nieuwe lijst* ≤ *oude lijst*, NOG OUD}
{VV* (*oude lijst*), NOG OUD}

↓A2.1

{VV* (*laagste* ∪ *oude lijst*), *laagste* ≤ *oude lijst*}
{VV (*nieuwe lijst* ∪ *laagste* ∪ *oude lijst*), ALFENK, *nieuwe lijst* ≤ *laagste* ≤ *oude lijst*}

De lezer gaat gemakkelijk na dat de eerste en de laatste overgang in dit schema inderdaad logisch volgen.

Het is wel duidelijk geworden dat het opstellen van een algoritme compleet met asserties een taai werkje is: de asserties zijn vrij lang, en steeds vrijwel gelijk aan de voorafgaande (zelfs als handigheden worden toegepast). De omvang van het karwei kan heel goed verkleind worden door alleen asserties te plaatsen op cruciale punten: voor en na een nog te verfijnen stukje, en bij een loop. Verder zou het nuttig zijn als er een techniek ontwikkeld werd die de vele herhalingen in de asserties voorkomt.
We zullen de asserties in de overige verfijningen achterwege laten. De lezer zal inzien dat de gebruikte methode ook daar opgaat, hoewel zich bij de algoritme die bepaalt of een woord lexicografisch eerder komt dan een ander, de opmerkelijke situatie voordoet, dat de voor de hand liggende beschrijving van de eisen die aan die algoritme gesteld worden zelf de vorm van een algoritme heeft.

5. PROGRAMMEREN AAN DE HAND VAN ASSERTIES

In de vorige paragraaf hebben we gezien hoe de betekenis van een (stukje) algoritme gedefinieerd kan worden met behulp van een tweetal asserties: een pre-conditie en een post-conditie. De algoritme A met de betekenis {p} A {q} kan omschreven worden als: voer p in q over. Bij het verfijnen van A, b.v. tot A1; A2, zal de taak van A1 en A2 zijn: voer p in r over en voer r in q over, voor een of andere assertie r. De opgave een algoritme {p} A {q} te ontwikkelen is dan gesplitst in twee hopelijk eenvoudiger opgaven, nl. algoritmen {p} A1 {r} en {r} A2 {q} te ontwikkelen. De taak waarvoor de programmeur hier staat is een geschikte r te zoeken waardoor de twee deelproblemen inderdaad eenvoudiger zijn. Bij voorkeur moet r "tussen p en q in" liggen (, maar het is gemakkelijk in te zien dat de simpele keuze r = p ∧ q of r = p ∨ q zinloos is).
Deze verfijningsmogelijkheid kan worden uitgedrukt in de bewijsregel:

$$\frac{\{p\}\ A1\ \{r\} \qquad \{r\}\ A2\ \{q\}}{\{p\}\ A1;\ A2\ \{q\}}$$

Zo kunnen we ook beschikken over de bewijsregels

$$\frac{\{p \wedge b\}\ A1\ \{q\} \qquad \{p \wedge \neg b\}\ A2\ \{q\}}{\{p\}\ \underline{\textit{if}}\ b\ \underline{\textit{then}}\ A1\ \underline{\textit{else}}\ A2\ \underline{\textit{fi}}\ \{q\}}$$

en

$$\frac{\{p \wedge b\}\ A1\ \{p\}}{\{p\}\ \underline{while}\ b\ \underline{do}\ A1\ \underline{od}\ \{p \wedge \neg b\}}$$

De eerste regel zegt dat het probleem {p} A {q} ook gesplitst mag worden in de twee deelproblemen {p ∧ b} A1 {q} en {p ∧¬b} A2 {q}, hetgeen bij geschikt gekozen b inderdaad vereenvoudiging kan opleveren. Er geldt namelijk: hoe sterker de pre-conditie (en hoe zwakker de post-conditie), hoe gemakkelijker de overgang. Aangezien p ∧ b en p ∧¬b inderdaad beide sterkere asserties zijn dan p (want uit p ∧ b volgt wel p, maar niet omgekeerd), vindt het programmeer-analogon van deze regel, n.l. de *if...then...else*-constructie veel emplooi. (Ga na wat het voordeel is van de keuze q voor b.)
De laatste regel geeft de mogelijkheid {p} A {p ∧¬b} te vereenvoudigen tot {p ∧ b} A1 {p}, en het zal nu duidelijk zijn waarom deze zo belangrijk is: tegelijk wordt de pre-conditie versterkt en de post-conditie verzwakt. (Er schuilt wel een addertje onder het gras: als A1 niet een werkelijke stap zet in de richting van het *false* worden van b, zal de zo geconstrueerde algoritme A niet eindigen.)
Op deze wijze wordt het programmeerprobleem {p} A {q} opgevat als het probleem om te bewijzen: ∃A: {p} A {q}. Door het bestaan van A met behulp van de bewijsregels aan te tonen, construeren we tegelijk de algoritme. Op het onderste niveau zijn de bewijsregels niet meer nodig en kunnen we volstaan met het toepassen van "axioma's". Gevallen hiervan zijn {p} A {q} waarbij p → q - dan kan voor A de "dummy statement" gekozen worden -, of waarbij p uit q verkregen kan worden door in q voor een variabele v systematisch een expressie e te substitueren - dan kan voor A de toekenning $v := e$ gekozen worden.
Voorbeelden: $\{x > 1 \wedge x = y\}$ A $\{x \geq 1 \wedge x = y\}$ met als oplossing voor A de dummy statement, en $\{x > a + 1\}$ A $\{x > a + 2\}$ met als oplossing $x := x + 1$.
We willen nu aan de hand van een voorbeeld laten zien dat het mogelijk is een algoritme stapsgewijze te ontwikkelen in termen van asserties (waarna de geschikte opdrachten tussen de asserties geplaatst kunnen worden) in plaats van aan de hand van opdrachten (waarna de geschikte asserties tussen de opdrachten geplaatst kunnen worden).
Als voorbeeld nemen we weer de alfabetisch te ordenen lijst. We moeten van VV (*oude lijst*) geraken tot VV (*nieuwe lijst*), ALFENK:

{VV (*oude lijst*)} → {VV (*nieuwe lijst*), ALFENK}

Als tussen-assertie proberen we VV (*oude lijst*), ALFENK:

{VV (*oude lijst*)} → {VV (*oude lijst*), ALFENK} → {VV (*nieuwe lijst*), ALFENK}

Om de eerste overgang simpeler te maken, realiseren we ons dat geldt: de lege lijst is alfabetisch geordend en bevat geen namen dubbel, zodat deze eerste overgang kan worden gedaan d.m.v. de toekenning

nieuwe lijst:= de *lege lijst*.

Rest nu nog de overgang

{*nieuwe lijst* is leeg, VV (*oude lijst*), ALFENK} → {VV (*nieuwe lijst*), ALFENK}

Als we de eerste assertie iets verzwakken en de tweede iets versterken, ontstaat (in de eerder gebruikte notatie):

{VV (*nieuwe lijst* ∪ *oude lijst*), ALFENK}
→ { NOG OUD, VV (*nieuwe lijst* ∪ *oude lijst*), ALFENK}

Dit is een fraai voorbeeld van {p} A {p ∧¬b}, hetgeen we met de while-bewijsregel te lijf kunnen. We moeten dan deze overgang realiseren:

{NOG OUD, VV (*nieuwe lijst* ∪ *oude lijst*), ALFENK}
→ {VV (*nieuwe lijst* ∪ *oude lijst*), ALFENK},

maar dan wel met een stap in de richting van het *false* worden van NOG OUD. Het is duidelijk dat er een element moet worden weggehaald uit *oude lijst*, dat dan, zonodig, aan *nieuwe lijst* wordt toegevoegd om VV (*nieuwe lijst* ∪ *oude lijst*) te handhaven. We krijgen dus te "bewijzen":

{NOG OUD, VV (*nieuwe lijst* ∪ *oude lijst*), ALFENK}
→ {VV (*nieuwe lijst* ∪ *element* ∪ *oude lijst*), ALFENK}
→ {VV (*nieuwe lijst* ∪ *oude lijst*), ALFENK}

Aangezien de laatste stap ingewikkelder is dan de eerste (het toevoegen van *element* aan *nieuwe lijst* dreigt de alfabetische volgorde te verstoren) richten we daarop eerst de aandacht. Wat we missen is de assertie *nieuwe lijst* ≤ *element*. De vraag is nu of we de overgang kunnen maken:

{NOG OUD, VV (*nieuwe lijst* ∪ *oude lijst*), ALFENK} →
{VV (*nieuwe lijst* ∪ *element* ∪ *oude lijst*), ALFENK, *nieuwe lijst* ≤ *element*}

We ontmoeten nu het probleem dat wellicht geen enkel element uit *oude lijst* aan de laatste voorwaarde voldoet. We hebben daarom nog iets extra nodig om deze overgang mogelijk te maken. De keuze die we maken is te zorgen dat steeds geldt *nieuwe lijst* ≤ *oude lijst*. We voegen deze assertie toe aan de asserties die we al hadden (gelukkig geldt de assertie voor een lege *nieuwe lijst*, zodat we in het begin geen extra moeilijkheden krijgen). De stand

van het probleem is nu:

{NOG OUD, VV (*nieuwe lijst* ∪ *oude lijst*), ALFENK, *nieuwe lijst* ≤ *oude lijst*}
→ {VV (*nieuwe lijst* ∪ *element* ∪ *oude lijst*), ALFENK,
nieuwe lijst ≤ *element* ∪ *oude lijst*}
→ {VV (*nieuwe lijst* ∪ *oude lijst*), ALFENK, *nieuwe lijst* ≤ *oude lijst*}

Voor de laatste stap is het weer wenselijk dat *element* ≤ *oude lijst*, dus het te kiezen element moet het laagste (alfabetisch eerste) zijn uit *oude lijst*. Dus:

{NOG OUD, VV (*nieuwe lijst* ∪ *oude lijst*), ALFENK, *nieuwe lijst* ≤ *oude lijst*}
→ {VV (*nieuwe lijst* ∪ *laagste* ∪ *oude lijst*), ALFENK,
nieuwe lijst ≤ *laagste* ≤ *oude lijst*}
→ {VV (*nieuwe lijst* ∪ *oude lijst*), ALFENK, *nieuwe lijst* ≤ *oude lijst*}

Tot zover dit voorbeeld van gestructureerd programmeren aan de hand van asserties; deze aanzet moge voldoende zijn om het idee te schetsen. Bij de beoordeling van de practische toepasbaarheid van deze methode, moet men wel bedenken dat de vertrouwdheid met de gebruikelijke methode van programmeren het oordeel gemakkelijk zal beïnvloeden. Het is denkbaar dat gelijke oefening in deze vorm van programmeren ook gelijke kunst zal baren.
Een paar opmerkingen:

- In de praktijk worden wiskundige stellingen in grote en informele stappen bewezen. Wellicht zijn er hier mogelijkheden om door een geschikte notatie en door het automatisch invullen van routine-zaken, het zuiver administratieve werk belangrijk te verminderen.
- Bij het bewijs van een stelling worden andere stellingen gebruikt. Iets overeenkomstig is hier het gebruik van procedures uit een bibliotheek, maar zo'n stelling zou ook de automatische generatie van een stuk programma kunnen inhouden dat aan bepaalde specificaties voldoet.

6. GOED PROGRAMMEREN

In de literatuur zijn verschillende termen in omloop voor het soort programmeertechnieken dat in het alfabetiseer-voorbeeld van paragraaf 3 gebruikt is. Tussen "structured programming" (DIJKSTRA [7]), "programming by action clusters" (NAUR [17]), "stepwise refinement" (WIRTH [25]) en "systema-

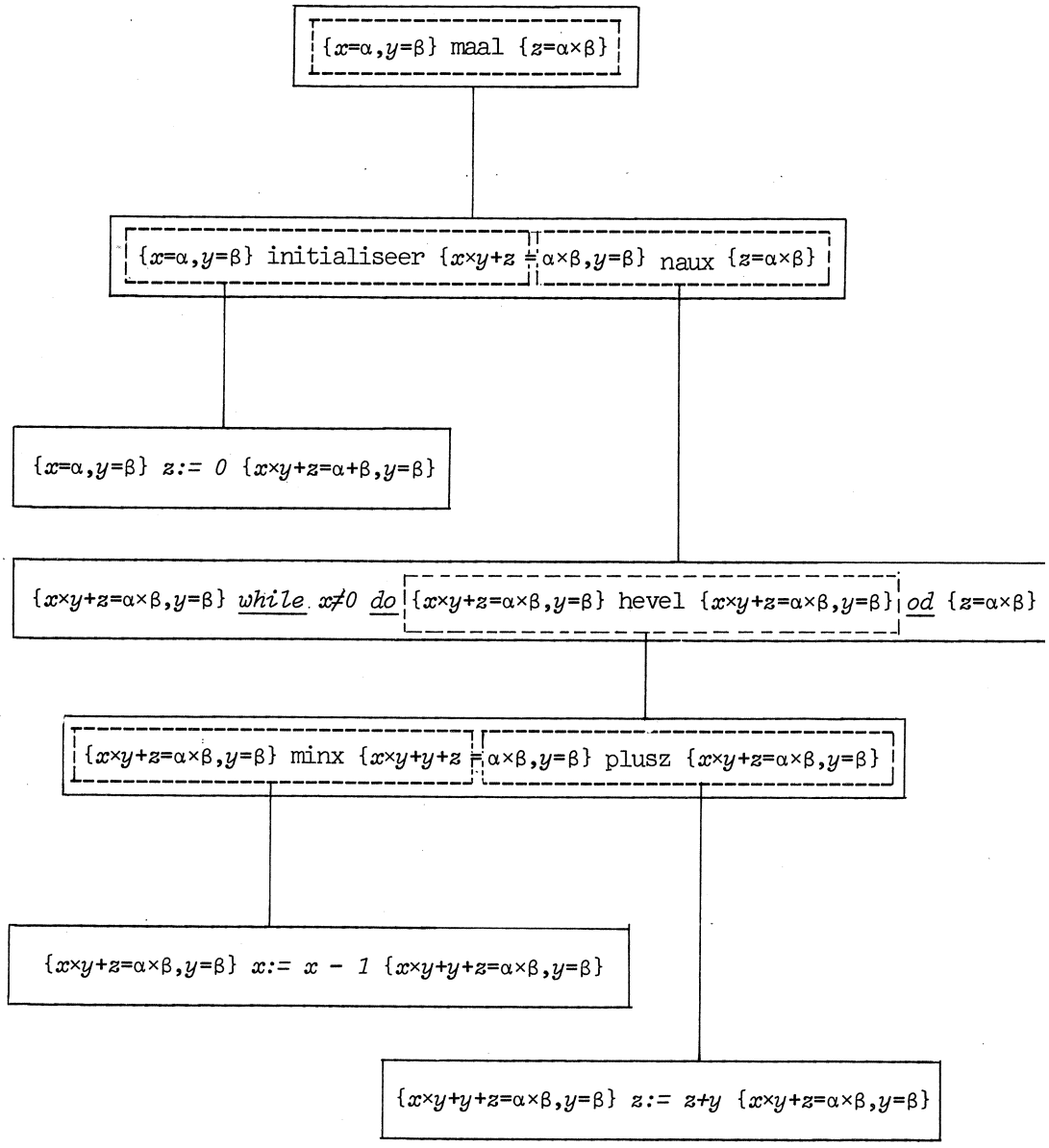


Fig.3. Een complete boom bij de programmatekst

z:= 0; while x ≠ 0 do x:= x - 1; z:= z + y od

tisches Programmieren" (WIRTH [26]) bestaan wel verschillen, maar die worden niet door alle auteurs even sterk benadrukt (vgl. ook LEDGARD [15]). We zullen in deze paragraaf aanwijzingen tot goed programmeren geven die een grootste gemene deler vormen van de bestaande technieken, plus een keuze uit de specifieke eigenschappen van enkele der technieken, plus enige extra wenken.

1. Een programma moet beschouwd worden als een boom van algoritmen. De wortel van deze boom is de algoritme los het probleem op, voorafgegaan door een pre-conditie die het gegevene representeert en gevolgd door een post-conditie die het verlangde resultaat uitdrukt. De pre- en post-condities van de andere knopen in de boom hangen af van de asserties in hun moederknoop op de wijze die geïllustreerd is in figuur 3. (In dat voorbeeld worden alleen acties verfijnd, maar de detaillering van condities als in *if* ... *then* en *while* ... *do* kunnen op soort gelijke wijze in de boom gepast worden.)
2. De stukken algoritme die niet verder worden uitgewerkt worden geheel in de eigenlijke programmeertaal geschreven; in de andere stukken worden vrijelijk andere uitdrukkingsmiddelen gebruikt.
3. De nog te verfijnen stukken van een algoritme in de boom worden tijdens het ontwikkelen van het programma voorzien van een pre- en een post-conditie.
4. De algoritme worden in beginsel ontwikkeld in volgorde van boven naar beneden in de boom.
5. Ontwerpbeslissingen moeten zo lang mogelijk worden uitgesteld: geen beslissingen over details boven in de boom. Dit geldt met name voor de keuze van data structures.
6. Data structures moeten parallel met de acties verfijnd worden. (Vgl. het alfabetiseer-voorbeeld: de lijsten werden hoog in de boom als arrays van namen opgevat, later werd elke naam als een array van letters beschouwd.)
7. Het kan nuttig zijn op zeker moment de top-down volgorde te onderbreken voor het schrijven van algoritmen aan de uiteinden onder in de boom. Dit geldt met name voor algoritmen die data structures moeten manipuleren die vreemd zijn aan de gebruikte programmeertaal.
8. Laat de linkerhand niet weten wat de rechter doet: beslissingen die mogelijk later herroepen worden (en dat zijn er vele) moeten binnen één algoritme blijven (information hiding, vgl. PARNAS [19]). Anders gezegd: houd zoveel mogelijk lokaal, b.v.

a. Gebruik nergens in het programma getalrepresentaties, behalve in initialisaties. Een besluit achteraf om het programma op een 8×8 i.p.v. een 10×10 bord te laten dammen kan dan geëffectueerd worden door verandering op één plek in het programma.
b. Vermijd het gebruik van globale variabelen (WULF & SHAW [27]).
c. Data structures waarvan de opbouw misschien nog veranderd zal worden moeten niet rechtstreeks, maar via vaste procedures gemanipuleerd worden. Dan hoeft op andere plaatsen in de boom slechts in minimale mate gerekend te worden op de eigenschappen van de data structures.

Een belangrijke toepassing van het lokaliteitsbeginsel is ook:

9. Vermijd goto's, zeker over grotere afstand. In de methode van gestructureerd programmeren is voor goto nauwelijks plaats, maar waar de gebruikte programmeertaal het vermijden moeilijk maakt, dient het gebruik van sprongen tot bepaalde vaste schema's beperkt te blijven. Over de wenselijkheid en mogelijkheid goto's te vermijden en dergelijke aangelegenheden zie BÖHM & JACOPINI [2], VAN DE RIET [20], WEGNER [20 : 23] en de in VAN DE RIET [20] vermelde literatuur.
10. Plaats ook asserties in de stukken programma die in de eigenlijke programmeertaal geschreven zijn. Tezamen met de pre- en post-condities genoemd in punt 3 wordt zo het bewijs van correctheid (behalve terminatie) van het gehele programma geleverd. Het is natuurlijk niet noodzakelijk tussen elk tweetal opdrachten een volledige assertie te plaatsen, zoals in het algemeen in een bewijs de kleine stapjes worden weggelaten.
11. Als na completering van het programma nog wijzigingen nodig zijn, dan moeten die niet eenvoudig worden aangebracht in de uiteindelijke programmatekst, maar in de hoogste algoritme in de boom waarop deze verandering effekt heeft, en vandaar naar beneden. Zo kan tegelijk het correctheidsbewijs aangepast worden, en kan worden voorkomen dat verbetering op de ene plaats tot een fout op een andere plaats leidt.
12. Het aan de computer aan te bieden programma bestaat uit de algoritme die verkregen wordt door op elk niveau de niet in de programmeertaal gestelde stukken te vervangen door de ervoor in de plaats geschreven sub-algoritmen van het lagere niveau. Vaak zal het nuttig zijn deze substitutie niet overal door te voeren, maar op sommige plaatsen een procedure-aanroep te plaatsen, en de sub-algoritme de vorm van een procedure te geven. Dit geldt met name voor sub-algoritmen die meer dan eens in de boom voorkomen, en het is zelfs noodzakelijk voor sub-algoritmen die in hun eigen verfijning voorkomen (recursie).

13. Hoewel de afleidingsboom van de programmatekst als documentatie bewaard kan blijven, is het nuttig de mogelijkheden te gebruiken die de programmeertaal biedt om die boom in de uiteindelijke programmatekst vast te leggen. Hierbij valt te denken aan asserties en benamingen van hogere algoritmen in commentaar, en aan het kiezen van een geschikte lay-out. (De afleidingsboom kan beschouwd worden als een haakjes-structuur in de programmatekst, en zo'n haakjes-structuur kan d.m.v. inspringen aangegeven worden.)

7. WENSEN

Bestaande programmeertalen lenen zich in wisselende mate tot gestructureerd programmeren. We hoeven niet eens zover te gaan als FORTRAN om ongewenste toestanden in dit opzicht tegen te komen. Ook een heel fatsoenlijke taal als ALGOL 68 laat nog wensen onvervuld. In het algemeen kan gezegd worden dat de mogelijkheden van bestaande talen t.a.v. data structures te beperkt zijn, en die t.a.v. control mechanismen (goto etc.) te ruim. Het zou ook wenselijk zijn als i.p.v. normale block-structuur een gegeneraliseerd soort afscherming van delen van algoritmen beschikbaar zou zijn. Bij normale block-structuur is het b.v. niet mogelijk twee blokken die in andere opzichten geheel zelfstandig zijn, de beschikking te geven tot een gemeenschappelijke variabele waartoe andere programmadelen geen toegang hebben. Het zou nuttig zijn als toegang tot een variabele ook nog gekwalificeerd kan worden: alleen inspectie, of ook recht tot toekenning.
Er zijn ook terreinen waarop bestaande talen helemaal niet thuis geven. Zo is het onmogelijk om asserties een meer dan commentariërende functie in een programmatekst toe te bedelen, hetgeen het gebruik van asserties om de correctheid van het programma te garanderen niet aanmoedigt. Ook is het nauwelijks mogelijk het top-down programmeerproces direkt in een programmeertaal toe te passen: dit moet op "kladpapier" gebeuren, waarna tenslotte een "echt" programma gedestilleerd wordt.
Hierdoor gaat de structuur van de verschillende abstractieniveaus in het uiteindelijke programma verloren: er bestaan dan ook geen gestructureerde programma's, maar alleen gestructureerd geprogrammeerde programma's. Het is wenselijk dat toekomstige programmeertalen mogelijkheden zullen bieden om de structuur, die conceptueel een belangrijk aspect van het programma is, als onvervreemdbaar deel in de programmatekst op te nemen.

De taal ALEPH (BOSCH, GRUNE & MEERTENS [3], GRUNE [13]) geeft voor bepaalde terreinen van programmeren het goede voorbeeld.
Maar ook al zouden er uitstekende instrumenten voor gestructureerd programmeren voorhanden zijn, ze zouden vrijwel ongebruikt blijven zolang het programmeeronderricht bijna steeds neerkomt op:

"1. worstelen met de faciliteiten van een zeker rekencentrum,
2. het inprenten van de details en eigenaardigheden van een zekere programmeertaal, meestal Fortran of Algol,
3. het construeren van een algoritme voor een paar proefproblemen."
(WIRTH [24] in 1970)

Het programmeeronderwijs zou van onderwijs in een programmeertaal moeten worden het onderwijs in programmeren. Sinds 1970 lijkt er op dit terrein weinig veranderd, als dit tenminste mag worden afgemeten aan het aantal gepubliceerde cursussen volgens deze aanpak (GEURTS [11], WIRTH [26]).

LITERATUUR

[1] BAKKER, J.W. DE, *Inleiding bewijsmethoden*, Colloquium programmacorrectheid, Math. Centrum, 1974.

[2] BÖHM, C. & G. JACOPINI, *Flow diagrams, Turing machines and languages with only two formation rules*, CACM 9 (1966).

[3] BOSCH., R., D. GRUNE & L. MEERTENS, *ALEPH, A Language Encouraging Program Hierarchy*, Math. Centrum, 1973.

[4] BRUNO, J. & K. STEIGLITZ, *The expression of algorithms by charts*, Algorithm specification, Ed. Randall Rustin, 1972.

[5] DIJKSTRA, E.W., *A constructive approach to the problem of program correctness*, BIT 8 (1968) 174-186.

[6] DIJKSTRA, E.W., *The structure of "THE"-multiprogramming system*, CACM 11 (1968) 341-346.

[7] DIJKSTRA, E.W., *Notes on structured programming*, EWD 249, TH Eindhoven, 1969.

[8] DIJKSTRA, E.W., *A short introduction to the art of programming*, EWD 316, TH Eindhoven, 1971.

[9] DIJKSTRA, E.W., *Betrouwbaarheid van programma's*, Abstracte informatica, vakantiecursus 1973, Math. Centrum.

[10] FLOYD, R.W., *Assigning meaning to programs*, Proc. of Symposia in Applied Math., Vol. 19, Mathematical aspects of computer science, American Math. Soc., 1967.

[11] GEURTS, L., *Cursus programmeren, deel 1*, De elementen van het programmeren, Math. Centrum, 1973.

[12] GEURTS, L., *Cursus programmeren, deel 2*, De programmeertaal ALGOL 60, Math. Centrum, 1973.

[13] GRUNE, D., *ALEPH, een grammaticale aanpak van programmacorrectheid*, Colloquium programmacorrectheid, Math. Centrum, 1974.

[14] HENDERSON, P. & R. SNOWDON, *An experiment in structured programming*, BIT 12 (1972) 38-53.

[15] LEDGARD, H.F., *The case for structured programming*, BIT 13 (1973) 45-57.

[16] MILLS, H.D., *Mathematical foundations for structured programming*, IBM, Gaithersburg, Md., 1972.

[17] NAUR, P., *Programming by action clusters*, BIT 9 (1969) 250-258.

[18] NAUR, P., *An experiment on program development*, BIT 12 (1972) 347-365.

[19] PARNAS, D., *On the criteria to be used in decomposing programs into modules*, CACM 15 (1972) 1053-1058.

[20] RIET, R.P. VAN DE, *Over goto's en programmacorrectheid*, Colloquium programmacorrectheid, Math. Centrum, 1974.

[21] WEGNER, E., *A hierarchy of control structures*, Machine oriented languages bulletin 1, 1972.

[22] WEGNER, E., *Tree-structured programs*, Machine oriented languages bulletin 2, 1973.

[23] WEGNER, E., *Tree-structured programs*, T.U. Berlin, 1973.

[24] WIRTH, N., *Programming and programming languages*, Proc. Int. Computing Symposium, Bonn, 1970.

[25] WIRTH, N., *Program development by stepwise refinement*, CACM 14 (1971) 221-227.

[26] WIRTH, N., *Systematisch Programmieren*, Teubner, 1972.

[27] WULF, W. & M. SHAW, *Global variable considered harmful*, Sigplan notices 8 (1972) 28-34.

[28] Structured Programming, Academic Press, 1972:

DIJKSTRA, E.W., *Notes on structured programming.*

HOARE, C.A.R., *Notes on data structuring.*

DAHL, O.J. & C.A.R. HOARE, *Hierarchical program structures.*

MATHEMATICAL CENTRE SYLLABUS 21, 1975, IV, 69-77.

ALEPH, EEN GRAMMATICALE AANPAK VAN PROGRAMMACORRECTHEID

D. GRUNE

1. ACHTERGRONDEN

Het is ongelooflijk hoe slecht er geprogrammeerd wordt. Het is ongelooflijk dat het in de software-branche een volkomen normale toestand is dat een contract niet wordt nagekomen. Wanneer we bij een ingenieursbureau een brug bestellen, kunnen we er zeker van zijn dat het resultaat nog vele jaren het landschap zal sieren; wanneer we bij een software-bureau een programma bestellen, kunnen we er even zeker van zijn dat het resultaat nog vele jaren de bureaus van vele systeem-programmeurs zal ontsieren.
Het is de taak van software-engineering de oorzaken van deze absurde toestand te analyseren en tegenmaatregelen te ontwerpen. Enige literatuur hierover is te vinden in BAUER [2], DAHL, DIJKSTRA, HOARE [5], en WIRTH [12, 13]. Hierna volgen enige gedachten over dit onderwerp welke in wisselwerking ontstaan zijn tijdens de ontwikkeling van ALEPH.
Wanneer een menselijke spreker (of schrijver) aan een menselijke toehoorder (of lezer) een mededeling doet, wordt deze mededeling door de toehoorder onbewust onmiddellijk aan een redelijkheidscontrole onderworpen, gebaseerd op een uitgebreide kennis van de omgeving. Wanneer bijvoorbeeld de nieuwslezer beweert dat de wereldbevolking de drie en een kwart biljoen heeft bereikt, denkt de luisteraar dat iemand zich vergist heeft, niet dat de bevolkingsexplosie uit de hand is gelopen. Omgekeerd maakt deze meegaandheid en dit begripsvermogen van de toehoorder het de spreker mogelijk, met een paar woorden ingewikkelde dingen te zeggen of wensen op zeer slordige wijze te formuleren. De vraag aan de winkeljuffrouw bij de bakker: "Mag ik nog zo'n taart met perziken die U gisteren ook had?" levert meestal het gewenste resultaat ook al was het een abrikozentaart en was het gisteren zondag. Ons hele denken en onze hele communicatie is gebaseerd op het feit dat we met een redelijke partner te maken hebben, die ongeveer evenveel weet als wijzelf. We zijn gewend dat wanneer we per ongeluk iets onzinnigs zeggen, we óf toch wel begrepen worden, óf dat iemand

wel komt vragen wat we bedoelen. (U hebt deze laatste zin best begrepen, maar als U erover gaat denken, zult U zien dat dat eigenlijk helemaal niet logisch is). Beter dan "bijna juist" worden onze mededelingen nooit. We zien dat onze hele dagelijkse ervaring met communicatie (onder andere) berust op twee steunpunten: de redelijkheids-controle ("Dat kunnen ze niet bedoeld hebben!") en de terugkoppeling ("Kom je morgen om acht uur?". "Bedoel je acht uur 's morgens of 's avonds?").
Bij de communicatie met een computer ontbreken deze twee steunpunten volkomen. We kunnen dan ook niet verwachten dat de communicatiemethoden die we dagelijks gebruiken (of een geformaliseerd afgietsel daarvan) ook voor de computer kunnen dienen: een computer werkt niet op een handvol "bijna juiste" opdrachten. Integendeel, we verwachten dat een goed computer-communicatiemiddel (een computertaal) sterk afwijkt van een natuurlijke taal en we beschouwen de eigenschap van een computertaal dat hij goed aansluit bij het dagelijks denken, niet als een pluspunt. Immers, de natuurlijke taal is een middel om op efficiënte wijze "bijna juiste" (maar in de praktijk voldoend nauwkeurige) opdrachten te produceren. Van een goede computertaal verwachten we echter dat hij in zich de methoden heeft om "geheel juiste" opdrachten te produceren, al zal dat ten koste gaan van de efficiëntie van deze opdrachtenproductie.
Alvorens in te gaan op deze "methoden" wil ik een paar woorden wijden aan de bestaande grote programmeertalen. FORTRAN, PL/I en in mindere mate ALGOL 60 zijn bewust of onbewust, ontworpen met de bedoeling nauw aan te sluiten bij de dagelijkse uitdrukkingsmiddelen. Dit, samen met het feit dat er uitstekende compilers voor bestaan, verklaart mijns inziens een groot deel van het enorme succes van FORTRAN. Volledig in overeenstemming met het bovenstaande is FORTRAN een voortreffelijk middel om op efficiënte wijze bijna correcte programma's te produceren. Dit is niet denigrerend bedoeld: vele programma's worden voor een heel speciaal doel geschreven, éénmaal gebruikt, en de resultaten worden op redelijkheid getoetst.(Voorbeeld: door een misverstand zijn in 4000 kaarten met getallen alle voorkomende nullen als o's geponst; gevraagd een programma om de schade te herstellen). In dergelijke gevallen, waarbij op korte termijn een éénmalig, niet-kritisch resultaat verlangd wordt (en dat zijn precies de omstandigheden waaronder onze dagelijkse communicatie functioneert!), kunnen talen als FORTRAN uitstekende diensten verlenen; de meerprijs van een volledig correct programma is dan te hoog. Voor gebruiks- en service-

programma's is deze meerprijs echter niet te hoog.

2. DE METHODEN

Een goede programmeertaal dient methoden te bevatten die ten eerste met redelijke geestelijke inspanning te hanteren zijn en ten tweede gemakkelijk leiden tot volledig correcte oplossingen. ALEPH is gebaseerd op twee zulke methoden die in de praktijk hun waarde bewezen hebben:

1e. het specificeren van alternatieven met verschillende toepasbaarheidsvoorwaarden ("toegangssleutels"), en
2e. het ontbinden van een probleem in deelproblemen (die gelijkvormig mogen zijn met het oorspronkelijke probleem).

De tweede methode staat algemeen bekend als "hiërarchisch programmeren", "top-down aanpak", "verdeel- en- heerstechniek", enz. Hierop is in de derde lezing in dit colloquium uitgebreid ingegaan.
De eerste methode heeft geen algemene naam. Hij wordt in de bestaande literatuur ingevoerd in de gedaante van een if statement, case statement of iets dergelijks en als een natuurlijk gegeven beschouwd. Hij is voor ALEPH even essentieël als de top-down techniek.
De praktisch ingestelde lezer, en met hem een ieder die wel eens een top-down programma, geschreven in ALGOL 60 of PL/I, gedraaid heeft, zal sceptisch staan tegenover de praktische toepasbaarheid van de top-down techniek. We zullen verderop zien dat deze bezwaren hun oorzaak vinden in de gebruikte ALGOL- en PL/I- compilers en niet in de top-down techniek.
Een klein voorbeeld van de toepassing van deze methoden is misschien nuttig. Voor het opzoeken van een naam in een alfabetische lijst hebben we volgende basis-algoritme:

de naam N opzoeken in een lijst:
    alternatief 1: de lijst is leeg (sleutel), in welk geval passend gereageerd wordt.
    alternatief 2: de lijst is niet leeg (sleutel), in welk geval we het probleem ontbinden in de twee volgende deelproblemen:
        probleem 2.1: neem een naam uit de lijst en noem hem M (kan altijd want de lijst was niet leeg).
        probleem 2.2: dit probleem bestaat uit 3 alternatieven:
            alternatief 2.2.1: de naam N komt alfabetisch vóór de naam M (sleutel);

probleem: de naam N opzoeken in de linker deellijst
alternatief 2.2.2: de naam N ís de naam M (sleutel); we hebben de naam gevonden.
alternatief 2.2.3: de naam N komt alfabetisch ná de naam M (sleutel);
probleem: de naam N opzoeken in de rechter deellijst, een probleem dat gelijkvormig is met het oorspronkelijke probleem.

We kunnen dit heel wat beknopter formuleren als we alternatieven scheiden door puntkomma's en deelproblemen door komma's:

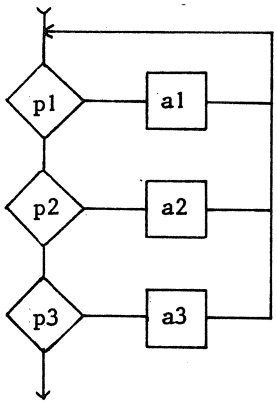
```
zoek N in L:
     L is leeg, naam niet gevonden;
     L is niet leeg, neem M uit L,
          (N links van M, zoek N in linker deel van L;
           N is M, naam gevonden;
           N rechts van M, zoek N in rechter deel van L).
```

We zien hier duidelijk hoe een probleem wordt gesplitst in alternatieven, hoe alternatieven worden ontbonden in deelproblemen (ook de toegangssleutel is een deelprobleem!), hoe deelproblemen op hun beurt weer worden gesplitst in alternatieven, enz. enz., tot uiteindelijk alle deelproblemen óf een standaardoplossing hebben (assignment, test op gelijkheid, en dergelijke) óf gelijkvormig zijn met een reeds geanalyseerd probleem (eventueel via recursie).
Een zo geformuleerde algoritme moet dus als volgt uitgevoerd worden. Eerst worden de sleutels van de gegeven alternatieven getest (in ALEPH gebeurt dat in de volgorde waarin ze neergeschreven staan, zodat eigenlijk de sleutels "L is niet leeg" en "N rechts van M" overbodig zijn). Dan worden de deelproblemen van het alternatief met de passende sleutel van links naar rechts uitgevoerd (de werkelijkheid in ALEPH is iets ingewikkelder omdat in beperkte mate na de sleutel nog andere sleutels mogen volgen).
We trekken hieruit de zeer belangrijke conclusie dat er maar één manier is om een gegeven deelprobleem D in een alternatief A te bereiken: geen der sleutels van alternatieven die voorafgaan aan A mag gepast hebben en de sleutel van A moet gepast hebben. Verder zijn alle deelproblemen in A, die voorafgaan aan D, opgelost. Deze regel bewijst grote diensten bij het af-

leiden van eigenschappen van de algoritme, zowel met de hand als mechanisch.

