Katholieke
Universiteit
Leuven

Fakulteit der Toegepaste
Wetenschappen
Afd. Toegepaste Wiskunde
en Programmatie

NAAR EEN BETERE BEHEERSING VAN DE UITVOERING VAN PROGRAMMA'S IN DE LOGIKA DER HORN-UITDRUKKINGEN

Maurice BRUYNOOGHE

Proefschrift aangeboden tot het behalen van de graad van doctor in de toegepaste wetenschappen

Mei 1979

Promotor : Prof. dr. ir. Y. D. WILLEMS
Co-promotor : Prof. ir. M. GOBIN

Aan Lieve,

zonder haar zou dit proefschrift

of vroeger

of nooit

tot stand gekomen zijn

VOORWOORD

Bij het voltooien van dit proefschrift is het passend om even terug te blikken. Wat we geworden zijn hebben we niet uitsluitend aan eigen verdienste te danken. De eigen verdienste beperkt er zich toe de geschonken talenten gebruikt en de aangeboden kansen benut te hebben. Onze dank gaat dan ook uit naar allen die de verwezenlijking van dit proefschrift mogelijk gemaakt hebben.

In de eerste plaats wil ik mijn ouders danken. Ondanks bescheiden financiële mogelijkheden gaven zij mij de kans om te studeren en zorgden zij voor een atmosfeer waarin de studie kon gedijen. Na het doorlopen van het middelbaar technisch onderwijs gaven zij me de kans om eerst het diploma van technisch ingenieur en vervolgens het diploma van burgerlijk ingenieur te behalen. Dit was wellicht slechts mogelijk dank zij de "democratisering van het onderwijs". Het stelsel der studiebeurzen, alsook andere sociale voorzieningen maakten voor mijn ouders de financiële lasten beter draagbaar. Het is dan ook billijk om allen te danken die deze sociale voorzieningen mogelijk maken, in het bijzonder de velen die plichtsgetrouw de staat geven wat, volgens de geldende belastingswetten, de staat toekomt.

Ook wil ik onder mijn leraren en professoren diegenen danken die meer bekommerd waren om het inzicht dat zij bijbrachten dan om de hoeveelheid kennis die ze mededeelden, en dit ondanks het feit dat het laatste veel gemakkelijker meetbaar is dan het eerste.

Voor de vijf jaren die aan dit onderzoek besteed werden gaat mijn dank uit naar :

- Het Nationaal Fonds voor Wetenschappelijk Onderzoek dat mij de nodige onderzoeksmandaten (navorsingsstagiair en aspirant) ter beschikking stelde.
- De professoren en promotoren Y.D. Willems en M. Gobin voor de initiatie in dit onderzoeksdomein, de aanmoediging om buitenlandse studiereizen te ondernemen en de steun in de loop van het onderzoek.
- De Afdeling Toegepaste Wiskunde en Programmatie, Het Ministerie van Nationale Opvoeding en Nederlandse Kultuur, Het Franse Centre National de la Recherche Scientifique, en Vlaamse Leergangen voor de financiële steunen bij het ondernemen van buitenlandse studiereizen en het bijwonen van congressen.

- Robert Kowalski, de grondlegger van de "logic programming"-gemeenschap, op wiens kennis, steun, aanmoediging en gastvrijheid wij steeds mochten rekenen. Zonder hem was dit onderzoek niet mogelijk geweest.
- De vele buitenlandse onderzoekers die wij mochten ontmoeten en wiens enthousiasme aanstekelijk werkte, o.a. Alain Colmerauer, Philippe Roussel, David Warren, Keith Clark, Sten-Ake Tarnlund, Luis Pereira, Maarten van Emden, Peter Szeredi,...
- De professoren uit het leescomité : J. Lewi, M. Gobin en Y.D. Willems voor hun hulp bij het opstellen van de definitieve tekst.
- In het bijzonder naar professor Y.D. Willems die ondanks een drukke agenda uiteindelijk toch tijdig de tijd vond om de tekst grondig en kritisch na te lezen en wiens opmerkingen en vragen tot belangrijke verbeteringen leidden.
- Alle leden van de Afdeling Toegepaste Wiskunde en Programmatie voor de prettige werkatmosfeer.
- Mevrouw Maggie Van Dooren, zij heeft zich ijverig doorheen het bij tijd en wijle erg slordige manuscript geworsteld en heeft met haar vaardige vingers alles heel netjes en ordentelijk op het papier getoverd.

INHOUDSTAFEL

Voorwoord i

Inhoudstafel iii

INLEIDING 0.1

HOOFDSTUK I : LOGIKA ALS PROGRAMMEERTAAL I.1

A. Logika en programmatie - ontwikkeling van de logika als een programmeertaal I.1

B. Horn-uitdrukkingen als programma's I.8

B.1. Een programmeertaal I.8

B.2. Gebaseerd op logika I.14

B.3. PROLOG I.26

B.3.1. Zuivere PROLOG I.26

B.3.2. Uitbreidingen I.28

HOOFDSTUK II : PROLOG-VERTOLKERS II.1

A. Inleiding II.1

B. Het principe van de oorspronkelijke PROLOG-vertolkers II.3

C. Een alternatief implementatieschema II.9

D. Een eenvoudig en algemeen algoritme II.11

E. Een belangrijke tekortkoming : buitensporig geheugengebruik II.14

F. De kopieermethode II.17

F.1. Principe van de geheugenbesparing II.17

F.2. Geheugenbesparende technieken II.18

F.3. Het ontdekken van cyclische strukturen ("occurcheck") II.23

F.4. Algoritme voor de kopieermethode II.25

F.5. "Garbage collection" II.30

G. Warren zijn oplossing : een gewijzigde gedeelde strukturenmethode II.31

H. Een vergelijking van de geheugenbesparingen verwezenlijkt door de kopieermethode en de gewijzigde gedeelde struktuurmethode II.33

HOOFDSTUK III : HET CONTROLEREN VAN DE SELEKTIE DER PROCEDURE-OPROEPEN III.1
A. Inleiding III.1
B. Voorbeelden III.5
B.1. De zeef van Eratosthenes III.5
B.2. Een sorteeralgoritme III.9
B.3. Een merkwaardige rij III.11
B.4. Meerdere producenten voor eenzelfde variabele III.13
B.5. Het 8-koninginnenprobleem III.14
B.6. Het profiel van binaire bomen III.18
C. Een overzicht van de diverse concepten III.19
D. Nabeschouwingen III.24

HOOFDSTUK IV : "INTELLIGENTE" SEQUENTIËLE EXPLORATIE IV.1
A. Inleiding IV.1
B. Processen als informatiebronnen IV.2
B.1. Een enkel sequentieel proces IV.2
B.2. Onafhankelijke parallelle processen IV.3
B.3. Informatiestromen tussen processen IV.4
B.4. Uitvoering van een proces IV.5
B.5. Afhankelijkheidsrelatie - afhankelijkheidsgraf IV.6
B.6. Een conflict IV.6
B.7. Het beperken van het aantal processen IV.8
B.8. Een familie van algoritmes IV.9
B.9. Algoritme 1 IV.9
B.10. Een eenvoudig voorbeeld IV.11
C. Voorbeelden IV.14
C.1. Koninginnenprobleem - oplossing met permutatie IV.14
C.2. Koninginnenprobleem - oplossing door het uitbreiden van een partiële optelling IV.30
D. Een verdere verfijning IV.34
D.1. Principe IV.34
D.2. Algoritme 2 IV.35
D.3. Voorbeelden IV.37
E. Diskussie IV.48
E.1. De methode van Mackworth IV.48
E.2. De methode van Freuder IV.52
F. Het vinden van alle oplossingen IV.55
G. Nabeschouwingen IV.62

V. BESLUIT V.1

VI. BIBLIOGRAFIE VI.1

INLEIDING

Ons onderzoek over "logic programming", het gebruik van logika als een programmeertaal, is ontstaan uit persoonlijke contacten met Robert Kowalski en Alain Colmerauer. Dit proefschrift vormt de weergave van dit onderzoek.

In het eerste hoofdstuk van dit werk schetsen we het ontstaan en de evolutie in het gebruik van logika als programmeertaal. Tevens geven we de voornaamste aspecten van PROLOG, een op logika gebaseerde taal die door de groep van Alain Colmerauer ontwikkeld werd.

Centraal in die ontwikkeling staat de interpretatie van Horn-uitdrukkingen ("Horn clauses" = logische formules van de vorm $A \leftarrow B_1,\ldots,B_m$ met $m \geqslant 0$) als procedures die toelaten om elementen van de relatie A te berekenen. Het lichaam van een dergelijke procedure bestaat uit de procedure-oproepen $B_1,\ldots,B_m$. De logika zelf laat in het midden of deze oproepen in een bepaalde volgorde, dan wel gelijktijdig moeten uitgevoerd worden. Echter, om tot een efficiënte uitvoering te komen zijn deze beslissingen van cruciaal belang. Het nemen van die beslissingen noemt men het "controleren" van het programma. In PROLOG worden de oproepen de een na de ander uitgevoerd ("sequentieel") en de volgorde waarin de gebruiker de oproepen neerschrijft is meteen ook de volgorde waarin ze uitgevoerd worden.

Wanneer voor een bepaalde procedure-oproep meerdere proceduredefinities beschikbaar zijn, vertakt de berekening zich in meerdere alternatieve richtingen. In PROLOG worden deze alternatieven de een na de ander afgehandeld. Het systeem houdt de nodige informatie bij om de staat van de berekening in de vertakkingspunten te herstellen. Eens een bepaald alternatief volledig afgehandeld, neemt de berekening een nieuwe start vanaf het meest recente vertakkingspunt met nog niet behandelde alternatieven ("sequentiële exploratie" - "backtracking"). Ook hier is de volgorde waarin de gebruiker de verschillende definities neerschrijft de volgorde waarin de verschillende alternatieven afgehandeld worden.

Bij het schrijven van dit inleidend hoofdstuk werden wij vooral geïnspireerd door het werk van Kowalski, van Emden, Hodges en Warren.

In het tweede hoofdstuk bestuderen we de implementatie van PROLOG. Het principe van de oorspronkelijke PROLOG-vertolker wordt uiteengezet, alsook een door de auteur ontwikkelde variante. Een algoritme op zeer hoog niveau beschrijft beide versies.

In de oorspronkelijke vertolker ("gedeelde strukturenmethode") worden de resultaten opgebouwd uit verwijzingen naar stukjes "zuivere code" uit de diverse gebruikte proceduredefinities. In de variante ("kopieermethode") worden de resultaten, zoals in LISP, effektief opgebouwd.

Vervolgens wordt aangeduid dat de kopieermethode zich leent tot een aantal belangrijke geheugenbesparende technieken. Een algoritme, dat deze geheugenbesparingen verwezenlijkt, wordt gegeven.

Tenslotte beschrijven we hoe David Warren erin geslaagd is om, door het verfijnen van de gedeelde strukturenmethode, analoge besparingen te verwezenlijken. Een vergelijking van beide methodes, die gelijktijdig en onafhankelijk van elkaar ontwikkeld werden, wijst uit dat ze erg aan elkaar gewaagd zijn. Onder licht voorbehoud kan men stellen dat de kopieermethode een iets grotere besparing verwezenlijkt.

De twee laatste hoofdstukken behandelen meer theoretisch werk. Ideeën, die kunnen leiden tot de ontwikkeling van een nieuwe generatie vertolkers, worden uiteengezet. De PROLOG-conventie dat de verschillende oproepen $B_1,\ldots,B_m$ de een na de ander worden uitgevoerd, wordt verlaten. We nemen aan dat de verschillende oproepen in principe gelijktijdig kunnen uitgevoerd worden. Met de procedure-oproepen associëren we processen.

Het controleren van het programma bestaat dan in het synchroniseren van de diverse processen. Dit wordt in het derde hoofdstuk bestudeerd. Ons uitgangspunt is de zogenoemde "luie evaluatie" waarbij de uitvoering aangedreven wordt door een verzoek om bepaalde resultaten te berekenen. Slechts de processen die deze gevraagde resultaten kunnen berekenen, worden aktief. Door het opleggen van beperkingen kunnen aktieve processen tot wachten gedwongen worden. Een wachtend proces kan op zijn beurt andere

processen verzoeken de informatie te berekenen die het wachtend proces moet toelaten om aan de opgelegde beperkingen te voldoen. Aan de hand van een aantal voorbeelden worden een reeks concepten ontwikkeld die de gebruiker toelaten op vrij eenvoudige wijze de nodige controle op te leggen. De ontwikkeling van deze concepten, die de logika van de programma's intact laat, gebeurde in nauwe samenwerking met Keith Clark (Queen Mary College, University of London).

In het laatste hoofdstuk wordt een methode ontwikkeld die de sequentiële exploratietechniek verbetert. Er wordt eveneens uitgegaan van de veronderstelling dat de processen die met de verschillende oproepen geassocieerd zijn in principe gelijktijdig en onafhankelijk van elkaar kunnen uitgevoerd worden. Blokkeert een proces en wordt het gedwongen om op zijn stappen terug te keren en een ander alternatief te beschouwen, dan heeft dit geen enkele invloed op de andere processen. Dit in tegenstelling met de oorspronkelijke methode waar alle door de diverse processen uitgevoerde stappen in een welbepaalde sequentie geplaatst worden (totale orde) en waar bij een blokkering naar het meest recente vertakkingspunt teruggekeerd wordt. Het is goed mogelijk dat het meest recente vertakkingspunt behoort tot een proces dat met de blokkering niets te maken heeft. De oorzaak van het blokkeren is niet weggenomen en de berekening is gedoemd om opnieuw te blokkeren.

Er moet echter voor gezorgd worden dat de partiële oplossingen, die van de verschillende processen afkomstig zijn, consistent blijven met elkaar. Men kan stellen dat een centraal proces over deze consistentie waakt. Zodra een inconsistentie optreedt, bepaald dit centrale proces welke processen tot die inconsistentie bijgedragen hebben. Een van deze processen wordt als "schuldige" aangewezen en wordt door de andere betrokken processen gedwongen om op zijn stappen terug te keren en een alternatieve oplossing te berekenen. Deze conflicten veroorzaken dus informatiestromen vanuit de betrokken processen naar het schuldige proces. Dit geeft aanleiding tot een partiële orde over de stappen van de berekening. Wordt nu een proces gedwongen om op zijn stappen terug te keren, dan heeft dit ook een invloed op de processen die van dit proces informatie ontvangen hebben. Zoals uit de tekst zal blijken kan deze invloed verrassend klein gehouden worden.

In het besluit worden de belangrijkste resultaten aangehaald en worden mogelijkheden voor meer toepassingsgericht onderzoek aangeduid.

HOOFDSTUK I : LOGIKA ALS PROGRAMMEERTAAL

A. LOGIKA EN PROGRAMMATIE - ONTWIKKELING VAN DE LOGIKA ALS EEN PROGRAMMEERTAAL

Logika en programmatie vormen op het eerste zicht twee totaal verschillende onderwerpen, nochtans zijn er bepaalde raakpunten.

LOGIKA

Logika bestudeert verzamelingen van beweringen, meer bepaald stelt ze zich de vraag of een verzameling van beweringen al of niet zinvol is, in andere woorden, of er al of niet een situatie mogelijk is waarin alle beweringen van een gegeven verzameling waar zijn. Een zinvolle verzameling van beweringen noemt men consistent, een zinloze verzameling noemt men contradictorisch.

PROGRAMMATIE

Een programma heeft tot doel een bepaald probleem op te lossen. De "kunst" van het programmeren bestaat erin de oplossing van een probleem te formuleren onder een zulkdanige vorm dat het - na een automatische transformatie - door de computer uitvoerbaar is. In andere woorden, de oplossing van het probleem moet geformuleerd worden in een programmeertaal waarvoor een vertolker of een vertaler beschikbaar is.

Het is de hoop van elke programmeur dat zijn programma het gestelde probleem oplost. Bewijzen dat dit inderdaad het geval is, vormt een bijzondere vorm van het consistentieprobleem, nl. de consistentie tussen een programma en de probleemspecifikatie. Dit is een eerste raakpunt tussen logika en programmatie en de meeste methodes voor het bewijzen van de correctheid van programma's maken gebruik van logika.

De term "het formuleren van een oplossing van het probleem" is erg vaag maar misschien nog niet vaag genoeg. Die term suggereert ten onrechte dat de oplossing expliciet in de formulering aanwezig moet zijn. Dit is niet altijd noodzakelijk; afhankelijk van de gebruikte taal

bevindt de formulering zich ergens tussen twee uitersten. Enerzijds een taal waarin men zeer gedetailleerd beschrijft hoe de oplossing moet bekomen worden; maar waarin de probleemspecifikatie totaal onzichtbaar geworden is (machinetaal). Anderzijds een taal waarin men enkel beschrijft aan welke voorwaarden de oplossing moet voldoen zonder uit te weiden over de methode om tot een oplossing te komen (taal van zeer hoog niveau). Een taal van de eerste soort noemt men imperatief, een taal van de tweede soort noemt men deklaratief.
Programmeren in een imperatieve taal vraagt gewoonlijk meer inspanning en tijd van de programmeur dan programmeren in een deklaratieve taal. Anderzijds zijn de machinekosten voor het verwerken en uitvoeren van een deklaratief programma gewoonlijk hoger dan voor een gelijkaardig imperatief programma. Tengevolge van het stijgen der programmeringskosten en het dalen der machinekosten wordt stilaan meer en meer de voorkeur gegeven aan meer deklaratieve talen. Deze trend wordt nog aanzienlijk versterkt door het feit dat men een beter inzicht krijgt in de implementatietechnieken voor deklaratieve talen zodat, zelfs met constante machineprijs, het gebruik van deklaratieve talen goedkoper wordt.

Het gemakkelijkste uitdrukkingsmiddel om een probleem te beschrijven is voor ons, mensen, onze natuurlijke taal. Wellicht is dus onze natuurlijke taal de ideale ultieme deklaratieve programmeertaal. Nochtans is het weining waarschijnlijk dat onze natuurlijke taal, in al haar rijkdom (o.a. dubbelzinnigheid, impliciete veronderstellingen), in de nabije toekomst als programmeertaal kan gebruikt worden. Dit sluit niet uit dat een goedgekozen beperkte deelverzameling van een natuurlijke taal kan gebruikt worden als programmeertaal voor een welbepaald beperkt toepassingsgebied. Lang voordat computers gebouwd werden was er reeds een wetenschap die zich bezig hield met het analyzeren van de beweringen die de mensen maken door middel van hun natuurlijke taal. Deze wetenschap is precies de logika; haar beoefenaars hebben geprobeerd om onze beweringen formeel te beschrijven en te manipuleren. Ze hebben een formeel systeem ontwikkeld dat zo dicht mogelijk bij de natuurlijke taal staat. De vraag rijst of formele logika een geschikte deklaratieve programmeertaal is.

VAN LOGIKA NAAR PROGRAMMATIE

Deze vraag werd in feite reeds heel wat vroeger gesteld; zij het in een totaal andere vorm. Met de opkomst van de computer werd gepoogd om het testen van de consistentie van een verzameling beweringen te automatizeren. Men had vooral interesse in wiskundige toepassingen : bewijzen dat een bepaalde bewering (stelling) het gevolg is van een verzameling andere beweringen (axioma's). Dit onderzoeksdomein is gekend als het automatisch bewijzen van stellingen. ("mechanical theorem proving"). In 1965 introduceerde J.A. Robinson het resolutieprincipe [Robinson 65]. Hij toonde aan dat het herhaald toepassen van één enkele afleidingsregel, door hem resolutie ("resolution") genoemd, volstond om te bewijzen dat een contradictorische verzameling beweringen inderdaad contradictorisch is (De complexiteit van het probleem is zó dat het onmogelijk is om een algoritme te ontwerpen dat voor elke verzameling beweringen stopt met de mededeling dat het een consistente of een contradictorische verzameling is). Het feit dat de methode slechts één afleidingsregel gebruikt, maakt haar bijzonder geschikt voor automatizatie. Hoop op een belangrijke vooruitgang rees onder de navorsers. Een "combinatorische explosie" heeft deze hoop verwoest. Inderdaad, het werd snel duidelijk dat enerzijds het resolutieprincipe veel redundantie bevatte en dat anderzijds een enorme hoeveelheid van beweringen afleidbaar is uit een relatief kleine verzameling van beweringen. De poging om, niet eens zó moeilijke stellingen te bewijzen, liepen vast in de massa der afgeleide beweringen.

Sindsdien werd het resolutieprincipe reeds in belangrijke mate verbeterd door o.a. eliminatie van redundantie en meer doelgerichte generatie van afgeleide beweringen. Meer details hierover kan men vinden in [Chang & Lee 73, Loveland 78].

Al deze verbeteringen ten spijt zijn de huidige automatische stellingbewijzers nog steeds ongeschikt om gebruikt te worden als vertolkers voor logische programma's. Zoals de titel van deze thesis laat vermoeden is het voorbarig om daaruit te besluiten dat logika als programmeertaal afgeschreven is. Inderdaad, niet alleen verfijnde men de resolutiemethode, tevens verwierf men nieuwe inzichten in het resolutieproces.

Boyer en Moore introduceerden in 1972 [Boyer & Moore 72] een methode, "gedeelde strukturen" genoemd ("structure sharing") waarbij een afgeleide bewering impliciet wordt voorgesteld door middel van verwijzingen naar de beweringen waaruit ze is afgeleid. Deze methode leidde tot zeer belangrijke geheugenbesparingen. Maar, wat voor ons belangrijker is, zij suggereerden tevens [Moore 73] dat er een verband zou bestaan tussen het uitvoeren van een resolutiestap in hun systeem en het uitvoeren van een procedure-oproep in een conventionele programmeertaal.

In de Verenigde Staten gebruikte Green [Green 69] een stellingbewijzer als basis voor wat hij een "question-answering system" noemde en zocht toepassingen buiten het enge terrein van het bewijzen van wiskundige stellingen. Wat later werd PLANNER ontwikkeld [Hewitt 72, Sussman & Winograd 71]. In grote lijnen verschilt het PLANNER systeem niet zoveel van een conventionele stellingbewijzer. Nochtans veroordeelden de ontwerpers de tot dan gebruikte aanpak om het stellingbewijzen te beschouwen als een duel tussen de ontwerper en degene die een bepaalde stelling wil laten bewijzen. In deze optiek was de stellingbewijzer een magische doos die op een mechanische, voor de gebruiker onbegrijpelijke wijze, een bewijs te voorschijn tovert. Integendeel, zij zagen PLANNER als een programmeertaal. Ze lieten de gebruiker toe om aan het systeem te zeggen wat het moet doen. Ze gingen daarbij zover dat ze, net zoals in een conventionele programmeertaal, de kennis omtrent het op te lossen probleem en de controleïnformatie, die het systeem naar een oplossing moet leiden, in elkaar verstrengelden. Een gevolg daarvan is dat een programma niet kan beschouwd worden als een verzameling beweringen en een berekening niet als het afleiden van een logische konsekwentie. Het feit dat ze PLANNER als een door de gebruiker gecontroleerd systeem beschouwden, liet de ontwerpers toe een ongewone strategie te gebruiken om de verschillende mogelijke oplossingswegen te onderzoeken. Ze behandelden deze verschillende wegen sequentieel : slechts wanneer een bepaald pad ten einde loopt (blokkeert of naar een oplossing leidt) wordt een ander pad onderzocht. In de ogen van een ontwerper van een traditionele stellingbewijzer is dit absurd : een bepaald pad kan oneindig zijn. Om zeker een eventuele oplossing te vinden moet een traditioneel systeem zijn aandacht ver-

delen over de verschillende paden. ("parallel" werken). We noemen deze nieuwe methode "sequentiële exploratie" ("backtracking").

Het ontstaan van PLANNER gaf aanleiding tot een felle controverse tussen de aanhangers van een procedure-gerichte aanpak en deze van een deklaratieve aanpak. In 1973 stelde Hayes [Hayes 73] dat deze controverse kunstmatig is, dat beide benaderingen samengaan, maar dat men een duidelijk onderscheid moet maken tussen enerzijds de probleembeschrijving (het deklaratieve aspect) en anderzijds de informatie, die de stellingbewijzer moet helpen in het vinden van een oplossing (het procedure-gerichte aspect). Hij stelde dat een dergelijke werkwijze de programma's een betekenis geeft die onafhankelijk is van een specifiek executiemechanisme.

Het jaar daarop definieerde Kowalski [Kowalski 74] een procedure-gerichte interpretatie voor zogenoemde Horn-uitdrukkingen ("Horn clauses"), een deelverzameling van de logika. Hij stelt dat Horn-uitdrukkingen kunnen beschouwd worden als de beschrijving van een probleem (het deklaratieve aspect) onafhankelijk van een bepaald executiemechanisme, maar tevens dat diezelfde Horn-uitdrukkingen kunnen beschouwd worden als procedures en dat het uitvoeren van een resolutiestap kan beschouwd worden als het uitvoeren van een procedureoproep. Colmerauer en Roussel [Colmerauer & Co 73] ontwikkelden een systeem "PROLOG" dat in essentie een vertolker voor Horn-uitdrukkingen is. Zij gaven de gebruiker de mogelijkheid de uitvoering van zijn Horn-uitdrukkingen te controleren. In feite is er ook hier nog steeds geen duidelijk onderscheid tussen de logika en de controle. Nochtans bestaat er een deelverzameling van deze controlemogelijkheden die de logika van de programma's niet beinvloedt. De basisbewerking in PROLOG is resolutie. Het systeem gebruikt eveneens de sequentiële exploratietechniek. Door middel van een eenvoudige controletaal bepaalt de gebruiker in welke orde de verschillende paden doorlopen worden.

PROLOG is niet de verwezenlijking van de droom om een vertolker te ontwikkelen voor probleemspecifikaties in de algemene vorm van logika. Dit is nu nog onbereikbaar, we missen een procedure-gerichte interpretatie voor logika in haar algemene vorm en tevens het nodige inzicht in de wijze waarop een algemene stellingbewijzer moet gecontroleerd worden. PROLOG is een compromis tussen wat haalbaar en wat wenselijk

is en als dusdanig een door zijn gebruikers erg gewaardeerde universele taal van zeer hoog niveau. De taal heeft een dubbele semantiek.

Enerzijds is er de deklaratieve semantiek waarbij men PROLOG programma's beschouwt als "nuttige stellingen" : stellingen die bijdragen tot de oplossing van het gestelde probleem. Dit laat toe om te redeneren over PROLOG programma's op een zeer hoog niveau, los van elk executiemechanisme. Men kan bewijzen, gebruik makend van de algemene logika, dat het programma consistent is met de probleemspecifikatie, met andere woorden, dat de relatie bepaald door het programma dezelfde is als deze bedoeld in de probleemspecifikatie ("partiële correctheid"). Tevens kan men bewijzen dat, voor een groep van problemen, het programma inderdaad tot een oplossing leidt (bewijs dat het programma "eindigt"). Dit laatste bewijs is eveneens los van een bepaald executiemechanisme en zegt dat een oplossing kan gevonden worden; een volledige stellingbewijzer zal dan ook de oplossing vinden. Momenteel is er heel wat onderzoek op dit terrein [Hogger 78][Clark & Tarnlund 77][Kowalski 79]. Deze correctheidsbewijzen schijnen heel wat gemakkelijker te zijn dan bij conventionele talen, enerzijds omdat programma en correctheidsbewijs allebei in hetzelfde formalisme geschreven zijn (logika), anderzijds omdat men zich in deze bewijzen niet moet bekommeren om de controle. Juist deze controle veroorzaakt heel wat moeilijkheden bij conventionele talen ("assignment", "while", "goto" ...).

Anderzijds is er de procedure-gerichte semantiek. Een PROLOG programma is tevens een handleiding die voor een welbepaalde stellingbewijzer aanduidt hoe een oplossing te vinden. Aspecten die hier aan bod komen zijn de efficiëntie van een programma en de praktische eindigheid : laat de opgegeven controle de stellingbewijzer toe om inderdaad een oplossing te vinden ?

Voor de programmeur is PROLOG een alternatief voor LISP :

- de notatie is eleganter (geen overrompeling van haakjes),
- er zijn geen niet-gedefinieerde operaties (zoals car van een lege lijst in LISP),
- de expressieve kracht is groter (dit komt onder andere tot uiting in het feit dat PROLOG veel dichter bij de pure logika staat dan LISP bij de pure lambda-calculus) en

- de ervaring van Warren [Warren 77] toont aan dat de implementatietechniek reeds ver genoeg gevorderd is om systemen te ontwerpen met een efficiëntie die vergelijkbaar is met deze van de beste LISP systemen.

De taal werd dan ook reeds gebruikt voor allerhande toepassingen op het gebied van kunstmatige intelligentie o.a. analyse van natuurlijke taal [Colmerauer & Co 73], symbolische integratie [Bergman & Kanoui 75], planformatie [Warren 74], ontwerp flatgebouwen [Markusz 77], wisselwerking tussen verschillende geneesmiddelen [Darvas & Co 78], ...

We besluiten deze sektie met de visie van Robert Kowalski op het gebruik van logika als een programmeertaal. Hij stelt dat logika te lang is opgesloten geweest in het beperkte wereldje der mathematici en dat het hoog tijd is om haar bekend te maken aan andere disciplines. Logika is een universeel middel voor het beschrijven van kennis en is als dusdanig bruikbaar in tal van disciplines. Terzelfdertijd kan (een deelverzameling van) logika gebruikt worden als programmeertaal. Dit biedt het grote voordeel dat hetzelfde formalisme kan gebruikt worden voor de studie van een discipline en voor de programmatie van problemen in die discipline. Programma's vormen dan dat deel van de kennis uit de discipline dat nuttig is voor het oplossen van problemen.

B. HORN-UITDRUKKINGEN ALS PROGRAMMA'S

In een eerste deel van deze sektie bekijken we logisch programmeren vanuit het standpunt van een conventionele programmeur, daarna vanuit het standpunt van een logikus. Tenslotte bestuderen we enkele specifieke kenmerken van PROLOG, de te Marseille ontwikkelde versie van een logische programmeertaal.

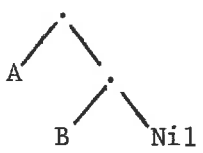
B.1. EEN PROGRAMMEERTAAL

GEGEVENSSTRUKTUREN

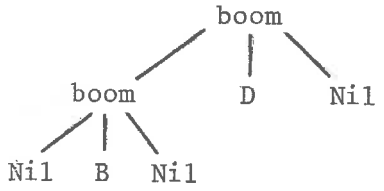
De gegevensstrukturen van deze programmeertaal bestaan uit geordende bomen. Een knooppunt van een dergelijke boom is een bouwelement ("constructor").
Het heeft een naam en een graad. De graad ($\geqslant 0$) is het aantal takken dat uit het knooppunt vertrekt. Een bouwelement van graad 0 wordt een constante genoemd; een bouwelement met een graad >0 wordt een complex genoemd.

Voorbeelden

1.



Deze gegevensstruktuur kan eveneens ondubbelzinnig voorgesteld worden als .(A,.(B,Nil)) of als A.(B.Nil). In deze tekst zullen we meestal de haakjes weglaten en A.B.Nil schrijven. We nemen aan dat de haakjes aangevuld worden zoals hierboven aangegeven is. In deze struktuur is "." een binair complex en zijn A,B en Nil constanten. "." kan geïnterpreteerd worden als een bouwelement voor een lijst. De linkertak beschrijft het eerste element van de lijst opgebouwd door ".", de rechtertak beschrijft de rest van de lijst. Nil wordt geïnterpreteerd als een lege lijst.

2. 

of boom (boom(Nil,B,Nil),D,Nil)

In deze struktuur is boom een ternair complex. Deze struktuur kan geïnterpreteerd worden als een boomstruktuur met een linkertak, de knooppuntinformatie en een rechtertak. Hier wordt Nil geïnterpreteerd als een lege boom.

Deze gegevensstrukturen zijn vergelijkbaar met "records" in een conventionele taal. Een bouwelement kan beschouwd worden als het type van een record, de takken als de velden van een record. Nochtans, terwijl een veld in een record een bepaald type heeft, kan een tak om het even welke gegevensstruktuur zijn.

LISP heeft slechts één enkel complex bouwelement en één enkel soort gegevensstruktuur namelijk de lijststruktuur. Horn-uitdrukkingen beschikken over een onbeperkt potentieel van complexe bouwelementen.

VARIABELEN

Een programma heeft de bedoeling iets te berekenen. Ook dat resultaat is een gegevensstruktuur. En, alhoewel het resultaat gewoonlijk welbepaald is, bij het begin van het programma is het (natuurlijk) onbekend. We stellen het daarom voor door een variabele. In de contekst van een welbepaalde berekening stelt een variabele een welbepaalde, maar tijdelijk onbekende gegevensstruktuur voor. Elk voorkomen van dezelfde variabele stelt dezelfde gegevensstruktuur voor. Het effekt van een berekening is dat meer en meer componenten van de onbekende gegevensstrukturen bekend worden.

Voorbeeld

Stel dat we een lijst A.B.Nil willen samenvoegen met een tweede lijst C.D.Nil. De resulterende lijst is initieel onbekend en kan voorgesteld worden door een variabele x. (Om variabelen te onderscheiden van constanten begint de naam van een variabele met kleine letter en deze van een constante met een hoofdletter). Een mogelijke berekening zal deze x gradueel verfijnen tot $A.x_1$, later tot $A.B.x_2$ en uiteindelijk tot A.B.C.D.Nil.

Het is niet uitgesloten dat een berekening een verkeerde weg opgaat en aan een bepaalde variabele foutieve waarden toekent. Vroeg of laat zal dit ontdekt worden en zullen de foutieve gegevensstrukturen verdwijnen.

Dit gebruik van variabelen als namen voor welbepaalde maar tijdelijk onbekende objecten staat in fel contrast met het gebruik van variabelen in conventionele talen. In een conventionele taal staat een variabele voor een geheugenplaats in de machine. De gebruiker moet zich zorgen maken over het onderscheid tussen het adres en de inhoud, tussen een wijzer naar een gegevensstruktuur en de gegevensstruktuur zelf, over de vraag of een adres reeds een bepaalde inhoud heeft en of hij die inhoud nog nodig heeft ..., allemaal erg machine georiënteerde concepten die in feite niet thuishoren in een hogere programmeertaal [Backus 78]. Het gebruik van variabelen in Horn-uitdrukkingen staat op een veel hoger niveau en de gebruiker hoeft zich geen zorgen te maken over al die machinegebonden concepten. De ontwerper van het systeem heeft deze problemen voor hem opgelost.

Zoals we in de volgende sektie zullen duidelijk maken is dit gebruik van variabelen gesteund op de onderliggende logika. Een variabele staat voor een onbekend object zoals in de wiskunde, niet voor een geheugenplaats zoals in conventionele talen. Ook LISP is gesteund op een wiskundig model, namelijk de lambda calculus, maar zuivere LISP is te beperkt bevonden. Praktische LISP systemen hebben een groot aantal concepten die van de zuivere lambda calculus afwijken en terug machine georiënteerde operaties zoals expliciete geheugenmanipulatie toelaten.

PROCEDURES EN DE BINDING TUSSEN FORMELE EN ACTUELE PARAMETERS

Een conventionele taal gebruikt selektiefunkties en assignatie voor het manipuleren van gegevensstrukturen. Zuivere LISP heeft funkties voor het opbouwen van strukturen (CONS) en voor het selekteren van deelstrukturen (CAR en CDR). Horn-uitdrukkingen gebruiken een totaal andere techniek, "unifikatie" ("unification") genoemd, voor het manipuleren van zijn gegevensstrukturen. Unifikatie is een gesofistikeerde vorm van patroonherkenning en vormt de basisoperatie van de taal. De uitvoering van een programma bestaat in essentie uit een reeks unifikaties. Vanuit het standpunt van conventionele programmeertalen is unifikatie een speciaal mechanisme voor het verwezenlijken van de binding tussen de actuele parameters in een procedure-oproep en de formele parameters in een proceduredefinitie. Actuele en formele parameters zijn beide gege-

vensstrukturen. De binding bestaat erin hen te vervangen door hun grootste gemene struktuur ("most general unifier"), dit is de meest algemene struktuur die hen identiek maakt. Dit proces heeft meestal tot gevolg dat bepaalde variabelen, zowel in de actuele als in de formele parameters, een waarde krijgen. We herinneren eraan dat die variabele, overal waar ze voorkomt, meteen dezelfde waarde krijgt.

Voorbeelden

- Actuele parameter : A.B.Nil
  Formele parameter : x.y
  x wordt gepreciseerd als A en y als B.Nil.
  Dit is een voorbeeld van selektie in een struktuur

- Actuele parameter : z
  Formele parameter : u.v
  z wordt u.v : konstruktie

- Actuele parameter : x
  Formele parameter : y
  y wordt x (of x wordt y) : x en y preciseren dezelfde niet nader gespecifieerde struktuur

- Actuele parameter : A.x
  Formele parameter : u.v.w
  Grootste gemene struktuur A.v.w
  u wordt A
  x wordt v.w
  Er gebeurt hier zowel selektie als konstruktie in eenzelfde unifikatie.

