14
44

*)

TRANSFORMATIE VAN SPECIFICATIE TOT IMPLEMENTATIE

Maarten M. Fokkinga
Universiteit Twente, Faculteit der Informatica
Postbus 217, NL 7500 AE Enschede
Nederland

Samenvatting Constructies van correcte en efficiente programma's zijn mogelijk door stapje-voor-stapje de specificatie (in wiskundige notatie) te transformeren onder behoud van betekenis. De tussenstadia zijn steeds beter executeerbare specificaties; het eindpunt is een wiskundige formule die tevens een legale expressie is een Functionele Taal zoals SASL, Miranda, Hope of Twentel. Een volgende stap, de omzetting tot een Imperatief (Pascal) programma is nu ook mogelijk.

Bij zo'n transformatieproces worden methoden en technieken toegepast die, veralgemeend en geschematiseerd, als afzonderlijke lemma's bewezen kunnen worden. Voor de praktische uitvoerbaarheid van deze manier van programma-constructie is het nodig dat de specificatie- en programmanotatie zeer beknopt is en geschikt voor algebraïsche manipulatie.

Bovenstaande wordt aan de hand van "Backtracking" en nog een probleem toegelicht.

*) Te verschijnen in: Software Specificatie Technieken, (eds Poirters, J.A.A.M. & Schoenmakers, G.J.), 1987

INLEIDING

In dit verhaal gaan wij in op het onderwerp Software Specificatie Technieken vanuit het gezichtspunt dat specificaties een hulpmiddel kunnen en moeten zijn bij de constructies van implementaties, dat is: programma's die aan de specificaties voldoen. We beschouwen alleen dat aspect van ~~programma's en~~ specificaties dat zich over de gewenste relatie tussen invoer en uitvoer uitspreekt, het zgn. functionele aspect. ~~van~~ Andere aspecten, zoals real-time beperkingen, laten we volkomen achterwege. Het onderwerp Software Specificatie Technieken kan natuurlijk ook vanuit andere gezichtspunten bekeken worden en wellicht leiden die tot conclusies die tegengesteld zijn aan wat wij beweren. Voor een algemene beschouwing verwijzen we naar [van Amstel 1987] en voor andere, specifieke, benaderingen raadplege men [Poirters & Schoenmakers 1987].

Wij gaan er van uit dat een formele specificatie een hulpmiddel moet zijn bij de programmaconstructie; en sterker nog, dat een gewenst programma geconstrueerd kan worden door stapje voor stapje de specificatie te transformeren totdat zij execu-

~~en~~teerbaar is geworden en voldoende efficient. Deze manier van programmaconstructie legt bepaalde eisen op aan de notatie waarin specificaties en ~~ge~~ programma's gesteld worden. Dat zijn met name de geschiktheid voor algebraïsche manipulatie, en daarmee samenhangend de beknoptheid en het wiskundige karakter van de notatie. Tijdens het transformatieproces wordt steeds weer een beroep gedaan op algebraïsche wetten en andere lemma's die tot het standaard pakket van technieken en methoden behoren van de Transformationele Programmeur. Met deze visie volgen wij Meertens [Meertens 1986], en veel van wat wij zeggen is direkt geïnspireerd door of ontleend aan dat werk; soortgelijk werk is [Bird 1986].

Wij zullen het volledige traject van Specificatie tot en met Implementatie doorlopen voor twee voorbeelden: het Wandelingen Probleem en het Acht Koninginnen (Backtracking) Probleem. Daarmee pogen we de lezer te overtuigen van de zinvolheid van de transformationele manier van programmaconstructie, en van de eigenschappen die we van de notatie zullen eisen. De twee delen die we aan de twee voorbeelden wijden, zijn onafhankelijk van elkaar te lezen.

DEEL A: HET WANDELINGEN PROBLEEM

In dit deel formuleren we het Wandelingen Probleem en doorlopen we het traject van Specificatie tot Implementatie daarvoor. We gebruiken hierbij een gedeelte van de notatie die door Meertens [Meertens 1986] is ontworpen. We benadrukken bij voorbaat al dat we slechts een gedeelte van zijn notatie en ideeën gebruiken en dat Meertens in zijn werk veel meer en veel diepzinniger uiteenzettingen geeft over de notatie en het gebruik ervan dan wij in staat zijn hier te doen.

De informele beschrijving van het Wandelingen Probleem luidt als volgt.

Beschouw <u>wandelingen</u> in het platte vlak, bestaande uit <u>stappen</u> (ter lengte 1m, met duur 1 sec en in richting N, O, Z of W). De <u>rand</u> van een wandeling bestaat uit de zijden van de kleinste rechthoek (met zijden in NZ en OW richting) waarbuiten de wandeling niet komt. Gevraagd wordt om voor willekeurige wandeling (beginnend op tijdstip 0 en gegeven als rij van stappen) de twee tijdstippen te bepalen waarop de wandelaar zich voor het eerst, respectievelijk het laatst, op de uiteindelijke rand bevindt.

Dit probleem is ontleend aan [EXIN 1985]. Volgens zeggen was het probleem bedoeld als een programmeeropgave waarbij het nut en zelfs de noodzaak van zgn. invarianten duidelijk zou blijken. Bij onze aanpak van dit programmeerprobleem blijken invarianten niet nodig te zijn, maar worden ze er desgewenst "gratis" bijgeleverd.

A1. Formele specificatie

We zullen een zeer beknopte en op het eerste gezicht enigszins cryptische notatie gebruiken. Om die eigenschappen te motiveren bekijken we eerst een afleiding van $a^2 - b^2 = (a+b)(a-b)$. (Als we aannemen dat optelling en aftrekking elk minder kosten dan vermenigvuldiging en kwadratering, dan is $(a+b)(a-b)$ een efficientere manier om $a^2 - b^2$ uit te rekenen dan het recept $a^2 - b^2$ zelf.)

$$
\begin{aligned}
a^2 - b^2 &= a^2 - ab + ab - b^2 \\
&= a^2 - ab + ba - b^2 \\
&= a(a-b) + b(a-b) \\
&= (a+b)(a-b)
\end{aligned}
$$

We merken hetvolgende op.
1. Voor wie niet bekend is met de gebruikte notatie komt dit over als goochelarij.
2. Wie in staat wil zijn dergelijke afleidingen ook zelf uit te voeren, moet op z'n minst vaardig zijn in de notatie en in het hanteren van wetten zoals $x - x = 0$, $xy = yx$ en $xy + xz = x(y+z)$. Vaardigheid wordt alleen door oefening verkregen.
3. Inzicht is nodig om de afleiding te sturen. Met name vereist de introductie van de termen $-ab+ab$ enige kennis van het uiteindelijke doel. Het valt niet te verwachten dat afleidingen zoals boven gevonden worden door zomaar wetten uit de rekenkunde toe te passen.
4. Sommige expressies in de afleiding zijn zelfs minder efficient dan de beginexpressie $a^2 - b^2$.
5. Door het ontbreken van haakjes is de syntactische structuur niet eenduidig bepaald, hoewel de betekenis --gelukkig-- ondubbelzinnig vast ligt. Bijvoorbeeld, staat $a^2 - ab + ab - b^2$

voor $((a^2 - ab) + (ab - b^2))$ of voor $(a^2 - ab + ab) - b^2$?
Zonder deze syntactische dubbelzinnigheid zou de afleiding langer en ingewikkelder zijn.