3. DE GRAMMATICALE FORMULERING

De bovenstaande formulering van "zoek N in L" vertoont sterke verwantschap met die van een "regel" uit een formele grammatica, en wel een context-gevoelige grammatica als we ook de parameters N en L beschouwen, en een context-vrije grammatica wanneer we deze buiten beschouwing laten. Deze formulering zullen we dan verder ook aanduiden als de grammaticale formulering van een algoritme.
De grammaticale formulering is minstens even machtig als de bekende if, case, for, while en repeat statements. Het bewijs hiervan is niet moeilijk: de semantiek van deze statements kan eenvoudig grammaticaal geformuleerd worden. In vele gevallen leidt de grammaticale formulering tot inzichtelijker resultaten. Beschouwen we de flow-chart:



die zeker een "nette" flow-chart genoemd kan worden.
In ALEPH is hij te formuleren als:

```
A: (p1,a1;p2,a2;p3,a3), A; +.
```

waarin het hele stuk tussen haken als sleutel dient van het eerste alternatief van A: het tweede alternatief is leeg. Een formulering met een while of repeat statement is ingewikkelder.
De grammaticale formulering heeft vele praktische en theoretische voordelen; drie ervan zullen we hier nader belichten.
Ten eerste is er veel bekend over grammatica's (AHO & ULLMAN [1], CHOMSKY [4], GINSBURG [8], en KNUTH [10] enz.) en over methoden om eigenschappen van grammatica's te bepalen. Vele van deze methoden kunnen gebruikt worden

om grammaticaal geformuleerde algoritmen (dat zijn ALEPH-programma's) te controleren en te analyseren. Vanuit computer-standpunt is ALEPH een zeer eenvoudige taal, die zich uitstekend leent voor mechanische manipulatie, implementatie en optimalisatie. Zo is de algoritme "zoek N in L" recursief; echter, vanuit grammaticaal standpunt is "zoek N in L" rechts-recursief, d.w.z. dat wanneer alle regels in de grammatica die geen rol spelen in de recursiviteit van "zoek N in L" als terminaal worden beschouwd, de overblijvende grammatica finite-state is. Voor implementatie is dan helemaal geen stack nodig. In meer praktische termen gesteld: na de uitvoering van de recursieve aanroep "zoek N in linker (rechter) deel van L" doen we niets meer met de stack-inhoud van "zoek N in L" zodat we deze dan ook wel vóór de recursieve aanroep hadden mogen weggooien. De stack-inhoud van de recursieve aanroep komt dan over die van "zoek N in L" heen, en er is geen enkele reden meer om de informatie van "zoek N in L" überhaupt op de stack te leggen: "zoek N in L" is niet recursief.
De grammaticale formulering maakt dergelijke optimalisatie-overwegingen gemakkelijk; de resultaten kunnen eenvoudig in de compiler geïmplementeerd worden. Een andere overweging is dat voor een ALEPH-programma geen display op de stack nodig is: de stack, indien nodig, bevat de parameters, de localen en het terugkeeradres. Het zijn deze overwegingen die de top-down techniek praktisch mogelijk maken. Een in ALGOL geschreven top-down programma wordt vertaald door een ALGOL-vertaler, waarvan onmogelijk verwacht kan worden dat hij de bovengenoemde optimalisaties aanbrengt: ten eerste zouden ze maar in een heel beperkt aantal gevallen effectief zijn, en ten tweede zouden ze zo goed als altijd onverantwoord zijn omdat er in een ALGOL-programma zoveel mechanisch onnaspeurbare dingen kunnen gebeuren. We zien hier dat een ALEPH-programma de geneesmiddelen in zich draagt voor zijn eigen (schijnbare) inefficiëntie: de hoge mechanische analyseerbaarheid maakt optimalisaties mogelijk die een ander systeem niet kan bieden.

Het tweede voordeel van de grammaticale formulering is gelegen in de hiërarchische structuur die een programma er door krijgt. Vandaar de naam ALEPH, A Language Encouraging Program Hierarchy.
Wanneer we een sorteer-algoritme formuleren als:

```
sorteer:
    is al gesorteerd;
```

splits in twee delen, sorteer linker deel,
sorteer rechter deel, meng.

dan hebben we daarmee een familie van algoritmen van het type "sorteer" beschreven, waaruit door specificatie van de deelproblemen op een lager niveau nog één bepaald algoritme geselecteerd moet worden. De lezer kan voor zichzelf vele sorteer-algoritmen selecteren door zich b.v. af te vragen:

a. Als "meng" leeg is, waaraan moet "splits in twee delen" dan voldoen? Kan dat zo dat "sorteer linker deel" triviaal wordt ("sorteer" is dan rechts-recursief)? Kunnen we ook op niet-triviale wijze splitsen [9]? Kan dat beter? [6].

b. Als "meng" niet leeg is, hoe kunnen we dan splitsen? Kan "sorteer linker (rechter) deel" triviaal zijn?

Dat deze programma-hiërarchie een leidraad kan vormen bij het ontwerpen van algoritmen wordt o.a. aangetoond in GEURTS [7], en in de vele lezingen van prof.dr. E.W. DIJKSTRA.

Het derde grote voordeel van de grammaticale formulering is de mogelijkheid de dynamische gang van zaken in het programma tot op aanzienlijke diepte statisch te analyseren. Van deze mogelijkheid wordt gebruik gemaakt om statisch (d.w.z. tijdens compilatie) bepaalde aspecten van de dynamische correctheid na te gaan. De volgende twee voorbeelden zullen dit verduidelijken.

Hoewel ze tot nog toe niet ter sprake zijn gekomen, bestaan er in ALEPH variabelen, en wel voornamelijk als localen van een "regel". Ze kunnen alleen als parameters aan andere "regels" meegegeven worden. Van parameters is bekend of ze als input-, output- of in/out- parameters gebruikt worden (ook dat wordt statisch gecontroleerd). Nu hebben variabelen de onaangename eigenschap dat hun waarde wel eens ongedefinieerd is; de toepassing van zo'n variabele als input-parameter levert dan ook zeer ongedefinieerde resultaten. Deze situatie wordt in ALEPH met zekerheid ontdekt tijdens compilatie: aangezien we de status (wel of niet gedefinieerde waarde) van alle parameters en localen aan het begin van een "regel" weten, aangezien we uit de parameter-beschrijvingen van de regels voor de deelproblemen én de status vóór zo'n deelprobleem de status ná dat deelprobleem kunnen bepalen, en aangezien er maar één manier is om een bepaald punt in een

"regel" te bereiken, kunnen we de status van alle parameters en alle localen op elk punt in de regel bepalen, en nagaan of ergens de waarde wordt gebruikt van een variabele die nog ongedefinieerd is. Op deze manier worden vele fouten in het programma gevonden die in de meeste andere systemen pas tijdens run-time (of helemaal niet) aan het licht komen BOSCH, GRUNE & MEERTENS [3]. Het tweede voorbeeld van een statische check op dynamische correctheid behelst een meer specifiek ALEPH probleem. Het selecteren van alternatieven door middel van sleutels werkt alleen goed, wanneer het proberen van een foute sleutel geen neven-effecten heeft ("Vragen staat vrij"). We stellen dus als eis dat een "regel" die, als sleutel gebruikt, faalt, geen neveneffecten mag hebben. Ook deze eis wordt statisch gecontroleerd en wel als volgt. Van elke "regel" is bekend of hij neven-effecten heeft als hij slaagt. Binnen elke regel wordt nu, op een wijze analoog aan die in het vorige voorbeeld, een gebied aangegeven waarbinnen de "regel" nog kan falen: binnen dit gebied mag geen "regel" met neven-effecten gebruikt worden. Door een test hierop worden bepaalde, vaak subtiele, logische fouten in het programma aan het licht gebracht BOSCH, GRUNE & MEERTENS [3].

4. BESLUIT

Het bovenstaande is een schetsmatige aanduiding van ALEPH in het kader van programmacorrectheid. Voor een korte, meer op de taal gerichte inleiding wordt de lezer verwezen naar BOSCH, GRUNE & MEERTENS [3]; een gedetailleerde beschrijving van ALEPH in de vorm van een Manual zal binnenkort verschijnen. ALEPH is ontstaan als voortzetting van het werk aan CDL (Compiler Description Language) KOSTER [11].

LITERATUUR

[1] AHO, A.V. & J.D. ULLMAN, *Parsing*, Vol. I in: *The Theory of Parsing, Translation and compiling*, Prentice-Hall, Englewood Cliffs N.J., 1972.

[2] BAUER, F.L. (ed.), *Advanced Course on Software Engineering*, Lecture Notes in Economics / Math. Systems 81, Springer Verlag, Berlin, 1973.

[3] BOSCH, R., D. GRUNE & L.G.L.T. MEERTENS, *ALEPH, A Language Encouraging Program Hierarchy*, Mathematical Centre Report IW 9/73, Amsterdam, 1973.

[4] CHOMSKY, N., *On Certain Formal Properties of Grammars*, Information and Control 2 (1959) 137-167.

[5] DAHL, O.J., DIJKSTRA E.W. & C.A.R. HOARE, *Structured Programming*, APIC Studies in Data Processing 8, Academic Press, London, 1972.

[6] EMDEN, M.H. VAN, *Increasing the Efficiency of Quicksort*, Comm. ACM 13 (1970) 563-567, 693-694.

[7] GEURTS, L., *Cursus Programmeren, Deel 1 en 2*, MC Syllabus 16.1 en 16.2, Mathematisch Centrum, Amsterdam, 1973.

[8] GINSBURG, S., *The Mathematical Theory of Context-free Languages*, McGraw-Hill, New York, 1966.

[9] HOARE, C.A.R., *Quicksort*, Comput. J. 5 (1962) 10-15.

[10] KNUTH, D.E., *On the translation of languages from left to right*, Information and Control 8 (1965) 607-639.

[11] KOSTER, C.H.A., *A Compiler Compiler*, Mathematical Centre Report MR 127/71, Amsterdam, 1971.

[12] WIRTH, N., *Program development by stepwise refinement*, Comm. ACM 14 (1971) 221-227.

[13] WIRTH, N., *Systematisches Programmieren*, Teubner, Stuttgart, 1972.

MATHEMATICAL CENTRE SYLLABUS 21, 1975, V, 79-92.

EEN CORRECTHEIDSBEWIJS VAN DE SCHORR-WAITE MARKERINGSALGORITME VOOR BINAIRE BOMEN

W.P. de ROEVER

1. INLEIDING

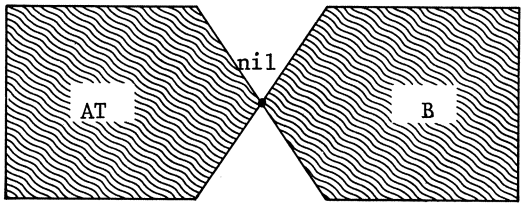
Stelt u zich voor dat u programmeert in een taal en met een systeem dat (binaire) lijststructuren manipuleert. Aangezien nieuw gegenereerde structuren naar reeds eerder gegenereerde structuren kunnen verwijzen, zullen reeds eerder gegenereerde structuren niet mogen worden overschreven bij het genereren van nieuwe structuren. Dit laatste proces zal moeten plaatsvinden in nog ongebruikte geheugenruimte, indien aanwezig. Indien echter geen vrije geheugenruimte meer beschikbaar is, wil dit niet zeggen dat het programma noodzakelijkerwijze wegens overschrijding van de geheugenruimte moet worden afgebroken: aangezien elke structuur nl. in het geheugen bleef staan, kan er best, wanneer het geheugen opgevuld is, ruimte in beslag genomen worden door structuren waarnaar op geen enkele wijze door het reeds gedeeltelijk geëxecuteerde programma verwezen wordt. Dit houdt in, dat, door alle structuren te markeren waarnaar wel verwezen wordt, de geheugenruimte in beslag genomen door structuren waarnaar niet meer verwezen wordt, kan worden geïdentificeerd en vervolgens gebruikt voor executie van de rest van het programma.
Het onderwerp van deze lezing is nu de correctheid van een algoritme dat binaire lijststructuren markeert zonder dat hiervoor extra geheugenruimte *) nodig is, op drie locaties na, zoals het geval ware geweest wanneer tijdens het markeringsproces een hulpstack zou zijn gebruikt.
Het bovenstaande probleem is voor het eerst door McCARTHY beschreven in [1]. Hij gebruikte echter een hulpstack tijdens het markeerproces, zodat hiervoor ruimte moest worden vrijgehouden, die maximaal wel de helft van de voor structuren gebruikte ruimte kon bedragen.
Deze situatie is door SCHORR & WAITE in [2] verbeterd en door WEGBREIT in [3] geperfectioneerd.

---

*) Wel is voor elk woord een extra markeringsbit nodig.

2. AXIOMATISERING VAN EINDIGE BINAIRE BOMEN

De verzameling van atomen zal met AT en de verzameling van eindige binaire bomen met B worden aangeduid; AT ∩ B = nil, de lege binaire boom.



Voor x ∈ AT geldt: at(x) ≡ true, voor x ∈ B-AT: at(x) ≡ false.
Voor x ∈ AT ∪ B geldt: NIL(x) ≡ true als x = nil, anders is NIL(x) false.
De functies car en cdr zijn *partiële functies* van B naar AT ∪ B; de functie cons is een *totale functie* van (AT∪B) × (AT∪B) naar B.
Deze primitieven voldoen aan de volgende axioma's:

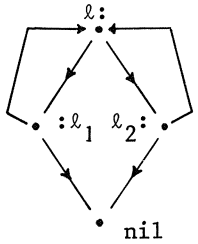
$$\vdash \forall x_1,x_2 \in AT \cup B\ [car(cons(x_1,x_2))=x_1 \wedge cdr(cons(x_1,x_2))=x_2]$$

$$\vdash \forall x \in AT \cup B\ [\neg at(x) \supset cons(car(x),cdr(x))=x]$$

$$\vdash \forall x \in AT \cup B\ [NIL(x) \supset at(x)]$$

$$\vdash NIL(nil).$$

Deze axioma's sluiten echter geen *circulaire* structuren uit.
Voorbeeld:



$$\ell = cons(\ell_1,\ell_2),\ \ell_1 = cons(\ell,nil),\ \ell_2 = cons(nil,\ell).$$

Circulaire structuren worden uitgesloten door gebruik te maken van het feit, dat het successievelijk toepassen van de car en cdr operaties, indien mogelijk, bij circulaire structuren tot een oneindige loop en dus tot een ongedefinieerd resultaat leidt. Zij E gedeclareerd door

```
proc E(x); value(x); E:= if at (x) then x
                         else cons(E(car(x)),E(cdr(x)))        (2.1)
```

dan voëren we als (inductie) axioma in:

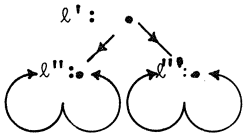
$$\vdash \forall x \in AT \cup B\ [E(x)=x],$$

dwz., $E = Id_{AT \cup B}$.

Voor binaire circulaire structuren geldt dat zij ontwikkeld kunnen worden als oneindige binaire bomen.

Beschouw ℓ, gedef. door $\ell = cons(\ell,\ell)$:

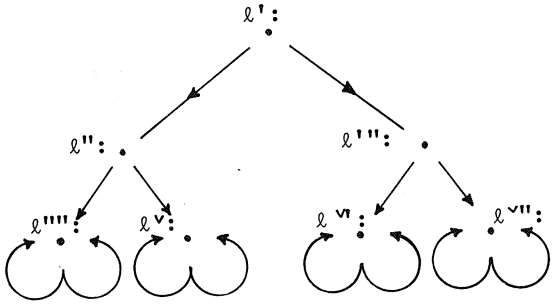
Structuur ℓ representeert hetzelfde partoon als ℓ':



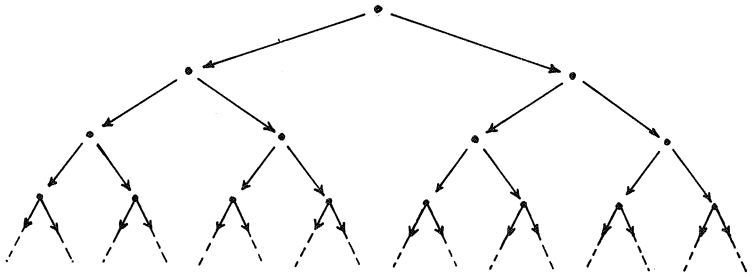
$\ell' = cons(\ell'',\ell''')$, $\ell'' = cons(\ell'',\ell'')$,

$\ell''' = cons(\ell''', \ell''')$

en ℓ' vertegenwoordigt weer hetzelfde patroon als



Door dit ontwikkelproces oneindig vaak te herhalen verkrijgt men een met ℓ equivalente oneindige binaire boom:



3. DE SCHORR-WAITE MARKERINGSALGORITME VOOR BINAIRE BOMEN

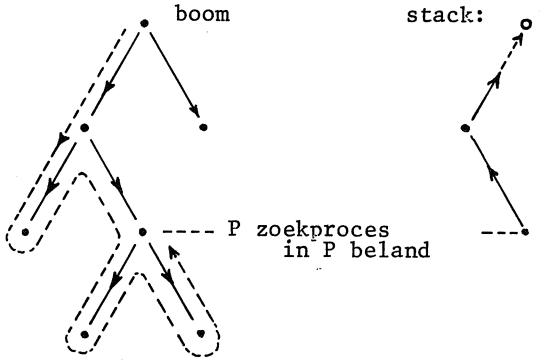
In principe doorloopt de SCHORR-WAITE markeringsalgoritme een eindige binaire boom in *pre-order*.

Deze methode van doorzoeken van een binaire boom laat zich als volgt recursief omschrijven: Het in pre-order doorzoeken van een boom x komt, zolang x géén atoom is, neer op het in pre-order doorzoeken van car(x) gevolgd door het in pre-order doorzoeken van cdr(x), terwijl, als x een atoom is, de zoekalgoritme termineert met het inspecteren van de informatie-inhoud van x.

In figuur:

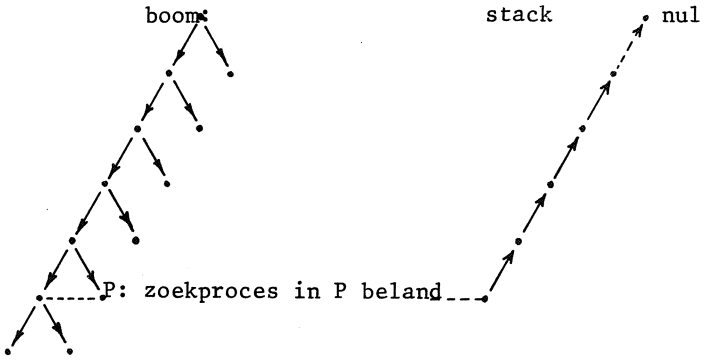
Teneinde een boom zo te doorzoeken moet het zoekproces over een stack met adressen van terugkeerknopen beschikken. Immers, is een linkerdeelboom doorzocht, dus een rechterdeelboom aanwezig, dan zal het zoekproces deze rechterdeelboom moeten bereiken; deze kan bv. via het op een stack liggen van het adres van de "vader" knoop van deze twee deelbomen bereikt worden.

In figuur:

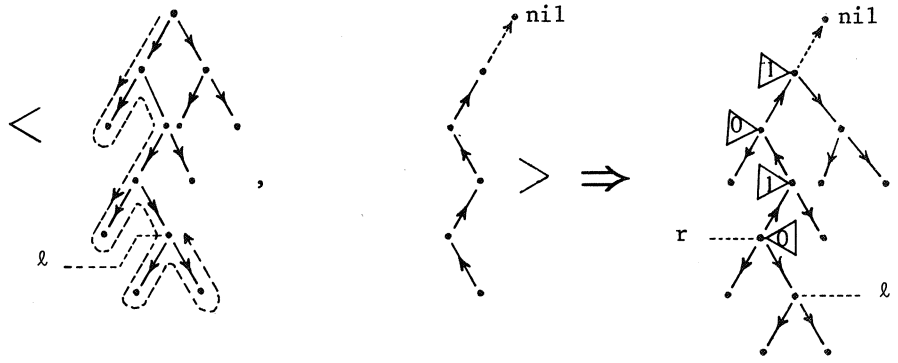


Deze methode van het doorzoeken van een boom met een stack van adressen van terugkeerknopen is in McCARTHY [1] beschreven. Er kleven echter nadelen aan daar in bepaalde gevallen deze stack dezelfde orde van grootte kan aannemen als de te doorzoeken boom!

Voorbeeld:



Deze situatie is door SCHORR & WAITE in [2] verbeterd door de stack met adressen van terugkeerknopen in de binaire boom *zelf* op te nemen. Dit houdt het vernietigen van een gedeelte van de vertakkingsstructuur van de oorspronkelijke boom in. Dat deze naderhand hersteld wordt, is in sectie 4 bewezen.
In figuur:



Hierbij valt het volgende op te merken:

1. De stack met adressen van terugkeerknopen wordt met een nil-knoop beëindigd.
2. Er is een hulppointer r nodig, die steeds naar de knoop boven ℓ wijst, daar er geen ruimte in de (binaire) boom zelf over is om de pointer tussen de deelboom geïdentificeerd door ℓ en zijn vaderknoop op te bergen.
3. Indien het zoekproces belandt in een knoop waarvan één van de pointers

een terugkeerknoop aanwijst, moet deze geïdentificeerd worden. Dit gebeurt met behulp van een markeringsbit. Hierbij is de volgende conventie gebruikt:

a. Valt er nog een rechterdeelboom te doorzoeken, dan staat deze*) in het car-veld, de terugkeerlink in het cdr-veld en het markeringsbit op 1.

b. Is de linkerdeelboom van een knoop reeds doorzocht en is het zoekproces met het doorzoeken van de rechterdeelboom bezig, dan staat de terugkeerlink in het car-veld, de pointer naar de oorspronkelijke linkerdeelboom in het cdr-veld en het markeringsbit op 0.

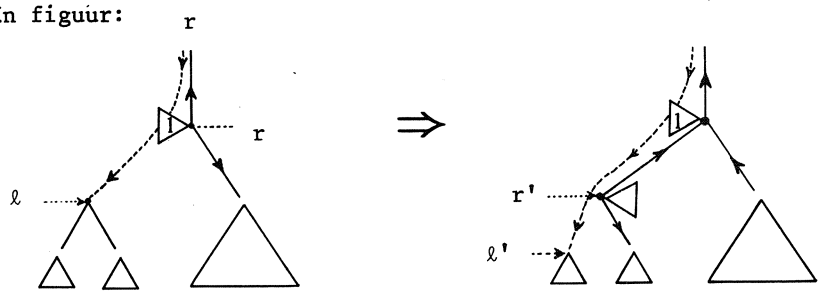
c. Na afloop van het doorzoeken van een (deelboom) wordt het markeringsbit op 1 gezet (de Schorr-Waite algoritme is een markeringsalgeritme!)

In overeenstemming met deze onderverdeling (in punt 3) vallen tijdens het zoekproces de volgende modulen te onderscheiden:

Situatie 3a: Deze correspondeert met het moduul LINKSOMLAAG, gedeclareerd door

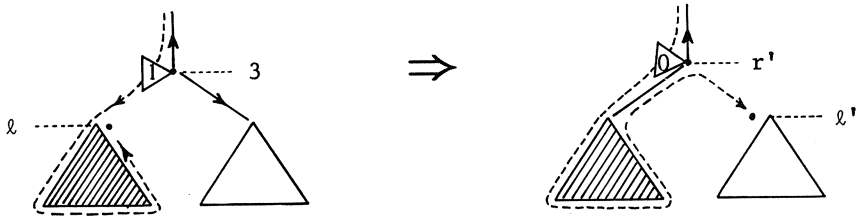
proc LINKSOMLAAG; <ℓ,r> := <car(ℓ),cons(cdr(ℓ),r,1)>

In figuur:



Situatie 3b: Deze correspondeert met het moduul RECHTSOMLAAG, gedeclareerd door

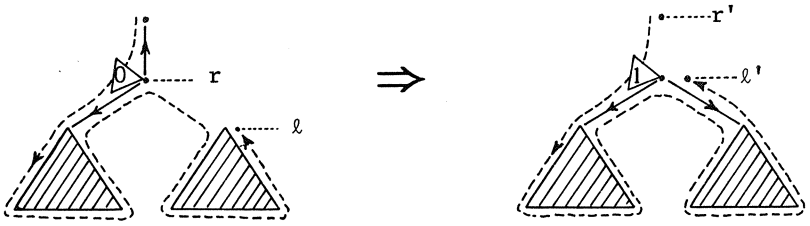
proc RECHTSOMLAAG; <ℓ,r> := <car(r),cons(cdr(r),ℓ,0)>



*) D.w.z., het *adres* van deze

Situatie 3c: Deze correspondeert met het moduul TERUGOMHOOG, gedeclareerd door

```
proc TERUGOMHOOG; <ℓ,r> := <cons(cdr(r),ℓ,1),car(r)>
```



Uit de volgorde waarin de knopen doorzocht moeten worden (in de figuren volgens de gerichte stippellijn) volgt dat deze markeringsalgoritme in de twee onderstaande modulen uiteenvalt:

```
proc LEFT; if at(ℓ) then BACK UP else LINKSOMLAAG; LEFT fi;

proc BACK UP; if NIL(r) then <ℓ,nil>
                  else if m(r) = 1 then RECHTSOMLAAG; LEFT
                       else TERUGOMHOOG; BACK UP fi
```

Hierin identificeert m(x) het markeringsbit van de knoop x. Het zoekproces wordt op gang gebracht door een aanroep van LEFT met (een pointer naar) de te doorzoeken boom, *waarin alle markeringsbits op 0 staan*, in het linker argument,en de lege binaire boom, nil, in het rechter argument - het begin van de stack met adressen van terugkeerknopen.
De beëindiging van het zoekproces wordt vastgesteld wanneer op het einde van de stack met adressen van terugkeerknopen wordt gestuit: d.w.z., het rechterargument voldoet aan NIL.
Samenvattend is deze versie van de Schorr-Waite markeringsalgoritme voor eindige binaire bomen als volgt te declareren:

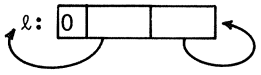
```
proc SCHORR-WAITE(ℓ); value ℓ;
     begin proc LEFT(ℓ,r); value ℓ,r;                                    (3.1)
           LEFT:= if at(ℓ) then BACK UP(ℓ,r) else
                     LEFT(car(ℓ),cons(cdr(ℓ),r,1)) fi;
           proc BACK UP(ℓ,r); value ℓ,r;                                 (3.2)
           BACK UP:= if NIL(r) then <ℓ,r>
                     else if m(r) = 1 then LEFT(car(r),cons(cdr(r),ℓ,0))
                          else BACK UP(cons(cdr(r),ℓ,1),car(r)) fi;
           LEFT(ℓ,nil)
     end
```

Deze algoritme kan worden aangepast aan het doorzoeken van circulaire structuren. Tijdens het zoekproces moet dan worden vastgesteld wanneer dit op een reeds doorlopen knoop stuit. Aangezien het reeds gebruikte markeringsbit tijdens het zoekproces zowel op 0 als op 1 kan staan, kan dit niet dienen om aan te geven of de desbetreffende knoop al dan niet bezocht is. Als oplossing van dit probleem wordt een extra markeringsbit gebruikt in elke knoop. Dit resulteert in de volgende veranderingen in LEFT:

```
proc LEFT; if at(ℓ)∨ bezocht (ℓ) then BACK UP
                   else MARKEERBEZOEKBIT; LINKSOMLAAG; LEFT fi
```

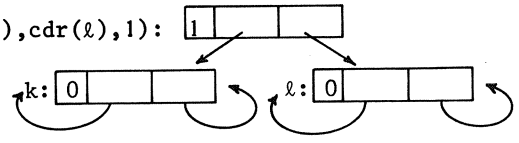
De gebruikte notatie is echter niet geschikt om met behulp van de bouwstenen at, car, cdr *en* cons *het markeren van circulaire datastructuren uit te drukken; dit moet uitgedrukt worden in een meer implementatie-georiënteerde notatie.* Deze moeilijkheid is als volgt in te zien:

Zij ℓ = cons(car(ℓ),cdr(ℓ),0):

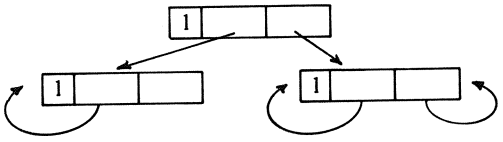


Dan beschrijft cons(car(ℓ),cdr(ℓ),1) *niet* het markeren van de circulaire structuur ℓ, doch resulteert in

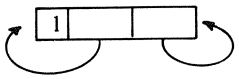
cons(car(ℓ),cdr(ℓ),1):



en *niet* in:



of, daarmee equivalent (vergelijk sectie 2):



Dit is in te zien door op te merken dat vanuit wiskundig standpunt een circulaire structuur equivalent is met een oneindig voortrepeterende boom. Het markeren van die circulaire structuur komt dus neer op het markeren van die oneindig voortrepeterende boom in de repetente knopen.
De hier gebruikte beschrijvingswijze is echter niet geschikt om deze *oneindige* operaties uit te drukken; dit kan wel gebeuren door gebruik te maken van de door SCOTT ontwikkelde theorie van λ-calculus modellen.
Dit probleem laat zich echter omzeilen, als aangegeven op de laatste bladzijde van deze syllabus.

4. EEN CORRECTHEIDSBEWIJS VAN EEN VERSIE VAN DE SCHORR-WAITE MARKERINGS-ALGORITME VOOR BINAIRE BOMEN

De correctheid van de SCHORR-WAITE markeringsalgoritme voor binaire bomen wordt uitgedrukt door

$$\vdash \forall \ell \in AT \cup B\ [\text{unmarked}(\ell) \supset L(\ell,\text{nil}) = \langle M(\ell),\text{nil}\rangle] \qquad (4.1)$$

met unmarked en M gedeclareerd door

```
proc unmarked(x); value x;
     unmarked:= if at(x) then true
                else m(x) = 0 ∧ unmarked(car(x)) ∧ unmarked(cdr(x))    (4.2)
```

```
proc M(x); value x;
     M:= if at(x) then x else cons(M(car(x)),M(cdr(x)),1)              (4.3)
```

en L gedeclareerd als LEFT in (3.1).
Assertie (4.1) wordt bewezen dmv. inductie naar de diepte van boom ℓ. Je komt er echter *niet* door

$$\text{unmarked}(\ell_1) \supset L(\ell_1,\text{nil}) = \langle M(\ell_1),\text{nil}\rangle$$

en

$$\text{unmarked}(\ell_2) \supset L(\ell_2,\text{nil}) = \langle M(\ell_2),\text{nil}\rangle$$

aan te nemen:

$$\text{unmarked}(\text{cons}(\ell_1,\ell_2,0)) \supset L(\text{cons}(\ell_1,\ell_2,0),\text{nil}) = \langle M(\text{cons}(\ell_1,\ell_2,0)),\text{nil}\rangle$$

is nl. niet uit deze aannamen tesamen met unmarked(cons(ℓ_1,ℓ_2,0)) te bewijzen; L(cons(ℓ_1,ℓ_2,0),nil) = (3.1) L(ℓ_1,cons(ℓ_2,nil,1)); op L(ℓ_1,cons(ℓ_2,nil,0)) wil je nu de (inductie) aanname toepassen, terwijl je alleen over een aanname voor L(ℓ_1,r) met r = nil beschikt.
Dien ten gevolge moet je een assertie over L(ℓ,r) bewijzen *voor willekeurige* r. Deze is:

$$\vdash \forall \ell \in AT \cup B,\ r \in B\ [\text{unmarked}(\ell) \supset L(\ell,r) = B(M(\ell),r)] \qquad (4.4)$$

In het bijzondere geval dat r = nil opgaat, volgt uit (4.4)

$$\vdash \forall \ell \in AT \cup B\ [\text{unmarked}(\ell) \supset L(\ell,\text{nil}) = B(M(\ell),\text{nil})]$$

en B(M(ℓ),nil) = (3.2) $\langle M(\ell),\text{nil}\rangle$! (B staat voor Back up).
Met andere woorden, (4.1) volgt uit (4.4).

Neem unmarked(cons(ℓ_1,ℓ_2,0)),
unmarked(ℓ_1) $\supset$ L(ℓ_1,r) = B(M(ℓ_1),r)
en
unmarked(ℓ_2) $\supset$ L(ℓ_2,r) = B(M(ℓ_2),r)
als hypothesen aan.

$$L(\text{cons}(\ell_1,\ell_2,0),r) = (3.1)\ L(\ell_1,\text{cons}(\ell_2,r,1)). \qquad (4.5)$$

Uit (4.2) volgt dat unmarked(ℓ_1) en unmarked(ℓ_2) geldt. (4.6)

Dien ten gevolge is op RHS van (4.5) één der (inductie) hypothesen toepasbaar:

$$L(\ell_1,cons(\ell_2,r,1)) = (hyp.)\ B(M(\ell_1),cons(\ell_2,r,1)). \qquad (4.7)$$

Uit (3.2) volgt

$$B(M(\ell_1),cons(\ell_2,r,1)) = L(\ell_2,cons(r,M(\ell_1),0)). \qquad (4.8)$$

Uit (4.6) volgt dat de inductie hypothese op RHS van (4.8) toe te passen is:

$$L(\ell_2,cons(r,M(\ell_1),0)) = B(M(\ell_2),\ cons(r,M(\ell_1),0)). \qquad (4.9)$$

Uit (3.2) volgt dat

$$B(M(\ell_2),cons(r,M(\ell_1),0)) = B(cons(M(\ell_1),M(\ell_2),1),r). \qquad (4.10)$$

Uit (4.3) volgt dat

$$cons(M(\ell_1),M(\ell_2),1) = M(cons(\ell_1,\ell_2,0)). \qquad (4.11)$$

Uit (4.5) - (4.11) volgt dus

$$L(cons(\ell_1,\ell_2,0),r) = B(M(cons(\ell_1,\ell_2,0)),r)!$$

Indien at(ℓ) geldt, volgt het bewijs van

$$unmarked(\ell) \supset L(\ell,r) = B(M(\ell),r) \qquad (4.12)$$

direct uit (4.2), (3.1) en (4.3).
Hiermee is (4.4) bewezen.

Hoe valt dit bewijs nu enigszins te formaliseren? We weten van E, gedeclareerd in (2.1), dat

$$\vdash\ \forall\ell \in AT \cup B\ [E(x)=x],\ (2.2),$$

geldt. We zullen hiervan gebruik maken door m.b.v. (2.2), (4.4) te laten

volgen uit

$$\vdash \forall \ell \in AT \cup B,\ r \in B\ [\text{unmarked}(E(\ell)) \supset L(E(\ell),r) = B(M(E(\ell)),r)] \qquad (4.13)$$

Assertie (4.13) wordt bewezen m.b.v. inductie naar de recursiediepte van $E(\ell)$. Eerste zal nader aangegeven worden wat

$$\forall x_1,\ldots,x_n\ [f(x_1,\ldots,x_n)=g(x_1,\ldots,x_n)]$$

inhoudt: $f(x_1,\ldots x_n)$ is desd gedefinieerd als $g(x_1,\ldots,x_n)$ gedefinieerd is, in welk geval $f(x_1,\ldots,x_n) = g(x_1,\ldots,x_n)$ geldt; hieruit volgt dat $f(x_1,\ldots,x_n)$ ongedefinieerd is desda $g(x_1,\ldots,x_n)$ dit is.
Declareer E_0 door

proc E_0(x); value x; E_0:= if at(x) then x else begin L: goto L end

en E_{n+1} door

proc E_{n+1}(x); value x; E_{n+1}:= if at(x) then x
else cons(E_n(car(x)),E_n(cdr(x)),m(x)).