- Actuele parameter : Append(A.B.Nil,C.Nil,z)
  Formele parameter : Append(x.u,v,x.w)
  Grootste gemene struktuur : Append(A.B.Nil,C.Nil,A.w)
  x ← A, u ← B.Nil, v ← C.Nil, z ← A.w

We zien dat unifikatie een zeer algemeen bindingsmechanisme is. Het laat zowel toe om componenten te selekteren in de gegevens als om (ge-

deeltelijke) resultaten op te bouwen. Dit mechanisme is veel algemener dan de patroonherkenning die gebruikt wordt in MICRO-PLANNER [Sussman & Mc Dermott 72]. MICRO-PLANNER kent geen partiële waarden voor variabelen; een variabele is ofwel niet bekend ofwel volledig bekend.

In het laatste voorbeeld beschouwden we Append(A.B.Nil,C.Nil,z) als een actuele parameter. We kunnen een dergelijke uitdrukking eveneens beschouwen als een procedure-oproep met naam Append en drie actuele parameters. Append(x.u,v,x.w) is dan de hoofding van een proceduredefinitie met drie formele parameters. Als we nog vertellen dat een hoofdprogramma bestaat uit één of meerdere procedure oproepen en het lichaam van een proceduredefinitie uit nul of meer procedure-oproepen, dan is het meteen duidelijk dat de uitvoering van een programma inderdaad bestaat uit een reeks unifikaties.

Het is best mogelijk dat er meerdere proceduredefinities zijn met dezelfde naam. Een bepaalde oproep kan geen, één of meerdere definities selekteren. Wanneer geen enkele definitie met de oproep overeenstemt beantwoordt de tot dan bekomen gegevensstruktuur niet aan het gestelde probleem en die struktuur moet dus verdwijnen. Wanneer de oproep met verschillende definities overeenstemt kan de tot dan bekomen struktuur op meerdere wijzen uitgebreid worden. Dit leidt eventueel tot verschillende oplossingen. Deze laatste mogelijkheid is verantwoordelijk voor het niet-determinisme in de taal.

PROCEDURES ALS RELATIES - DE UITVOERING VAN EEN PROGRAMMA

Procedures definiëren relaties. Een procedure Append die twee lijsten samenvoegt tot één beschrijft een oneindige ternaire relatie. Enkele elementen uit deze relatie zijn : (A.Nil,B.Nil,A.B.Nil)
(Nil,Nil,Nil)
(A.B.Nil,D.Nil,A.B.D.Nil)
Een bepaalde oproep selekteert de deelverzameling die overeenstemt met het patroon van de oproep. Een oproep Append(A.Nil,B.Nil,x) selekteert (A.Nil,B.Nil,A.B.Nil). Een oproep Append(x,y,B.Nil) selekteert (Nil,B.Nil,B.Nil) en (B.Nil,Nil,B.Nil).

Proceduredefinities met een ledig lichaam sommen een deel van de elementen van de relatie op.

Voorbeeld : Append(Nil,x,x) ←

staat voor alle elementen met Nil in de eerste positie

o.a. (Nil,Nil,Nil)

(Nil,B.Nil,B.Nil)

(Nil,B.D.Nil,B.D.Nil)

Proceduredefinities met een niet ledig lichaam definiëren elementen van de relatie in functie van andere relaties.

Voorbeeld :

- grootvader(x,y) ← vader(x,z) vader(z,y) : Als de koppels(Jan,Piet) en (Piet,Pol) tot de relatie vader behoren, dan behoort het koppel (Jan,Pol) tot de relatie grootvader, of andersom, het koppel(Jan, Pol) behoort tot de relatie grootvader indien er een z bestaat zó-dat de koppels(Jan,z) en (z,Pol) behoren tot de relatie vader.
- Append(x.u,v,x.w) ← Append(u,v,w) : Als een element (u,v,w) behoort tot de relatie Append, dan behoort ook het element (x.u,v,x.w) tot deze relatie. Volgens de eerste definitie van Append behoort (Nil,B.Nil,B.Nil) tot de relatie, dus, volgens deze definitie ook (A.Nil,B.Nil,A.B.Nil) en dus ook (C.A.Nil,B.Nil,C.A.B.Nil).

Gebruik makend van de twee bovenstaande definities voor Append kunnen we de uitvoering van een eenvoudig programma illustreren. Een mogelijk hoofdprogramma is

← Append(A.Nil,B.Nil,z) Antwoord(z)

Hierin is Antwoord(z) een pseudo-oproep waarin we het resultaat (z) laten terechtkomen. De oproep Append(A.Nil,B.Nil,z) heeft slechts met de tweede definitie een grootste gemene struktuur. Deze struktuur is Append(A.Nil, B.Nil,A.w). De waarde van z wordt doorgegeven aan de andere oproepen in het hoofdprogramma en deze van u en v worden doorgegeven aan de oproepen in het lichaam van de gebruikte definitie. De nog uit te voeren oproepen

zijn Append(Nil,B.Nil,w) uit het lichaam van de definitie en Antwoord(A.w) uit het hoofdprogramma. Het effekt van deze eerste stap is dat men stelt dat het resultaat van de vorm A.w is op voorwaarde dat er een w bestaat zó dat (Nil,B.Nil,w) behoort tot de relatie Append. De nieuwe oproep van Append selekteert de eerste definitie en resulteert in w ← B.Nil. De toegepaste definitie heeft een ledig lichaam en de enige overblijvende oproep is Antwoord(A.B.Nil). Het argument van deze pseudo-oproep is het resultaat.

Met dit eenvoudig voorbeeld besluiten we deze informele inleiding tot onze logische programmeertaal. De aandachtige lezer zal opmerken dat nog heel wat vragen onbeantwoord blijven. We hebben niets gezegd over de wijze waarop het niet-determinisme opgevangen wordt. We hebben eveneens gezwegen over de volgorde waarin de verschillende procedure-oproepen uitgevoerd worden. Deze aspecten hebben in eerste instantie geen invloed op de mening en de resultaten van een programma. De volgorde waarin de verschillende unifikaties worden uitgevoerd is van geen invloed op het uiteindelijk resultaat. De vraag of men de verschillende resultaten het een na het ander ofwel gelijktijdig berekent heeft eveneens geen invloed. Deze problemen zijn daarom niet onbelangrijk, integendeel ze zijn zeer belangrijk voor wie een vertolker ontwerpt en zelfs voor de gebruiker die bekommerd is om de efficiëntie van zijn programma. Dit illustreert nog een essentieel verschilpunt met conventionele talen. Bij een conventionele taal is de volgorde van de operaties rigoureus bepaald. Hier kan men deze volgorde, voor wat betreft de mening van het programma, buiten beschouwing laten. Dit is te danken aan het unieke karakter van de "logische variabele" die staat voor een welbepaalde onbekende struktuur.

B.2. GEBASEERD OP LOGIKA

LOGISCHE FORMULES - HORN-UITDRUKKINGEN

Logika bestudeert de consistentie van beweringen. Om meer vat te hebben op het consistentieprobleem worden de beweringen omgezet in logische formules en wordt de consistentie van deze formules bestudeerd. De formalisering is zover gegaan dat de studie van de consistentie van formules een

onderwerp op zichzelf geworden is en de band met de beweringen die ze voorstellen, volledig verloren gegaan is. Formele logika werd een onderwerp voor een selekte groep wiskundigen. Vele boeken over logika bevatten een, voor niet-ingewijden, moeilijk te begrijpen gegoochel met symbolen. Niet dat de resultaten onbelangrijk zijn, de zopas informeel gedefinieerde programmeertaal is erop gebaseerd. (Op de predikatenlogika van de eerste orde). Nochtans is voor het logisch programmeren het verband tussen de logische formules en de beweringen die ze voorstellen essentieel. Inderdaad, logisch programmeren is niet zomaar wat goochelen met symbolen, maar bestaat uit het opstellen van een verzameling beweringen die nuttig zijn om een gesteld probleem op te lossen. Daarom zullen wij hier vooral het verband tussen formules en beweringen benadrukken. We zullen de formele predikatenlogika van eerste orde eerder informeel benaderen en slechts de aspecten beschrijven die een beter inzicht geven in het logisch programmeren.

Aan de hand van een reeks voorbeelden kijken we hoe logische formules opgebouwd worden.

(1) Jan is de vader van Piet.
Vader(Jan,Piet)

(2) Nil is een ledige lijst.
Ledig(Nil)

(3) Lieve geeft koek aan Riet.
Geven(Lieve,Koek,Riet)

Dit zijn elementaire zinnen, ze beschrijven relaties tussen objecten. De bijhorende formules noemen we atomen. Een atoom bestaat uit een predikaat (Vader,Ledig,Geven) en een aantal argumenten. Elk predikaat heeft een welbepaald aantal elementen (Vader heeft er 2,...). In bovenstaande voorbeelden zijn alle argumenten constanten (Jan,Piet,...).

Elementaire zinnen kunnen we combineren tot samengestelde zinnen :

(4) Jan is niet de vader van Piet.
$\sim$ Vader(Jan,Piet)

(5) Jan is feilbaar indien Jan een mens is.
Feilbaar(Jan) $\leftarrow$ Mens(Jan)

(6) Jan is ziek en Piet is gezond.

Ziek(Jan) $\wedge$ Gezond(Piet)

(7) Jan is ziek of gezond.

Ziek(Jan) $\vee$ Gezond(Jan)

Men neemt hier een zwakke interpretatie aan van de zin : men sluit niet uit dat Jan zowel ziek als gezond is.

(8) Jan is ziek indien hij niet gezond is.

Ziek(Jan) $\leftarrow \sim$ Gezond(Jan)

(9) Jan is alleen ziek indien hij niet gezond is.

$\sim$ Gezond(Jan) $\leftarrow$ Ziek(Jan)

of $\sim$ Ziek(Jan) $\leftarrow$ Gezond(Jan)

(10) Jan is ziek indien en slechts indien hij niet gezond is.

Ziek(Jan) $\leftrightarrow \sim$ Gezond(Jan)

Deze samengestelde formules zijn opgebouwd uit atomen met behulp van de logische connectoren $\sim$, $\vee$, $\wedge$, $\leftarrow$ en $\leftrightarrow$. Er is een zekere redundantie aanwezig, (7) zegt hetzelfde als (8), (8) en (9) samen zeggen hetzelfde als (10).

We kunnen nog een ander soort zinnen opstellen :

(11) Alle mensen zijn feilbaar.

In andere woorden :

Iemand is feilbaar indien hij een mens is.

$\forall x$[Feilbaar(x) $\leftarrow$ Mens(x)]

(12) Er bestaat een ledige lijst

$\exists x$[Ledig(x)]

(13) Er bestaat een lijst met eerste element A en rest Nil.

$\exists x$[Componenten(x,A,Nil)]

(14) Voor elke x en y bestaat er een lijst met eerste element x en rest y

$\forall x \forall y \exists z$ Componenten(z,x,y)

(15) Voor elke x en y is x het eerste element en y de rest van de lijst
$\ell(x,y)$
$\forall x \forall y$ Componenten($\ell(x,y)$,x,y)

$\forall$ is de universele kwantor, $\exists$ is de existentiële kwantor, x,y en z zijn variabelen, $\ell(x,y)$ is een samengestelde term. ℓ is een funktiesymbool. Elk funktiesymbool heeft een welbepaald aantal argumenten.

De tijd is rijp om een en ander ietwat formeler te definiëren :

term :
- een constante is een term vb Jan, Piet, Koek
- een variabele is een term vb x,y,z
- als $t_1,\ldots,t_n$ termen zijn en f een funktiesymbool is, dan is $f(t_1,\ldots,t_n)$ een term

atoom :
- als P een predikaat is en $t_1,\ldots,t_n$ termen zijn dan is $P(t_1,\ldots,t_n)$ een atoom

formule :
- een atoom is een formule
- als F en G formules zijn, dan zijn ook $\sim F$, $F \vee G$, $F \wedge G$, $F \leftarrow G$ en $F \leftrightarrow G$ formules
- als F een formule is en een variabele x bevat die niet geassocieerd is met een kwantor in F (x is vrij), dan zijn $\forall x[F]$ en $\exists x[F]$ formules. In deze nieuwe formules is x geassocieerd met een kwantor (x is gebonden).

We gebruiken slechts formules die geen vrije veranderlijken bevatten.

Toen we de logische connectoren invoerden, zagen we reeds dat verschillende zinnen dezelfde mening kunnen hebben ((7) en (8)). Dit is eveneens zo voor zinnen die kwantoren bevatten.

(16) Het is niet waar dat geen enkele lijst ledig is $\sim \forall x[\sim \text{Ledig}(x)]$

Deze zin heeft dezelfde mening als (12). Er is nog een ander fenomeen.

De beweringen (2) en (12) en ook (14) en (15) hebben niet dezelfde mening. Nochtans wanneer, in een verzameling beweringen, de ene vervangen wordt door de andere en de ingevoerde constante (Nil) of funktiesymbool (ℓ) niet voorkomt in de andere beweringen, dan blijft de eventuele consistentie van die verzameling ongewijzigd. Men kan dus de existentiële kwantor verwijderen door het element dat zeker bestaat een naam te geven. Deze naam hangt eventueel af van de context :

$\forall y \exists x$ Vader(x,y) "iedereen heeft een vader"

De naam van de vader is natuurlijk afhankelijk van de persoon :

$\forall y$ Vader(f(y),y).

Dit alles werd eveneens geformaliseerd en er werd een normaalvorm gedefinieerd. Een willekeurige verzameling beweringen kan omgezet worden tot een verzameling beweringen van de vorm

$$\forall x_1 \forall x_2 \ldots \forall x_k [B_1 \vee B_2 \ldots \vee B_m \leftarrow A_1 \wedge A_2 \ldots \wedge A_n].$$

Hierbij zijn alle B_i en A_i atomen. De nieuwe verzameling heeft dezelfde consistentie als de oude. Om de notatie te vereenvoudigen schrijven we een bewering in standaarvorm als :

$$B_1, B_2 \ldots, B_m \leftarrow A_1, \ldots, A_n.$$

Een dergelijke bewering noemen we een uitdrukking ("clause"). Voor m=1 herkent de aandachtige lezer de vorm van de proceduredefinities beschreven in de sektie B.1.

Een uitdrukking met m=1 noemt men een Horn-uitdrukking ("Horn-clause"). Deze normaalvorm werd voorgesteld in [Kowalski 74b] en is equivalent met een meer verspreide normaalvorm waarin een bewering geschreven wordt als :

$$\forall x_1 \forall x_2 \ldots \forall x_k [B_1 \vee B_2 \ldots \vee B_m \vee \sim A_1 \vee \ldots \vee \sim A_n].$$

Onze interesse beperkt zich tot Horn-uitdrukkingen : de reden hiervoor zullen we later verklaren.

Onze voorbeelden hebben reeds aangetoond dat dezelfde bewering op verschillende manieren kan geformuleerd worden en dat elke formulering aanleiding geeft tot een verschillende maar equivalente formule. De vraag rijst of we de dingen zo kunnen formuleren dat de bijhorende formule direct in normaalvorm komt te staan of, in andere woorden , of we met een formule in normaalvorm een begrijpelijke Nederlandse zin kunnen

associëren. Alhoewel het antwoord betwist wordt voor uitdrukkingen in het algemeen, is men het erover eens dat dit goed mogelijk is voor Horn-uitdrukkingen.

Voorbeeld

- Append(Nil,x,x) ←
  Append(ℓ(x,u),v,ℓ(x,w)) ← Append(u,v,w)

Men kan Append(x,y,z) lezen als : het samenvoegen van de lijst x met de lijst y resulteert in de lijst z. Men kan Nil lezen als : de ledige lijst. Men kan ℓ(x,y) lezen als : een lijst met eerste element x en rest y.
De eerste Horn-uitdrukking kan men lezen als : het samenvoegen van de ledige lijst met een willekeurige lijst x resulteert in die lijst x.
De tweede Horn-uitdrukking kan men lezen als : het samenvoegen van een lijst met eerste element x en rest u met een lijst v resulteert in een lijst met eerste element x en rest w indien het samenvoegen van de lijst u met de lijst v resulteert in de lijst w.

- Grootvader(x,y) ← Vader(x,z) Vader(z,y)
  Deze Horn-uitdrukking is equivalent met de formule :
  $\forall x \forall y[\text{Grootvader}(x,y) \leftarrow \exists z[\text{Vader}(x,z) \wedge \text{Vader}(z,y)]]$
  (Dus existentiële kwantifikatie over de variabelen die alleen in de rechterkant voorkomen). De Horn-uitdrukking kan dus gelezen worden als : x is de grootvader van y indien er een z bestaat zó dat x de vader is van z en z de vader is van y.

Gezien de voor ons interessante beweringen direct in normaalvorm kunnen geformuleerd worden, gaan we hier niet nader in op de omzetting naar normaalvorm.

CONSISTENTIE

Het is onze bedoeling logika te gebruiken voor het oplossen van problemen. We willen de bovenstaande Horn-uitdrukking over de relatie

Grootvader gebruiken, samen met de Horn-uitdrukkingen Vader(Jan,Piet) ← en Vader(Piet,Pol) ← om de vraag "wie is de grootvader van Pol ?" te beantwoorden. Wanneer we aan deze Horn-uitdrukkingen de bewering "Pol heeft geen grootvader" toevoegen, dan is het intuïtief duidelijk dat de resulterende verzameling beweringen contradictorisch is, inderdaad, Jan is de grootvader van Pol.

We kunnen deze notie van een contradictorische verzameling wat preciéser beschrijven.

- Een verzameling $S = \{C_1,\ldots,C_n\}$ van uitdrukkingen is contradictorisch indien en slechts indien ze onwaar is in elke interpretatie I van S.
- S is onwaar in een interpretatie I indien en slechts indien minstens één van de uitdrukkingen $C_1,\ldots,C_n$ onwaar is in I.
- Een uitdrukking C is onwaar in I indien er een uitdrukking C' bestaat die afgeleid is uit C door de variabelen in C te vervangen door termen die geen variabelen bevatten en C' onwaar is in I. (Een uitdrukking C' afgeleid uit C door bepaalde variabelen in C te vervangen door termen noemen we een variant van C; een variant die geen variabelen bevat noemen we een grondvariant).
- Een grondvariant $C = B_1,\ldots,B_m \leftarrow A_1,\ldots,A_n$ is onwaar in I indien en alleen indien alle atomen $A_1,\ldots,A_n$ waar zijn in I en geen enkel atoom $B_1,\ldots,B_m$ waar is in I. Een interpretatie I van S wordt bekomen door waar of onwaar toe te kennen aan alle grondatomen die afleidbaar zijn van de atomen die voorkomen in S. Met een interpretatie I van S is een domein verbonden, een mogelijk domein bestaat uit alle grondtermen die kunnen opgebouwd worden uit de funktiesymbolen en constanten die voorkomen in S. Dit domein is bekend als het Herbrand universum. Men heeft bewezen dat men voor het opbouwen van grondvarianten niet alle mogelijke termen moet beschouwen, maar dat men zich mag beperken tot termen uit dit Herbrand universum.

Voorbeeld

Grootvader(x,y) ← Vader(x,z) Vader(z,y)
Vader(Jan,Piet) ←
Vader(Piet,Pol) ←

Het Herbrand universum is {Jan,Piet,Pol}

Een interpretatie waarvoor deze verzameling waar is moet zeker de volgende toekenningen doen :

waar voor Vader(Jan,Piet) (om de 2de Horn-uitdrukking waar te maken)

waar voor Vader(Piet,Pol) (om de 3de Horn-uitdrukking waar te maken)

waar voor Grootvader(Jan,Pol) (om de variant Grootvader(Jan,Pol) ← Vader(Jan,Piet) Vader(Piet,Pol) waar te maken.

Voor de overige atomen zoals Vader(Piet,Jan) ... Grootvader(Jan,Jan) heeft men verschillende mogelijkheden o.a. allemaal onwaar of allemaal waar.

Voegt men de uitdrukking

← Grootvader(x,Pol)

toe, dan bekomt men een contradictorische verzameling. Inderdaad, van de drie grondvarianten afleidbaar uit deze uitdrukking, namelijk

← Grootvader(Pol,Pol)

← Grootvader(Piet,Pol)

en ← Grootvader(Jan,Pol)

is de derde steeds onwaar in elke interpretatie die alle grondvarianten van de drie eerste Horn-uitdrukkingen waar maakt. Door te ontkennen dat Pol een grootvader heeft zal een systeem dat contradicties opspoort de waarden zoeken die de verzameling contradictorisch maken. Juist die waarden zijn het antwoord op de gestelde vraag.

Wanneer een verzameling uitdrukkingen functiesymbolen bevat, bestaat het Herbrand universum uit een oneindig aantal termen, en dus kan men uit een uitdrukking met variabelen een oneindig aantal varianten afleiden. Dit suggereert dat het een moeilijk probleem is om na te gaan of een verzameling uitdrukkingen al of niet contradictorisch is. Church, en ook Turing bewezen dat het onmogelijk is om een methode te ontwikkelen die voor elke verzameling uitdrukkingen zegt of de verzameling al of niet consistent is. Het beste wat men kan verwachten is een methode die van een contradictorische verzameling zegt dat ze inderdaad contradictorisch is. Wanneer men de methode toepast op een consistente verzameling, dan zal ze eventueel nooit eindigen. Voor ons

is dit geen echte beperking, een programma berekent meestal een oplossing voor een probleem waarvan men weet dat de oplossing bestaat.

HET AUTOMATISCH BEWIJZEN VAN STELLINGEN-RESOLUTIE

Een procedure die aantoont dat een verzameling uitdrukkingen contradictorisch is kan gebruikt worden voor het bewijzen van stellingen. Inderdaad, de gegevens, samen met de negatie van het te bewijzen vormen een contradictorische verzameling. In 1930 ontwikkelde Herbrand een dergelijke procedure. De methode werd in belangrijke mate verbeterd door Robinson in 1965 [Robinson 65] en werd bekend als de resolutiemethode. De idee is om uit de verzameling uitdrukkingen nieuwe uitdrukkingen af te leiden en uiteindelijk een uitdrukking af te leiden die triviaal onwaar is in elke interpretatie. Een dergelijke uitdrukking is de ledige uitdrukking $\square$, het is de uitdrukking $B_1,\ldots,B_m \leftarrow A_1,\ldots,A_n$ met m=n=0. Inderdaad, voor elke interpretatie geldt dat alle atomen in het rechterlid waar zijn, terwijl geen enkel atoom in het linkerlid waar is. Sinds 1965 werden heel wat verfijningen aan de resolutiemethode aangebracht. Men wijzigt de oorspronkelijke verzameling uitdrukkingen zowel door toewijzingen als door weglatingen, maar men draagt er zorg voor dat de nieuwe verzameling contradictorisch is indien en alleen indien de oude verzameling contradictorisch is. In al deze methodes is resolutie de basisbewerking voor het afleiden van nieuwe uitdrukkingen.

De evolutie van automatische stellingsbewijzers tot vertolkers voor logische programma's hebben we geschetst in sektie A, hier zullen we ons beperken tot het concept resolutie. We beperken ons opnieuw tot Horn-uitdrukkingen. Een algemene formulering kan o.a. gevonden worden in [Chang & Lee 73].

Uit een uitdrukking $C_1 = A \leftarrow B_1,\ldots,B_{i-1},B_i,B_{i+1},\ldots,B_m$ en een uitdrukking $C_2 = C \leftarrow D_1,\ldots,D_n$ met $C \equiv B_i$ kan men een uitdrukking $A \leftarrow B_1,\ldots,B_{i-1},D_1,\ldots,D_n,B_{i+1},\ldots,B_m$ afleiden. Inderdaad, uit het feit dat A waar is indien B_1 en ... en B_i en ... en B_m waar zijn en dat B_i waar is indien D_1 en ... en D_n waar zijn volgt dat A waar is indien B_1 en ... en B_{i-1} en D_1 en ... en D_n en B_{i+1} en ... B_m waar zijn.

Nu is het meestal zo dat twee Horn-uitdrukkingen C_1 en C_2 geen gelijke atomen B_i en C hebben maar dat er varianten bestaan waarin die atomen identiek zijn.

Voorbeeld

C_1 : $A(x,y) \leftarrow B(y,f(x))$

C_2 : $B(a,u) \leftarrow D(u)$

Uit de varianten $A(x,a) \leftarrow B(a,f(x))$ en $B(a,f(x)) \leftarrow D(f(x))$ kan men $A(x,a) \leftarrow D(f(x))$ afleiden; uit $A(b,a) \leftarrow B(a,f(b))$ en $B(a,f(b)) \leftarrow D(f(b))$ kan men $A(b,a) \leftarrow D(f(b))$ afleiden; ... . De tweede afgeleide Horn-uitdrukkingen is een variant van de eerste en alle andere Horn-uitdrukkingen afleidbaar uit C_1 en C_2 zijn varianten van die eerste afgeleide uitdrukking. Die eerste afgeleide Horn-uitdrukking is de meest algemene Horn-uitdrukking die uit C_1 en C_2 afleidbaar is. Ze vertegenwoordigt alle varianten die door toepassing van bovenstaande regel uit C_1 en C_2 afleidbaar zijn. Robinson's resolutieprincipe bestaat erin dat men dergelijke meest algemene uitdrukkingen berekent en toevoegt aan de verzameling uitdrukkingen. Hij bewees dat door het systematisch toepassen van resolutie tussen alle mogelijke paren uitdrukkingen van een contradictorische verzameling, men vroeg of laat de ledige uitdrukking afleidt. (Bij niet-Horn-uitdrukkingen kunnen meer dan twee atomen in eenzelfde resolutiestap betrokken worden).

Resolutie op de atomen B_i en C van $A \leftarrow B_1,\ldots,B_i,\ldots,B_m$ en $C \leftarrow D_1,\ldots,D_n$ bestaat erin dat men de varianten berekent waarin B_i en C vervangen worden door hun grootste gemene atoom. Deze berekening wordt unifikatie genoemd. Door het opleggen van beperkingen aan de waarden van de variabelen in B_i en C worden beide atomen aan elkaar gelijk gemaakt. Hiertoe is vereist dat hun argumenten twee aan twee gelijk worden. Men onderscheidt vier gevallen :

- twee argumenten zijn gelijk : in orde
- minstens een van beide is een variabele : ga over op de variant waarin (alle voorkomens van) de variabele vervangen wordt door het andere argument (een beperking op de mogelijke waarden van de variabele).
- beide zijn samengestelde termen met hetzelfde funktiesymbool : de argumenten van de termen moeten twee aan twee gelijk gemaakt worden
- in alle andere gevallen is unifikatie onmogelijk.

Bij het begin van de resolutie hebben beide uitdrukkingen niets met elkaar te maken en moet men zorgen dat ze verschillende namen gebruiken voor hun variabelen. Het effekt van unifikatie tussen twee termen t_1 en t_2 is zodanig dat er beperkingen opgelegd worden die ervoor zorgen dat t_1 en t_2 in elke mogelijke interpretatie hetzelfde object aanduiden. Men moet er zorg voor dragen dat men geen variabele vervangt door een samengestelde term die de variabele bevat (vb. x door f(x)) : wat ook de beperking is die men aan x oplegt, er blijven interpretaties mogelijk waarin x en f(x) verschillende objecten aanduiden en unifikatie is dus onmogelijk ("occurcheck").

Formeel worden de beperkingen voor de variabelen voorgesteld als $x \leftarrow t$: "x wordt beperkt tot waarden van de vorm t" of "x wordt vervangen door t". Alle beperkingen samen noemt men een substitutie.

Voorbeeld

$A(x,y) \leftarrow B(y,f(x))$

$B(a,u) \leftarrow D(u)$

Het grootste gemene atoom wordt bekomen door de substitutie $\sigma = \{y \leftarrow a,\ u \leftarrow f(x)\}$. De afgeleide uitdrukking is $A(x,a) \leftarrow D(f(x))$. σ noemt men de meest algemene unifikator ("most general unifier").

Als σ de meest algemene unifikator is van B_i en C dan kan men de nieuwe uitdrukking formeel beschrijven als $(A \leftarrow B_1,\ldots,B_{i-1},D_1,\ldots,D_n,B_{i+1},\ldots,B_m)\sigma$

DE BEPERKING TOT HORN-UITDRUKKINGEN

We beperken ons tot Horn-uitdrukkingen omdat we Horn-uitdrukkingen gemakkelijk kunnen interpreteren als procedures en resolutie tussen Horn-uitdrukkingen als programmatietransformaties of als het uitvoeren van procedure-oproepen. Deze interpretatie geeft ons inzicht in wat resolutie precies doet en hoe we resolutie moeten controleren om nuttige resultaten te bekomen. Voor niet Horn-uitdrukkingen missen we dit inzicht momenteel, evenmin is het erg duidelijk voor welk soort problemen we niet-Horn-uitdrukkingen kunnen gebruiken.

De beperking tot Horn-uitdrukkingen doet niets af van de theoretische mogelijkheden van het logisch programmeren. Elke berekenbare funktie kan onder de vorm van Horn-uitdrukkingen geformuleerd worden [Andreka & Nemeti 76]. (Dit zegt niets over de beperking in de expressiemogelijkheden, we komen hierop terug in sektie B.3).

De beperking tot Horn-uitdrukkingen brengt ons terug tot de taal die we beschreven in sektie B.1. We onderscheiden vier types uitdrukkingen :

- $A \leftarrow B_1,\ldots,B_m$ met $m > 0$

  Een "procedure" : om aan te tonen dat A voldoet (waar is) : toon aan dat B_1 en ... en B_m voldoen (waar zijn)

- $A \leftarrow$

  Dit is een procedure met een ledig lichaam of een elementair feit : "A voldoet" of "A is waar"

- $\leftarrow B_1,\ldots,B_n$

  Een probleemstelling : als $x_1,\ldots,x_k$ de variabelen zijn van $B_1 \ldots B_n$ dan kan men dit lezen als "vindt een $x_1,\ldots,x_k$ die een oplossing zijn voor B_1 en ... en B_n". In feite zegt deze uitdrukking : het is niet waar dat B_1 en ... B_n. Het systeem dat contradicties opspoort reageert hierop met het zoeken van waarden voor $x_1,\ldots,x_k$ zó dat B_1 en ... B_n wel waar zijn.

- $\square$

  de ledige uitdrukking, kan geïnterpreteerd worden als "stop, een contradictie is gevonden".

De uitvoering van een programma zoals beschreven in sektie B.1. bestaat uit resoluties tussen de probleemstelling en de procedures. Elke resolutiestap resulteert in een nieuwe probleemstelling. Resoluties tussen procedures onderling kan men beschouwen als programmatransformaties : het afleiden van nieuwe procedures.

B.3. PROLOG

B.3.1. Zuivere PROLOG

De programmeertaal PROLOG werd te Marseille ontwikkeld door A. Colmerauer en P. Roussel [Colmerauer & Co 73]. Een deelverzameling van PROLOG blijft binnen het kader van de logika der Horn-uitdrukkingen en noemen we daarom zuivere PROLOG. Dit deel beschrijven we in deze sektie.

De PROLOG-vertolker is in essentie een automatische stellingbewijzer. Nochtans beschouwen de ontwerpers die stellingbewijzer niet als een magische doos die oplossingen te voorschijn tovert. Integendeel, zij zien de stellingbewijzer als een systeem dat op een mechanische, voorspelbare en begrijpelijke wijze zijn gegevens verwerkt. Bovendien geven zij de gebruiker de mogelijkheid om die verwerking te beïnvloeden, te controleren. Dit gebeurt door middel van een eenvoudige maar toch krachtige controletaal. De beperking tot Horn-uitdrukkingen samen met de eenvoud van de controletaal liet de ontwerpers toe om een eenvoudig en efficiënt systeem op te bouwen.

Een PROLOG-programma bestaat uit een aantal proceduredefinities en één probleemstelling. Resolutie in zichzelf laat heel veel afleidingen toe op een dergelijke verzameling Horn-uitdrukkingen. Een eerste beperking opgelegd door de ontwerpers is dat zij alleen resolutie toelaten tussen de probleemstelling en een proceduredefinitie. Resolutie tussen procedures onderling is uitgesloten. Dit is in feite geen echte beperking, resoluties tussen procedures onderling resulteren in nieuwe procedures en is dus een vorm van programmatransformatie. Het is normaal dat een eenvoudig systeem dergelijke faciliteit niet ter beschikking stelt.

Binnen deze beperking is er nog veel keuze. Een probleemstelling kan meerdere procedureoproepen bevatten. Elk van deze oproepen kan gekozen worden om uit te voeren. Deze keuze kan belangrijk zijn voor de efficiëntie van een programma.

Voorbeeld

← Vader(Jan,x) Vader(x,y)

Het uitvoeren van Vader(Jan,x) zal resulteren in een waarde voor x (vb. Piet). De tweede oproep wordt dan Vader(Piet,y) en zal direct een oplossing geven. Dit is veel beter dan te beginnen met de tweede oproep, inderdaad, die pikt een willekeurig paar x-y op (vb. Jan-Piet). Pas dan wordt nagegaan of de x al of niet voldoet aan de eerste oproep (Vader(Jan,Jan)).

Het is dus noodzakelijk dat de gebruiker deze keuze kan beïnvloeden. Dit gebeurt op een zeer eenvoudige wijze. Het systeem selekteert steeds de linker oproep in de probleemstelling terwijl de nieuwe oproepen afkomstig uit het lichaam van de gebruikte definitie links worden toegevoegd ("depth first left to right selection"). Deze nieuwe oproepen behouden onderling de volgorde die ze hadden in de definitie. De gebruiker controleert dus de selektie door het ordenen van de oproepen in de probleemstelling en de proceduredefinities. Deze selektiemethode stemt overeen met een "sequentiële" uitvoering van procedure-oproepen, pas wanneer een bepaalde oproep volledig afgewerkt is wordt de volgende oproep begonnen.

Nadat beslist is welke procedure-oproep wordt uitgevoerd blijft er soms nog keuze tussen verschillende proceduredefinities. Soms kunnen dus verschillende nieuwe probleemstellingen afgeleid worden. Eén conventionele stellingbewijzer verdeelt zijn aandacht over deze verschillende nieuwe problemen uit vrees dat sommige nooit eindigen. Wordt alle energie aan een dergelijk probleem besteedt, dan wordt nooit een oplossing gevonden. In PROLOG controleert de gebruiker de uitvoering en daarom hebben de ontwerpers beslist alle aandacht te wijden aan de eerste nieuwe probleemstelling. Slechts wanneer die volledig afgewerkt is wordt de tweede afgeleid en behandeld. De volgorde waarin definities gebruikt worden is de volgorde waarin ze door de gebruiker opgegeven werden. Deze techniek is gekend als "sequentiële exploratie" ("backtracking") : slechts wanneer een bepaald pad volledig doorlopen is wordt het volgende pad onderzocht. Door het ordenen der definities bepaalt de gebruiker in welke volgorde de verschillende paden doorlopen en de verschillende oplossingen gevonden worden. Hij moet daarbij zorgen

dat een eventueel oneindig pad slechts onderzocht wordt nadat een eerste oplossing reeds gevonden is. De sequentiële exploratietechniek leidt tot een eenvoudiger implementatie en een besparing van geheugen.

De gebruiker controleert dus de uitvoering op twee manieren : door het ordenen van de proceduredefinities en door het ordenen van de procedure-oproepen in de definities en in de probleemstelling. Deze controle is vrij rudimentair, maar juist daarom gemakkelijk te begrijpen en te gebruiken. Tevens laat ze de logika van het programma ongemoeid, een programma kan nog steeds beschouwd worden als een verzameling Horn-uitdrukkingen. Onafhankelijk van een executiemechanisme kan bewezen worden dat het correct is en dat het theoretisch eindigt (dat er een pad bestaat dat naar een oplossing leidt; praktische eindigheid wil zeggen dat een bepaalde vertolker dat pad inderdaad vindt).

B.3.2. Uitbreidingen

Grafische voorstelling van de uitvoering van een programma

De oplossingsruimte, d.w.z. de verzameling bestaande uit de oorspronkelijke probleemstelling en alle afgeleide probleemstellingen, kan voorgesteld worden als een OF-boom.

Voorbeeld : een permutatieprogramma
Zij Perm (ℓ1,ℓ2) een relatie tussen de lijsten ℓ1 en ℓ2 zodanig dat ℓ2 een permutatie is van ℓ1.
Zij Delete (e,ℓ1,ℓ2) een relatie tussen het element e en de lijsten ℓ1 en ℓ2 zodanig dat ℓ2 uit ℓ1 afgeleid is door het verwijderen van e.