6. De expressies zijn leesbaar en beknopt gemaakt door het gebruik van infix-operatoren in plaats van funkties, door de belangrijkste (= vaak gebruikte) operator juist niet te schrijven, en door haakjes weg te laten in ruil voor geschikt gekozen ontledingsregels ("vermenigvuldiging heeft voorrang boven optelling"). Beschouw bijvoorbeeld eens:

$$\mathrm{add}(\mathrm{minus}(\mathrm{exp}(2,a), \mathrm{mult}(a,b)), \mathrm{minus}(\mathrm{mult}(a,b), \mathrm{exp}(2,b)))$$

Met zo'n notatie zijn dergelijke afleidingen praktisch gesproken ondoenlijk: enerzijds is het schrijfwerk te veel en anderzijds zijn de patronen (de "idiomen") onherkenbaar voor het menselijk oog.

7. Vrijwel niemand schrijft de afleiding ooit nog voluit op. Ten eerste is die gelijkheid vanwege z'n grote toepasbaarheid tot het standaard pakket gereedschappen gaan behoren en ten tweede is het voor een geoefende niet meer nodig om in zulke kleine stappen als boven te werk te gaan.

Deze 7 opmerkingen gelden net zo voor de notatie en manipulatie die we nu gaan hanteren.

A2. De notatie

Verreweg de belangrijkste een meest voorkomende operaties zijn de functie-compositie en functie-applicatie. Deze beiden worden daarom niet expliciet geschreven. Daarmee is $f\ g\ a$ syntactisch dubbelzinnig, omdat het zowel $(f \circ g)(a)$ alsook $f(g(a))$ kan betekenen. Gelukkig is dat geen semantische dubbelzinnigheid, omdat $(f \circ g)(a) = f(g(a))$. (Alleen wanneer zowel $f \circ g$ alsook $f(g)$ zinvol zijn, is er mogelijk ook een semantische dubbelzinnigheid. Maar dat doet zich in ons verhaal niet voor.) De twee-eenheid applicatie-compositie wordt appositie genoemd; (apposition is engels voor: naast elkaar plaatsen).

Voorts gebruiken we de volgende operaties en functies:

$+$	optelling	$4+3=7$
$-$	aftrekking	$4-3=1$
$\downarrow$	minimum	$4\downarrow 3 = 3$
$\uparrow$	maximum	$4\uparrow 3 = 4$

lijsten, bijv. [] en $[a, b, a, c] \neq [a, a, b, c] \neq [a, b, c]$.

$:$	op (kop van)	$a:[b,a,c] = [a,b,a,c]$
$++$	(aaneengevoegd) met	$[a,b]{+}{+}[a,c] = [a,b,a,c]$
$\#$	lengte	$\#[a,b,a,c] = 4$
inits	initiele delen van	$\text{inits}\,[a,b,c] = [[\,],[a],[a,b],[a,b,c]]$

Bijzondere operaties:

$/$	insert (of: reduce)	$+/[a,b,c] = a+b+c$
$*$	(appose) to all	$f*[a,b,c] = [f\,a, f\,b, f\,c]$
$\triangleleft$	filter	$\text{odd}\triangleleft[1,2,1,7,4] = [1,1,7]$

De linkerargumenten moeten altijd zo klein mogelijk gekozen worden, de rechterargumenten mogen ontbreken of dienen zo groot mogelijk gekozen te worden. Bij ontbrekend rechter-argument, bijv. (4+), resulteert er een functie die alsnog op een (rechter)argument toegepast kan worden. Dus 4+3 kan ook ontleed worden als (4+)3 . De "bijzondere operaties" $/ * \triangleleft$ spelen alleen syntactisch een bijzondere rol: zij <u>moeten</u> een linker-argument hebben, de overige operaties en functies mogen als zelfstandige objecten voorkomen, zoals de $+$ in $+/[a,b,c]$.

Hier is ter illustratie een formule (die straks zal verschijnen), gevolgd door dezelfde formule maar dan voorzien van haakjes die de ontleding aangeven (en met $\circ$ voor de compositie):

$$0 \downarrow \downarrow / (\# \, s:) * (r_{s:w} \, s:) \triangleleft \text{inits } w$$

$$= 0 \downarrow \Bigl(\downarrow / \Bigl((\# \circ (s:)) * \Bigl((r_{s:w} \circ (s:)) \triangleleft \Bigl(\text{inits} \bigl(w \bigr) \Bigr) \Bigr) \Bigr) \Bigr)$$

De ontledingsafspraken hebben als voordeel dat veel haakjes niet geschreven hoeven worden en dat het makkelijk wordt om over functies (= programma's) te redeneren als zelfstandige objecten. Een nadeel is dat het type van de objecten nodig is om te bepalen of een appositie een compositie dan wel een applicatie is, en dat de ontledingsregels afwijken van wat gebruikelijk is in de wiskunde.

A3. De specificatie

We stellen nu een specificatie op van het informeel gegeven probleem.

Gegeven zijn:

Stap == N|O|Z|W

w :: [Stap]

(Hier wordt in de notatie van Miranda, [Turner 1985], een type Stap gedefinieerd; een stap is één van de vier mogelijkheden, genaamd N, O, Z en W. Voorts wordt gesteld dat w een lijst van stappen is; w is de gegeven wandeling.)

Gevraagd wordt:

t w en T w

waarbij

t w = 1ste tijdstip waarop de wandelaar op de w-rand is
= lengte vd kleinste vd op de w-rand eindigende initiele delen van w
= minimum vd lengtes vd op w-rand eindigende initiele delen van w

$= \downarrow/ \quad \#* \quad r_w \triangleleft \quad \text{inits } w$

$= \downarrow/ \; \#* \; r_w \triangleleft \; \text{inits } w$

$T\ w = \uparrow/ \; \#* \; r_w \triangleleft \; \text{inits } w$

$r_w\ w'$ = w' eindigt op de rand van w

$= \neg$(w' eindigt binnen de rand van w)

$= \neg$(min. x-verplaatsing tgv w <

netto x-verplaatsing tgv w' <

max. x-verplaatsing tgv w

& --- net zo voor y ---

)

$= \neg$ ((mx w)< (x w')< (Mx w) & ---)

x w = netto verplaatsing in de x-richting tgv w

= som vd verplaatsingen in x-richting vd stappen van w

= +/ dx* w

= +/ dx* w

mx w = min. x-verplaatsing tgv w

= minimum vd netto x-verplaatsingen tgv de initiele delen van w

= ↓/ x* inits w

= ↓/ x* inits w

Mx w = ↑/ x* inits w

dx s = 1 als s=O

= -1 als s=W

= 0 als s=N of s=Z

y, my, My, dy analoog.

Opmerkingen.

1. In de formele specificatie wordt gebruik gemaakt van begrippen die in de informele probleemstelling geen rol spelen. In dit geval is

dat met name de keuze van een cartesisch coördinatenstelsel met de x-as en y-as in WO resp. ZN richting. We hebben dit zoveel mogelijk proberen te vermijden. Er is bijvoorbeeld niet vastgelegd waar de oorsprong van dat coordinatenstelsel gekozen wordt. We hadden desnoods het coördinatenstelsel geheel kunnen vermijden door slechts over "verplaatsing in de θ-richting" etc te spreken.