Hieruit volgt dat $E_{n+1}(x)$ desd termineert als de recursiediepte van $E_{n+1}(x)$ *kleiner of gelijk* n+1 is; dwz., x is een boom van diepte kleiner of gelijk n+1.
We bewijzen

1. $\forall \ell \in AT \cup B,\ r \in B\ [\text{unmarked}(E_0(\ell)) \supset L(E_0(\ell),r) = B(M(E_0(\ell)),r)]$ (4.14)

en

2. $\forall \ell \in AT \cup B,\ r \in B\ [\text{unmarked}(E_n(\ell)) \supset L(E_n(\ell),r) = B(M(E_n(\ell)),r)] \vdash$
$\forall \ell \in AT \cup B,\ r \in B\ [\text{unmarked}(E_{n+1}(\ell))$
$\supset L(E_{n+1}(\ell),r) = B(M(E_{n+1}(\ell)),r)]$ (4.15)

Dit is om de volgende redenen *voldoende* om (4.13) te bewijzen:
Uit (4.14) volgt, door herhaald toepassen van (4.15), dat geldt

$$\forall k \in N : \vdash \forall \ell \in AT \cup B,\ r \in B\ [\text{unmarked}(E_k(\ell)) \supset \supset L(E_k(\ell),r) = B(M(E_k(\ell)),r)] \qquad (4.16)$$

De aanroep E(ℓ) termineert voor willekeurige ℓ, desda de recursiediepte van E(ℓ) *eindig* is, zeg k; het kan worden geverifieerd m.b.v. de declaraties van E en E_k dat dan E(ℓ) = E_k(ℓ) geldt. Uit (4.16) volgt dan dus dat

$$\forall r \in B\ [\text{unmarked}(E(\ell)) \supset L(E(\ell),r) = B(M(E(\ell)),r)]$$

geldt. Daar ℓ willekeurig gekozen was, volgt (4.13) hieruit.

Er rest on (4.14) en (4.15 te bewijzen; (4.14) volgt uit (4.12) daar E_0(ℓ) termineert desda at(ℓ) geldt en (4.14) volgt uit (4.5) - (4.11) door $\ell_1 = E_n(\text{car}(\ell))$ en $\ell_2 = E_n(\text{cdr}(\ell))$ te nemen, daar

$$E_{n+1}(\ell) = \text{cons}(E_n(\text{car}(\ell)),E_n(\text{cdr}(\ell)),m(x))$$

als ℓ geen atoom is.

Probleem:

Laten het car, cdr en m veld aangegeven worden door functies van *geheugenadressen* naar *geheugenadressen*, atomen of {0,1}.
Laat $\lambda\ .\ t_1,\ldots,t_n\ .\ f(t_1,\ldots,t_n)$ staan voor de waarde die procedure f, hieronder gedeclareerd, vertegenwoordigt:

proc $f(t_1,\ldots,t_n)$; value $t_1,\ldots,t_n$; $f(t_1,\ldots,t_n)$.

Dan valt een gemarkeerde boom te representeren door het 4-tal <ℓ,car,cdr,m>, waarin ℓ voor het adres van de wortel van die boom staat en waarvan ℓ, car en cdr aan het volgende inductieprincipe voldoen: Zij E gedeclareerd door:

```
proc E(ℓ,car,cdr); value ℓ,car,cdr;
     E:= if at(ℓ) then <ℓ,car,cdr>
         else <ℓ,λt. if t=ℓ then car(ℓ) else π2(E(car(ℓ),car,cdr)) fi,
                 λt. if t=ℓ then cdr(ℓ) else π3(E(cdr(ℓ),car,cdr)) fi>,
```

waarin π_i staat voor de projectiefuncties van een 3-tuple op zijn i-de coördinaat, i = 1,2,3. Het inductieprincipe wordt dan gegeven door

$$\boxed{\vdash E(\ell,\text{car},\text{cdr}) = <\ell,\text{car},\text{cdr}>}$$

LEFT en BACK UP worden als volgt gedeclareerd:

```
     proc LEFT(ℓ,r,car,cdr,m); value ℓ,r,car,cdr,m;
          LEFT:= if at(ℓ) then BACK UP(ℓ,r,car,cdr,m) else
LEFT(car(ℓ),ℓ,λt. if t=ℓ then cdr(ℓ) else car(t),
               λt. if t=ℓ then r else cdr(t),λt. if r=ℓ then r else m(t)) fi);

     proc BACK UP(ℓ,r,car,cdr,m); value ℓ,r,car,cdr,m;
          BACK UP:= if NIL(r) then <ℓ,nil,car,cdr,m> else
                    if m(r)=1 then
LEFT(car(r),r,λt. if t=r then cdr(r) else car(t),
               λt. if t=r then ℓ else cdr(t),λt. if t=ℓ then 0 else m(t)),
                  else
BACK UP(r,car(r),λt. if t=r then cdr(r) else car(t),
               λt. if t=r then ℓ else cdr(t),λt. if t=ℓ then 1 else m(t)) fi.
```

Opgave: Bewijs de correctheid van LEFT en BACK UP voor binaire bonen!

Hoogstwaarschijnlijk kan dit bewijs aangepast worden aan een correctheidsbewijs voor een versie van de SCHORR-WAITE markeringsalgoritme, die circulair lijststructuren markeert.

LITERATUUR

[1] McCARTHY, J., *Recursive functions of symbolic expressions and their computation by Machine*, Part I, CACM 3, 184.

[2] SCHORR, H. & W.M. WAITE, *An efficient Machine-Independent procedure for garbage collection in various list structures*, CACM 10, 501-506.

[3] WEGBREIT, B., *A space-efficient list structure tracing algorithm*, IEEE Trans. Comp., 9(Sept. 1972), 1009-1010.

MATHEMATICAL CENTRE SYLLABUS 21, 1975, VI, 93-109.

VAN ABSTRACTE VARIABELE NAAR CONCRETE REPRESENTATIE

L.G.L.T. MEERTENS

1. VARIABELEN EN TOEKENNING

De uitdrukkingsmacht van algoritmen stijgt wezenlijk uit boven die van formules door twee elementen: *repetitie* en *variabelen*. Beide zijn essentieel. Vanuit een meer theoretisch standpunt kan gesteld worden, dat deze elementen niet fundamenteel zijn, maar speciale gevallen van een tweetal fundamenteler principes: (recursieve) *procedures* en *parameters*, waarbij we de overeenkomst hebben

(terug)sprong	-	(recursieve) aanroep
toekenning	-	parameteroverdracht.

Dit kan geïllustreerd worden met de volgende twee stukjes programma:

```
k:= 0;                  proc r = (int k) void:
while k ≤ 9             if k ≤ 9
do print (k);           then print (k);
   k+: =1                    r (k+1)
od                      fi;
                        r (0)
```

Men kan zich afvragen in hoeverre het bestaan van aparte notaties in de meeste programmeertalen voor repetitie en variabelen een relict is uit het machinecodetijdperk, en in hoeverre dit een erkenning inhoudt van de bijzondere rol die beide innemen bij het opstellen van algoritmen (zie ook discussie over "grammaticale formulering" in GRUNE [1]). Een vraag die dan ook rijst, is of de argumentatie tegen de sprong-zonder-restricties niet kan worden uitgebreid tot de toekenning-zonder-restricties (zie b.v. WULF & SHAW [2]). In dit verhaal wordt aan deze vragen voorbijgegaan; wel kan opgemerkt worden dat veel van het betoogde m.b.t. variabelen ook opgaat voor parameters. De sleutel tot het variabele-begrip ligt in de toe-

kenning, waarvan de betekenis kan worden uitgedrukt door de algemene regel

$$\{p[v \leftarrow e]\}\ v := e\ \{p\}.$$

Hierin stelt p een uitspraak voor die al of niet kan gelden, en $p[v \leftarrow e]$ de uitspraak die verkregen wordt door in p voor alle voorkomens van de variabele v de uitdrukking e te substitueren. Zulke in het programma geplaatste uitspraken worden *asserties* genoemd. De regel zegt nu: als we willen bereiken dat (achteraf) de assertie p geldt, dan kan dat door de toekenning v:=e, op voorwaarde dat we v en e zo kiezen dat (vooraf) de assertie $p[v \leftarrow e]$ gold. Deze regel blijkt een machtig hulpmiddel bij het programmeren.
Een voorbeeld: We willen bereiken dat voor een variabele V, die als waarde een verzameling kan aannemen, geldt: $\forall x \in V$: q(x). Aangezien de lege verzameling $\emptyset$ geen elementen bevat, geldt op triviale wijze: $\forall x \in \emptyset$: q(x), wat we ook kunnen schrijven als $(\forall x \in V$: q(x))$[V \leftarrow \emptyset]$. Door de regel

$$\{\forall x \in \emptyset\text{: } q(x)\}\ V := \emptyset\ \{\forall x \in V\text{: } q(x)\}$$

toe te passen, leiden we af dat het doel bereikt wordt met de toekenning $V := \emptyset$.

2. ABSTRACTE VARIABELEN EN CONCRETE REPRESENTATIE

In het bovenstaande komt een variabele V voor waaraan een verzameling kan worden toegekend. Willen we de macht van de toekenningsregel volledig gebruiken, dan moeten we in staat zijn zulke variabelen te hanteren. Nu kennen de meeste -al dan niet hogere- programmeertalen bij hun primitieve waardetypen geen verzamelingen. *) Wie een algoritme wil implementeren waarin verzamelingen een rol spelen, zal zelf een representatie moeten verzinnen, opgebouwd uit de waardetypen die hem wel ter beschikking staan. Dat geldt in het algemeen voor allerlei waardetypen die gebruikt kunnen worden om een algoritme te beschrijven. Zo is het goed mogelijk, dat in een algoritme de toekenning z3:= z1 × z2 voorkomt, waarbij z1, z2 en z3 als waarde een complex getal hebben. In ALGOL 68 kan deze schrijfwijze zonder meer worden overgenomen, maar in een ALGOL-60-programma moet dit anders worden uitgedrukt, bijv. door

*) De programmeertaal PASCAL WIRTH [3] kent "sets", maar deze zijn uitsluitend deelverzamelingen van een tevoren gedefinieerde *eindige* verzameling.

```
re3:= re1 × re2 - im1 × im2;
im3:= re1 × im2 + im1 × re2.
```

Ook zonder de voorafgaande uiteenzetting laat zich raden, dat hier het product van twee complexe getallen wordt berekend. We kunnen stellen: het paar reële variabelen (re3, im3) is een *concrete representatie* van de *abstracte variabele* z3. Het verband wordt gegeven door z = re + i × im, de zogenaamde cartesiaanse representatie. Dit is niet de enige mogelijkheid; een niet ongebruikelijke representatie is de polaire, waarbij z = mod × exp(i × arg). In het ALGOL-60-programma had dan ook heel wel kunnen staan:

```
mod3:= mod1 × mod2;
arg3:= arg1 + arg2.
```

3. GESTRUCTUREERD PROGRAMMEREN

Abstracte variabelen vloeien op natuurlijke wijze voort uit de methode van het *top-down* oftewel *gestructureerd* programmeren. Het wezenlijke van deze methode schuilt in het gebruik maken van een aantal abstractieniveaus. De begrippen "abstract" en "concreet", als boven gehanteerd, moeten als betrekkelijk worden gezien; zo zal een integer, die wij conceptueel als ondeelbaar beschouwen, op een zeer concreet niveau geïmplementeerd zijn met een congolomeraat variabelen die slechts twee waarden kunnen aannemen. Gelukkig hoeven we daarvan geen weet te hebben bij het bedenken van een betere algoritme om getallen in factoren te ontbinden.

Het voordeel van gestructureerd programmeren, toegespitst op variabelen, zal gemakkelijk onderschat worden door degene die niet getracht heeft zijn programmeerarbeid door deze denkdiscipline te laten gidsen. In de abstracte beschrijving van de algoritme kunnen variabelen worden gebezigd voor alle soorten waarden die voor de opsteller van de algoritme een betekenis hebben: waarin hij kan denken. Dit geeft een veel grotere uitdrukkingsvrijheid dan de gangbare programmeertalen bieden, en opent daardoor de mogelijkheid een algoritme op te stellen waarvan de correctheid eenvoudig is in te zien of te bewijzen. Ondertussen heeft de opsteller zich nog in het geheel niet vastgelegd op de concrete representatie; daar heeft hij nog de vrijheid díe representatie te kiezen die hem goed uitkomt, hetzij qua eenvoud van implementatie, hetzij qua efficiëntie van het uiteindelijke

programma. Zo valt de complexe optelling gemakkelijker te formuleren in de cartesiaanse representatie, maar is de polaire representatie handzamer voor de vermenigvuldiging en helemaal voor het worteltrekken. Deze gang van zaken biedt tegelijk het voordeel dat het correctheids-bewijs op geheel analoge wijze kan worden gestructureerd: het correctheids-bewijs van de abstracte algoritme hoeft slechts te worden aangevuld met het bewijs van de correctheid van de concrete representatie. Door deze scheiding in twee onafhankelijk bewijsbare gedeelten blijft de ingewikkeldheid van het gehele bewijs binnen redelijke grenzen. Een uitgewerkt voorbeeld, dat misschien meer overtuigingskracht heeft dan de noodgedwongen eenvoudige illustraties in dit verhaal, is te vinden in HOARE [4]. Voor een verdere uitwerking van de voordelen van deze splitsing in abstractieniveaus raadplege men GEURTS [5].

4. INTERPRETATIE EN ABSTRACTE TOEKENNING

Het verband tussen een abstracte variabele v_a en zijn concrete representatie v_c (in het algemeen een conglomeraat van variabelen) wordt gegeven met behulp van een *interpretatiefunctie* I, en wel door $v_a = I(v_c)$.
(In het voorbeeld van de complexe getallen hebben we z = I(re, im), met I(re, im) = re + i × im.)
Bij het concretiseren van de algoritme moeten alle voorkomens van V_a verdwijnen, ten gunste van V_c. Een eerste stap zal zijn: het systematisch vervangen van V_a door $I(V_c)$, waardoor b.v. if abs V_a > 1 then ... overgaat in if abs $I(V_c)$ > 1 then ... Dit recept faalt echter waar het om toekenningen gaat. Het is niet mogelijk $V_a := e_a$ te concretiseren met $I(V_c) := e_a$, omdat i.h.a. $I(V_c)$ geen variabele is. In plaats daarvan kunnen we proberen $V_a := e_a$ te concretiseren met een toekenning als $V_c := e_c$. De vraag is dan: hoe moeten we e_c kiezen? Hiertoe roepen we de regel $\{p[V_a \leftarrow e_a]\}\ V_a := e_a\ \{p\}$ te hulp: de betekenis van $V_a := e_a$ is het transformeren van $p[V_a \leftarrow e_a]$ tot p. We willen dus door een geschikte keus van een concrete uitdrukking e_c bereiken: als $V_a = I(V_c)$, dan $\{p[v_a \leftarrow e_a]\}\ v_c := e_c\ \{p\}$, oftewel $p[v_a \leftarrow e_a] \Rightarrow p[v_c \leftarrow e_c]$. Aangezien p in abstracte termen gedefinieerd is, moeten we gebruik maken van $v_a = I(v_c)$ om in in p voor v_c een expressie te kunnen substitueren, en wel door p te vervangen door $p[v_a \leftarrow I(v_c)]$. We moeten dan verkrijgen:

$p[v_a \leftarrow e_a] \Rightarrow p[v_a \leftarrow I(v_c)][v_c \leftarrow e_c]$, wat uitgewerkt kan worden tot $p[v_a \leftarrow e_a] \Rightarrow p[v_a \leftarrow I(e_c)]$. Deze implicatie is geldig als we e_c zo kiezen dat $e_a = I(e_c)$.

Dit weinig verrassende resultaat kan ook op een andere wijze worden afgeleid, die een interessante zienswijze biedt. We zouden ons kunnen voorstellen dat bij het concretiseren de concrete representatie niet de abstracte variabele *vervangt*, maar *daarnaast* komt. (Het behoud van de abstracte variabele vormt uiteraard een doublure, en daarom kunnen we *achteraf*, als het programma is opgesteld, alle verwijzingen in de programmatekst naar de abstracte variabele verwijderen.) Hierbij moet de assertie $V_a = I(V_c)$ invariant worden gehouden. Waar de geldigheid van $V_a = I(V_c)$ teloor gaat door een toekenning $V_a := e_a$, moet deze weer hersteld worden; we kunnen proberen hiervoor een toekenning $V_c := e_c$ te gebruiken.

We willen bereiken:

$$\{v_a = I(v_c)\}\ v_a := e_a;\ v_c := e_c\ \{v_a = I(v_c)\}.$$

Volgens de toekenningsregel van achter naar voren werkend, krijgen we

$$\{e_a = I(e_c)\}\ v_a := e_a\ \{v_a = I(e_c)\}\ v_c := e_c\ \{v_a = I(v_c)\}$$

en het doel wordt kennelijk bereikt door e_c zo te kiezen, dat $v_a = I(v_c) \Rightarrow e_a = I(e_c)$.

We richten nu onze aandacht op de tussen-assertie $v_a = I(e_c)$. In het algemeen zal de uitdrukking e_c een functie zijn van v_c, zodat $v_a = I(e_c) = I(f(v_c)) = J(v_c)$, en we verkrijgen dan

$$\{v_a = I(v_c)\}\ v_a := e_a\ \{v_a = J(v_c)\}.$$

Dit kan zo beschouwd worden dat we van de interpretatiefunctie I (die ook maar op een afspraak berust) zijn overgegaan op een nieuwe interpretatiefunctie J. De toekenning $v_c := e_c$ draagt er dan zorg voor dat we weer op I terugkomen. Nu zal daar in het algemeen geen haast bij zijn. Als op de cruciale punten maar geldt: $v_a = I(v_c)$, dan mag zo'n invariant in de tussenliggende gedeelten best tijdelijk verstoord zijn. Het is heel wel mogelijk dat $v_a = I(v_c)$ een invariant is van een **while**-**do**-**od**-constructie, terwijl $v_a = J(v_c)$ een invariant is van een daarbinnen liggende soortgelijke constructie. Als we J maar zo kiezen, dat uit $v_a = I(v_c)$ volgt: $e_a = J(v_c)$, dan krijgen we de regel

$$\{v_a = I(v_c)\}\ v_a := e_a\ \{v_a = J(v_c)\}.$$

Wanneer we dan achteraf de toekenning aan de abstracte variabele, zoals afgesproken, uit het programma schrappen, houden we over

$$\{v_a=I(v_c)\}\{v_a=J(v_c)\}.$$

We zien hier hoe de toekenning $v_a := e_a$ is geïmplementeerd *zonder dat daarmee enige actie correspondeert* louter door van interpretatie te veranderen. Door ons abstracte standpunt te verleggen verandert de *betekenis* van de onveranderde werkelijkheid, ongeveer zoals het beeld op ons netvlies verandert, wanneer we van een ander standpunt naar een voorwerp kijken. Bij een toekenning

$$\{v_a=J(v_c)\}\ v_c := e_c\ \{v_a=I(v_c)\}$$

gaan we naar het oorspronkelijke standpunt terug en verplaatsen daarbij het voorwerp zó, dat er op het oog -d.w.z. op het abstracte niveau- niets gebeurt.

Toegepast op een eenvoudig voorbeeld: De abstracte reële variabele a wordt gerepresenteerd door een paar, bestaande uit een reële variabele m en een integer variabele e, waarbij $I(m,e) = m \times 2^e$. (Deze representatie geeft de vrijheid m zo te kiezen dat $m = 0 \vee \frac{1}{2} \leq |m| < 1$, wat in bepaalde gevallen profijtelijk kan zijn) De toekenning a×:= 2 kan dan worden gerealiseerd door over te gaan op een zodanige J, dat uit a = I(m,e) volgt: a×2 = J(m,e). Nu is $I(m,e) \times 2 = m \times 2^e \times 2 = m \times 2^{e+1}$, zodat kennelijk $J(m,e) = m \times 2^{e+1}$. Als we later weer eens terug willen naar I, dan kan dat door de toekenning e+:= 1 (of eventueel door m×:= 2). Opgemerkt zij tot slot dat de toekenningen m×:= 2; e-:= 1 de assertie a = I(m,e) invariant laten: op het abstracte niveau gebeurt er niets. Dit soort toekenningen komt vaak voor om invarianten van de concrete representatie (zoals $m = 0 \vee \frac{1}{2} \leq |m| < 1$) te herstellen.

5. VOORBEELD: EEN WILLEKEURIGE RANGSCHIKKING

Om het idee van de vrijheid van representatie wat verder toe te lichten, beschouwen we het probleem een aantal gegeven elementen, b.v. letters, in een willekeurige volgorde te rangschikken. Dit kan gedaan worden door de elementen in een hoed te doen, en ze daar weer een voor een uit te halen, iedere maal willekeurig trekkend uit de nog in de hoed zittende elementen:

```
hoed:= de te rangschikken elementen;
rij:= ε {de lege rij};
while hoed niet leeg
do trek een willekeurig element el uit hoed;
   voeg el aan rij toe
od
{rij bevat de gevraagde rangschikking}.
```

Het correctheidsbewijs hiervan moge achterwege blijven. Merk op dat hoed en rij tezamen steeds juist de te rangschikken elementen bevatten (behalve op het ogenblik dat het getrokken element el tussen hoed en rij zweeft).

Een implementatie in ALGOL 68:

```
string hoed:= "anagram";
string rij:= "";
while hoed ≠ ""
do int r = entier (random × upb hoed) + 1;
   char el = hoed[r]; hoed:= hoed[:r-1] + hoed[r+1:];
   rij+:= el
od.
```

Gebruik makend van de observatie dat het aantal elementen in hoed en rij tezamen invariant is, kunnen we tot een minder voor de hand liggende representatie komen:

Zij n het aantal te rangschikken elementen (voor "anagram": n = 7). Zij verder s een variabele voor een rij van (vaste) lengte n, en k een integervariabele.

Dan wordt hoed gerepresenteerd door het paar (s,k), en wel met de interpretatie: hoed = (s[k+1],...,s[n]), *waarbij de volgorde van de* s[i] *niet terzake doet.*[*)] Verder wordt rij eveneens gerepresenteerd door (s,k), maar nu met rij = (s[1],...,s[k]) oftewel rij = s[1:k]. De implementatie luidt dan:

---

*)Dus, b.v. , (a,a,b) = (a,b,a) ≠ (a,b). Een dergelijk waardetype, waarvoor wel de term "bag" is gesuggereerd, ligt als het ware in tussen *verzamelingen* (waarbij {a,a,b} = {a,b,a} = {a,b}) en *rijen* (waarbij deze drie gevallen verschillend zijn). Helaas worden bags, die bij algoritmen vaak een rol spelen, door de wiskunde nogal stiefmoederlijk bedeeld. Voorbeelden van mogelijke toepassingen in de wiskunde zijn: de eigenwaarden van een matrix, of de kanten van een graaf (waarbij immers twee knopen door meerdere kanten verbonden kunnen zijn).

```
int n = 7; [1:n]char s:= "anagram";
int k:= 0;
{hoed = (s[1],...,s[n])= "anagram"  ∧ rij = s[1:0] = ε}
while k ≠ n
do {hoed = (s[k+1],...,s[n]) ∧ rij = s[1:k]}
   int r = entier (random × (n-k)) + k + 1; {k+1 ≤ r ≤ n}
   char el = s[r];
   {hoed = (s[k+1],...,s[r-1],s[r+1],...,s[n]  ∧ rij = s[1:k]}
   s[r]:= s[k+1];
   {hoed = (s[k+2],...,s[n]) ∧ rij = s[1:k]}
   k+:= 1;
   {hoed = (s[k+1],...,s[n]) ∧ rij = s[1:k-1]}
   s[k]:= el
   {hoed = (s[k+1],...,s[n]) ∧ rij = s[1:k]}
od
{k = n, dus rij = s[1:n] = s}.
```

Merk op dat niet bij voortduring in het do-part geldt:

hoed = (s[k+1],..., s[n]) ∧ rij = s[1:k],

maar dat dit wel geldt bij het begin en het einde; m.a.w., deze interpretatie is met betrekking tot het do-part, en dus tot de hele loop-clause, *invariant*. Wat in de eerste implementatie werd bewerkstelligd door hoed:= hoed[:r-1] + hoed[r+1:], wordt in deze versie teweeggebracht door tijdelijk over te gaan op een andere interpretatie, nl.
hoed = (s[k+1],...,s[r-1],s[r+1],...,s[n]), zonder dat daarmee een actie correspondeert. Het is misschien aardig van een gedeelte van deze versie te laten zien hoe het aan de hand van de asserties nagenoeg geheel "automatisch" geprogrammeerd kan worden (vgl.[5]).
Op een gegeven moment willen we van de assertie:

hoed = (s[k+1],...,s[r-1],s[r+1],...,s[n]) ∧ rij = s[1:k], (p)

die duidelijk tot uitdrukking brengt dat het element s[r] zojuist uit hoed is gelicht, naar de assertie:

hoed = (s[k+1],...,s[n]) ∧ rij = s[1:k-1], (q)

die laat zien dat het getrokken element nog aan rij moet worden toegevoegd. Het is niet duidelijk hoe q rechtstreeks in p te transformeren, maar een

bijdrage daartoe wordt geleverd door de substitutie k ← k + 1. We krijgen dan {q[k←k+1]} k+:= 1 {q}, zodat het probleem nu herleid is tot van p te geraken tot q[k←k+1]:

hoed = (s[k+2],...,s[n]) ∧ rij = s[1:k]. (t)

Het verschil tussen p en t kan iets explicieter tot uitdrukking worden gebracht door de asserties te herschrijven als

hoed = (s[k+1],s[k+2],...,s[r-1],s[r+1],...,s[n]) ∧ rij = s[1:k] (p')

respectievelijk

hoed = (s[k+2],...,s[r-1],s[r],s[r+1],...,s[n]) ∧ rij = s[1:k], (t')

of aangezien in hoed de volgorde van de s[i] niet ter zake doet,

hoed = (s[r],s[k+2],...,s[r-1],s[r+1],...,s[n]) ∧ rij = s[1:k]. (t")

Het is nu duidelijk, dat t" in p getransformeerd kan worden door de substitutie s[r] ← s[k+1], zodat we hebben:

{p} s[r]:= s[k+1] {t} k+:= 1 {q}.

We zullen dit voorbeeld besluiten met na te gaan wat in beide implementaties van de abstracte algoritme nodig is om na afloop de elementen weer uit rij in hoed te stoppen, zodat uitvoering van de loop-clause nogmaals een willekeurige rangschikking zou opleveren.
Voor de eerste implementatie wordt dit:

hoed:= rij; rij:= "".

Voor de tweede implementatie willen we van hoed = ( ) ∧ rij = s[1:n] naar hoed = (s[1],...,s[n]) ∧ rij = s[1:0], wat, bij invariante interpretatie, kan worden uitgedrukt als k = 0. We zien dus dat de gevraagde overheveling wordt bereikt door:

k:= 0.

6. REPRESENTATIE EN EFFICIENTIE

Eerder is al even gezinspeeld op de mogelijkheid door een geschikte representatiekeuze de efficiëntie van de algoritme te vergroten. Als we voor het zojuist uitgewerkte voorbeeld de alleszins redelijke veronderstelling maken, dat in ALGOL-68-implementaties de duur van de string-toekenning evenredig toeneemt met de lengte van de string, dan is de orde van het proces bij de tweede implementatie gereduceerd van n^2 tot n. Sommige operaties

van een abstracte algoritme kunnen nu eenmaal aanmerkelijk efficiënter geïmplementeerd worden bij de ene representatie dan bij de andere. Als we bijvoorbeeld een grote verzameling V hebben, waarvan de elementen paren zijn die bestaan uit een naam en een nummer, dan zal een abstracte operatie als

$$N := \{nr \mid (naam, nr) \in V\},$$

(d.w.z., het zoeken van de nummers die bij een gegeven naam horen), veel efficiënter geimplementeerd kunnen worden indien in de concrete representatie van V de elementen op de naam zijn gesorteerd. Deze representatie wordt in telefoonboeken gebruikt. In ALGOL 68 zouden we kunnen schrijven

```
flex [1:n] struct (string naam, int nummer) lijst.
```

Naast het verband tussen V en zijn representatie, dat gegeven wordt door de interpretatie V = {lijst[i] i ∈ [1:upb lijst]}, moet bij het opzoeken dan de assertie gelden:

$$i \leq j \Rightarrow \text{naam of lijst}[i] \leq \text{naam of lijst}[j] \quad \text{voor } i,j \in [1:\text{upb lijst}].$$

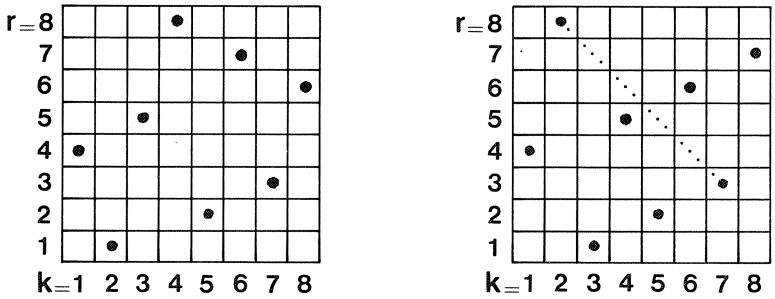
Als er zorg voor wordt gedragen dat deze assertie een invariant van de algoritme is, door bij de initialisering van V de geldigheid te "vestigen" en na iedere toekenning aan (een van de elementen van) lijst te herstellen, dan mag de geldigheid van deze invariant worden gebruikt bij het correctheidsbewijs van de implementatie van operaties als het opzoeken van een naam. We zien hier een nieuw element: *bij de concrete representatie hoort een invariant die in termen van de abstracte variabele geen betekenis heeft.* Een ander geregeld terugkerend verschijnsel is het *toevoegen van variabelen aan de concrete representatie* om invarianten te formuleren, die juist wel een betekenisrelatie met de abstracte variabele hebben. Als we een vector v representeren met v = c[1:n], en geregeld moet de norm ||v|| bepaald worden, dan zal het zin hebben een variabele sc2 in te voeren met invariant $sc2 = \sum_{i=1}^{n} c[i]^2$, wat dan ook geformuleerd kan worden als $sc2 = \|v\|^2$.

Men zou nu kunnen tegenwerpen dat de gewone programmeur, die van abstracte variabelen en invarianten geen weet heeft, eigenlijk intuïtief al hetzelfde doet als in deze beschouwingswijze wordt gesuggereerd. Het is aannemelijk dat daar een element van waarheid in zit. Het feit dat programmeurs er soms in slagen ook grote, ingewikkelde programma's te schrijven

en vervolgens door ontluizen "operationeel" te maken, wijst erop dat bij het programmeren op zijn minst onbewust van structureringstechnieken gebruik is gemaakt. Waar het om gaat is dat de programmeur vaak wel gedwongen is een representatie te kiezen (met toegevoegde invarianten) die een efficiënt programma mogelijk maakt, maar dat dit tevens een hachelijke zaak is: hij heeft, zo niet op papier, dan toch in zijn achterhoofd, de abstracte algoritme, en de taak de toegevoegde invarianten te vestigen en te herstellen, maakt daarvan geen deel uit. Indien deze taak dan onvolledig wordt uitgevoerd, zal het programma hem toch correct kunnen lijken. Een bijkomend probleem is, dat juist deze soort fouten -die, naar de ervaring leert, veel vaker voorkomen dan fouten in de onderliggende abstracte algoritme- aanleiding geeft tot verschijnselen die het opsporen van de fout niet gemakkelijk maken: de fout zal zich immers doorgaans op een heel andere plaats van het programma manifesteren dan waar hij gemaakt is, nl. bij de concrete uitwerking van de efficiënt geïmplementeerde abstracte operatie, en soms bij proefdraaien in het geheel niet aan het licht treden. De conclusie moet zijn dat het van essentieel belang is bij de concrete representatie van een abstracte variabele *expliciet* de bijbehorende invariant te formuleren, en te zorgen dat deze hersteld wordt bij iedere toekenning aan variabelen uit het conglomeraat dat voor de concrete representatie gebruikt wordt.

7. VOORBEELD: DE ACHT DAMES

Het (vaker als voorbeeld gekozen [6:9]) probleem van de acht dames luidt: geef alle configuraties van acht dames op een schaakbord waarbij geen van de dames het veld bestrijkt waarop een van de andere staat. In onderstaande figuur zijn twee configuraties getekend: links een oplossing, en rechts een configuratie die niet aan de eisen voldoet (dame (2,8) bestrijkt veld (7,3)).



Een configuratie C is te beschouwen als een verzameling paren (k,r), waarbij k en r de kolom- en rij-coördinaat van de afzonderlijke dames voorstellen. De door een dame bestreken velden worden gekenmerkt door het feit dat de k- of r-coördinaat, of de som (voor \-diagonalen) of het verschil (voor /-diagonalen) daarvan, gelijk zijn aan de overeenkomstige uitdrukking voor het veld van die dame. Als we definiëren:

$$OK(C) = \forall(k,r),\ (k',r')\in C:\ (k,r) \neq (k',r') \Rightarrow$$
$$k \neq k' \wedge r \neq r' \wedge k+r \neq k'+r' \wedge k-r \neq k'-r',$$

m.a.w., geen dame bestrijkt het veld van een andere dame, dan wordt een oplossing gekarakteriseerd door $OK(C) \wedge |C| = 8$ (met de stilzwijgende afspraak dat $k \in [1:8]$ en $r \in [1:8]$ voor alle (k,r) uit C). Het is duidelijk dat in een OK-configuratie per kolom hoogstens één dame voorkomt, en dat dus in een oplossing in iedere kolom een dame voorkomt, m.a.w., $K(C) = \{k|(k,r)\in C\} = [1:8]$.
Als we een C' hebben met $OK(C') \wedge K(C') = [1:|C'|]$, $|C'| \geq 1$, dan zal de configuratie $C = \{(k,r)\in C'|k \neq |C'|\}$, die uit C' ontstaat door de dame uit de hoogste vertegenwoordigde kolom te verwijderen, voldoen aan $OK(C) \wedge K(C) = [1:|C|]$. Omgekeerd geldt dus: als we bij gegeven |C| een methode hebben om al zulke configuraties C te vinden, dan kunnen we ook alle configuraties C' met $|C'| = |C|+1$ vinden, ieder door het toevoegen van een element aan C (waarbij voor het toegevoegde element (k,r) geldt: $k = |C'| = |C| + 1$). Voor $|C| = 0$ weten we de enig mogelijke configuratie: $C = \emptyset$, die ook voldoet, aangezien $OK(\emptyset) \wedge K(\emptyset) = [1:0] = \emptyset$.
We krijgen dan, in pseudo-ALGOL-68-notatie, de algoritme

```
config C:= ∅; breid uit
```

waarbij breid uit recursief is gedefinieerd door

```
proc breid uit = void : {OK(C) ∧ K(C) = [1:|C|]}
if |C| = 8 then {OK(C) ∧ |C| = 8} print (C) else
   for all (k,r) such that OK(C') ∧ K(C') = [1:|C'|]
                          where config C' = C ∪ {(k,r)}
   do C ∪:= {(k,r)};
      {OK(C) ∧ K(C) = [1:|C|]}
      breid uit;
      C-:= {(k,r)}
   od
fi.
```

Merk op dat de correctheid er deels op berust dat het netto effect op C van de routine breid uit nihil is, wat makkelijk te bewijzen is (door recursie-inductie), maar niet met de eenvoudige assertie-methode.
Het gedeelte

```
for all (k,r) such that OK(C') ∧ K(C') = [1:|C'|]
                    where config C' = C ∪ {(k,r)}
do ... od
```

kunnen we, gebruik makend van k = |C| + 1 en r ∈ [1:8], iets verder uitwerken tot

```
int k = |C| + 1;
for r to 8
do if OK(C ∪ {(k,r)}) then ... fi od.
```

Voor de concrete representatie van C kunnen we gebruik maken van

```
[1:8] int rij, int k
```

met interpretatie C = {(i, rij[i]) i ∈ [1:k]}, of afgekort, C = C(k) (wat inhoudt: k = |C|), maar daarnaast zal ook de interpretatie C = C(k-1) gebruikt worden. C ∪:= {(k,r)} wordt dan eenvoudig geïmplementeerd door rij[k]:= r en het tegelijk overstappen van de C(k-1) op de C(k)-representatie. We krijgen dan

```
[1:8] int rij, int k:= 0; {C=C(0)=∅}
proc breid uit = void:
if k = 8 then print ((rij, new line)) else
   {C=C(k)}
   k+:= 1;
   {C=C(k-1)}
   for r to 8
   do if OK(C ∪ {(k,r)} then
         {C=C(k-1)}
         rij[k]:= r;
         {C=C(k)}
         breid uit;
         {C=C(k)}
         {C=C(k-1)}
      fi
   od;
   {C=C(k-1)}
   k-:= 1
   {C=C(k)}
fi;
```

We zien hier weer een fraai voorbeeld hoe de abstracte toekenning C-:= {(k,r)} wordt teweeggebracht, louter door het overstappen van de interpretatie C = C(k) op C = C(k-1). De laatste is een invariant van de loop-clause; de invariant van de routine breid uit moet te zijner tijd worden hersteld door k-:= 1, wat op het abstracte niveau geen effect heeft.
Nu zitten we nog met de test OK(C ∪ {(k,r)}). Eenvoudig valt in te zien dat dit equivalent is met

$$OK(C) \wedge (\forall(k',r')\in C:\ k \neq k' \wedge r \neq r' \wedge k+r \neq k'+r' \wedge k-r \neq k'-r').$$

Gelukkig weten we uit de invariant van de abstracte algoritme dat OK(C) geldt. De rest herschrijven we als

$$(\forall(k',r')\in C:\ k \neq k') \wedge (\forall(k',r')\in C:\ r \neq r') \wedge$$
$$(\forall(k',r')\in C:\ k+r \neq k'+r') \wedge (\forall(k',r')\in C:\ k-r \neq k'-r',$$

wat we zullen afkorten tot

$$VER(k) \wedge HOR(r) \wedge$$
$$DIA1(k+r) \wedge DIA2(k-r).$$

VER(k) is de vraag: $k \notin K(C)$, dus $|C| + 1 \notin [1:|C|]$, hetgeen bevestigend mag worden beantwoord. Voor $k,r \in [1:8]$ geldt dat $k+r \in [2:16]$ en $k-r \in [-7:7]$, en in de hoop de test efficiënt te implementeren, voegen we toe

$$[1:8]\ \underline{bool}\ hor,\ [2:16]\ \underline{bool}\ dia1,\ [-7:7]\ \underline{bool}\ dia2$$

met invarianten

$$hor[i] = HOR(i),\ i \in [1:8],$$
$$dia1[i] = DIA1(i),\ i \in [2:16] \text{ en}$$
$$dia2[i] = DIA2(i),\ i \in [-7:7].$$

De test OK(C ∪ {(k,r)}) kan nu eenvoudig geïmplementeerd worden als hor[r] ∧ dia1[k+r] ∧ dia2[k-r], maar we hebben nu wel de taak op ons genomen de invarianten te handhaven.
Voor $C = \emptyset$ vinden we

$$hor[i] = HOR(i) = (\forall(k',r')\in\emptyset:\ i \neq r') = \underline{true},\ i\in[1:8],$$

en overeenkomstig dia1[i], $i \in [2:16]$ en dia2[i], $i \in [-7:7]$.
Voor de toekenning Cu:= {(k,r)} krijgen we

HOR(i)[C ← C ∪ {(k,r)}] =
(∀(k',r') ∈ C ∪ {(k,r)}: i ≠ r') =
((∀(k',r') ∈ C: i ≠ r') ∧ i ≠ r) =
(HOR(i) ∧ i ≠ r).

