Implementeren is Programmeren

0 Inleiding

Het *implementeren* van een programma is het formuleren van dat programma met een zo fijne mate van detail dat het resultaat direkt door een computer kan worden uitgevoerd. Dit formuleren wordt ook wel *coderen* genoemd. In ruimere zin wordt onder implementeren verstaan het zo nauwkeurig formuleren van een algoritme dat het in een (gekozen) programmeertaal kan worden opgeschreven. In deze studie gebruiken we alleen de strengere betekenis, waarbij men "machinetaal" in plaats van "programmeertaal" kan lezen: implementeren is dan het formuleren van een programma in machinetaal.

Als uitgangspunt voor een implementatie dient altijd een programma dat is geformuleerd in een programmeertaal zoals Pascal, Guarded Commands, of (desnoods[0]) C. Er zijn twee redenen om het te implementeren programma eerst volledig in programmeertaal te formuleren, een goede en een zeer goede.

De goede reden is dat de vertaling van programmertaal naar machinetaal geheel mechanisch, door de computer zelf, kan worden uitgevoerd, met behulp van een voor dit doel geconstrueerd programma: een zogenaamde *compiler*. De studie van implementaties is dan ook vooral van belang voor ontwerpers van compilers.

De zeer goede reden is dat het, zelfs voor kleine problemen, onbegonnen werk is het programma meteen in machinetaal op te schrijven: de sprong van probleemoplossing naar machinecode is veel te groot om in één keer te kunnen maken. Om het ontwerp hanteerbaar, overzichtelijk en controleerbaar te houden is het beter het programma eerst in een "hogere" –van machinedetails vrije– programmeertaal te formuleren en daarbij de eigenschappen van de machine te negeren, en vervolgens het programma in de machinetaal te implementeren, waarbij het op te lossen probleem geen rol meer speelt. Het programma vormt in feite zo de *specificatie* voor het te maken machinetaalprogramma.

Dit principe heet *Separation of Concerns* en het is de enige manier om alle behalve de meest triviale ontwerpen hanteerbaar te houden. Iedere goede ontwerper past het principe (bewust of onbewust) toe: geen architect ontwerpt

[0] Eigenlijk is C geen programmertaal, maar een C-programma is een beter uitgangspunt dan geen programma.

huizen door te denken aan de kleikorreltjes waarvan stenen worden gebakken.

Het motto bij dit vak is *Implementeren is Programmeren*, waarmee twee zaken tot uitdrukking worden gebracht:

- Het implementeren van een programma is een "gewone" programmeeraktiviteit en hiervoor *kunnen* alle bekende "gewone" programmeertechnieken worden gebruikt.
- Om systematisch te kunnen implementeren en om vertrouwen te kunnen hebben in het resultaat *moeten* hiervoor ook programmeertechnieken worden gebruikt: wie niet kan programmeren kan ook niet implementeren.

Het moge duidelijk zijn dat de studie van implementaties zonder enige kennis en beheersing van programmeertechnieken zinloos is.

Het implementeren van een programma is vooral een proces van systematische *programmatransformaties*. Door middel van een programmatransformatie wordt een gegeven programma omgevormd tot een *gelijkwaardig* nieuw programma; gelijkwaardig betekent hier dat het nieuwe programma hetzelfde probleem oplost als het oude: het nieuwe programma "doet hetzelfde". Bij implementaties worden programmatransformaties op twee manieren gebruikt. Ten eerste moeten de variabelen uit het programma worden voorgesteld door de registers en geheugenplaatsen van de machine –dit heet *gegevensrepresentatie*– en moeten de bewerkingen op deze variabelen worden vertaald in gelijkwaardige bewerkingen op de registers en geheugenplaatsen. Ten tweede moeten bewerkingen in het programma die niet direkt door de machine kunnen worden uitgevoerd worden *verfijnd* tot stukjes programma die uit eenvoudigere, wel beschikbare bewerkingen zijn opgebouwd. Als de machine alleen kan optellen en aftrekken moet vermenigvuldiging, bijvoorbeeld, worden geïmplementeerd met behulp van alleen optellen en aftrekken.

voorbeeld 0: We beschouwen een (zogenaamde) multiple assignment, zoals in Guarded Commands kan worden opgeschreven: $x, y := y, x$. Het effekt hiervan is dat de waarden van de variabelen x en y worden verwisseld; dit kan worden weergegeven door middel van de volgende annotatie, waarin X en Y namen zijn voor de beginwaarden van x en y:

$\{\ x = X \wedge y = Y\ \}$
$x, y := y, x$
$\{\ y = X \wedge x = Y\ \}$

Dit kan worden getransformeerd tot een gelijkwaardig stukje programma waarin alleen enkelvoudige assignments voorkomen; de gelijkwaardigheid blijkt uit de asserties. Hierbij wordt gebruik gemaakt van een extra variabele, h:

$\{\ x = X \wedge y = Y\ \}$
$h := x$
; $\{\ h = X \wedge y = Y\ \}$
$x := y$
; $\{\ h = X \wedge x = Y\ \}$
$y := h$
$\{\ y = X \wedge x = Y\ \}$

We hebben nu de multiple assignment $x, y := y, x$ geïmplementeerd met behulp van enkelvoudige assignments; dit is een programmatransformatie.