2. Niet alle begrippen die duidelijk in de informele probleemstelling gebruikt worden, zijn hierboven apart gedefinieerd. Bijvoorbeeld, de wandelaar is niet rechtstreeks formeel weergegeven. Desgewenst kunnen we stellen dat de wandelaar (die wandeling w aflegt) een functie is van de tijd, en wel zo dat voor ieder tijdstip zijn x- en y-positie wordt gegeven:

$$\text{wandelaar } t = (x\ w', y\ w')$$
$$\underline{\text{where}} \quad [w'] = (t = \#) \lhd \text{inits } w$$

(Ter verklaring: het predicaat $(t=\#)$ moet ontleed worden als $(t=)\circ\#$ zodat $(t=\#)\,w' = ((t=)\circ\#)\,w' = (t=)(\#\,w') = \text{true}$ precies wanneer w' lengte t heeft. De definitie $[w'] = xxx$ introduceert w' als naam voor het enige element van de singletonlijst xxx .)

3. Er zijn natuurlijk ook andere formuleringen mogelijk die hetzelfde specificeren. We kunnen bijvoorbeeld T w formuleren aan de hand van t voor de "van achter naar voren doorlopen" wandeling:

$$T\,w = (\#w) - t\ \mathrm{rev}\ \mathrm{inv}{*}\ w$$

met

$$\mathrm{rev}\ [a, b, \ldots, z] = [z, \ldots, b, a]$$

$$\mathrm{inv}\ N = Z, \quad \mathrm{inv}\ Z = N, \quad \mathrm{inv}\ O = W, \quad \mathrm{inv}\ W = O.$$

Maar het is niet op slag duidelijk dat

$$\uparrow/\ \#{*}\ r_w{\triangleleft}\ \mathrm{inits}\ w$$
$$= (\#w) - \downarrow/\ \#{*}\ r_{\mathrm{rev}\ \mathrm{inv}{*}\ w}{\triangleleft}\ \mathrm{inits}\ \mathrm{rev}\ \mathrm{inv}{*}\ w$$

Bij het ontwerpen van een programma moet men zich niet verplicht voelen de structuur van de specificatie te volgen: de specificatie dient er alleen voor om zo duidelijk mogelijk de in/uitvoer-eisen van het programma vast te leggen.

4. Al bij het opstellen van de eerste formele specificatie hebben we enige probleemanalyse gedaan, zie bijvoorbeeld de manier waarop

we tot de uiteindelijke formulering van t w = ---
zijn gekomen. De formele specificatie is dus
al de eerste stap van het programma-ontwerp.

5. Expliciete voorkomens van probleem-irrelevante representatie-onderdelen vormen vaak een bron van fouten. Dit is heel duidelijk bij arrays, waar indices nogal eens één uit de pas lopen: die indices spelen geen rol in de gegeven probleemstelling, maar wel in de representatie van begrippen erván. (Dáárom is het zo nodig om de bedoeling van die indices door middel van "asserties" of "invarianten" vast te leggen.) Ook bij constructies zoals

$$\underline{S}\ i:\ 0 \le i < n-1:\ a[i]$$
$$\underline{A}\ i:\ 0 \le i < n-1:\ a[i] \ne x$$

worden namen geïntroduceerd, i in deze gevallen, die meestal in de informele probleemstelling geen rol spelen. In de door ons gebruikte notatie kunnen we die naamgeving vermijden door $+/a$ te schrijven en $\&/(x\ne)*a$.

6. De gegeven specificatie is toevallig al executeerbaar, zij het dat de tijdscomplexiteit nog onbevredigend is, (in ons geval: tenminste kwa-

dratisch in de lengte van de wandeling). In het algemeen zullen er in een formele specificatie ook niet-executeerbare operaties gebruikt worden, zoals: het minimum van een oneindige rij natuurlijke getallen, de gesorteerde versie van zo'n rij, en een filter op het Universum van alle mogelijke waarden.

7. De te bepalen waarden t w en T w liggen volkomen vast. In het algemeen laat de probleemstelling nog enige vrijheid toe in de op te leveren waarden. In de formele weergave blijkt dat dan uit het gebruik van nondeterministische operatoren, of van verzamelingen waarvan slechts ~~zomaar~~ een element gevraagd wordt. We gaan hier verder niet op in.

A4. Van specificatie tot implementatie

We zullen uit de specificatie nu een andere formulering, en zelfs ook een Pascal-programma, afleiden, met een tijdscomplexiteit die lineair is in de lengte van de gegeven wandeling. Deze afleiding zal er op papier uit zien als een recht-toe recht-aan haast automatisch verlopende afleiding. Maar let wel: we tonen hoe de afleiding had kunnen verlopen; de werkelijke afleiding gaat meestal met vallen en opstaan gepaard, en vereist soms inzicht, ervaring en Geluk bij de keuze van de toe te passen techniek en methode.

Een van de standaardtechnieken die ons ter beschikking staan is om te proberen voor t en T een inductieve definitie op te stellen, dwz. om te proberen $t\,[]$ expliciet uit te drukken en $t\ s{:}w$ uit te drukken in $t\ w$ (en net zo voor $T[]$ en $T\ s{:}w$). Uiteraard moeten we dan ook voor de daarbij benodigde hulpgrootheden proberen inductieve definities op te stellen. Hier gaan we.

$t\,[]$

$=$ {per definitie van t}

$\quad \downarrow/\ \#*\ r_w \triangleleft\ inits\ w \qquad$ met $w=[]$

$=$ {$w=[]$, definitie van inits}

$\quad \downarrow/\ \#*\ r_{[]} \triangleleft\ [[]]$

$=$ {[] eindigt op de rand van [], formeel: $r_{[]}[] = true$. Dit is apart te bewijzen, mbv. $x[] = +/dx*[] = +/[] = 0$ etc.}

$$\downarrow/\ \#*\ [[\,]]$$
$= \{$definitie $*\}$
$$\downarrow/\ [\#[\,]]$$
$= \downarrow/\ [0]$
$= 0.$

$t\ s{:}w$
$= \downarrow/\ \#*\ r_{s:w} \lhd \text{ inits } s{:}w$
$= \{\text{inits } x{:}y = [\,]{:}\ (x{:})*\ \text{inits } y\}$
$\quad \downarrow/\ \#*\ r_{s:w} \lhd [\,]{:}\ (s{:})*\ \text{inits } w$

en nu ligt het onderscheid tussen $r_{s:w}[\,] = \text{true}$ en $r_{s:w}[\,] = \text{false}$ van groot belang: het beginpunt van de wandeling $s{:}w$ ligt wel/niet op de rand. We behandelen deze gevallen afzonderlijk.

