stichting
mathematisch
centrum

MC

AFDELING INFORMATICA IN 13/77 APRIL

L. AMMERAAL

INLEIDING TOT PROGRAMMAVERIFICATIE

2e boerhaavestraat 49 amsterdam

Inleiding tot programmaverificatie *)

door

L. Ammeraal

SAMENVATTING

Eigenschappen van programma's kunnen worden geformuleerd en bewezen met behulp van bewijsregels. Voor de assignment statement en de conditional statement worden zowel voorwaartse als achterwaartse bewijsregels besproken, waarbij gebruik wordt gemaakt van eenvoudige voorbeelden. Bij de behandeling van de while-statement wordt gewezen op de kwestie van terminatie en op de representeerbaarheid van pre- en postcondities. Tenslotte komt automatische programmaverificatie ter sprake.

TREFWOORDEN: *correctness proof, program verification*

---

*) Tekst van een lezing gehouden op het voorjaarssymposium van het Nederlands Genootschap voor Informatica op 14 april 1977.

1. Het begrip "programmakorrektheid"

Er gaapt een kloof tussen de theoretische en de praktische informatica, of, zo men wil, tussen informatica en informatieverwerking. Ik wil een bescheiden bijdrage leveren tot het overbruggen van deze kloof. Dit heeft als consequentie dat mijn verhaal een theoretisch informaticus nauwelijks zal boeien omdat hij "niets nieuws hoort", terwijl anderzijds de praktijkman raar tegen deze "theoretische poespas" zal aankijken. Nu moet men in dit laatste opzicht een niet te bekrompen standpunt innemen. Een zekere hoeveelheid basiskennis moet eenvoudig aanwezig zijn om verder te kunnen. Zo leren bijvoorbeeld de meeste kinderen het verband tussen de zijden van een rechthoekige driehoek, ook al zullen zij misschien niet direct inzien dat je er buiten de school iets aan hebt. Zou het dan voor een programmeur "onrealistisch" zijn, zich bezig te houden met het verband tussen een beginvoorwaarde P, een programmadeel S en een eindvoorwaarde Q? Het volgende voorbeeld verduidelijkt deze vraag. Laten x en y twee programmavariabelen zijn die aanvankelijk dezelfde waarde hebben, d.w.z. voor P nemen we de voorwaarde x = y. Nu wordt programmadeel S uitgevoerd, waarvoor we kiezen: x := x + y; y := x + y. (In sommige programmeertalen schrijft men hiervoor: X = X + Y; Y = X + Y). Gevraagd wordt nu naar de voorwaarde (Q) die het verband tussen x en y na de uitvoering van S uitdrukt. We schrijven deze vraag als

$$\{x = y\} \quad x := x + y;\ y := x + y \quad \{?\}.$$

Dit probleem is inderdaad onrealistisch. Toch zou ik iemand die door beredeneren het juiste antwoord vindt, liever in dienst nemen voor onverschillig welk programmeerwerk, dan iemand van wie alleen bekend is dat

hij postzegels verzamelt. (Laatstgenoemd selektiekriterium heb ik werkelijk eens horen noemen door een alom gerespecteerd hoofd van een rekencentrum.)

Wat betekent "programmakorrektheid"? Men zegt wel eens dat een programma korrekt is als het doet wat we ervan verwachten. Hiermee komen we niet veel verder. Meer houvast biedt de volgende definitie, waarin P en Q voorwaarden zijn die worden gesteld aan de waarden van variabelen, en waarin S één of meer programmastatements voorstelt.

> We noemen S *korrekt met betrekking tot* P en Q,
> als het voldoen aan P vóór
> de uitvoering van S impliceert dat aan Q
> voldaan is na de uitvoering van S.

Notatie: {P} S {Q}.

Voorbeeld: Het programmadeel x := x + y; y := x + y is korrekt m.b.t. de voorwaarden x = y en 3x = 2y, dus

{x=y} x := x + y; y := x + y {3x=2y}.

We komen nog op dit voorbeeld terug.

Een subtiele kwestie bij {P} S {Q} is buiten beschouwing gebleven. Het betreft de vraag of het voldoen aan de beginvoorwaarde P garandeert dat S termineert, d.w.z. gedefinieerd is. Is deze garantie aanwezig, dan spreekt men van *totale* en anders van *partiële* korrektheid. We gaan hier nu niet op in.