Om de invariant te herstellen moeten we dus een toekenning vinden bij

{(hor[i] ∧ i≠r) = HOR(i)} v_c:= e_c {hor[i] = HOR(i)}, i∈[1:8],

wat lukt met hor[r]:= false. Voor de twee andere invarianten vinden we overeenkomstig dia1[k+r]:= false en dia2[k-r]:= false.
Na de toekenning C-:= {(k,r)} moeten de invarianten ook weer hersteld worden. Aangezien de eenvoudige assertie-methode voor niet-triviale recursie tekort schiet, lukt dat niet via een analoge redenering. Via recursie-inductie kan het echter heel eenvoudig: We nemen als inductie-aanname dat breid uit niet alleen C maar ook hor, dia1 en dia2 ongemoeid laat, en hoeven dan alleen de zichtbaar aangerichte schade te herstellen. Aangezien het punt na then, waar de toekenningen gaan plaatsvinden, alleen bereikt wordt indien hor[r] ∧ dia1[k+r] ∧ dia2[k-r], kan de invariant hersteld worden door aan hor[r], dia1[k+r] en dia2[k-r] weer true toe te kennen. De uitgewerkte algoritme wordt nu:

```
[1:8] int rij, int k:= 0; [1:8] bool hor, [2:16] bool dia1,
        [-7:7] bool dia2;
for i to 8 do hor[i]:= true od;
for i from 2 to 16 do dia1[i]:= true od;
for i from -7 to 7 do dia2[i]:= true od;
proc breid uit = void:
if k = 8 then print ((rij, newline)) else
   k+:= 1;
   for r to 8
   do if hor[r] ∧ dia1[k+r] ∧ dia2[k-r] then
         rij[k]:= r; hor[r]:= dia1[k+r]:= dia2[k-r]:= false;
         breid uit;
         hor[r]:= dia1[k+r]:= dia2[k-r]:= true
      fi
   od;
   k-:= 1
fi;
breid uit.
```

8. HOGERE PROGRAMMEERTALEN EN ABSTRACTE VARIABELEN

In GEURTS [5] wordt een pleidooi gehouden voor faciliteiten die het mogelijk maken bij het gestructureerd programmeren de structuur van de verschillende abstractieniveaus in het uiteindelijk programma te kunnen bewaren. Hier zullen daarover enkele losse gedachten worden ontvouwd, toegespitst op abstracte variabelen, waarbij ALGOL 68 als vertrekpunt wordt genomen. De gewenste faciliteit zou inhouden dat we in de programmatekst toekenningen $v_a := e_a$ kunnen opnemen en dan verder kunnen specificeren welke concrete representatie daarbij hoort. Een belangrijk middel daartoe zijn de door Hoare voorgestelde *records* HOARE [10], die in recentere programmeertalen hun weg gevonden hebben en in ALGOL 68 als gestructureerde variabelen en waarden verschijnen. Voegen we daarbij nog de mode- en operatiedefinitie van ALGOL 68, dan kunnen we al schrijven:

```
complex z3:= z1 × z2,
```

en daar bijv. bij definieren

```
mode complex = struct (real re, im);
op × = (complex a,b) complex:
(re of a × re of b - im of a × im of b,
 re of a × im of b + im of a × re of b),
```

maar evengoed

```
mode complex = struct (real mod, arg);
op × = (complex a,b) complex:
(mod of a × mod of b, arg of a + arg of b).
```

Lastiger wordt het als we bags willen invoeren. We kunnen wel schrijven: `mode bag of char = flex [1:0] char`, maar dit heeft het nadeel dat de standaard-gelijkheidsoperatie voor strings ook op onze karakter-bags werkt, maar dan niet zoals we zouden willen, of, wanneer we de operatie herdefiniëren, dat we de oude voor strings kwijt zijn. Dit komt doordat ook `mode string = flex [1:0] char`. We kunnen dit probleem met een trucje omzeilen, maar een fraaie oplossing is het niet. Een niet te omzeilen probleem is dat we wat we op deze wijze voor karakter-bags opbouwen, voor andere typen bags telkens opnieuw van de grond af moeten opbouwen. Wat we eigenlijk willen hebben is iets als

```
mode bag = (mode m) mode : flex [1:0] m
```

waarna bag(char) s hetzelfde betekent als flex [1,0] char s.
Verder zijn sommige waardetypen, of klassen daarvan, bevoorrecht, aangezien daarvoor denotaties of standaardschrijfwijzen bestaan (zoals 13, "abc", nil, of ( ) voor een lege rij). In ALGOL 68 is het voor de programmeur niet mogelijk een schrijfwijze als ∅ in te voeren voor de lege verzameling (trucjes daargelaten). Andere voorrechten worden verleend bij de coercies (impliciete typetransformaties), zoals integer → reëel getal → complex getal, of karakter → string. Hoe de programmeur de vrijheid geschonken kan worden zelf coercies te definiëren, zonder gevaar voor dubbelzinnigheden, is nog onduidelijk. Dit zijn overigens slechts aanzetten, die het eigenlijke probleem alleen maar aan de rand benaderen. We mogen niet rusten voordat we kunnen schrijven: hoed:= rij; rij:= ε, of een qua abstractieniveau gelijkwaardige formulering, en dan zodanige specificaties m.b.t. de concrete representatie kunnen geven, dat de uiteindelijke implementatie neerkomt op: k:= 0.

LITERATUUR

[1] GRUNE, D., *ALEPH, een grammaticale aanpak van programma-correctheid*, Hoofdstuk 4 van deze syllabus.

[2] WULF, W. & M. SHAW, *Global variable considered harmful*, SIGPLAN Notices, 8 (1972) 28-34.

[3] WIRTH, N., *The programming Languages PASCAL*, Acta Informatica, 1 (1971) 35-63.

[4] HOARE, C.A.R., *Proof of a structured program: "The sieve of Eratosthenes"*, The Computer Journal, 15 (1972) 321325.

[5] GEURTS, L.J.M., *Gestructureerd programmeren*, Hoofdstuk 3 van MC Syllabus 21 (1974)

[6] BRON, C., *Over het nut van recursieve programmeertechnieken*, Informatie, 12 (1971) 221-227.

[7] WIRTH, N., *Program Development by Stepwise Refinement*, Comm. A.C.M., 14 (1971) 221-227.

[8] DIJKSTRA, E.W., *Notes on Structured Programming*, in: *Structured Programming*, APIC Studies in Data Processing 8, Academic Press, London, 1972.

[9] NAUR, P., *An Experiment on Program Development*, BIT, 12 (1972) 347-365.

[10] HOARE, C.A.R., *Record Handling*, ALGOL Bulletin 21.3.6 (1965).

MATHEMATICAL CENTRE SYLLABUS 21, 1975, VII, 111-126.

EEN BEWIJSMETHODE VOOR RECURSIEVE PROCEDURES

J.W. de BAKKER

1. RECURSIEVE PROCEDURES ALS KLEINSTE DEKPUNTEN

We bespreken in dit hoofdstuk een methode voor het bewijzen van eigenschappen van recursieve procedures, die gebaseerd is op de zogenaamde "kleinste dekpunt"-karakterizering. Deze benadering gaat in feite terug tot KLEENE [9], (p.348), maar is in 1969 in de programmeertheorie opnieuw geïntroduceerd door SCOTT [12], die er tevens een belangrijke bewijsregel aan verbond.
We zullen trachten om, zonder een volledig strenge opbouw van deze theorie te geven, de belangrijkste ideeën eruit weer te geven en in diverse toepassingen te illustreren.
Als voorbeeld gebruiken we de faculteitfunctie, eerst weergeven in de schrijfwijze van ALGOL 60:

```
integer procedure f(x);
      f:= if x=0 then 1 else x*f(x-1)
```

Voor dit type declaraties gebruiken we in het vervolg liever de kortere notatie

(1) $f(x) \Leftarrow$ if x=0 then 1 else x*f(x-1)

We veronderstellen bekend hoe een functie als f berekend dient te worden (in dit speciale geval zeg voor gehele $x \geq 0$). Naast deze, laten we zeggen, *operationele* betekenis van (1) willen we als alternatief komen tot een zogenaamde *mathematische* karakterizering. We introduceren daartoe eerst enkele hulpmiddelen.
Het biedt voordelen de beschouwde functies mede te bespreken aan de hand van hun *graaf*, d.i., de verzameling van paren argumenten met bijbehorende functiewaarden. We hebben dus bij een functie f als bijbehorende graaf T_f:

$$T_f = \{(x,y) \mid y = f(x)\}.$$

Verder leidt het uitdrukkingsmiddel van recursie al snel tot het beschouwen van niet overal gedefinieerde functies (*partiële* i.p.v. *totale* functies). Men zie bijv. de f uit (1) in negatieve argumenten, of het extreme geval van de functie g gedefinieerd door

$$g(x) \Leftarrow g(x)$$

die kennelijk voor geen enkel argument x een waarde oplevert, d.w.z., voor deze g hebben we $T_g = \emptyset$ (de lege verzameling). We reserveren het symbool Ω voor de nergens gedefinieerde functie, zodat $T_g = \emptyset \Leftrightarrow g = \Omega$. Als een functie f ongedefinieerd is in argument x, schrijven we $f(x) = \omega$. Verder veronderstellen we alle functies uitgebreid tot ω, met $f(\omega) = \omega$ als definitie. We definiëren de inclusie- en verenigingsoperaties op partiële functies in termen van dezelfde operaties op hun grafen, m.a.w.

$$f \subseteq g \overset{df}{\Leftrightarrow} T_f \subseteq T_g$$

$$f = f_1 \cup f_2 \overset{df}{\Leftrightarrow} T_f = T_{f_1} \cup T_{f_2}$$

$$f = \bigcup_{i \in I} f_i \overset{df}{\Leftrightarrow} T_f = \bigcup_{i \in I} T_{f_i}.$$

We willen nu de betekenis van (1) nader analyseren, door een methode aan te geven om de graaf van f te construeren, waarbij we de functie f niet als reeds bekend veronderstellen. Wat we hiertoe doen is f successievelijk benaderen door de (niet recursieve) functies f_i, i=0,1,..., gedefinieerd door

$$(2) \qquad \begin{cases} f_0(x) = \omega \\ f_i(x) = \underline{\text{if}}\ x=0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ x*f_i(x-1) = 1,2,\ldots \end{cases}$$

We hebben dus de ene functie f, gedefinieerd in de oneindige verzameling {0,1,2,...}, vervangen door de oneindig vele functies f_i, i=0,1,..., waarbij iedere f_i gedefinieerd blijkt in de eindige verzameling {0,1,...,i-1}. Met de f_i corresponderen de grafen

$$(3) \qquad \begin{cases} T_{f_0} = \emptyset \\ T_{f_i} = \{(x,y) \mid y = \underline{\text{if}}\ x=0\ \underline{\text{then}}\ 1\ \text{else}\ x*f_{i-1}(x-1)\},\ i=1,2,\ldots \end{cases}$$

We beweren nu dat

LEMMA 1. $f \subseteq \bigcup_{i=0}^{\infty} f_i$

Bewijs (schets):
We tonen aan dat $\forall x,y[\text{Als } y=f(x) \text{ dan } \exists i[y=f_i(x)]]$ door inductie naar de "recursiediepte" r, d.i. het aantal aanroepen van f bij de berekening van f(x) uit x.

a. r = 0. Dan is x = 0, en f(x) = f(0) = 1 = $f_1(0)$, zodat we voor i 1 kunnen kiezen.

b. Zij de bewering bewezen voor recursiediepte $\leq$ r - 1, en stel de berekening van f(x) vraagt een diepte r. Dan is kennelijk de substitutie f(x) = x*f(x-1) toegepast, met r - 1 als recursiediepte voor het berekenen van f(x-1). Op grond van de inductieaanname geldt f(x-1) = $f_i(x-1)$ voor geschikte i. Dan is f(x) = x*$f_i(x-1) \overset{(2)}{=} f_{i+1}(x)$, zodat i + 1 de gezochte index is. □

Opmerking. Het begrip "recursiediepte" is precies gedefinieerd in bijv. DE BAKKER & DE ROEVER [3] en DE BAKKER & MEERTENS [4]. In ons speciale geval zou het bewijs eenvoudiger kunnen, doordat direct in te zien is (door inductie naar x) dat voor de gezochte i zodat f(x) = $f_i(x)$, steeds x + 1 kan worden gekozen. Dergelijke inductie naar argument is echter niet steeds mogelijk, inductie naar recursiediepte wel.

Als volgende stap willen we nu aantonen dat ook $\bigcup_{i=0}^{\infty} f_i \subseteq f$ geldt. Hiertoe is het van belang nog eens naar het rechterlid van (1) te kijken: if x=0 then 1 else x*f(x-1). We merken op dat hier zowel de x als de f als variabelen gezien kunnen worden, d.w.z. de algemene gedaante van (1) is

$$f(x) \Leftarrow F(f)(x)$$

(waarbij, zo men wil, F = $\lambda f \cdot \lambda x \cdot$ if x=0 then 1 else x*f(x-1)). We zien hier F dus als *functionaal* die functies op functies afbeeldt, zodat we bijv. voor (2) ook kunnen schrijven:

$$(4) \qquad \begin{cases} f_0 = \Omega \\ f_i = F(f_{i-1}), \quad i=1,2,\ldots \end{cases}$$

Uit de manier waarop f berekend wordt volgt, dat $\forall x[f(x) = F(f)(x)]$, ofwel $F(f) = f$, d.i., f is een *dekpunt* van de functionaal F. (Voor een commentaar op het geval dat f een functie van meer veranderlijken is, zie het eind van deze sectie.) Bovendien kan geverifieerd worden, dat F monotoon is, d.i., dat voor alle f, g, als $f \subseteq g$, dan $F(f) \subseteq F(g)$. Het bewijs van $\bigcup_{i=0}^{\infty} f_i \subseteq f$ is nu makkelijk.

LEMMA 2. $\bigcup_{i=0}^{\infty} f_i \subseteq f$

Bewijs. We tonen aan dat voor alle i, $f_i \subseteq f$, door inductie naar i.

a. $f_0 \subseteq f$ is duidelijk.

b. Stel $f_{i-1} \subseteq f$. Dan $f_i = F(f_{i-1}) \subseteq F(f) = f$, op grond van (4), de inductieaanname, de monotonie van F, en de dekpunteigenschap. □

Schrijven we $F^i(\Omega)$ voor $\underbrace{F(F(\ldots F}_{i\times F}(\Omega)\ldots))$ dan hebben we als conclusie:

STELLING 1. $f = \bigcup_{i=0}^{\infty} f_i = \bigcup_{i=0}^{\infty} F^i(\Omega)$

Bewijs. Lemma's 1,2 en definitie (4). □

<u>Opmerking</u> 1. Als $g = F(g)$, dan volgt als in lemma 2 dat $\bigcup_{i=0}^{\infty} f_i \subseteq g$, dus $f \subseteq g$. We zien dat f een dekpunt van F is, bevat in ieder dekpunt, m.a.w. f is het *kleinste dekpunt*. Dat F meerdere dekpunten kan hebben is duidelijk. Beschouw bijv. F gedefinieerd door: $F(f) = f$ voor alle f. Het kleinste dekpunt van deze F is Ω, in overeenstemming met het feit dat de functie f met declaratie $f(x) \Leftarrow f(x)$ nergens gedefinieerd is.

2. Stelling 1 heeft een wijder toepassingsgebied dan dat van recursieve procedures. Zo is bijvoorbeeld in de formele talentheorie met de grammatica G: $S \rightarrow aSb|\varepsilon$ de vergelijking $L = aLb \cup \varepsilon$ te associëren, met als kleinste oplossing $L = \bigcup_{i=0}^{\infty} L_i$, waarbij $L_0 = \emptyset$, $L_i = aL_{i-1}b \cup \varepsilon$. L is dus juist de door G voortgebrachte taal $\{a^n b^n | n \geq 0\}$.

De karakterizering van stelling 1 stelt ons in staat de bewijsregel te verklaren die bekend staat onder de naam van *Scott's inductieregel*. We beginnen met een eenvoudige versie. Stel we willen een bewering over functies van de vorm $f \subseteq g$ bewijzen, met $f(x) \Leftarrow F(f)(x)$ als declaratie voor f. Een toepassing van Scott's inductieregel zou in dit geval als volgt kunnen gaan: Om tot $f \subseteq g$ te kunnen concluderen, is het voldoende a en b te

bewijzen:

a. $\Omega \subseteq g$.

b. Voor willekeurige functies X: als $X \subseteq g$, dan $F(X) \subseteq g$.

De geldigheid van de regel volgt direct uit stelling 1: Uit a volgt $\Omega = f_0 \subseteq g$, en m.b.v. b concluderen we, dat als $f_{i-1} \subseteq g$, dan $f_i \subseteq g$. Tezamen geeft dit $f = \bigcup_{i=0}^{\infty} f_i \subseteq g$.

In de meest algemene vorm luidt Scott's regel: Zij P(f) een bewering over een functie f, waarbij $f(x) \Leftarrow F(f)(x)$. Dan geldt: uit

a. $P(\Omega)$

b. Als P(X) dan P(F(X))

kan tot P(f) worden geconcludeerd.

{De vorm van P is hier niet volledig vrij, maar de voorwaarden waaraan P moet voldoen blijven hier onbesproken. In onze voorbeelden is P steeds toelaatbaar.}

In de volgende secties zullen we diverse toepassingen van de inductieregel geven. Eerst echter nog een intermezzo:

{*De dekpunteigenschap bij functies van meer veranderlijken*

Er bestaat enige onduidelijkheid t.a.v. de dekpunteigenschap, F(f) = f, bij functies van meer veranderlijken. Beschouw als voorbeeld:

$$f(x,y) \Leftarrow \underline{\text{if}}\ x=0\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ f(x-1,f(x,y)).$$

Stel dat deze functie wordt berekend volgens het mechanisme van call by value, met resultaat f_v. We hebben dan:

$$f_v = \lambda xy\cdot\ \underline{\text{if}}\ x=0\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ \omega.$$

Volgens MANNA et al. (bijv. [10,11]) zou hiervoor niet gelden dat $F(f_v) = f_v$, omdat, bijvoorbeeld, $F(f_v)(1,0) = 0$, maar $f_v(1,0) = \omega$. We hebben

$$F = \lambda f.\lambda xy\ \underline{\text{if}}\ x=0\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ f(x-1,\ f(x,y)),$$

dus, f_v voor f substituerend en het resultaat toepassend in (1,0):

$F(f_v)(1,0) =$

$(\lambda xy\cdot\ \underline{\text{if}}\ x=0\ \underline{\text{then}}\ 0$

$\quad \underline{\text{else}}\ (\lambda uv\cdot\ \underline{\text{if}}\ u=0\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ \omega)$

$\quad\quad (x-1,\ (\lambda uv\cdot\ \underline{\text{if}}\ u=0\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ \omega)(x,y)))(1,0).$

De uitkomst hiervan is afhankelijk van de gekozen λ-conversievolgorde. Wordt hierbij het analogon van het call by value mechanisme gekozen, dan

geldt wel degelijk dat $F(f_v)(1,0) = \omega$ (waarbij ω dan wordt opgevat als aanduiding voor een uitdrukking waarvoor de gekozen λ-conversievolgorde niet tot een normaalvorm leidt), en in het algemeen $F(f_v) = f_v$. Indien echter een andere conversievolgorde wordt gekozen, kan de uitkomst van $F(f_v)(1,0)$ inderdaad 0 zijn. De bewering dat call by value berekening niet tot het kleinste dekpunt leidt, moet daarom m.i. verfijnd worden tot: Als de berekeningswijze afwijkt van het λ-conversie mechanisme, dan *kan* een verschillend resultaat bereikt worden. Nadere opheldering van deze problematiek is overigens gewenst.}

2. EERSTE TOEPASSINGEN

2.1. *Toepassingen op functies*

We bespreken een tweetal toepassingen van Scott's inductieregel, ontleend aan MANNA, NESS & VUILLEMIN [10].

Voorbeeld 1. Zij

$$f(x) \Leftarrow \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ x\ \underline{\text{else}}\ f(f(a(x))).$$

We zullen aantonen dat $\forall x[f(f(x)) = f(x)]$, door toepassing van Scott's regel op de bewering $P(X) \overset{df}{\Leftrightarrow} \forall x[f(X(x)) = X(x)]$. (We laten de quantificatie $\forall x$ in het vervolg weg.)

a. Verificatie van $P(\Omega)$:

$$\begin{aligned} f(\Omega(x)) &= \underline{\text{if}}\ p(\Omega(x))\ \underline{\text{then}}\ \Omega(x)\ \underline{\text{else}}\ f(f(h(\Omega(x)))) = \\ &= \underline{\text{if}}\ p(\omega)\ \underline{\text{then}}\ \omega\ \underline{\text{else}}\ f(f(\omega)) = \\ &= \underline{\text{if}}\ p(\omega)\ \underline{\text{then}}\ \omega\ \underline{\text{else}}\ \omega = \\ &= \omega = \\ &= \Omega(x). \end{aligned}$$

b. Verificatie van: Als $P(X)$ dan $P(F(X))$. Neem aan $f(X(x) = X(x)$. Dan

$$\begin{aligned} &f(\underline{\text{if}}\ p(x)\ \underline{\text{then}}\ x\ \underline{\text{else}}\ X(X(a(x)))) = \\ &\underline{\text{if}}\ p(x)\ \underline{\text{then}}\ f(x)\ \underline{\text{else}}\ f(X(X(a(x)))) \overset{(\text{aanname})}{=} \\ &\underline{\text{if}}\ p(x)\ \underline{\text{then}}\ f(x)\ \underline{\text{else}}\ X(X(a(x))) = \\ &\underline{\text{if}}\ p(x)\ \underline{\text{then}}\ (\text{if}\ p(x)\ \underline{\text{then}}\ x\ \underline{\text{else}}\ f(f(a(x))))\ \underline{\text{else}}\ X(X(a(x))) = \\ &\underline{\text{if}}\ p(x)\ \underline{\text{then}}\ x\ \underline{\text{else}}\ X(X(a(x)) \end{aligned}$$

waaruit we zien dat inderdaad $f(F(X)(x)) = F(X)(x)$. Combinatie van a en b

levert dat inderdaad f(f(x)) = f(x). □

Voorbeeld 2. Als tweede voorbeeld beschouwen we de functie append(x,y), die werkt op, zeg, rijen letters, waarbij we de lege rij aanduiden met ε. Append gebruikt de hulpfuncties:

h(x) : de eerste letter van de rij x.
t(x) : het resultaat van het weglaten van de eerste letter van x.
a·x : de rij verkregen door de letter a links van de rij x te zetten. (N.B.: "·" is dus slechts gedefinieerd met als linkerargument een *letter*.)

We definiëren:

append(x,y) ⇐ if x=ε then y else h(x)· append(t(x),y).

We schrijven x*y voor append(x,y), en willen bewijzen dat (x*y)*z = = x*(y*z). Als bewering P(X) waarop we Scott's regel toepassen kiezen we X(x,y*z) = X(x,y)*z.

a. P(Ω) is duidelijk.

b. Neem P(X) aan. We tonen aan dat dan P(F(X)) geldt. We hebben

F(X)(x,y*z) = if x=ε then y*z else h(x)·X(t(x),y*z) (op grond van de aanname)
= if x=ε then y*z else h(x)·(X(t(x),y)*z) (op grond van def. append)
= if x=ε then y*z else (h(x)·X(t(x),y))*z =
= (if x=ε then y else h(x)·X(t(x),y))*z =
= F(X)(x,y)*z.

Uit a en b concluderen we dat inderdaad append(x,y*z) = append(x,y)*z, d.i., dat x*(y*z) = (x*y)*z. □

4.2. *Toepassingen op while statements*

Beschouw de while statement while p do A, afgekort tot p*A. We definiëren p*A als gelijkwaardig met de procedure P gedeclareerd door

procedure P; if p then begin A; P end

We willen op deze P de methode van sectie 1 toepassen. Daartoe beschouwen we P als werkend op de toestandsvector σ, d.i., de vector van de (variabele, waarde) paren die door het programma worden verwerkt. (Vgl. ook

hoofdstuk 1 van deze syllabus.) We schrijven daarom ook wel voor de declaratie van P:

$$P(\sigma) \Leftarrow \underline{\text{if}}\ p(\sigma)\ \underline{\text{then}}\ P(A(\sigma))\ \underline{\text{else}}\ \sigma$$

Deze benadering stelt ons in staat om bewijzen over programma's met while statements (en overigens ook met recursieve procedures in het algemeen) volgens Scott's regel te geven. In plaats van de functionele notatie gebruiken we hier liever de zogenaamde μ-notatie: Zij $T(X)(\sigma) \stackrel{df}{=} \underline{\text{if}}\ p(\sigma)$ $\underline{\text{then}}\ X(A(\sigma))\ \underline{\text{else}}\ \sigma$. We schrijven voor T(X) ook $(p \rightarrow A;X,I)$, waarbij

a. i.p.v. de $\underline{\text{if}}$ ... $\underline{\text{then}}$ ... $\underline{\text{else}}$ ... notatie de kortere $(\ldots \rightarrow \ldots , \ldots)$ notatie is toegepast.

b. ";" compositie aanduidt volgens: $(A;X)(\sigma) = X(A(\sigma))$.

c. I de identiteitsfunctie (corresponderend met de dummy statement) voorstelt.

Zij nu $\mu X[T(X)] = \bigcap\{X:X = T(X)\} = \bigcup_{i=0}^{\infty} T^i(\Omega)$ het kleinste dekpunt van T. Dan kunnen we voor p*A dus ook $\mu X[(p \rightarrow A;X,I)]$ schrijven. We geven nu een aantal toepassingen:

1. $p*A = (p \rightarrow A;p*A,I)$.
   Dit is niets anders dan de dekpunteigenschap.

2. $\mu X[(p \rightarrow A_1;X,A_2)] = \mu X[(p \rightarrow A_1;X,I)];A_2$.
   Bewijs voor de lezer.

3. $\mu X[T_1(T_2(X))] = T_1(\mu X[T_2(T_1(X))])$.
   Noem de procedure links L en rechts R.
   a. $L \subseteq T_1(R)$. Het is voldoende te bewijzen: Als $X \subseteq T_1(R)$ dan $T_1(T_2(X)) \subseteq T_1(R) = T_1(T_2(T_1(R)))$, hetgeen volgt uit de monotonie van de T's.
   b. $T_1(R) \subseteq L$. We bewijzen: Als $T_1(X) \subseteq L$, dan $T_1(T_2(T_1(X))) \subseteq L =$ $= T_1(T_2(L))$, idem.

Als speciaal geval van 3 hebben we

3'. $\mu X[A;T(X)] = A;\ \mu X[T(A;X)]$.

4. $\mu X[T(X)] = \mu Y[T(Y)]$ is duidelijk op grond van de definities (mits de gebruikelijke voorzorgen betreffende vrije en gebonden variabelen in acht worden genomen). $\mu X[T(X,X)] = \mu Y[\mu X[T(X,Y)]]$ is ook niet moeilijk te bewijzen.

5. $(p_1 \vee p_2) * A = p_1*A;p_2*(A;p_1*A)$. (Vgl. de laatste bewering van hoofdstuk 1

van deze syllabus):

We hebben

$$
\begin{aligned}
(p_1 \vee p_2) * A &= \mu X[(p_1 \vee p_2 \to A;X,I)] = \\
&= \mu X[(p_1 \to A;X,(p_2 \to A;X,I))] \overset{(4)}{=} \\
&= \mu X[\mu Y[(p_1 \to A;Y,(p_2 \to A;X,I))] \overset{(2)}{=} \\
&= \mu X[\mu Y[(p_1 \to A;Y,I)];(p_2 \to A;X,I)] \overset{df}{=} \\
&= \mu X[p_1 * A;(p_2 \to A;X,I)] \overset{(3')}{=} \\
&= p_1 * A;\mu X[p_2 \to A;p_1 * A,I)] \overset{df}{=} \\
&= p_1 * A;p_2 * (A;p_1 * A).
\end{aligned}
$$

5'. (Voor de lezer bekend met reguliere expressies). Zij a^* de kleinste oplossing van $L = aL \cup \varepsilon$, d.i., $a^* = \mu X[aX \cup \varepsilon]$. Het bewijs van 5 kan precies zo worden overgenomen om $(a_1 \cup a_2)^* = a_1^*(a_2 a_1^*)^*$ te bewijzen.

6. Bewijs van Hoare's while statement axioma (vgl. hoofdstuk 1 van de syllabus). Hoare's regel kan gebracht worden in de vorm: Als

(5.1) $\forall \sigma,\sigma'[u(\sigma) \wedge p(\sigma) \wedge \sigma A \sigma' \to u(\sigma')]$

dan

(5.2) $\forall \sigma,\sigma'[u(\sigma) \wedge \sigma p^* A \sigma' \to u(\sigma')]$.

Neem (5.1) aan. Volgens Scott's regel is het voor het bewijs van (5.2) voldoende om te bewijzen dat: Als

(5.3) $\forall \sigma,\sigma'[u(\sigma) \wedge \sigma X \sigma' \to u(\sigma')]$

dan

(5.4) $\forall \sigma,\sigma'[u(\sigma) \wedge \sigma(p \to A;X,I)\sigma' \to u(\sigma')]$.

Voor (5.4) hebben we als gelijkwaardige bewering:

$$\forall \sigma,\sigma'[u(\sigma) \wedge [\{p(\sigma) \wedge \sigma A;X\sigma'\} \vee \{\neg p(\sigma) \wedge \sigma I \sigma'\}] \to u(\sigma')]$$

ofwel

$$\forall \sigma,\sigma'[u(\sigma) \wedge p(\sigma) \wedge \sigma A;X\sigma' \to u(\sigma')]$$

<u>en</u>

$$\forall \sigma,\sigma'[u(\sigma) \wedge \neg p(\sigma) \wedge \sigma = \sigma' \to u(\sigma')].$$

De tweede bewering is direct duidelijk, terwijl de eerste bewering uit (5.1) en (5.3) als volgt is af te leiden: Stel dat $u(\sigma)$, $p(\sigma)$ en $\sigma A;X\sigma'$ alle gelden. Dan is er een σ'', zodat $u(\sigma)$ en $p(\sigma)$ en $\sigma A \sigma''$ en $\sigma'' X \sigma'$. Volgens (5.1) geldt dan $u(\sigma'')$. Uit $u(\sigma'')$ en $\sigma'' X \sigma'$ volgt m.b.v. (5.3) dat inderdaad $u(\sigma')$. Hiermee is het bewijs van (5.4), en daarmee van Hoare's axioma, voltooid.

3. VOLGENDE TOEPASSINGEN

We bespreken nu een tweetal iets minder elementaire toepassingen.

3.1. *Een bewering van Dijkstra*

We beschouwen nogmaals correctheidsuitspraken van de vorm

$$\forall\sigma,\sigma'[p(\sigma)\wedge\sigma P\sigma' \rightarrow q(\sigma')].$$

We kunnen deze bewering herschrijven als

$$\forall\sigma[p(\sigma) \rightarrow \forall\sigma'[\sigma P\sigma' \rightarrow q(\sigma')]].$$

We zien hieruit dat iedere p met de eigenschap dat geldigheid hiervan vóór uitvoering van P, geldigheid van q na afloop van P impliceert, bevat is in het predicaat $\forall\sigma'[\sigma P\sigma' \rightarrow q(\sigma')]$. M.a.w., we kunnen dit predicaat opvatten als de "weakest precondition" in de zin van DIJKSTRA [7]. Houden we ook rekening met het feit dat deze "weakest precondition" in DIJKSTRA [7] geacht wordt terminatie te impliceren, dan krijgen we voor Dijkstra's $f_P(q)$ constructie:

$$\forall\sigma[f_P(q)(\sigma) \leftrightarrow \exists\sigma'[\sigma P\sigma'] \wedge \forall\sigma'[\sigma P\sigma' \rightarrow q(\sigma')]].$$

Voegen we bovendien nog de eis toe dat P een *functie* is $(\forall\sigma,\sigma',\sigma''[\sigma P\sigma'\wedge\sigma P\sigma'' \rightarrow \sigma'=\sigma''])$, dan verkrijgen we tenslotte:

$$\forall\sigma[f_P(q)(\sigma) \leftrightarrow \exists\sigma'[\sigma P\sigma'\wedge q(\sigma')]].$$

Hebben we dit eenmaal ingezien, dan zijn de "basic properties" en de "axioma's" uit DIJKSTRA [7] makkelijk bewijsbaar. Bovendien valt een nieuw licht op de hoofdstelling uit DIJKSTRA [7], daar als volgt geformuleerd: Zij P een procedure gedeclareerd door procedure P;T(P). Stel nu dat het volgende geldt voor willekeurige predicaten q,r: Als

$$\forall\sigma[q(\sigma) \rightarrow f_X(r)(\sigma)]$$

dan

$$\forall\sigma[q(\sigma) \rightarrow f_{T(X)}(r)(\sigma)].$$

We mogen dan concluderen tot

$$\forall\sigma[q(\sigma)\wedge f_P(t)(\sigma) \rightarrow f_P(r)(\sigma)]$$

waarin t het identiek ware predicaat voorstelt (zodat $f_P(t)(\sigma) \leftrightarrow \exists\sigma'[\sigma P\sigma']$ $\leftrightarrow$ P termineert in σ).