Neem aan: $r_{s:w}[\,] = \text{true}$ en schrijf r' voor $r_{s:w}$. Dan

$t\ s{:}w$
$= \{$volgens bovenstaande afleiding$\}$
$\quad \downarrow/\ \#*\ r' \lhd [\,]{:}\ (s{:})*\ \text{inits } w$
$= \{$aanname $r'[\,] = \text{true}\}$
$\quad \downarrow/\ \#*\ [\,]{:}\ r' \lhd (s{:})*\ \text{inits } w$
$= \{$wegens $f * x{:}y = (f\ x){:}\ f * y\}$
$\quad \downarrow/\ 0{:}\ \#*\ r' \lhd (s{:})*\ \text{inits } w$
$= \{$wegens $\downarrow/\ x{:}y = x \downarrow \downarrow/\ y\}$
$\quad 0 \downarrow \downarrow/\ \#*\ r' \lhd (s{:})*\ \text{inits } w$

$=$ { allen van $(s:)*$ inits w hebben lengte ≥ 1,
dus is het minimum ≥ 0 }
0.

Neem aan $r_{s:w}\,[\,] = false$ en schrijf r' voor $r_{s:w}$. Dan

$t\ s:w$
$=$ {volgens eerder begonnen afleiding}
$\downarrow/\ \#*\ r' \lhd [\,]:\ (s:)*\ inits\ w$
$=$ {aanname $r'\ [\,] = false$, $[\,]$ wordt weggefilterd}
$\downarrow/\ \#*\ r' \lhd (s:)*\ inits\ w$
$=$ {wegens $p \lhd f* = f*(p\ f)\lhd$ }
$\downarrow/\ \#*\ (s:)*\ (r'\ s:)\lhd\ inits\ w$
$=$ {wegens $f*g* = (f\ g)*$ }
$\downarrow/\ (\#\ s:)*\ (r'\ s:)\lhd\ inits\ w$
$=$ { $(\#\ s:)\ x = \#\ s:x = 1+\#x = (1+\#)\ x$ }
$\downarrow/\ (1+\#)*\ (r'\ s:)\lhd\ inits\ w$
$=$ {wegens $(1+\#) = (1+)\ \#$ en $(f\ g)* = f*g*$ }
$\downarrow/\ (1+)*\ \#*\ (r'\ s:)\lhd\ inits\ w$
$=$ { wegens $\downarrow/\ f* = f\ \downarrow/$ voor monotone f }
$1+\ \downarrow/\ \#*\ (r'\ s:)\lhd\ inits\ w$
$=$ {we proberen $t\ s:w$ in $t\ w$ uit te drukken}
$1+\ \downarrow/\ \#*\ r''\lhd\ inits\ w$
met $r'' = r'\ s:$
$=$ {wegens
$r''\ w'$
$=$ { per definitie van r'' }

$r' \quad s : w'$

$= \{$definitie van r' $\}$

$\neg(((mx\ s{:}w) < (x\ s{:}w') < (Mx\ s{:}w))\ \&\ {-}{-}{-}y{-}{-}{-})$

$= \{$volgens aanname is $r'\ [\,] = false$, i.e.

true $= \neg r\ [\,]$

$= ((mx\ s{:}w) < (x\ [\,]) < (Mx\ s{:}w))\ \&\ {-}{-}{-}y{-}{-}{-}$

$= \{$definitie mx en $Mx\}$

$((x\ [\,]) \downarrow (dx\ s) + mx\ w) < x\ [\,] < ((x\ [\,]) \uparrow (dx\ s) + Mx\ w)$

zodat $mx\ s{:}w = (dx\ s) + mx\ w$

$Mx\ s{:}w = (dx\ s) + Mx\ w$

$\}$

$\neg(((dx\ s) + mx\ w) < (x\ s{:}w') < ((dx\ s) + Mx\ w))\ \&\ {-}{-}{-}y{-}{-}{-})$

$= \{$definitie van $x\}$

$\neg(((dx\ s) + mx\ w) < ((dx\ s) + x\ w') < ((dx\ s) + Mx\ w))\ \&\ {-}{-}{-})$

$= \neg((mx\ w) < (x\ w') < (Mx\ w))\ \&\ {-}{-}{-}y{-}{-}{-})$

$= r_w\ w'$

$\}$

$1 + \downarrow/\ \#{*}\ r_w \triangleleft$ inits w

$= 1 + t\ w$

Aldus hebben we verkregen

$t\ [\,] = 0$

$t\ s{:}w = 0$ als $\neg(((mx\ s{:}w) < 0 < (Mx\ s{:}w))\ \&\ {-}{-}{-}y{-}{-}{-})$

$= 1 + t\ w$ anders

De definitie maakt gebruik van mx, Mx, my en

My, dus ook deze moeten inductief uitgedrukt worden. Dat is vrij gemakkelijk en we vinden:

$$
\begin{aligned}
mx\ [\,] &= 0 \\
mx\ s{:}w &= \downarrow/\ x*\ inits\ s{:}w \\
&= \{inits\ x{:}y = [\,]{:}\ (x{:})*\ inits\ y\} \\
&\quad \downarrow/\ x*\ [\,]{:}\ (s{:})*\ inits\ w \\
&= \{wegens\ f*\ x{:}y = (f\ x){:}\ f*y\} \\
&\quad \downarrow/\ (x\,[\,]){:}\ x*\ (s{:})*\ inits\ w \\
&= \{\downarrow/\,x{:}y = x\downarrow\downarrow/y \ \ en\ \ x\ s{:} = \{def\ x\}\ (dx\ s)+x\} \\
&\quad 0\downarrow\downarrow/\ ((dx\ s)+x)*\ inits\ w \\
&= \{x+y = (x+)\ y \ \ en\ \ (f\ g)* = f*g*\} \\
&\quad 0\downarrow\downarrow/\ ((dx\ s)+)*\ x*\ inits\ w \\
&= \{\downarrow/f* = f\downarrow/ \ \ voor\ monotone\ f\} \\
&\quad 0\downarrow (dx\ s)+\ \downarrow/\ x*\ inits\ w \\
&= \{definitie\ mx\} \\
&\quad 0\downarrow (dx\ s)+\ mx\ w
\end{aligned}
$$

Analoog voor my, Mx en My. Dus nu zijn we er in geslaagd een inductieve definitie voor t (en mx, Mx, my en My) te vinden, ja zelfs af te leiden uit de formele specificatie. Er rest ons nog om hetzelfde te doen voor T. Dat blijkt enigszins problematisch: zo'n poging zal niet lukken tenzij je, geïnspireerd door de formules of door inzicht in het probleem, vier hulpfuncties TN, TO, TZ en TW introduceert, met TN w = laatste tijdstip op Noordelijke rand van w, etc. Deze vier functies kunnen zelf wel eenvoudig inductief uitgedrukt worden, volgens een soortgelijke afleiding als we met t hebben gedaan. Kortheidshalve laten we dat hier achterwege en beschouwen we T in het geheel niet meer.

Ten aanzien van bovenstaande afleidingen kunnen we nu dezelfde opmerkingen maken als bij de transformatie van $a^2 - b^2$ tot $(a+b)(a-b)$:

1. Voor wie de notatie niet kent, is dit goochelarij.
2. Vaardigheid in de notatie is nodig, en dat vereist oefening.
3. Soms is inzicht in het probleem nodig om de afleiding te sturen; sommige stappen gaan echter vanzelf.
4. De uiteindelijk verkregen definitie lijkt wel efficienter, maar de efficientie is niet per stap verbeterd of de leidraad geweest.
6. In een Pascal-notatie was de afleiding ondoenlijk geweest.
5. Dank zij de syntactische ontledingsregels zijn er weinig haakjes nodig en blijven de formules leesbaar.