□

opgave 0: Waarom is $x := y\ ;\ y := x$ geen goede implementatie van $x, y := y, x$?

□

voorbeeld 1: We beschouwen nu de gegevensrepresentatie van het vorige programma. Het geheugen van de machine geven we weer als een array-variabele S; de elementen van dit array heten *woorden* en (de inhoud van) het geheugenwoord met adres a geven we aan met $S{\cdot}a$ [1]. Verder nemen we aan dat de machine naast het geheugen ook registers heeft voor de opslag van gegevens. De registers geven we aan met $A, B, C, \cdots$. Als we nu besluiten dat programmavariabele x in het geheugen met het woord met adres 47 zal overeenkomen, en y met het woord met adres 103, en als we tenslotte besluiten dat de extra variabele h door register B zal worden voorgesteld, dan kunnen we al deze beslissingen samenvatten in een (zogenaamde) *representatie-invariant*:

$x = S{\cdot}47 \ \wedge \ y = S{\cdot}103 \ \wedge \ h = B \quad .$

[1] In Pascal of C zouden we $S[a]$ schrijven.

De representatie-invariant legt dus het verband vast tussen de variabelen uit het te implementeren programma en de variabelen die in de implementatie beschikbaar zijn; in ons geval zijn dat het geheugen en de registers. De representatie-invariant vormt de handleiding voor de programmatransformatie: hierin lezen we, bijvoorbeeld, dat alle bewerkingen op variabele x moeten worden vervangen door gelijkwaardige bewerkingen op variabele $S{\cdot}47$. Als we dit zorgvuldig en consequent uitvoeren, gaat ons programma er zó uitzien; omdat de representatie-invariant *overal* geldt nemen we deze, om het sober te houden, *nergens* in de annotatie van het programma op:

$$
\begin{array}{ll}
 & \{\ x{=}X \wedge y{=}Y\ \} \\
 & B := S{\cdot}47 \\
;\ & \{\ h{=}X \wedge y{=}Y\ \} \\
 & S{\cdot}47 := S{\cdot}103 \\
;\ & \{\ h{=}X \wedge x{=}Y\ \} \\
 & S{\cdot}103 := B \\
 & \{\ y{=}X \wedge x{=}Y\ \}
\end{array}
$$

□

voorbeeld 2: Er zijn machines waarbij de drie assignment statements in het laatste programma direkt kunnen worden gecodeerd als instructies in de machinetaal. Als onze machine zo is zijn we klaar. Er zijn echter ook machines met de eigenschap dat elke instructie ten hoogste één geheugenadres[2] kan bevatten. Als dit zo is kan de assignment $S{\cdot}47 := S{\cdot}103$ niet als één instructie worden gecodeerd. Daarom moet deze assignment nog verder worden verfijnd, namelijk tot assignments waarin maximaal één referentie aan het geheugen voorkomt. Dat lukt natuurlijk alleen maar als we een register gebruiken; omdat de waarde van variabele h niet verloren mag gaan en deze volgens de representatie-invariant in register B staat moeten we hiervoor een *ander* register gebruiken, zeg A:

$$
\begin{array}{ll}
 & \{\ h{=}X \wedge y{=}Y\ \} \\
 & A := S{\cdot}103 \\
;\ & \{\ h{=}X \wedge A{=}Y\ \} \\
 & S{\cdot}47 := A \\
 & \{\ h{=}X \wedge x{=}Y\ \}
\end{array}
$$

[2] zogenaamde één-adres-machines.

□

Deze voorbeelden laten zien dat het nodig kan zijn meerdere programmatransformaties uit te voeren om tot een bruikbare implementatie te komen. Deze transformaties kunnen vaak onafhankelijk van elkaar plaats vinden en er is in het algemeen geen "juiste" volgorde voor aan te geven; een goede vuistregel is wel: *hoofdzaken eerst, details later.* De transformatie uit voorbeeld 2 behelst de implementatie van bewerkingen van de vorm $S{\cdot}a := S{\cdot}b$; de transformatie hiervan tot $A := S{\cdot}b \ ; \ S{\cdot}a := A$ kan geheel in isolement worden bestudeerd: dit is een detail-implementatie.

samenvatting:

Het implementeren van een programma komt neer op:

- De keuze van een *gegevensrepresentatie*, waarbij de relatie tussen de programmavariabelen en geheugen en registers van de machine wordt vastgelegd met behulp van een *represenatie-invariant*.
- *Programmatransformaties*, waarmee de bewerkingen op de programmavariabelen worden vertaald in gelijkwaardige bewerkingen op geheugenwoorden en registers, en waarmee te ingewikkelde bewerkingen worden verfijnd tot eenvoudigere.