In plaats van de wat arrogant klinkende term "korrektheidsbewijs" spreekt men tegenwoordig liever over de "verificatie" van een programma. Een streng wiskundig bewijs van een programma is in de praktijk teveel gevraagd. Het voornaamste is dat we de korrekte werking van ons programma *zo goed mogelijk* verifiëren alvorens het aan de machine toe te vertrouwen.

2. Bewijsregels voor de assignment statement

Het bekendst is de achterwaartse bewijsregel voor de assignment statement. Volgens deze regel, ook wel *axioma van Hoare* [1] genoemd, behoort het vraagteken in

$$\{?\}\quad x := f\quad \{Q\}$$

vervangen te worden door de voorwaarde die men verkrijgt door in Q de uitdrukking f te substitueren voor x. Passen we dit toe op

$$\{?\}\quad x := x + y\quad \{2y=x\}$$

dan krijgen we als beginvoorwaarde $2y = x + y$, oftewel $y = x$. We zien de juistheid hiervan als volgt in: De eindvoorwaarde $2y = x$ betekent dat $2y$, na uitvoering van $x := x + y$, gelijk is aan een of ander (vast) getal a en dat dit getal a op dat moment de waarde van x is. De waarde van y verandert niet door $x := x + y$; dus vooraf is eveneens $2y = a$ waar. Omdat x de waarde a krijgt, moet vóór de uitvoering van $x := x + y$ de som $x + y$ gelijk zijn aan a. Dus geldt vooraf $2y = x + y$, q.e.d.

Het axioma van Hoare is eenvoudig in het gebruik; men kan er gemakkelijk

$$\{x=y\}\quad x := x + y;\ y := x + y\quad \{3x=2y\}$$

mee verifieren. Helaas geeft het geen antwoord op onze oorspronkelijke vraagstelling, die we formuleerden als

$$\{x=y\}\quad x := x + y;\ y := x + y\quad \{?\}$$

Hoe komen we nu aan de eindvoorwaarde $3x = 2y$? Men komt in de verleiding gewoon eens wat te proberen en te kijken wat er uit komt, b.v.

beginvoorwaarde	eindvoorwaarde
x = y = 3	x = 6, y = 9
x = y = 4	x = 8, y = 12
x = y = 5	x = 10, y = 15

Het is onjuist hieruit te concluderen dat P ≡ x = y steeds leidt tot Q ≡ 3x = 2y. Men zou Q ≡ 3x = 2y hoogstens een vermoeden kunnen noemen, dat nog met het axioma van Hoare geverifieerd dient te worden.
Veel mooier is het, het rijtje voorwaarden x = y = 3, x = y = 4, x = y = 5 die we voor P hebben ingevuld, te generaliseren tot

$$x = x_0,\ y = y_0,\ x_0 = y_0,$$

waarbij x_0 en y_0 geen programmavariabelen zijn, maar gewoon vaste getallen voorstellen zoals in de wiskunde gebruikelijk is. We bedoelen hiermee, dat x_0 en y_0 niet ten gevolge van de uitvoering van een programma van waarde kunnen veranderen. Dus $x_0 = y_0$ geldt ook na de uitvoering van x := x + y; y := x + y. Bovendien gaat men gemakkelijk na dat erna geldt $x = x_0 + y_0$ en $y = (x_0+y_0) + y_0$.
Uit

$$\begin{cases} x_0 = y_0 \\ x = x_0 + y_0 \\ y = (x_0+y_0) + y_0 \end{cases}$$

kan men x_0 en y_0 elimineren, waarna we inderdaad 3x = 2y vinden. Deze methode heet *symbolische executie*. In tegenstelling tot het axioma van Hoare werken we bij symbolische executie voorwaarts; we voeren a.h.w. de programmastatements uit met symbolen in plaats van met getallen. Deze bijzonder elegante en natuurlijke werkwijze is terug te voeren tot een *axioma van Floyd* [2]; we schrijven dit als

$$\{P\}\ \ x := f\ \ \{(\exists x_0)(P' \wedge x = f')\},$$

waarbij P' en f' uit P resp. f worden verkregen door hierin x_0 voor x te substitueren. We lezen genoemde eindvoorwaarde als: Er is een (oude waarde van x die we voorstellen door) x_0, zodanig dat P' èn x = f' waar zijn.