7. We zijn in zeer kleine stappen te werk gegaan. Bij Transformationeel Programmeren in de praktijk zullen veel stappen samengenomen of overgeslagen worden. Daarenboven merken we nog op dat de notatie zich leent voor algebraïsche manipulatie: de meeste transformaties berusten op heel algemeen toepasbare algebraïsche wetten zoals $f* g* = (f\ g)*$ en $p \lhd f* = f*(p\ f)\lhd$. Dit zijn gelijkheden tussen functies. De functies zijn niet alleen op lijsten van toepassing, maar ook op verzamelingen en zgn. bags; zie [Meertens 1986] voor een uniforme behandeling van deze datastructurering.

Aldus hebben we vijf inductief gedefinieerde functies verkregen: t, mx, Mx, my, My (T beschouwen we niet meer). In plaats van dit vijftal functies kunnen we ook één inductief gedefinieerde functie f geven met een vijftal als resultaat, zó dat

$$f\ w = (t\ w,\ mx\ w,\ Mx\ w,\ my\ w,\ My\ w)$$

We volgen hiermee de zgn. Tupling-strategie. Ieder van de vijf functies is op zichzelf lineair in de lengte van w, en het zal blijken dat hun combinatie tot f dat ook is (terwijl hun combinatie als vijftal dat niet is). Zie [Pettorossi 1984] voor een uitvoerige behandeling van de Tupling-strategie. De definitie van f leiden we nu als volgt af.

```
f []  = {volgens wens}
        ( t [], mx [], Mx [], my [], My [])
      = {definities t, mx, Mx, my, My}
        (0, 0, 0, 0, 0)
f s:w = {volgens wens}
        (t s:w, mx s:w, Mx s:w, my s:w, My s:w)
      = {definities t, mx, Mx, my, My}
        (tsw, mxsw, Mxsw, mysw, Mysw)

         where
         tsw  = 0  if ¬(mxsw<0<Mxsw & mysw<0<Mysw)
              = 1 + t w    otherwise
         mxsw = (mx w) ↓ (dx s) + mx w
         mysw, Mxsw, Mysw  analoog
      = {ind.hyp: f w = (t w, mx w, ---, ---, ---)}
        (tsw, mxsw, Mxsw, mysw, Mysw)
         where
         tsw  = 0  if ¬(mxsw<0<Mxsw & mysw<0<Mysw)
              = 1+tw     otherwise
```

$$\begin{aligned} mxsw &= mxw \downarrow (dx\ s) + mxw \\ &mysw,\ Mysw,\ Mxsw \text{ analoog} \\ &(tw,\ mxw,\ Mxw,\ myw,\ Myw) = f\ w \end{aligned}$$

In feite staat hier een functie-definitie van de vorm

$$\begin{aligned} f\ [\,] &= a0 \\ f\ s{:}w &= h\ (s,\ f\ w) \end{aligned}$$

waarbij h de functie is die uit het vijftal $(tw, mxw, ---)$ het vijftal $(tsw, mxsw, ---)$ bepaalt. Middels een standaardtechniek maken we van deze lineair recursieve definitie een iteratieve (= "staart-recursieve"). Dat gaat als volgt. Er geldt $f[s1, s2, \ldots, sn] = h(s1, h(s2, \ldots, h(sn, a0) \ldots)$ zodat we het resultaat ook "van binnen naar buiten" kunnen opbouwen. Daartoe zullen we een g definieren zo dat

$$g\ (w',\ f\ w'') = f\ (w' \mathbin{+\!+} w'') \qquad \ldots (*)$$

Hier is de afleiding van die definitie.

$g\ ([\,],\ a)$
$\quad = \{$ dus $w' = [\,]$ en $a = f\ w''$ in $(*)\ \}$

$$
\begin{array}{ll}
 & f\ [\,]{+}{+}w'' \\
= & \{\text{wegens } [\,]{+}{+}w'' = w'' \text{ en } a = f\ w''\} \\
 & a
\end{array}
$$

$$
\begin{array}{ll}
g\ (w{+}{+}[s],\ a) & \\
= & \{\text{dus } w' = w{+}{+}[s] \text{ en } a = f\ w'' \text{ in } (*)\} \\
 & f\ (w{+}{+}[s]{+}{+}w'') \\
= & \{\text{associativiteit van } {+}{+}\} \\
 & f\ (w{+}{+}([s]{+}{+}w'')) \\
= & \{\text{ind. hyp. } (*)\} \\
 & g\ (w,\ f([s]{+}{+}w'')) \\
= & \{\text{volgens definitie van } f\} \\
 & g\ (w,\ h\ (s,\ f\ w'')) \\
= & \{\text{volgens aanname } a = f\ w'' \text{ enige regels terug}\} \\
 & g\ (w,\ h(s,\ a))
\end{array}
$$

pag 26a

Wie deze transformatie ~~afleiding~~ eenmaal gezien en begrepen heeft, zal verder weinig behoefte voelen om die ~~transformatie~~ ooit nog eens af te leiden zoals we hierboven hebben gedaan. Integendeel, zo'n transformatie behoort al gauw tot het standaardpakket. Merk voorts op dat de wandeling w door g van achter naar voor wordt verwerkt. Hadden we een iteratief algoritme willen hebben dat de wandeling van voor naar achter verwerkt, dan

[Invoegen op pag 26]

En dus geldt nu voor f:

$$
\begin{aligned}
f\ w \ &= \{\text{volgens } (*) \text{ met } w' = w \text{ en } w'' = [\,]\} \\
&\quad g\ (w,\ f\,[\,]) \\
&= \{\text{volgens def. } f\,[\,]\} \\
&\quad g\ (w,\ (0,0,0,0,0))
\end{aligned}
$$

hadden we moeten beginnen met de functies t, mx, ... met inductie te definieren voor de gevallen $[\,]$ en $w{+}{+}[s]$.

De vertaling naar (pseudo-)Pascal ligt nu voor de hand en geeft geen enkel probleem. De iteratief gedefinieerde g wordt nu een iteratie (vandaar de karakterisering "iteratief" voor de definitievorm van g), waarbij variabelen de rol van de parameters overnemen. Bovendien kiezen we er maar gelijk voor om een deelwandeling te representeren door een index i in een vast array w van stappen. Hier is het programma.

```
fct f' (w: array of Stap): int;
    var  i:  0.. #w;
              tw, mxw, Mxw, myw, Myw:  int;
begin
    i:=#w;  (tw, mxw, Mxw, myw, Myw):=(0,0,0,0,0);
    while  i ≠ 0
    do     (tw, mxw, Mxw, myw, Myw):=
                      h (w[i], tw, mxw, Mxw, myw, Myw);
              i:=i-1
    od;
    f' := tw
```

end

De vertaling levert desgewenst ook een invariant waarmee direct is aan te tonen dat dit programma correct is, (althans: even correct als de functies f en g):

$$(tw, mew, Mew, myw, Myw) = f(w[i+1..\#w])$$

oftewel

$$g(w[1..i], (tw, mew, Mew, myw, Myw)) = f(w[1..\#w])$$

In de eerste formulering staat dat de variabelen de gewenste informatie bevatten over het alreeds verwerkte deel van de wandeling; in de tweede formulering staat dat de variabelen voldoende informatie bevatten om te samen met het nog niet verwerkte deel van de wandeling het gewenste eindresultaat te bepalen.