□

opgave 1: Ontwerp een implementatie van $x, y := y, x$ voor een machine die naast het geheugen over slechts één register, genaamd A, beschikt.

□

1 De Von Neumann machine

De (klassieke) Von Neumann machine is een praktische uitvoering van de door Alan M. Turing uitgevonden Universele Turing Machine. Dit is een machine die alle mogelijke berekeningen kan uitvoeren die maar door een machine kunnen worden uitgevoerd: daarom heet de machine "universeel". Dit vereist natuurlijk wel dat bij elk gebruik van de machine wordt gespecificeerd welke berekening deze keer zal worden uitgevoerd: de machine moet worden voorzien van een programma. Ook de Von Neumann machine is een universele machine; elk programma kan in instructies van de machine worden gecodeerd en kan, als het geheugen maar groot genoeg is, door de machine worden uitgevoerd.

Een belangrijk kenmerk van de Von Neumann machine is haar *eenvoud*, die een direkt gevolg is van de stand van de techniek in de tijd –ca. 1945– waarin de machine werd ontwikkeld: het bouwen van een betrouwbaar werkende machine was in die tijd een grote opgave en alleen van het eenvoudigst mogelijke ontwerp was praktische realisatie haalbaar.

Vanwege deze eenvoud leent de klassieke Von Neumann machine zich uitstekend voor de studie van implementaties; bovendien blijkt deze machine enige tekortkomingen te hebben –zij is tè eenvoudig– die de behoefte aan verfijndere mechanismen duidelijk maken. Bijna alle tegenwoordige computers zijn, soms zeer uitgebreide en gecompliceerde[3], varianten van de klassieke Von Neumann machine.

1.0 De opbouw van de machine

De belangrijkste twee onderdelen van de Von Neumann machine zijn het *geheugen*, waarin het programma en de te bewerken gegevens worden opgeslagen, en de (centrale) *processor*, die het programma uitvoert. Daarnaast heeft iedere machine voorzieningen om programma's en gegevens in de machine te brengen en om resultaten van berekeningen weer naar buiten te brengen. Omdat deze voorzieningen van machine tot machine kunnen verschillen laten wij ze verder buiten beschouwing. Alleen al bij een personal computer, bijvoorbeeld, omvatten de in- en uitvoer voorzieningen ten minste een toetsenbord, beeldscherm en diskettestation, en verder, al naar gelang de behoeften van de gebruiker, een "muis", joystick, printer, telefoonaansluiting (in de vorm van een *modem*), netwerkaansluiting, tapestreamer (voor zogenaamde *backups* van het geheugen), CD-ROM speler, paginascanner, microfoon, luidsprekers, of zelfs een televisieontvanger. Deze grote variatie in mogelijke *randapparatuur* weerspiegelt de grote variatie in gebruiksmogelijkheden van de machine; de computer is met recht een universele rekenmachine, waarbij "rekenen" ruim kan worden opgevat: rekenen omvat alles wat in een formeel symbolenspel kan worden gevangen.

* * *

Het geheugen is een in beginsel oneindige, maar in de praktijk altijd eindige, verzameling *woorden*. Ieder woord kan een waarde bevatten, waarbij de verzameling mogelijke waarden die een woord kan aannemen eindig is. We nemen

[3]Het manual van de Motorola MC68030 microprocessor beslaat, bijvoorbeeld, meer dan 600 pagina's.

echter aan dat het aantal mogelijke waarden groot genoeg is; later zullen we aangeven hoe groot "groot genoeg" is.

Alle woorden hebben dezelfde eigenschappen. In het bijzonder zijn alle woorden even goed en even snel toegankelijk en hebben ze alle dezelfde opslagcapaciteit: het geheugen is *homogeen* en we spreken ook wel van een *random access* geheugen[4].

Om de woorden van elkaar te kunnen onderscheiden zijn ze opeenvolgend genummerd, vanaf 0. Deze nummers worden traditioneel *adressen* genoemd: een adres is een getal waarmee een geheugenwoord correspondeert; we zeggen ook wel dat een adres een woord *identificeert*. Door het gebruik van opeenvolgende nummers zijn de woorden tevens geordend: het geheugen is ook *lineair*. Als a het adres van een woord is kunnen we, bijvoorbeeld, ook spreken over de woorden met adressen $a+1\,, a+2\,, a+3$, enzovoort.

De processor kan de inhoud van ieder woord inspecteren –lezen– en kan de inhoud van ieder woord vervangen –schrijven–; het woord is de eenheid van adresseerbaarheid. Het geheugen kan aldus worden beschouwd als een *array* van woorden. In het vervolg beschrijven we het geheugen daarom met behulp van een variabele S met de volgende declaratie:

$$S \;:\; \mathsf{array}\;[0..]\;\mathsf{of}\;\mathit{word} \quad .$$

Hier is geen bovengrens voor de afmeting van het geheugen aangegeven: omdat deze bovengrens van machine tot machine een andere waarde kan hebben, nemen we hier aan dat het geheugen onbegrensd groot is.