Het is makkelijk in te zien dat

1. In de vorm als nu gegeven, de bewering niet juist is. Kies bijvoorbeeld

q identiek waar, en r identiek onwaar (t en o respectievelijk). Duidelijk is dat $f_X(o)$, $f_{T(X)}(o)$ en $f_P(o)$ alle aan o gelijk zijn. De stelling zou dan als speciaal geval hebben: Als
"Als $\forall\sigma[t(\sigma) \rightarrow o(\sigma)]$ dan $\forall\sigma[t(\sigma) \rightarrow o(\sigma)]$" dan $\forall\sigma[t(\sigma) \wedge f_P(t)(\sigma) \rightarrow o(\sigma)]$, welke laatste bewering equivalent is met $\forall\sigma\neg\exists\sigma'[\sigma P\sigma']$. We hebben dus voor willekeurige P bewezen dat deze nergens termineert, hetgeen absurd is.

2. Vervangen we de bewering door

Als

$$\text{Als } \forall\sigma[q(\sigma) \wedge f_X(t)(\sigma) \rightarrow f_X(r)(\sigma)]$$
$$\text{dan } \forall\sigma[q(\sigma) \wedge f_{T(X)}(t)(\sigma) \rightarrow f_{T(X)}(r)(\sigma)]$$

dan

$$\forall\sigma[q(\sigma) \wedge f_P(t)(\sigma) \rightarrow f_P(r)(\sigma)],$$

dan zien we dat deze direct is af te leiden met Scott's regel, toegepast op $P(X) \leftrightarrow \forall\sigma[q(\sigma) \wedge f_X(t)(\sigma) \rightarrow f_X(r)(\sigma)]$ waarvan $P(\Omega)$ trivaal is, en de stap: Als $P(X)$ dan $P(T(X))$, juist de hypothese van de gecorrigeerde bewering is.

3.2. *Implementatie van recursie m.b.v. een stapel*

De volgende toepassing wordt gegeven in een model waarin de implementatie van recursie met behulp van een stapel wordt behandeld. Deze toepassing werd geinspireerd door een passage in hoofdstuk 2 van deze syllabus, waarbij de procedure P, gedeclareerd door

```
procedure P; begin A;P;B;P;C end
```

werd geïmplementeerd door

```
stack(4);
while unstack (proces)
do    if proces = 1 then A else
      if proces = 2 then B else
      if proces = 3 then C else {proces = 4}
      begin stack(3); stack(4); stack(2); stack(4); stack(1)
      end
od
```

{De nu volgende beschouwing is niet meer geheel elementair van karakter en

is meer bestemd voor de lezer die reeds enige ervaring met dit type formele redenaties heeft opgedaan.}

We willen een model opbouwen waarin de correctheid van bedoelde implementatie kan worden geformuleerd en bewezen.

We introduceren allereerst een minimale programmeertaal. Iedere statement S is van een van de volgende vier vormen:

1. I , de identiteits- of dummy statement;
2. $A_i;S'$, met $A_i \in \{A_1,\ldots,A_m\}$, de collectie elementaire acties, en S' een statement;
3. $P_j;S'$, met $P_j \in \{P_1,\ldots,P_n\}$, de collectie procedures met declaratie $\{P_j \Leftarrow S_j\}_{j=1}^{n}$, en S' een statement;
4. $S' \cup S''$, met S' en S'' statements.

("$\cup$" kan gebruikt worden om conditionele statements te modelleren, waarvoor we verwijzen naar DE BAKKER & DE ROEVER [3].) We beschikken verder over een aantal stapelprimitieven (vgl. DE BAKKER [2]), namelijk:

1. T_{A_i}, i=1,...,n, en T_{P_j}, j=1,...,n, met als bedoelde interpretatie: Zet (de opdracht tot het later uitvoeren van) A_i en P_j op de stapel.
2. T_S wordt herleid tot T_{A_i} of T_{P_j} door de definities:
   a. $T_I = I$
   b. $T_{S_1 \cup S_2} = T_{S_1} \cup T_{S_2}$
   c. $T_{A_i;S} = T_S;T_{A_i}$, $T_{P_j;S} = T_S;T_{P_j}$
3. p_i, i=1,...,m, en q_j, j=1,...,n, zijn predicaten werkend op de stapel, en juist dan waar, als A_i resp. P_j op de top van de stapel liggen. Als conventie wordt verder toegevoegd dat de predicaten worden opgevat als deelverzamelingen van de eenheidsrelatie I, bestaande uit die paren (σ,σ) waarvoor geldt dat het betreffende predicaat in σ waar is. De predicaten p_i en q_j zijn alle onderling disjunct. Voor $\bigcup_{i=1}^{m} p_i \cup \bigcup_{j=1}^{n} q_j$ schrijven we ook $p_{>0}$. Voorts is $p_0 = I \backslash p_0$, en dus $p_0 \cup p_{>0} = I$, $p_0 \cap p_{>0} = p_0;p_{>0} = \Omega$.
4. Er is een ontstapeloperatie D: verwijder het bovenste element van de stapel. D wordt gekarakterizeerd door $T_{A_i};D = T_{P_j};D = I$.
5. Verder eisen we nog dat

a. $$T_{A_i};p_k = \begin{cases} T_{A_i} & (i=k) \\ \Omega & (i\neq k) \end{cases}$$

$$T_{A_i};q_j = \Omega$$

b. $$T_{P_j};q_\ell = \begin{cases} T_{P_j} & (j=\ell) \\ \Omega & (j\neq\ell) \end{cases}$$

$$T_{P_j};p_i = \Omega$$

c. $T_S;A_i = A_i;T_S$, $T_S;P_j = P_j;T_S$

d. $p_0;A_i = A_i;p_0$, $p_0;P_j = P_j;p_0$

(We hebben overigens niet getracht een minimale collectie van eisen van 1 t/m 5 te vinden.)

De geldigheid van de in het begin van deze subsectie beschreven implementatie kan nu worden geformuleerd als:

$$(6) \qquad \begin{cases} p_0;S \\ = \\ p_0;T_S;p_{>0}{}^*[\bigcup_{i=1}^{m} p_i;D;A_i \cup \bigcup_{j=1}^{n} q_j;D;T_{S_j}] \end{cases}$$

of, in woorden:

Voor iedere statement S, uitgevoerd met initieel lege stapel, geldt dat deze als volgt kan worden geëxecuteerd:

a. Zet de opdracht S uit te voeren op de stapel (T_S).

b. Ga verder als volgt te werk:

(α) Is de stapel leeg, dan klaar. Anders {$p_{>0}$ is waar}.

(β) Onderzoek de top van de stapel.

Ligt hierop een elementaire actie {p_i is waar}, verwijder dan A_i van de stapel, en voer A_i vervolgens uit;

ligt hierop de procedure aanroep P_j {q_j is waar}, verwijder dan P_j van de stapel, en zet de procedure body S_j erop {T_{S_j}}.

(γ) Ga terug naar (α).

We zullen nu het bewijs van (6) geven.

Deel I. Bewijs van $\supseteq$. We passen Scott's regel toe en bewijzen:

Als voor alle S

$$p_0;T_S;X \subseteq p_0;S$$

dan voor alle S

$$p_0;T_S;[p_{>0};\{\bigcup_{i=1}^{m} p_i;D;A_i \cup \bigcup_{j=1}^{n} q_j;D;T_{S_j}\};X \cup p_0] \subseteq p_0;S$$

(Schrijven we π voor $\{\bigcup_{i=1}^{m} p_i;D;A_i \cup \bigcup_{j=1}^{n} q_j;D;T_{S_j}\}$, dan passen we dus Scott's regel toe op $p_{>0} * \pi = \mu X[p_{>0};\pi;X \cup p_0]$.) Bij het verifiëren van de conclusie passen we inductie toe naar de complexiteit van S:

1. $S \equiv I$ ($\equiv$ staat voor: is identiek aan). We hebben dan:
   $p_0;T_S;[\ldots] \equiv p_0;T_I;[\ldots] = p_0;I;[\ldots] = p_0;[\ldots] = p_0;[p_{>0};\pi;X \cup p_0] =$
   $= \Omega;\pi;X \cup p_0;p_0 = p_0 = p_0;I \equiv p_0;S.$

2. $S \equiv A_i;S'$. Dan
   $p_0;T_S;[\ldots] \equiv p_0;T_{A_i};S';[\ldots] = p_0;T_{S'};T_{A_i};[\ldots] =$
   $= p_0;T_{S'};T_{A_i};[p_{>0};\pi;X \cup p_0] = p_0;T_{S'};T_{A_i};p_i;D;A_i;X = p_0;T_{S'};A_i;X =$
   $= A_i;p_0;T_{S'};X \subseteq$ (inductieaanname voor S') $A_i;p_0;S' = p_0;A_i;S' \equiv p_0;S.$

3. $S \equiv P_j;S$. Dan
   $p_0;T_S;[\ldots] = p_0;T_{P_j};S';[\ldots] = p_0;T_{S'};T_{P_j};[\ldots] =$
   $= p_0;T_{S'};T_{P_j};[p_{>0};\pi;X \cup p_0] = p_0;T_{S'};T_{P_j};q_j;D;T_{S_j};X = p_0;T_{S'};T_{S_j};X =$
   $= p_0;T_{S_j};S';X \subseteq$ (inductieaanname voor $S_j;S'$) $p_0;S_j;S' = p_0;P_j;S' \equiv$
   $\equiv p_0;S.$

4. $S \equiv S' \cup S'$. Overgelaten aan de lezer.

Uit 1 t/m 4 volgt dat het bewijs van deel I m.b.v. Scott's regel is voltooid.

Deel II. Bewijs van $\subseteq$, ofwel:

(7) $$p_0;S \subseteq p_0;T_S;p_{>0} * \pi.$$

We gebruiken een tweetal hulpbeweringen (8) en (9):

(8) $$T_{S_1};p_{>0} * \pi;T_{S_2};p_{>0} * \pi \subseteq T_{S_1};S_2;p_{>0} * \pi.$$

Zij, voor $S_j = S_j(P_j)$, $S_j(S)$ het resultaat van substitutie van S voor ieder voorkomen van P_j in S_j. Dan

(9) $$S_j(T_S;p_{>0} * \pi) \subseteq T_{S_j(S)};p_{>0} * \pi.$$

Resultaat (8) volgt m.b.v. een redenatie lijkend op die van deel I, resultaat (9) is uit resultaat (8) af te leiden. Uitwerking hiervan wordt achterwege gelaten. We gebruiken (9) voor het bewijs van (7) als volgt: We passen weer inductie toe naar de complexiteit van S. We geven alleen het geval dat $S \equiv P_j;S'$, de andere gevallen zijn niet moeilijker. We willen aantonen dat $p_0;S \equiv p_0;P_j;S' \subseteq p_0;T_{P_j};S';p_{>0} * \pi$. We tonen eerst aan dat $P_j \subseteq T_{P_j};p_{>0} * \pi$, als volgt:

$$S_j(T_{P_j};p_{>0} * \pi) \overset{(9)}{\subseteq} T_{S_j(P_j)};p_{>0} * \pi = T_{S_j};p_{>0} * \pi \overset{df\ \pi}{=}$$

$$= T_{P_j};p_{>0} * \pi.$$

Dus, op grond van het kleinste dekpuntresultaat (en gebruik makend van het feit dat $\bigcap\{X:T(X)=X\} = \bigcap\{X:T(X) \subseteq X\}$, zie bv. DE BAKKER [1]) vinden we dat $P_j \subseteq T_{P_j};p_{>0} * \pi$. Verder geldt $p_0;S' \subseteq p_0;T_{S'};p_{>0} * \pi$, op grond van de inductieaanname. Dus: $p_0;P_j;S' \subseteq p_0;T_{P_j};p_{>0} * \pi;S' = p_0;T_{P_j};p_{>0} * \pi;p_0;S' \subseteq$ $\subseteq p_0;T_{P_j};p_{>0} * \pi;p_0;T_{S'};p_{>0} * \pi = p_0;T_{P_j};p_{>0} * \pi;T_{S'};p_{>0} * \pi \overset{(9)}{\subseteq}$ $\subseteq p_0;T_{P_j};S';p_{>0} * \pi$.

Hiermede is het bewijs van (7), en dus ook van (6), voltooid.

4. VERDERE TOEPASSINGEN

De in secties 2 en 3 besproken voorbeelden vormen slechts een klein deel van de problemen waarop Scott's regel toegepast is. Vele andere voorbeelden zijn te vinden in DE BAKKER [1,2], DE BAKKER & DE ROEVER [3], MANNA, NESS & VUILLEMIN [10], MANNA & VUILLEMIN [11], en elders.
De theorie is ook toegepast op het oplossen van meer geavanceerde problemen. Zo is door HITCHCOCK & PARK [8] een diepgaande studie gemaakt van methoden om terminatie van programma's te bewijzen met behulp van zogenaamde *well founded* relaties. In DE BAKKER & MEERTENS [4,5] is de volledigheid van de inductieve assertiemethode aangetoond via een verfijning van de structuur hiervan. Hierbij speelt Scott's regel ook weer een belangrijke rol.
Een uitbreiding van de axiomatische theorie uit DE BAKKER & DE ROEVER [3], met diverse toepassingen, wordt gegeven door De Roever in een binnenkort te verschijnen rapport.
Ander onderzoek, dat overigens nog in een beginstadium verkeert, betreft

toepassing van Scott's theorie op onderzoek van parallel programmeren, bv. in CADIOU & LEVY [6]. Recentelijk is door Park voorgesteld hierbij maximale dekpunten te gebruiken, hetgeen diverse nieuwe perspectieven opent.

LITERATUUR

[1] BAKKER, J.W. de, *Recursive Procedures*, Mathematical Centre Tracts 24, Mathematisch Centrum, 1971.

[2] BAKKER, J.W. de, *Recursion, Induction and Symbol Manipulation*, in: Mathematical Centre Trachts 37, Mathematisch Centrum, 1971, p.1-32.

[3] BAKKER, J.W. de & W.P. de ROEVER, *A calculus for recursive program schemes*, in: *Automata, Languages and Programming* (M. Nivat, ed.), North-Holland, 1973, p.167-196.

[4] BAKKER, J.W. de & L.G.L.T. MEERTENS, *Simple recursive program schemes and inductive assertions*, Mathematical Centre Report MR 142, Amsterdam, 1972.

[5] BAKKER, J.W. de & L.G.L.T. MEERTENS, *On the completeness of the inductive assertion method*, Mathematical Centre Report IW 12, Amsterdam, 1973.

[6] CADIOU, J.M. & J.L. LÉVY, *Mechanizable proofs about parallel processes*, Proc. 14th Annual Symposium on Switching and Automata Theory, 1973.

[7] DIJKSTRA, E.W., *An axiomatic basis for programming language constructs*, te verschijnen.

[8] HITCHCOCK, P. & D. PARK, *Induction rules and proofs of termination*, in: *Automata, Languages and Programming* (M. Nivat, ed.), North-Holland, 1973, p.225-251.

[9] KLEENE, S.C., *Introduction to Metamathematics*, North-Holland, 1952.

[10] MANNA, Z., S. NESS & J. VUILLEMIN, *Inductive methods for proving properties of programs*, C.ACM, 16 (1973) 491-502.

[11] MANNA, Z. & J. VUILLEMIN, *Fixpoint approach to the theory of computation*, C.ACM, 15 (1972) 528-536.

[12] SCOTT, D. & J.W. DE BAKKER, *A theory of programs*, Notes of an IBM Seminar, unpublished (1969).

MATHEMATICAL CENTRE SYLLABUS 21, 1975, VIII, 127-138.

COMPLEXITEIT VAN ALGORITMEN

P. van EMDE BOAS

1. INLEIDING

Stel dat u bij het schrijven van een programma meer dan eens het maximum van drie reële getallen wenst te doen uitrekenen. U besluit hiertoe een procedure "max3" in het leven te roepen, die er als volgt uit zou kunnen zien:

```
proc max3 = (real x,y,z) real:
    (real m;
    (x ≥ y  and  y ≥ z | m:= x)
    (y ≥ x  and  y ≥ z | m:= y)
    (z ≥ x  and  z ≥ y | m:= z)
     m);
```

Hiertoe aangezet door het hoongelach van uw naaste collega's, zult u waarschijnlijk de bovenstaande procedure verwerpen en vervangen door iets in de geest van

```
proc max3 = (real x,y,z) real:
    (real m;
    m:= (x ≥ y|x|y);
    m:= (z ≥ m|z|m);
    m)
```

of nog liever door

```
proc max3 = (real x,y,z) real:
    (x ≥ y|(x ≥ z|x|z)|(y ≥ z|y|z)).
```

Wat zijn nu de argumenten om de ene procedure te prefereren boven een andere? De toeschouwer zal zich snel kunnen overtuigen, dat ieder der drie versies mathematisch gezien correct is; de berekende waarde is het maximum

van x, y en z. Het verschil tussen de procedures is evenwel gelegen in de hoeveelheid benodigd werk. Wat de eerste versie vermag in zes vergelijkingen en één tot drie assignments kan versie twee met twee vergelijkingen en twee assignments en versie drie met twee vergelijkingen sec.
Een voor de hand liggende vraag is natuurlijk of versie drie nu ook "optimaal" is. Kunnen we het met nog minder bewerkingen? We zullen dan wel het probleem goed moeten definiëren, door vast te leggen wat de toegestane bewerkingen zijn. Immers, als de te gebruiken computer of programmeertaal een procedure als max3 als elementaire opdracht kent, is één bewerking voldoende. Als we denken in termen van vergelijkingen tussen de verschillende argumenten, dan zijn minstens twee vergelijkingen nodig, omdat anders de drie argumenten niet met elkaar in verband kunnen worden gebracht.

Als we ons echter richten op het aantal vergelijkingen maar willekeurig veel rationale bewerkingen van de argumenten voor, tussen en na de vergelijkingen toestaan, wordt het probleem verre van triviaal. Waarschijnlijk zijn ook dan nog twee vergelijkingen nodig, maar het bewijs hiervoor vereist vermoedelijk de nodige kennis van de algebraïsche meetkunde.

Het voorafgaande moge dienen als illustratie van de problematiek bestudeerd onder de titel "complexiteit van algoritmen". We hebben hier te maken met een "jong" vak (± 10 jaren oud), dat snel bezig is tot een substantieel onderdeel der theoretische informatica uit te groeien. Representatief voor deze snelle groei is bijv. de in 1972 door IRLAND en FISCHER samengestelde literatuurlijst over het vak van RUSTIN [16]: lengte: 57 pagina's.
In de komende anderhalf uur zal ik trachten u een indruk van het vak te geven. Inspiratiebron hierbij is het verslag van een symposium, gehouden 20-22 maart 1972 op het IBM Thomas J. Watson Research Center, Yorktown Heights, NY, MILLER & THATCHER [9].

2. ANALYSE VAN ALGORITMEN VS. ABSTRACTE COMPLEXITEITSTHEORIE

Er bestaat binnen het kader van de complexiteitstheorie een duidelijke onderverdeling in twee terreinen, die ik zal aanduiden als *analyse van algoritmen* resp. *abstracte complexiteitstheorie*. De verhouding tussen deze twee terreinen is ongeveer die tussen een wiskundig ingenieur en een wiskundige. Waar de beoefenaar van de analyse van algoritmen er op uit is om

boven- en ondergrenzen van de complexiteit van een concreet probleem aan te geven, houdt de abstracte complexiteits-theoreticus zich bezig met de wiskundige analyse van de structuur die ontstaat door aan een abstract gedefinieerd model van berekenbaarheid een complexiteitsbegrip toe te voegen. Deze abstracte theorie is voortgekomen uit het werk van BLUM [2]. Zij kan beschouwd worden als een onderdeel van de recursietheorie, die op zijn beurt ontstaan is uit de behoefte aan een formele definitie van het begrip *berekenbaarheid*.

Gedurende de dertiger jaren van deze eeuw ontstonden naast elkaar een groot aantal mathematische definities van het begrip berekenbaarheid, waarvan de bekendste zijn de definities volgens TURING (Turing machine), KLEENE (stelsels van definiërende vergelijkingen) en MARKOV (Markov algoritmen). Wiskundig blijkt de klasse van berekenbare functies *) voor al deze definities gelijk te zijn. Bovendien blijken alle intuïtief berekenbare functies ook formeel berekenbaar te zijn. De zogeheten *These van Church* spreekt uit, dat dit formele begrip samenvalt met het intuïtieve begrip van berekenbaarheid.

Een volledig machine-onafhankelijke beschrijving van een "universum voor berekeningen" is gegeven door ROGERS [14] in het begrip *effectieve enumeratie*. Dit is een lijst van partiële functies $(\phi_i)_i$, programma's genaamd, die voldoet aan de volgende eigenschappen:

(i) alle functies ϕ_i zijn partieel recursief en alle partieel recursieve functies komen voor in de lijst;

(ii) de functie $u(<i,x>) = \phi_i(x)$ **) is partieel recursief (universele machine axioma);

(iii) er bestaat een totale (d.w.z. overal gedefinieerde) recursieve functie $S(n,m)$, zodat $\phi_n(<m,x>) = \phi_{S(n,m)}(x)$. (S-n-m axioma).

Het S-n-m axioma maakt het mogelijk een deel van het argument te interpreteren als een parameter, die desgewenst effectief meegegeven kan worden als onderdeel van het programma.

Het begrip effectieve enumeratie is door BLUM als volgt uitgebreid tot het begrip *complexiteitsmaat*. Naast de lijst $(\phi_i)_i$ beschouwen we een twee-

---

*) Onder een functie verstaan we in het vervolg een partiële functie van $\mathbb{N} \to \mathbb{N}$; $0 \in \mathbb{N}$.

**) $<x,y>$ is een vaste recursieve paarfunctie, monotoon stijgend in x en y, die het mogelijk maakt, waar gewenst, $\mathbb{N}$ op te vatten als $\mathbb{N}^2$, $\mathbb{N}^3$ enz.

de lijst partieel recursieve functies $(\Phi_i)_i$ (rekentijden), die voldoet aan de volgende twee Blum axioma's:

(B1) de functies ϕ_i en Φ_i hebben hetzelfde domein: $\phi_i(x)$ convergeert d.e.s.d.a. $\Phi_i(x)$ gedefinieerd is;

(B2) het is beslisbaar of een programma ϕ_i op argument x binnen y stappen klaar komt of niet: de relatie $\Phi_i(x) \leq y$ is recursief.

Voor de hand liggende voorbeelden van complexiteitsmaten zijn het aantal stappen van de berekening of het aantal gebruikte cellen op de band van een universele Turing machine.
Binnen de abstracte theorie kan men nu wiskundig de juistheid verifiëren van de ervaringsfeiten, dat ieder programma willekeurig inefficiënt gemaakt kan worden en dat iedere totale functie berekend kan worden door een programma, waarvan de rekentijd monotoon stijgend is. De abstracte theorie ontleent zijn belang echter aan een aantal stellingen, die de hoop om ooit effectief alle programma's te kunnen "optimaliseren", de grond in boren. Als representatief voorbeeld van een dergelijke stelling noem ik de versnellingsstelling (speed-up theorem) van BLUM [1].

Zij R een totale functie zodat $R(x,y) \geq y$. Dan bestaat er een totale functie f met de eigenschap, dat voor ieder programma ϕ_i voor f (d.w.z. ϕ_i = f als functie) een programma ϕ_j bestaat, zodat voor bijna alle argumenten x geldt dat $R(x,\Phi_j(x)) \leq \Phi_i(x)$.

Denken we bij R(x,y) aan een functie als 2^{x+y}, dan zien we dat ieder programma ϕ_i voor f vervangen kan worden door één dat binnen log Φ_i stappen termineert (afgezien dan van een aantal beginargumenten). Als u nu ook nog weet, dat van dit programma ϕ_j het bestaan bewezen wordt, zonder aan te geven hoe ϕ_j gevonden moet worden (dat is in het algemeen ook onmogelijk), dan ziet u hoe futiel het is om voor alle functies een optimaal programma te willen construeren.

Een andere stelling, de gaten-stelling, (gap-theorem) BORODIN [3] laat zien, zien, dat het onmogelijk is een methode te geven om effectief en uniform de capaciteit van een installatie te vergroten, door de maximale rekentijden systematisch te verruimen; er zijn altijd begrenzingen, zodat na verruiming geen enkele functie berekenbaar wordt die het niet al was.

Formeel: zij C_t de klasse functies berekenbaar door een programma ϕ_j met

rekentijd Φ_j bijna overal begrensd door t ($\Phi_j(x) \leq t(x)$ voor alle x op eindig veel uitzonderingen na). De gaten-stelling spreekt uit, dat voor iedere totale functie R met $R(x,y) \geq y$ een grens t te vinden is, zodanig dat voor t', gedefinieerd door $t'(x) = R(x,t(x))$, geldt $C_t = C_{t'}$.

Ondanks deze sombere profetieën, is de meer concrete leer van de analyse van algoritmen gezegend met de attentie van een schare van beoefenaren, die de thans uitstervende club van abstractelingen in omvang ruimschoots overtreft. We zullen deze laatste groep dan ook maar verder met rust laten.

3. ASPECTEN VAN DE ANALYSE VAN ALGORITMEN

De beoefening van de analyse van algoritmen laat zich als volgt schetsen:

1) Er wordt een mathematisch probleem uitgezocht, waarvoor een algoritmische oplossingsmethode bestaat (bijv. omdat het een "eindig" probleem is).

2) De bestaande oplossingsmethoden worden geïnventariseerd en vergeleken naar efficiëntie.

3) Er wordt gezocht naar een zinvolle maat om de moeilijkheid van het probleem te meten.

4) Men probeert algoritmen te vinden die, gemeten in deze maat, op niet triviale wijze efficiënter zijn.

5) Indien na lang speuren verdere verbeteringen onvindbaar blijken, zal men pogen te bewijzen dat het beste programma "optimaal" is, door voor het onderhavige probleem een ondergrens te bewijzen.

We zullen ons in het vervolg vooral bezig houden met de laatste drie onderwerpen.

4. COMPLEXITEITSMATEN

Zoals boven genoemd (sectie 3, sub 3) vraagt ieder wiskundig probleem om een zinvolle complexiteitsmaat. In het algemeen kiest men hiervoor niet een abstracte complexiteitsmaat als bedoeld in sectie 2, maar het aantal

bewerkingen van een zeker type dat vaak zeer sterk probleemgericht is.

In het geval van polynoomevaluatie bekijkt men in het algemeen het aantal rationale bewerkingen, als optellingen, vermenigvuldigingen en delingen, hoewel bij multiple lengte bewerkingen vaak de optellingen worden verwaarloosd als te zijn "goedkope" bewerkingen.
Bij discussie over sorteerprogramma's is men vooral geïnteresseerd in het aantal uit te voeren verwisselingen en/of vergelijkingen.
Een geschikte maat voor numerieke problemen, waar het om het rendement van een iteratieve methode handelt, is de zogeheten *efficiency index* van OSTROWSKI [11]. Dit getal heeft de vorm $\gamma = (\log p)/M$, waarbij p de convergentie-orde en M het aantal bewerkingen per iteratieslag is.

Geen van de bovengenoemde maten is een complexiteitsmaat in de abstracte zin. Om een minder probleemgerichte indruk van een probleem te krijgen, worden dan ook regelmatig programma's geschetst om het vraagstuk op te lossen op een zeker type universele Turing machine of een soort registerautomaat. Deze beschouwingen, die vaak hoogst inexact zijn, leiden tot ordeschattingen van de complexiteit van het probleem.
Voor eindige problemen kan men kijken naar de zogeheten *combinatorische complexiteit*. Dit is ruwweg het minimale aantal binaire logische elementen in een electronische schakeling (and, or schakelingen en invertoren), die gegeven een codering van een instantie van het probleem op de invoerdraden een codering van de oplossing op de uitvoerdraden doet verschijnen.

Aangezien een computer die T basiscycli doorloopt, kan worden vervangen door T copieën van de computer, die in serie geschakeld zijn en zo een eindige automaat vormen, levert de combinatorische complexiteit van een probleem een ondergrens voor het product van de rekentijd en de geheugenomvang voor een algoritme die het probleem oplost SAVAGE (cf. [17]).

5. VOORBEELDEN VAN PROBLEMEN

Om een bovengrens van een probleem aan te geven is in feite de enige methode het aangeven van een algoritme en het schatten van de door deze algoritme gevraagde rekentijd. Bij deze constructie zijn alle onderdelen van het menselijk vernuft toegestane hulpmiddelen.
Naast een schatting van de maximale rekentijd (worst case analysis) is ook

de gemiddelde rekentijd een geliefd studieobject.
Een ordeschatting van de rekentijd wordt soms verkregen door beschouwing van algoritmen, die een probleem oplossen door recursieve aanroepen van kleinere instanties van hetzelfde probleem. Dit leidt tot recurrente betrekkingen voor de complexiteit van het probleem.

5.1. *Sorteren*

Een recurrentiebeschouwing als boven gesuggereerd laat vrij eenvoudig inzien, dat het sorteren van n "records" naar een lineair geordende "key" een rekentijd van de orde n log n vraagt. Er is ook een ondergrens van dezelfde orde bewezen, zoals men kan nalezen in KNUTH [8]. Verbeteringen moeten hier dus worden gezocht in de constante factor C, waarvoor Cn log n de rekentijd voor sorteren aangeeft.

5.2. *Matrixvermenigvuldiging*

Bij dit probleem bekijken we (zoals bij alle algebraïsche problemen) als maat het aantal (rationale) bewerkingen, en wel in het bijzonder de vermenigvuldigingen.
Botweg toepassen van de definitie $c_{ij} = \sum_{k=1} a_{ik} b_{kj}$ levert een algoritme, die n^3 vermenigvuldigingen vraagt. Op het eerste gezicht lijkt dit aantal niet te verbeteren. In [18] laat STRASSEN echter zien hoe twee 2×2 matrices vermenigvuldigd kunnen worden in 7 vermenigvuldigingen*). Door dit proces recursief toe te passen op blokmatrices in plaats van getallen, kan men een matrix-vermenigvuldigingsroutine creëren met rekentijd van de orde $O(n^{2\log 7}) \approx O(n^{2.8})$, aangezien het aantal optellingen niet harder blijkt te groeien dan het aantal vermenigvuldigingen. Niemand weet of deze exponent $^2\log 7$ optimaal is. Voor 2×2 matrices is 7 vermenigvuldigingen bewezen nodig te zijn, terwijl voor n = 3 dit aantal 22, 23 of 24 bedraagt (het exacte aantal is voorzover mij bekend nog niet bepaald).

---

*) Dit "mirakel" wordt als volgt teweeg gebracht. Zij $A = \begin{pmatrix} a11 & a12 \\ a21 & a22 \end{pmatrix}$, $B = \begin{pmatrix} b11 & b12 \\ b21 & b22 \end{pmatrix}$. Stel $C = A \cdot B = \begin{pmatrix} c11 & c12 \\ c21 & c22 \end{pmatrix}$. Dan laten de getallen c_{ij} zich als volgt uitrekenen: h1:= (a11+a22)(b11+b22); h2:= (a21+a22) b11; h3:= a11(b12-b22); h4:= a22(-b11+b21); h5:= (a11+a12) b22; h6:= (-a11+a21)(b11+b12); h7:= (a12-a22)(b21+b22); c11:= h1+h4-h5+h7; c21:= h2+h4; c12:= h3+h5; c22:= h1+h3 - h2+h6. Voor een "verklaring" van een en ander zie FIDUCCIA [4].

Er is alleen een triviale ondergrens ($O(n^2)$), aangezien alle argumenten gelezen moeten worden. De complexiteit van het probleem ligt dus in tussen $O(n^2)$ en $O(n^{2.8})$.

5.3. *Polynoomevaluatie*

Het evalueren van een polynoom is intensief onderzocht. Voor het n^e-graads polynoom levert het Horner schema een algoritme in n optellingen en n vermenigvuldigingen. Moet eenzelfde polynoom op vele argumenten worden geëvalueerd, dan kan het gunstig zijn bij voorbaat aan de coëfficiënten te knoeiten en daarmede n/2 vermenigvuldigingen wegwerken (preconditioning).

5.4. *Selectie*

Een ander probleem, waarbij de bovengrens weer verrassend laag blijkt te zijn, betreft het selectieprobleem. Uit een lijst van k ongesorteerde objecten wordt gezocht het j^e object, geordend naar grootte ($1 < j < k$).

5.5. *Benadering nulpunt*

Een voorbeeld van de numerieke toepassingen betreft het benaderen van een nulpunt van een polynoom, met behulp van een rationaal iteratieproces. Men denke aan een oneindige-precisie systeem. De "dure" operaties zijn vermenigvuldiging van twee, of deling door multi-lengte getallen, en alleen die operaties worden geteld. Een maat voor de efficiency is, zoals we reeds eerder vermeldden, de grootheid $\gamma = (\log p)/M$, waarbij p de convergentie-orde en M het aantal getelde "dure" operaties is.
In deze maat blijkt de efficiency van Newton-iteratie voor een kwadratisch polynoom gelijk 1 te worden. Newton op een kubisch polynoom levert $\gamma = 1/2$.
Een inverse interpolatie-iteratie-methode gegeven door TRAUB, die gebruik maakt van het n^e-graads polynoom en zijn eerste e-1 afgeleiden in de m laatst berekende benaderingen $x_{1-i},\ldots,x_{i-m}$ blijkt voor iedere $\delta > 0$ voor n,m voldoende groot te voldoen aan $\gamma > \log(e+1-\delta)/\sqrt{2en}$. Deze grootheid is optimaal voor $e = 4$, hetgeen voor geschikt gekozen m leidt tot $\gamma > .82\ n^{-1/2}$.
Gebruik makende van de stelling van LIOUVILLE kan men bewijzen, dat voor iedere interpolatiemethode geldt $\gamma \leq 1$. Deze resultaten zijn ontleend aan PATTERSON [12].

5.6. *Grafentheorie en combinatoriek*

Een belangrijk toepassingsgebied zijn de algoritmen van combinatorische en grafentheoretische aard. We kunnen hierbij denken aan problemen als het vinden van kortste afstanden in een graaf, minimale opspannende bomen, het handelsreizigersprobleem, knapzakprobleem. Vele van deze problemen staan bekend als onhanteerbaar vanwege een exponentiële groei van de rekentijd. Het ontbreekt op dit terrein aan niet-triviale ondergrenzen.
Sommige deelproblemen kunnen soms bijzonder efficiënt worden opgelost. Als voorbeeld noem ik een algoritme, beschreven door HOPCROFT & TARJAN [6] om in O(V log V) stappen vast te stellen, of tween planaire grafen met V hoekpunten isomorf zijn of niet.
Een ander voorbeeld is het vinden van een minimale equivalentierelatie $\equiv$ die consistent is met een lijst van n aannamen van de vorm $x \equiv y$ voor x,y elementen van een gegeven verzameling V. Een mogelijke methode is de representatie van de verzameling als een bos, aanvankelijk bestaande uit losse triviale bomen, waarbij voor iedere assertie $x \equiv y$ de twee deelbomen die x en y bevatten, zonodig in elkaar worden gehangen. Zonder verdere voorzorgen te treffen leidt dit tot een $O(n^2)$ algoritme, maar dankzij een aantal handigheden kan dit worden gereduceerd tot O(n loglog n). Deze resultaten zijn ontleend aan FISHER [5].
De grootste moeilijkheden rusten in het algemeen bij het geven van niet triviale ondergrenzen. Als illustratie verwijs ik naar de situatie bij het geven van ondergrenzen voor de combinatorische complexiteit van Boolese functies. Zoals bekend bestaan er 2^{2^n} verschillende Boolese functies van n Boolese variabelen. Op grond van telargumenten blijkt, dat het overgrote deel hiervan combinatorische complexiteit $O(2^n)$ moet hebben. Doch het vereist veel vindingrijkheid om een serie functies te construeren waarvoor een ondergrens van de combinatorische complexiteit kwadratisch in n toeneemt NECIPORUK [10].