Het programma :
(a) Perm(Nil,Nil) ←
(b) Perm(x.y,u.v) ← Delete(u,x.y,w) Perm(w,v)
(c) Delete(x,x.y,y) ←
(d) Delete(u,x.v,x.w) ← Delete(u,v,w)
← Perm(A.B.Nil,v)

(a) kan gelezen worden als : de lege lijst is een permutatie van de lege lijst,

(b) kan gelezen worden als : de lijst u.v. is een permutatie van de lijst x.y indien er een w bestaat zó dat v een permutatie is van w en w bekomen is door verwijdering van u uit x.y,

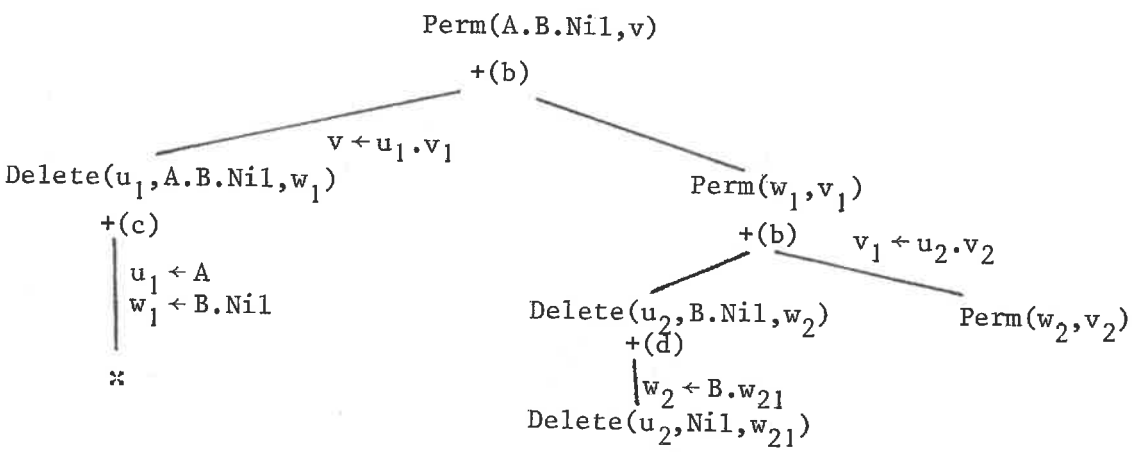
(c) kan gelezen worden als : y is de lijst die overblijft nadat men het element x uit de lijst x.y verwijderd heeft,

(d) kan gelezen worden als : als w de lijst is die overblijft na verwijdering van het element u uit v, dan is x.w de lijst die overblijft na verwijdering van het element u uit x.v.

Een gedeelte van de oplossingsruimte is gegeven in fig. 1.1. De PROLOG-vertolker leidt de probleemstellingen af in de volgorde $C_o, C_1, C_2, C_3, \ldots$ Uit C_1 kan zowel C_2 als C_6 afgeleid worden. (Daarom spreken we van een OF-boom). De vertolker leidt slechts C_6 af wanneer de volledige deelboom met C_2 als wortel afgehandeld is en kan verdwijnen. De vertolker hoeft dus niet de volledige oplossingsruimte bij te houden. Slechts de probleemstellingen gelegen op het pad tussen de wortel en de huidige probleemstelling moeten bijgehouden worden, samen met hun nog niet behandelde vertakkingspunten.

Voorbeeld :
Huidige probleemstelling C_5
C_o, C_1, C_2, C_3 en C_5 liggen op het pad van de wortel tot C_5.
Nog te onderzoeken tak : C_1 met procedure (d).

Een pad van oorspronkelijke tot huidige probleemstelling kan beschreven worden als een EN-boom. Voor C_5 ziet die eruit als volgt :



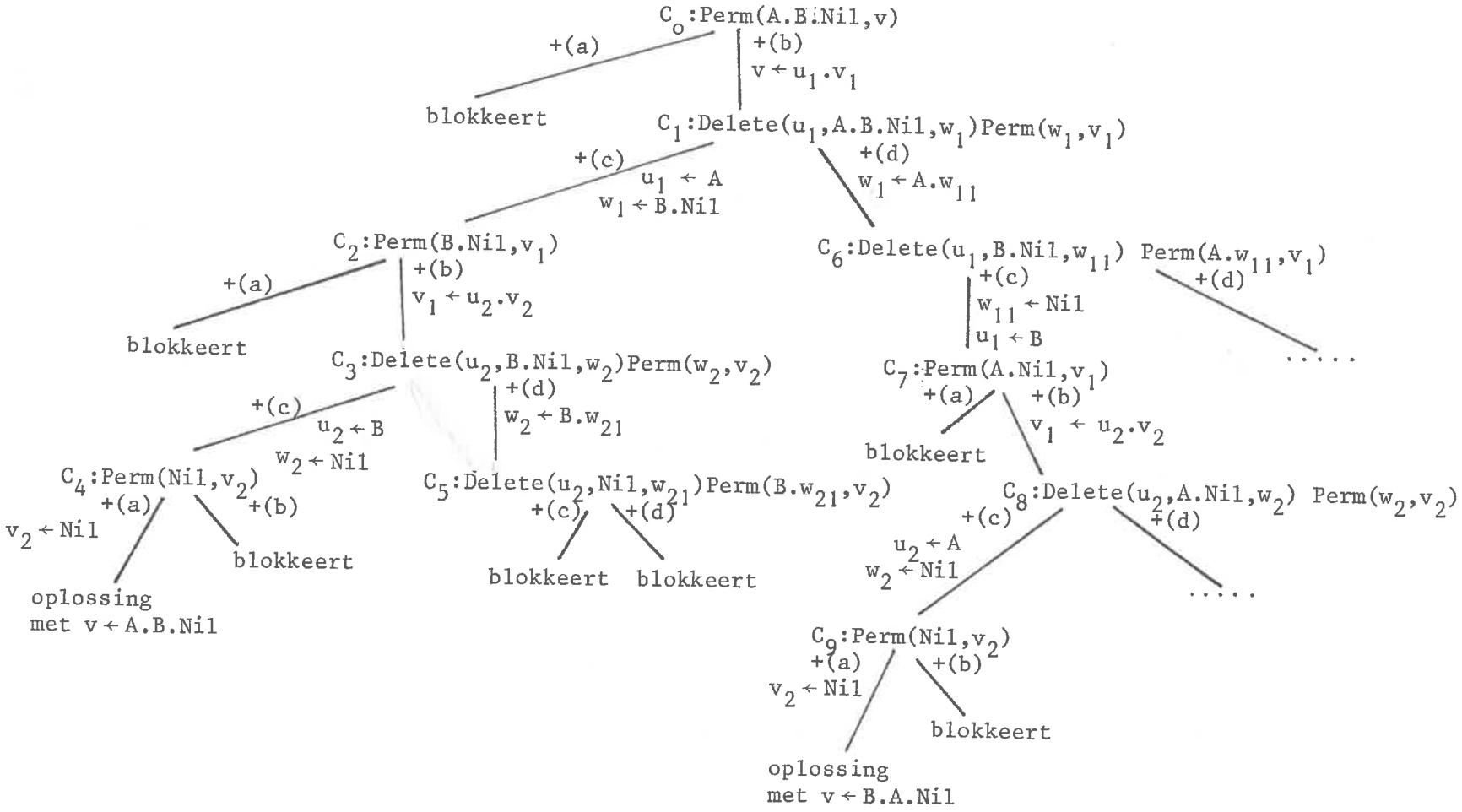


Fig. 1.1 : Een gedeelte van de oplossingsruimte (OF-boom) voor een eenvoudig programma.

De eindknooppunten verschillend van ⁜ beschrijven de huidige probleemstelling. Deze is dus ← Delete(u_2,Nil,w_{21}) Perm(w_2,v_2) met w_2 ← B.w_{21} of ← Delete(u_2,Nil,w_{21}) Perm(B.w_{21},v_2).
Bemerk dat de substituties die in de OF-boom op één pad liggen, hier verspreid zijn over de volledige EN-boom. Daar steeds de linkse oproep geselekteerd wordt beschrijft deze EN-boom eenduidig de afleiding die tot de huidige probleemstelling leidt. De wortel Perm(A.B.Nil,v) stemt overeen met de oorspronkelijke probleemstelling C_o. Samen met procedure (b) expandeert deze in twee oproepen Delete(u_1,A.B.Nil,w_1) en Perm(w_1,v_1) die beiden moeten uitgevoerd worden. (Daarom is dit een EN-boom). Zij vormen C_1 . In de volgende stap wordt Delete uitgevoerd met procedure (c). Deze procedure heeft een ledig lichaam (aangeduid met ⁜) en Perm(w_1,v_1) blijft over; nu echter met w_1 ← B.Nil. C_2 is dus Perm(B.Nil,v_1). Met procedure (b) expandeert dit in C_3=Delete(u_2,B.Nil,w_2) Perm(w_2,v_2). Na toepassing van procedure (d) op Delete bekomen we C_5. Deze EN-boom beschrijft dus ondubbelzinnig de sekwentie van probleemstellingen die aanleiding geeft tot de huidige probleemstelling. Voegen we voor elke oproep een lijst toe met de nog te proberen proceduredefinities (procedure (d) voor Delete(u_1,A.B.Nil,w_1)), dan bevat deze EN-boom alle informatie die de vertolker nodig heeft.

Snoeien in de EN-boom

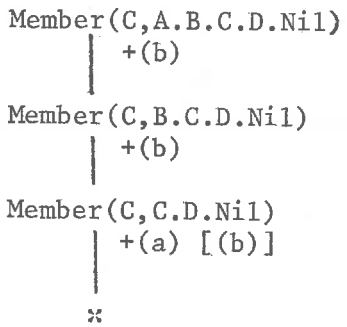
Een eenvoudig programma :

(a) Member(x,x.y) ←

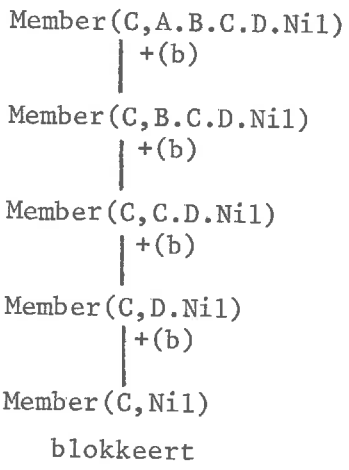
(b) Member(x,y.z) ← Member(x,z)

←Member(C,A.B.C.D.Nil)

De uitvoering van dit programma die tot een oplossing leidt kan als volgt voorgesteld worden :



Member(C,C.D.Nil) is een vertakkingspunt, we kunnen eveneens procedure (b) gebruiken. Dit leidt tot :



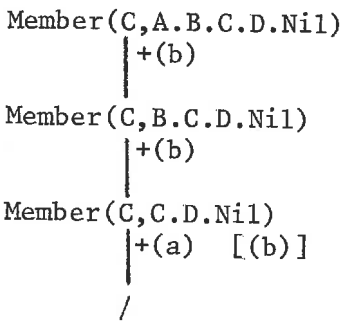
Voor een gebruiker die test of een element in een lijst voorkomt is het meestal onbelangrijk om te weten of het meermaals voorkomt ofwel weet hij dat het slechts éénmaal kan voorkomen. De exploratie van de tweede tak is dus nutteloos. De ontwerpers van PROLOG geven de gebruiker de mogelijkheid om die nutteloze evaluatie te voorkomen en dat vertakkingspunt weg te snoeien. Dit gebeurt door een pseudoprocedure-oproep "/". Het effekt van de uitvoering van "/" is dat alle bestaande vertakkingspunten in de deelboom met de vader van "/" als wortel verdwijnen;

Voorbeeld :

(a) Member(x,x.y) ← /

(b) Member(x,y.z) ← Member(x,z)

←Member(C,A.B.C.D.Nil)



Het effekt van "/" is dat het vertakkingspunt Member(C,C.D.Nil) met procedure (b) verdwijnt. (Ook alle vertakkingspunten die voorkomen in de uitvoering van de eventuele oproepen links van "/" verdwijnen). Het enige effekt van "/" op de uitvoering van bovenstaand programma is dat de nutteloze evaluatie van de andere tak vermeden wordt. Dit is echter niet altijd zo, soms wordt hierdoor ook de logika van het programma gewijzigd. Vervangen we de oproep door ←Member(x,A.B.C.D.Nil) dan zal het programma met "/" slechts één resultaat x←A berekenen, terwijl het oorspronkelijke programma vier resultaten berekent : x←A, x←B, x←C en x←D. De eventuele resultaten zijn nog altijd een logische konsekwentie van de gegevens maar soms worden niet alle resultaten gevonden. Programma's met "/" kunnen slechts volledig begrepen worden wanneer men weet hoe de vertolker werkt. Ze kunnen niet langer beschouwd worden als Horn-uitdrukkingen die een betekenis hebben onafhankelijk van een bepaald executiemechanisme. Sommigen noemen de "/" dan ook de "go to" van PROLOG. Het is een eenvoudig middel om de efficiëntie van PROLOG programma's te verbeteren, maar het maakt de programma's moeilijker om te begrijpen. Meestal heeft men bij het gebruik van de "/" niet de bedoeling de logika van het programma te wijzigen. Nochtans zoals uit bovenstaand voorbeeld blijkt, wordt de logika slechts bewaard voor bepaalde input-output patronen (input : de gegeven argumenten, output : de te berekenen argumenten) van de procedureoproepen. Deze input-output patronen worden niet expliciet gemaakt en nog minder wordt gecontroleerd of alle oproepen eraan voldoen. Wellicht ware het wenselijk andere, nettere concepten te ontwikkelen om hetzelfde resultaat te bekomen. Het expliciet maken van de input-output patronen kan toelaten om formeel of informeel aan te tonen dat de andere procedures tot mislukken gedoemd zijn en om, tijdens de uitvoering, het programma te onderbreken met een "controlefout" wanneer een oproep niet voldoet aan dit patroon.

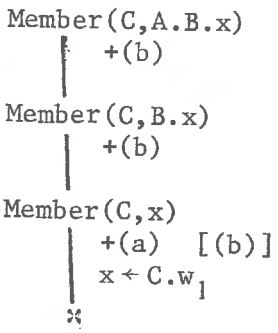
Soms is het gebruik van "/" meer subtiel, nemen we dezelfde procedure :

(a) Member(x,x.y) ←

(b) Member(x,y.z) ← Member(x,z)

met een oproep Member(C,A.B.x)

De uitvoering



leidt tot een oplossing met $x \leftarrow C.x_1$. Member(C,x) kan echter ook met procedure (b) uitgevoerd worden en in feite heeft dit programma een oneindig aantal oplossingen, namelijk $x \leftarrow x_1.C.x_2$, $x \leftarrow x_1.x_2.C.x_3$ ... $x \leftarrow x_1.x_2 \ldots .x_n.C.x_{n+1}$. Het selekteren van procedure (b) voor procedure (a) zou zelfs aanleiding geven tot een oneindig pad. Opnieuw kan de "/" gebruikt worden om ervoor te zorgen dat de vertolker slechts één oplossing zoekt. Indien procedure (a) vervangen wordt door Member(x,x.y) ← / worden de andere oplossingen weggesnoeid.

Negatie

De beperking tot Horn-uitdrukkingen vermindert de expressieve mogelijkheden van de taal. Men kan proberen om deze beperking te omzeilen door het toelaten van negatieve atomen in het lichaam van een procedure.

Voorbeeld :
In de plaats van de uitdrukking
Different(x,y) V Equal(x,y)
kan men de pseudo-Horn-uitdrukking
Different(x,y) ← ∿Equal(x,y) schrijven

Met Equal(x,x)← gegeven, kunnen we daarmee ←Different(A,B) bewijzen ?

Uitvoering van de procedure Different leidt tot ←∿Equal(A,B). We kunnen de oproep Equal(A,B) uitvoeren. Dit mislukt. Intuïtief zijn we geneigd om daaruit te besluiten dat ∿Equal(A,B) waar is en dus ook Different(A,B) waar is. Dit suggereert dat we negatieve atomen de baas kunnen.

Proberen we ook eens ←Different(A,x). Dit herleidt tot ←∿Equal(A,x). Voeren we de oproep Equal(A,x) uit, dan vinden we een oplossing(x←A). Daaruit besluiten dat ∿Equal(A,x) niet waar is en dus ook Different(A,x) niet waar is, botst met onze intuïtie (vb. x←B).

Onder welke voorwaarden kunnen we negatieve atomen toelaten in procedures ? Keith Clark heeft dit probleem bestudeerd [Clark 78]. We geven hier de belangrijkste resultaten.

We hebben daarnet nogal losjes omgesprongen met de logika. Uit

Different(x,y) ∨ Equal(x,y) ←

Equal(x,x) ←

en ←Different(A,B)

kunnen we geen contradictie afleiden. Daartoe is ook ∿Equal(A,B) vereist. We hebben intuïtief verondersteld dat Equal(x,x)← de relatie Equal volledig definieert. In andere woorden, voor elk paar x-y waarvoor Equal(x,y) niet te bewijzen is, hebben we verondersteld dat ∿Equal(x,y) geldig is. De veronderstelling dat elke relatie volledig gedefinieerd is, is gekend als de "closed world assumption". Zonder deze veronderstelling kunnen we uit het feit dat het bewijs van Equal(A,B) blokkeert niet besluiten dat Equal(A,B) niet waar is, het kan evengoed waar zijn en behoren tot het niet gedefinieerde deel van de relatie.

Een tweede probleem betreft negatieve atomen die veranderlijken bevatten. Dus een oproep van de vorm ←∿Equal(A,x) : "vind een x zó dat het paar A-x niet aan de relatie Equal voldoet". Een meer expliciete notatie is ←∃x ∿Equal(A,x). Een PROLOG-vertolker kan een dergelijk probleem niet aan maar wel het afgeleide probleem ←Equal(A,x) of meer expliciet ←∃xEqual(A,x) : "vind een x zó dat het paar (A,x) aan de relatie Equal voldoet".

Dit afgeleide probleem heeft drie mogelijke antwoorden :

1. Geen enkele x voldoet, de poging om Equal(A,x) te bewijzen mislukt.
   dus ∿∃xEqual(A,x) : "het is niet waar dat er een x bestaat zó dat het paar A-x aan de relatie Equal voldoet".
   of ∀x ∿Equal(A,x) is waar
   Het antwoord op ←∃x ∿Equal(A,x) is "ja elke x".
   Uit het blokkeren van het bewijs ←Equal(A,x) kunnen we correct besluiten dat ∿Equal(A,x) waar is.

2. Elke x voldoet
   of $\forall x$ Equal(A,x) is waar
   dus $\sim\forall x$ Equal(A,x) is niet waar
   en ook $\exists x \sim$Equal(A,x) is niet waar.
   Als $\leftarrow$Equal(A,x) kan bewezen worden zonder aan x een waarde toe te kennen kunnen we correct besluiten dat $\leftarrow\sim$Equal(A,x) niet waar is en de procedure die deze oproep bevat blokkeert.

3. Sommige x voldoen bijvoorbeeld $x \leftarrow A$
   dus $\exists x$ Equal(A,x) is waar
   Daaruit kunnen we niet besluiten dat $\exists x \sim$Equal(A,x) niet waar is !
   Inderdaad voor andere waarden van x bijvoorbeeld $x \leftarrow B$ kan deze uitdrukking waar zijn. Een systeem dat niet in staat is om "andere" waarden te genereren is niet in staat om dit laatste geval te behandelen.

Opmerkingen

1. Een antwoord van de vorm "elke x voldoet" is eerder uitzonderlijk tenzij wanneer men test of een element in een bepaalde relatie staat met zichzelf zoals in $\leftarrow$Equal(x,x). Bevat de relatie slechts één voorkomen van x, dan gaat het om een relatie waaraan elk mogelijk object voldoet.
2. Een konstruktief antwoord in geval 3 is praktisch onmogelijk. Er zijn een oneindig aantal objecten verschillend van A. De vraag in zichzelf heeft ook weinig zin tenzij men de waarden van x beperkt tot een bepaald domein vb. $\leftarrow$Persoon(x) $\sim$ Vader(Jan,x). Een (positieve) procedure oproep genereert zinvolle waarden voor de variabele x.

Clark schrijft : "every variable appearing in a negated condition should have its range specified by some unnegated condition".

Clark [Clark 78] bewijst dat dit redeneren op bewijzen ("metalevelproof") equivalent is met een bewijs in de predikatenlogika van eerste orde met alle veronderstellingen expliciet gemaakt ("object level proof").

We kunnen besluiten dat het gerechtvaardigd is om een negatieve conditie te verwerken door de positieve conditie te geven aan de PROLOG-vertolker en zijn besluit (geslaagd-mislukt) om te keren onder de voorwaarden dat - de relatie volledig gedefinieerd is.

- de PROLOG-vertolker geen waarden toekent aan de variabelen die voorkomen in de relatie (verifikatie, geen generatie).

Die tweede voorwaarde kan eventueel vervangen worden door een strengere voorwaarde : op het ogenblik dat de negatieve conditie uitgevoerd wordt bevat ze geen vrije variabelen.

Daar procedureoproepen dezelfde vorm hebben als termen laat PROLOG een eenvoudige definitie toe van een procedure "Not" :

Not(x) ← x / Fail

Not(x) ←

Voorbeeld van een oproep : ←Not(Vader(Jan,Piet)). Kan Vader(Jan,Piet) bewezen worden, dan wordt de pseudo-oproep "/" uitgevoerd. Deze verhindert dat de tweede procedure ooit gebruikt wordt. Daarna wordt de oproep Fail uitgevoerd, dit moet een niet gedefinieerde procedure zijn. Deze blokkeert dus en de hele oproep Not(Vader(Jan,Piet)) blokkeert. Blokkeert de oproep Vader(Jan,Piet), dan wordt de tweede procedure uitgevoerd en de conclusie is dat Not(Vader(Jan,Piet)) waar is.

Deze definitie van "Not" controleert de bovenvermelde tweede voorwaarde niet en kan dus aanleiding geven tot zware fouten. Frank McGabe heeft, in een versie ontwikkeld te London [McGabe 78] "Not" geimplementeerd als een pseudoprocedure die de vertolker oproept met de positieve conditie en erover waakt dat geen waarden toegekend worden aan variabelen in de oproep. Een tussenoplossing zou erin bestaan om een pseudoprocedure "ground" te voorzien die het programma onderbreekt met een "controlefout" wanneer de negatieve conditie vrije variabelen bevat (te strenge voorwaarde) :

Not(x) ← ground(x) x / Fail

Not(x) ←

Dit gebruik van Not verhoogt sterk de expressieve mogelijkheden van de taal, het maakt evenwel de logika der Horn-uitdrukkingen niet equivalent met de algemene logika. Clark toont dat sommige afleidingen on-

mogelijk zijn. Meer formeel : de uitbreiding van de Horn-uitdrukkingen met negatie als "onmogelijk te bewijzen" is geen "volledig" afleidingssysteem voor eerste orde logika.

Manipulatie van resultaten

In de bovenstaande verwezenlijking van "Not" wordt het resultaat van een deelprobleem (de oproep x in Not(x) ← ground(x) / Fail) getransformeerd en gebruikt in het probleem dat "Not" oproept. (Een sukses wordt omgezet in een mislukking en omgekeerd). Ondanks het gebruik van pseudoprocedures bekomt men een eenvoudig en logisch zuiver concept : de negatie. Soms is het wenselijk om de resultaten van een deelprobleem op een meer algemene wijze te kunnen benutten.

Voorbeeld :

Gegeven een relatie : Salaris

Salaris(Peeters, 35.870) ←

Salaris(Jansens, 24.300) ←

⋮

We kunnen een relatie Goedsalaris definiëren als volgt : (personen met een goed salaris - hun salaris)

Goedsalaris(x,y) ← Salaris(x,y) Groter(y,30.000). Alle bedienden met een goed salaris bekomen we door de oproep ←Goedsalaris(x,y).

Een vraag als :

"hoeveel bedienden zijn er met een goed salaris ?".

of

"wat is het gemiddeld salaris van de bedienden met een goed salaris ?".

is echter heel wat moeilijker op te lossen met Horn-uitdrukkingen.

Indien wij zouden beschikken over een manier om alle antwoorden te verzamelen in een lijst dan zou een dergelijk probleem veel eenvoudiger zijn. Daarom suggereert de auteur een procedure "Bag" die gebruikt kan worden om :

- een lijst te maken van alle bedienden met een goed salaris :

  ←Bag(Goedsalaris(x,y),x,ℓ) : het resultaat is een lijst ℓ (Peeters. ... .Nil) van de component x uit de oplossingen van Goedsalaris(x,y).

- een lijst te maken van goedbetaalde bedienden met hun salaris :

←Bag(Goedsalaris(x,y),g(x,y),ℓ) : het resultaat is een lijst ℓ

← g(Peeters,35.870). ... .Nil.

In het algemeen is Bag van de vorm Bag(vraag,antwoord,lijst) met "vraag" een procedureoproep, "antwoord" een term en "lijst" een vrije variabele. Het effekt is dat een "lijst" van "antwoord-"en op de "vraag" wordt opgebouwd. Deze "lijst" kan dan gebruikt worden voor verdere berekeningen (aantal, gemiddelde, ...).

Net zoals bij Not is de controle (het ordenen der oproepen) erg belangrijk.

←Bag(Salaris(x,y),y,ℓ) Equal(x,Peeters)

en

←Equal(x,Peeters) Bag(Salaris(x,y),y,ℓ)

zullen aanleiding geven tot een totaal verschillende ℓ.

Dubbelzinnigheid kan vermeden worden door te eisen dat de variabelen die voorkomen in de eerste twee argumenten van de "Bag"-oproep nergens anders gebruikt worden. Bovenstaand voorbeeld moet dan herschreven worden als :

R(x,y) ← Equal(x,Peeters) Salaris(x,y)

←Bag(R(x,y),y,ℓ).

Het effekt van een oproep Bag(vraag,antwoord,ℓ) is dan dat de vertolker de oproep "vraag" uitvoert en dat de resultaten beschikbaar komen in de lijst ℓ. De wisselwerking tussen Bag en andere oproepen is beperkt tot deze resultatenlijst ℓ.

Met behulp van enkele pseudoprocedures heeft de auteur deze "Bag" verwezenlijkt in een experimentele PROLOG-implementatie. De definitie is als volgt :

Bag(vraag,antwoord, lijst) ← Openbag(lijst)
B(vraag,antwoord)

B(vraag,antwoord) ← vraag Writebag(antwoord) Fail

B(vraag,antwoord) ← Closebag(vraag)

Met :

Openbag(lijst) : creatie van "lijst", de speciale variabele waar de antwoorden moeten bewaard worden.

Writebag(antwoord) : antwoord wordt gekopieerd op "lijst", deze kopie verdwijnt niet wanneer onmiddellijk daarna (uitvoering Fail) de vertolker terugkeert om een ander pad in "vraag" te exploreren.

Closebag(vraag) : sluit "lijst" of met Nil, zorgt ervoor dat "lijst" een normale variabele wordt voor de procedure die "Bag" opriep.

Unifikatie-circulaire gegevensstrukturen

Toen we het hadden over unifikatie vermeldden we dat unifikatie tussen een variabele en een term die deze variabele bevat, onmogelijk is omdat het beide termen niet gelijk maakt maar een (oneindige) term creëert die naar zichzelf verwijst. Het vraagt relatief veel tijd om het unifikatie-algoritme dergelijke gevallen te laten opsporen. Daarenboven is de situatie vrij zeldzaam in PROLOG-programma's. In de oorspronkelijke implementatie is de test dan ook achterwege gelaten. Nochtans vindt men in [Tärnlund 78] een programma waar de test ("occurcheck" genoemd) essentieel is. Hij maakt daarbij gebruik van een "verschillijst" d(k,ℓ). De elementen van de verschillijst d(k,ℓ) zijn de elementen van de lijst die men bekomt door ℓ van k af te trekken, vb. d(a.b.c.z,z) bevat de elementen a, b en c. Zijn programma, dat een lijst omkeert is als volgt :

Reverse(d(u,u),d(x,x)) ←

Reverse(d(u,w),d(v.x,y)) ← Reverse(d(u,v.w),d(x,y))

← Reverse(d(a.b.w_1,w_1),d(x_1,y_1))

Unifikatie tussen de oproep en de eerste procedure creëert de circulaire struktuur $w_1 \leftarrow a.b.w_1$. De tweede procedure geeft :

← Reverse(d(a.b.w_1,v_2.w_1),d(x_2,y_1)) en $x_1 \leftarrow v_2.x_2$.

Nogmaals de tweede procedure toepassen resulteert in

← Reverse(d(a.b.w_1,v_3.v_2.w_1),d(x_3,y_1)) en $x_2 \leftarrow v_3.x_3$

Met de eerste procedure wordt het probleem nu opgelost, de bindingen zijn $v_3 \leftarrow a$, $v_2 \leftarrow b$, $x_3 \leftarrow y_1$ en dus $x_1 \leftarrow b.a.y_1$.

Een rigoureuze implementatie moet de nodige controles uitvoeren (telkens wanneer een variabele voor de tweede maal ontmoet wordt in de hoofding van een procedure kan er een circulaire struktuur ontstaan). Een compromis tussen correctheid en efficiëntie bestaat erin de gebruiker de beschikking te geven over een pseudoprocedure die controleert of een struktuur al of niet circulair is en alle verantwoordelijkheid voor het uitvoeren van de nodige testen over te laten aan de gebruiker. De eerste procedure van Reverse zou dan moeten geschreven worden als Reverse(d(u,u),d(x,x)) ← occur(u).

Occur(u) zal de procedure doen blokkeren wanneer u circulair is.

Pseudoprocedures

We hebben het reeds meermaals gehad over pseudoprocedures. Ze werden niet als Horn-uitdrukkingen gedefinieerd, maar werden beschouwd als (primitieve) procedures die in de vertolker ingebouwd zijn. Daarenboven beschrijven ze niet altijd een relatie of eigenschap maar verwezelijken soms neveneffecten en wijzigen soms de verdere uitvoering van het programma. Meestal is hun effect afhankelijk van het ogenblik van uitvoering en dus afhankelijk van de controlestruktuur van het programma. In zekere zin maken deze pseudoprocedures het verschil uit tussen zuivere PROLOG en een praktisch bruikbaar systeem. Nochtans lijkt het ons belangrijk dat ze zoveel mogelijk de logika van het programma onaangetast laten en, zo dit niet het geval dreigt te zijn, het programma onderbreken met een controlefout. Voor een gedetailleerde beschrijving van de pseudoprocedures van diverse PROLOG-systemen verwijzen we naar hun handleidingen. Toch willen we even uitweiden over de belangrijkste groepen nog onbesproken primitieve procedures.

a. Rekenkundige relaties

Rekenkundige relaties zoals opstelling, aftrekking, vermenigvuldiging, deling,... kunnen gedefinieerd worden door middel van procedures.

```
Plus(0,0,0) ←
Plus(0,1,1) ←
   ⋮
```

De volledige relatie definiëren vergt echter een oneindig aantal procedures. Zelfs het deel definiëren dat voor de oplossing van een bepaald probleem vereist is, is praktisch onmogelijk. Daarom is het wenselijk dat de vertolker beschikt over een aantal primitieve procedures om dergelijke relaties te verwezenlijken. Dergelijke procedures zijn deterministisch en geven een controlefout wanneer zij een niet-deterministische oproep krijgen (vb. Plus(x,y,24)). Een minimaal stel primitieve procedures omvat :

Plus(x,y,z) : de relatie $x+y=z$

Deel(dt,dℓ,q,r) : de relatie $dt = dℓ \times q + r$

kleiner(x,y) : de relatie $x < y$ en

getal(x) : de eigenschap x is een getal.

Ze onderbreken eveneens het programma met een fout wanneer hun argumenten geen natuurlijke getallen zijn. Andere relaties kunnen in termen van deze primitieve relaties gedefinieerd worden :

Min(x,y,z) ← Plus(y,z,x)

Vermenigvuldiging(x,y,z) ← Deel(z,x,y,0)

Kleiner-of-gelijk(x,y) ← kleiner(x,y)

Kleiner-of-gelijk(x,x) ←

Al deze procedures laten de logika van een programma onaangetast. Ze kunnen enkel, wanneer de controlestruktuur niet voldoet, het programma onderbreken met een controlefout.

b. Lezen van gegevens-- uitschrijven van resultaten

Wanneer we een vertolker een procedure Sorteer een een probleemstelling Sorteer(3.2.4.Nil,ℓ) geven, dan wensen we niet te weten of dit al dan niet aanleiding geeft tot een contradictie, maar willen we de waarde van ℓ kennen waarvoor die contradictie ontstaat. Een overeenkomst om, telkens een oplossing gevonden wordt, de waarden van de variabelen in de oorspronkelijke probleemstelling uit te schrijven, is weinig soepel. Eveneens kan het nuttig zijn om bepaalde argumenten in te lezen. Een eenvoudige oplossing voor dit probleem is het voorzien van een aantal primitieve procedures zoals Leesterm, Schrijfterm, Nieuwelijn, ... Bovenstaande probleemstelling zou dan kunnen geformuleerd worden als :

← Leesterm(k) Schrijfterm("gegeven:") Schrijfterm(k)
Sorteer(k,ℓ) Nieuwelijn Schrijfterm("na sorteren:") Schrijfterm(ℓ)
Een mogelijk resultaat zou zijn :
gegeven : 3.2.4.Nil
na sorteren : 2.3.4.Nil

Deze oplossing is wel eenvoudig, maar er zijn een aantal bezwaren. Een poging tot oplossing kan doodlopen. Eer dit ontdekt wordt kunnen reeds gedeeltelijke resultaten uitgeschreven zijn. Tevens, en dit is erger, kunnen bepaalde gegevens reeds gelezen zijn. Wanneer blijkt dat de huidige interpretatie van de gegevens verkeerd is kunnen ze niet opnieuw gelezen worden. Om logische fouten te vermijden moet de gebruiker er zorg voor dragen dat alle gegevens gelezen worden vooraleer de oplossing van het probleem zich uitsplitst in alternatieve paden.

Een conceptueel zuivere oplossing kan als volgt bekomen worden :

←Input(i1) Output(o1)
Leesterm(k,i1,i2) Schrijfterm("gegeven:",o1,o2)
Schrijfterm(k,o2,o3) Sorteer(k,ℓ)
Schrijfterm(NL,o3,o4) Schrijfterm("na sorteren:",o4,o5)
Schrijfterm(ℓ,o5,o6).

Het effect van input(i1) is dat, voor de logika van het programma, alle gegevens worden gelezen en op de lijst i1 geplaatst worden. Dit wil niet zeggen dat de gegevens werkelijk gelezen worden, dit kan uitgesteld worden tot op het ogenblik dat ze werkelijk gebruikt worden. Output(o1) zegt aan de vertolker dat, wanneer een oplossing gevonden is, alle elementen van de lijst o1 moeten uitgeschreven worden. Leesterm(k,i1,i2) stelt i1←k.i2 (zonodig wordt nu een gegeven gelezen) k is het "gelezen" gegeven en i2 is de lijst waaruit het volgende gegeven moet gelezen worden. Schrijfterm(k,o2,o3) stelt o2←k.o3, dus het element k wordt toegevoegd op de resultatenlijst en de volgende toevoeging moet gebeuren op de deellijst o3. (NL betekent dat een nieuwe lijn moet genomen worden tijdens het uitschrijven). Deze procedures Input, Output, Leesterm en Schrijfterm laten de logische betekenis van een programma intakt. (Schrijfterm kan gedefinieerd worden als : Schrijfterm(t,t.o,o)← Leesterm echter niet als Leesterm(t,t.i,i)← omdat t eventueel moet gelezen worden).

c. De wisselwerking tussen gebruiker en vertolker

De vertolker krijgt als gegevens een aantal proceduredefinities en een probleemstelling. Deze gegevens kan de gebruiker voorbereiden met behulp van een editieprogramma. De vertolker bestaat dan uit twee belangrijke delen. Een eerste deel leest en analyseert de gegevens en bereidt de eigenlijke uitvoering voor. Het tweede deel zorgt voor de uitvoering van het programma. Een wijziging van het systeem vraagt een wijziging van het editieprogramma (geschreven in een andere taal).

De ontwerpers van PROLOG [Colmerauer & Co 73] hebben een andere methode toegepast. Het deel dat het programma inleest en analyseert (=supervisieprogramma) werd in PROLOG zelf geschreven. De vertolker voert een PROLOG-

programma uit dat resideert in een bijhorende "gegevensbank". Dit supervisieprogramma nu leest en analyseert Horn-uitdrukkingen. Gaat het om een proceduredefintie, dan wordt deze toegevoegd in de gegevensbank, gaat het om een probleemstelling, dan wordt de controle tijdelijk afgestaan aan deze probleemstelling en wordt het gebruikersprogramma uitgevoerd. Wijzigingen kunnen nu aangebracht worden door het supervisieprogramma te wijzigen (slechts één taal : PROLOG). Deze aanpak veronderstelt een aantal primitieve procedures voor het "editeren" (toevoegingen en weglatingen) van de gegevensbank. Met behulp van deze procedures kan men het supervisieprogramma wijzigen. Tevens laat dit toe om programma's te schrijven die zichzelf wijzigen. Dit opent de deur voor duistere praktijken : de resultaten van een dergelijk programma zijn niet langer een logische konsekwentie van de gegevens (proceduredefinities).

Het supervisieprogramma beschouwt de gegevens als een lijst van tekens en moet in staat zijn om een dergelijke lijst om te zetten in een term. Bijvoorbeeld om de lijst J.A.N.Nil om te zetten in de term JAN. Ook hiervoor zijn een aantal pseudo-procedures vereist. Diezelfde procedures zijn meteen ter beschikking van gebruikers die tekst willen verwerken.

HOOFDSTUK 2 : PROLOG-VERTOLKERS

A. INLEIDING

Om de implementatieproblemen te verduidelijken herhalen we nog eens hoe PROLOG de uitvoering van een logisch programma controleert.