We zijn er stilzwijgend aan voorbij gegaan dat de axioma's van Hoare en Floyd de *zwakste* beginvoorwaarde, resp. *sterkste* eindvoorwaarde leveren. Deze begrippen zijn wel van belang omdat, strikt genomen, het antwoord op b.v. de vraag

"Wat geldt *na* de uitwerking van x := x + y als *ervóór* geldt x = y?"

ook b.v. x = x zou kunnen zijn. Dit is uiteraard niet de bedoeling. Onze schrijfwijze

$$\{P\} \quad S \quad \{?\}$$

moet daarom worden opgevat als de vraag

"Wat is de sterkste eindvoorwaarde behorende bij de beginvoorwaarde P en de statement S?"

Evenzo lezen we

$$\{?\} \quad S \quad \{Q\}$$

als

"Wat is de zwakste beginvoorwaarde behorende bij de statement S en de eindvoorwaarde Q?"

(Let erop dat deze beginvoorwaarde wel moet garanderen dat na de uitvoering van S aan Q is voldaan!)

3. De conditionele statement

We proberen nu antwoord te geven op

$$\{y>0\} \quad \underline{if}\ x > 1\ \underline{then}\ x := x + 1\ \underline{else}\ y := 0\ \underline{fi} \quad \{?\}.$$

We doen dit door twee gevallen te onderscheiden:

Geval A: Aanvankelijk is $x > 1$.

Dan geldt na afloop: $y > 0 \wedge x > 2$.

Geval B: Aanvankelijk is $x \leq 1$.

Dan geldt na afloop: $y = 0 \wedge x \leq 1$.

Het bedoelde antwoord luidt daarom:

$(y>0 \wedge x>2) \vee (y=0 \wedge x\leq 1)$.

(Lees $\wedge$ als "en" en $\vee$ als "of").

Meer algemeen gesproken luidt de *voorwaartse regel voor de conditionele statement*:

De gevraagde sterkste eindvoorwaarde in

$$\{P\}\ \underline{\textit{if}}\ B\ \underline{\textit{then}}\ S\ \underline{\textit{else}}\ T\ \underline{\textit{fi}}\ \{?\}$$

is $Q_1 \vee Q_2$, waarin Q_1 en Q_2 de oplossingen zijn van

$$\{P\wedge B\}\ S\ \{?\},\ \text{resp.}\ \{P\wedge\neg B\}\ T\ \{?\}.$$

Ook van de achterwaartse regel geven we eerst een voorbeeld. Gevraagd wordt het antwoord van

$$\{?\}\ \underline{\textit{if}}\ x = y\ \underline{\textit{then}}\ x := 0\ \underline{\textit{else}}\ y := 0\ \underline{\textit{fi}}\ \{x=y\}.$$

We weten dat ôf $x := 0$, ôf $y := 0$ is uitgevoerd als aan de eindvoorwaarde $x = y$ is voldaan. Volgens het axioma van Hoare gold in het eerste geval aanvankelijk $0 = y$ (verkregen door 0 voor x te substitueren in $x = y$) en in het tweede geval $x = 0$ (verkregen door 0 voor y te substitueren in $x = y$). I.v.m. hetgeen tussen *if* en *then* staat, moet tevens in het eerste geval $x = y$ en in het tweede geval $x \neq y$ gegolden hebben. We vinden daarom voor de gevraagde beginvoorwaarde:

$$(0=y \wedge x=y) \vee (x=0 \wedge x\neq y),$$

wat vereenvoudigd kan worden geschreven als

$$(x=0 \wedge x=y) \vee (x=0 \wedge x\neq y),\ \text{oftewel}\ \underline{\underline{x = 0}}.$$

(Blijkbaar moeten we aan x de bijzonder strenge eis $x = 0$ stellen, terwijl y volkomen willekeurig gekozen kan worden. Deze beginvoorwaarde is verrassend, i.v.m. de ogenschijnlijke symmetrie van het gegeven.)