6. P = NP ?

De als titel gebruikte kreet dekt het centrale onopgeloste probleem binnen de concrete complexiteitstheorie. Zoals we reeds eerder opmerkten is een frequent gebruikt model in de theorie het model van de deterministische Turing machine, of een van zijn varianten (met meerdere banden en leeskop-

pen enz.). Voor problemen, waarvoor de rekentijd harder dan exponentieel groeit, is de precieze keuze van het model niet zo belangrijk, aangezien de verschillende modellen elkaar kunnen simuleren, op straffe van een vaak polynomiaal begrensde "overhead".
Een ander model is dat van de niet-deterministische Turing machine. Deze machine kan, in tegenstelling tot zijn deterministische broertje, onderweg een aantal arbitraire keuzen maken. Hij wordt gebruikt om invoergegevens te accepteren of te verwerpen, waarbij een invoer geaccepteerd wordt als er een mogelijke berekening bestaat, die leidt tot een accepterende toestand.
Alhoewel hier dus niet meer sprake is van een algoritme in de echte zin van het woord, is dit toch een geaccepteerd model, temeer daar de berekening van een niet-deterministische machine zich laat simuleren door een deterministische machine. Deze laatste simulatie vereist echter exponentieel meer rekentijd, althans zo lijkt het op het eerste gezicht.
Men zou zich dus kunnen voorstellen, dat er verzamelingen van invoergegevens (talen) zijn, waarvoor een niet-deterministische herkenning polynomiale rekentijd vraagt (in termen van de lengte van de invoer), terwijl deze zelfde verzameling niet in minder dan exponentiële rekentijd te herkennen is door een deterministische machine. Helaas is tot op heden voor het ontbreken van niet-triviale ondergrenzen nog niet één voorbeeld voor zo'n verzameling gegeven. Dit heeft geleid tot het probleem "P = NP ?". P (NP) staat hierbij voor de klasse talen, die (niet-)deterministisch herkend kunnen worden binnen polynomiale tijd.
Er is een groot aantal combinatorische problemen, die zich laten formuleren als een ja-nee vraag en die binnen zo'n formulering op te vatten zijn als een taal in NP. Denken we bijvoorbeeld aan een probleem als 0-1 lineaire integer programmering. Gezocht wordt naar een 0-1 oplossing van de vergelijking $Cx = d$, waarbij C en d een matrix resp. vector van gehele getallen zijn. De bijbehorende taal bestaat uit de (coderingen van) paren C, d waarvoor deze oplossing bestaat; deze taal is bevat in NP. In het artikel van R.M. KARP, *Reducibility among combinatorial problems* [7], is een lijst van 21 van deze problemen opgenomen, waaronder oude bekenden als het vinden van maximale klieken, knapzakproblemen, bepaling van chromatische getallen van grafen, het vinden van Steiner-bomen en het handelsreizigersprobleem.

Het opvallende fenomeen is, dat al deze problemen zich tot elkaar laten reduceren op deterministische wijze binnen polynomiale tijd. Dit wil zeggen dat voor ieder van deze problemen een vertaling bestaat, die een instantie van het probleem (in polynomiale tijd) overvoert in een equivalente instantie van een standaardtype (zoals 0-1 lineaire integer programmering). Het gevolg hiervan is, dat deze problemen hetzij allemaal binnen polynomiale tijd kunnen worden opgelost, hetzij geen van alle.

De genoemde problemen zijn bovendien *volledig*. Dit wil zeggen, dat ieder probleem in NP zich in polynomiale tijd laat reduceren tot een probleem uit de lijst. Hieruit volgt dat P = NP dan en slechts dan geldt als alle problemen in de lijst in polynomiale tijd oplosbaar zijn.
Ook in 1973 is de oplossing van dit tartende probleem (ondanks enkele geruchten die niet op waarheid bleken te berusten) nog niet opgelost. De meningen over de vermoedelijke uitkomsten zijn verdeeld. Getuige de weddenschap tussen BLUM en PATTERSON, waarbij de errste $ 100.- op $P \neq NP$ zet tegen $ 1.- voor P = NP, bestaat er wel een voorkeur voor het ongelijkteken. Doch wie weet hoe een duivelskunstenaar uit de praktijk verzamelde wiskundigen nog eens voor aap zet met een polynomiale algoritme voor het handelsreizigersprobleem?

LITERATUUR

[1] BLUM, M., *A machine independent theory of the complexity of recursive functions*, J.ACM, 14 (1967) 322-336.

[2] BLUM, M., e.a., *Time bounds for selections*, J.CSS, 7 (1973) 448-461.

[3] BORODIN, A., *Computational complexity and the existence of complexity gaps*, J.ACM, 19 (1972) 158-174.

[4] FIDUCCIA, C.M., *On obtaining upper bounds on the complexity of matrix multiplication*, in [MT 72].

[5] FISCHER, M.J., *Efficiency of equivalence algorithms*, in [MT 72].

[6] HOPCROFT, J.E. & R.E. TARJAN, *Isomorphism of planar graphs*, in [MT 72].

[7] KARP, R.M., *Reducibility among combinatorial problems*, in [MT 72].

[8] KNUTH, D., *The art of computer programming. Sorting & Searching*, Vol. 3, Addison Wesley, Reading (Mass.), 1973.

[9] MILLER, R.E. & J.W. THATCHER (eds), *Complexity of computer computations*, Plenum Press, N.Y., 1972.

[10] NECIPORUK, E.I., *A boolean function*, Soviet Math. Dokl., 7 (1966) 999-1000.

[11] OSTROWSKI, A.M., *Solutions of equations and systems of equations*, Academic Press, New York, 1966.

[12] PATTERSON, M.S., *Efficient iterations for algebraic numbers*, in [MT 72].

[13] PRATT, V.R. & F.F. YAO, *On lower bounds for computing the i^{th} largest element*, in Proc. 14^{th} SWAT Symp., Iowa City, Oct. 15-17 1973, pp.70-81.

[14] ROGERS JR., H., *Gödel numberings of partial recursive functions*, J.S.L., 23 (1958) 331-341.

[15] REINGOLD, E.M. & A.I. STOCKS, *Simple proofs of lower bounds for polynomial evaluations*, in [MT 72].

[16] RUSTIN, R. (ed), *Computational complexity*, Courant Comp.Sci.Symp. 7, Algorithmic Press Inc., N.Y., 1973.

[17] SAVAGE, J.E., *Computational work and time*, in [Ru 73].

[18] STRASSEN, V., *Gaussian elimination is not optimal*, Num. Math., 13 (1969) 354-356.

MATHEMATICAL CENTRE SYLLABUS 21, 1975, IX, 139-150.

OVER HET NUT VAN TRANSPARANTE PROGRAMMA'S

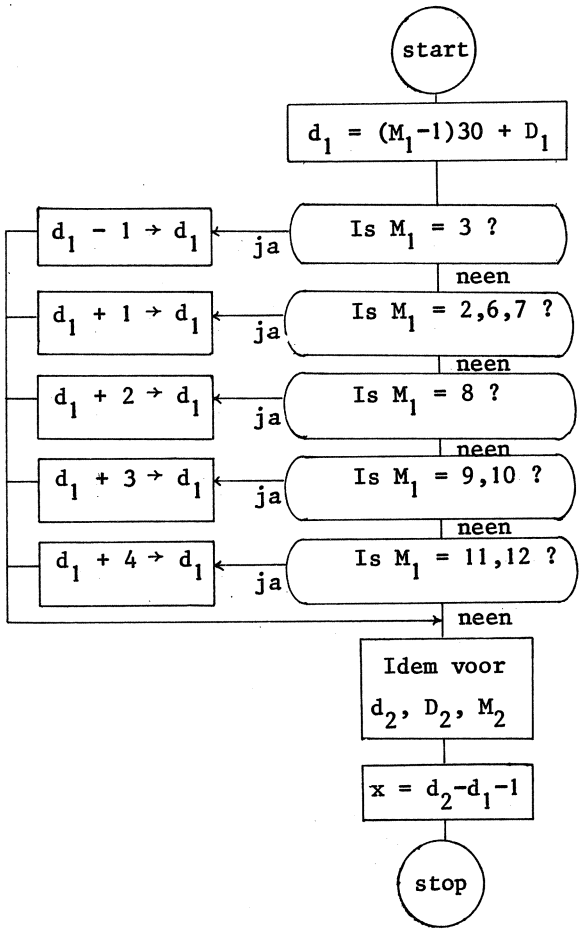
L. AMMERAAL

1. INLEIDING

Bij het programmeren dient correctheid met alle mogelijke middelen te worden bevorderd. Het ideale middel is het geven van een correctheidsbewijs. Omdat het in de praktijk nog niet doenlijk is van iedereen die een programma schrijft te eisen dat hij de correctheid ervan bewijst, verdient het vooralsnog aanbeveling ook enige aandacht te schenken aan minder pretentieuze middelen. Als een correctheidsbewijs, om welke reden dan ook, praktisch niet haalbaar is, dan is het gewenst dat het programma een betrouwbare indruk maakt. Het zal dikwijls voorkomen dat een programmeur niet aan een formeel correctheidsbewijs toekomt, maar gewoon "ziet" dat zijn programma correct is. Komt hij wel aan zo'n bewijs toe, dan dient dit niet onnodig ingewikkeld te zijn. In beide gevallen heeft het dus zin te trachten een programma zodanig te construeren, dat de correctheid direct of via een zo kort mogelijke weg is te doorzien. Vaak wint een programma aan doorzichtigheid als we het kunnen vereenvoudigen, hetgeen aan de hand van een drietal voorbeelden zal worden toegelicht.

2. VEREENVOUDIGDE PROGRAMMA'S

Het eerste voorbeeld is zeer elementair. Het is ontleend aan een oud artikel over programmeren van M. EUWE [1]. Gevraagd wordt het aantal dagen te berekenen, dat ligt tussen twee gegeven data (D_1, M_1) en (D_2, M_2) van hetzelfde jaar dat geen schrikkeljaar is. Euwe geeft het volgende blokschema als oplossing.



Het verifiëren van de correctheid van dit programma is een enigszins vervelend karweitje, waarbij men moet oppassen voor rekenfouten. Het volgende ALGOL 68 - programma laat zien, dat het veel gemakkelijker kan.

```
begin int maandduur = (31,28,31,30,31,30,31,31,30,31,30,31);
      [1:12] int vooraf;
      int d1, m1, d2, m2;
      vooraf[1]:= 0;
      for i to 11
      do vooraf[i+1]:= vooraf[i] + maandduur[i]
      od;
      read ((d1,m1,d2,m2));
      print ((vooraf[m2]+d2) - (vooraf[m1]+d1)-1)
end
```

De correctheid van deze oplossing is veel gemakkelijker in te zien. Het tweede programma is minder gekunsteld dan het eerste; er wordt geen gebruik gemaakt van de toevallige omstandigheid dat het aantal dagen van een maand in de buurt van de 30 ligt. Als men eenmaal op het idee komt een array te definiëren waarin het aantal dagen voorafgaande aan elke maand genoteerd staat, dan schrijft men het tweede programma zo op. Voor de eerste oplossing moet men eerst op het idee komen 30-vouden te gebruiken en vervolgens voor elke maand i de correctie berekenen, die m.b.v. de terminologie van de tweede oplossing gedefinieerd wordt door vooraf[i] - 30(i-1). Dit vervelende handrekenwerk moet gedaan worden bij het schrijven van het programma èn door iedereen die het programma kritisch wil lezen. Letten we alleen op de correctheid, dan is de eerste oplossing even goed als de tweede: beide oplossingen zijn correct. We zien dus al aan dit eenvoudige voorbeeld dat we aan een programma niet alleen de eis moeten stellen dat het correct is, maar dat we bovendien moeten verlangen dat het zo *transparant* mogelijk is.

Een machtig middel om tot een meer transparant programma te komen is *recursie*. Een voorbeeld hiervan is het volgende probleem, waarbij we gebruik maken van een ALGOL 60-achtige notatie. Gegeven zijn de gehele getallen n, ℓ_1, u_1, ℓ_2, u_2, ..., ℓ_n, u_n ($n \geq 1$). Gevraagd wordt de "body" van een procedure te schrijven, waarvan de "heading" luidt

```
procedure multiple(n,ℓ,u,p); value n; integer n;
integer array ℓ, u; procedure p;
```

en waarvan de werking (enigszins slordig) wordt weergegeven door

```
for r[1]:= ℓ[1] step 1 until u[1] do
for r[2]:= ℓ[2] step 1 until u[2] do
 '              '            '
 '              '            '
 '              '            '
for r[n]:= ℓ[n] step 1 until u[n] do p(n,r).
```

Men kan denken aan een draaiende kilometerteller, waarvan het aantal posities variabel is en waarvan de i-de positie van links niet van 0 t/m 9 maar van ℓ_i t/m u_i telt. De eerste oplossing luidt

```
begin integer array r[1:n];
      integer i;
      boolean work;
      i:= 1; work:= true;
      while work ∧ i≤n do
      begin r[i]:= ℓ[i];
            if ℓ[i] > u[i] then work:= false;
            i:= i+1
      end;
      while work do
      begin p(n,r); i:= n;
            while (if i>0 then r[i] = u[i] else false) do
            begin r[i]:= ℓ[i];
                  i:= i-1
            end;
            if i>0 then r[i]:= r[i]+1 else work:= false
      end
end .
```

In het eerste deel van dit block wordt a.h.w. de kilometerteller in de beginstand gezet; en passant wordt nagegaan of we niet bezig zijn met het ontaarde geval dat er een positie i is, waarvoor de verzameling toegestane "cijfers" leeg is. Zo niet, dan heeft in het tweede deel de logische variabele work de waarde true zolang de eindstand niet is overschreden. Het verhogen van de tellerstand zal het meest frekwent gebeuren d.m.v. de laatste conditionele statement

```
if i>0 then r[i]:= r[i]+1 else work:= false
```

waarin i = n.
De while-statement die hier vlak voor staat zorgt ervoor dat wanneer de meest rechtse positie en eventueel zijn buren de hoogste stand hebben bereikt, deze posities weer in de beginstand worden gezet. Na deze while-statement wijst i naar de eerste positie van rechts geteld, die niet in zijn hoogste stand werd aangetroffen, ŏf i = 0, in welk geval alle posities in de hoogste stand staan, zodat we klaar zijn.
Een programmeur die na enig gepuzzel deze oplossing gevonden heeft, die nog blijkt te werken ook, zal er waarschijnlijk tevreden mee zijn. Toch doen we er goed aan ons af te vragen of al deze rompslomp voor zo'n

betrekkelijk eenvoudig probleem nu wel nodig is. Het is immers veel natuurlijker het doorlopen van de i-de positie te beschrijven met het "step-until-mechanisme":

```
for r[i]:= ℓ[i] step 1 until u[i] do .
```

In de oorspronkelijke probleemstelling zien we dat deze regel gevolgd moet worden door zo'n zelfde regel, waarin i met 1 verhoogd is, als i < n en door p(n,r) als i = n. Dit laat zich uitstekend beschrijven met behulp van een recursieve procedure. We vinden zo de oplossing

```
begin integer array r[1:n];
      procedure recursive (i); value i; integer i;
      for r[i]:= ℓ[i] step 1 until u[i] do
      if i<n then recursive (i+1) else p(n,r);
      recursive (1)
end .
```

Deze oplossing is veel sneller te doorzien dan de vorige. Krijgt iemand, die in dit probleem geïnteresseerd is, deze oplossing onder ogen, dan zal hij zeker de moeite nemen de tekst aandachtig te bekijken en als hij niet volkomen vreemd staat tegenover recursieve procedures, dan zal hij dit stukje tekst na inspectie inderdaad als oplossing accepteren. De eerste oplossing daarentegen zal voor een gebruiker die even achterdochtig en even gemakzuchtig is als onze vorige proefpersoon, veel eerder aanleiding geven om naar de computer te stappen en de procedure te gaan testen. Dit betekent in feite dat hij empirisch gaat onderzoeken wat deductief onderzocht kan worden. Nu kan men tegenwerpen dat het gaan testen van de eerste oplossing te wijten is aan de gemakzucht van de gebruiker, want ook deze oplossing kan langs deductieve weg worden geverifieerd. Dit neemt echter niet weg, dat het grote voordelen heeft zo'n deductieve weg niet onnodig lang en kronkelig te maken. Uit praktisch oogpunt verdient het geen aanbeveling kritiekloos een programma te accepteren als object voor een correctheidsbewijs, maar kan veel beter eerst worden nagegaan of een doorzichtiger algoritme voor hetzelfde probleem te vinden is. Beschouwingen over de correctheid van een programma zijn vooral belangrijk bij het ontwerpen van dat programma. Het volgende voorbeeld is in dit verband illustratief.

Door ELSPAS e.a. [2] wordt een algoritme van WENSLEY [3] aangehaald als object voor een correctheidsbewijs. Gevraagd wordt bij twee gegeven

gehele getallen P en Q ($0 \leq P < Q$) het quotient P/Q te benaderen met gegeven nauwkeurigheid ε, waarbij slechts de bewerkingen optellen, aftrekken en halveren gebruikt mogen worden. Het algoritme van WENSLEY wordt door ELSPAS weergegeven d.m.v. een blokschema dat ongeveer overeenkomt met de volgende ALGOL-achtige tekst:

```
A:= 0; B:= Q/2; D:= 1; y:= 0;
while D≥eps do
begin ① if P ≥ A+B then
          begin y:= y+D/2;
                A:= A+B
          end;
          B:= B/2;
          D:= D/2
      ②
end.
```

Hierna is y de gevraagde benadering van P/Q. Elspas geeft een uitvoerig bewijs van de correctheid van deze algoritme. Het berust op de relaties

$$A = Qy$$

$$B = \frac{QD}{2}$$

$$\frac{P}{Q} - D < y \leq \frac{P}{Q},$$

die gelden in de punten ① en ②. De verificatie van deze relaties (ook wel *asserties* genoemd) is niet helemaal triviaal. De vraag dringt zich op hoe men aan deze relaties komt. Van strikt wiskundig standpunt bekeken is deze vraag niet relevant, maar praktisch en didactisch gesproken is dit juist het kritieke punt: we zijn immers niet zozeer in dit ene voorbeeld geïnteresseerd, maar in algemeen toepasbare methoden.
In plaats van dieper in te gaan op deze algoritme proberen we eerst eens zelf een oplossing van dit probleem te bedenken. Uit

$$0 \leq P < Q$$

volgt

$$0 \leq \frac{P}{Q} < 1.$$

Het gezochte getal ligt dus op het interval [0,1). De gegeven mogelijkheid om te halveren leidt tot de gedachte dit eenheidsinterval herhaaldelijk te halveren, waarbij we voortdurend een deelinterval [a,b) van [0,1) blijven beschouwen waarop P/Q gelegen is, d.w.z. zoiets als

```
a:= 0; b:= 1;
while b-a ≥ eps do
begin c:= (a+b)/2;
      if c ≤ P/Q then a:= c else b:= c
end .
```

Dit is natuurlijk niet goed, omdat hierin P/Q voorkomt. We komen nu op het idee

$$c \leq P/Q$$

anders te willen schrijven, n.l. als

$$c * Q \leq P.$$

Nu is vermenigvuldigen ook niet toegestaan. We zouden daarom willen beschikken over een variabele C, die gelijk is aan cQ. Omdat c ontstaat als lineaire combinatie van a en b, voeren we bovendien de variabelen A en B in, met de betekenis A=aQ en B=bQ. We krijgen zodoende de algoritme

```
a:= 0; A:= 0;
b:= 1; B:= Q;
while b-a ≥ eps do
begin c:= (a+b)/2; C:= (A+B)/2;
      if C ≤ P then begin a:= c; A:= C end
               else begin b:= c; B:= C end
end .
```

De relaties

$$A = aQ$$
$$B = bQ$$
$$C = cQ$$
$$a \leq \frac{P}{Q} < b,$$

die gebruikt zouden kunnen worden bij een correctheidsbewijs, hebben nu geleid tot de constructie van de algoritme.

Als men deze algoritme vergelijkt met de oorspronkelijke dan valt het op dat deze algoritme wel transparanter is dan de vorige, maar niet noemenswaard korter of eenvoudiger van struktuur. De doorzichtigheid is te danken aan de keuze van de namen van de hulpvariabelen A, B en C. Gebruik makend van de terminologie van [5] zou men de paren (a,A), (b,B), c,C) de concrete representatie van de abstracte variabelen a, b en c kunnen noemen. Het vermoeden is gewettigd dat we ook de algoritme van blz. 144 doorzichtiger kunnen weergeven als we de methode eenmaal begrijpen. Nu stamt deze algoritme uit de tijd dat iedere computergebruiker gewoon was binair te denken of althans zeer vertrouwd was met de binaire vaste-komma-representatie

$$0,b_1b_2b_3\ldots b_n = b_1\cdot\frac{1}{2} + b_2\cdot\frac{1}{4} + \ldots + b_n\cdot\frac{1}{2^n}$$

van de echte breuk P/Q. Met deze representatie voor ogen ligt het idee voor de hand, zo'n bitrij $b_1b_2\ldots b_n$ eerst op nul te stellen en daarna, van links naar rechts werkend, elk bit zo nodig en zo mogelijk in de 1-stand te zetten. Dit betekent dat een variabele x de startwaarde 0 krijgt en dat we daarna achtereenvolgens voor i = 1,2,3,... nagaan of $d_i = 2^{-i}$ bij x opgeteld dient te worden. Dit is het geval als $x+d_i \leq P/Q$; na deze verhoging $x := x+d_i$ van x geldt uiteraard nog steeds $x \leq P/Q$. De essentie van de algoritme wordt zodoende

```
x:= 0; d:= 1;
while d ≥ eps do
begin d:= d/2;
      if x+d ≤ P/Q then x:= x+d
end .
```

Omdat, van links naar rechts gaand, elke binaire positie de kans krijgt op zijn maximale waarde 1 gezet te worden, bestaat er geen gevaar dat x niet snel genoeg zal groeien; nadat i bits zijn afgewerkt is de fout kleiner dan 2^{-i}, m.a.w. $\frac{P}{Q} - x < d$.
In de zojuist getoonde voorlopige versie van de algoritme komt nog P/Q voor. Door een geschikte concrete representatie van de abstracte variabelen x en d kan deze moeilijkheid worden opgelost. We herschrijven

```
x+d ≤ P/Q als
X+D ≤ P,  waarbij
X = Qx    en
D = Qd.
```

We dienen er nu nog slechts voor te zorgen dat X en D aan het begin van de algoritme gelijk zijn aan het Q-voud van x resp. d en dit voortdurend blijven. De algoritme wordt dan

```
x:= 0; X:= 0;
d:= 1; D:= Q;
while d ≥ eps do
begin d:= d/2; D:= D/2;
      if X+D ≤ P then
      begin x:= x+d;
            X:= X+D
      end
end .
```

De invariante asserties zijn de reeds gebruikte relaties

$X = Qx$

$D = Qd$

$\frac{P}{Q} - d < x \leq \frac{P}{Q}$, oftewel $P-D < X \leq P$.

We hebben nu, op triviale kleinigheden na, de oorspronkelijke algoritme en de relaties van blz. 140 terug. Door bewust te streven naar een geschikte representatie van de variabelen en door "correctheidsoverwegingen" bij het programmeren te betrekken is een aanzienlijk doorzichtiger versie van deze algoritme verkregen.

3. EXTERNE INVLOEDEN OP DE PROGRAMMAKWALITEIT

De algemeen voorkomende situatie in de praktijk van het programmeren is verre van ideaal. Doordat men een programma te ingewikkeld en te weinig transparant maakt, ontstaan fouten die later uiterst moeilijk te verbeteren zijn. De voornaamste factor die bepalend is voor de kwaliteit van een programma is het denkproces van de schrijver van dat programma. De specifieke denkgewoonten bij het programmeren worden van buitenaf door verschillende factoren beïnvloed. Het is misschien de moeite waard een poging te doen enkele van deze factoren aan te wijzen en als mogelijke oorzaak van bovengenoemde ongelukkige situatie ter discussie te stellen.

In de eerste plaats wordt er in brede kring in een te lage taal geprogrammeerd. Het is langzamerhand algemeen bekend dat het veelvuldig gebruik van goto-statements de doorzichtigheid van een programma bepaald niet bevordert. Daarom is het betreurenswaardig dat er nog steeds op grote schaal geprogrammeerd wordt in talen die noch de compound statement noch de if-then-else-constructie kennen en daardoor de programmeur dwingen voortdurend goto-statements te gebruiken. Op het gebied van programmeertalen bestaan vele misverstanden. Een voorbeeld hiervan is de nog vrij veel voorkomende opvatting, dat de keuze van een programmeertaal van ondergeschikt belang zou zijn. Men stelt zich een scherp onderscheid voor tussen een "denkfase", waarin geen programmeertaal gebruikt wordt en een "doefase", waarin het denkwerk achter de rug is en alleen nog een flinke dosis vlijt en handvaardigheid nodig is om de vereiste regels programmatekst te produceren. Men kan tot deze zienswijze komen, doordat men de sterke samenhang niet onderkent die er is tussen exact denken en taalgebruik. Voor het bedenken en formuleren van een algoritme heeft men een taal nodig. Een natuurlijke taal geeft aanleiding tot ambiguïteiten, of is te vaag of te wijdlopig. Daarom geeft men de voorkeur aan meer formele uitdrukkingsmiddelen, zoals wiskundige notatie en blokschema's. Omdat een algoritme ermee wordt geformuleerd, kan men deze samenvatten onder het begrip *algoritmische taal*. Hoe meer nu de programmeertaal, die we hierna moeten gebruiken om onze algoritme door een machine te laten uitvoeren, verwant is aan de algoritmische taal, des te minder werk blijft er over voor de bovengenoemde "doefase". Ideaal is de situatie dat beide talen samenvallen. Door zo'n taal te leren, verkrijgt men niet alleen een middel om een machine aan het werk te zetten, maar leert men bovendien zich uit te drukken als een algoritme moet worden geformuleerd. Of deze ideale situatie wordt bereikt hangt af van de programmeertaal en van de toepassing waarvoor men deze taal wil gebruiken; als voor een toepassing het verschil tussen algoritmische taal en programmeertaal blijft bestaan, dan dient dit verschil zo klein mogelijk te zijn.

In de tweede plaats zijn de *werkomstandigheden* bij het programmeren vaak ongeschikt. De laatste zin van EUWE's artikel [1] luidt

> "Programmeren is een nauwkeurig werk dat in alle rust
> dient te geschieden".

Aan dit laatste ontbreekt het maar al te vaak. Dikwijls hinderen programmeurs die op één kamer gehuisvest zijn, elkaar in ernstige mate bij hun werk, zonder dat men zich dat overigens realiseert. Veel programmafouten ontstaan doordat men niet in de gelegenheid is zich te concentreren. Ook een te sterke nadruk op de tijdsfactor kan ongunstig werken. Als men iemand een niet-triviaal programmeerprobleem opgeeft, is de voornaamste vraag niet hoe lang hij er over doet, maar in hoeverre de man in staat geacht moet worden het probleem op te lossen. Er is ook in dit opzicht een overeenkomst tussen programmeren en wiskunde.

Programmeercursussen dienen machine-onafhankelijk te zijn. De aandacht moet gericht zijn op het construeren van algoritmen en niet op allerlei bijzonderheden van een machine of van een taalimplementatie. Een voorbeeld van een algoritme-georiënteerde cursus is GEURTS [4]. Tijdens en na de cursus moet de graad van moeilijkheid van de op te lossen programmeerproblemen geleidelijk toenemen.

Er bestaat verschil van opvatting over de rol die de *computer* bij het programmeren zou moeten spelen. Traditioneel wordt de computer gebruikt om de fouten in een programma op te sporen. Sommigen willen de computer de fouten zelfs laten verbeteren. Dit laatste is alleen te verdedigen als middel om een computer in staat te stellen zo goed en zo kwaad als het gaat verder te gaan met het zoeken naar fouten; de aangebrachte "verbetering" zal immers zelden overeenstemmen met de bedoeling van de programmeur. Over het testen is vele malen terecht opgemerkt, dat hiermee de correctheid van een programma niet kan worden aangetoond. In de praktijk is het gevaar niet denkbeeldig, dat we zelf te weinig kritische aandacht aan onze programma's schenken, omdat we eraan gewend raken dat de computer dit voor ons doet. Hoe machtig de computer heden ten dage ook is, er is één ding dat hij niet kan en ook nooit zal kunnen, namelijk bedenken wat wij van hem willen. Wij zullen hem dit zelf moeten meedelen en het is onze verantwoordelijkheid, dat dit correct gebeurt.

LITERATUUR

[1] EUWE, M., *Computer en wiskunde-onderwijs*, Euclides, 34 (1958), 97-102.

[2] ELSPAS, B. a.o., *An assessment of techniques for proving program correctness*, Computing Surveys, 4 (1972) 97-147.

[3] WENSLEY, J.H., *A class of non-analytical iterative processes*, Computer J., 1 (1958) 163-167.

[4] GEURTS, L., *Cursus programmeren*, MC Syllabus 16.1, Mathematisch Centrum, Amsterdam, 1973.

[5] MEERTENS, L.G.L.T., *Van abstracte variabele naar concrete representatie*, Syllabus Colloquium Programmacorrectheid, hoofdstuk 7.

MATHEMATICAL CENTRE SYLLABUS 21, 1975, X, 151-165.

RECURSIEVE PROCEDURES EN EENVOUDIGE INDUCTIE ASSERTIES

M.M. FOKKINGA
Technische Hogeschool, Delft

1. INLEIDING

Al meermalen in dit colloquium is het woord "assertie" gevallen. Een *assertie* is een uitspraak op een plaats in een programmatekst, die in die kontekst geïnterpreteerd moet worden. Inderdaad, goed gekozen asserties vormen een machtig hulpmiddel voor de dokumentatie van een programma en voor de *overtuiging* voor de correctheid ervan. Maar meer nog, door middel van asserties kan men op grond van de programmatekst de -partiële- correctheid van een programma formeel *bewijzen*.
Het tweetal asserties dat een programmatekst omsluit noemen we een correctheidsbewering voor dat programma. Onder partiële correctheid wordt dan verstaan dat de assertie aan het einde van de tekst geldig is ná terminatie van het programma, mits vóóraf de assertie aan het begin van de tekst geldig was. Totale correctheid garandeert bovendien dat het programma inderdaad termineert.
In een formeel bewijs gaan we er van uit dat we weten welke correctheidsbeweringen waar zijn -en dus bewezen geacht mogen worden- voor de elementaire opdrachten en welke correctheidsbeweringen we mogen doen voor een samengestelde tekst op grond van alreeds bewezen correctheidsbeweringen voor de tekstgedeeltes. Deze uitgangspunten zijn geformuleerd als respectievelijk de axioma's en de samenstellings- of afleidingsregels voor het systeem waarin we werken. Als er géén andere correctheidsbeweringen waar zijn dan degene die in dit systeem bewezen kunnen worden, dan noemen we het systeem *volledig* en kunnen we zeggen dat de afleidingsregels en axioma's de semantiek van de programmeertaal volledig vastleggen.
Zo is voor de taal PASCAL door HOARE [7] een systeem van axioma's en regels gegeven dat *per definitie* van bijna alle taalconstructies (n.l. uitgezonderd de goto en de arithmetiek) de semantiek vastlegt.

Enige voorbeelden mogen dit verduidelijken. Laten we in het vervolg met $p, p_1, p_2, \ldots, q, q_1, q_2, \ldots$ eigenschappen (predicaten) aanduiden.

Voorbeeld 1

Een zinvol axioma voor de meervoudige toekenningsopdracht is

$$\text{ASS: } \{p(\bar{E}(\bar{x}))\}\ \bar{x} := \bar{E}(\bar{x})\ \{p(\bar{x})\}$$

en als samenstellingsregels zijn heel zinvol

$$\text{WH: } \frac{\{B \wedge p\}\ S\ \{p\}}{\{p\}\ \underline{\text{while}}\ B\ \underline{\text{do}}\ S\ \{p \wedge \neg B\}}$$

$$\text{SEQ: } \frac{\{p_1\}\ S_1\ \{q\} \quad \{q\}\ S_2\ \{p_2\}}{\{p_1\}\ S_1;S_2\ \{p_2\}}$$

$$\text{CONS: } \frac{\{p\} \text{ impliceert } \{p_1\} \quad \{p_1\}\ S\ \{q_1\} \quad \{q_1\} \text{ impliceert } \{q\}}{\{p\}\ S\ \{q\}}$$

(Deze axioma's en regels zijn intuïtief gerechtvaardigd -ingeval er in $\bar{E}(\bar{x})$ geen neveneffecten optreden- : we zijn er zeker van dat we hiermee geen beweringen kunnen afleiden die tegenstrijdig zijn met een "werkelijke" uitvoering van een programma.)
Voor de berekening van de grootste gemene deler*) van A en B volgens de algoritme van Euclides kunnen we gebruik maken van het programma:

```
{A>B≥0}
{A>B≥0 ∧ [A,B]=[A,B]}
(m,n):= (A,B);
{m>n≥0 ∧ [m,n]=[A,B]}
while n≠0
do {m>n≥0 ∧ [m,n]=[A,B] ∧ n≠0}
   {n>m-n · m+n≥0 ∧ [n, m-n · m÷n]=[A,B]}
```

*) [A,B] staat voor: de grootste gemene deler van A en B.

```
    (m,n):= (n, m-n · (m÷n));
    {m>n≥0 ∧ [m,n]=[A,B]}
od;
{m>n=0 ∧ [m,n]=[A,B]}
{m=[A,B]}
GGD:= m
{GGD=[A,B]}
```

waarvan nu tevens de -partiële- correctheid bewezen is m.b.v. de methode van de asserties, omdat voor ieder -welgevormd- tekstgedeelte de omsluitende asserties een bewezen bewering vormen volgens de gegeven axioma's en regels.*)

"Formeler" opgeschreven luidt het bewijs als volgt:

(i) { Li }	Si	{ Ri }	Vi
(1) {A>B≥0 }	impliceert	{A>B≥0 ∧ [A,B]=[A,B]}	lemma
(2) {R1 }	(m,n):= (A,B)	{m>n≥0 ∧ [m,n]=[A,B]}	ASS
(3) {n≠0 ∧ R2}	impliceert {n>m-n·m÷n≥0	∧ [n,m-n·m÷n]=[A,B]}	lemma
(4) {R3 }	(m,n):= (n,m-n·(m÷n))	{m>n≥0 ∧ [m,n]=[A,B]}	ASS
(5) {L3 }	S4	{R4 }	3,4 : C+S
(6) {R2 }	while n≠0 do S4 od	{R4 ∧ n = 0 }	5 : WH
(7) {R6 }	impliceert	{m = [A,B] }	lemma
(8) {R7 }	GGD:= m	{GGD = [A,B] }	ASS
(9) {A>B≥0 }	HET VOLL. PROGR: S2;S6;S8	{GGD = [A,B] }	1,2,6,7,8:C+S

Voorbeeld 2

Vermaard is al het axioma voor de toekenningsopdracht,
ASS: $\{p(E(\bar{x}))\}$ x:= E(x) $\{p(x)\}$. Maar intuïtief gerechtvaardigd is ook het volgende axioma ASS': $\{p(x)\}$ x:= E(x) $\{\exists x_0 : p(x_0) \wedge x = E(x_0)\}$. Mits E geen neveneffecten heeft zal ASS' geen tegenstrijdigheden opleveren met de "werkelijkheid", en zelfs is ze naar ons gevoel ook volledig, d.w.z. iedere ware bewering over de toekenningsopdracht x:= E(x) kan in de vorm van ASS' geschreven worden. Dus voortaan kunnen we met recht ASS' als definitie van de semantiek nemen.

---

*) Het is ook mogelijk de correctheidsbewering {A,B>0} ... {GGD=[A,B]} te bewijzen.

Maar ASS is ook volledig (en gelijkwaardig met ASS'; ga na welke p je moet kiezen in ASS' om ASS uit ASS' af te leiden, en omgekeerd) en wordt meestal gebruikt in praktische toepassingen.

Voorbeeld 3

Voor recursieve procedures heeft Hoare de volgende regel opgesteld:

(ALS:) {p} Body-van-P {q} is af te leiden volgens het systeem van axioma's en regels, met de *hypothese* {p} call P {q} voor ieder voorkomen van call P in de Body-van-P

(DAN:) {p} call P {q}

Deze bewijsregel is gerechtvaardigd door bijvoorbeeld uit de premisse een bewijs van {p} Body-van-P {q} af te leiden met inductie naar de recursiediepte (rd). Want bij rd=1 leidt een evaluatie van de body niet tot een aanroep op P en hebben we derhalve de hypothese niet nodig en staat in de premisse al een bewijs voor dit geval. En bij grotere rd leiden de binnenaanroepen call P tot een recursiediepte < rd, dus mogen we daarvoor de inductiehypothese aannemen. Als daarmee {p} Body-van-P {q} bewezen kan worden, zoals in de premisse staat, dan kunnen we volgens het principe van volledige inductie stellen

{p} Body-van-P {q}

voor evaluaties die tot willekeurige recursiediepte leiden, dus {p} call P {q}.
Maar deze afleidingsregel kan bijv. ook afgeleid worden uit een andere bewijsregel, Scott's inductieregel, die sterker is dan de boven gegeven regel van HOARE. Scott's inductieregel is reeds in hoofdstuk 8 van dit colloquium uitvoerig behandeld DE BAKKER [1]; een andere naam ervoor is *computational induction*. Zie bijv. ook MANNA, NESS & VUILLEMAN [8], DE BAKKER & DE ROEVER [4], DE BAKKER & MEERTENS [3].