A5. Conclusie

Aldus hebben we voor één specifiek probleem laten zien dat de formele specificatie een hulpmiddel kan zijn bij het ontwerpproces van het gewenste programma. Opdat de geschetste methode praktisch doenlijk is, is het nodig dat de notatie beknopt is en zich leent voor algebraïsche manipulatie. Het behandelde voorbeeld is heel klein-schalig van aard; of deze aanpak ook opgaat bij groot-schalige programmatuur valt nog te bezien.

DEEL B: BACKTRACKING

In dit deel behandelen we de specificatie en programma-ontwikkeling van typische "backtracking" problemen, met name het bekende Acht Koninginnen Probleem. Net als in deel A willen we aantonen dat uit de specificatie een programma afgeleid kan worden en dat de praktische uitvoerbaarheid van deze methode zekere eisen oplegt aan de specificatietaal. In dit deel vestigen we de aandacht op het naast elkaar bestaan van sets, bags en lists. (set = verzameling; bag = zak = een "lijst" waarin de volgorde niet terzake doet, ofwel een "set" waarin de elementen meermalen kunnen voorkomen; list = lijst). Onze behandeling hiervan is deels geïnspireerd door Meertens' werk [Meertens 1986], maar omwille van de eenvoud in presentatie kiezen we voor een conventionelere notatie.

B1. De notatie

Om overspecificatie te voorkomen, en daardoor meer vrijheid te hebben bij de programma-ontwikkeling, is het nodig om naast lijsten ook bags en sets te gebruiken. Geïnspireerd door

[Meertens 1986] kiezen we zoveel mogelijk identieke notaties voor lists, bags en sets. ~~Met name~~ We gebruiken structuur als verzamelnaam voor lists, bags en sets. Alleen de volgende operaties hebben we nodig:

: voor het toevoegen van een element aan een structuur (vóóraan bij lijsten)

voor de lengte/cardinaliteit van een structuur

Expliciete opsommingen geven we aan met de lijsthaken [ ], de baghaken ⦃ ⦄ en de sethaken { }. Dus voor verschillende a, b en c geldt:

$$[a, b, a, c] \neq [a, a, b, c] \neq [a, b, c]$$
$$⦃a, b, a, c⦄ = ⦃a, a, b, c⦄ \neq ⦃a, b, c⦄$$
$$\{a, b, a, c\} = \{a, a, b, c\} = \{a, b, c\}$$

Tenslotte kennen we ook nog de structuur-abstracties, uitdrukkingen van de volgende vorm:

$$[expr \mid x \leftarrow X,\ cond,\ x' \leftarrow X',\ cond']$$

en net zo met ⦃ ⦄- en { }-haken. De resultaatstructuur wordt gevormd door x de structuur X te laten doorlopen (bij lijsten de volgorde in acht nemend) en, bij iedere x die aan de conditie cond voldoet, x' de structuur X' te laten doorlopen en, bij iedere x' die aan cond' voldoet, de ~~expressie~~ waarde expr in de resultaatstructuur te plaatsen. Bij een lijst-abstractie moeten X en X' lijsten zijn, bij een bag-abstractie moeten zij bags zijn en bij een set-abstractie sets. Bijvoorbeeld,

$$
\begin{aligned}
& [(x,y) \mid x \leftarrow [1..3],\ y \leftarrow [1..x],\ x+y \neq 3] \\
= \ & [(1,1),\ (2,2),\ (3,1),\ (3,2),\ (3,3)]
\end{aligned}
$$

We gebruiken ⦃ en ⦄ als "structuur-generieke" haken, d.w.z. de haken ⦃⦄ staan voor één van $[\,]$, $\{\!\!\{\ \}\!\!\}$ of $\{\}$ al naar gelang de kontekst dat bepaalt. Dus na de definitie

$$
\begin{aligned}
\text{count}\,(⦃⦄) &= 0 \\
\text{count}\,(x{:}Y) &= 1 + \text{count}(Y)
\end{aligned}
$$

geldt $\text{count}(X) = \#X$ voor elke lijst, bag of set X. En na

$$
\begin{aligned}
\text{set}\,(⦃⦄) &= \{\} \\
\text{set}\,(x{:}Y) &= x{:}Y
\end{aligned}
$$

geldt dat set de voor de hand liggende conversie van structuren naar sets is. En na

$$\text{any } (x : X) = x$$

levert any de kop van een lijst en niet-deterministisch een of ander element van een bag of set. We gaan in dit verhaal niet in op de mogelijke problemen die niet-determisme met zich meebrengt; wij zullen de "functie" any slechts op het buitenste nivo, als allerlaatste aktie, gebruiken.

B2. De specificatie

We beschouwen voortdurend het bekend verondersteldе Acht Koninginnen Probleem, zie ook [Wirth 1971, 1976] en geven waar dat nodig is aanwijzingen voor de veralgemening tot "typische backtracking problemen".

We representeren een koningin op het schaakbord door een tweetal (i,j) waarbij i het rij-nummer is en j het kolomnummer; $1 \le i \le 8$ en $1 \le j \le 8$. Een stel koninginnen representeren we door een bag $\{\!\{ \ldots, (i,j), \ldots \}\!\}$ van koninginnen. De probleemstelling luidt nu formeel: lever S op of any (S), waarbij

$$
\begin{aligned}
S &= \{ s \mid s \leftarrow S_8,\ legal(s) \} \\
\text{-- } S_n &= \text{alle plaatsingen van } n \text{ koninginnen} \\
S_0 &= \{ \{\!\{\}\!\} \} \\
S_n &= \{ c:s \mid c \leftarrow [\![1..8]\!] \times [\![1..8]\!],\ s \leftarrow S_{n-1} \}
\end{aligned}
$$

Hierbij is legal(s) de test of alle koninginnen in s elkaar niet slaan. Deze functie kunnen we als volgt definieren.

$$
\begin{aligned}
&legal\ (c:s) = veilig\ (c, s)\ \&\ legal\ (s) \\
&legal\ ([\![\,]\!]) = True \\
&veilig\ (c, c':s) = slaatniet\ (c, c')\ \&\ veilig\ (c, s) \\
&veilig\ (c, [\![\,]\!]) = True \\
&slaatniet\ ((i,j), (i',j')) = i \neq i'\ \&\ j \neq j'\ \&\ |i-i'| \neq |j-j'|
\end{aligned}
$$

Hiermee is de specificatie voltooid. In beginsel is de specificatie executeerbaar. Maar aangezien $\# S_8 = 64^8$, zal zelfs wanneer alleen any(S) gevraagd wordt, de benodigde rekentijd onacceptabel groot zijn: veel meer dan duizend jaar.

We veralgemenen nu bovenstaande definities tot het algemene geval. Zij S te definieren door

$$S = \{ s \mid s \leftarrow S_N,\ legal(s) \}$$
$$S_0 = \text{---}$$
$$S_n = \{ cons_n(c, s) \mid c \leftarrow C_n,\ s \leftarrow S_{n-1} \}$$

voor een of andere test legal, constructors $cons_n$ †.
De opgave om S of any(S) te berekenen is dan per definitie een "backtracking" probleem. Merk op dat de kwalificatie "backtracking" in feite zinloos is voor het probleem en voor bovenstaande definities! Die kwalificatie is alleen van toepassing op de berekeningen die door programmatelisten worden opgeroepen. Men kan aantonen dat bij zgn. lazy evaluation de opgeroepen berekeningen van bovenstaand programma (en de nog af te leiden programma's) het schema van backtracking volgen.