De uitvoering start met de gegeven probleemstelling als aktieve probleemstelling. De basiscyclus van de vertolkers is als volgt :

- selekteer de linkse oproep in de aktieve probleemstelling (de oproepen zijn geordend)
- voer deze oproep uit met de eerste procedure waarmee deze oproep overeenstemt (de procedures zijn geordend). Hierbij worden probleemstelling en gebruikte procedure nader gespecifieerd (unifikatie-substitutie) zodanig dat oproep en procedurehoofding gelijk worden aan hun grootste gemene struktuur.
- de nieuwe aktieve probleemstelling wordt bekomen door de uitgevoerde oproep te vervangen door de oproepen uit het lichaam van de gebruikte procedure. (Deze oproepen, die geordend zijn, worden dus links toegevoegd).

Deze cyclus wordt herhaald totdat :

- een oplossing gevonden is (de aktieve probleemstelling bevat geen oproepen)

of

- geen enkele procedure overeenstemt met de uit te voeren oproep.

Op dat ogenblik keert men terug naar de vorige aktieve probleemstelling en probeert men om de geselekteerde oproep uit te voeren met de volgende overeenstemmende procedure (sequentiële exploratie).

Zoals gezegd kan de uitvoering geïllustreerd worden met een EN-boom. Deze EN-boom laat ons toe om de implementatieproblemen te verduidelijken. Daarom bestuderen we de ontwikkeling van de EN-boom voor een eenvoudig voorbeeld.

Het programma :

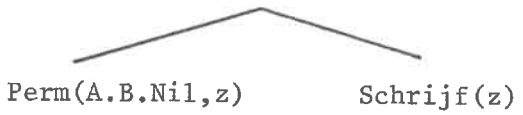
(a) Perm(Nil,Nil) ←

(b) Perm(x.y,u.v) ← Delete(u,x.y,w) Perm(w,v)

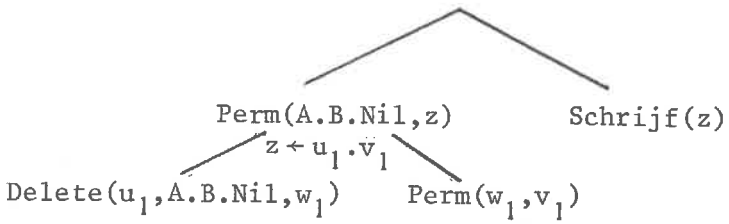
(c) Delete(x,x.y,y) ←

(d) Delete(u,x.y,x.z) ← Delete(u,y,z)
← Perm(A.B.Nil,z) Schrijf(z)

De initiële EN-boom is als volgt :



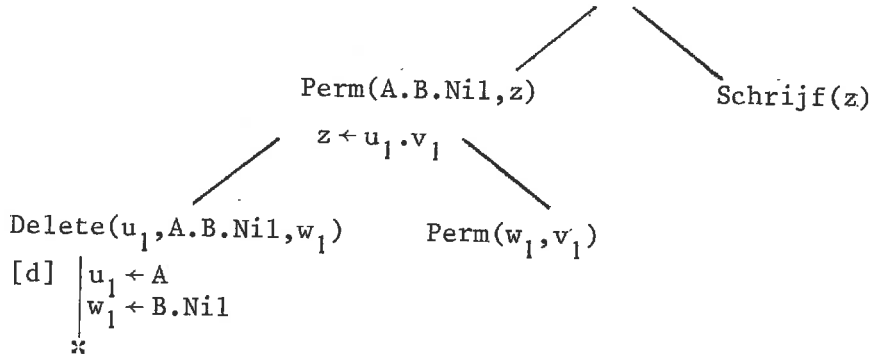
De linker oproep (Perm) stemt overeen met procedure (b). Om verwarring te vermijden is het essentieel dat de probleemstelling en de gebruikte definitie verschillende namen hebben voor hun variabelen. Om alle conflicten te vermijden nemen we een kopie van de gebruikte definitie : $Perm(x_1.y_1,u_1.v_1) \leftarrow Delete(u_1,x_1.y_1,w_1)\ Perm(w_1,v_1)$.
De substitutie $\{x_1 \leftarrow A,\ y_1 \leftarrow B.Nil,\ z \leftarrow u_1.v_1\}$ maakt oproep en hoofding aan elkaar gelijk. Deze substitutie kan opgesplitst worden in een substitutie voor <u>globale</u> variabelen (variabelen uit de probleemstelling) $\{z \leftarrow u_1.v_1\}$ en een substitutie voor <u>lokale</u> variabelen (deze uit de gebruikte definitie) $\{x_1 \leftarrow A, y_1 \leftarrow B.Nil\}$. Nu wordt de EN-boom als volgt uitgebreid :



De uitgevoerde oproep Perm(A.B.Nil,z) creëert de oproepen $Delete(u_1,A.B.Nil,w_1)$ en $Perm(w_1,v_1)$. Deze laatsten zijn uit de gebruikte definitie afgeleid door toepassing van de lokale bindingen. Dit vereenvoudigt onze voorstelling : we kunnen de lokale bindingen weglaten. De globale binding wordt in het knooppunt van de uitgevoerde oproep bewaard en geldt overal waar de betreffende variabele voorkomt. (Door de globale binding niet toe te passen vermijden we dat het reeds bestaande deel van de EN-boom gewijzigd wordt). De nieuwe probleemstelling wordt bekomen door de globale bindingen toe te passen op de eindknooppunten van de EN-boom en is dus :

$\leftarrow Delete(u_1,A.B.Nil,w_1)\ \ Perm(w_1,v_1)\ Schrijf(u_1.v_1)$

In de volgende stap wordt de procedure Delete uitgevoerd met Delete(x_2,x_2.y_2,y_2) ← , een kopie van procedure (c). Dit resulteert in :



De lokale bindingen $x_2 \leftarrow u_1$ en $y_2 \leftarrow$ B.Nil zijn terug weggelaten. In het knooppunt Delete noteren we dat ook procedure (d) in aanmerking komt om deze oproep uit te voeren. De nieuwe probleemstelling is nu : ←Perm(B.Nil,v_1) Schrijf(A.v_1).
De vorige probleemstelling kan hersteld worden door het weglaten van de substitutie in het knooppunt Delete en van de takken vertrekkend uit dat knooppunt.

Hiermee is het mechanisme hopelijk voldoende geïllustreerd. Dit mechanisme implementeren is de taak van de ontwerper van een vertolker. Maar, waar wij streefden naar een zo eenvoudig mogelijke voorstelling moet de ontwerper streven naar een zo efficiënt mogelijk ontwerp.

B. HET PRINCIPE VAN DE OORSPRONKELIJKE PROLOG-VERTOLKER

PROLOG en zijn eerste vertolker werden te Marseille ontwikkeld door A. Colmerauer en P. Roussel [Colmerauer & Co 73]. PROLOG is een bijprodukt van het onderzoek over het automatisch bewijzen van stellingen. Die eerste implementatie is dan ook gebaseerd op technieken gebruikt bij het implementeren van automatische stellingbewijzers. Het belangrijkste probleem bij dergelijke implementaties is het naamconflict tussen variabelen in verschillende uitdrukkingen. In sektie A hebben wij dat conflict opgelost door het geven van unieke namen aan de variabelen in de kopie van de gebruikte proceduredefinitie. Een andere oplossing kon erin bestaan dat men, vooraleer een oproep uit te voeren, unieke namen geeft aan alle

variabelen in de probleemstelling. Dit zou echter een meer ingewikkelde voorstelling gegeven hebben : de nieuwe toestand zou niet langer een uitbreiding zijn van de oude toestand.

Bij automatische stellingbewijzers is het naamconflict moeilijker om op te lossen. Daar heeft men niet alleen resolutie tussen een gegeven uitdrukking (proceduredefinitie) en een afgeleide uitdrukking (probleemstelling), maar ook tussen afgeleide uitdrukkingen onderling. Daarenboven kan dezelfde afgeleide uitdrukking meermaals gebruikt worden.

Bij de eerste automatische stellingbewijzers gaf men verschillende namen aan de variabelen in de verschillende gegeven uitdrukkingen. Telkens een uitdrukking afgeleid werd kregen haar variabelen nieuwe unieke namen. Alle uitdrukkingen kregen aldus unieke namen voor hun variabelen. Tevens werden de afgeleide uitdrukkingen op dezelfde wijze voorgesteld als de gegeven uitdrukking. Deze eerste stellingbewijzers maakten dus geen gebruik van de gelijkenis in struktuur tussen de afgeleide uitdrukking en de uitdrukkingen waaruit ze afgeleid is (Bij onze voorstelling in sektie A heeft elke nieuwe afgeleide uitdrukking een stuk gemeenschappelijk met de vorige afgeleide uitdrukking). Deze voorstelling van uitdrukkingen had voor gevolg dat die eerste stellingbewijzers enorm veel geheugen verbruikten.

Boyer en Moore [Boyer & Moore 72] brachten een zeer belangrijke verbetering aan. Hun methode is gekend als "gedeelde strukturen" ("structure sharing"). Zij stellen uitdrukkingen niet expliciet voor als expressies maar impliciet als paren bestaande uit een skelet en een omgeving. Het skelet is als een uitdrukking zoals wij die kennen, maar de variabelen in die uitdrukking hebben (eventueel) een waarde ; die waarde kan men terugvinden in de omgeving. Skelet verwijst hier naar de "zuivere code" van de uitdrukking; verder zullen we de term "skelet" ook gebruiken voor elke verwijzing naar een onderdeel van die zuivere code. Om de expliciete uitdrukking te bekomen moet men de variabelen in het skelet vervangen door hun waarde. (Voor de gegeven uitdrukkingen is de omgeving ledig, geen enkele van de variabelen heeft een waarde). Wordt nu een uitdrukking afgeleid, dan wordt het skelet impliciet voorgesteld, door verwijzingen naar de twee samenstellende uitdrukkingen.

Voorbeeld : $C_1 = A \leftarrow B_1 \ldots B_i \ldots B_m$

$C_2 = C \leftarrow D_1 \ldots D_n$

Indien $B_i\theta \equiv C\theta$ dan kan men

$$C_{12} = (A \leftarrow B_1 \ldots B_{i-1}\ D_1 \ldots D_n\ B_{i+1} \ldots B_m)\theta$$

afleiden.

De afgeleide uitdrukking C_{12} bevat de skeletten $A, B_1 \ldots B_{i-1}, B_{i+1}, \ldots, B_m$ van de uitdrukking C_1 en $D_1 \ldots D_n$ van de uitdrukking C_2.

Om het naamconflict op te lossen maakt men tijdens het uitvoeren van de resolutie een onderscheid tussen variabelen uit de eerste en de tweede uitdrukking. Het mechanisme dat de waarde van een variabele ophaalt kent dus het verschil tussen x-in-C_1 en x-in-C_2. Ook in de substitutie die tijdens de resolutie ontstaat wordt dit onderscheid gemaakt. Men kan dus componenten hebben van de vorm x-in-C_1 $\leftarrow$ skelet-in-C_2. Om de waarde te kennen van de term die x-in-C_1 vervangt moet men gaan kijken naar de waarde die skelet-in-C_2 heeft in de omgeving van C_2. De omgeving van de nieuwe uitdrukking bestaat uit

- de omgeving van uitdrukking C_1 (voor de skeletten afkomstig van C_1)
- de omgeving van uitdrukking C_2 (voor de skeletten afkomstig van C_2)
- de nieuwe substitutie

Een vrij eenvoudige manipulatie van indices laat toe om het gewenste onderscheid te maken tussen variabelen afkomstig van verschillende uitdrukkingen. De details hiervan zijn voor ons niet zo belangrijk en kan men vinden in [Boyer & Moore 72]. Deze methode maakt de voorstelling van afgeleide uitdrukkingen zeer compact : informatie betreffende de twee samenstellende uitdrukkingen (constante grootte) en de substitutie (grootte evenredig met het aantal componenten).

De PROLOG-ontwerpers gebruikten eveneens de methode van de gedeelde strukturen. Dank zij beperkingen specifiek aan het uitvoeringsmechanisme van PROLOG waren belangrijke vereenvoudigingen mogelijk. We zullen deze vereenvoudigde versie wat grondiger bekijken.

GEDEELDE STRUKTUREN

Daar er geen resolutie is tussen probleemstellingen onderling hoeven we niet bezorgd te zijn voor naamconflicten tussen de diverse probleemstellingen. Tevens wordt elke probleemstelling slechts eenmaal gebruikt :

dit sluit uit dat een variabele z, voorkomend in een bepaalde probleemstelling, eerst aan een bepaalde term gebonden wordt en later nog eens aan een andere term. Dit alles laat ons toe om met een variabele uit een probleemstelling een welbepaalde lokatie te associëren. Deze lokatie bevat een verwijzing naar de waarde van die variabele en duiden we aan met een naam. Wordt naar een nieuwe probleemstelling overgegaan, dan kan deze variabele haar lokatie (naam) behouden. Het overgaan op een nieuwe probleemstelling vereist een proceduredefinitie. De namen van de variabelen uit deze definitie moeten verschillend zijn van de reeds bestaande namen in de probleemstelling. Dit gebeurt door met elke variabele uit de definitie een nieuwe lokatie (naam) te associëren.

Voorbeeld

Een wijzer naar een skelet sk stellen we voor door →sk. De oorspronkelijke probleemstelling Perm(A.B.Nil,z) Schrijf(z) kunnen we dan voorstellen als :

→Perm(A.B.Nil,z),E_o E_o : [z:-

→Schrijf(z),E_o

De omgeving E_o bestaat uit de niet gebonden of vrije variabele z (naam : z-in-E_o). De probleemstelling is gegeven door het skelet Perm in omgeving E_o en het skelet Schrijf in omgeving E_o.

Wordt de oproep Perm uitgevoerd met de proceduredefinitie Perm(x.y,u.v) ← Delete(u,x.y,w) Perm(w,v) dan associëren we met deze definitie de omgeving E_1 met variabelen x,y,u,v en w. Unifikatie geeft aanleiding tot de substitutie :

- x-in-E_1 ← (→A,E_o) : de variabele x uit omgeving E_1 wordt gebonden aan de waarde die skelet A heeft in omgeving E_o (omdat dit skelet geen variabelen bevat is de omgeving hier niet relevant)
- y-in-E_1 ← (→(B.Nil),E_o)
- z-in-E_o ← (→(u.v),E_1)

De resulterende probleemstelling is :

→Delete(u,x.y,w),E_1

→Perm(w,v),E_1

→Schrijf(z),E_o

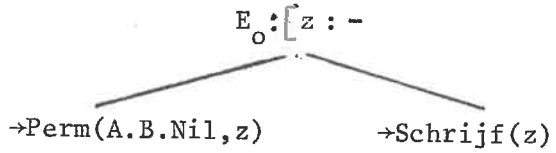
met E_o : [z : → u.v, E_1

en E_1 : [x : → A , E_o
y : → B.Nil, E_o
u : -
v : -
w : -

De nieuwe probleemstelling is dus :

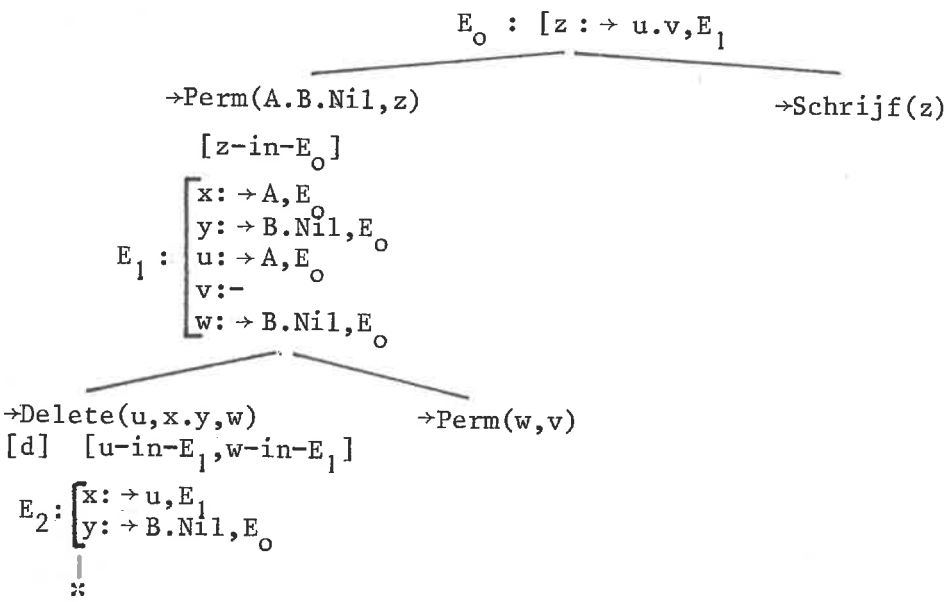
← Delete(u,A.B.Nil,w) Perm(w,v) Schrijf(u.v)

We kunnen dit mechanisme gemakkelijk illustreren met de vertrouwde EN-boom. De oorspronkelijke EN-boom is :



In plaats van te schrijven →Perm(...),E_o en →Schrijf(...),E_o plaatsen we E_o in hun gemeenschappelijk vaderknooppunt.

Na uitvoering van de oproepen Perm en Delete wordt deze EN-boom :



Bemerk dat nog een tweede procedure (d) beschikbaar is om de oproep Delete uit te voeren. Bemerk eveneens dat de nieuwe EN-boom niet alleen een uitbreiding is van de vorige, maar dat er ook wijzigingen zijn in de vorige omgevingen. Bepaalde variabelen die vrij waren (vb z-in-E_o)

zijn nu gebonden. Om naar de vorige EN-boom terug te keren moeten deze variabelen terug vrij gemaakt worden. Daarom worden ze genoteerd in het knooppunt van de oproep die ze bindt (de lijsten [z-in-E_0] en [u-in-E_1, w-in-E_1]).
We herinneren er nogmaals aan dat →Perm(A.B.Nil,z), →A, →u,... wijzers zijn naar skeletten. De enige in het systeem aanwezige skeletten zijn de skeletten (en deelskeletten) van de gegeven proceduredefinities en van de oorspronkelijke probleemstelling .

De voorstelling van procedure-oproepen door paren skelet-omgeving is vergelijkbaar met wat in een Algolachtige taal gebeurt. Het skelet is de proceduretekst die voor alle oproepen dezelfde is, de omgeving bevat de voor elke oproep specifieke informatie over de variabelen.

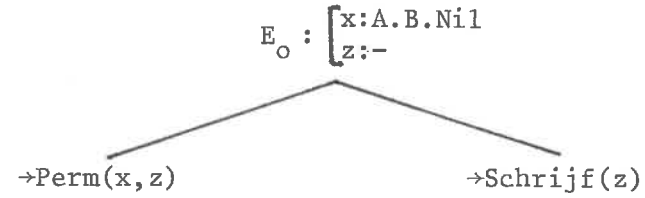
Kenmerkend voor dit implementatieschema is de voorstelling van de bindingen der variabelen door middel van gedeelde strukturen t.t.z. een paar skelet-omgeving.

We kunnen besluiten dat de voor PROLOG specifieke beperkingen niet alleen toelaten om uitdrukkingen als procedures te interpreteren maar tevens om gebruik te maken van de klassieke implementatieschema's (denk aan Algol) waarbij de verschillende instanties van een procedure enerzijds een gemeenschappelijke proceduretekst ("zuivere code") en anderzijds een eigen omgeving hebben. Ook vereenvoudigt de gedeelde strukturentechniek zich in sterke mate doordat elke variabele een welbepaalde lokatie heeft en men dus onmiddellijk toegang krijgt tot de (eventuele) waarde van die variabele.

Een implementatie van een vertolker is niets anders dan een algoritme dat op een voordelige wijze de EN-boom beschrijft, alsook de verschillende uit te voeren bewerkingen erop (selektie van de volgende uit te voeren oproep, uitvoeren van een oproep, terugkeren naar een vorige toestand). Vooraleer een dergelijk algoritme te geven willen we eerst een alternatief implementatieschema bestuderen.

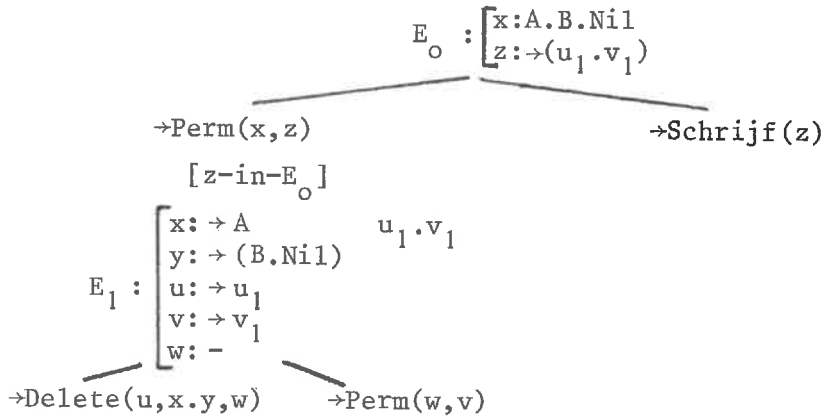
C. EEN ALTERNATIEF IMPLEMENTATIESCHEMA

Een alternatief implementatieschema werd door de auteur ontwikkeld [Bruynooghe 76]. Het enige verschil met het vorige schema betreft de voorstelling van de bindingen der variabelen. In plaats van de binding van een variabele voor te stellen door een paar skelet (zuiver code)-omgeving wordt ze nu voorgesteld door een struktuur die de waarde van de binding (de expressie die bekomen wordt door het skelet in de omgeving te evalueren) rechtstreeks vertegenwoordigt. Een dergelijke struktuur noemen we verder een kopie. De opeenvolgende probleemstellingen zelf worden dus nog steeds voorgesteld door verwijzingen naar de skeletten van de oproepen en hun bijhorende omgevingen. We hernemen hetzelfde voorbeeld en, om de voorstelling eenvoudig te houden, initialiseren we het nu als :



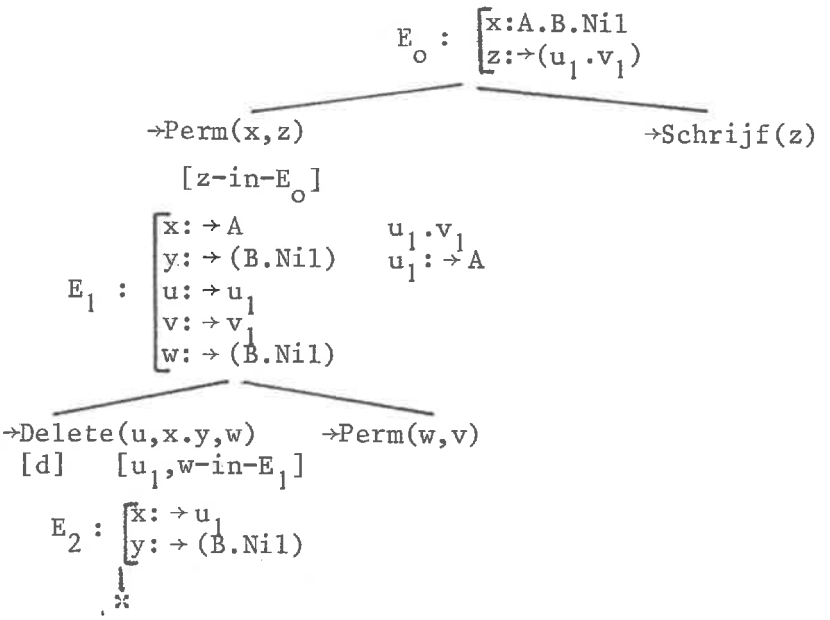
De skeletten Perm en Schrijf zijn geassocieerd met de omgeving E_o. De variabele z-in-E_o is vrij terwijl x-in-E_o gebonden is aan de struktuur A.B.Nil (van het "type" kopie). Om de oproep Perm(x,z) uit te voeren initialiseren we een omgeving E_1 met de variabelen x,y,u,v en w. Unifikatie heeft nu plaats tussen Perm(x,z) in omgeving E_o en Perm(x.y, u.v) in omgeving E_1. Unifikatie tussen de eerste argumenten heeft tot gevolg dat x-in-E_1 gebonden wordt aan de component A uit de kopie A.B.Nil en y-in-E_1 aan de component B.Nil uit dezelfde kopie. (Omdat het hier gaat om een verwijzing naar een reeds bestaande struktuur schrijven we x-in-E_1 ← (→A).) Unifikatie tussen de tweede argumenten heeft tot gevolg dat z-in-E_o moet gebonden worden aan de waarde die het skelet u.v heeft in omgeving E_1. We moeten dus het skelet u.v kopiëren. In deze nieuwe struktuur duiden we de kopieën van de variabelen u en v aan door de namen u_1 en v_1. Deze "nieuwe" variabelen u_1 en v_1 moeten verwijzen naar (de waarden van) u-in-E_1 en v-in-E_1. Daar deze laatsten in dit geval vrij zijn kunnen we even goed u-in-E_1 laten verwijzen naar u_1 en v-in-E_1 naar v_1. Later zal blijken dat deze mogelijkheid te verkiezen is.

De EN-boom wordt nu :



De kopie $u_1.v_1$ behoort bij de beschrijving van het nieuwe knooppunt Perm(x,z). De variabele z-in-E_o verwijst naar deze kopie. In het knooppunt moeten we, zoals vroeger, noteren dat de globale variabele z-in-E_o gebonden werd.

Het uitvoeren van de oproep Delete met de procedure Delete(x,x.y,y) ← resulteert in de creatie van een omgeving E_2 met variabelen x en y, en in de bindingen x-in-E_2 ← ($\rightarrow u_1$), u_1 ← ($\rightarrow$A) (de lokatie geassocieerd met u_1 die als "vrij" geïnitialiseerd was verwijst nu naar A), y-in-E_2 ← →(B.Nil) en w ← →(B.Nil). De EN-boom wordt dus :

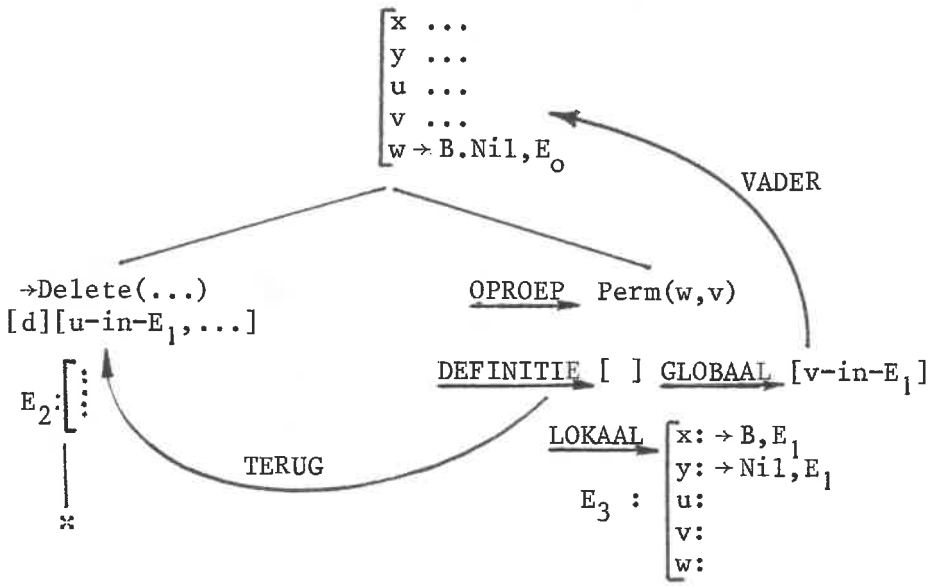


Op dit niveau van beschrijving is er slechts weinig verschil tussen beide methodes. Waar de methode met de gedeelde strukturen twee velden per variabele nodig heeft (skelet en omgeving) heeft de kopieermethode slechts één veld nodig. Echter, het kopieëren van de skeletten vraagt bijkomende geheugenruimte en doet wellicht het voordeel te niet. (Merk op dat het onnodig is om constanten zoals A,B,Nil te kopieëren). Eens we de implementatie van dichterbij bestuderen zal er een belangrijk verschil inzake geheugengebruik naar voor komen. Dit verschil ligt trouwens aan de basis van de ontwikkeling van de alternatieve methode.

D. EEN EENVOUDIG EN ALGEMEEN ALGORITME

De uitvoering van een programma werd geïllustreerd met behulp van een EN-boom. Nu moet die uitvoering beschreven worden door middel van een algoritme. Een eerste stap hiertoe is het kiezen van een geschikte gegevensstruktuur om de EN-boom voor te stellen. In de EN-boom onderscheiden we drie soorten knooppunten : de uitgevoerde oproepen, de geselekteerde (huidige) oproep en de nog uit te voeren oproepen. De enige informatie in een knooppunt met een nog uit te voeren oproep is een wijzer naar zijn skelet. Nu zijn de oproepen in de definities en in de oorspronkelijke probleemstelling geordend zodat het skelet van een nog uit te voeren oproep kan teruggevonden worden via het skelet van een uitgevoerde oproep of via het skelet van de geselekteerde oproep. Knooppunten met nog uit te voeren oproepen hoeven dus niet voorgesteld te worden.

De knooppunten met de uitgevoerde oproepen kunnen op een stapel geplaatst worden ("knooppuntenstapel"). Een element op deze stapel bevat, naast de informatie die in een knooppunt van de EN-boom aanwezig is, ook informatie om zich in deze EN-boom te oriënteren.



Zoals in bovenstaande figuur geïllustreerd is, bevat een stapelelement de volgende informatie :

- TERUG : oriëntatie-informatie : een wijzer naar het vorige stapelelement.
- OPROEP : knooppuntinformatie : een wijzer naar het skelet van de uitgevoerde oproep.
- VADER : oriëntatie-informatie : een wijzer naar het stapelelement dat de vader van dit knooppunt beschrijft.
- DEFINITIE : knooppuntinformatie : een wijzer naar de volgende definitie die voor deze oproep kan gebruikt worden. (Daar de definities geordend zijn is hiermee de volledige lijst bepaald)
- LOKAAL : knooppuntinformatie : informatie over de bindingen van de variabelen van de gebruikte definitie (lokale variabelen).
- GLOBAAL : knooppuntinformatie : lijst van de variabelen, behorend tot andere knooppunten ("globale" variabelen) die door het uitvoeren van deze oproep gebonden werden.

De toestand kan gekarakteriseerd worden door

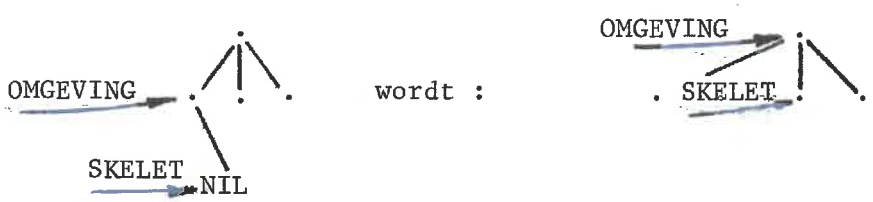
- SKELET : een wijzer naar het skelet van de uit te voeren oproep.
- OMGEVING : een wijzer naar het stapelelement dat de bindingen bevat van de variabelen in de uit te voeren oproep (de vader van het nieuwe knooppunt).
- PROCEDURE : een wijzer naar de te gebruiken proceduredefinitie.

We zullen nu een algoritme op zeer hoog niveau beschrijven dat een oproep uitvoert en de nieuwe toestand bepaalt. Dit algoritme spreekt over de opvolger van een oproep en de opvolger van een proceduredefinitie. Indien een dergelijke opvolger niet bestaat (opvolger van de laatste oproep in een definitie of van de laatste procedure van een rij procedures met dezelfde naam) is het resultaat NIL.

1. Plaats een nieuw element op de stapel met :
   - TERUG := het oude topelement van de stapel
   - OPROEP := SKELET
   - VADER := OMGEVING
   - DEFINITIE := de opvolger van PROCEDURE
   - LOKAAL := elke variabele in de gebruikte definitie PROCEDURE krijgt een lokatie en wordt als "vrij " geïnitialiseerd.
   - GLOBAAL := een lege lijst

2.-Unifikatie tussen SKELET met de bindingen gegeven in OMGEVING en de hoofding van PROCEDURE met de bindingen gegeven in het nieuwe topelement van de stapel.
  -Eventueel worden, tijdens de unifikatie, bepaalde variabelen in LOKAAL van het topelement gebonden alsook variabelen in LOKAAL van andere stapelelementen; deze laatsten worden genoteerd in de lijst GLOBAAL van het topelement.
  -SKELET := de eerste oproep in het lichaam van PROCEDURE (NIL indien ledig lichaam).
  -OMGEVING := het topelement van de stapel.

3. Indien de unifikatie slaagt,
   dan verder gaan :
   zolang SKELET=NIL :

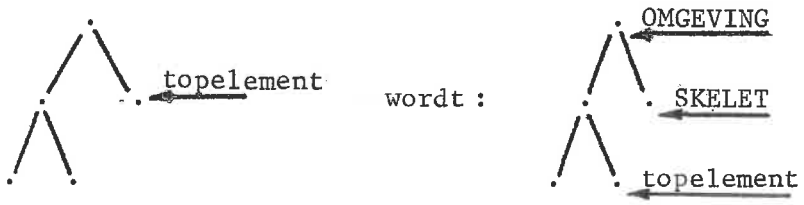


verlaat de huidige procedure en ga verder in de procedure die de oproep bevat :
SKELET := opvolger van OPROEP uit OMGEVING;
OMGEVING := VADER uit OMGEVING.

PROCEDURE := eerste proceduredefinitie van SKELET.

anders terugkeren :

herhaal :



PROCEDURE := DEFINITIE uit het topelement
Herinitialiseer als vrij alle variabelen op de lijst GLOBAAL
uit het topelement
SKELET := OPROEP uit het topelement
OMGEVING := VADER uit het topelement
Verwijder het topelement

totdat PROCEDURE ≠ NIL.

Nota's

1. Om de beschrijving eenvoudig te houden hebben we geen rekening gehouden met het geval waarbij een oplossing gevonden is en het geval waarbij geen enkele alternatieve oplossingsmogelijkheid meer bestaat.
2. Onze hoofdbekommernis is de eenvoud, niet de efficiëntie.
3. Dit algoritme is voldoende algemeen om beide implementatiemethodes te beschrijven. (Bij de kopieermethode maakt het unifikatie-algoritme eventueel kopies van skeletten, deze worden aan het topelement van de stapel toegevoegd).

Dit algoritme kan beschouwd worden als een sterk vereenvoudigde versie van de oorspronkelijke PROLOG-vertolker [Battani & Meloni 73]. Het belangrijkste verschil betreft de wijzer TERUG die, in hun voorstelling, slechts impliciet aanwezig is.

E. EEN BELANGRIJKE TEKORTKOMING : BUITENSPORIG GEHEUGENGEBRUIK

Vergelijken we het zopas beschreven algoritme met een algoritme voor een Algolachtige taal. Bij dit laatste wordt, bij het binnenkomen van een procedure, eveneens een element op de stapel toegevoegd, echter, bij het verlaten van de procedure wordt het terug verwijderd.

In ons algoritme wordt het element niet verwijderd. De reden ligt bij het niet-determinisme van de taal : we moeten in staat zijn om de vorige toestand te herstellen en de oproep opnieuw uit te voeren met een andere proceduredefinitie. Dit heeft tot gevolg dat de stapel zeer groot kan worden. Nochtans, in vele gevallen is geen andere proceduredefinitie beschikbaar en is het dus niet nodig om alle oude toestanden te kunnen herstellen. Om het geheugengebruik te beperken ware het wenselijk dat we in dergelijke gevallen eveneens elementen van de stapel kunnen verwijderen. Kijken we even , met behulp van een eenvoudig voorbeeld, hoe de EN-boom van de gedeelde strukturenmethode er in dergelijke gevallen uitziet.