2. DOELSTELLING EN ENIG FORMALISME

We zullen ons nu bezig houden met theoretische aspecten van de methode van inductieve asserties. Wat we hopen te bereiken is inzicht in de kracht en de werking van de "eenvoudige" assertiemethode, met name voor recursieve

procedures. Dat is: we leggen ons de beperking op om alléén correctheidsbeweringen voor *elementaire* opdrachten in de premisse van een regel toe te laten. Dus Hoare's regel voor recursieve procedures en Scott's inductieregel worden niet gebruikt. We streven niet na een zo sterk mogelijk bewijs te geven -daarvoor verwijs ik naar DE BAKKER & MEERTENS [3]-, maar juist enig inzicht te verschaffen.

We zullen zien dat er voor iedere recursieve procedure (stel van recursieve procedures) een regel te formuleren is, die volledig is, maar waarvan het aantal beweringen in de premisse dan en slechts dan eindig is als de recursieve procedure "regulier" is. Bovendien is zo'n regel een karakterisering voor de recursieve procedure P in de volgende zin:

> als voor enig ander programma T de correctheidsbewering van de regel ook voor T geconcludeerd kan worden uit de premisse, voor alle specificaties van de asserties
> dan en slechts dan
> is P een uitbreiding van T wat het invoer-uitvoer-gedrag betreft.

Allereerst hebben we enig formalisme nodig om onze beweringen ondubbelzinnig te kunnen formuleren. Maar voor de leesbaarheid zal ik het zoveel mogelijk beperken; de lezer vergeve mij de mogelijke onvolkomenheden.
De *programma's* die we beschouwen zijn stelsels van recursieve procedures gegeven door declaraties van de volgende vormen:

vb. $\{P \Leftarrow A_1;A_2;P;A_3 \cup A_4$

of

vb. $\begin{cases} P_1 \Leftarrow A_1;P_2;A_1;P_2 \cup A_1;P_2 \\ P_2 \Leftarrow ((A_4;P_2)\cup A_3);\ A_2 \end{cases}$

waarbij we veronderstellen dat het in-uitvoergedrag van de hoofdletters $A_{1,2\ldots}$ bekend is. Zoals gewoonlijk wordt het in-uitvoergedrag van een samenstelling van opdrachten uit dat van de onderdelen verkregen d.m.v. een aaneenschakeling van de gedragingen in geval van een ;-samenstelling en d.m.v. een niet-deterministische keuze tussen de gedragingen van de onderdelen in geval van een ∪-samenstelling. Voor proceduresymbolen moeten we de copy-rule toepassen, d.w.z. het in-uitvoergedrag van een procedure wordt volkomen gegeven door dat van zijn body.

Opmerking 1. Het niet-deterministisch gedrag bepaald door ∪ is gemakkelijk

deterministisch te maken door vóór ieder daarmee samengesteld onderdeel S_i een "opdracht" B_i te plaatsen met zó een gedrag dat voor iedere invoer hoogstens één B_j die invoer doorlaat en alle andere B_k géén uitvoer geven.

Voorbeeld

Laten de gedragingen van B_1 en B_2 elkaar "uitsluiten", dan is het gedrag van $S_1 \cup S_2$ mogelijk niet-deterministisch, maar dan komt het gedrag van $B_1;S_1 \cup B_2;S_2$ overeen met het deterministisch gedrag van

$$\underline{\text{if}}\ B_1\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2$$ (of beter: $\underline{\text{if}}\ B_1\ \underline{\text{then}}\ S_1\ \underline{\text{else if}}\ B_2\ \underline{\text{then}}\ S_2\ \underline{\text{d}}\uparrow$)

Opmerking 2. We zullen ook voor de elementaire opdrachten niet-deterministische gedragingen toelaten, d.w.z. voor een invoer kunnen er verscheidene uitvoeren gespecificeerd zijn. Hiermee zijn de programma's in het algemeen dus in-uitvoerrelaties geworden i.p.v. in-uitvoerfuncties. We zullen dan ook met $x(S)y$ aanduiden dat x en y in de in-uitvoerrelatie zitten bepaald door S.

We kunnen voor opdrachten S_1 en S_2 beweren dat de relatie bepaald door S_1 geheel omvat wordt door die van S_2, i.e. $\forall x,y: x(S_1)y \rightarrow x(S_2)y$. We formuleren dit als "$S_1 \subseteq S_2$" en het inclusieteken heeft dan zijn gewone betekenis voor de relaties, i.e. verzamelingen, (S_1) en (S_2).
Nu het formalisme voor correctheidsbeweringen. Als in Hoare's notatie $\{p\}\ S\ \{q\}$ gesteld wordt, dan betekent dit

$$\forall x,y: p(x) \wedge x(S)y \rightarrow q(y),$$

ofwel, als we de predicaten ook noteren in relatievorm,

$$\forall x,y: x(p)x \wedge x(S)y \rightarrow x(S)y \wedge y(q)y,$$

zodat volgens onze afspraken voor de ;-samenstelling

$$\forall x,y: x(p;S)y \rightarrow x(S;q)y\ ,\quad \text{i.e.}$$
$$p;S \subseteq S;q$$

waarbij uit de kontekst duidelijk moet zijn wat de interpretatie voor de asserties p en q is.
Wanneer A en C inclusies zijn van de vorm $S_1 \subseteq S_2$ of $p;S_1 \subseteq S_1;q$ e.d. en de interpretatie voor de asserties (kleine letters p, q,...) wordt gesymboliseerd met de aanduiding I, dan zetten we $A \models_I C$ voor "A impliceert C" en we laten de aanduiding I weg als "voor alle interpretaties voor de

asserties, A de bewering C impliceert".

Voorbeeld

Hoare's regel WH luidt nu:

$$p;B;A \subseteq A;p \models p;(\underline{\text{while}}\ B\ \underline{\text{do}}\ A) \subseteq (\underline{\text{while}}\ B\ \underline{\text{do}}\ A);p;\overline{B}$$

Definiëren we de while opdracht als de recursieve procedure W

$$W \Leftarrow B;A;W \cup \overline{B}$$

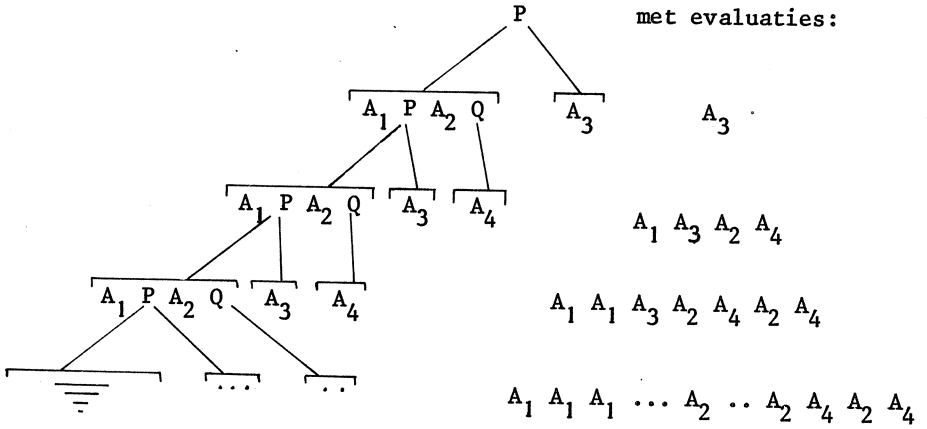
dan kunnen we stellen $p;B;A \subseteq A;p \models p;W \subseteq W;p;\overline{B}$.

3. FORMULERING VAN DE REGEL

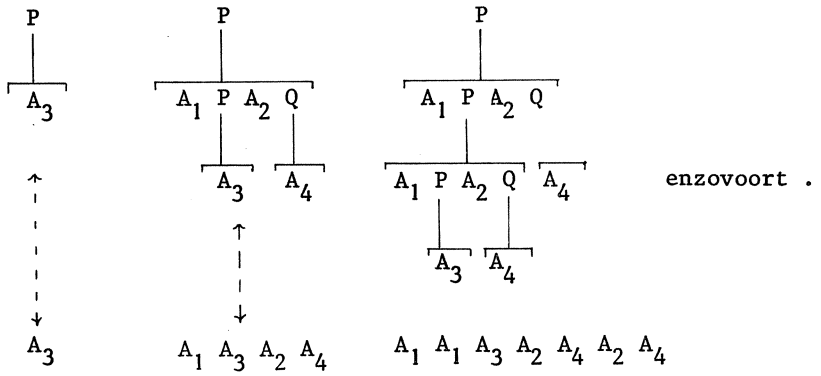
Beschouw eens het programma

$$P \Leftarrow A_1;P;A_2;Q \cup A_3$$
$$Q \Leftarrow A_4$$

Dan kunnen we een "boom" opzetten die alle mogelijke evaluaties (van P) bevat, n.l.



Dit soort bomen komt veelvuldig voor in formele talentheorie; meestal wordt slechts dat gedeelte gegeven dat precies één "woord" laat afleiden, bijv.



Ieder van de laatstgeschetste zullen we een *afleidingsboom* noemen en de eerstgeschetste de *volledige afleidingsboom*. Uit de definitie van de copy-rule als interpretatie voor procedureaanroepen en uit uw gevoel voor wat de "woorden" van een volledige afleidingsboom zijn, volgt dat voor iedere in- en uitvoer x(P)y er een woord w is in de volledige afleidingsboom voor P, zodat x(w)y, en omgekeerd. D.w.z. de relatie (P) is omvat door de vereniging van de relaties bepaald door de woorden uit de boom, en omgekeerd. Noteren we de verzameling van woorden uit een boom B als $L(B)$ en de daardoor bepaalde in-uitvoerrelatie als $x(L(B))y$ voor desbetreffende x en y dan hebben we

Stelling 1. $P \subseteq L(B_P)$, $L(B_P) \subseteq P$, ofwel $P = L(B_P)$.

Terminologie. Beschouw in de voorgaande figuren nog eens de (volledige) afleidingsboom en het woord $A_1\ A_3\ A_2\ A_4$. We zeggen dat in dat woord A_3 *direkt volgt* op A_1, A_2 direkt volgt op A_3, enz. Definieer nu voor de overeenkomstige voorkomens van die symbolen in de boom dezelfde "direkt na" relatie. We zeggen ook dat A_1 een *linkersymbool* is in dat woord, en A_4 een *rechtersymbool*. Definieer nu ook voor de overeenkomstige voorkomens van die symbolen in de boom zo een "is linker voorkomen" en "is rechter voorkomen" eigenschap. Dan kunnen we definiëren dat de woorden van een boom precies de rijen van symbolen zijn die in de boom achtereenvolgens direkt na elkaar voorkomen, beginnend met een linkervoorkomen en eindigend met een rechter-

voorkomen. We zullen dit in het vervolg als definitie aannemen.

We geven nu de definitie van een stelsel correctheidsbeweringen (voor elementaire opdrachten) met betrekking tot correctheidsassertie-symbolen p_{in}, p_{ex} en programma P. Als we dit stelsel, noem het $\acute{A}$, nemen als premisse van de regel met conclusie $p_{in};P \subseteq P;p_{ex}$, dan is deze regel de gezochte volledige, karakteriserende regel. Het bewijs daarvan volgt in het volgende hoofdstuk. Nu eerst de

Definitie

Laat $\{p_i\}_{i\in\mathbb{N}}$ een verzameling nieuwe, i.e. nog nergens voorkomende predicaatletters zijn. Associeer met ieder voorkomen van een opdrachtsymbool in de boom B-(voor-P) precies één zo'n predicaatletter, dan bestaat $\acute{A}$ uit

1) voor direkt opvolgende symbolen A en A' met geassocieerde p en p' de bewering $p;A \subseteq A;p'$
2) voor ieder linker/rechter voorkomen van symbolen A/A' met geassocieerde p/p' de bewering $p_{in} \subseteq p/\!\!/p';A' \subseteq A';p_{ex}$.

4. STELLINGEN EN BEWIJZEN

De wezenlijke stelling met bewijs ligt besloten in het

Hoofdlemma

(i) $\acute{A} \models p_{in};L(B) \subseteq L(B);p_{ex}$.

(ii) $p_{in};L(B) \subseteq L(B);p_{ex} \models_{*} \acute{A}$
(*: mits de asserties $\{p_i\}_{i\in\mathbb{N}}$ in $\acute{A}$ geschikt geïnterpreteerd worden)

(iii) voor willekeurig programma T
$\acute{A} \models p_{in};T \subseteq T;p_{ex}$ impliceert $T \subseteq L(B)$.

Bewijs

(i) Voor willekeurige interpretatie I voor de asserties redeneren we als volgt. Zij $A_1;A_2;\ldots;A_n$ een woord uit $L(B)$, dan hebben we per definitie $p_{in} \subseteq p_1$, $p_1;A_1 \subseteq A_1;p_2,\ldots,p_n;A_n \subseteq A_n;p_{ex}$ in $\acute{A}$ en hiermee kunnen we in n stappen concluderen dat hun geldigheid die van $p_{in};A_1;\ldots;A_n \subseteq A_1;\ldots;A_n;p_{ex}$ impliceert, i.e. $\acute{A} \models_I p_{in};A_1;\ldots;A_n \subseteq A_1;\ldots;A_n;p_{ex}$. Dit geldt voor alle woorden uit $L(B)$, dus (met enkele stappen) $\acute{A} \models_I p_{in};L(B) \subseteq L(B);p_{ex}$. □

(ii) Merk op dat de restrictie in de keuze voor de interpretatie van de p_i in $\acute{A}$ ook zó gelezen kan worden: als $p_{in};L(B) \subseteq L(B);p_{ex}$ geldt dan kan er

een interpretatie voor de asserties p_i in A gevonden worden zó dat A ook geldig is.
Deze geschikte interpretatie is bijvoorbeeld de volgende (in DE BAKKER & MEERTENS [3] staat een karakterisering wélke het allemáál kunnen zijn):

$x(p)x \leftrightarrow$ "x kan als resultaat verkregen worden uitgaande van een invoer die aan p_{in} voldoet, door een berekening door $A_1,\ldots,A_{i-1}$ waarbij $A_1;\ldots;A_i$ een beginstuk van een woord uit de boom B is en p met A_i geassocieerd is", i.e. $\exists x_0$: $x_0(p_{in};A_1;\ldots;A_{i-1})x$ en p is met A_i geassocieerd.

Inderdaad, nu is de geldigheid van de $p_{in} \subseteq p$ en $p;A \subseteq A;p'$ in A gemakkelijk aan te tonen, en voor de $p;A \subseteq A;p_{ex}$ redeneren we als volgt.
Voor alle x en y: als $x(p;A)y$, i.e. $x(p)x \wedge x(A)y$, dan per definitie $x_0(p_{in};A_1;\ldots;A_{n-1})x \wedge x(A)y$ waarbij $A_n \equiv A$ en $A_1;\ldots;A_n$ een woord uit B is, dus $x_0(p_{in};L(B))y$ en derhalve volgens de premisse $x_0(L(B);p_{ex})y$, dus i.h.b. $y(p_{ex})y$ en aangezien ook nog geldt $x(A)y$ volgt $x(A;p_{ex})y$. □

(iii) We tonen $T \subseteq L(B)$ aan. Dus zij $x_0(T)y_0$ dan moeten we $x_0(L(B))y_0$ aantonen.
Beschouw daartoe het antecedent met de volgende interpretatie voor de asserties p_i in A en p_{in},p_{ex} (een interpretatie die van x_0 afhangt):

$x(p)x \leftrightarrow x_0(A_1;\ldots;A_{i-1})x$ waarbij p met A_i geassocieerd is en $A_1\ldots A_i$ een beginstuk van een woord uit de boom B is,
$x(p_{in})x \leftrightarrow x=x_0$,
$x(p_{ex})x \leftrightarrow x_0(A_1;\ldots;A_n)x$ voor een woord $A_1\ldots A_n$ uit B.

Nu is het gemakkelijk de geldigheid van de beweringen in A aan te tonen:

- als $p;A \subseteq A;p'$ in A is, dan voor alle x en y:
  als $x(p;A)y$ dan $x(p)x \wedge x(A)y$, dus $x_0(A_1;\ldots;A_{i-1})x \wedge xAy$ met $A_i \equiv A$, dus $x_0(A_1;\ldots;A_i)y \wedge x(A)y$, dus $x(A)y \wedge y(p')y$, i.e. $x(A;p')y$.
- als $p_{in} \subseteq p$ dan triviaal.
- als $p;A \subseteq A;p_{ex}$ in A is, dan net als hierboven.

Welnu, volgens het antecedent volgt uit A de geldigheid van $p_{in};T \subseteq T;p_{ex}$.
Omdat verondersteld was $x_0(T)y_0$ hebben we zelfs $x_0(p_{in})x_0 \wedge x_0(T)y_0$ dus $x_0(T;p_{ex})y_0$ en i.h.b. $y_0(p_{ex})y_0$ hetgeen volgens de gekozen interpretatie voor de asserties betekent $x_0(A_1\ldots A_n)y_0$ voor een of ander woord uit B, dus $x_0(L(B))y_0$. □

Nu volgt onmiddellijk de volgende

Stelling 2

(i) Zowel (a) $A \models p_{in};P \subseteq P;p_{ex}$

Als ook (b) $p_{in};P \subseteq P;p_{ex} \models_{*} A$

(*: mits de asserties in A geschikt gekozen worden).

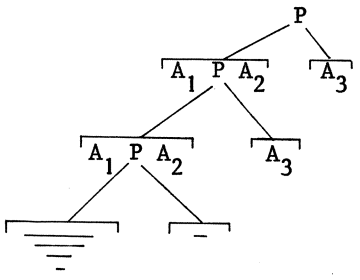
(ii) Voor willekeurig programma T

$A \models p_{in};T \subseteq T;p_{ex}$ slals*) $T \subseteq P$.

Bewijs

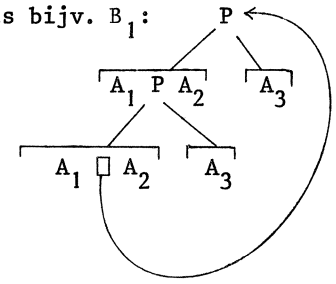
Stelling 1, $P = L(B)$, toegepast op Hoofdlemma (i), (ii) geeft (i(a)(b)). En (ii,⇒) volgt zo ook uit Hoofdlemma (iii), terwijl (ii⇐) volgt uit Hoofdlemma (i) tesamen met hetvolgende feit: als een in-uitvoerbewering geldt voor relatie (P), dan geldt die zeker voor iedere deelrelatie (T), i.e. $T \subseteq P$, $p_{in};P \subseteq P;p_{ex} \models p_{in};T \subseteq T;p_{ex}$. □

We hebben hiermee ons doel bereikt om een regel te formuleren die volledig en karakteriserend is. Dit is eigenlijk niet verwonderlijk, we hebben immers inductieve asserties geplaatst in iedere mogelijke evaluatie van de procedure. Maar het is wel onbevredigend dat het aantal beweringen in de premisse oneindig is, zo gauw het stel procedures recursief is. We kunnen proberen een vereenvoudiging in het stel beweringen aan te brengen. Dat komt er op neer in de volledige afleidingsboom voor P een vereenvoudiging aan te brengen door bijv. daartoe geschikte symbolen te schrappen, mét alles wat eruit voortsproot, en in plaats daarvan een verwijzing te geven naar een voorkomen van dat symbool elders in de boom. Noemen we deze handeling "opknopen", dan is voor de procedure $P \Leftarrow A_1;P;A_2 \cup A_3$ de volledige afleidingsboom B:

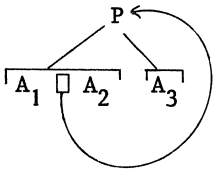


*) Afkorting van "als" gelezen van links naar rechts en "als" gelezen van rechts naar links.

en een opgeknoopte volledige afleidingsboom is bijv. B_1:



en ook B_2:



We definiëren de relatie "volgt direkt na" en de eigenschappen "is linker/rechter voorkomen" zó dat ze behouden blijven voor de voorkomens van symbolen die onder het opknopen onaangeroerd blijven. Omdat we de weggeschrapte symbolen met elders voorkomende identificeren, definiëren we bovendien dat voorkomens van symbolen "direkt na" elkaar volgen, als hun identificaties in de oorspronkelijke boom "direkt na" elkaar volgen. (Deze en ook de andere definities kunnen gemakkelijk geformaliseerd worden, zie FOKKINGA [5],[6].)

Voorbeeld

Ten gevolge van het opknopen is in B_1 de bovenste A_1 nu óók "direkt na" de onderste A_1, en is de bovenste A_2 nu óók "direkt gevolgd door" de onderste A_2. De bovenste A_3 ligt nu óók "direkt tussen" de onderste A_1 en A_2.

Herinner u nu de definitie dat de woorden van een boom precies de rijen van symbolen zijn die achtereenvolgens direkt na elkaar voorkomen, beginnend met een linker- en eindigend met een rechtersymbool.

Voorbeeld. $L(B_1) = \{A_1^n;A_3;A_2^m \mid m=n(\mathrm{mod}2)\}$, $L(B_2) = \{A_1^n;A_3;A_2^m\}$.

Met de definitie van Á onveranderd (maar wel opgeknoopte volledige afleidingsbomen toelatend) blijft het Hoofdlemma en het bewijs ervan correct ! Als nu ook nog stelling 1, $P = L(B)$, blijft gelden na opknoping van bomen, dan zou ook "de stelling", stelling 2, nog gelden. Maar in het algemeen is de verzameling van woorden echt uitgebreid, en daarmee hun totale in-uitvoergedrag.
Laat B nu eens een volkomen willekeurige opgeknoopte volledige afleidingsboom zijn, niet gerelateerd aan P, en Á het stel inductieve beweringen, gebaseerd op boom B. Mogelijkerwijs is Á eindig. Het Hoofdlemma blijft,

zoals gezegd, geldig. Om toch de stelling te concluderen is een noodzakelijk en voldoende voorwaarde dat zowel $P \subseteq L(B)$ alsook $P \supseteq L(B)$ geldt.

Willen we dat "$A \therefore p_{in};P \subseteq P;p_{ex}$" een regel is *met algemene geldigheid*, dus *voor iedere specificatie van de elementaire opdrachten in* P en A, dan zijn de condities $P \subseteq L(B)$ en $P \supseteq L(B)$ gelijkwaardig met $L(B\text{-voor-}P) \subseteq L(B)$ en $L(B\text{-voor-}P) \supseteq L(B)$, i.e. $L(B\text{-voor-}P) \equiv L(B)$, waarbij we nu syntactische inclusie en gelijkheid bedoelen en B-voor-P de volledige afleidingsboom voor P is.
Onder deze voorwaarde, $L(B\text{-voor-}P) \equiv L(B)$, geldt nu

Stelling 3

A kan precies dan eindig gekozen worden, als $L(B\text{-voor-}P)$ regulier is (en dan noemen we P *regulier*).

Bewijs

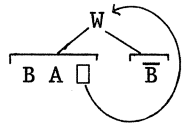
A is eindig precies dan als B eindig is (in het aantal voorkomens van elementaire opdrachtsymbolen). Als B eindig is, kunnen we B als eindige automaat opvatten die precies $L(B)$ accepteert en volgt er dat $L(B) \equiv L(B\text{-voor-}P)$ regulier is.
Omgekeerd, als $L(B\text{-voor-}P)$ regulier is, dan kunnen we een eindige opgeknoopte volledige afleidingsboom B construeren aan de hand van een grammatica voor $L(B\text{-voor-}P)$ in reguliere vorm, zó dat $L(B) \equiv L(B\text{-voor-}P)$.
De A bij deze boom is dan eindig.

5. EEN VOORBEELD

We besluiten met een uitwerking voor de while opdracht. Die opdracht kan gedefinieerd worden als een recursieve procedure

$$W \Leftarrow B;A;W \cup \overline{B} .$$

Beschouwd als een contextvrije grammatica is hij reeds in reguliere vorm.
Een eindige boom voor W is:



En volgens de definitie is het hierop gebaseerde stel inductiebeweringen:

A: $p_{in} \subseteq p_B \quad p_B;B \subseteq B;p_A \quad p_A;A \subseteq A;p_B \quad p_A;A \subseteq A;p_{\tilde{B}}$

$p_{in} \subseteq p_{\bar{B}} \quad p_{\bar{B}};\bar{B} \subseteq \bar{B};p_{ex}$

Hiermee gelijkwaardig is

A': $p_{in} \subseteq p_B \quad p_B;B;A \subseteq A;p_{\overline{B}} \quad p_B;B;A \subseteq A;p_{\overline{B}}$

$p_{in} \subseteq p_{\overline{B}} \quad p_{\overline{B}};\overline{B} \subseteq p_{ex}$

en zelfs

A'': $p_{in} \subseteq p \quad p;B;A \subseteq A;p$

$p;\overline{B} \subseteq p_{ex}$

En daarmee vinden we volgens de stellingen 2 en 3:

(i) zowel $p_{in} \subseteq p,\ p;B;A \subseteq A;p,\ p;\overline{B} \subseteq p_{ex} \models p_{in};W \subseteq W;p_{ex}$

als ook $p_{in};W \subseteq W;p_{ex} \models_{*} p_{in} \subseteq p,\ p;B;A \subseteq A;p,\ p;\overline{B} \subseteq p_{ex}$

(ii) voor willekeurig programma T

$p_{in} \subseteq p,\ p;B;A \subseteq A;p,\ p;\overline{B} \subseteq p_{ex} \models p_{in};T \subseteq T;p_{ex}$ slals $T \subseteq W$.

en derhalve kunnen we als regel -die volledig en karakteriserend is- formuleren:

$$\frac{p_{in} \subseteq p,\ p;B;A \subseteq A;p,\ p;\overline{B} \subseteq p_{ex}}{p_{in};W \subseteq W;p_{ex}} \quad \text{voor } W \Leftarrow B;A;W \cup \overline{B}$$

hetgeen precies Hoare's regel is waarin CONS reeds is opgenomen.

LITERATUUR

[1] DE BAKKER, J.W., *Syllabus van het colloquium programmacorrectheid*, hoofdstuk 8, Mathematisch Centrum Amsterdam, 1974.

[2] DE BAKKER, J.W. & MEERTENS, L.G.L.T., *Simple recursive program schemes and inductive assertions*, Mathematical Centre Report MR 142, Amsterdam, 1972.

[3] DE BAKKER, J.W. & MEERTENS, L.G.L.T., *On the completeness of the inductive assertion method*, Prepublication, Mathematical Centre Report IW 12/73, Amsterdam, 1973.

[4] DE BAKKER, J.W. & DE ROEVER, W.P., *A calculus for recursive program schemes*, in Proc. IRIA Symposium on Automata, Formal Languages and Programming, M. Nivat (ed), North-Holland, Amsterdam, 1972.

[5] FOKKINGA, M.M., *Inductive assertion patterns and recursive procedures*, Report T.H. Delft, 1973.

[6] FOKKINGA, M.M., *Inductive assertion patterns and recursive procedures*, in Proc. Symp. on Programming, april 1974, to appear in: Lecture notes in comp. science, Springer-Verlag, Berlin.

[7] HOARE, C.A.R. & WIRTH, N., *An axiomatic definition of the programming language PASCAL*. Report E.T.H. Zürich.

[8] HOARE, Z, NESS, S. & VUILLEMIN, J., *Inductive methods for proving properties of programs*, in Proc. ACM Conf. on Prov. Ass. about Progr., January 1972.

MATHEMATICAL CENTRE SYLLABUS 21, 1975, XI, 167-175.

ELASTISCHE DATASTRUCTUREN, HUN INVLOED OP PROGRAMMACORRECTHEID

M. REM
Technische Hogeschool, Eindhoven

1. INLEIDING

In het algemeen zullen de langste woorden in de titel de meeste informatie over de inhoud geven, en deze titel vormt daarop geen uitzondering.
Het langste woord uit de titel is *programmacorrectheid*.
Er zijn met betrekking tot de programmacorrectheid ruwweg twee benaderingswijzen: enerzijds is er de kunst van syntactisch correcte programma's te bewijzen dat ze de door de programmeur gewenste eindtoestand bewerkstelligen, anderzijds is er het opstellen van methodologieën om correcte programma's te ontwerpen. Aangezien programma's niet "van God gegeven" zijn, maar door mensen geconstrueerd moeten worden, lijkt de tweede benaderingswijze de meest veelbelovende. Toch zou onrecht gedaan worden aan de eerste benaderingswijze als we het hierbij zouden laten. Resultaten van correctheidsbewijzen, zowel als de bewijsmethoden zelf, kunnen n.l. van groot nut zijn bij het opstellen van zo'n ontwerpmethodologie. Bewijsbaarheidszorgen kunnen op die manier een leidraad vormen bij het programmaontwerp. De beide benaderingswijzen mogen niet los van elkaar gezien worden. Ondanks dat zal hier verder de term programmacorrectheid uitsluitend in de betekenis van de tweede benaderingswijze gebruikt worden.
Het op een na langste woord uit de titel is *datastructuur*.
We zullen het hebben over het inbedden van de door het programma te manipuleren gegevens in de ter beschikking staande datastructuur, of -om het iets nauwkeuriger te zeggen- het representeren van de toestandsruimte in een lineaire datastructuur. Daarbij zullen we voor een datastructuur het volgende begrip gebruiken: "als een *element* het kleinste detail is waarop we informatie wensen te beschouwen (een *atomair* gegeven), dan kan op ieder moment de toestand van een datastructuur gerepresenteerd worden door een string van elementen". In tegenstelling tot de invloed van sequencing primitives op de programmacorrectheid, die al geruime tijd in het middelpunt van de belangstelling staat, is er naar de invloed van de keuze van de datastructuur op de programmacorrectheid nog vrij weinig gekeken.
Het *elastische*, het op twee na langste woord uit de titel, komt voort uit

het streven naar beheersing van de complexiteit van de programmeertaak door middel van *structurering*.
Een systeem is gestructureerd indien het mogelijk is dusdanige deelsystemen te onderscheiden, dat, in vergelijking met de onderlinge invloed tussen de objecten in één deelsysteem, de invloeden tussen objecten van verschillende deelsystemen verwaarloosbaar zijn. (in het jargon: "de intrasystematische bindingen zijn veel sterker dan de intersystematische", zie ANDO & SIMON [1]).
Een dergelijke structuur geeft de mogelijkheid om tijdens het beschouwen van een deelsysteem, van de andere deelsystemen slechts de *interface* (de zwakke intersystematische bindingen) met de huidige te beschouwen, en aldus te abstraheren van de inhoud van de andere deelsystemen. Hoe "dunner" (gladder/prettiger) de interface is, des te gemakkelijker deze abstractie zal gaan.
Dit is de bekende "verdeel en heers"-techniek, waarvoor we de Engelse term *separation of concerns* zullen gebruiken.

2. SEPARATION OF CONCERNS

Het is een nuttige separation of concerns gebleken om door middel van invariante relaties te abstraheren van het aantal malen dat een repeatable statement wordt uitgevoerd (zie DIJKSTRA [2]). Als we een programma moeten maken dat een relatie R tot stand brengt, dan kiezen we een invariant P en een conditie B zó dat $(P \wedge \neg B) \rightarrow R$.
Het programma krijgt dan de volgende structuur:

maak P waar;
<u>while</u> B <u>do</u> een stap in de richting $\neg$ B onder invariantie van P<u>od</u>

We kunnen hierbij een collectie "stappen" bedenken die alle P invariant laten. Van deze collectie kunnen we dan een geschikte stap uitkiezen, een waarvoor het bereiken van $\neg$B gegarandeerd kan worden. We zullen hier niet verder op ingaan. Duidelijk is echter dat hier het probleem van het in stand houden van de invariant gescheiden is van het probleem van de eindigheid.
Een dergelijke separation of concerns zouden we ook graag willen toepassen bij het inbedden van gegevens. We zouden willen abstraheren van het aantal elementen van de datastructuur (het aantal "array-elementen") dat de te manipuleren informatie inneemt. Bij array-achtige datastructuren komen we

dan echter bedrogen uit. Er kan altijd het punt komen dat de informatie niet meer past, wat betekent dat de uitvoering van een operatie zo sterk afhangt van de toevallige toestand van de datastructuur, (dat er zo'n sterke discontinuïteit in het kostenverloop zit) dat men er niet van kan abstraheren, en dat het letten op de huidige maat van de informatie een zorg is die door het hele probleem heen diffundeert. Dit is een voorbeeld van een slechte interface. Het is zo'n ongelukkige interface als die in een (hypothetische) machine waarbij iedere machine-instructie een microseconde duurt, behalve een vermenigvuldiging waarvan de uitkomst een priemveelvoud van 7 plus 4 is, die instructie duurt een orde van grootte langer (zeg een seconde, of als dat nog niet lang genoeg is: een minuut). De zorg dit geval te vermijden zal door alles heen diffunderen. (Voorbeeld afkomstig van DIJKSTRA).

3. ELASTISCHE DATASTRUCTUREN

Veel problemen ontlenen hun complexiteit (of zelfs hun bestaansrecht) aan de gesignaleerde "starheid" van het geheugen.
Om er een paar te noemen:

(1) de implementatie van string-operaties als daaronder ook de vervanging van een deelstring door een mogelijk langere voorkomt;
(2) de implementatie van een "linked list" omdat tussenvoegingen en wegnemingen van lijst-elementen mogelijk moeten zijn;
(3) de implementatie van meerdere stacks;
(4) overflow in Hash-tabellen; enz.

Teneinde tijdens de inbeddingsproblematiek te kunnen abstraheren van het aantal benodigde elementen, dient de datastructuur *elastisch* te zijn. Een elastische datastructuur is een datastructuur waarin de operaties "*neem* een element *weg*" en "*voeg* een element *tussen*" met vergelijkbaar gemak kunnen worden uitgevoerd als een *inspectie* of een *toekenning* aan een element. (Met een dusdanig vergelijkbaar gemak dat het toegestaan is te abstraheren van het kostenverschil.)
In de volgende paragrafen zullen we onderzoeken hoe de operaties op elastische datastructuren dienen te worden gedefinieerd. Het zal blijken dat het daar voorgestelde mechanisme een van de weinige mogelijkheden is om de gebruiker ten volle van de elasticiteit te laten profiteren.

4. SELECTIEMECHANISME

Teneinde in staat te zijn operaties op de datastructuur te definiëren, moet er een *selectiemechanisme* zijn om elementen te kunnen selecteren. Dit mechanisme dient zó gekozen te worden dat de elasticiteit behouden blijft. Door deze eis valt *adresseren* -d.w.z. het opvolgend nummeren van de elementen- als selectiemechanisme door de mand. Immers, dit zou betekenen dat na een tussenvoeging of een wegneming van een element alle volgende elementen hernummerd moeten worden, en dat alle verwijzingen daarnaar moeten worden aangepast. Dit zou tot gevolg hebben dat door de keuze van het selectiemechanisme de elasticiteit van de datastructuur weer verloren gaat.
We kunnen de elasticiteit van de datastructuur behouden door de elementen in twee disjuncte klassen te verdelen: *markers* en *symbolen*. Op ieder ogenblik dienen de aanwezige markers verschillend te zijn. De enige te bereiken elementen zijn dan de directe buurelementen van markers. Uiteraard kunnen markers worden verzet teneinde ook andere elementen te selecteren.
Markers kunnen elkaars directe buren zijn. De vraag, die dan naar voren komt, is: "Wordt met een marker altijd zijn directe buurelement geselecteerd, zelfs als dit ook een marker is, of steeds het dichtstbijzijnde *symbool*?" Aangezien we niet willen dat door een toekenning (die tot taak heeft een symbool in de datastructuur te vervangen door een ander symbool) de selectiemogelijkheid van andere symbolen verstoord wordt, kiezen we voor het alternatief dat het dichtstbijzijnde symbool wordt geselecteerd. Dit verschaft ons de conceptuele duidelijkheid de markers op te vatten als de handvaten naar de symbolen in de datastructuur.
Deze keuze heeft feitelijk tot gevolg dat markers die elkaars buren zijn hun onderlinge volgorde verliezen. Immers, stel dat twee symbolen a en b onderling gescheiden worden door twee markers m en n. De toestand van de datastructuur is dan

```
     m n                          n m
...a| |b...     of      ...a| |b...
```

In beide gevallen kunnen via m en n slechts a en b geselecteerd worden, en de juiste volgorde van m en n doet niet meer ter zake. Een dergelijke (ongeordende) groep markers noemen we een *cluster*.