Het woord met adres a noteren we als $S{\cdot}a$, voor elk mogelijk adres a. Het woord met adres 47, bijvoorbeeld, is dus $S{\cdot}47$ en het woord met adres $a+1$ is $S{\cdot}(a{+}1)$.

* * *

De processor voert *sequentieel*, dat wil zeggen: de één na de ander, instructies uit. Deze instructies staan in opeenvolgende woorden in het geheugen. Om de volgende uit te voeren instructie te identificeren bevat de processor een register, PI –voor ProgrammaIndex–, waarvan de waarde een adres is; dit is het adres van de volgende uit te voeren instructie.

Verder beschikt de processor over een klein aantal, één of meer, *rekenregisters* waarin, net als in geheugenwoorden, waarden kunnen opgeslagen. In het eenvoudigste model hebben de rekenregisters precies dezelfde eigenschappen als

[4] Hier komt de term RAM vandaan, die voor *Random Access Memory* staat.

de geheugenwoorden. Zij verschillen alleen van de geheugenwoorden doordat zij niet in het geheugen maar in de processor zijn ondergebracht, wat voor de gebruiker van de machine trouwens niet zichtbaar is, en doordat zij niet door adressen maar door expliciete *namen* worden onderscheiden. Hier zullen wij $A, B, C, \cdots$ als namen voor de rekenregisters gebruiken. De klassieke Von Neumann machine had slechts één rekenregister, de *accumulator* genaamd en aangeduid met de letter A.

Veronderstel dat we een machine die bezig is met de uitvoering van een programma midden in die berekening stopzetten en uitschakelen. De *toestand* van de machine omvat nu al die gegevens die we moeten bewaren om deze berekening op een later tijdstip te kunnen voortzetten precies vanaf het punt van stopzetten, *alsof de onderbreking nooit had plaatsgevonden*. De toestand van de machine bestaat uit:

- de inhoud van het gehele geheugen, S, èn
- de inhoud van alle rekenregisters, $A, B, C, \cdots$, èn
- de inhoud van de programmaindex, PI.

Iedere instructie in het repertoire van de machine beschrijft nu een *toestandsverandering* en bij het uitvoeren van een instructie vindt deze toestandsverandering plaats. De werking van de processor kan nu eenvoudig worden beschreven met behulp van het volgende programmaatje:

```
do true  →  |[ var IR : word;
               IR, PI := S·PI, PI+1
            ;  "execute the instruction in IR"
            ]|
od   .
```

Hierin is IR –voor InstructieRegister– een hulpregister van de processor voor het tijdelijk opslaan van de uit te voeren instructie; IR maakt geen deel uit van de toestand van de machine: na het uitvoeren van een instructie is de waarde van IR niet meer van belang. Bij de aktie "execute the instruction in IR" ondergaat de machine de toestandsverandering die met deze instructie overeenkomt.

De werking van een processor wordt nu bepaald door welke instructies deze processor kan uitvoeren en welke toestandsveranderingen hierdoor mogelijk

zijn. De handleiding van een processor bestaat dan ook vooral uit een beschrijving van al deze instructies en hun toestandsveranderingen.

De klassieke Von Neumann machine had een zeer eenvoudig en beperkt instructierepertoire. Iedere instructie bestond hierbij uit twee delen, een *operator* en een *adres*. De operator specificeert de gewenste toestandsverandering; hierbij is altijd precies één geheugenwoord betrokken, waarvan het adres wordt aangegeven in het adresdeel van de instructie. Omdat de machine slechts één rekenregister heeft en dit register bij vrijwel alle instructies een rol speelt, hoeft het gebruik van het rekenregister niet in de instructies te worden gecodeerd; bij machines met meer registers moet natuurlijk wel in de instructie worden aangegeven welk register een rol speelt.

De volgende tabel bevat een aantal mogelijke instructies met de daarbij behorende toestandsveranderingen; deze zijn voorgesteld als assignments aan de variabelen die de toestand van de machine uitmaken, namelijk S, A, en PI. Hierbij staat a steeds voor het adresdeel van de instructie terwijl de operatoren zijn aangegeven met namen – load, store, etc. –:

instructie:		toestandsverandering:
load	a	$A := S{\cdot}a$
store	a	$S{\cdot}a := A$
add	a	$A := A + S{\cdot}a$
subt	a	$A := A - S{\cdot}a$
mult	a	$A := A * S{\cdot}a$