Voorbeeld

(a) Append(Nil,x,x) ←

(b) Append(x.u,v,x.w) ← Append(u,v,w)

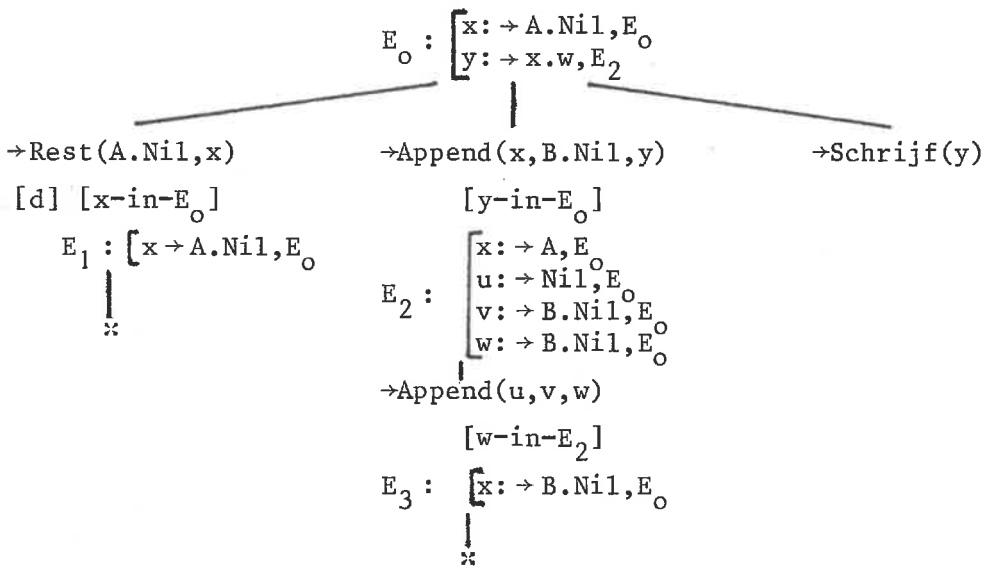
(c) Rest(x,x) ←

(d) Rest(x.y,z) ← Rest(y,z)

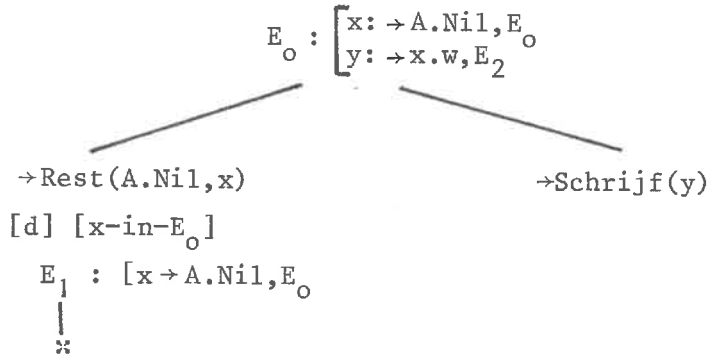
← Rest(A.Nil,x) Append(x,B.Nil,y) Schrijf(y)

(Een element (ℓ1,ℓ2) behoort tot de relatie Rest indien de lijst ℓ2 de staart is van de lijst ℓ1).

Na uitvoering van Rest en Append krijgen we :



De geselekteerde oproep is nu Schrijf(y). We bemerken dat de oproep Rest ook kan uitgevoerd worden met procedure (d) terwijl voor beide oproepen van Append geen andere procedures beschikbaar zijn (Een eenvoudige test leert dat procedure (b) onbruikbaar is voor Append(u,v,w)). Daar de berekening vanaf deze Append-oproepen niet kan herbeginnen (geen vertakkingspunten in de OF-boom) en de Append-procedures verlaten zijn is het wenselijk dat deze knooppunten kunnen verdwijnen :



Dat het skelet van Append verdwenen is, is geen probleem; wanneer de berekening vanaf Rest herbegint en de procedure Rest verlaten wordt, zal de opvolger van Rest geselekteerd worden en die opvolger is Append(x,B.Nil,y). Dit is dus in orde.
Wanneer het systeem vanaf Rest herbegint moet niet alleen x in E_o terug als vrije variabele geïnitialiseerd worden, maar ook y in E_o; deze laatste informatie is verloren gegaan. Ook hieraan kan verholpen worden : vooraleer knooppunten weg te laten wordt deze informatie opgepikt en in het knooppunt van Rest toegevoegd. (een meer elegante oplossing is mogelijk).

Er is echter nog een zwaarder probleem : de variabele y-in-E_o verwijst naar E_2 en E_2 is verdwenen : er is een "zwerfwijzer" ("dangling pointer") ontstaan. Hieraan verhelpen is veel moeilijker en vraagt een fundamentele wijziging van het implementatieschema. Inderdaad, alle zwerfwijzers opsporen en de bijhorende bindingen E_i bewaren (en wellicht verplaatsen) vereist het doorlopen van de volledige bestaande EN-boom en dit is onaanvaardbaar inefficiënt.

De problemen waren blijkbaar van dien aard, dat men in de oorspronkelijke PROLOG-vertolker van elke geheugenbesparende techniek heeft afgezien. De stapel groeit totdat een oplossing gevonden is of tot het pad

in de OF-boom doodloopt. Slechts dan keert het systeem op zijn stappen terug en worden elementen van de stapel verwijderd.

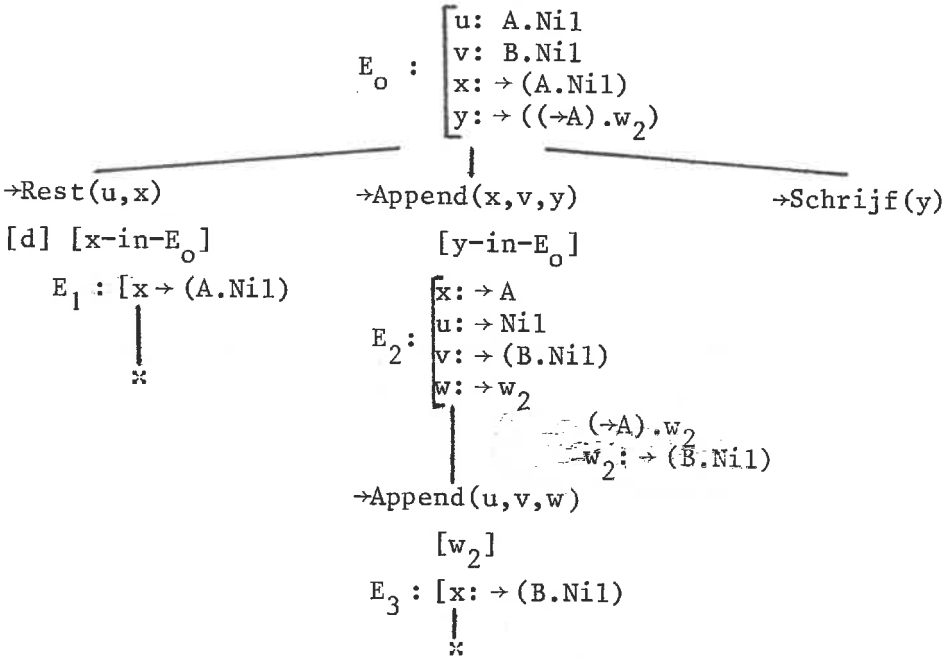
Bekijkt men het resultaat als een stellingbewijzer, dan kan men wellicht tevreden zijn over het geheugenbeheer. Beschouwd als de implementatie van een programmeertaal moet men toch eerder van een wanbeheer spreken. Het is heel mooi om een taal te ontwikkelen waarbij de gebruiker zich geen zorgen hoeft te maken over het geheugenbeheer. (In een conventionele taal staat de assignatie en het geheugenbeheer centraal zie [Backus 78]). Wanneer de vertolker aan wanbeheer doet is het echter weinig waarschijnlijk dat men veel gebruikers van conventionele talen zal overtuigen.

Onafhankelijk van elkaar hebben David Warren en de auteur elk een methode ontwikkeld om het geheugenbeheer te verbeteren. Warren zijn aanpak leidde tot een wijziging in de gedeelde strukturenmethode, terwijl ons werk resulteerde in de alternatieve kopieermethode.

F. DE KOPIEERMETHODE

F.1. PRINCIPE VAN DE GEHEUGENBESPARING

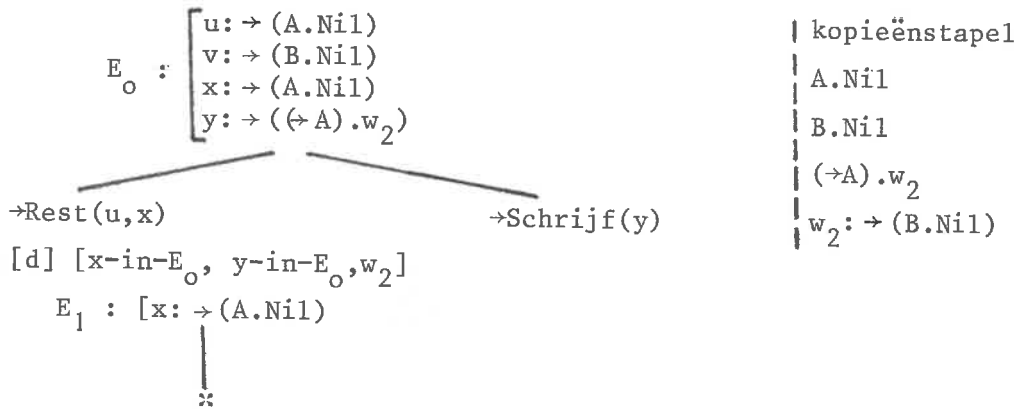
We nemen hetzelfde voorbeeld als daarnet en kijken hoe de EN-boom er nu uitziet :



De knooppunten die de twee Append-oproepen beschrijven, kunnen verdwijnen indien we

- de lijsten der te herinitialiseren variabelen aanpassen.
- de kopieën der skeletten op een afzonderlijke stapel plaatsen.

We krijgen dan :



Wanneer de kopieënstapel ongewijzigd blijft kunnen we de topelementen van de knooppuntenstapel weglaten zonder dat er zwerfwijzers ontstaan. Inderdaad :

- variabelen uit de elementen E_i verwijzen ofwel naar de kopieënstapel (zoals hier) ofwel naar variabelen uit een element E_j met $j < i$ (knooppunt j dichter bij de bodem van de stapel; men zorgt dat hieraan voldaan is wanneer men twee variabelen aan elkaar bindt).
- variabelen uit de kopieënstapel (zoals w_2) verwijzen eveneens naar de kopieënstapel, nooit naar de knooppuntenstapel (men zorgt dat hieraan voldaan is wanneer men twee variabelen aan elkaar bindt).

F.2. GEHEUGENBESPARENDE TECHNIEKEN

We kunnen een drietal basissituaties onderscheiden waarin geheugenbesparing mogelijk is.

1. Staartrecursie ("tail end recursion")

In een Algolachtige taal kan men een procedure die met haar laatste bevel zichzelf oproept gemakkelijk omvormen tot een procedure waarin de recursie vervangen is door een iteratie. Onze geheugenbesparende vertolker kan, zonder dat de programmatekst gewijzigd wordt, een dergelijk gedrag simuleren. In plaats van telkens een nieuw element op de stapel toe

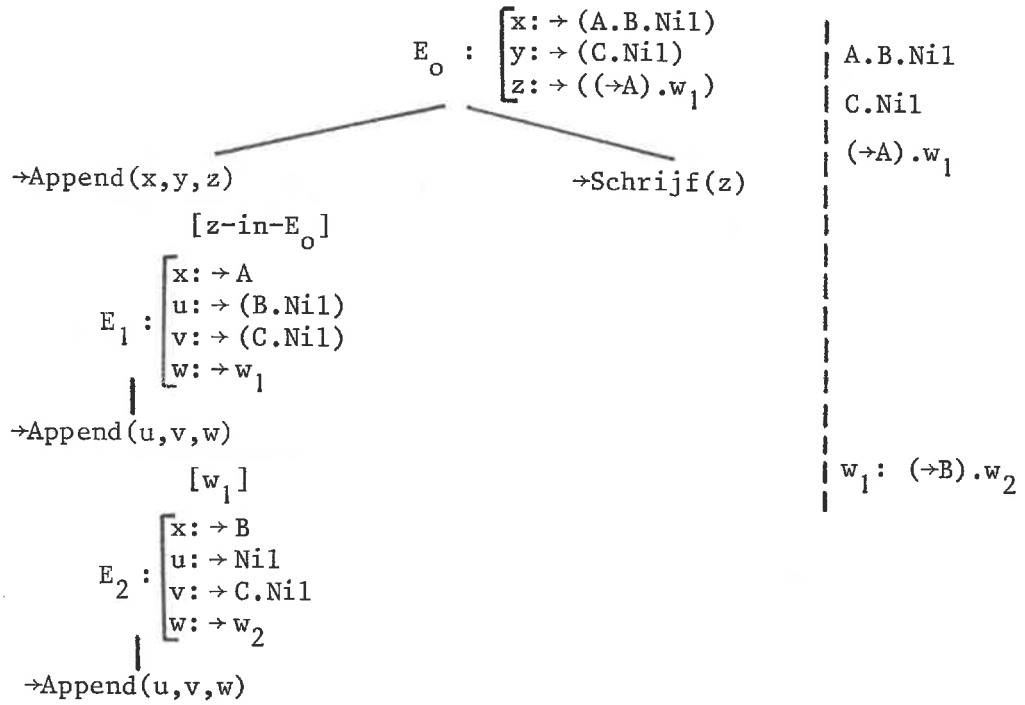
te voegen kan de vertolker hetzelfde element opnieuw gebruiken.

Voorbeeld

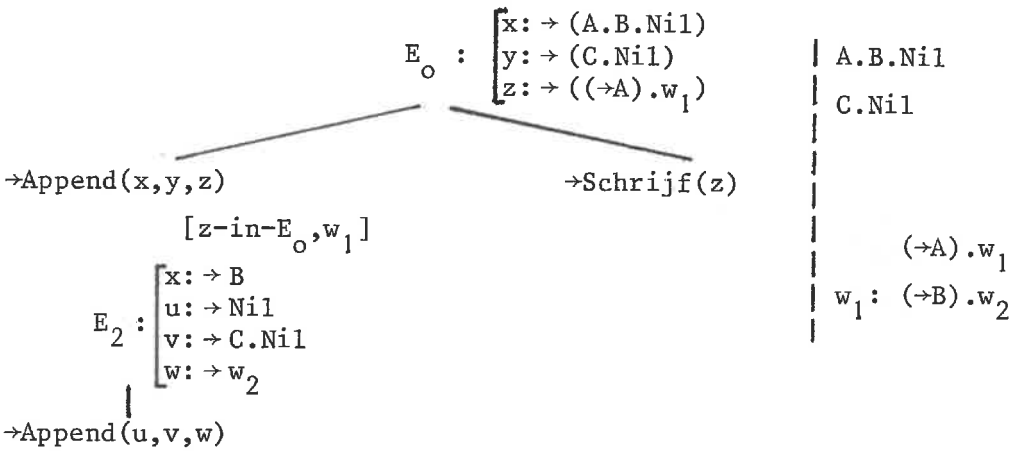
Append(Nil,x,x) ←

Append(x.u,v,x.w) ← Append(u,v,w)

← Append(A.B.Nil,C.Nil,z) Schrijf(z)



De geselekteerde groep is nu Append(u,v,w) met de bindingen gegeven in E_2. Deze EN-boom kan vereenvoudigd worden tot :



Voor beide EN-bomen verloopt de verdere uitvoering (exploratie van de OF-boom) van het programma identiek.

Algemeen kunnen we bovenstaande situatie als volgt omschrijven :

- Een oproep voor een procedure P (Append(x,y,z)); de proceduredefinitie heeft de vorm $P \leftarrow P_1, P_2, \ldots, P_n$. Het knooppunt dat de oproep P beschrijft bevat een element $E_p(E_1)$ met de bindingen van de variabelen in de procedure P.
- Door herhaald toepassen van de verschillende geheugenbesparende technieken zijn alle knooppunten van de deelbomen die de uitvoering van de oproepen $P_1, \ldots, P_{n-1}$ beschrijven, uit de EN-boom (knooppuntenstapel) verdwenen. (Dit komt erop neer dat geen enkele van de oproepen die door deze knooppunten beschreven zijn, over alternatieve proceduredefinities beschikt. Het veld PROCEDURE was overal NIL).
- Slechts één procedure is bruikbaar voor de oproep P_n (Append(u,v,w)). Het knooppunt dat de oproep P_n beschrijft bevat een element $E_{pn}(E_2)$ dat de bindingen van de variabelen van de gebruikte procedure bevat. Deze voorwaarden zijn vrij eenvoudig te controleren :
  - P_n is de laatste oproep van de proceduredefinitie en beschikt over geen alternatieve procedures.
  - De knooppunten P_n en P (de vader van P_n) zijn de twee topelementen van de stapel.

  De inkrimping van de stapel is ook eenvoudig : vervang E_p door E_{pn} en verwijder het topelement (het knooppunt P_n).

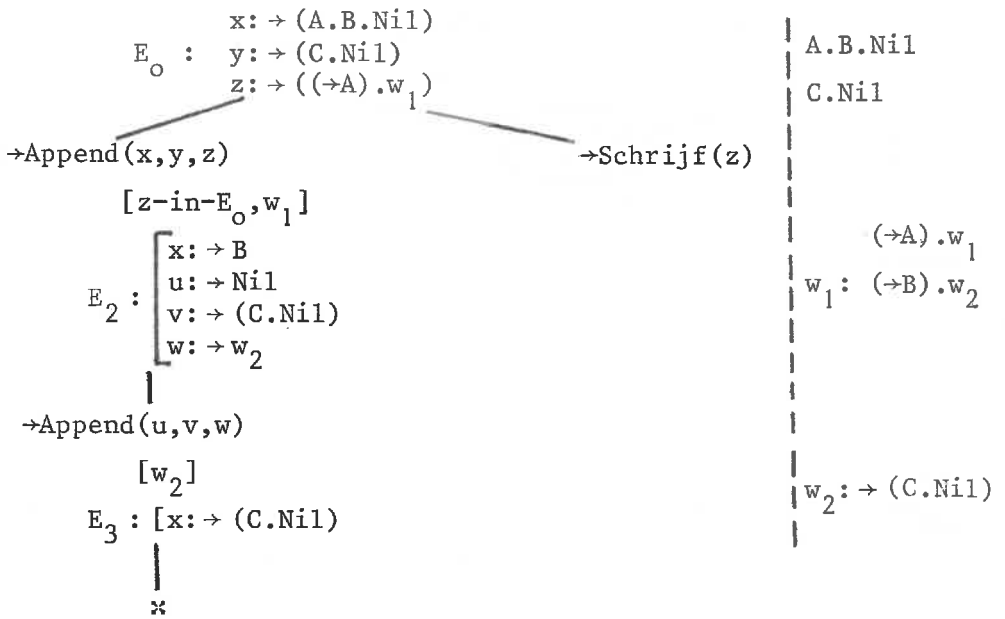
Bemerk dat staartrecursie veel vaker voorkomt in PROLOG dan in een conventionele taal. In een conventionele taal zal de Append-procedure het resultaat x.w slechts vormen na het uitvoeren van de recursieve oproep en zal er dus geen staartrecursie zijn.

De zopas beschreven inkrimping moet met de nodige omzichtigheid uitgevoerd worden. Eventueel bevat E_{pn} wijzers naar variabelen uit E_p. Het weglaten van E_p veroorzaakt dan zwerfwijzers. Dit moet natuurlijk vermeden worden, hetzij door :

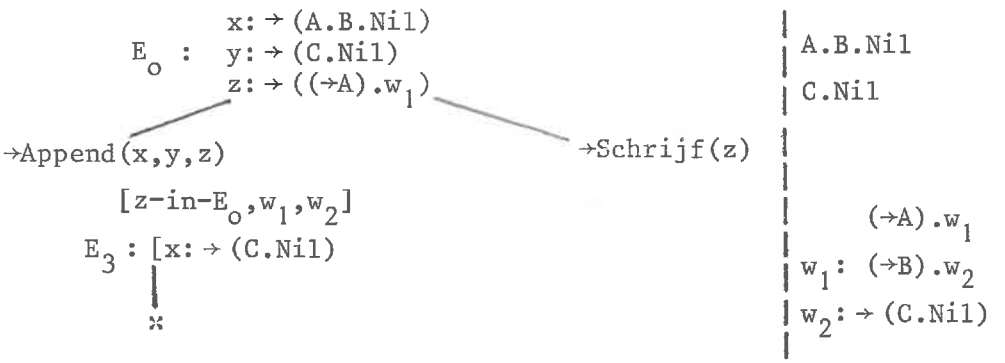
- de wijzers in E_{pn} op de gepaste manier aan te passen (opgepast, twee variabelen uit E_{pn} kunnen naar dezelfde vrije variabele uit E_p verwijzen, uiteindelijk moet de eerste vrij worden en moet de tweede naar de eerste wijzen).
- vooraleer de unifikatie begint, vast te stellen dat staartrecursie zal mogelijk zijn en dan reeds E_p en E_{pn} om te wisselen. De elementen uit E_{pn} staan dan onmiddellijk op hun definitieve plaats en zullen niet verwijzen naar elementen uit E_p.

2. Het beëindigen van een deterministische procedure-oproep

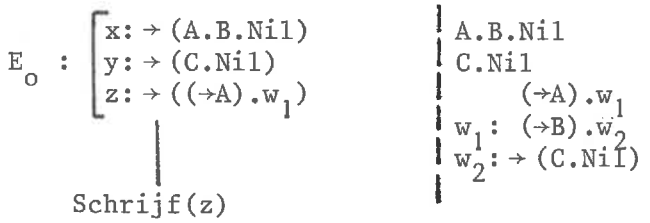
Een oproep is beëindigt wanneer alle oproepen in de deelboom met die oproep als wortel uitgevoerd zijn. We noemen de uitvoering deterministisch wanneer in diezelfde deelboom geen enkele oproep over alternatieve definities beschikt (het veld DEFINITIE is overal NIL). Werken we het laatste voorbeeld verder uit, dan zien we dat hieraan voldaan is :



We kunnen nogmaals staartrecursie toepassen :



Nu moet de eerste oproep in het lichaam van Append(Nil,x,x) uitgevoerd worden; dit is echter ledig. De oproep Append(A.B.Nil,C.Nil,z), is dus beëindigd en is deterministisch. Het knooppunt dat de oproep beschrijft kan verdwijnen.



Door het herhaald toepassen van staartrecursie is het voorbeeld typisch voor het beëindigen van een deterministische oproep. De situatie kan als volgt gekenmerkt worden :

- Slechts één procedure was beschikbaar om de oproep beschreven in het topelement van de stapel uit te voeren.
- Het lichaam van deze procedure moet nu uitgevoerd worden, het is echter ledig.

Nadat de volgende oproep geselekteerd is kan men het topelement van de knooppuntenstapel laten verdwijnen.

3. Het optimaliseren van de plaats voorzien voor de bindingen der variabelen

De elementen E_i bevatten de waarden van de variabelen die voorkomen in de skeletten der proceduredefinities. Het is de bedoeling dat men die waarden kan terugvinden wanneer men die variabelen in een skelet ontmoet. De eerste maal dat men een variabele ontmoet is ze vrij. Komt in een bepaalde proceduredefinitie een variabele slechts éénmaal voor, dan weet men dat die variabele bij elke ontmoeting vrij is. Indien het unifikatiealgoritme dergelijke variabelen kan herkennen, is het niet nodig dat men ze in het knooppuntelement plaatst.

Voorbeeld : de variabele y in Member(x,x.y)← en in Member(x,y.z)← Member(x,z). Een dergelijke variabele noemen we een loze variabele.

Een tweede optimalisatie betreft variabelen die alleen voorkomen in de hoofding van een definitie (o.m. alle variabelen in een definitie met een ledig lichaam). Eens de unifikatie tussen oproep en hoofding ten einde, kunnen we nooit meer een skelet ontmoeten dat die variabelen bevat. Ze kunnen dus uit het E-element verwijderd worden. Dit is eenvoudig want op dat ogenblik is het betreffende E-element in het topelement van de knooppuntenstapel.

Voorbeeld : x in Append(Nil,x,x)←
en in Append(x.u,v,x.w)← Append(u,v,w)

Dergelijke variabelen noemen we tijdelijke variabelen. Deze laatste optimalisaties zijn geïnspireerd door het werk van David Warren [Warren 77a]. Met de kopieermethode is de klasse der loze en tijdelijke variabelen echter ruimer dan met zijn aangepaste gedeelde strukturenmethode.

F.3. HET ONTDEKKEN VAN CYCLISCHE STRUKTUREN ("occurcheck")

Het binden van een variabele x aan een term die x bevat (f(x)) moet in principe verhinderd worden. Over de wenselijkheid van deze tijdrovende test hebben we gesproken in Hoofdstuk 1. Hier willen we kijken hoe het aantal testen kan verminderd worden. Unifikatie tussen een procedure-oproep en de hoofding van een proceduredefinitie kan slechts een cyclische struktuur veroorzaken indien een variabele minstens tweemaal in de hoofding voorkomt. Inderdaad, een cyclische struktuur kan slechts ontstaan door tussenkomst van een variabele in de hoofding (vb x) en vereist minstens twee stappen (de volgende is willekeurig) :

- de creatie van een term die de variabele x bevat. Voorbeeld : u←f(x) (ook u←x)
- het binden van die term aan de variabele x. Voorbeeld : x←g(u) (ook x←u maar dan niet samen met u←x)

De variabele x moet dus minstens tweemaal in de hoofding voorkomen en slechts bij de tweede ontmoeting kan een cyclische struktuur ontstaan. Wanneer het unifikatiealgoritme een binding x←g(u) (u←f(x)) creëert met x een variabele (f(x) een skelet) uit de hoofding dan is de "occurcheck" overbodig indien het de eerste ontmoeting is met die variabele. Alleen bij de tweede en volgende ontmoetingen moet een test uitgevoerd worden.

Met de kopieermethode kan het aantal testen nog verder beperkt worden. Bij deze methode krijgt een vrije variabele (x-in-E_i) die voorkomt in een struktuur een kopie (x_i) op de kopieënstapel, x-in-E_i is dan niet langer vrij, maar aan x_i gebonden. Dit laat ons toe om vrij eenvoudig een aantal situaties te herkennen waarin het ontstaan van cyclische strukturen onmogelijk is :

1. Een vrije variabele uit de knooppuntenstapel ($x\text{-in-}E_i$) moet gebonden worden aan een bestaande component van de kopieënstapel. Daar $x\text{-in-}E_i$ vrij is kan ze niet in de kopieënstapel voorkomen.
2. Een vrije variabele uit de knooppuntenstapel ($x\text{-in-}E_i$) moet gebonden worden aan de waarde van een skelet in omgeving E_j en in E_j zijn de in het skelet voorkomende variabelen niet aan $x\text{-in-}E_i$ gebonden.

Voorbeeld

Oproep Delete(u,v,w) in E_o : $\begin{cases} u: - \\ v: \rightarrow (A.B.Nil) \\ w: - \end{cases}$

Hoofding Delete(x,x.y,y)

in initiële omgeving E_1 : $\begin{cases} x: - \\ y: - \end{cases}$

Unifikatie van

- u met x resulteert in $x\text{-in-}E_1 \leftarrow \rightarrow u\text{-in-}E_o$.
  Geen test nodig want eerste voorkomen van x.
- v met x.y resulteert in $u\text{-in-}E_o \leftarrow \rightarrow A$.
  Tweede voorkomen van x en geen test nodig omwille van regel 1.
  en $y\text{-in-}E_1 \leftarrow \rightarrow (B.Nil)$.
  Geen test nodig want eerste voorkomen van y.
- w met y resulteert in $w\text{-in-}E_o \leftarrow \rightarrow (B.Nil)$.
  Tweede voorkomen van y en geen test nodig omwille van regel 1.

Natuurlijk worden niet alle overbodige testen uitgeschakeld.

Voorbeeld

oproep Append(u,v,w)

in E_o : $\begin{cases} u: \rightarrow Nil \\ v: \rightarrow (B.Nil) \\ w: \rightarrow w_1 \end{cases}$ $\left| \begin{array}{l} \quad B.Nil \\ w_o: (\rightarrow A).w_1 \end{array} \right.$

hoofding Append(Nil,x,x) in E_1: [x:-

unifikatie van

u met Nil resulteert in niets

v met x resulteert in $x\text{-in-}E_1 \leftarrow \rightarrow (B.Nil)$

w met x resulteert in $w_1 \leftarrow \rightarrow (B.Nil)$

w_1 is een variabele op de kopieënstapel en kan dus voorkomen in de term waaraan x gebonden is. Een test op cyclische strukturen is hier noodzakelijk.

F.4. ALGORITME VOOR DE KOPIEERMETHODE

We zullen gebruik maken van drie stapels :

- de knooppuntenstapel die de EN-boom beschrijft.
- de kopieënstapel die de kopieën der skeletten bevat.
- de herstartstapel die wijzers bevat naar de variabelen die opnieuw geïnitialiseerd moeten worden wanneer de vertolker terugkeert en start met de exploratie van een ander pad in de OF-boom.

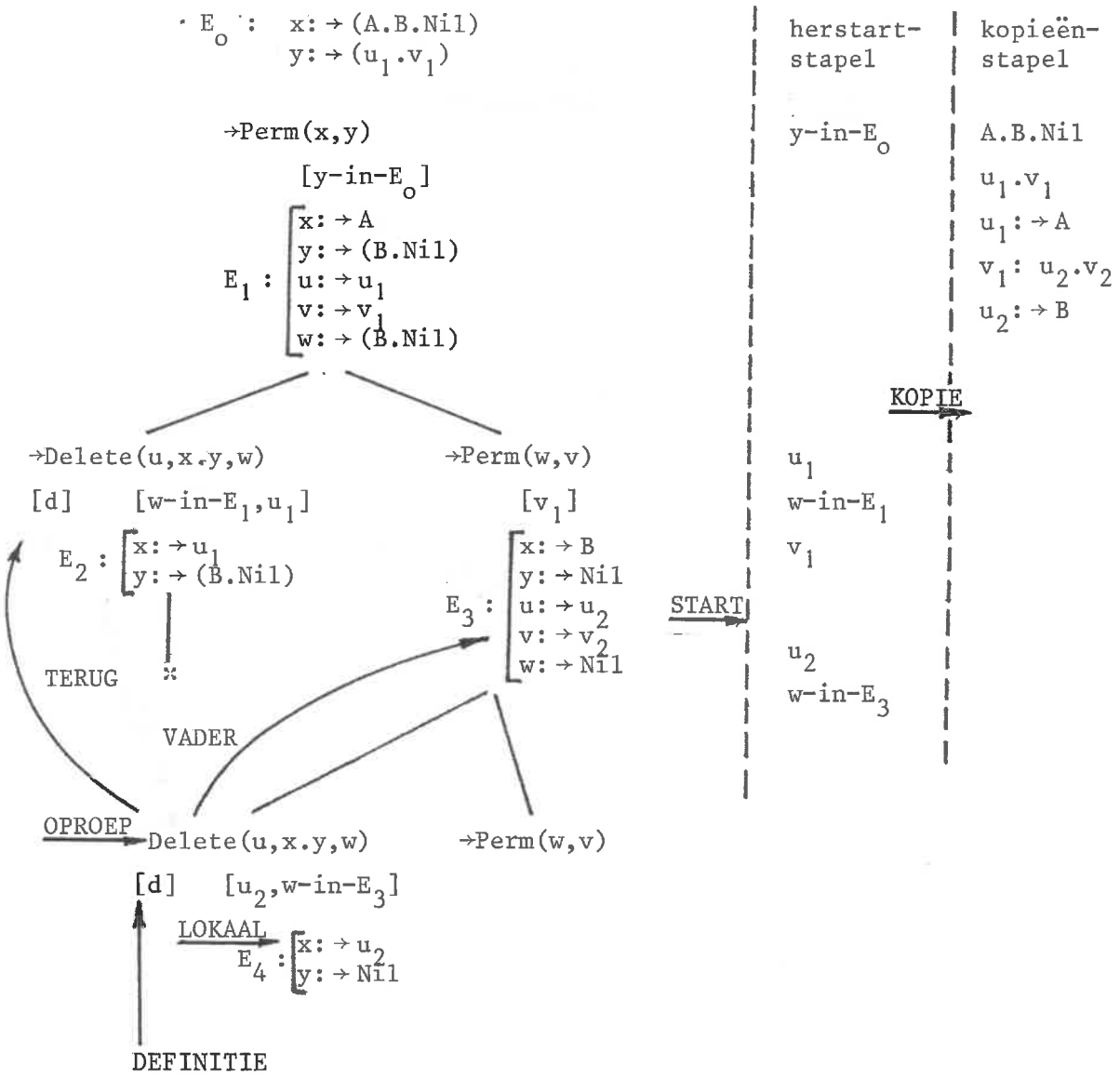
Een element uit de knooppuntenstapel bestaat uit de volgende componenten :

- OPROEP : een wijzer naar het skelet van de uitgevoerde oproep,
- VADER : een wijzer naar het knooppunt dat de vader van dit knooppunt beschrijft.
- DEFINITIE : een wijzer naar de volgende definitie die voor deze oproep kan gebruikt worden (NIL indien geen alternatieve definitie bestaat).
- LOKAAL : een zone die plaats bevat voor de bindingen van de variabelen in de gebruikte definitie.

Indien DEFINITIE ≠ NIL dan is deze oproep het begin van een nog niet onderzocht pad in de OF-boom en moet het knooppuntelement eveneens de volgende velden bevatten (dergelijk knooppunt noemen we "herstartpunt").

- TERUG : een wijzer naar het vorige herstartpunt in de knooppuntenstapel.
- START : een wijzer naar de herstartstapel, meer bepaald, naar de top die deze stapel had vóór het uitvoeren van OPROEP.
- KOPIE : een wijzer naar de top die de kopieënstapel had vóór het uitvoeren van OPROEP.

In onderstaande figuur zijn de zes componenten van de informatie in een knooppunt Delete(u,x.y,w) aangeduid.



De toestand waarin de uitvoering van een programma zich bevindt kan gekarakteriseerd worden door :

- SKELET : een wijzer naar het skelet van de uit te voeren oproep.
- OMGEVING : een wijzer naar het knooppuntelement dat de bindingen bevat van de variabelen in de uit te voeren oproep.
- PROCEDURE : een wijzer naar de te gebruiken proceduredefinitie.
- ALTERNATIEF : een wijzer naar het meest recente knooppunt dat een herstartpunt is.

Wegens de geheugenbesparende technieken zijn de eindknooppunten in de EN-boom

- ofwel onuitgevoerde oproepen (niet op de knooppuntenstapel)
- ofwel uitgevoerde oproepen die een mogelijk herstartpunt zijn (veld DEFINITIE ≠ NIL, ze behoren tot de lijst die begint met ALTERNATIEF).

Dit betekent dat we het voorlaatste element van de knooppuntenstapel als volgt kunnen vinden : (in het algoritme hebben we soms dat element nodig).

Indien ALTERNATIEF = topelement dan
  voorlaatste element := maximum van
    - VADER uit het topelement
    - TERUG uit het topelement

anders
  voorlaatste element := maximum van
    - ALTERNATIEF
    - VADER uit het topelement

Het algoritme dat een oproep uitvoert en de nieuwe toestand bepaalt kan nu beschreven worden als :

1. Plaats een nieuw element op de knooppuntenstapel met :
   - OPROEP := SKELET
   - VADER := OMGEVING
   - DEFINITIE := eerste "geschikte" (zie verder) opvolger van PROCEDURE
     Indien DEFINITIE ≠ NIL dan ook
     . TERUG := ALTERNATIEF
     . START := top van de herstartstapel
     . KOPIE := top van de kopieerstapel
   - LOKAAL : elke variabele uit PROCEDURE krijgt een plaats en wordt geïnitialiseerd als "vrije" variabele.
   - Indien DEFINITIE ≠ NIL wordt ALTERNATIEF aangepast:
     ALTERNATIEF := nieuw knooppunt.
2. Unifikatie tussen SKELET met de bindingen gegeven in OMGEVING en de hoofding van PROCEDURE met de bindingen gegeven in het nieuwe topelement van de knooppuntenstapel.

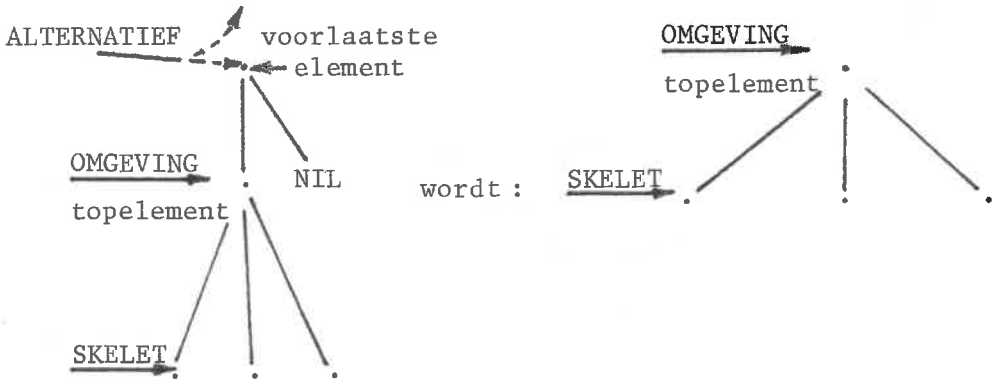
Tijdens de unifikatie worden variabelen uit de zones LOKAAL van de knooppunten gebonden. Wijzers naar deze variabelen worden op de herstartstapel geplaatst indien ze zich bevinden tussen de bodem van de stapel en ALTERNATIEF.
Eveneens worden kopieën van skeletten gemaakt, deze worden op de kopieënstapel geplaatst. Ook kunnen variabelen op deze kopieënstapel gebonden worden. Wijzers ernaar worden op de herstartstapel toegevoegd indien ze zich bevinden tussen de bodem van de kopieënstapel en START uit ALTERNATIEF.
Na unifikatie wordt :

SKELET := eerste oproep in het lichaam van PROCEDURE. (NIL indien geen).

OMGEVING := topelement van de knooppuntenstapel.

3. Indien de unifikatie slaagt,
dan verder gaan :
- tijdelijke variabelen verdwijnen uit LOKAAL van OMGEVING

- staartrecursie ?