---

† en choice-verzamelingen C_n.

B3. Van specificatie naar implementatie

We pogen nu de benodigde rekentijd aanzienlijk te verminderen door de te onderzoeken sets S_n aanzienlijk te verkleinen tot, zeg, S'_n. Dat gaat als volgt. Zoek geschikte tests $legal_n$ die een noodzakelijke voorwaarde zijn opdat uiteindelijk legal vervuld kan zijn; er moet dus gelden:

$$legal(s) \Rightarrow legal_N(s)$$
$$legal_n(c:s) \Rightarrow legal_{n-1}(s)$$

Zij in het bijzonder acc_n een acceptabiliteits-test, zó dat

$$legal_n(c:s) = legal_{n-1}(s) \;\&\; acc_n(c,s)$$

Dan kunnen we S ook definieren m.b.v. sets S'_n die voldoen aan $S'_n = \{s \mid s \leftarrow S_n, legal_n(s)\}$, terwijl we S'_n ook direkt in termen van S'_{n-1} kunnen definieren (en zodoende alle grote verzamelingen S_n vermijden). Inderdaad,

$$\begin{aligned} S &= \{s \mid s \leftarrow S_N, legal(s)\} \\ &= \{s \mid s \leftarrow S_N, legal'(s) \;\&\; legal_N(s)\} \end{aligned}$$

$$
\begin{aligned}
&= \{s \mid s \leftarrow S'_N,\ legal'(s)\} \\
S'_0 &= \ldots \\
S'_n &= \{s \mid s \leftarrow S_n,\ legal_n(s)\} \\
&= \{c{:}s \mid c \leftarrow C_n,\ s \leftarrow S_{n-1},\ legal_n(c{:}s)\} \\
&= \{c{:}s \mid c \leftarrow C_n,\ s \leftarrow S_{n-1},\ legal_{n-1}(s)\ \&\ acc_n(c,s)\} \\
&= \{c{:}s \mid c \leftarrow C_n,\ s \leftarrow S'_{n-1},\ acc_n(c,s)\}
\end{aligned}
$$

Bij het Acht Koninginnen Probleem is bijvoorbeeld:

$$
\begin{aligned}
legal_n &= legal \\
acc_n &= veilig
\end{aligned}
$$

zoals direkt uit de gegeven definitie voor legal in de specificatie valt af te lezen.

Bovenstaande techniek kan wellicht verscheidene malen worden toegepast. Maar daarnaast zijn er ook andere optimalisaties mogelijk. Beschouw weer het Acht Koninginnen Probleem. Uiteindelijk komt er op iedere rij precies één koningin. Omdat een plaatsing van koninginnen een bag is, doet de volgorde niet ter zake. Dus we kunnen nu bijvoorbeeld afdwingen dat de n-de koningin rij-nummer n heeft. Dit geeft nu als definitie:

$$S = S''_8$$
$$S''_0 = \{\{\!\!\{\ \}\!\!\}\}$$
$$S''_n = \{(n,j):s \mid j \leftarrow \{\!\!\{1..8\}\!\!\},\ s \leftarrow S''_{n-1},\ \text{veilig}((n,j),s)\}$$

De grootte-orde van de algoritme is hiermee bevredigend teruggebracht. Maar de implementatie kan nog aanzienlijk versneld worden. Immers, bij sets moet in de implementatie vroeg of laat getest worden of er in de representatie elementen dubbel voorkomen. Echter, wij zien dat er door de generatoren $j \leftarrow \{\!\!\{1..8\}\!\!\}$ en $s \leftarrow S''_{n-1}$ geen elementen $(n,j):s$ dubbel worden voortgebracht. De test op dubbele voorkomens omzeilen we nu door zelf de representatie in de hand te nemen: in plaats van sets S''_n nemen we nu lijsten L_n (die een opsomming zonder duplicaten zijn van de S''_n).

$$L = L_8$$
$$L_0 = [\{\!\!\{\ \}\!\!\}]$$
$$L_n = [(n,j):s \mid j \leftarrow \{\!\!\{1..8\}\!\!\},\ s \leftarrow L_{n-1},\ \text{veilig}((n,j),s)\}$$

Nu is ook de volgorde van het generatie-proces vastgelegd: bij iedere j ~~[illegible]~~ wordt geheel L_{n-1} doorlopen. Wanneer niet geheel L wordt gevraagd, maar slechts any(L), dan kan dat generatieproces beter geformuleerd worden als $s \leftarrow L_{n-1}, j \leftarrow \{1..8\}$. Immers, de generatie van een volgende j kost niet veel rekentijd, terwijl de generatie van een volgende s uit L_{n-1} het gedeeltelijk doorlopen van L_{n-2} (én L_{n-3} enz.) vereist en dus véél duurder is dan de generatie van een volgende j. Wanneer niet alle paren (s, j) nodig zijn, geeft $s \leftarrow L_{n-1}, j \leftarrow \{1..8\}$ een besparing op de behoefte aan s'en, terwijl met de omgekeerde volgorde juist bespaard wordt op de behoefte aan j's.

Hiermee is de ontwikkeling van een efficient algoritme voltooid. Er rest ons nog het algoritme in een Pascal-achtige vorm te coderen.

B4. Codering in Pascal

We willen nu het verkregen algoritme zo getrouw mogelijk in Pascal omzetten. Daarbij nemen we als uitgangspunt dat er één globaal array s is waarin de gegenereerde plaatsingen worden opgeslagen, totdat zij worden afgedrukt. Ten gevolge van dit uitgangspunt is het niet eenvoudig de definities van L en Ln naar Pascal om te zetten. We zullen straks uit L en Ln nog andere definities L' en L'n afleiden, die ~~een gemakkelijker vertaling naar Pascal~~ makkelijker naar Pascal te vertalen zijn.

We geven allereerst, in Figuur 1, een vertaling van de definities van L en Ln naar Pascal-met-coroutines. Procedure L heeft tot taak alle oplossingen af te drukken. Coroutine L(n) heeft tot taak om (met behulp van coroutine L(n-1)) alle oplossingen van n koninginnen in array s[1..n] te genereren. Zodra een volgende oplossing gereed is, geeft de coroutine de besturing weer terug aan de aanroeper; de aanroeper kan hem later weer doorstarten vanaf het punt waar hij bij de laatste return gebleven was. Merk op dat de structuur van de definities goed behouden blijft onder de vertaling naar Pascal.

```
procedure L;
  {drukt alle oplossingen af}
  var s: array [1..8] of 1..8;
  function veilig (n,j: integer): boolean;
      {veilig := (n,j) is veilig voor s[1..n-1]}
      ---
  coroutine L (n: integer);
      {genereert álle oplossingen met n
       koninginnen, in s[1..n]}
      var j: 1..8;
   begin
      if n=0
      then return
      else for j := 1 to 8 do
               for each return of L(n-1) do
                  if veilig (n,j) then
                  begin s[n] := j;
                        return
    end           end;
begin
   for each return of L(8) do write (s[1..8])
end
```