We geven nu ook de *achterwaartse regel voor de conditionele statement* in zijn algemene gedaante:

De gevraagde zwakste beginvoorwaarde in

$$\{?\} \quad \textit{if}\ B\ \textit{then}\ S\ \textit{else}\ T\ \textit{fi} \quad \{Q\}$$

is $(P_1 \wedge B) \vee (P_2 \wedge \neg B)$, waarin P_1 en P_2 de oplossingen zijn van

$$\{?\}\ S\ \{Q\}, \text{ resp. } \{?\}\ T\ \{Q\}.$$

4. De while-statement

We behandelen nu enige verificatie-aspecten van de while-statement. Deze heeft de gedaante:

$$\textit{while}\ B\ \textit{do}\ S\ \textit{od}.$$

De statement S wordt slechts uitgevoerd zolang vooraf aan de voorwaarde B is voldaan. De volgende bewijsregel is algemeen bekend:
Als

$$\{P \wedge B\}\ S\ \{P\},$$

dan geldt

$$\{P\}\ \textit{while}\ B\ \textit{do}\ S\ \textit{od}\ \{P \wedge \neg B\}.$$

Men noemt P een *invariante voorwaarde*. In de literatuur zijn voorbeelden van het gebruik van deze bewijsregel vaak simplistisch van aard. We geven zo'n voorbeeld. Gegeven zijn de gehele getallen $a \geq 0$ en $b > 0$. Gevraagd wordt (zonder te delen) het quotient q en de rest r bij deling van a door b te bepalen. We zoeken dus q en r zodanig dat $a = qb + r$ en $0 \leq r < b$. Aan $a = qb + r \wedge r \geq 0$ is voldaan als $q = 0$ en $r = a$. Bovendien blijft $a = qb + r$ gelden als q met 1 wordt verhoogd en r met b wordt verlaagd. Ook $r \geq 0$ blijft gelden als, vóór deze verlaging van b, voldaan is aan $r \geq b$. Zo onstaat het programma

```
q := 0; r := a;
while r ≥ b do q := q + 1; r := r - b od
```

Bij gebruik van de genoemde bewijsregel kiezen we $a = qb + r \wedge r \geq 0$ voor P en $r \geq b$ voor B.

Wij krijgen dan:
Als

$$\{a=qb+r \wedge r\geq 0 \wedge r\geq b\} \quad S \quad \{a=qb+r \wedge r\geq 0\},$$

dan geldt

$$\{a=qb+r \wedge r\geq 0\} \quad \textit{while}\ r \geq b\ \textit{do}\ S\ \textit{od} \quad \{a=qb+r \wedge r\geq 0 \wedge r<b\}.$$

(Voor S moet uiteraard q := q + 1; r := r - b worden gelezen).

Omdat toepassing van b.v. de regel van Hoare voor de assignment laat zien dat

$$\{a=qb \wedge r\geq b\} \quad q := q + 1;\ r := r - b \quad \{a=qb+r \wedge r\geq 0\}$$

geldt en omdat vóór de uitvoering van de while-statement voldaan is aan $a = qb + r \wedge r \geq 0$ mogen we concluderen dat na de while-statement q en r inderdaad voldoen aan $a = qb + r$ èn $0 \leq r < b$.

In verhandelingen over dit onderwerp wordt vaak de indruk gewekt dat deze bewijsmethode ook met success op "realistische" programma's kan worden toegepast. De praktijkman heeft hier doorgaans niet veel vertrouwen in en vreest dat het geheel veel te ingewikkeld wordt. Ik wil nu proberen duidelijk te maken dat je na de behandeling van zo'n eenvoudig voorbeeld inderdaad niet kunt zeggen: "Voor andere programma's gaat het net zo". Het zal blijken dat het niet alleen "te ingewikkeld" wordt, maar dat er meer aan vast zit. In de eerste plaats ontbrak zelfs in het eenvoudige voorbeeld een essentieel element, namelijk de vraag naar *terminatie*. In dit voorbeeld