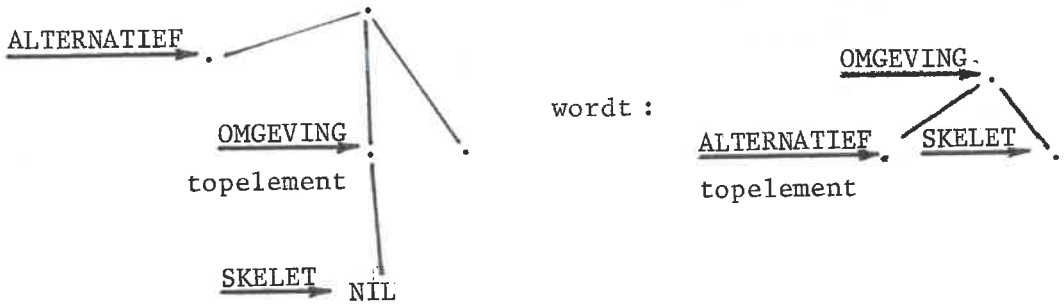
Indien opvolger van OPROEP uit OMGEVING = NIL en
ALTERNATIEF ⩽ VADER uit OMGEVING
dan staartrecursie

LOKAAL uit VADER van OMGEVING wordt vervangen door
LOKAAL uit OMGEVING

OMGEVING := VADER uit OMGEVING

Het topelement wordt verwijderd, OMGEVING is de nieuwe top.

- beëindigen van een deterministische procedure-oproep ?



Indien SKELET = NIL en DEFINITIE uit OMGEVING = NIL (of ALTERNATIEF ≠ OMGEVING)
dan beëindigen van een deterministische oproep :

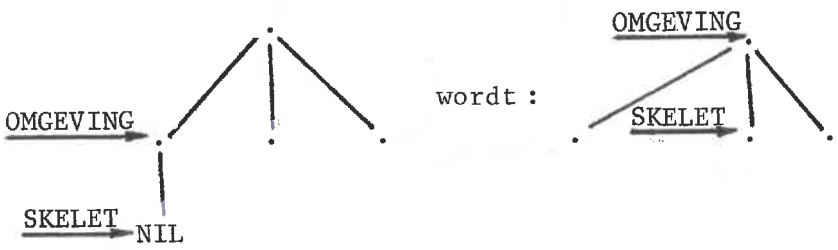
SKELET := opvolger van OPROEP uit OMGEVING

OMGEVING := VADER uit OMGEVING

Het topelement wordt verwijderd en de nieuwe top is het maximum van ALTERNATIEF en OMGEVING.

- bepalen van de nieuwe toestand

zolang SKELET = NIL

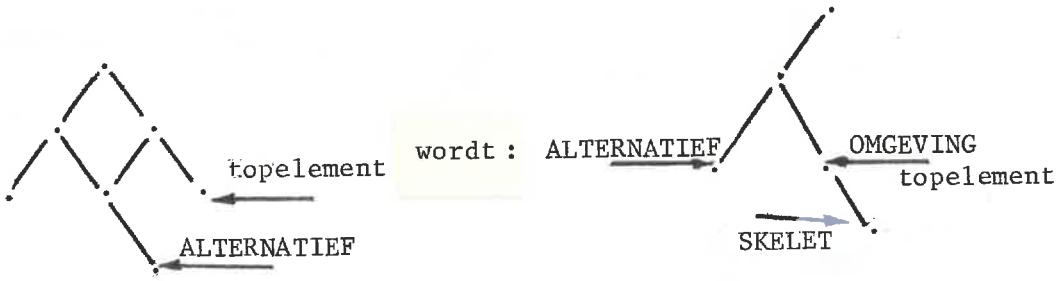


SKELET := opvolger van OPROEP uit OMGEVING

OMGEVING := VADER uit OMGEVING

PROCEDURE := eerste, voor SKELET "geschikte" proceduredefinitie.

anders terugkeren :



topelement := ALTERNATIEF (n knooppunten verdwijnen met n $\geq$ 0)
PROCEDURE := DEFINITIE uit topelement
SKELET := OPROEP uit topelement
OMGEVING := VADER uit topelement
ALTERNATIEF := TERUG uit topelement
Top van de kopieënstapel := KOPIE uit topelement
Alle variabelen die in de herstartstapel genoteerd zijn vanaf START uit topelement worden opnieuw als "vrij" geïnitialiseerd.
Top van de herstartstapel := START uit topelement
Het topelement wordt verwijderd en het nieuwe topelement is het maximum van ALTERNATIEF en OMGEVING.

Opmerking

We hebben hier gesproken over de eerste "geschikte" proceduredefinitie en de eerste "geschikte" opvolger. De verschillende geheugenbesparende technieken zijn gebaseerd op het herkennen dat de gebruikte definitie de "laatste" is die moet geprobeerd worden. Een triviale manier bestaat in het herkennen dat het de laatste is in de rij. Mits wat bijkomende berekeningen kan men heel wat beter doen (bijvoorbeeld herkennen dat Append(x,u,...) niet bruikbaar is voor een oproep Append(Nil,...)). Deze bijkomende inspanning kan tot belangrijke geheugenbesparingen leiden. (ook het gebruik van de "/"). Een ander uiterste bestaat erin een procedure slechts "geschikt" te vinden wanneer unifikatie met de oproep mogelijk is. Een mogelijk compromis bestaat erin dat men controleert of de struktuur van de verschillende argumenten vergelijkbaar is : vb. f(x) is vergelijkbaar met f(g(A)) maar niet met g(y); Nil met x maar niet met x.y.

F.5. "GARBAGE COLLECTION"

Het laten verdwijnen van bepaalde delen van de knooppuntenstapel kan sommige elementen van de kopieënstapel ontoegankelijk maken. Dit is niet erg : wanneer de vertolker naar een herstartpunt terugkeert wordt de top van de kopieënstapel toch gerecupereerd. Nochtans, wanneer de nood aan geheugen groot is kunnen die ontoegankelijke elementen gerecupeerd worden en kan de stapel compacter gemaakt worden. Dit gebeurt door een "garbage collector". Een lineair algoritme hiervoor, dat in feite geen bijkomend geheugen vraagt (2 bits per element), vindt men in [Morris 78].

G. WARREN ZIJN OPLOSSING : EEN GEWIJZIGDE GEDEELDE STRUKTURENMETHODE

Toen we aantoonden dat de kopieermethode toelaat belangrijke geheugenbesparingen te verwezenlijken, zegden we terloops dat de gedeelde strukturenmethode gelijkaardige besparingen kan toelaten door de komponenten E_i die de bindingen der variabelen bevatten uit de knooppunten te verwijderen en in een aparte "omgevingsstapel" te plaatsen. De zo te verwezenlijken besparingen zijn echter heel wat kleiner omdat de component die de bindingen beschrijft, relatief groot is en volledig blijft bestaan. Warren is erin geslaagd om dit principe te verfijnen. Wanneer een binding $x \leftarrow \rightarrow SK$, E_i verwezenlijkt wordt zijn niet alle variabelen uit E_i nodig om de waarde van SK te kennen, enkel deze die in SK voorkomen zijn vereist. In een procedure onderscheidt Warren daarom twee soorten variabelen.:

- globale variabelen : variabelen die minstens éénmaal in een complex skelet voorkomen, vb x in f(x).
- lokale variabelen : variabelen die in geen enkel skelet voorkomen.

Voorbeeld

Append(Nil,x,x)$\leftarrow$: x is lokaal

Append(x.u,v,x.w) $\leftarrow$ Append(u,v,w) : x,u en w zijn globaal, v is lokaal.

Nu wordt het element E_i dat de bindingen der variabelen beschrijft opgesplitst in een L_i voor de lokale variabelen en een G_i voor de globale variabelen. De componenten L_i blijven op de knooppuntenstapel terwijl de componenten G_i op een aparte "globale stapel" geplaatst worden (te vergelijken met de kopieënstapel). De codering der proceduredefinities laat het unifikatie-algoritme toe om lokale en globale variabelen van elkaar te onderscheiden. Het unifikatie-algoritme kan met vier soorten bindingen geconfronteerd worden :

- twee lokale variabelen : de variabele dichtst bij de top van de stapel wordt gebonden.
- een lokale ($x\text{-in-}L_i$) en een globale variabele ($y\text{-in-}G_j$) : de lokale variabele wordt gebonden : $x\text{-in-}L_i \leftarrow \rightarrow y, G_j$.
- twee globale variabelen : een van beide (willekeurig) wordt gebonden.
- een variabele (x) en een skelet SK. De bindingen van de variabelen uit SK kunnen gevonden worden in een element G_i van de globale stapel : $x \leftarrow \rightarrow SK, G_i$.

Op deze manier wordt voorkomen dat de geheugenbesparende technieken, toegepast op de knooppuntenstapel, zwerfwijzers veroorzaken. Het onderscheid tussen lokale en globale variabelen moet gemaakt worden op het ogenblik dat de proceduredefinities gecodeerd worden. Dit is een belangrijk nadeel dat Warren probeert te ondervangen door de gebruiker toe te laten om zogenoemde "modedeklaraties" op te geven : de gebruiker kan aan het systeem mededelen dat oproepen altijd een bepaald patroon zullen hebben.

Voorbeeld

Append(x.u,v,x.w) ← Append(u,v,w)

x,u en w zijn alle drie globaal. Wanneer het eerste argument van de oproep steeds van de vorm r.s is, dan kan u een lokale variabele worden, inderdaad, er zal nooit een verwijzing naar het skelet x.u ontstaan. (nooit "konstruktie" van het skelet x.u). De gebruiker kan dit opgeven door middel van een modedeklaratie.

De struktuur van een knooppuntelement is ongeveer dezelfde als met de kopieermethode, alleen is de wijzer KOPIE naar de kopieënstapel vervangen door een wijzer GLOBAAL voor de globale stapel. Dit veld GLOBAAL is in elk knooppunt noodzakelijk. Warren schenkt trouwens geen aandacht aan de mogelijkheid om de velden TERUG en START weg te laten wanneer het veld DEFINITIE gelijk is aan NIL. Ook de mogelijkheid om "staartrecursie" toe te passen ziet hij over het hoofd. Wel onderscheidt hij loze en tijdelijke variabelen. Deze optimalisatie moet in zijn methode beperkt blijven tot de lokale variabelen. Zoals we gezien hebben kan ze bij de kopieermethode uitgebreid worden tot alle variabelen.

Warren heeft, samen met Luis en Fernando Pereira een "vertaler" ontwikkeld in assembler voor de DEC 10. Een belangrijke innovatie hierbij is dat men, bij het uitvoeren van een oproep, geen beroep doet op een unifikatie-algoritme dat alle mogelijke situaties voorziet, maar dat de proceduredefinities vervangen zijn door reeksen bevelen die aangepast zijn aan de vorm van de diverse definities ("vertaling"). Deze innovatie, samen met het feit dat de eigenlijke vertolker in assembler geschreven is (de "vertaling" gebeurt door een PROLOG-programma) resulteert in een systeem dat in snelheid vergelijkbaar is met vertaalde LISP [Warren 77a], [Warren & Co 77].

H. EEN VERGELIJKING VAN DE GEHEUGENBESPARINGEN VERWEZENLIJKT DOOR DE KOPIEERMETHODE EN DE GEWIJZIGDE GEDEELDE STRUKTURENMETHODE

In beide methodes vereist de voorstelling van een knooppunt, naast de plaats voor de bindingen der variabelen, zes velden namelijk OPROEP, VADER, DEFINITIE, TERUG, START en KOPIE of GLOBAAL. Nochtans, in de kopieermethode kunnen de velden TERUG, START en KOPIE achterwege gelaten worden wanneer DEFINITIE = NIL (het veld DEFINITIE kan ook weggelaten worden door dit geval te coderen met een speciale bit in een der twee overblijvende velden); in de gewijzigde gedeelde strukturenmethode kunnen alleen de velden TERUG en START weggelaten worden.

Ook de "garbage collection", op de kopieënstapel of globale stapel is iets minder optimaal bij de gedeelde strukturenmethode. Inderdaad, een element op de globale stapel bestaat uit de bindingen van een groep variabelen $x_1,\ldots,x_n$. Wanneer x_i en x_k toegankelijk blijven, dan moet de volledige zone x_i, x_{i+1} ... x_{k-1}, x_k blijven bestaan, ook wanneer de variabelen $x_{i+1}\ldots x_{k-1}$ niet meer toegankelijk zijn.

Blijft nog het vergelijken van de plaats nodig voor de bindingen der variabelen. Deze vergelijking is moeilijker. Alvast hoeven we geen rekening te houden met loze en tijdelijke variabelen, de eersten nemen helemaal geen plaats in, de laatsten verdwijnen onmiddellijk na de unifikatie. In de gewijzigde gedeelde strukturenmethode is het eenvoudig : een lokale variabele vereist twee velden op de knooppuntenstapel (skelet en omgeving) en een globale variabele twee velden op de globale stapel. Met de kopieermethode vraagt elke variabele één veld op de knooppuntenstapel, terwijl de nodige plaats op de kopieënstapel afhangt van het patroon van de oproep. Wel kunnen we opmerken dat het kopiëren van een term met n argumenten, naast de kopieën van de argumenten, n+1 velden vereist, namelijk één veld om het funktiesymbool te identificeren en voor elk argument een wijzer naar de kopie van het argument. Algemene besluiten trekken is onmogelijk. Om zich een idee te vormen over de plaats ingenomen door de bindingen der variabelen moet men kijken naar diverse procedures en het geheugen, vereist voor het uitvoeren van een oproep, berekenen.

Om een vergelijking te maken tussen beide methodes nemen we onze toevlucht tot de reeks programma's die David Warren gebruikte voor het vergelijken van zijn "PROLOG-compiler" met andere systemen (LISP, POP-2). Procedures waarin alle variabelen loos of tijdelijk zijn laten we natuurlijk achterwege.

De procedures [Warren 77a]

(a) nreverse(x.ℓo,ℓ) ← nreverse(ℓo,ℓ1) concatenate(ℓ1,x.Nil,ℓ)

(b) concatenate(x.ℓ1,ℓ2,x.ℓ3) ← concatenate(ℓ1,ℓ2,ℓ3)

(c) qsort(x.ℓ,r,ro) ← partition(ℓ,x,ℓ1,ℓ2) qsort(ℓ2,r1,ro) qsort(ℓ1,r,x.r1)

(d) partition(x.ℓ,y,x.ℓ1,ℓ2) ← x<= y / partition(ℓ,y,ℓ1,ℓ2)

(e) partition(x.ℓ,y,ℓ1,x.ℓ2) ← partition(ℓ,y,ℓ1,ℓ2)

(f) d(u+v,x,du+dv) ← /d(u,x,du) d(v,x,dv)

(g) d(u-v,x,du-dv) ← /d(u,x,du) d(v,x,dv)

(h) d(u*v,x,du*v+u*dv) ← /d(u,x,du) d(v,x,dv)

(i) d(u/v,x,(du*v-u*dv)/v ∿ 2) ← /d(u,x,du) d(v,x,dv)

(j) d(u ∿n,x,du*n*u ∿n1) ← integer(n) n1=n-1 d(u,x,du)

(k) d(-u,x,-du) ← /d(u,x,du)

(l) d(exp(u),x,exp(u)*du) ← /d(u,x,du)

(m) d(log(u),x,du/u) ← /d(u,x,du)

(n) serialise(ℓ,r) ← pairlists(ℓ,r,a) arrange(a,t) numbered(t,1,n)

(o) pairlists(x.ℓ,y.r,pair(x,y).a) ← pairlists(ℓ,r,a)

(p) arrange(x.ℓ,tree(t1,x,t2)) ← split(ℓ,x,t1,ℓ1,ℓ2) arrange(ℓ1,t1)
arrange(ℓ2,t2)

(q) split(x.ℓ,x,ℓ1,ℓ2) ← /split(ℓ,x,ℓ1,ℓ2)

(r) split(x.ℓ,y,x.ℓ1,ℓ2) ← before(x,y)/split(ℓ,y,ℓ1,ℓ2)

(s) split(x.ℓ,y,ℓ1,x.ℓ2) ← before(y,x)/split(ℓ,y,ℓ1,ℓ2)

(t) before(pair(x1,y1), pair(x2,y2)) ← x1 < x2

(u) numbered(tree(t1,pair(x,n1),t2),no,n) ← numbered(t1,no,n1),
n2=n1+1 numbered(t2,n2,n)

(v) query(c1,d1,c2,d2) ← density(c1,d1) density(c2,d2)
d1 > d2 20*d1 < 21*d2

(w) density(c,d) ← pop(c,p) area(c,a) d=(p*100)/a

Voor een oproep van elk van deze procedures hebben we berekend hoeveel geheugenplaatsen ingenomen worden door de bindingen der variabelen op de knooppuntenstapel en op de globale of de kopieënstapel. Dit is gebeurd voor de kopieermethode en voor de verbeterde gedeelde strukturenmethode met en zonder modedeklaraties. Voor de gedeelde strukturenmethode is de berekening vrij eenvoudig, maar het is wat moeilijker om na te gaan welke elementen precies op de kopieënstapel geplaatst worden. We moeten rekening houden met de vorm van de oproep :

Voorbeeld

nreverse(x.ℓo,ℓ) ← nreverse(ℓo,ℓ1) concatenate(ℓ1,x.Nil,ℓ). De oproepen "nreverse" zijn steeds van de vorm x.y en het skelet x.ℓo moet dus niet gekopieerd worden. Wanneer echter de oproep "concatenate" zal uitgevoerd worden zal het skelet x.Nil moeten gekopieerd worden. Inderdaad, het tweede argument in de hoofding van de procedure "concatenate" is geen lijststruktuur maar enkel een variabele. Deze kopie wordt meegerekend met de procedure "nreverse".

De resultaten van de berekening zijn gegeven in volgende tabel :

GEHEUGENVERBRUIK

	Verbeterde gedeelde strukturenmethode				kopieermethode	
	zonder mododeklaraties		met mododeklaraties			
	knooppunten-stapel (alleen variabelen)	globale stapel	knooppunten-stapel (alleen variabelen)	globale stapel	knooppuntenstapel (alleen variabelen)	kopieën-stapel
(a)	4	4	6	2	4	3
(b)	2	6	4	4	3	3
(c)	8	6	10	4	7	3
(d)	4	6	6	4	5	3
(e)	4	6	6	4	4	3
(f)	2	8	6	4	5	3
(g)	2	8	6	4	5	3
(h)	2	8	2	8	5	9
(i)	2	8	2	8	5	15
(j)	2	8	2	8	5	9
(k)	2	4	4	2	3	2
(l)	2	4	2	4	3	5
(m)	2	4	2	4	3	3
(n)	10	0	10	0	5	0
(o)	0	10	2	8	3	9
(p)	4	8	6	6	6	4
(q)	4	4	8	0	4	0
(r)	4	6	6	4	5	3
(s)	4	6	5	4	5	3
(t)	0	8	4	0	2	0
(u)	6	8	12	0	6	0
(v)	8	0	8	0	4	0
(w)	8	0	8	0	4	0
totaal	86	130	128	82	101	83
	samen: 216		samen: 210		samen: 184	

Het verschil in totaal geheugengebruik tussen de gedeelde strukturenmethode met en zonder modedeklaraties is veroorzaakt door het feit dat, met modedeklaraties, bepaalde variabelen tijdelijk worden en verdwijnen.

Besluiten

Wat het totale geheugengebruik betreft is er een licht voordeel voor de kopieermethode (184 tegen 210 of 216, dit is een verschil van ruim 10 %). Belangrijker is echter de grootte van respektievelijk de globale en de kopieënstapel. Immers, op de knooppuntenstapel kunnen diverse geheugenbesparende technieken toegepast worden; elementen van de kopieënstapel (globale stapel) echter, worden slechts vrijgegeven wanneer het systeem op zijn stappen terugkeert ("backtracking").

De kopieënstapel is duidelijk kleiner dan de globale stapel wanneer geen modedeklaraties gebruikt worden (83 tegen 130). Met behulp van modedeklaraties kan de gebruiker de globale stapel tot een vergelijkbare grootte terugschroeven (83-82). Dat hiervoor de tussenkomst van de gebruiker nodig is moet echter als een nadeel beschouwd worden.

Ons besluit is dat de kopieermethode toch wel iets beter is :

- een kleinere knooppuntenstapel (minder vaste informatie)
- de "garbage collection" op de kopieënstapel is beter dan op de globale stapel
- tussenkomst van de gebruiker is essentieel om de globale stapel te beperken tot een grootte die vergelijkbaar is met de grootte van de kopieënstapel.

Dit besluit moet toch met de nodige voorzichtigheid gehanteerd worden. Procedure (i) is met de kopieermethode uitgesproken slechter dan met de gedeelde strukturenmethode (15 velden op de kopieënstapel tegen 8 op de globale stapel). Het betreft hier een procedure die een zeer groot resultaat opbouwt. Programma's met veel dergelijke procedures zullen zich uitgesproken slechter gedragen met de kopieermethode dan met de gedeelde strukturenmethode. Een voordeel van de gedeelde strukturenmethode is dat het omgekeerde verschijnsel (met modedeklaraties) blijkbaar veel zeldzamer is (Het is niet helemaal onmogelijk : een kopie van een term met n argumenten vraagt n+1 velden op de kopieënstapel; zijn die argumenten n verschillende variabelen dan heeft men n globale variabelen en dus 2n velden op de globale stapel; voor n groot is dit een verschil van 50 %).

Nota :

Warren gebruikt de modedeklaraties eveneens om een betere vertaling der proceduredefinities te bekomen, meer bepaald om het unifikatieproces te versnellen.

HOOFDSTUK 3 : HET CONTROLEREN VAN DE SELEKTIE DER PROCEDURE-OPROEPEN

A. INLEIDING

In PROLOG worden de verschillende procedure-oproepen "sequentieel" uitgevoerd : de linker oproep wordt geselekteerd en vervangen door het lichaam van de gebruikte proceduredefinitie. Deze selektieregel is eenvoudig om aan te leren en te gebruiken. Nochtans zijn er programma's die voor geen enkele volgorde der procedure-oproepen tot een aanvaardbare uitvoering leiden.

Voorbeeld

Een rij van de vorm

$p(x_o,y_o)$ ... $p(x_i,y_i).p(x_j,y_j)$ ...

moet voldoen aan twee voorwaarden :

- Een relatie "Twee" die zegt dat voor elk element $p(x_i,y_i)$ y_i het tweevoud is van x_i.
- Een relatie "Drie" die zegt dat voor elk paar opeenvolgende elementen $p(x_i,y_i)$ en $p(x_j,y_j)$ x_j het drievoud is van y_i.

Met x_o gegeven is hiermee de rij volledig bepaald (Om de rij eindig te houden, kunnen we ze met Nil afsluiten zodra $x_i \geqslant 10.000$).
Het volgende programma beschrijft deze rij :

```
Rij(r) ← Twee(r) Drie(r)
Twee(Nil) ←
Twee(p(xi,yi).r) ← Tweevoud(xi,yi) Twee(r)
Drie(p(xi,yi).Nil) ← xi ⩾ 10.000
Drie(p(xi,yi).p(xj,yj).r) ← xi < 10.000 Drievoud(yi,xj) Drie(p(xj,yj).r)
←Rij(p(1,y).z)
```

De struktuur van dit programma is zeer eenvoudig. De procedure Rij zegt dat r aan de twee voorwaarden moet voldoen. Nochtans, wat ook de orde is van de oproepen Twee en Drie in de procedure Rij, voor PROLOG is het zo goed als onmogelijk om een oplossing te vinden. Inderdaad, de relaties "Tweevoud", "Drievoud", "⩾" en "<" bestaan in principe uit een oneindig aantal proceduredefinities. Een oplossing kan slechts gevonden

worden wanneer de procedures "Twee" en "Drie" met elkaar samenwerken : wanneer "Twee" y_o berekent uit x_o en daarna "Drie" x_1 uit y_o; daarna moet "Twee" y_1 uit x_1 berekenen .... Met PROLOG kan men een dergelijke samenwerking niet organizeren. Men is gedwongen om een nieuw programma te ontwikkelen :

$Rij(p(x_i,y_i).Nil) \leftarrow x_i \geq 10.000\ Tweevoud(x_i,y_i)$
$Rij(p(x_i,y_i).p(x_j,y_j).r) \leftarrow x_i < 10.000\ Tweevoud(x_i,y_i)$
$\qquad Drievoud(y_i,x_j)\ Rij(p(x_j,y_j).r)$
$\leftarrow Rij(p(1,y).z)$

Alhoewel in dit nieuwe programma de twee afzonderlijke voorwaarden niet zo goed tot uiting komen, toch is het wellicht niet moeilijker om te begrijpen dan het oorspronkelijke. Het is trouwens vrij eenvoudig uit het oorspronkelijke programma af te leiden. Na het vervangen van de oproepen Twee en Drie in de procedure Rij door hun respektievelijke definities en het vervangen van de recursieve oproepen Twee en Drie door een recursieve oproep van Rij ("folding") bekomt men het nieuwe programma. Nochtans, aan de hand van de verdere voorbeelden zal de lezer zich ervan kunnen overtuigen dat het omvormen van een logisch programma tot een efficiënt PROLOG-programma niet steeds eenvoudig is en dat het PROLOG-programma veel ondoorzichtiger kan zijn dan het oorspronkelijke logisch programma.

De wijze waarop een programma wordt uitgevoerd (algoritme = volgorde van de stappen in de berekening) wordt bepaald zowel door de logika van het programma als door de controle ([Kowalski 76] : Algorithm = Logic + Control). Wanneer de controle de logika intact laat is de probleemspecifikatie enkel door de logika bepaald en kan de programma-ontwikkeling in twee fazen gebeuren : eerst de logika, dan de controle. Door het wijzigen van de controle bekomt men een ander algoritme om hetzelfde probleem op te lossen.

PROLOG heeft een weinig soepele controle en, zoals uit bovenstaand voorbeeld blijkt, is het niet altijd mogelijk om het gewenste gedrag (algoritme) te bekomen. Bovendien wijzigt het controleconcept "/" de logika en dus de betekenis van de programma's.

In dit hoofdstuk willen we de expressieve mogelijkheden van de logika der Horn-uitdrukkingen verruimen (met dezelfde logika andere en betere algoritmes bekomen) door het ontwikkelen van een aantal nieuwe controle-concepten. Een dergelijke opzet is van nature pragmatisch. We hebben ons laten leiden door de idee dat de concepten :

- eenvoudig moeten zijn en gemakkelijk te hanteren
- moeten toelaten om de controle modulair en gestruktureerd te ontwikkelen : zonder rekening te houden met alle details van het programma moet men kunnen beginnen met de controle op te leggen aan het hoogste niveau. Deze controle moet men mechanisch kunnen overdragen naar de lagere niveaus en, waar nodig, moet men aan deze lagere niveaus bijkomende controle kunnen opleggen.

Om onze concepten te verduidelijken zullen we met de verschillende procedure-oproepen van een probleemstelling processen associëren. Een proces voert één stap uit (unifikatie tussen oproep en hoofding van de gebruikte definitie), creëert zijn zonen (een proces voor elke oproep in het lichaam van de gebruikte proceduredefinitie) en verdwijnt. Met elke oproep in de opeenvolgende probleemstellingen is dus een proces geassocieerd. Voor wat de logika der programma's betreft, kunnen al deze processen gelijktijdig uitgevoerd worden. Het controleren van het programma heeft tot doel dit "netwerk" van processen te synchroniseren.

Bij het controleren van dit netwerk vertrekken we van een zogenoemde "luie evaluatie" (lazy evaluation) : een proces komt slechts voor uitvoering in aanmerking wanneer het kan bijdragen tot de berekening van een component waarvoor een aanvraag (verzoek) bestaat. Op zichzelf is dit natuurlijk onvoldoende om een efficiënte uitvoering te verzekeren. Inderdaad, een procedure kan met een willekeurig input/output patroon gebruikt worden, maar niet alle patronen resulteren in een behoorlijk gedrag. Daarom zullen we aan de oproepen (processen) bijkomende beperkingen opleggen. Wanneer een proces door een bepaalde aanvraag geaktiveerd wordt maar niet aan de beperkingen voldoet, wordt het verdaagd. Het verdaagde proces kan vragen om de ontbrekende informatie te berekenen. Eens deze informatie beschikbaar, verdwijnt de aanvraag en kan het verdaagde proces hernemen.

De modulaire opbouw vereist dat een beperking die aan een proces opgelegd is, eveneens geldt voor alle processen die door dit proces direct ("zonen") of indirect ("afstammelingen" van de zonen) gecreëerd worden.

Verder veronderstellen we dat de uitvoering eerlijk is : een proces dat voor uitvoering in aanmerking komt zal na een eindig aantal stappen inderdaad uitgevoerd worden.

In de volgende sektie zullen we aan de hand van een reeks voorbeelden deze controle-concepten nauwkeuriger beschrijven.

Daar de controle de logika intact laat, blijven uit de logika afgeleide resultaten over correctheid en theoretische eindigheid geldig. Door het expliciet opgeven van de beperkingen is het daarenboven gemakkelijker om zich ervan te overtuigen dat de uitvoering van de programma's ook werkelijk eindigt. Een programma zal alle oplossingen berekenen of in een gemakkelijk vast te stellen "dood" punt (een punt waarin geen enkel proces kan uitgevoerd worden) terecht komen indien elk proces (elke oproep) dat aan de opgelegde beperkingen voldoet :

- met een eindig aantal proceduredefinities overeenstemt (dit sluit niet uit dat een relatie door een oneindig aantal definities kan gedefinieerd zijn).
- en slechts een eindig aantal afstammelingen heeft. (indien men veronderstelt dat de afstammelingen voldoen aan de hun opgelegde beperkingen).

Opmerkingen

1. In een systeem waar meerdere processoren beschikbaar zijn zou men ook een "ijverige evaluatie" ("eager evaluation") kunnen beschouwen. In dit geval kan een proces dat aan alle beperkingen voldoet reeds componenten berekenen voordat ze aangevraagd worden.
2. Deze concepten werden ontwikkelde in nauwe samenwerking met Keith Clark (Queen Mary College - University of London) en worden voorgesteld in een artikel dat eerlang zal verschijnen.
3. Frank McGabe heeft de belangrijkste van deze concepten in IC-PROLOG ingebouwd. [McGabe 78]
4. In dit hoofdstuk noemen we een oproep (proces) deterministisch wanneer hij onder de gegeven beperkingen met maximaal één proceduredefinitie overeenstemt. In het andere geval is het een niet-deterministische oproep en kan hij aanleiding geven tot vertakkingspunten in de oplossingsruimte (OF-boom).

B. VOORBEELDEN

B.1. DE ZEEF VAN ERATOSTHENES ([Kahn & McQueen 77],[Hoare 78]

Dit is een algoritme om de (oneindige) rij der priemgetallen te berekenen. Het algoritme vertrekt van de geordende rij der natuurlijke getallen groter dan 1. Deze getallenrij wordt "gezeefd", ze wordt doorheen een initieel lege rij filters gestuurd. Het eerste element dat er doorheen komt is een nieuw priemgetal en vormt de basis van een nieuwe filter. Deze nieuwe filter wordt aan de rij toegevoegd en de volgende elementen moeten er doorheen. Het effekt van een filter is dat alle veelvouden van het priemgetal uit de rij vallen.

Dit algoritme kan vrij gemakkelijk als een logisch programma geformuleerd worden. Hiertoe maken we gebruik van de volgende relaties :

- Priem : een unaire relatie met als argument de rij der priemgetallen in stijgende orde.
- Getallenrij : een unaire relatie met als argument een rij van opeenvolgende natuurlijke getallen.
- Zeef : een binaire relatie tussen twee getallenrijen, de tweede rij bevat de getallen die overblijven nadat de eerste rij "gezeefd" wordt. Dus alle getallen die een veelvoud zijn van een voorafgaand getal worden verwijderd. (In dit programma zijn beide rijen geordend).
- Filter (p,ℓ1,ℓ2) is een ternaire relatie. ℓ2 is uit ℓ1 afgeleid door alle veelvouden van p te verwijderen. ℓ2 is dus de rij die bekomen wordt door ℓ1 doorheen de filter met basis p te sturen.
- Veelvoud : is n een veelvoud van p, dan behoort (n,p,T) tot deze relatie, anders behoort (n,p,F) ertoe.
- m=n+1 : een binaire relatie tussen natuurlijke getallen die geldt wanneer m de opvolger is van n (Deze twee laatste relaties beschouwen we als gedefinieerd door een expliciete opsomming van al hun elementen).

Het programma :

```
Priem(plist) ← Getallenrij(2.intlist) Zeef(2.intlist,plist)
Zeef(p.ℓ1,p.ℓ3) ← Filter(p,ℓ1,ℓ2) Zeef(ℓ2,ℓ3)
Filter(p,n.ℓ1,n.ℓ2) ← Veelvoud(n,p,F) Filter(p,ℓ1,ℓ2)
Filter(p,n.ℓ1,ℓ2) ← Veelvoud(n,p,T) Filter(p,ℓ1,ℓ2)
Getallenrij(n.m.ℓ) ← m=n+1 Getallenrij(m.ℓ)
←Priem(plist)
```

Dit programma heeft tot doel opeenvolgende "benaderingen" (in de zin dat er meer bekend is over het gezochte resultaat) van de rij der priemgetallen te berekenen. Voeren we dit programma uit dan krijgen we na één stap :

←Getallenrij(2.intlist) Zeef(2.intlist,plist)

Geen van beide processen, Getallenrij en Zeef, eindigt. Een "sequentiële" uitvoering is daarom alleszins onvoldoende. Daar de oproep Getallenrij plist niet bevat kan deze procedure onmogelijk een benadering van plist berekenen. De procedure Zeef kan dit wel maar, wegens het niet-determinisme van Filter en Veelvoud is er weinig kans dat Zeef meer dan één priemgetal zal berekenen. Getallenrij en Zeef moeten hun akties zó coördineren dat de uitvoering deterministisch blijft en het enige tot een oplossing leidende pad volgt.

In dit programma moet de uitvoering gedreven worden door een verzoek om plist te berekenen. We duiden dit verzoek expliciet aan door middel van een uitgaande (naar boven gerichte) pijl ↑ : ←Priem(plist ↑). Dit verzoek heeft tot gevolg dat het door Priem gecreëerde proces Zeef geselekteerd wordt. Echter, om de berekening op het rechte pad te houden, mogen Zeef en haar afstammelingen geen veronderstellingen maken over de vorm van intlist. Deze informatie moet komen van Getallenrij : we zeggen dat Getallenrij de producent en Zeef de konsument is van intlist. Het feit dat een proces konsument is van een variabele wil zeggen dat het geen toelating heeft om die variabele of gelijk welke component in die variabele te binden. Ditzelfde geldt voor al haar afstammelingen. Wordt die procedure of een afstammeling geselekteerd en is voor het uitvoeren ervan het binden van een dergelijke variabele noodzakelijk, dan wordt de uitvoering verdaagd totdat een ander proces die variabele gebonden heeft. Opdat dit laatste zou gebeuren wordt een verzoek om een benadering van die variabele te berekenen, aan het systeem toegevoegd. Zodra een eerste benadering gevonden wordt verdwijnt het verzoek en wordt de uitvoering van de verdaagde oproep hervat.

Is een geselekteerde procedure-oproep konsument van een vrije variabele, dan is deze oproep slechts uitvoerbaar indien de hoofding van de te gebruiken procedure-oproep op de overeenstemmende plaats geen constante of complexe term bevat. Deze voorwaarde is nodig maar wegens de algemeenheid van unifikatie, niet voldoende.

Voorbeeld : een proces P(A,x) dat konsument is van x en een hoofding P(y,y). Het verboden effekt van deze unifikatie is dat x gebonden wordt aan A, de oproep moet dus verdaagd worden.

De konsumentenbeperking op de variabele intlist in de oproep van Zeef duiden we aan met een binnenkomende (naar beneden gerichte) pijl ↓. (Om deze oproep voorlopig te onderscheiden van andere oproepen van dezelfde procedure die aan geen beperkingen onderworpen zijn, geven we de oproep een index 1). Priem(plist) ← Getallenrij(2.intlist) $Zeef_1$(2.intlist↓,plist). Alhoewel redundant, toch is het nuttig om deze beperking expliciet te maken in alle afstammelingen van $Zeef_1$ (dit kan mechanisch gebeuren). (Deze procedure is enkel bedoeld voor de oproep $Zeef_1$ en krijgt daarom ook een index 1). $Zeef_1$(p.ℓ1↓,p.ℓ3) ← $Filter_1$(p,ℓ1↓, ℓ2) Zeef(ℓ2,ℓ3). Zeef(ℓ2,ℓ3) is aan geen beperking onderworpen en doet dus een beroep op de oorspronkelijke procedure Zeef. Filter(p,ℓ1,ℓ2) heeft wel een beperking. Als we deze eveneens mechanisch op de Filter-procedures overbrengen krijgen we :

$Filter_1$(p,(n.ℓ1)↓,n.ℓ2) ← Veelvoud(n↓,p,F) $Filter_1$(p,ℓ1↓,ℓ2)

$Filter_1$(p,(n.ℓ1)↓,ℓ2) ← Veelvoud(n↓,p,T) $Filter_1$(p,ℓ1↓,ℓ2).