Om het verloop van de berekening te kunnen laten afhangen van reeds berekende resultaten, zoals bij de uitvoering van selecties en repetities, beschikt de machine verder over zogenaamde *spronginstructies*. Deze onderscheiden zich van de "normale" instructies doordat de toestandsveranderingen nu betrekking hebben op de programmaindex PI, overigens hebben spronginstructies geen invloed op de toestand van de machine. Het effekt van het toekennen van een nieuwe waarde, zeg a, aan PI is dat de machine verder gaat met het uitvoeren van de instructies in het geheugen vanaf adres a:

instructie:		toestandsverandering:
jump	a	$PI := a$
zjump	a	if $A=0 \rightarrow PI := a \;[]\; A\neq 0 \rightarrow$ skip fi
njump	a	if $A<0 \rightarrow PI := a \;[]\; A\geq 0 \rightarrow$ skip fi

Het effekt van jump a is dat de machine *altijd* verder gaat met de instructies vanaf adres a, dat wil zeggen onafhankelijk van de toestand waarin de machine zich op dat moment bevindt; dit wordt een *onvoorwaardelijke sprong* genoemd. Het effekt van zjump a en njump a is daarentegen wel afhankelijk van de toestand van de machine: deze instructies leiden alleen tot een verandering van PI als de waarde van het rekenregister aan een zekere voorwaarde voldoet –zoals nul zijn bij zjump en negatief zijn bij njump–; deze instructies worden daarom *voorwaardelijke sprongen* genoemd.

1.1 Over de mate van detail

De *betekenis* van een instructie is dus de toestandsverandering die bij uitvoering ervan teweeg wordt gebracht. Bij het implementeren zijn we vooral in deze betekenis geïnteresseerd, veel meer dan in de precieze *vorm* waarin de instructie moet worden opgeschreven. Die vorm is bovendien afhankelijk van de machine waarvoor de instructie is bedoeld. Daarom verdient het bij het ontwerpen van implementaties de voorkeur om de gewenste toestandsverandering op te schrijven, in plaats van instructiecodes uit een of ander machinerepertoire. Zoals we al hebben gezien kunnen we hiervoor assignments en expressies uit de "gewone" programmeertaal gebruiken. De precieze vorm van instructies, dat wil zeggen hun *codering*, beschouwen we dan als een detailkwestie die we het liefst zo lang mogelijk uitstellen: er zal ooit aandacht aan moeten worden besteed, maar bij het ontwerp van een implementatie willen we ons niet door coderingsaspecten laten afleiden.

voorbeeld 3: De assignment $B := S{\cdot}47$ geeft duidelijk aan welke toestandsverandering is bedoeld: register B krijgt als waarde de inhoud van geheugenwoord 47. In de ene machine wordt dit gecodeerd als load B 47, in een andere machine misschien als move 47 B : wat wordt nu waarnaar verplaatst? Om te weten wat dit betekent moeten we de conventies van zo'n schrijfwijze uit het hoofd kennen.

In het geheugen van de machine worden de instructies voorgesteld door bitrijtjes; in de code van de Motorola 68000 familie[5] wordt bijvoorbeeld $D1 := S{\cdot}47$ gecodeerd als:

0010 001 000 111 000 0000 0000 0010 1111 .

Het gebruik van (zogenaamde) *hexadecimale notatie* maakt zo'n rijtje

[5] waarin de registers $D0, D1, \cdots$ heten.

weliswaar korter, in dit voorbeeld tot:

```
2238 002F    ,
```

maar niet duidelijker. Het hoeft geen betoog dat we het opschrijven van zulke rijtjes bij voorkeur vermijden of tenminste uitstellen zolang dat kan.

□

2 Het gebruik van spronginstructies

2.0 Labels

Iedere spronginstructie bevat een adres en na het uitvoeren van de sprong haalt de processor de instructie in het geheugenwoord met dit adres op en voert deze uit. Om te bewerkstelligen dat een spronginstructie naar een gegeven instructie verwijst moeten we dus het adres van deze instructie[6] bepalen en in de spronginstructie opnemen. Dit vereist dat al bij het opschrijven van de code bekend is *waar* in het geheugen deze code bij uitvoering zal komen te staan, en dat is, zoals we later zullen zien, niet wenselijk. Zelfs als deze plaats al bekend is, is het uittellen van de adressen van individuele instructies weliswaar niet moeilijk maar wel bewerkelijk en gevoelig voor fouten: een foutief adres in een spronginstructie kan bij programmauitvoering volkomen onvoorspelbare gevolgen hebben.

Om deze problemen te vermijden gebruiken we bij voorkeur *labels* om instructies in de programmacode te identificeren. Een label is een naam –een identifier– die vóór een instructie wordt geplaatst; als de code met asserties is geannoteerd plaatsen we de label vóór de preconditie van de bedoelde instructie. Zo'n label representeert dan het geheugenadres van de instructie waar hij voorstaat. In spronginstructies gebruiken we nu zulke labels in plaats van adressen. Voordat het programma in uitvoering wordt genomen moeten eerst de bij de labels behorende adressen worden berekend en moeten de labels in de spronginstructies hierdoor worden vervangen. Dit kan echter door de machine zelf worden gedaan, met behulp van een zogenaamd *assembleerprogramma*. Aldus hoeven wij, als programmeurs, ons niet met het uittellen van adressen bezig te houden.

voorbeeld 4: Het volgende stukje code is een implementatie van de as-

[6] Preciezer: het adres van het geheugenwoord dat deze instructie bevat.

signment $x := |x|$; hierbij nemen we aan dat variabele x is ondergebracht in het geheugenwoord met adres 47, dat wil zeggen dat de representatie-invariant is:

$x = S{\cdot}47$.