Figuur 1

In plaats van coroutines kunnen we ook procedures-als-parameters gebruiken. Coroutine L(n) wordt dan een gewone procedure, maar de bewerkingen die nog met alle resultaten van de aanroep van L(n-1) moeten worden uitgevoerd, worden dan als procedure-argument aan L(n-1) meegegeven. En zo heeft L(n) zelf ook een procedure-parameter die nog op al zijn resultaten moet worden toegepast. Aldus verkrijgen we het programma van Figuur 2.

```
procedure L;
   {drukt alle oplossingen af}
   var s: array [1..8] of 1..8;
   function veilig (n,j: integer): boolean;
      {veilig := (n,j) is veilig voor s[1..n-1]}
      ---
   procedure L (n: integer; procedure p);
      {genereert alle oplossingen in s[1..n]
       en ieder wordt onderworpen aan p}
      procedure nextp;
         {de eigenlijke akties van coroutine L(n)}
         var j: 1..8;
      begin
         for j := 1 to 8 do
            if veilig (n,j) then
            begin  s[n] := j;
                      p
            end
      end nextp;
   begin
      if n=0
      then p
      else L (n-1, nextp)
   end;
begin L (8, write_s) end
```

Figuur 2

Voor de derde en laatste implementatie in Pascal, transformeren we eerst nog eens de definities van de L_n. Merk op dat de lijsten L_n elementsgewijze lineair recursief zijn. We kunnen deze omvormen tot elementsgewijze iteratieve definities op een manier die precies overeenkomt met de omvorming van lineaire recursie naar iteratie (zoals de transformatie van f naar g in deel A). Dit gaat als volgt. Stel je ten doel een definitie voor $L'_n(s)$ te ontwikkelen, zó dat voor alle n

$$L_8 = [s' \mid s \leftarrow L_n ;\ s' \leftarrow L'_{n+1}(s)] \qquad \ldots (*)$$

Met andere woorden: $L'_{n+1}(s)$ breidt iedere s uit L_n uit tot oplossingen s' in L_8. Voor de constructie van definities voor de L'_n gaan we nu met inductie naar $8-n$ te werk. Voor $n=8$ vinden we uit (*) dat we kunnen definiëren:

$$L'_9(s) = [s]$$

Voor $n<8$ redeneren we als volgt:

$[s' \mid s \leftarrow L_{n-1};\ \ s' \leftarrow L'_n(s)]$

$= \{$volgens wens $(*)\}$

L_8

$= \{$per inductie hypothese$\}$

$[s' \mid s \leftarrow L_n;\ \ s' \leftarrow L'_{n+1}(s)]$

$= \{$def $L_n\}$

$[s' \mid s \leftarrow L_{n-1};\ j \leftarrow [1..8];\ \text{veilig}((n,j), s);\ s' \leftarrow L'_{n+1}((n,j):s)]$

En uit de eerste en laatste regel volgt nu een suggestie voor de definitie van $L'_n(s)$:

$$L'_n(s) = [s' \mid j \leftarrow [1..8];\ \text{veilig}((n,j):s);\ s' \leftarrow L'_{n+1}((n,j):s)]$$

† →

Inderdaad is L'_n nu elementsgewijze iteratief (= tail recursive). De omzetting naar Pascal geeft geen enkel probleem. Procedure $L'(n)$ vindt in een globaal array zijn s-argument $s[1..n-1]$ klaar staan en drukt alle uitbreidingen $s[1..8]$ die een oplossing zijn, af. Zie Figuur 3.

---

† Volgens $(*)$ kunnen we L nu definieren als:

$$L = L'_1(\{\})$$

```
procedure L;
   {drukt alle oplossingen af}
   var s: array [1..8] of 1..8;
   function veilig (n, j: integer): boolean;
      {veilig := (n, j) is veilig voor s[1..n-1]}
      ---
   procedure L' (n: integer);
      {vult oplossing s[1..n-1] aan tot oplossingen
       s[1..8] en drukt ze af}
      var j: 1..8;
   begin
      if n = 9
      then write (s[1..8])
      else for j := 1 to 8 do
                  if veilig (n, j) then
                  begin s[n] := j;
                        L' (n+1)
    end               end;
begin
   L' (1)
end
```

Figuur 3

De elementsgewijze iteratieve definitie blijkt dus eenvoudiger naar Pascal om te zetten. Die definitie-vorm is ook nodig voor Branch-and-Bound problemen; dat zijn problemen waarbij S net zo te definieren is als bij Backtracking, ~~maar~~ en niet geheel S of any(S) wordt gevraagd maar een "optimaal" element van S; zie [Fokkinga 1986].

B5. Conclusie

We hebben het gehele traject van Specificatie tot Implementatie afgelegd voor een typisch backtracking probleem. Zowel het gebruik van bags is uitgebuit (bij de transformatie van S'_n naar S''_n), alsook het gebruik van sets (bij de overgang van sets op lijsten). Hadden we ons te vroeg op lijsten vastgelegd, dan waren sommige transformaties niet mogelijk geweest of in ieder geval wel bemoeilijkt.

De transformaties waren sowieso ondoenlijk geweest als we uitsluitend een Pascal-achtige notatie gebruikt zouden hebben: de

Pascal-teksten zijn al gauw ~~te~~ veel te groot en lenen zich bovendien nauwelijks voor algebraïsche manipulatie. Sommige transformatiestappen kunnen nog preciezer gerechtvaardigd worden in de notatie van deel A, zie [Fokkinga 1987].

LITERATUUR VERWIJZINGEN

Amstel, J.J. van: Specificaties - wie, wat, hoe? In [Poirters en Schoenmakers 1987].

Bird, R.S.: An Introduction to the Theory of Lists. Technical Monograph PRG-56 ~~, Oct 1984~~, Oxford University, Oct 1986.

EXIN: Examenopgaven 1985 voor de module P1 (programmeren en datastructuren) van het AMBI examen. Amsterdam, 1985.

Fokkinga, M.M.: Backtracking and Branch-and-Bound Functionally Expressed. Memorandum INF-86-18, Universiteit Twente, 1986.

Fokkinga, M.M.: in voorbereiding, 1987.

Meertens, L.: Algorithmics - towards programming as a mathematical activety. Proc. CWI Symp. on Mathematics and Computer Science, CWI Monographs Vol. 1 (eds. J.W. de Bakker, M. Hazewinkel, J.K. Lenstra), North-Holland, 1986, pp. 289-234.

Pettorossi, A.: Methodologies for Transformations and Memoing in Applicative Languages. Thesis CST-29-84, University of Edinburgh, 1984.

Poirters, J.A.A.M. & Schoenmakers, G.J. (eds): Software Specificatie Technieken.

Turner, D.A.: Miranda - a non-strict Functional Language with Polymorphic Types. In Functional Programming Languages and Computer Architecture, (ed. J.P. Jouannaud). Springer, LNCS Vol. 201, 1985, pp. 1-16.

Wirth, N.: Program Development by Stepwise Refinement. Comm. ACM, Vol. 14, 1971, No. 4, pp. 221-227.

Wirth, N.: Algorithms + Data Structures = Programs. Prentice-Hall, 1976, Ch. 3.4 and 3.5.