5. SCHEIDING

Tot nu toe werd U enigszins misleid door te suggereren dat er voor de operaties op de datastructuur steeds een element (of liever: een symbool) geselecteerd moet worden. In conventionele datastructuren is dat inderdaad het geval, en de misleiding zal U dan ook wellicht ontgaan zijn, maar in datastructuren waarop ook de operatie "voeg tussen" gedefinieerd is, is het selecteren van een element niet meer voldoende. Voor deze laatste operatie moet n.l. een *scheiding* tussen twee elementen geselecteerd worden, te weten de plaats waar het nieuwe symbool tussengevoegd moet worden.
Nu wordt de dubbelrol van de markers duidelijk: enerzijds wordt een marker gebruikt om een symbool te selecteren, anderzijds om een scheiding te selecteren. Deze dualiteit suggereert het selectiemechanisme zó te kiezen dat met iedere marker in de datastructuur precies één symbool en één scheiding geselecteerd wordt.

1. Welk *symbool* selecteert een marker dan?

   Er zijn twee voor de hand liggende oplossingen om iedere marker precies één symbool te laten selecteren:

   1.1. alle markers wijzen in dezelfde richting (d.w.z. hetzij links, hetzij rechts);

   1.2. we onderscheiden links-wijzende en rechts-wijzende markers.

   Aangezien er zowel toepassingen zijn waarin het voorafgaande symbool, als toepassingen waarin het opvolgende symbool geselecteerd moet worden, kiezen we voor 1.2. (We zullen dergelijke toepassingen zien.)
   We zullen verder de volgende notatie gebruiken:

   a, b, c voor symbolen;
   p, r voor links-wijzende markers;
   s, t voor rechts-wijzende markers.

2. Welke *scheiding* selecteert een marker?

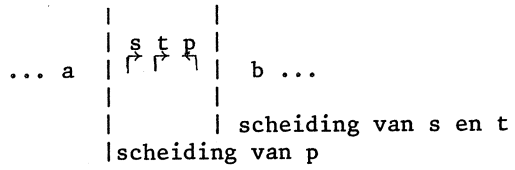
   Of: waar vindt met een gegeven marker de tussenvoeging plaats? Er zijn twee eisen waaraan de tussenvoegingsoperatie moet voldoen:

   (a) Na een tussenvoeging wijst de marker, waarmee de operatie is uitgevoerd, naar het tussengevoegde symbool.

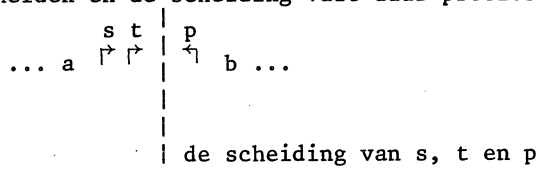
   (b) Tussenvoegingen met behulp van geclusterde markers van hetzelfde type (dus hetzij links-, hetzij rechtswijzend) vinden op dezelfde scheiding plaats.

Met deze eisen zijn er twee mogelijkheden voor de definitie van de scheiding:

2.1. Een links-wijzende (rechts-wijzende) marker selecteert de scheiding direct links (rechts) van zijn cluster:



2.2. De links- en rechts-wijzende markers in de cluster worden onderling gescheiden en de scheiding valt daar precies tussen in:



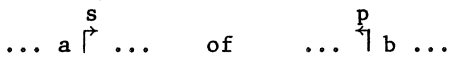
We zullen nog geen keuze tussen 2.1 en 2.2 doen, teneinde deze vrijheid te kunnen exploiteren in het voldoen aan de volgende eis:

6. HET PARTITIONEREN VAN DE DATASTRUCTUUR

Als het niet zo slecht geklonken had, zouden we geschreven hebben: "Het structureren van de datastructuur", want dat is wat we graag willen. We willen in staat zijn deelstukken van de datastructuur te onderscheiden, en wel zodanig dat we tijdens het beschouwen van het ene stuk mogen abstraheren van het bestaan van de andere stukken.
Voor deze verdeling willen we weer markers gebruiken, aangezien markers onze enige "handvaten" aan de datastructuur zijn. Dit betekent dat we weer een dubbelrol voor de markers introduceren: enerzijds worden ze gebruikt om een symbool of een scheiding te selecteren, anderzijds stellen we nu voor ze te gebruiken om stukken van de datastructuur af te bakenen. We zullen zien dat deze beide rollen tegenstrijdige eisen aan de markers opleggen.
Stel, bijvoorbeeld, dat we een verzameling objecten van verschillende soorten willen ordenen tot disjuncte verzamelingen. Men zou dit kunnen doen door een datastructuur met behulp van markers te verdelen, en de objecten achtereenvolgens in het juiste deelstuk te plaatsen. Het probleem is dan echter dat in de begintoestand alle markers geclusterd zijn, en dat ze

dientengevolge hun onderlinge volgorde verloren hebben. Is het voor de eerste rol van de markers belangrijk gebleken, dat buurmarkers geen onderlinge volgorde hebben, voor de markers die als "grenspalen" gebruikt worden is het juist essentieel dat ze hun onderlinge volgorde behouden als ze toevallig naast elkaar staan. Dit laatste kan worden bereikt door deze grenspalen "dikte" te geven. We doen dit op de volgende wijze:

Om een datastructuur te verdelen in substructuren gebruiken we *grenspalen* bestaande uit een marker en het symbool waar hij *vandaan* wijst, dus



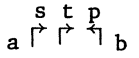
Dat hiermede de volgorde van de grenspalen behouden blijft als ze toevallig elkaars buren zijn, volgt direct uit het feit dat symbolen een onderlinge volgorde hebben. Een dergelijke substructurering heeft alleen zin als de frequentie waarmee grenspalen worden neergezet en verwijderd, een orde lager is dan de frequentie waarmee operaties op de substructuren worden uitgevoerd (zie SIMON [3]). Daartoe zal een willekeurige tussenvoeging nooit een grenspaal mogen doorklieven. Dit betekent dat een scheiding nooit tussen de twee delen van een grenspaal mag vallen. Alleen definitie 2.2 voldoet hieraan. Het is op dit moment dat we besluiten dat 2.2 de definitie is die we kiezen.

7. RACK

We zijn nu in staat de datastructuur *rack*, de elastische tegenhanger van het array, te definiëren. Een rack bestaat initieel uit een grenspaal waarvan de marker rechts-wijzend is, gevolgd door een grenspaal waarvan de marker links-wijzend is. Door tussenvoegingen en het plaatsen van markers kunnen er in de loop van het bestaan van de rack symbolen, markers en meer grenspalen tussen deze buitenste grenspalen voorkomen:

Als voorbeeld zullen we met behulp van een rack een *left queue* implementeren. Een left queue is een lineaire buffer waarin het toevoegen van symbolen aan het rechter einde, en het wegnemen aan het linker einde, plaatsvindt. In deze queue beweegt zich een cursor van links naar rechts ("als

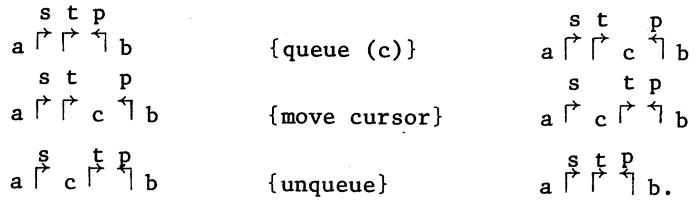
een forel tegen de stroom") op zijn weg alle objecten inspecterend die hij tegenkomt. Voor deze cursor gebruiken we een rechts-wijzende marker t. De lege queue is dan van de vorm:



Operaties op de queue en inlichtingen over de toestand van de queue kunnen dan als volgt worden vertaald:

queue (c)	voeg symbool c tussen in de scheiding van p;
unqueue	neem het symbool van s weg;
queue empty?	s en p geclusterd?
inspect	inspecteer het symbool van t;
move cursor	zet t vooruit over zijn symbool;
whole queue inspected?	t en p geclusterd?

Deze operaties brengen de volgende toestandsveranderingen in de datastructuur teweeg:



8. SLOTOPMERKINGEN

Als men, uitgaande van de stelling "programmacorrectheid is het opstellen van ontwerpmethoden om correct te programmeren", ontwerpmethodologische eisen oplegt aan het manipuleren met datastructuren, dan doet de behoefte aan elastische geheugens zich voelen.
Als men een elastisch geheugen zou hebben, dan is het hier beschreven model een van de weinige mogelijkheden om de gebruiker een volledig elastische datastructuur te verschaffen. Dit betekent met name dat *random-access* en elasticiteit tegenstrijdige eisen zijn. Overigens heeft men in een systeem met starre datastructuren en een virtueel geheugenbeheer ook geen echte random-access. Het wild springen door het gegevensbestand is ook daar niet aan te raden.

LITERATUUR

[1] ANDO, A. & H.A. SIMON, *Aggregation of variables in dynamic systems*, Econometrica 29 (1961) 111-138.

[2] DIJKSTRA, E.W., *Notes on structured programming*, in: O.-J. DAHL, E.W. DIJKSTRA & C.A.R. HOARE, *Structured programming*, Academic Press, New York, 1972.

[3] SIMON, H.A., *The sciences of the artificial*, M.I.T. Press, Cambridge, Mass., 1969.

MATHEMATICAL CENTRE SYLLABUS 21, 1975, XII, 177-186.

PROGRAMMACORRECTHEID EN GRAMMATICA'S

A. van WIJNGAARDEN

1. INLEIDING

De bedoeling van deze voordracht is om aan de hand van een frivool maar niet triviaal probleem de macht van twee-niveau grammatica's te laten zien bij het bewijzen van de juistheid van een programma.

2. PROBLEEMSTELLING

Op een smal rotspad beweegt zich een rij van a schapen en ontmoet een rij van b schapen die zich in de tegengestelde richting beweegt. De schapen lopen alle kop aan staart en de twee rijen houden halt op een afstand ter grootte van een schaap, bijvoorbeeld als $a = 2$ en $b = 3$:

">>.<<<".

Het probleem is nu hoe de twee rijen schapen elkaar kunnen passeren als gegeven is dat

1) een schaap zich nooit achteruit beweegt;
2) een schaap dat voor zich een open plaats ziet, deze plaats kan opvullen, daarbij een open plaats achterlatend (schuiven);
3) een schaap dat voor zich een schaap van tegengestelde richting en daarachter een open plaats ziet, over dit schaap kan heenspringen en deze plaats kan opvullen, daarbij een open plaats achterlatend (springen);
4) geen schaap zich buiten het gebied ter lengte van $a + b + 1$ schaaplengten begeeft voordat de twee rijen elkaar hebben gepasseerd en door een afstand ter lengte van een schaap zijn gescheiden:

"<<<.>>".

Met behulp van terugkrabbelen kunnen we gemakkelijk alle manieren vinden waarop de schapen te werk kunnen gaan. Uitgaande van de stand

1: ">>.<<<"

zijn slechts twee "zetten" mogelijk, leidend tot

2: ">.><<<"

of tot

2': ">><.<<".

In stand 2 zijn twee zetten mogelijk, leidend tot

3: "><>.<<"

of tot

3': ".>>><<<".

Stand 3' is een "dode stand" en dus moet stand 3 worden gekozen die kan leiden tot

4: "><><.<"

of tot

4': "><.><<".

Stand 4' leidt tot

".<>><<",

welke stand alleen leidt tot de dode stand

"<.>><<",

of tot

"><<>.<",

welke stand alleen leidt tot de dode stand

"><<<.>".

Dus moet stand 4 worden gekozen die kan leiden tot

5: "><.<><"

of tot de dode stand

5': "><><<.".

Dus moet stand 5 worden gekozen die kan leiden tot

6: ".<><><"

of tot

6': "><<.><".

Stand 6' leidt alleen tot de dode stand

"><<<.>".

Dus moet stand 6 worden gekozen die alleen leidt tot

7: "<.><><",

8: "<<>.><".

Stand 8 kan leiden tot

9: "<<><>."

of tot de dode stand

9': "<<.>><".

Dus moet stand 9 worden gekozen die alleen leidt tot

10: "<<><.>",

11: "<<.<>>",

12: "<<<.>>",

waarmee een oplossing is gevonden.
Op gelijke wijze leidt stand 2' tot

3': ">.<><<"

of tot

3'': ">><<.<".

Stand 3'' leidt alleen tot de dode stand

">><<<.".

Dus moet stand 3' worden gekozen die leidt tot

4': ".><><<"

of tot

4'': "><.><<".

Stand 4'' leidt tot een van de twee dode standen

"<.>><<"

of

"><<<.>".

Dus moet stand 4' worden gekozen die alleen leidt tot

5': "<>.><<".

Stand 5' leidt tot

6': "<><>.<"

of tot de dode stand

"<.>><<".

Dus moet stand 6' worden gekozen die leidt tot

7': "<><><."

of tot

7'': "<><.><".

Stand 7'' leidt tot een van de twee dode standen

"<><<.>" of "<<.>><".

Dus moet stand 7' worden gekozen die alleen leidt tot

8': "<><.<>".

Stand 8' leidt tot

9': "<.<><>"

of tot de dode stand

"<><<.>".

Dus moet stand 9' worden gekozen die alleen leidt tot

10': "<<.><>",

11': "<<<>.>",

12': "<<<.>>",

waarmee de tweede oplossing is gevonden.
Samenvattend hebben we dus de volgende oplossingen gevonden:

1: >>.<<<	1 : >>.<<<
2: >.><<<	2': >><.<<
3: ><>.<<	3': >.<><<
4: ><><.<	4': .><><<
5: ><.<><	5': <>.><<
6: .<><><	6': <><>.<
7: <.><><	7': <><><.
8: <<>.><	8': <><.<>
9: <<><>.	9': <.<><>
10: <<><.>	10': <<.><>
11: <<.<>>	11': <<<>.>
12: <<<.>>	12': <<<.>>

3. SCHAPENGRAMMATICA

Natuurlijk kan men een programma schrijven dat aldus door middel van terugkrabbelen voor willekeurige waarden van a en b alle oplossingen bepaalt. Zo'n programma is evenwel hopeloos inefficiënt want men kan een strategie aangeven die zonder proberen in elke stand de juiste zet(ten) bepaalt. Om die strategie te vinden bekijken we de gevonden oplossingen eens nader. We noemen een stel schapen dat kop aan kop staat, dus "><", een paartje. We zien dan dat, althans voor ons voorbeeld, de volgende strategie juist is:

```
begin
   while zet mogelijk
   do if sprong mogelijk # 2, 4, 5, 7, 8, 10, 2', 4', 5', 7', 8', 10' #
    then spring # kan maar op één wijze #
    elif schuif op twee wijzen mogelijk # 1, 3, 3', 6' #
    then schuif zo dat er geen twee paartjes ontstaan
         die door de open plaats gescheiden zijn
         # dit kan mogelijk op twee wijzen, 1 #
    else schuif # 6, 9, 11, 9', 11' #
      fi
   od
    # klaar, 12, 12' #
end
```

Er van uitgaande dat deze strategie inderdaad juist is gaan we haar "programmeren" met behulp van een twee-niveau grammatica, dat wil zeggen, we maken een twee-niveau grammatica met een beginsymbool (axioma) "probleem" waarvan de terminale producties juist bestaan uit alle mogelijke lijstjes van toestanden waarvan we boven twee voorbeelden - met dezelfde begintoestand - hebben gegeven. Voor hen die bij het woord "grammatica" denken aan een "taal" kunnen wij ook zeggen dat wij de grammatica zoeken van een zeer speciale programmeertaal, nl. een waarvan elk "programma" (pardon, "probleem") een schapenontmoeting inclusief haar gehele afwikkeling beschrijft. Zulk een programmeertaal kent alleen een syntaxis want de semantiek is leeg: de "probleemstelling" is tegelijk van de "oplossing" voorzien!
Zulk een twee-niveau grammatica is een formeel spel met kleine en grote syntactische symbolen (letters). Men hoeft zich daarbij niets te denken, maar wellicht wordt het spel interessanter als "men" met de letter "l" een schaap associeert dat naar links wil, met "r" een schaap dat naar rechts wil en met "v" de vrije plaats.
Na enig proberen poneren wij de volgende grammatica:

Schapengrammatica

M1) L :: ; lL.
M2) R :: ; Rr.
M3) X :: ; Xrl.
M4) D :: L ; Rr.
M5) E :: D.
M6) F :: lXD ; R.
M7) G :: L ; DXr.
M8) S :: l ; r ; v.
M9) W :: S ; WS.
H1) probleem : pRrvlL , RrvlL.
H2) pWS : pW , pS.
H3) pS : S symbool.
H4) DrvlXE : qDvrlXE.
H5) DXrvlE : qDXrlvE.
H6) DXrlvF : qDXvlrF.
H7) GvrlXD : qGlrvXD.
H8) DXrvR : qDXvrR.
H9) LvlXD : qLlvXD.

H10) LvR : .
H11) qW : d symbool , pW , W.

symbool	voorstelling
l symbool	<
r symbool	>
v symbool	.
d symbool	→

"Men" associeert wellicht met L een mogelijk lege rij naar links gerichte schapen, met R een mogelijk lege rij naar rechts gerichte schapen en met D of E een mogelijk lege rij gelijkgerichte schapen. Met X associeert men dan een mogelijk lege rij paartjes. Met F associeert men dan een naar links gericht schaap gevolgd door een mogelijk lege rij paartjes gevolgd door een mogelijk lege rij gelijkgerichte schapen, of een mogelijk lege rij naar rechts gerichte schapen. Evenzo associeert men dan met G een mogelijk lege rij naar links gerichte schapen, of een mogelijk lege rij gelijkgerichte schapen gevolgd door een mogelijk lege rij van paartjes gevolgd door een naar rechts gericht schaap. Met S associeert men dan een schaap danwel de vrije plaats en met W een rij van schapen en vrije plaatsen.

Door inspectie van de hyperregels ziet men dat alleen H1, H2 en H11 noties produceren die met p beginnen. In H1 wordt die p door minstens drie letters (rvl) gevolgd, zodat H2 van toepassing is. Voorts ziet men dat alleen H4, ..., H9 noties produceren die met q beginnen en dat die q door minstens twee of drie letters wordt gevolgd, zodat H11 van toepassing is. In H11 wordt dus de p ook door minstens twee letters gevolgd, zodat H2 van toepassing is. Tenslotte produceert H2 zelf een notie p gevolgd door een rij van minstens twee letters om tot een notie p gevolgd door een rij letters die één korter is en tot een notie p gevolgd door één letter. Door herhaalde toepassing van H2 wordt dus iedere notie p gevolgd door twee of meer letters omgezet tot een rij noties p gevolgd door één letter die door H3 worden geproduceerd tot voorstelbare symbolen. De terminale productie van een notie die met p begint is dus een "zichtbare" voorstelling van wat achter die p staat. Zo is de voorstelling van de terminale productie van prrvlll die van onze schapen: >>.<<<. Men kan dus p interpreteren als "print". H1 print de beginstand, gevolgd door een notie die verwerkt wordt door een van de hyperregels H4, ..., H9, die elkaar de notie toespelen via

H11, welke de nieuwe stand print, tot het spel ophoudt met H10.

Men vergewist zich door proberen dat de voorstelling van een bepaalde terminale productie van "probleem" is

```
 >>.<<<→>.><<<→><>.<<→><><.<→><.<><
→.<><><→<.><><→<<>.><→<<><>.→<<><.>
→<<.<>>→<<<.>>
```

in overeenstemming met een van beide hierboven behandelde voorbeelden.

Wij zullen nu bewijzen dat de gegeven grammatica ons probleem oplost. Daarbij kunnen wij verder p en q negeren omdat zij uitsluitend dienen om de standen vast te leggen. Allereerst zien wij dat H1 inderdaad de beginstand karakteriseert, nl. een aantal schapen naar rechts, de vrije plaats en een aantal schapen naar links: RrvlL. Welke regel kan hier iets mee uitvoeren? Door alleen te kijken naar de geëiste volgorde rvl ziet men dat alleen H4 en H5 een kans maken en bij nadere inspectie dat zij in beide gevallen door passende keuze van D, X en E inderdaad toepasbaar zijn. Dus als $\Rightarrow$ betekent "leidt tot toepassing van" vinden wij

$$H1 \Rightarrow H4, H5.$$

Voorts verifiëren wij dat de regels H4, ..., H10 inderdaad de spelregels gehoorzamen. In H4 en H8 schuift een schaap naar rechts en in H5 en H9 naar links. In H6 springt een schaap naar rechts en in H7 naar links. In H10 gebeurt niets maar dan is ook de eindtoestand bereikt.

Het resultaat van H4 (afgezien van de q) is van de vorm DvrlXE. Dit is zeker niet van de vorm vereist voor H4 of H5 want daarin volgt op de v een l. Evenmin is H6 van toepassing want daarin volgt op de v (zie M6) hetzij een l danwel een mogelijk lege rij van letters r, dus niet rl. H8 is niet van toepassing omdat daarin v hoogstens door letters r wordt gevolgd en H9 is niet van toepassing omdat daarin de v door l wordt gevolgd. H10 is niet van toepassing omdat rechts van v een l voorkomt. De enig overblijvende kandidaat is dus H7 en inderdaad kunnen, wat ook de D, X en E uit H4 mogen zijn, de G, X en D in H7 zo worden gekozen dat de zaak klopt. Dus vinden wij:

$$H4 \Rightarrow H7$$

en op analoge wijze

$$H5 \Rightarrow H6.$$

Wat meer werk is de bepaling van de opvolgers van H6 en H7. H6 levert een

resultaat van de vorm DXv1rF. H4 accepteert alles van de vorm D'rv1X'E'. Dus moet gelden DX = D'r en rF = X'E' wil H4 toepasbaar zijn. De eerste voorwaarde eist dat X leeg is en D de vorm Rr heeft. Dit is echter ook voldoende. Immers door te kiezen D' = R is aan de eerste voorwaarde voldaan. Voorts kan men als F = 1X''D'' kiezen X' = r1X'' en E' = D'' en als F = R'' kan men kiezen X' = en E' = rR'' waardoor in beide gevallen aan de tweede voorwaarde voldaan is. Op dezelfde wijze kan men nagaan onder welke voorwaarden een van de regels H5, ..., H10 toepasbaar is. Als resultaat vinden wij

H6 ⇒ (X = | (D = L | H9 |: F = R | H4, H5 | H4) | H6)

en op analoge wijze

H7 ⇒ (X = | (D = R | H8 |: G = L | H4, H5 | H5) | H7).

Deze resultaten zijn weliswaar wat gecompliceerder maar het belangrijkste ervan is dat er altijd één opvolger en zelfs soms twee opvolgers van H6 en H7 zijn.

Voor de opvolgers van H8 en H9 vinden wij tenslotte

H8 ⇒ (X ≠ | H6 |: D ≠ L | H8 | H10)

en

H9 ⇒ (X ≠ | H7 |: D ≠ R | H9 | H10).

Hieruit zien wij dat ook H8 en H9 altijd een opvolger hebben. De enige toestand die geen opvolger heeft is H10 maar dit is de gewenste eindstand.

Hiermede hebben wij bijna de juistheid van onze grammatica aangetoond. De beginstand is de gegevene, de eindstand, zo die ooit wordt bereikt, is de gewenste, aan de spelregels wordt voldaan en zolang de eindstand niet is bereikt wordt steeds een zet ondernomen. Alleen moeten wij nog bewijzen dat de eindstand H10 inderdaad bereikt wordt.

Daartoe voeren wij het begrip "taak" in, dat is de som van de afstanden, gemeten in schaaplengten, die elk schaap verwijderd is van zijn positie in de eindstand. Bij de aanvang heeft elk rechtsgericht schaap nog een afstand $b + 1$ en elk linksgericht schaap nog een afstand $a + 1$ af te leggen. In de beginstand is de taak dus $2ab + a + b$ en in de eindstand uit de aard der zaak 0. Iedere zet vermindert echter de taak met 1 of 2 al naar gelang geschoven of gesprongen wordt. Dus kan er maar een eindig aantal zetten worden uitgevoerd en dus moet de eindstand worden bereikt. Het juiste aantal zetten is overigens direct te bepalen. Immers ieder

linksgericht schaap moet ieder rechtsgericht schaap passeren wat door een sprong gebeurt. Er vinden dus ab sprongen plaats die de taak met $2ab$ verminderen. Er blijft dus nog een taak $a + b$ over die door $a + b$ schuiven moet verdwijnen. Er zijn dus totaal $ab + a + b$ zetten nodig. In ons voorbeeld $a = 2$, $b = 3$ hadden wij dan ook volgens beide manieren 11 zetten nodig. Hiermee is de juistheid van de grammatica bewezen.

4. SCHAPENPROGRAMMA

Wij gaan nu in een klassieke programmeertaal, zeg ALGOL 68, een programma schrijven dat ons probleem oplost. Daarbij baseren wij ons op de gegeven grammatica om daarmee ook de juistheid van ons programma te garanderen.

De rij schapen houden we bij als een string opgebouwd uit a karakters ">", b karakters "<" en 1 karakter ".". Daarmee houden wij ons geheel aan het voorbeeld van de grammatica. Nu trachten wij een wat handzamer manier te vinden om de toestand te karakteriseren. Daarbij willen wij alleen in de omgeving van de "." kijken, die we daartoe angstvallig blijven volgen. Door inspectie zien wij dat r|v alleen en altijd in H6 en v|r alleen en altijd in H7 voorkomt. H6 en H7 kunnen wij dus herkennen door te kijken naar "><." en ".><". Als deze test faalt weten wij dat de toestand een andere is dan H6 of H7. Nu merken wij op dat rv|r| alleen voor *kan* komen in H4 en dat r|rv| alleen voor *kan* komen in H5. Deze gevallen kunnen wij herkennen door te kijken naar ">.<><" en "><>.<". Als deze testen falen dan weten wij dat als de toestand toch H4 of H5 is, daarin de X leeg is. Maar dan zijn H4 en H5 identiek! Nu komt rv| alleen in H4 en H5 voor. Als wij nu dus toch ">.<" vinden dan zijn zowel H4 als H5 toepasbaar. Wij zullen ons programma zo maken dat achtereenvolgens beide keuzen vervolgd worden. Als ook deze test mislukt moet de toestand H8, H9 of H10 zijn. Onder deze conditie komt rv dus ">." alleen in H8 voor en v| dus ".<" alleen in H9. Als beide tests falen is de toestand dus H10, de eindtoestand. Wij hebben nu een solide basis om ons programma op te bouwen. Het resultaat is nu zonder veel moeite te lezen. Voor de aardigheid hebben wij de mogelijkheden $a = 0$ en $b = 0$ meegenomen. Deze gevallen behoeven dus nu een afzonderlijk bewijs, dat evenwel door directe verificatie is te leveren. Immers als $a = 0$ dan werkt alleen de test op H9 en als $b = 0$ alleen de test op H8 en schuiven de schapen rustig op. Natuurlijk kunnen

wij ook onze grammatica hiervoor laten zorgen. Dit kan eenvoudig door H1 te wijzigen in H1') probleem: pRvL, RvI.
Dan wordt de opvolgingsregel van H1'

H1' ⇒ (R = | (L = | H10 | H9) |: L = | H8 | H4, H5)

en de rest van de redenering blijft onveranderd geldig.

Schapenprogramma

```
begin
   proc p = (int i, j, string s) void :
      # i is regelnummer, j de plaats van de punt in s; s is links en
        rechts aangevuld met drie spaties om het testen te beveiligen #
     begin print ((newline, i, s[4:n]));
           if s[j-2:j] = "><."                                    # H6 #
        then p(i+1, j-2, s[:j-3] + ".<>" + s[j+1:])
        elif s[j:j+2] = ".><"                                     # H7 #
        then p(i+1, j+2, s[:j-1] + "<>." + s[j+3:])
        elif s[j-1:j+3] = ">.<><"                                 # H4 #
        then p(i+1, j-1, s[:j-2] + ".>" + s[j+1:])
        elif s[j-3:j+1] = "><>.<"                                 # H5 #
        then p(i+1, j+1, s[:j-1] + "<." + s[j+2:])
        elif s[j-1:j+1] = ">.<"                                   # H4, H5 #
        then print (newline);
             p(i+1, j-1, s[:j-2] + ".>" + s[j+1:]);
             print (newline);
             p(i+1, j+1, s[:j-1] + "<." + s[j+2:])
        elif s[j-1:j] = ">."                                      # H8 #
        then p(i+1, j-1, s[:j-2] + ".>" + s[j+1:])
        elif s[j:j+1] = ".<"                                      # H9 #
        then p(i+1, j+1, s[:j-1] + "<." + s[j+2:])
          fi
     end ;
   int a, b; read ((a, b)); int n = a + b + 7;
     if a ≥ 0 ∧ b ≥ 0
   then p(1, a + 4, "..." + a × ">" + "." + b × "<" + "...")
     fi
end
```

UITGAVEN IN DE SERIE MC SYLLABUS

Onderstaande uitgaven zijn verkrijgbaar bij het Mathematisch Centrum, 2e Boerhaavestraat 49 te Amsterdam-1005, tel. 020-947272.

MCS 1.1 F. GÖBEL & J. VAN DE LUNE, *Leergang Besliskunde, deel 1: Wiskundige basiskennis*, 1965. ISBN 90 6196 014 2.

MCS 1.2 J. HEMELRIJK & J. KRIENS, *Leergang Besliskunde, deel 2: Kansberekening*, 1965. ISBN 90 6196 015 0.

MCS 1.3 J. HEMELRIJK & J. KRIENS, *Leergang Besliskunde, deel 3: Statistiek*, 1966. ISBN 90 6196 016 9.

MCS 1.4 G. DE LEVE & W. MOLENAAR, *Leergang Besliskunde, deel 4: Markovketen, en wachttijden*, 1966. ISBN 90 6196 017 7.

MCS 1.5 G. DE LEVE & J. KRIENS, *Leergang Besliskunde, deel 5: Inleiding tot de mathematische besliskunde*, 1966. ISBN 90 6196 018 5.

MCS 1.6a B. DORHOUT & J. KRIENS, *Leergang Besliskunde, deel 6a: Wiskundige programmering 1*, 1968. ISBN 90 6196 032 0.

MCS 1.7a G. DE LEVE, *Leergang Besliskunde, deel 7a: Dynamische programmering 1*, 1968. ISBN 90 6196 033 9.

MCS 1.7b G. DE LEVE & H.C. TIJMS, *Leergang Besliskunde, deel 7b: Dynamische programmering 2*, 1970. ISBN 90 6196 055 X.

MCS 1.7c G. DE LEVE & H.C. TIJMS, *Leergang Besliskunde, deel 7c: Dynamische programmering 3*, 1971. ISBN 90 6196 066 5.

MCS 1.8 J. KRIENS et al., *Leergang Besliskunde, deel 8: Minimaxmethode, netwerkplanning, simulatie*, 1968. ISBN 90 6196 034 7.

MCS 2.1 G.J.R. FÖRCH et al., *Colloquium stabiliteit van differentieschema's, deel 1*, 1967. ISBN 90 6196 023 1.

MCS 2.2 L. DEKKER et al., *Colloquium stabiliteit van differentietieschema's, deel 2*, 1968. ISBN 90 6196 035 5.

MCS 3.1 H.A. LAUWERIER, *Randwaardeproblemen, deel 1*, 1967. ISBN 90 6196 024 X.

MCS 3.2 H.A. LAUWERIER, *Randwaardeproblemen, deel 2*, 1968. ISBN 90 6196 036 3.

MCS 3.3 H.A. LAUWERIER, *Randwaardeproblemen, deel 3*, 1968. ISBN 90 6196 043 6.

MCS 4 H.A. LAUWERIER, *Representaties van groepen*, 1968. ISBN 90 6196 037 1.

MCS 5 J.H. VAN LINT et al., *Colloquium discrete wiskunde*, 1968. ISBN 90 6196 044 4.

MCS 6 K.K. KOKSMA, *Cursus ALGOL 60*, 1969. ISBN 90 6196 045 2.

MCS 7.1 *Colloquium moderne rekenmachines, deel 1*, 1969. ISBN 90 6196 046 0.

MCS 7.2 *Colloquium moderne rekenmachines, deel 2*, 1969. ISBN 90 6196 047 9.

MCS 8 H. BAVINCK & J. GRASMAN, *Relaxatietrillingen*, 1969. ISBN 90 6196 056 8.

MCS 9.1 T.M.T. COOLEN et al., *Colloquium elliptische differentiaalvergelijkingen, deel 1*, 1970. ISBN 90 6196 049 5.

MCS 10 J. FABIUS & W.R. VAN ZWET, *Grondbegrippen van de waarschijnlijkheidsrekening*, 1970. ISBN 90 6196 057 6.

MCS 11 H. BART et al., *Colloquium halfalgebra's en positieve operatoren*, 1971. ISBN 90 6196 067 3.

MCS 12 T.J. DEKKER, *Numerieke algebra*, 1971. ISBN 90 6196 068 1.

MCS 13 F.E.J. KRUSEMAN ARETZ, *Programmeren voor rekenautomaten; De MC ALGOL 60 vertaler voor de EL X8*, 1971. ISBN 90 6196 069 X.

MCS 14 H. BAVINCK et al., *Colloquium approximatietheorie*, 1971. ISBN 90 6196 070 3.

MCS 15.1 T.J. DEKKER et al., *Colloquium stijve differentiaalvergelijkingen, deel 1*, 1972. ISBN 90 6196 078 9.

MCS 15.2 P.A. BEENTJES et al., *Colloquium stijve differentiaalvergelijkingen, deel 2*, 1973. ISBN 90 6196 079 7.

*MCS 15.3 P.A. BEENTJES et al., *Colloquium stijve differentiaalvergelijkingen, deel 3*, 1972.

MCS 16.1 L. GEURTS, *Cursus programmeren, deel 1: De elementen van het programmeren*, 1973. ISBN 90 6196 080 0.

MCS 16.2 P.A. BEENTJES et al., *Cursus programmeren, deel 2: De programmeertaal ALGOL 60*, 1973. ISBN 90 6196 087 8.

MCS 17.1 P.S. STOBBE, *Lineaire algebra, deel 1*, 1974. ISBN 90 6196 090 8.

MCS 17.2 P.S. STOBBE, *Lineaire algebra, deel 2*, 1974. ISBN 90 6196 091 6.

MCS 18 F. VAN DER BLIJ et al., *Een kwart eeuw wiskunde, Syllabus van de Vakantiecursus 1971*, 1974. ISBN 90 6196 092 4.

MCS 19 A. HORDIJK et al., *Optimaal stoppen van Markovketens*, 1974. ISBN 90 6196 093 2.

MCS 20 T.M.T. COOLEN et al., *ALGOL 60 procedures voor begin- en randwaardeproblemen*, 1974. ISBN 90 6196 094 0.

MCS 21 J.W. DE BAKKER (Red.), *Colloquium Programmacorrectheid*, 1974. ISBN 90 6196 103 3.

*MCS 22 R. HELMERS et al., *Werkweek Statistiek*, 1973. ISBN 90 6196 104 1.

*MCS 23.1 J.W. DE ROEVER (Red.), *Colloquium Onderwerpen uit de Biomathematica, deel 1*, 1973. ISBN 90 6196 105 X.

*MCS 23.2 J.W. DE ROEVER (Red.), *Colloquium Onderwerpen uit de Biomathematica, deel 2*, 1973. ISBN 90 6196 115 7.

*MCS 24.1 P.J. VAN DER HOUWEN, *Numerieke integratie van differentiaalvergelijkingen, deel 1: Eenstapsmethoden*, 1974. ISBN 90 6196 106 8.

*MCS 25 L. GEURTS (Red.), *Colloquium Structuur van Programmeertalen*, 1974. ISBN 90 6196 116 5.

*MCS 26 N.M. TEMME (Red.), *Colloquium Niet-lineaire Analyse*, 1975. ISBN 90 6196 117 3.

De met een * gemerkte uitgaven moeten nog verschijnen.