Verder nemen we aan dat de programmacode wordt geplaatst in de geheugenwoorden met adressen vanaf 870; om dit zichtbaar te maken wordt hier iedere instructie in de linker kantlijn voorafgegaan door zijn adres:

```
870     { x = X }
        load   47
871     { x = X ∧ A = X }
        njump  873
872     { x = X ∧ 0 ≤ X }
        jump   876
873     { x = X ∧ A = X ∧ X < 0 }
        subt   47
874     { x = X ∧ A = 0 ∧ X < 0 }
        subt   47
875     { A = −X ∧ X < 0 }
        store  47
876     { x = |X| }
```

Dit stukje code is wat ingewikkelder dan men wellicht zou verwachten; dit is een gevolg van het zeer beperkte repertoire instructies: hier is alleen gebruik gemaakt van de op pagina 8 behandelde instructies. Met gebruik van labels ziet hetzelfde stukje code er zó uit:

$\{\ x = X\ \}$
load 47
$\{\ x = X \wedge A = X\ \}$
njump L
$\{\ x = X \wedge 0 \leq X\ \}$
jump M
L: $\{\ x = X \wedge A = X \wedge X < 0\ \}$
subt 47
$\{\ x = X \wedge A = 0 \wedge X < 0\ \}$
subt 47
$\{\ A = -X \wedge X < 0\ \}$
store 47
M: $\{\ x = |X|\ \}$

□

opgave 2: Construeer volledig geannoteerde code voor de implementatie van $x := x \,\mathsf{max}\, y$.

□

2.1 Bewijsregels

Uitspraken over de toestand van de berekening worden in het programma vastgelegd in de vorm van asserties; hierin komen de programmavariabelen voor. Om in programma's in machinecode ook asserties te kunnen gebruiken hebben we ook *bewijsregels* nodig voor de spronginstructies, net zoals we bewijsregels voor assignments en de andere programmaconstructies hebben. Omdat asserties alleen programmavariabelen bevatten, komt de programmaindex PI niet in asserties voor: PI is geen programmavariabele. Anderzijds beïnvloeden de spronginstructies juist alleen PI en niet de waarden van de programmavariabelen. Het uitvoeren van een spronginstructie verstoort daarom geen asserties: de asserties in het programma zijn invarianten van de spronginstructies.

Op deze overwegingen zijn de volgende regels gebaseerd. We behandelen hier alle mogelijke spronginstructies in één klap, door regels te geven voor een algemenere spronginstructie, die we noteren als $\mathsf{JUMP}(\beta)\ L$, waarin β een boolean expressie is over de toestand van de machine en L een label. De bijbehorende toestandsverandering is dan als volgt, waarin $addr{\cdot}L$ het adres weergeeft dat met de label L overeenkomt:

instructie:	toestandsverandering:
JUMP(β) L	if $\beta \rightarrow PI := addr{\cdot}L$ [] $\neg\beta \rightarrow$ skip fi

voorbeeld 5: De instructies jump , zjump en njump kunnen nu worden gedefinieerd als JUMP(true) , JUMP($A{=}0$) en JUMP($A{<}0$) .
□

Als nu in een machinecodeprogramma de label L voorkomt, en wel voor een instructie met preconditie R:

$$\begin{array}{ll} L: & \{\ R\ \} \\ & \cdots \end{array} \quad ,$$

en als een spronginstructie met preconditie P en postconditie Q naar label L verwijst:

$$\begin{array}{l} \{\ P\ \} \\ \mathsf{JUMP}(\beta) \quad L \\ \{\ Q\ \} \end{array} \quad ,$$

dan is het gebruik van *deze* spronginstructie *correct* als geldt:

(0) $P \ \wedge \neg\beta \ \Rightarrow \ Q$ én bovendien:

(1) $P \ \wedge \ \beta \ \Rightarrow \ R$.

Regel (0) drukt uit dat Q een geldige postconditie van de spronginstructie moet zijn. Deze regel volgt onmiddellijk uit de definitie van JUMP(β) L, zoals hierboven gegeven. Regel (1) drukt uit dat de assertie bij een label moet volgen uit de preconditie en de guard van *elke* spronginstructie die naar die label verwijst. Uiteraard moet de assertie R ook een geldige postconditie zijn van de er direkt aan voorafgaande instructie, maar dat is geen onderdeel van de correctheid van de spronginstructies die naar L verwijzen.