De recursieve oproepen van Filter hebben dezelfde beperkingen als de oorspronkelijke en doen dus eveneens beroep op $Filter_1$.

Ook in de procedure $Zeef_1$ moeten de oproepen nog verder op elkaar afgestemd worden. De bedoeling van het algoritme is dat Filter ℓ2 produceert en Zeef ℓ2 konsumeert. We krijgen dus :

$Zeef_1$(p.ℓ1↓,p.ℓ3) ← $Filter_1$(p,ℓ1↓,ℓ2) Zeef(ℓ2↓,ℓ3)

De recursieve oproep van Zeef krijgt dus een konsumentenbeperking op het volledige eerste argument. Deze beperking is in overeenstemming te brengen met $Zeef_1$(2.intlist↓), inderdaad, daar 2 constant is kunnen we evengoed schrijven Zeef((2.intlist)↓). We hebben dus slechts één type oproep en kunnen het stellen met één definitie :

Zeef((p.ℓ1)↓,p.ℓ3) ← Filter(p↓,ℓ1↓,ℓ2) Zeef(ℓ2↓,ℓ3)

Brengen we de beperkingen in de Filter-oproep over op de Filter-procedures, dan krijgen we :

Filter(p↓,(n.ℓ1)↓,n.ℓ2) ← Veelvoud(n↓,p↓,F) Filter(p↓,ℓ1↓,ℓ2)

Filter(p↓,(n.ℓ1)↓,ℓ2) ← Veelvoud(n↓,p↓,T) Filter(p↓,ℓ1↓,ℓ2)

Daar de recursieve oproepen aan dezelfde beperkingen onderhevig zijn, is ook dit ene stel procedures voldoende.

Onze volgende beslissing betreft het niet-determinisme in Filter. Daar beide oproepen van Veelvoud konsument zijn van n en p sluiten ze elkaar uit (ofwel F ofwel T als derde argument, maar niet beide) en er is dus slechts een van beide Filter-procedures toepasselijk. Om te verhinderen dat de berekening eventueel de verkeerde richting opgaat is het essentieel dat de oproep van de procedure Veelvoud uitgevoerd wordt vóór dat de recursieve Filter-oproep uitgevoerd wordt, ja zelfs vóór dat Filter zijn eventueel resultaat (n.ℓ2 of ℓ2) kenbaar maakt aan andere procedures. In feite wensen we de uitvoering van Veelvoud te zien als een uitbreiding van de unifikatie tussen de oproep en de hoofding van Filter. We wensen deze unifikatie slechts als geslaagd te beschouwen wanneer ook de oproep van Veelvoud slaagt. Dit betekent dat we een oproep van Filter slechts als uitvoerbaar beschouwen als ook de gecreëerde oproep van Veelvoud uitvoerbaar is. Dit alles zullen we aanduiden door een vertikale streep : |. De oproepen links van de streep worden als een uitbreiding van het patroon van de hoofding beschouwd :

Filter(p↓,(n.ℓ1)↓,n.ℓ2) ← Veelvoud(n↓,p↓,F) | Filter(p↓,ℓ1↓,ℓ2)

Filter(p↓,(n.ℓ1)↓,ℓ2) ← Veelvoud(n↓,p↓,T) | Filter(p↓,ℓ1↓,ℓ2)

Blijft nog het beheersen van de procedure Getallenrij. De procedure-oproep m=n+1 moet deterministisch gemaakt worden. Dit gebeurt door aan n een konsumentenbeperking op te leggen :

Getallenrij(n.m.ℓ) ← m=n↓+1 Getallenrij(m.ℓ).

Verhinderen dat de recursieve oproep uitgevoerd wordt vooraleer de oproep m=n+1 uitgevoerd is, kunnen we doen door te stellen dat Getallenrij(m.ℓ) slechts uitvoerbaar is indien m <u>volledig</u> bepaald is. We duiden deze beperking aan door een index <u>val</u>. Deze beperking is in overeenstemming te brengen met de oproep Getallenrij(2.intlist); 2 is immers een constante. Deze beperking laten doordringen tot de afstammelingen resulteert in de procedure Getallenrij(n_{val}.m.ℓ) ← m=n_{val}+1 Getallenrij(m.ℓ) (n_{val} in het lichaam van de procedure is geen beperking maar een eigenschap die automatisch voldaan is wanneer de oproep op Getallenrij uitvoerbaar was en maakt de konsumentenbeperking overbodig.) Het volledige gecontroleerde programma wordt :

```
Priem(plist) ← Getallenrij(2val.intlist) Zeef((2.intlist)↓,plist)
Zeef((p.ℓ1)↓,p.ℓ3) ← Filter(p↓,ℓ1↓,ℓ2) Zeef(ℓ2↓,ℓ3)
Filter(p↓,(n.ℓ1)↓,n.ℓ2) ← Veelvoud(n↓,p↓,F)| Filter(p↓,ℓ1↓,ℓ2)
Filter(p↓,(n.ℓ1)↓,ℓ2) ← Veelvoud(n↓,p↓,T)| Filter(p↓,ℓ1↓,ℓ2)
Getallenrij(nval.m.ℓ) ← m=nval+1 Getallenrij(mval,ℓ)
←Priem(plist↑)
```

Zowel met een "lui" als met een "ijverig" evaluatiemechanisme zal de uitvoering van het programma deterministisch zijn en zullen opeenvolgende benaderingen van de rij der priemgetallen berekend worden.
Men verkrijgt een eindige deelrij door de procedure Getallenrij eindig te maken en de procedure Zeef aan te passen. Dit kan met het volgende programma :

```
Getallenrij(nval.Nil) ← nval ≥ 10.000|
Getallenrij(nval.m.ℓ) ← nval < 10.000|m=nval+1  Getallenrij(mval.ℓ)
Priem(plist) ← Getallenrij(2val.intlist) Zeef((2.intlist)↓,plist)
Zeef(Nil↓,Nil) ←
Zeef((p.ℓ1)↓,p.ℓ3) ← Filter(p↓,ℓ1↓,ℓ2) Zeef(ℓ2↓,ℓ3)
Filter(p↓,Nil↓,Nil) ←
Filter(p↓,(n.ℓ1)↓,n.ℓ2) ← Veelvoud(n↓,p↓,F)| Filter(p↓,ℓ1↓,ℓ2)
Filter(p↓,(n.ℓ1)↓,ℓ2) ← Veelvoud(n↓,p↓,T)| Filter(p↓,ℓ1↓,ℓ2)
←Priem(plist↑)
```

B.2. EEN SORTEERALGORITME

In [Hansen 78] vinden we een algoritme om, met behulp van m processoren, een rij van m getallen te sorteren in een tijd evenredig met m. Dit programma heeft een zeer eenvoudig logisch equivalent. De logische formulering maakt gebruik van :

- een binaire relatie Sort(ℓ,s) waarbij s de gesorteerde lijst ℓ is.
- een ternaire relatie S(ℓ,min,r) waarbij min het kleinste element is uit de lijst ℓ en de lijst r, op min na, dezelfde elementen bevat als ℓ (niet noodzakelijk in dezelfde volgorde).
- de binaire relaties $\leqslant$ en $>$ gedefinieerd door hun elementen.

Het programma is als volgt :

```
Sort(Nil,Nil) ←
Sort(x.y,min.s) ← S(x.y,min,r) Sort(r,s)
S(x.Nil,x,Nil) ←
S(x.y.z,min,y.r) ← x≤y  S(x.z,min,r)
S(x.y.z,min,x.r) ← x>y  S(y.z,min,r)
←Sort(5.3.9.1.12.Nil,s)
```

Om de uitvoering van dit programma deterministisch te houden moet enige controle-informatie toegevoegd worden. Het programma heeft tot doel de gesorteerde lijst s te berekenen. Dit duiden we aan door :
←Sort(5.3.9.1.12.Nil,s↑).
In de recursieve procedure Sort hebben we een variabele r die in de hoofding niet voorkomt. Daar x.y gegeven en min.s te berekenen valt is het vrij duidelijk dat de oproep S de producent van r en de recursieve oproep van Sort de konsument van r moet zijn, dus Sort(r↓,s). Deze beperking kan eveneens toegepast worden op de initiële oproep : Sort((5.3.9.1.12.Nil)↓,s↑). Deze beperking expliciet maken in alle afstammelingen van de oproepen Sort resulteert in :

```
←Sort((5.3.9.1.12.Nil↓,s↑)
Sort(Nil↓,Nil) ←
Sort((x.y)↓,min.s) ← S((x.y)↓,min,r) Sort(r↓,s)
S((x.Nil)↓,x,Nil) ←
S((x.y.z)↓,min,y.r) ← x↓≤y↓  S((x.z)↓,min,r)
S((x.y.z)↓,min,x.r) ← x↓>y↓  S((y.z)↓,min,r)
```

Wegens de konsumentenbeperking sluiten x≤y en x>y elkaar uit. Alhoewel een oproep van S met twee proceduredefinities kan overeenstemmen, moet dus een van beide tot een blokkering leiden. Door de testen x≤y en x>y te beschouwen als een uitbreiding van het patroon in de hoofding (|) wordt de uitvoering volledig deterministisch. Inderdaad, de opgelegde beperkingen sluiten uit dat meerdere procedures met een oproep overeenstemmen. Het programma, samen met alle controle-informatie is dan als volgt :

```
←Sort((5.3.9.1.12.Nil)↓,s↑)
Sort(Nil↓,Nil) ←
Sort((x.y)↓,min.s) ← S((x.y)↓,min,r) Sort(r↓,s)
S((x.Nil)↓,x,Nil) ←
S((x.y.z)↓,min,y.r) ← x↓ ≤ y↓ | S((x.z)↓,min,r)
S((x.y.z)↓,min,x.r) ← x↓ > y↓ | S((y.z)↓,min,r)
```

De uitvoering die Hansen voor ogen staat is een ijverige evaluatie waarbij elk proces S, dat door de opeenvolgende processen Sort gecreëerd wordt, een eigen processor toegewezen krijgt. Ook in onze luie evaluatie is er (pseudo) parallellisme tussen de processen S en Sort.

We krijgen een meer klassieke sequentiële uitvoering door te eisen dat Sort slechts mag starten wanneer zijn eerste argument volledig bepaald is :

```
Sort(Nil_val,Nil) ←
Sort((x.y)_val,min,s) ← S((x.y)_val,min,r) Sort(r_val,s)
```

B.3. EEN MERKWAARDIGE RIJ

Dit voorbeeld, afkomstig van Dijkstra, wordt eveneens behandeld in [Kahn & McQueen 77]. Gevraagd wordt om een geordende rij op te stellen van alle getallen die behoren tot de verzameling $\{2^a 3^b 5^c$ met $a,b,c>0\}$. Deze rij heeft een merkwaardige eigenschap : vermenigvuldigen we haar met 2 en ook eens met 3 en met 5 en voegen we de drie aldus bekomen rijen samen tot een geordende rij (waarbij we eventuele duplikaten weglaten), dan bekomen we, op het eerste element na, de oorspronkelijke rij. Deze eigenschap is daarenboven konstruktief : als het eerste element (getal 1) gegeven is kan de eigenschap gebruikt worden om de volledige rij te berekenen.

Om dit algoritme als een logisch programma te schrijven doen we een beroep op de volgende relaties :

- Oplossing : een unaire relatie met als argument een lijst die aan de hogervermelde eigenschap voldoet.
- Mengen(ℓ1,ℓ2,ℓ12) : ℓ1,ℓ2 en ℓ12 zijn alle drie geordende lijsten zonder duplikaten. Daarenboven bevat ℓ12 dezelfde elementen als ℓ1 en ℓ2 samen.

- Vijf(ℓ1,ℓ2)
  Drie(ℓ1,ℓ2)
  Twee(ℓ1,ℓ2) : dit zijn binaire relaties waarbij ℓ2 uit ℓ1 afgeleid is door de elementen van ℓ1 met respektievelijk 5, 3 en 2 te vermenigvuldigen.
- x<y : de binaire relatie kleiner dan
- x=y*z : deze ternaire relatie definieert de vermenigvuldiging

Ons logisch programma is als volgt :

```
←Oplossing(1,ℓ)
Oplossing(u.ℓ) ← Vijf(u.ℓ,ℓ1) Drie(u.ℓ,ℓ2)
                 Twee(u.ℓ,ℓ3)
                 Mengen(ℓ1,ℓ2,ℓ12) Mengen(ℓ12,ℓ3,ℓ)
Mengen(x.ℓ1,x.ℓ2,x.ℓ12) ← Mengen(ℓ1,ℓ2,ℓ12)
Mengen(x.ℓ1,y.ℓ2,x.ℓ12) ← x<y  Mengen(ℓ1,y.ℓ2,ℓ12)
Mengen(x.ℓ1,y.ℓ2,y.ℓ12) ← y<x  Mengen(x.ℓ1,ℓ2,ℓ12)
Vijf(u.ℓ1,v.ℓ2) ← v=5*u  Vijf(ℓ1,ℓ2)
Drie(u.ℓ1,v.ℓ2) ← v=3*u  Drie(ℓ1,ℓ2)
Twee(u.ℓ1,v.ℓ2) ← v=2*u  Twee(ℓ1,ℓ2)
```

De producent-konsument verhouding in de procedure Oplossing is terug eenvoudig : Vijf, Drie en Twee konsumeren u.ℓ om respektievelijk ℓ1, ℓ2 en ℓ3 te produceren. De eerste oproep van Mengen konsumeert ℓ1 en ℓ2 om ℓ12 te produceren, de tweede konsument ℓ12 en ℓ3 om ℓ te produceren. We krijgen dus :

```
←Oplossing(1.ℓ↑)
Oplossing(u.ℓ) ← Vijf((u.ℓ)↓,ℓ1) Drie((u.ℓ)↓,ℓ2)
                 Twee((u.ℓ)↓,ℓ3)
               Mengen((ℓ1↓,ℓ2↓,ℓ12) Mengen(ℓ12↓,ℓ3↓,ℓ)
```

Deze beperkingen mechanisch doorvoeren in de afstammelingen van de diverse procedures resulteert in :

```
Mengen((x.ℓ1)↓,(x.ℓ2)↓,x.ℓ12) ← Mengen(ℓ1↓,ℓ2↓,ℓ12)
Mengen((x.ℓ1)↓,(y.ℓ2)↓,x.ℓ12) ← x↓<y↓ Mengen(ℓ1↓,(y.ℓ2)↓,ℓ12)
Mengen((x.ℓ1)↓,(y.ℓ2)↓,y.ℓ12) ← y↓<x↓ Mengen((x.ℓ1)↓,ℓ2↓,ℓ12)
Vijf((u.ℓ1)↓,v.ℓ2) ← v=5*u↓  Vijf(ℓ1↓,ℓ2)
Drie((u.ℓ1)↓,v.ℓ2) ← v=3*u↓  Drie(ℓ1↓,ℓ2)
Twee((u.ℓ1)↓,v.ℓ2) ← v=2*u↓  Twee(ℓ1↓,ℓ2)
```

De opgelegde beperkingen maken de oproepen van de vermenigvuldigingsrelatie deterministisch. Enig niet-determinisme in de procedures Mengen blijft echter mogelijk. Onder de opgelegde beperkingen sluiten $x<y$ en $y<x$ elkaar uit. Opdat dit ogenblikkelijk zou gebeuren moeten deze testen beschouwd worden als een uibreiding van het patroon van de oproep ($|$). De beperkingen laten eveneens toe dat de eerste procedure van Mengen gebruikt wordt wanneer het eerste element van de twee inputlijsten nog onbekend is. Ten onrechte worden dan beide eerste elementen aan elkaar gelijkgesteld. We kunnen dit verhinderen door te eisen dat een oproep van Mengen slechts mag uitgevoerd worden wanneer het eerste element van beide inputlijsten bekend is ("val" beperking die we hier aanduiden in de hoofding van de procedures). Het volledig gecontroleerd programma wordt :

$\leftarrow$Oplossing(1.$\ell\uparrow$)

Oplossing(u.ℓ) $\leftarrow$ Vijf((u.ℓ)$\downarrow$,ℓ1) Drie((u.ℓ)$\downarrow$,ℓ2)
Twee((u.ℓ)$\downarrow$,ℓ3)
Mengen(ℓ1$\downarrow$,ℓ2$\downarrow$,ℓ12) Mengen(ℓ12$\downarrow$,ℓ3$\downarrow$,ℓ)

Mengen((x_{val}.ℓ1)$\downarrow$,(x_{val}.ℓ2)$\downarrow$,x.ℓ12) $\leftarrow$ Mengen(ℓ1$\downarrow$,ℓ2$\downarrow$,ℓ12)

Mengen((x_{val}.ℓ1)$\downarrow$,(y_{val}.ℓ2)$\downarrow$,x.ℓ12) $\leftarrow$ $x_{val}<y_{val}$ | Mengen(ℓ1$\downarrow$,(y_{val}.ℓ2)$\downarrow$,ℓ12)

Mengen((x_{val}.ℓ1)$\downarrow$,(y_{val}.ℓ2)$\downarrow$,y.ℓ12) $\leftarrow$ $y_{val}<x_{val}$ | Mengen((x_{val}.ℓ1)$\downarrow$,ℓ2$\downarrow$,ℓ12)

Vijf((u.ℓ1)$\downarrow$,v.ℓ2) $\leftarrow$ v=5*u$\downarrow$ Vijf(ℓ1$\downarrow$,ℓ2)

Drie((u.ℓ1)$\downarrow$,v.ℓ2) $\leftarrow$ v=3*u$\downarrow$ Drie(ℓ1$\downarrow$,ℓ2)

Twee((u.ℓ1)$\downarrow$,v.ℓ2) $\leftarrow$ v=2*u$\downarrow$ Twee(ℓ1$\downarrow$,ℓ2)

Tijdens een <u>luie</u> evaluatie zullen de diverse procedures slechts een element berekenen van de rijen ℓ1,ℓ2,ℓ3 en ℓ12 wanneer er een expliciet verzoek bestaat en zullen deze rijen in feite slechts één aktief element bevatten. Een <u>ijverige</u> evaluatie kan op de vraag vooruitlopen en, telkens een nieuw element van de lijst ℓ gevonden wordt, een nieuwe element berekenen van de rijen ℓ1,ℓ2 en ℓ3 en, wanneer mogelijk, ook van ℓ12.

B.4. <u>MEERDERE PRODUCENTEN VOOR EENZELFDE VARIABELE</u>

Tot nu toe hadden we in onze voorbeelden slechts één producent voor elke variabele. Dit is niet altijd het geval. Nemen we terug het voorbeeld dat we in het begin van dit hoofdstuk gebruikten :

```
←Rij(p(1,y).z)
Rij(r) ← Twee(r) Drie(r)
Twee(p(x_i,y_i).r ← Tweevoud(x_i,y_i) Twee(r)
Drie(p(x_i,y_i).p(x_j,y_j).r) ← Drievoud(y_i,x_j) Drie(p(x_j,y_j).r)
```

In de procedure Rij hebben we geen producent-konsument relatie voor de variabele r. Zowel Twee als Drie zijn producent van bepaalde componenten van r. De controle moet op een lager niveau gebeuren.

De oproepen Tweevoud en Drievoud moeten deterministisch gemaakt worden. Dit kan gebeuren door een konsumentenbeperking op een van hun argumenten. Het eerste argument is hiertoe aangewezen. Tevens is het wenselijk dat de rij niet uitgebreid wordt vooraleer de aanwezige elementen volledig bepaald zijn. Dit kan door te eisen dat de oproepen Twee en Drie slechts uitvoerbaar zijn wanneer welbepaalde componenten gekend zijn. Het volgende programma verwezenlijkt het gewenste gedrag :

```
←Rij(p(1,y).z)
Rij(r) ← Twee(r) Drie(r)
Twee(p(x_i_val,y_i),r) ← Tweevoud(x_i_val,y_i) Twee(r)
Drie(p(x_i,y_i_val).p(x_j,y_j).r) ← Drievoud(y_i_val,x_j) Drie(p(x_j,y_j).r)
```

Een alternatieve oplossing kan erin bestaan de hoofding van Drie te schrijven als $Drie((p(x_i,y_i))_{val}.p(x_j,y_j).r)$
Beide alternatieven resulteren in een gedrag waarbij Twee en Drie om beurten een component berekenen.

B.5. HET 8-KONINGINNENPROBLEEM

Gevraagd wordt om 8-koninginnen op een schaakbord te plaatsen en wel zó dat ze elkaar niet kunnen aanvallen. Dus maximaal één koningin op elke rij, op elke kolom en op elke diagonaal. Het probleem kan veralgemeend worden tot het plaatsen van n-koninginnen op een bord met zijde n.

Daar er slechts n rijen en kolommen zijn en elke rij en kolom maximaal één koningin mag bevatten, moet elke rij en elke kolom juist één koningin bevatten. Een dergelijke opstelling is een kandidaatoplossing. Ze is slechts een oplossing indien geen twee koninginnen op dezelfde diagonaal

geplaatst zijn. Met een procedure Generator die de kandidaatoplossingen berekent en een procedure Controle die de diagonalen controleert, krijgen we volgend programma :

Koninginnen(n,bord) ← Generator(n,bord) Controle(bord)
←Koninginnen(8,bord↑).

Een goede werking van de procedure Koninginnen vereist dat n gegeven is (n_{val}) en dat Generator de producent en Controle de konsument van bord is :

Koninginnen(n_{val},bord) ← Generator(n_{val},bord) Controle(bord↓)

Dit is echter onvoldoende om een efficiënte werking te verzekeren. Het is essentieel dat Controle niet wacht totdat Generator een volledige opstelling berekent heeft, maar de opeenvolgende benaderingen van die opstelling controleert en de berekening onderbreekt van zodra twee konginnnen zich op dezelfde diagonaal bevinden. In dat geval moet Generator op zijn stappen terugkeren en een andere opstelling opbouwen. Met onze huidige concepten is dit gedrag mogelijk maar niet verplicht. Opdat het "Controle"-proces het "Generator"-proces zo dicht mogelijk zou volgen voeren we een nieuwe concept in : een "ijverige" konsument. We schrijven Controle(bord↓↑), dit wil zeggen dat Controle een ijverige konsument is van de variabele bord. Het gewenste effekt is dat Controle en al zijn afstammelingen (dus niet alle oproepen!) gedreven worden door een verzoek om bord te berekenen (daarom een uitgaande pijl) en dat dit verzoek prioriteit heeft over alle andere aanvragen. Nochtans blijft de konsumentenbeperking geldig : van zodra het nodig is om bord te binden wordt de oproep verdaagd en wordt de eigenlijke producent geaktiveerd en verzocht om een eerste benadering op te stellen. Het is dus Controle die de uitvoering aandrijft en ten gepaste tijde Generator om meer informatie verzoekt.

We moeten natuurlijk nog de procedures Generator en Controle nader specifiëren. Generator moet alle mogelijke kandidaatoplossingen genereren. Nummeren we de rijen en kolommen van 1 tot n dan is de positie van een koningin bepaald door haar rij- en kolomnummer. Alle mogelijke opstellingen bekomen we door, voor een willekeurig gekozen orde der rijnummers, alle permutaties der kolomnummers te beschouwen. We kunnen dit als volgt beschrijven :

```
Generator(n_val,bord) ← Nummers(n_val,rij)
                        Permutatie(rij↓,kolom)
                        Opstelling(rij↓,kolom↓,bord)
```

De beperking op n is afkomstig van de procedure Koninginnen. Een goede werking vereist dat Permutatie rij konsumeert en dat Opstelling rij en kolom konsumeert.

Kiezen we een lijstvoorstelling voor rij, dan kunnen we de procedure Nummers schrijven als :

```
Nummers(0_val,Nil) ←
Nummers(n_val,n.ℓ) ← n_val > 0 | m=n_val-1
                                 Nummers(m_val,ℓ)
```

De beperking dat het eerste argument moet gegeven zijn, maakt de oproepen $n>0$ en $m=n-1$ deterministisch. Tevens wordt $n>0$ als een uitbreiding beschouwd van het patroon van de hoofding. Dit verzekert dat slechts een der beide procedures met een oproep Nummers overeenstemt.

```
Permutatie(Nil↓,Nil) ←
Permutatie((x.y)↓,u.v) ← Verwijder(u,(x.y)↓,w) Permutatie(w↓,v)
Verwijder(x,(x.y)↓,y) ←
Verwijder(u,(x.y)↓,x.z) ← Verwijder(u,y↓,z)
```

Hierin is Verwijder(u,v,w) een ternaire relatie die bepaalt dat w uit v afgeleid is door het verwijderen van een willekeurig element u. Naast de door de oproep van Permutatie(rij↓,kolom) opgelegde beperking hebben we dat Verwijder de producent en Permutatie de konsument is van de lokale variabele w.

Stellen we een opstelling voor als een lijst van koninginnen waarbij een koningin wordt voorgesteld door p(r,k) met r het rijnummer en k het kolomnummer, dan kunnen we de procedure Opstelling definiëren als :

```
Opstelling(Nil↓,Nil↓,Nil)
Opstelling((x.y)↓,(u.v)↓,p(x,u).r) ← Opstelling(y↓,v↓,r)
```

Beschouwen we een opstelling als een geordende rij, dan kan de Controle erin bestaan om elke koningin met al haar opvolgers te vergelijken. Staan ze op eenzelfde diagonaal, dan is de oplossing onaanvaardbaar.

```
Controle(Nil↓) ←
Controle((p.q)↓) ← Opvolgers(p↓,q↓) Controle(q↓)
Opvolgers(p↓,Nil↓) ←
Opvolgers(p↓(q.r)↓) ← Diagonaal(p↓,q↓) Opvolgers(p↓,r↓)
```

Deze beperkingen laten toe dat de procedure-oproepen, die van de initiële Controle-oproep afstammen, in parallel uitgevoerd worden en dat er dus gelijktijdig om meerdere koninginnen verzocht wordt. Door het invoeren van enkele "val"-beperkingen verzekeren we dat de koninginnen één per één berekend worden :

```
Controle(Nil_val) ←
Controle((p_val.q)↓) ← Opvolgers(p_val,q↓) Controle(q↓)
Opvolgers(p_val,Nil_val) ←
Opvolgers(p_val,(q_val.r)↓) ← Diagonaal(p_val,q_val) Opvolgers(p_val,r↓)
```

We veronderstellen dat Diagonaal gedefinieerd is door een opsomming van alle posities die niet op dezelfde diagonaal liggen.

Het volledige programma is dan :

```
←Koninginnen(8_val,bord)
Koninginnen(n_val,bord) ← Generator(n_val,bord) Controle(bord↓↑)
Generator(n_val,bord) ← Nummers(n_val,rij) Permutatie(rij↓,kolom)
                        Opstelling(rij↓,kolom↓,bord)
Nummers(0_val,Nil) ←
Nummers(n_val,n.ℓ) ← n_val > 0 | m=n_val-1 Nummers(m_val,ℓ)
Permutatie(Nil↓,Nil) ←
Permutatie((x.y)↓,u.v) ← Verwijder(u,(x.y)↓,w) Perm(w↓,v)
Verwijder(x,(x.y)↓,y) ←
Verwijder(u,(x.y)↓,x.z) ← Verwijder(u,y↓,z)
Opstelling(Nil↓,Nil↓,Nil) ←
Opstelling((x.y)↓,(u.v)↓,p(x,u).r) ← Opstelling(y↓,v↓,r)
Controle(Nil_val) ←
Controle((p_val.q)↓) ← Opvolgers(p_val,q↓) Controle(q↓)
Opvolgers(p_val,Nil_val) ←
Opvolgers(p_val,(q_val.r)↓) ← Diagonaal(p_val,q_val) Opvolgers(p_val,r↓)
```

B.6. HET PROFIEL VAN BINAIRE BOMEN

Stellen we een eindknooppunt met informatie x voor door ℓ(x) en een binaire boom met linkertak x en rechtertak y door x⁂y, dan kunnen we een procedure Profiel definiëren die de lijst der eindknooppunten berekent :

```
Profiel(x⁂y,v) ← Profiel(x,s) Profiel(y,t) Append(s,t,v)
Profiel(ℓ(x),x.Nil) ←
Append(Nil,x,x) ←
Append(x.u,v,x.w) ← Append(u,v,w)
```

Kijken of twee bomen hetzelfde profiel hebben kan met een procedure Zelfdeprofiel :

```
Zelfdeprofiel(x,y) ← Profiel(x,z) Profiel(y,z)
```

Een goede werking vereist dat beide oproepen van de procedure Profiel hun akties coördineren. Indien x en y in hun eerste eindknooppunt verschillen, is het nutteloos om de verdere eindknooppunten te berekenen. We hebben hier te doen met twee evenwaardige producenten van de variabele z. Leggen we geen beperkingen op, dan kan de ene sterk vooruit lopen op de andere. We kunnen ze synchroniseren door gebruik te maken van het "ijverige konsument"-concept :

```
Zelfdeprofiel(x↓,y↓) ← Profiel(x↓,z) Profiel(y↓,z↓↑)
```

Hierin is Profiel(x,z) de producent en Profiel(y,z) de (ijverige) konsument van z. Deze ijverige konsument heeft prioriteit en wordt gedreven door een verzoek om z te berekenen maar wordt verdaagd zodra het noodzakelijk wordt om z te binden. Op dat moment wordt de producent verzocht om een eerste benadering van z op te stellen.
Beide oproepen van Profiel hebben een verschillend patroon. De beperkingen uit dit patroon mechanisch opleggen aan de definities resulteert in twee verschillende stellen definities :

```
Profiel((x⁂y)↓,v) ← Profiel(x↓,s) Profiel(y↓,t) Append(s↓,t,v)
```

Append wordt konsument van s, dit verzekert dat slechts één definitie van Append met de oproep overeenstemt :

Profiel(ℓ(x)↓,x.Nil) ←

Append(Nil↓,x,x) ←

Append((x.u)↓,v,x.w) ← Append(u↓,v,w)

Daarnaast hebben we :

Profiel((x*y)↓,v↓) ← Profiel(x↓,s) Profiel(y↓,t) Append(s↓,t,v↓)

Profiel(ℓ(x)↓,(x.Nil)↓) ←

Append(Nil↓,x,x) ←

Append((x.u)↓,v,(x.w)↓) ← Append(u↓,v,w↓)

In beide stellen is de enige nieuwe beperking de konsumentenbeperking op s in Append(s,t,v).

Bemerk dat de uitvoering van Append(s↓,t,v↓) met Append(Nil↓,x,x↓) resulteert in de binding van t aan v. Het gevolg is dat alle procedures die t bevatten en van Profiel(y,z↓↑) afstammen eveneens een konsumentenbeperking opgelegd krijgen (Profiel(y↓,t) wordt Profiel (y↓,t↓)). Dit is een voorbeeld van een bijkomende beperking die een nevenprodukt is van de unifikatie.

C. EEN OVERZICHT VAN DE DIVERSE CONCEPTEN

Bij het uitvoeren van een programma stelt het evaluatiemechanisme zich tot doel bepaalde variabelen volledig te berekenen. Deze gezochte variabelen zijn de drijvende krachten van de berekening. We onderscheiden twee soorten. Enerzijds zijn er de globaal gezochte variabelen (resultaten). We hebben ze aangeduid met een uitgaande pijl (vb. Priem(plist↑)). Anderzijds zijn er de zogenoemde ijverig gekonsumeerde variabelen (vb. Controle(bord↓↑)). Het zoeken naar deze variabelen heeft een permanent karakter. Eens een eerste benadering gevonden, wordt verder gezocht naar de nog ontbrekende componenten.

Een ijverige konsument zoals Controle heeft gewoonlijk tot doel na te gaan dat een berekende struktuur aan bepaalde voorwaarden voldoet. Verdere berekeningen hebben slechts zin wanneer dit inderdaad het geval is.

Daarom plaatsen wij ijverige konsumenten in de hoogste prioriteitsklasse. Ook processen die van een ijverige konsument afstammen en de ijverig gekonsumeerde variabele bevatten (we bedoelen zowel de oorspronkelijke ijverig gekonsumeerde variabele als variabelen uit de struktuur waaraan die variabele gebonden werd) komen in die prioriteitsklasse.

Processen die een globaal gezochte variabele bevatten komen in de tweede prioriteitsklasse. De andere processen vallen in de derde en laagste prioriteitsklasse.

Bij elke stap van de uitvoering wordt, op een eerlijke wijze, een proces geselekteerd uit de hoogste niet ledige prioriteitsklasse.

Processen kunnen ook nog onderhevig zijn aan beperkingen. We hebben twee soorten beperkingen gebruikt :

- De "val"-beperking : de eis dat een bepaalde component volledig bepaald is (vb. : Getallenrij(n_{val}.m.ℓ).
- De konsumentenbeperking : de eis dat een proces een bepaalde component niet mag berekenen (binden) (vb. : Filter(p↓,ℓ1↓,ℓ2)).

Voldoet een proces niet aan de opgelegde beperkingen, dan wordt het verdaagd. Daarbij kan het andere processen verzoeken om een eerste benadering van de ontbrekende informatie (variabele(n)). We onderscheiden twee basisgevallen :

1. Het verdaagde proces behoort tot de tweede prioriteitsklasse. Processen die de ontbrekende informatie (variabelen) bevatten worden geaktiveerd : totdat een eerste benadering gevonden is komen ze in de tweede prioriteitsklasse. Eens die benadering gevonden komt het verdaagde proces terug in aanmerking om uitgevoerd te worden.
2. Het verdaagde proces behoort tot de hoogste prioriteitsklasse.
   a. De variabele die de verdaging veroorzaakt is geen ijverig gekonsumeerde variabele. Alleen de processen die van de oorspronkelijke ijverige konsument afhangen en de gevraagde variabele bevatten, worden geaktiveerd : totdat een eerste benadering gevonden is komen ze in de hoogste prioriteitsklasse.
   b. De verdaging is veroorzaakt door een ijverig gekonsumeerde variabele. Slechts nádat alle andere processen uit de hoogste prioriteitsklasse uitgevoerd of verdaagd werden, worden de producenten van de variabele geaktiveerd : totdat een eerste benadering gevonden is komen de processen, die de gevraagde variabele bevatten en niet van de oorspronkelijke ijverige konsument afstammen, in de hoogste prioriteitsklasse.

Het zijn dus de variabelen die bepalen in welke prioriteitsklassen de processen terecht komen.

Een laatste concept had slechts een zeer lokaal effekt. Het betrof "|" zoals in :

Filter(p↓,(n.ℓ1)↓,ℓ2) ← Veelvoud(n↓,p↓,T)|Filter(p↓,ℓ1↓,ℓ2)

De oproepen die "|" voorafgaan moeten beschouwd worden als een uitbreiding (extra voorwaarden) van het patroon in de hoofding. Samen met de oproep die de betreffende proceduredefinitie gebruikt, worden ze uitgevoerd.

Voorbeeld van een evaluatie met een ijverige konsument

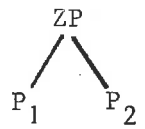
Beschouwen we de procedure Zelfdeprofiel zoals omschreven in de vorige sektie (B.6). Een mogelijke oproep is :

←Zelfdeprofiel((ℓ(A)::ℓ(B))↓,(ℓ(A)::ℓ(C))↓)

Deze enige oproep wordt geselekteerd en uitgevoerd. Dit leidt tot :

←$Profiel_1$((ℓ(A)::(ℓ(B))↓,z0) $Profiel_2$((ℓ(A)::ℓ(C))↓,z0↓↑)

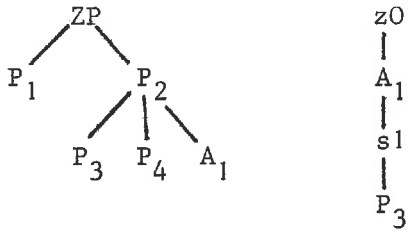
(Om een onderscheid te maken tussen de verschillende oproepen gebruiken we indices). We gebruiken twee strukturen om de uitvoering te visualiseren. Enerzijds, een zeer schematische EN-boom. Anderzijds, een boomstruktuur die de processen op het hoogste prioriteitsniveau bevat en die aangeeft omwille van welke variabelen die processen tot dat prioriteitsniveau behoren.