is gemakkelijk na te gaan dat de uitvoering van de while-statement in eindige tijd voltooid is, maar zo'n onderzoek hoort er echt bij. De gegeven bewijsregel drukt *partiële* en geen *totale* korrektheid uit. Voor de meeste praktische programma's zal de terminatie wel te bewijzen zijn; toch dient men te weten dat het principieel onmogelijk is, op algorithmische wijze voor elk willekeurig programma vast te stellen of de uitvoering ervan ooit eindigt. We gaan hier niet verder in op dit z.g. "halting problem", waaraan in de meeste elementaire boeken over theoretische informatica ruime aandacht wordt geschonken. Er zijn ook programma's waarvan we niet weten of hun uitvoering ooit eindigt, omdat ze betrekking hebben op een onopgelost wiskundig probleem. Als voorbeeld noem ik het volgende probleem waaraan de naam van Fermat verbonden is:
Zijn er positieve gehele getallen n, x, y en z zodanig dat $x^n + y^n = z^n$ voor $n \geq 3$?
Men vermoedt dat dit niet zo is, maar kan dit vermoeden (of het tegendeel) tot op heden niet bewijzen. Als gevolg hiervan weten we niet of het volgende programma termineert.

```
n := 3; x := y := 1; z := 2;
while x ↑ n + y ↑ n ≠ z ↑ n
do if x < y then x := x + 1
   elif y + 1 < z then y := y + 1; x := 1
   elif n > 3 then n := n - 1; z := z + 1; x := y := 1
   else n := z + 2; x := y := 1; z := 2
   fi
od
```

(In dit programma kan men *elif* lezen als *else if*; verder veronderstellen we multipele-lengte-aritmetiek waarbij geen capaciteitsoverschrijding kan optreden).

(Het is overigens een aardige opgave, te verifieren dat alle geordende viertallen (n,x,y,z) met $n \geq 3$ en $1 \leq x \leq y < z$ door dit programma worden gegenereerd, als het inderdaad niet termineert).

Een andere essentiële moeilijkheid, waaraan zeker niet voorbijgegaan mag worden, betreft de *representeerbaarheid* van de voorwaarden (ook wel "asserties" genoemd) die men in de programmatekst {tussen akkoladen} zou willen plaatsen. Het is in de logica en bij het programmeren een goed gebruik, vooraf af te spreken in welke taal we onze uitdrukkingen zullen weergeven. We zouden bijvoorbeeld kunnen afspreken, dat asserties worden uitgedrukt in uitsluitend de volgende symbolen (uiteraard op syntactische korrekte wijze gegroepeerd):

- programmavariabelen, b.v. x, y,
- constanten, b.v. 3, -2.5,
- de rekenkundige operatoren +, −, *,
- de relationele operatoren =, ≠, <, ≤, >, ≥,
- de logische connectieven ∧, ∨, ¬, ≡, ⊃,
- haakjes,
- de kwantoren ∀ ("voor alle") en ∃ ("er bestaat een").

Kunnen we in deze logische taal elke voorwaarde representeren die we willen? De vraag is bijvoorbeeld of we in deze assertietaal de sterkste eindvoorwaarde behorende bij

{x = 1 ∧ i = 1 ∧ n ≥ 0} *while* i ≤ n *do* x := 2 * x; i := i + 1 *od* {?}

kunnen uitdrukken. (Merk op dat $x = 2^n \wedge n \geq 0 \wedge i > n$ volgens onze afspraak geen geldige assertie is omdat onze assertietaal geen machtsverheffen kent). Voor wie het interesseert zij vermeld dat in de gekozen assertietaal inderdaad een uitdrukking ekwivalent met $x = 2^n \wedge n \geq 0 \wedge i > n$ bestaat; we moeten dan gebruik maken van de β-funktie van Gödel en krijgen een formule die zo ingewikkeld is dat we er voor praktisch gebruik niets aan hebben. Misschien meent iemand dat we uit alle narigheden zijn als we enkele operatoren (b.v. machtsverheffen) aan onze assertietaal toevoegen. Deze mening is te optimistisch. Voeg bijvoorbeeld eerst enkele operatoren toe, en probeer vervolgens de sterkste eindvoorwaarde, of, wat even moeilijk is, een geschikte invariante voorwaarde, behorende bij het volgende voorbeeld in deze taal uit te drukken:

{x = 1 ∧ i = 1 ∧ n ≥ 0} *while* i ≤ n *do* x:= x * x + 1; i := i + 1 *od* {?}.