Een speciaal geval is de onvoorwaardelijke sprong JUMP(true) L; als we in regels (0) en (1) voor β true invullen krijgen we de volgende bewijsregels voor de onvoorwaardelijke sprong:

(2) $P \ \wedge$ false $\Rightarrow \ Q$ én bovendien:

(3) $P \ \wedge$ true $\Rightarrow \ R$.

Formule (2) geldt uiteraard voor elke Q: een onvoorwaardelijke sprong mag elke assertie als postconditie hebben, zelfs false. De enige bewijsverplichting die hier overblijft is dus dat de preconditie van de sprong de assertie bij de label impliceert; formule (3) komt immers neer op:

$$P \;\Rightarrow\; R \quad .$$

opgave 3: Verifieer met behulp van deze regels de correctheid van alle asserties in het programma in voorbeeld 4.
□

2.2 Selectie en repetitie

We bekijken nu de implementatie, met behulp van spronginstructies, van een selektie van de volgende vorm, waarin B_0 en B_1 boolean expressies zijn en S_0 en S_1 statements:

$\{\ P\ \}$
if $B_0 \;\rightarrow\; S_0$
[] $B_1 \;\rightarrow\; S_1$
fi
$\{\ R\ \}$

Dit is een *samengestelde* statement, met als samenstellende delen de expressies B_0 en B_1 en de statements S_0 en S_1. Voor deze statements geldt (kennelijk) $\{\, P \wedge B_0 \,\}\ S_0\ \{\, R \,\}$ en $\{\, P \wedge B_1 \,\}\ S_1\ \{\, R \,\}$. We proberen nu de code voor de implementatie van deze selektie samen te stellen uit stukjes code voor de implementatie van B_0, B_1, S_0 en S_1. Deze stukjes code geven we aan met $code{\cdot}B_0$, $code{\cdot}B_1$, $code{\cdot}S_0$ en $code{\cdot}S_1$, waarbij $code{\cdot}B_0$ en $code{\cdot}B_1$ staan voor stukjes code die de waarde van de boolean expressies B_0 en B_1 uitrekenen en in het rekenregister achterlaten; hierbij nemen we aan dat het getal 0 de boolean waarde false voorstelt en dat true door een van 0 verschillende waarde, bijvoorbeeld -1, wordt voorgesteld. Met behulp hiervan kan de selektie als volgt worden geïmplementeerd[7]:

$IF:$ $\{\ P\ \}$
$code{\cdot}B_0$
$\{\ P \wedge\ (B_0 \equiv A{\neq}0)\ \}$

[7] Vele variaties zijn mogelijk.

$$
\begin{array}{ll}
 & \mathsf{zjump}\ \ L_1 \\
 & \{\ P \wedge B_0\ \} \\
 & \mathit{code}{\cdot}S_0 \\
 & \{\ R\ \} \\
 & \mathsf{jump}\quad FI \\
L_1: & \{\ P \wedge \neg B_0\ \} \\
 & \mathit{code}{\cdot}B_1 \\
 & \{\ P \wedge \neg B_0 \wedge\ (B_1 \equiv A{\neq}0)\ \} \\
 & \mathsf{zjump}\ \ L_2 \\
 & \{\ P \wedge B_1\ \} \\
 & \mathit{code}{\cdot}S_1 \\
 & \{\ R\ \} \\
 & \mathsf{jump}\quad FI \\
L_2: & \{\ P \wedge \neg B_0 \wedge \neg B_1\ \} \\
 & \mathsf{jump}\quad L_2 \\
FI: & \{\ R\ \}
\end{array}
$$

opgave 4: De uitvoering van het volgende fragment uit bovenstaande code eindigt niet:

$$
\begin{array}{ll}
L_2: & \{\ P \wedge \neg B_0 \wedge \neg B_1\ \} \\
 & \mathsf{jump}\quad L_2
\end{array}
$$

Dat is ook de bedoeling. Waarom?

□

We beschouwen nu de implementatie van een repetitie van de vorm:

$$
\begin{array}{l}
\{\ P\ \} \\
\mathsf{do}\ B\ \ \rightarrow\ \ S\ \{\ P\ \}\ \mathsf{od} \\
\{\ R\ \}
\end{array}
$$

Ook hier zijn vele variaties mogelijk; de hier gegeven implementatie is zo gestructureerd dat bij elke "slag" van de repetitie slechts één spronginstructie wordt uitgevoerd:

$$
\begin{array}{ll}
DO: & \{\ P\ \} \\
 & \mathsf{jump}\quad L_0 \\
L_1: & \{\ P \wedge B\ \} \\
 & \mathit{code}{\cdot}S \\
L_0: & \{\ P\ \}
\end{array}
$$