We leren uit deze beschouwing dat we moeten oppassen voor een te roos-kleurige kijk op programmaverificatie. Niet alleen bij grote, realistische, maar ook bij kleine, onrealistische programma's blijkt het beschikbare verificatiegereedschap ontoereikend.

5. Programmaverificatie: met de hand of machinaal?

Het is stellig niet de bedoeling van mijn laatste opmerking het potentiële profijt van programmaverificatie in twijfel te trekken. Het analyseren van programmatekst, liefst tijdens (of vóór?) het ontstaan ervan, dient zonder meer te worden aangemoedigd. Ook al kunnen we geen formeel bewijs van een programma opschrijven en al blijft grondig testen harde noodzaak, toch leert de ervaring dat zelfs een bescheiden poging het programma vóór het testen te verifiëren al vruchten afwerpt. Hierbij zijn alle wettige middelen geoorloofd, zelfs het inschakelen van een machine! Er verschijnen nog al wat publicaties over *automatische* programmaverificatie. Deze publikaties vallen op door hun optimistiche toon en door hun povere resultaten. Automatische programmaverificatie is nog geen middel tegen welke kwaal dan ook, maar eerder een slecht verteerbaar laboratorium-brouwsel. Toch is er wel degelijk hoop op langere termijn. We hebben gezien dat bewijsregels voor de assignment statement en voor de conditionele statement een rechttoe-rechtaan-karakter hebben. Hun implementatie is een kwestie van niet al te moeilijke *formulemanipulatie*. Verder dient erop te worden gewezen dat er al een zekere hulp bij verificatie geboden wordt door compilers voor goed ontworpen programmeertalen. Essentieel bij programma-verificatie is immers dat *statische* controle wordt verricht op programma-eigenschappen die we geneigd zijn een *dynamisch* karakter toe te dragen. Met andere woorden: Stel niet uit tot "run-time" wat je "at compile-time" kunt doen. Als voorbeeld noem ik het gebruik van identifiers voor constanten, een faciliteit die wel ALGOL 68 en PASCAL bieden maar niet bestaat in ALGOL 60, FORTRAN of PL/1. Zo'n identifier links van het :=-teken is syntactisch fout omdat hij geen variabele, maar een constante voorstelt. Een ander voorbeeld is het syntactisch verwerpen van niet-geheeltallige

waarden aan integere variabelen. Een goede ALGOL 68-compiler zal beide assignment statements diskwalificeren in:

int c = 1000; *int* v; c := 1; v := 3.25;

Wij zien hier dat *veiligheid* wordt betaald met overspecificatie oftewel *redundantie*. Denk b.v. ook aan controlecijfers in codes; een ander voorbeeld is het controleponsen.
Ik verwacht dat er talen en vertalers zullen komen waarbij het veiligheidsaspect nog meer op de voorgrond treedt. Er zullen compilers met verificatiefaciliteiten worden ontwikkeld. De gebruiker krijgt de verificatie van zijn programma niet geheel cadeau. Hij zal zelf òf controle-informatie in de vorm van asserties moeten verstrekken, òf moeten nagaan of de gegenereerde asserties met zijn bedoelingen overeenstemmen.

REFERENTIES

1. HOARE, C.A.R., *An Axiomatic Basis of Computer Programming*, CACM, Vol.12, No.10 (October 1969), 576-580.

2. FLOYD, R.W., *Assigning Meanings to Programs*, Proc. Symp. Appl. Math. 19, American Math. Soc. (1967) 19-32.

3. BAKKER, J.W. DE (red.), *Colloquium programmacorrectheid*, MC Syllabus 21, Mathematisch Centrum (1975).

4. GRIES, D., *An Illustration of Current Ideas on the Derivation of Correctness Proofs and Correct Programs*, IEEE Transactions on Software Engineering, Vol. SE-2, No. 4 (December 1976).

5. IGARASHI, S., R.L. LONDON & D.C. LUCKHAM, *Automatic Program Verification I: A Logical Basis and its Implementation*, Acta Informatica 4, 145-182 (1975).

6. HANTLER, S.L., J.C. KING, *An introduction to proving the correctness of programs*, IBM Research, RC 5893 (# 25476) (1976).