$code{\cdot}B$
$\{\ P \wedge\ (B \equiv A{<}0)\ \}$
njump L_1
OD : $\{\ R\ \}$

opgave 5: Verifieer de correctheid van de hier gegeven implementaties.
□

3 Allocatieonafhankelijkheid

Het kiezen van een plaats in het geheugen voor de code van het uit te voeren programma en het kiezen van geheugenplaatsen voor de variabelen uit dat programma wordt *(geheugen)allocatie* genoemd. Bij de klassieke Von Neumann machine is de programmacode afhankelijk van de allocatie: de adressen van de variabelen komen immers in de instructies voor waarmee de waarden van deze variabelen worden gelezen en geschreven, en in de spronginstructies komen de adressen van andere instructies voor. Dit heeft nogal wat gevolgen:

- De geheugenallocatie moet *bekend* zijn op het moment dat de code wordt gecomponeerd.
- De programmacode moet worden *gewijzigd* zodra de keuze van de plaatsen voor de variabelen wordt herzien.
- De programmacode moet ook worden *gewijzigd* zodra de keuze van de plaats voor de code zelf wordt herzien.
- De programmacode is *gebonden* aan de plaatsen van de variabelen.

Vanwege deze eigenschappen heeft het gebruik van allocatie-afhankelijke code de volgende bezwaren:

- De geheugenallocatie moet bekend zijn op het moment dat de code wordt gecomponeerd, maar de allocatie hangt af van wat er zich reeds in het geheugen van de machine bevindt op het moment dat het programma in uitvoering wordt genomen, zoals het operating system en eventuele andere programma's in uitvoering. Daarom is vroegtijdige allocatie onmogelijk.
- Het wijzigen van de programmacode ten gevolge van herzieningen van de allocatie is bewerkelijk, vooral als dit tijdens (een onderbreking van)

de uitvoering van het programma moet plaatsvinden. Het herzien van de allocatie tijdens programmauitvoering, ook wel *dynamische relocatie* genoemd, kan nodig zijn omwille van een flexibel *geheugenbeheer*.

- In een (zogenaamd) *multiprocessing* systeem kunnen meer dan één programma's tegelijkertijd in uitvoering zijn. Het is zelfs denkbaar dat *dezelfde* programmacode meer dan eens wordt uitgevoerd door verschillende gelijktijdig verlopende berekeningen. Dit heet *codesharing*. Ieder van zulke berekeningen vereist echter een *eigen* toewijzing van geheugenplaatsen voor de variabelen van die berekeningen: de berekeningen zijn onafhankelijk en zij voeren weliswaar dezelfde code uit maar hun variabelen hebben meestal niet dezelfde waarden. Als de programmacode aan de plaatsen van deze variabelen is gebonden, is codesharing onmogelijk en zal voor iedere uitvoering van een programma een aangepaste kopie van de code moeten worden gecreëerd en in het geheugen worden geplaatst. Dit is nodeloos inefficiënt. De behoefte aan codesharing doet zich bovendien ook voor bij de uitvoering van *recursieve* procedures, wat immers als een speciale vorm van multiprocessing kan worden opgevat.

Het eerste bezwaar kan worden ondervangen door het gebruik van een (zogenaamde) *relocating loader*, waarmee op het moment waarop de code in uitvoering wordt genomen de adressen in de code overeenkomstig de allocatie worden aangepast. De relocating loader kan ook worden gebruikt om de code aan te passen aan een gewijzigde allocatie, voorzover deze wijziging tenminste niet tijdens de uitvoering van het programma plaatsvindt. De behoefte aan dynamische relocatie kan verder worden vermeden door middel van technieken voor geheugenbeheer[8] zoals *paging*, al vereist een efficiënte realisatie hiervan wel speciale voorzieningen in de machine.

De onmogelijkheid van codesharing is echter een fundamentele tekortkoming van allocatie-afhankelijke code; omdat iedere moderne programmeertaal recursie toelaat en omdat multiprocessing op steeds grotere schaal, ook in "kleine" computers, wordt gebruikt, is allocatie-afhankelijke code ongewenst.

Het uitgangspunt voor de studie van implementaties is dan ook dat de code van programma's allocatie-onafhankelijk dient te zijn. Het middel hiertoe is het gebruik van *indexregisters* en *indirekte adressering*. Als gevolg hiervan zullen er (vrijwel) geen geheugenadressen meer in de code voorkomen, maar wel *relatieve* plaatsaanduidingen[9] ten opzichte van in indexregisters

[8] In de meeste *personal computers* is nauwelijks sprake van geheugenbeheer.
[9] in de manuals vaak *displacements* of *offsets* genoemd.

gehouden adressen. Deze werkwijze heeft bovendien als voordelen dat relocating loaders overbodig worden en dat de programmacode korter wordt, omdat relatieve plaatsaanduidingen in de regel beknopter zijn dan (absolute) geheugenadressen.

Eindhoven, 8 mei 1996

Rob R. Hoogerwoord
faculteit der Wiskunde en Informatica
Technische Universiteit Eindhoven
postbus 513
5600 MB Eindhoven