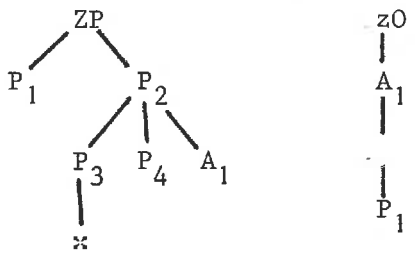
(In deze figuren schrijven we ^{ZP} voor Zelfdeprofiel P voor Profiel en A voor Append.) $Profiel_2$ behoort tot de hoogste prioriteitsklasse omdat het de ijverig gekonsumeerde variabele z_o bevat. $Profiel_2$ (de oorspronkelijke ijverige konsument) wordt geselekteerd en uitgevoerd. We krijgen nu :

←$Profiel_1$((ℓ(A)::ℓ(B))↓,z0) $Profiel_3$(ℓ(A)↓,s1) $Profiel_4$(ℓ(C)↓,t1)
$Append_1$(s1↓,t1,z0↓↑)

als nieuwe probleemstelling

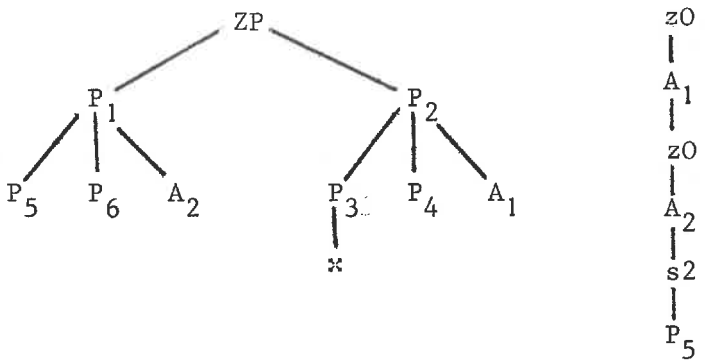


Het proces $Append_1$ dat tot de hoogste prioriteitsklasse behoort, wordt geselekteerd en verdaagd omdat het s1 niet mag binden. Volgens regel 2.a. worden afstammelingen van $Profiel_2$ die s1 bevatten aktief. $Profiel_3$ wordt geselekteerd en uitgevoerd. Dit resulteert in de binding s1 ← A.Nil. Het proces $Append_1$ wordt terug in overweging genomen maar opnieuw verdaagd omdat het z0 wil binden. Nu moet, (volgens regel 2.b.) een beroep gedaan worden op producenten van z0 die niet afstammen van $Profiel_2$.



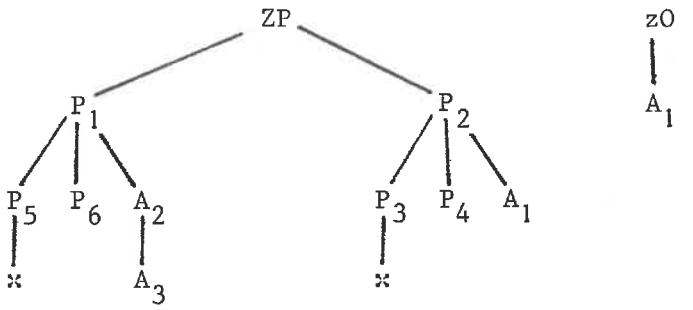
$Profiel_1$ wordt geselekteerd en uitgevoerd :

←$Profiel_5$(ℓ(A)↓,s2) $Profiel_6$(ℓ(B)↓,t2) $Append_2$(s2↓,t2,z0)
$Profiel_4$(ℓ(C)↓,t1) $Append_1$((A.Nil)↓,t1,z0↓↑)



Nu wordt het proces $Append_2$ geselekteerd (bevat z0), maar verdaagd omwille van de konsumentenbeperking op s2. $Profiel_5$ wordt aktief, de uitvoering van dit proces resulteert in de binding s2 ← A.Nil. Nu kan $Append_2$ eveneens

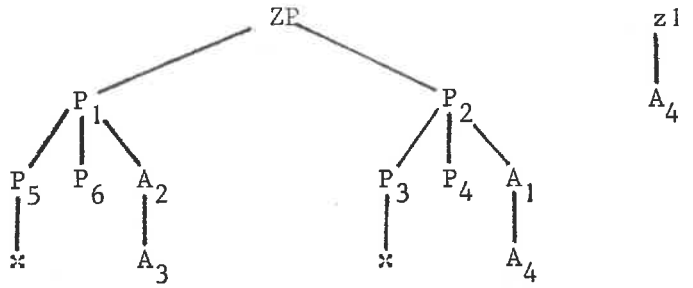
uitgevoerd worden. z0 wordt gebonden aan A.z1 en we keren terug naar de ijverige konsument. De nieuwe toestand is : (de producent heeft nu een eerste blad berekend)



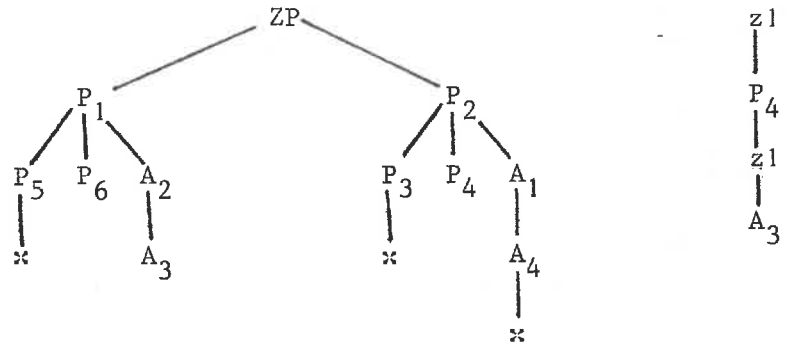
←Profiel$_6$(ℓ(B)↓,t2) Append$_3$(Nil↓,t2,z_1) Profiel$_4$(ℓ(C)↓,t1)
Append$_1$((A.Nil)↓,t1,(A.z_1)↓↑).

Append$_1$ kan nu uitgevoerd worden (controle van het eerste blad). Daar z0 gebonden werd aan A.z1 wordt z1 de nieuwe ijverig gekonsumeerde variabele. We krijgen :

←Profiel$_6$(ℓ(B)↓,t2) Append$_3$(Nil↓,t2,z1) Profiel$_4$(ℓ(C)↓,t1)
Append$_4$(Nil↓,t1,z1↓↑)



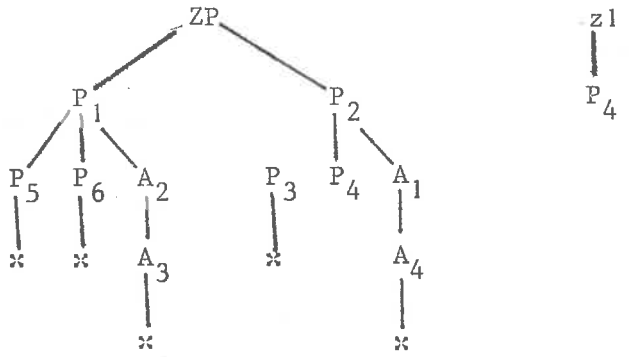
Het proces Append$_4$, de enige afstammeling van Profiel$_2$ die z1 bevat, behoort tot de hoogste prioriteitsklasse en wordt geselekteerd. De uitvoering resulteert in de binding t1 ← z1 en Profiel$_4$, eveneens afstammeling van Profiel$_2$, bevat nu de ijverig gekonsumeerde variabele



←Profiel$_6$(ℓ(B)↓,t2) Append$_3$(Nil↓,t2,z1) Profiel$_4$(ℓ(C)↓,z1↓↑)

Wegens de konsumentenbeperking wordt $Profiel_4$ verdaagd en opnieuw komen (volgens regel 2.b.) de producenten die niet van $Profiel_2$ afstammen, in de hoogste prioriteitsklasse. $Append_3$ wordt uitgevoerd, dit resulteert in t2 ← z1 en nu wordt $Profiel_6$ aktief. Dit leidt tot de binding z1 ← B.Nil (de producent heeft het tweede blad berekend)

←$Profiel_4$(ℓ(C)↓,(B.Nil)↓↑)



$Profiel_4$ kan nu uitgevoerd worden (controle van het tweede blad) maar blokkeert. Het paar <ℓ(A)⁜ℓ(B),ℓ(A)⁜ℓ(C)> behoort niet tot de relatie Zelfdeprofiel. Wat belangrijker is : we zien dat het ijverige konsumentenmechanisme de procedure-oproepen $Profiel_1$ en $Profiel_2$ toelaat hun akties te synchroniseren zodat ze om beurten een element van hun profiel berekenen.

D. NABESCHOUWINGEN

De droom om stellingen automatisch te bewijzen is voorlopig stukgesprongen op de onmogelijkheid om een goede selektiestrategie te ontwikkelen. Door zich te beperken tot meer doorzichtige logische programma's werd het inzicht in de diverse aspekten die een uitvoering efficiënt maken verruimd. Kowalski stelde [Kowalski 76] : "Algorithm = Logic + Control". Toch is het nog steeds niet erg duidelijk hoe de controlecomponent er moet uitzien; de automatische generatie ervan lijkt nog verder af. We kennen het eenvoudige controlemechanisme van PROLOG; toch mist het voor bepaalde programma's de nodige soepelheid en daarenboven wijzigt het soms de logika van de programma's. Dit maakt het onmogelijk om het programma als een

zuivere specifikatie te beschouwen en maakt eventuele resultaten over correctheid en eindigheid ongeldig.

In dit hoofdstuk hebben we een aantal concepten ontwikkeld die de logika van de programma's ongemoeid laten en die zoals uit de voorbeelden blijkt, zeer ruime mogelijkheden bieden. Tijdens de ontwikkeling van deze concepten werden wij (K.L. Clark en de auteur) vanuit diverse richtingen beïnvloed.

In een eerste benadering probeerden A.Colmerauer en de auteur om de programma's uit te breiden met informatie die moest toelaten om de prioriteit van de diverse procedure-oproepen te berekenen en de oproep met de hoogste prioriteit te selekteren. Deze informatie had een absoluut karakter, ze werd beschouwd als eigen aan elke procedure en hield dus geen rekening met de context noch met het doel van de berekening. Dit leidde tot een star en eerder onnatuurlijk systeem dat wegens het herhaald berekenen van de diverse prioriteiten, potentieel inefficiënt was.

Keith Clark en Barry Lichtman lieten zich inspireren door de "luie" evaluatiemechanismen van LISP [Henderson & Morris 76],[Friedman & Wise 76]. Deze evaluatiemechanismen waren gebaseerd op het theoretische werk over "call by need" van [Vuillemin 73],[Wadsworth 71]. Een luie evaluator voor LISP berekent (produceert) een expressie (een eerste benadering ervan) slechts wanneer ze werkelijk nodig is (op verzoek van een konsument). Het niet-determinisme in logische programma's, o.a. de onbepaaldheid van invoer en uitvoer, maakt dat een bepaalde procedure in principe zowel kan produceren als konsumeren. Daardoor stelt het gebruiken van de ideeën over luie evaluatie zware problemen.

Een botsing van ideeën alsmede contact met het werk van [Schwartz 77] en van [Kahn en McQueen 77] (dit laatste ook al geïnspireerd door "call by need") brachten ons op nieuwe wegen. We leerden de diverse procedure-oproepen beschouwen als processen die via hun variabelen informatie uitwisselen. Ons probleem bestond dan in het synchroniseren van de diverse processen en het controleren van de informatiestroom. We menen dit op een doorzichtige wijze verwezenlijkt te hebben door het opleggen van beperkingen. Bepaalde processen krijgen beperkingen opgelegd en deze worden dan consistent doorgevoerd in alle sub-processen die van het oorspronkelijke proces afstammen. Deze sub-processen krijgen daarnaast eventueel nog nieuwe lokale beperkingen opgelegd.

Dit standpunt dat processen hun akties moeten synchroniseren en dat de informatiestroom tussen de processen moet gecontroleerd worden, brengt ons dicht bij het meer praktisch gerichte werk over "concurrent programming" van o.a. [Hansen 78] en [Hoare 78]. Zoals uit onze voorbeelden blijkt, hebben bepaalde door hen opgeloste problemen een verrassend eenvoudig logisch equivalent. Zich informeel van de correctheid van deze logische programma's overtuigen is niet moeilijk. Het kan gebeuren op grond van de logische specifikatie, los van elke beschouwing over parallellisme en we kunnen stellen dat dit beduidend eenvoudiger is dan met de oorspronkelijke programma's. Onder ander het werk van [Clark & Tarnlund 77] suggereert dat formele bewijzen tot de mogelijkheden behoren. Zich ervan overtuigen dat het gecontroleerde logische programma het gewenste gedrag simuleert is eveneens vrij gemakkelijk. Wij durven beweren dat onze gecontroleerde logische programma's een hogere graad van vertrouwen toelaten dan hun tegenhangers in meer conventionele talen voor parallelle programmatie. Natuurlijk kennen wij voorlopig alleen de pseudo-parallelle uitvoering waarbij unifikaties als ondeelbare primitieve operaties beschouwd worden. Echt parallellisme is een nog onontgonnen terrein, het vereist dat verschillende unifikaties gelijktijdig gebeuren. Dit maakt het noodzakelijk om unifikatie op te splitsen in meer primitieve operaties. Bijvoorbeeld, het binden van een variabele, een operatie tijdens dewelke het proces een exclusieve toegang tot de variabele zou hebben. De gebruikelijke sequentiële exploratietechniek die de diverse unifikaties als een totaal geordende verzameling beschouwt, wordt eveneens ongeschikt en technieken zoals beschreven in het volgende hoofdstuk worden onontbeerlijk.

Is echt parallellisme nog veraf, de hoge graad van vertrouwen in onze gecontroleerde logische programma's maakt het wellicht interessant om na te gaan of een deelklasse van onze logische programma's kan omgezet worden, met behoud van de correctheidseigenschappen, in een meer conventioneel formalisme waarvoor parallelle uitvoeringsmechanismen beschikbaar zijn.

We spreken wel over een deelklasse; inderdaad, conceptueel is de communicatie tussen onze logische processen veel ruimer dan in meer conventionele benaderingen. Een informatiekanaal (lees variabele) kan gedeeld worden door een onbeperkt en dynamisch veranderend aantal processen en de informatie kan in verschillende richtingen stromen. Ook is de informatiestroom niet

beperkt tot een rij van elementen (een lijststruktuur) maar kan gelijk welke vorm aannemen (vb. boomstruktuur). Ook het niet-determinisme waarbij verschillende definities voor eenzelfde oproep kunnen gebruikt worden en dus verschillende alternatieve oplossingen moeten nagegaan worden, heeft geen direct equivalent.

HOOFDSTUK 4 : "INTELLIGENTE" SEQUENTIELE EXPLORATIE

A. INLEIDING

Nemen we een voorwerp vast met onze rechterhand en merken we dat het te warm is, dan zullen we geen tweede poging wagen met de linkerhand. Een logisch programma uitgevoerd door een vertolker die de sequentiële exploratietechniek gebruikt zal dit wel overwegen. Daaruit, zoals [Sussman & McDermott 72], besluiten dat de sequentiële exploratietechniek ongeschikt is, is voorbarig. De problemen ontstaan omdat de sequentiële exploratietechniek de informatie waaruit een toestand opgebouwd is, als een ondeelbaar geheel beschouwt. Is de huidige toestand onaanvaardbaar, dan wordt heel systematisch een andere toestand opgebouwd. Het voorwerp vastnemen met de linkerhand is een andere toestand en komt dus in aanmerking. Om de sequentiële exploratietechniek te verbeteren moeten we in staat zijn om die globale informatie op te splitsen en voor elk deel een bron aan te wijzen. In ons voorbeeld kan de informatie opgesplitst worden in :

- de hoge temperatuur van het voorwerp
- het vastnemen van het voorwerp door middel van een hand
- het gebruik van de rechterhand

De conflictsituatie waarbij we onze rechterhand dreigen te verbranden, kan nu teruggebracht worden tot de hoge temperatuur van het voorwerp en het gebruik van een hand om het voorwerp vast te nemen. Om het conflict op te lossen moet de toestand zo gewijzigd worden dat één van de bronnen die de conflictsituatie veroorzaakt, andere informatie genereert. Een wijziging van de derde informatiebron die besliste om de rechterhand te gebruiken is dus niet relevant en zal niet overwogen worden.

In dit hoofdstuk willen we een methode ontwikkelen die zich op dergelijke "intelligente" wijze gedraagt. Natuurlijk zal de analyse van conflictsituaties niet semantisch maar zuiver syntaktisch zijn. Het sukses zal dus gedeeltelijk afhangen van de gebruikte gegevensstrukturen en ook wel van de orde waarin de verschillende oproepen geselekteerd worden.

B. PROCESSEN ALS INFORMATIEBRONNEN

B.1. EEN ENKEL SEQUENTIEEL PROCES

In het inleidend hoofdstuk hebben we de uitvoering van een logisch programma beschouwd als de sequentiële exploratie van een OF-boom. Elk knooppunt bevat een probleemstelling bestaande uit een aantal procedure-oproepen. Uit deze procedure-oproepen wordt er één geselekteerd en uitgevoerd. Het aantal takken dat uit het knooppunt vertrekt, is gelijk aan het aantal proceduredefinities dat met de geselekteerde oproep overeenstemt (unifikatie). De grootte van de OF-boom en dus de efficiëntie van de uitvoering is erg afhankelijk van de volgorde waarin de oproepen geselekteerd worden. Juist daarom hebben we zoveel aandacht besteed aan het controleren van de selektie.

Typisch voor deze aanpak is dat de staat van de berekening globaal beschouwd wordt. Wanneer de berekening in een dood punt komt (Fig. 4.B.1) worden geen vragen gesteld over de oorzaak daarvan. De "naïeve" reaktie is dat de toestand moet gewijzigd worden en dat dus een andere tak van de OF-boom moet doorlopen worden. Daarom zullen we deze methode in het vervolg de "naïeve sequentiële exploratietechniek" noemen.

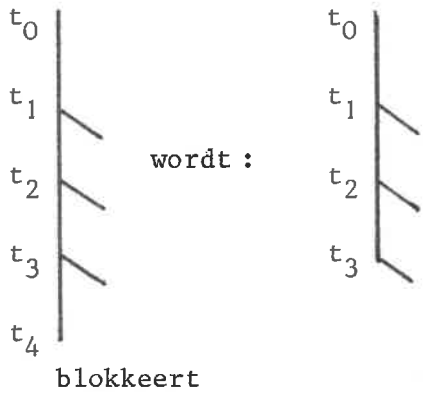


Fig. 4.B.1. : Het proces staat in toestand t_0. In de toestanden t_1, t_2 en t_3 heeft het proces keuze tussen verschillende alternatieven. Het proces blokkeert in toestand t_4, keert terug naar toestand t_3 en probeert een ander alternatief.

B.2. ONAFHANKELIJKE PARALLELLE PROCESSEN

Zijn we in staat om dat ene proces op te splitsen in onafhankelijke processen, dan kunnen deze processen hun taak gelijktijdig uitvoeren. Elk proces kan werken in zijn eigen lokale omgeving. Blokkeert een proces, dan moet het een vroegere toestand herstellen en een alternatieve oplossing proberen. Dit heeft echter geen enkele invloed op de andere processen (Fig. 4.B.2).

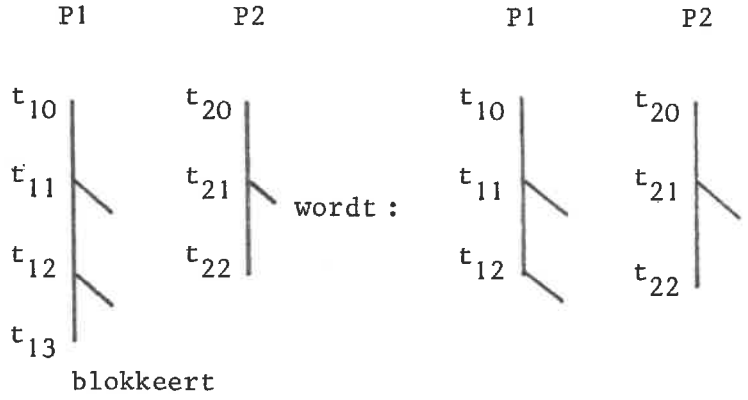


Fig. 4.B.2. : Twee onafhankelijke processen P1 en P2. Proces P1 blokkeert in toestand t_{13}. Het neemt een nieuwe start vanaf toestand t_{12}. Dit heeft geen enkele invloed op proces P2.

De verschillende procedure-oproepen in een logisch programma zouden we kunnen beschouwen als zelfstandige taken. Met elke taak zouden we dan een onafhankelijk proces associëren. Het is echter weinig waarschijnlijk dat we een oplossing van het totale probleem bekomen wanneer we de oplossingen samenvoegen die de diverse processen berekend hebben. Inderdaad, meestal bevatten de verschillende procedure-oproepen gemeenschappelijke variabelen. Het is weinig waarschijnlijk dat de bindingen, die door verschillende processen voor eenzelfde variabele berekend werden, met elkaar in overeenstemming te brengen zijn. Wellicht zal het ene proces bijvoorbeeld x binden aan A terwijl een ander proces dezelfde x aan B bindt.

Elk proces alle oplossingen laten berekenen en dan uit de verschillende verzamelingen oplossingen een globale oplossing distilleren, brengt ons van de regen in de drop : we eindigen met een nieuw combinatorisch probleem dat moeilijker kan zijn dan het oorspronkelijke. Trouwens, het is goed mogelijk dat bepaalde deelproblemen een oneindig aantal oplossingen hebben.

De opsplitsing in onafhankelijke processen is dus niet algemeen houdbaar.

B.3. INFORMATIESTROMEN TUSSEN PROCESSEN

Een zekere uitwisseling van informatie tussen de diverse processen is dus onafwendbaar. Meteen is het ook onvermijdelijk dat het blokkeren van een proces zijn weerslag heeft op andere processen. Deze wederzijdse beïnvloeding wordt geïllustreerd in fig. 4.B.3 en fig. 4.B.4.

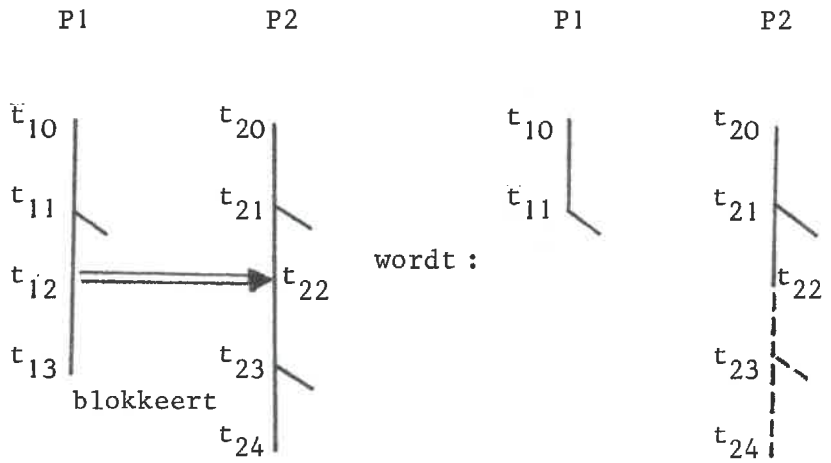


Fig. 4.B.3. : In toestand t_{22} maakt proces P2 gebruik van informatie afkomstig uit toestand t_{12} van proces P1. Proces P1 blokkeert (in t_{13}) en keert terug naar toestand t_{11}. De door proces P2 gebruikte informatie is niet langer beschikbaar en de berekening vanaf t_{22} is niet langer gerechtvaardigd en wordt in vraag gesteld.

In fig. 4.B.3 zien we dat het blokkeren van proces P1 een deel van de door proces P2 uitgevoerde berekeningen in vraag stelt. Fig. 4.B.4 illustreert een conflict tussen de informatie afkomstig van twee verschillende processen en toont het ontstaan van een geïnduceerde informatiestroom.

Hoe meer informatie de processen uitwisselen, hoe groter de weerslag van het blokkeren van een proces. Het is dus voordelig om het aantal informatiestromen zo klein mogelijk te houden. Anderzijds is het noodzakelijk dat de globale oplossing consistent blijft (=dat ze geen tegenstrijdige bindingen bevat.) Van zodra een tegenstrijdige binding dreigt te ontstaan is een informatie-uitwisseling onvermijdelijk.

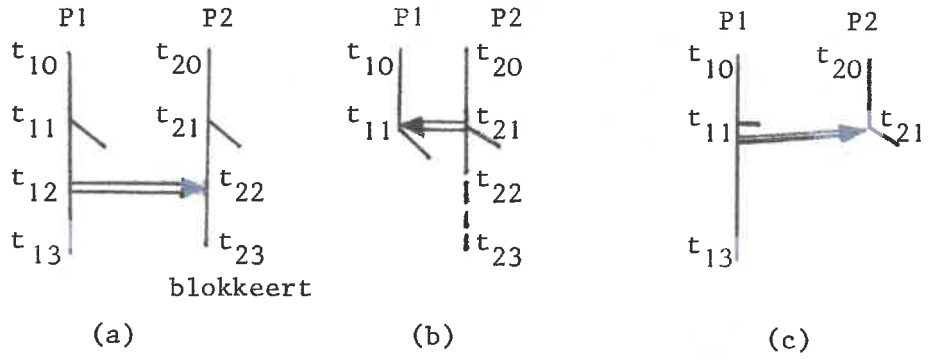


Fig. 4.B.4. : In toestand t_{22} betrekt proces P2 informatie uit toestand t_{12} van proces P1. In toestand t_{23} blokkeert proces P2. Dit wil zeggen dat er een conflict is tussen de informatie van het eigen proces en de informatie van proces P1 : het in t_{21} gekozen alternatief is onverenigbaar met het in t_{11} gekozen alternatief. Als gevolg hiervan kunnen we de gelijktijdigheid van de processen in de punten t_{11} en t_{21} niet langer handhaven. Er zijn twee mogelijkheden. Ofwel gebeurt de keuze in t_{11} na de keuze in t_{21} (fig. b) en de in t_{21} gemaakte keuze schakelt het tot nu toe in t_{11} gebruikte alternatief uit (een informatiestroom van P2 naar P1). Ofwel gebeurt de keuze in t_{11} voor de keuze in t_{21} (fig. c) en de in t_{11} gemaakte keuze schakelt het tot nu toe in t_{21} gebruikte alternatief uit (een informatiestroom in P1 naar P2). In beide gevallen wordt er dus een informatiestroom geïnduceerd.

B.4. DE UITVOERING VAN EEN PROCES

We hebben processen geassocieerd met procedure-oproepen. De uitvoering van een proces bestaat dan uit drie elementen :

1. het oppikken van een proceduredefinitie
2. de unifikatie tussen oproep en hoofding van de definitie. Het proces is de informatiebron voor de resulterende bindingen.
3. creatie van de "zonen" : een subproces voor elke oproep in het lichaam van de proceduredefinitie.

De subprocessen ontvangen onvermijdelijk informatie van het proces dat ze creëert. De absoluut minimale informatiestroom is dus gegeven door de struktuur van de EN-boom die een hiërarchie van processen bepaalt.

Het uitvoeren van een proces kan dus beschouwd worden als één enkele primitieve operatie die op een bepaald tijdstip plaats heeft.

Wanneer de gegenereerde informatie strijdig is met andere reeds aanwezige informatie ontstaat er een conflict. Verder komen we op deze conflictsituatie terug.

B.5. AFHANKELIJKHEIDSRELATIE - AFHANKELIJKHEIDSGRAF

De informatiestromen definiëren een relatie tussen de processen. We zeggen dat proces i van proces j afhangt wanneer er een informatiestroom is van proces j naar proces i. Deze relatie is natuurlijk transitief. Ze houdt ook in dat proces i na proces j uitgevoerd wordt. De relatie definieert dus een partiële orde over de tijdstippen t_i waarop de diverse processen P_i kunnen uitgevoerd worden.

Een proces P_i is direct afhankelijk van een proces P_j wanneer er geen proces P_k is zodat P_i van P_k afhangt en P_k van P_j afhangt. Wegens de transitiviteit bevat de directe afhankelijkheidsrelatie alle informatie die nodig is om de volledige afhankelijkheidsrelatie te bepalen. De directe afhankelijkheidsrelatie definieert een afhankelijkheidsgraf over de processen. Deze graf wordt gradueel opgebouwd en uitgebreid naarmate de uitvoering van het logisch programma vordert. In de volgende sekties van dit hoofdstuk zullen we hoofdzakelijk aantonen hoe deze afhankelijkheidsgraf wordt opgesteld en hoe hij gebruikt wordt om op een intelligente wijze blokkeringen op te heffen.

B.6. EEN CONFLICT

Een conflict ontstaat tussen de processen $P_1,\ldots,P_n$ wanneer uit de informatie die deze processen gegenereerd hebben een tegenstrijdigheid kan afgeleid worden (eenzelfde variabele die aan twee verschillende termen gebonden is of een proces P_q dat tot die informatie toegang heeft en met geen enkele proceduredefinitie overeenstemt.) We lossen het conflict op door een proces P_i (de "schuldige") te kiezen en, op grond van de informatie afkomstig van de andere processen $(P_1,\ldots,P_{i-1},P_{i+1},\ldots,P_n)$ te besluiten dat de door P_i gebruikte proceduredefinitie niet geschikt is. Dit houdt in dat er informatiestromen ontstaan van de processen $P_1,\ldots,P_{i-1},P_{i+1},\ldots,P_n$ naar het "schuldige" proces P_i. Gezien de door de informatiestromen gedefinieerde afhankelijkheidsrelatie een partiële ordening moet blijven, wordt de keuze van P_i beperkt : we moeten een proces kiezen dat geen informatie levert aan de andere processen. Uit de verzameling $P_1,\ldots,P_n$ komen alleen de processen in aanmerking die "laatst" komen in de partiële ordening die tussen deze processen bestaat. De berekening die het schuldige proces gemaakt heeft, wordt ongedaan gemaakt.

Beschikt het schuldige proces P_i over geen alternatieve proceduredefinities, dan is het conflict nog niet opgelost, het is alleen maar verschoven. Dit betekent dat proces P_i zijn deeltaak niet kan oplossen, dat er een conflict is tussen de processen die aan P_i informatie leveren. Wegens de transitiviteit van de informatiestromen zijn dit de processen $P_1,\ldots,P_{i-1},P_{i+1},\ldots,P_n$. Tussen deze processen moeten wij een nieuwe schuldige zoeken.

We krijgen een beter beeld van de gevolgen van onze keuze van P_i wanneer we uit de verzameling $P_1,\ldots,P_n$ (alle processen die direct of indirect bijdragen tot het ontstaan van het oorspronkelijke conflict) die processen verwijderen die over geen alternatieve proceduredefinities beschikken. Uit de overblijvende processen (noemen we ze $P_1,\ldots,P_m$) kunnen we kiezen tussen die processen die in de partiële ordening als laatsten komen.

Die keuze is erg belangrijk. Tot op dat ogenblik liet de partiële ordening toe om de "laatste" processen van de verzameling $P_1,\ldots,P_m$ als gelijktijdig te beschouwen. Nu moeten we een proces P_i kiezen en de geïnduceerde informatiestromen hebben voor gevolg dat P_i na de andere processen $P_1,\ldots,P_{i-1},P_{i+1},\ldots,P_m$ moet uitgevoerd worden. Het oplossen van conflicten heeft dus voor gevolg dat de gelijktijdigheid ingeperkt wordt.

De keuze van de schuldige beïnvloedt de efficiëntie van de sequentiële exploratie. Die keuze wordt zolang mogelijk uitgesteld en hoe langer men kan wachten, hoe meer informatie over de ontwikkeling van de sequentiële exploratie er beschikbaar is. Bij de naïeve sequentiële exploratietechniek worden de processen bij hun ontstaan reeds totaal geordend. Dit heeft twee grote nadelen :

(1) Op dat ogenblik beschikt men over minder informatie over de wijze waarop de sequentiële exploratie zich zal ontwikkelen.

(2) Men kan geen gebruik maken van het feit dat bepaalde deeltaken gedeeltelijk of totaal onafhankelijk zijn.

De hier ontwikkelde methode laat dus niet alleen toe om de tijdelijke of totale onafhankelijkheid van de deeltaken te benutten, maar ook om onvermijdelijke totale ordeningen tussen deeltaken zo lang mogelijk uit te stellen. Dit laatste is erg belangwekkend voor systemen waar elke gebruikerscontrole ontbreekt (vb. automatische stellingbewijzers) en het systeem alle beslissingen zelf moet treffen. De grotere hoeveelheid informatie laat wellicht toe om betere keuzen te maken.

Wij willen dit hier niet verder uitwerken. Om logische programma's efficiënt te kunnen uitvoeren, achten wij gebruikerscontrole noodzakelijk. Bij het kiezen van het "schuldige" proces laten wij ons door die controle leiden en kiezen het proces dat volgens de geldende selektiestrategie het laatst geselekteerd werd. Om de verdere uiteenzetting eenvoudig te houden zullen we PROLOG's eenvoudige selektiestrategie gebruiken en aannemen dat de processen van links naar rechts geselekteerd worden. Tussen de kandidaten $P_1,\ldots,P_m$ wordt dus het proces P_i, dat zich het meest rechts in de EN-boom bevindt, als schuldige aangewezen. (De partiële ordening die zich ontwikkelt, laat dus steeds de totale ordening van de naïeve sequentiële exploratie toe.)

Deze keuze zal niet altijd de beste zijn, maar wanneer de controle, die de gebruiker oplegt, behoorlijk is, kunnen we verwachten dat ook deze keuze behoorlijk is.

Er kunnen nog andere keuzemogelijkheden ontstaan. Dezelfde informatie kan in verschillende processen beschikbaar zijn. Speelt deze informatie mee in de opbouw van een conflict, dan is het voldoende dat slechts één van die processen tot het conflict bijdraagt. Ook hier zullen wij eenvoudigheidshalve de oudste (eerst geselekteerde) producent als echte producent beschouwen. Ook kan het gebeuren dat het uitvoeren van een proces aanleiding geeft tot conflicten in meerdere variabelen. In dit geval zullen we ons beperken tot het conflict in de variabele met de oudste producent. (Ook in deze gevallen zou men kunnen onderzoeken welke keuze het voordeligst is voor de efficiëntie van de sequentiële exploratie.)

B.7. HET BEPERKEN VAN HET AANTAL PROCESSEN

Wanneer een proces slechts van één ander proces direct afhankelijk is, en daarenboven over geen alternatieve procedures beschikt, heeft het weinig zin om dat proces als zelfstandig te beschouwen. Het kan alleen maar de zaak ingewikkelder maken. We laten het proces opslorpen door het proces waarvan het direct afhankelijk is. De informatie die bij het proces hoorde, wordt bij het opslorpend proces ondergebracht. Eenzelfde proces kan dan met verschillende procedure-oproepen geassocieerd zijn. Bij het uitvoeren van een volledig deterministisch programma worden zelfs alle processen door het initiële proces opgeslorpt. Inderdaad, elk nieuw proces wordt, daar het deterministisch is, door het ene bestaande proces opgeslorpt.

B.8. EEN FAMILIE VAN ALGORITMES

Tot nu toe hebben we in het midden gelaten welke informatiestromen er precies optreden. We hebben wel geëist dat de globale oplossing moet consistent blijven en we hebben opgemerkt dat het voordelig lijkt om het aantal informatiestromen zo klein mogelijk te houden. Maar ook andere beschouwingen zoals eenvoud van ontwerp, het soort toepassingen (men moet niet meer "intelligentie" inbouwen dan nodig is),... spelen een rol.

De toch wel eenvoudige principes die we geformuleerd hebben kunnen aanleiding geven tot een ganse waaier van verwezenlijkingen.

Zelfs de naïeve sequentiële exploratietechniek past in dit schema. Wanneer we aannemen dat elk proces informatie ontvangt van alle reeds uitgevoerde processen, dan is elk proces alleen direct afhankelijk van het vorige en krijgen we een totale ordening. Dit resulteert in hetzelfde gedrag als bij de naïeve sequentiële exploratietechniek.

Een vrij eenvoudige mogelijkheid is als volgt : telkens een proces een binding genereert die strijdig is met reeds aanwezige informatie, ontvangt het een informatiestroom die uitgaat van de producent van die informatie. We nemen meteen aan dat het proces toegang heeft tot en gebruik maakt van alle informatie waarover deze producent beschikt. De informatiestromen gaan dus gepaard met een totale informatie-overdracht. Het herzien van een proces heeft dan onvermijdelijk tot gevolg dat alle processen die van dit proces afhangen volledig opnieuw moeten uitgevoerd worden.

Eerst geven we een algoritme en daarna zullen we, aan de hand van enkele voorbeelden, nagaan hoe de sequentiële exploratie zich met deze methode gedraagt. Later zullen we de methode nog wat verfijnen.

B.9. ALGORITME 1

Herhaal

1. Selekteer een proces (oproep).
2. Voer dit proces uit. Het proces beschikt over alle informatie (bindingen) uit processen waarvan dit proces afhangt.

3.a. Indien de informatie waarover het proces beschikt zodanig is dat geen enkele proceduredefinitie ermee overeenstemt, dan is er een conflict tussen alle processen waarvan het geselekteerde proces afhangt.

3.b. Indien de globale informatie inconsistent is (de nieuwe bindingen zijn in strijd met bindingen uit processen waarvan het pas uitgevoerde proces niet afhangt) dan is er een conflict tussen :
- het uitgevoerde proces en alle processen waarvan het afhangt,
- en alle processen die bijdragen tot de vroeger gegenereerde strijdige informatie.

4. Indien geen conflict dan verder gaan, anders :
   a. In de verzameling V van processen die tot het conflict bijdragen, bepalen we de deelverzameling D (de "verdachten") bestaande uit alle processen behorend tot V waarvan geen enkel proces behorend tot V afhangt (de "laatsten")
   b. In de verzameling D (= $P_1,\ldots,P_m$) kiest men een schuldige P_i.
   c. Men voegt nieuwe informatiestromen toe : het schuldig proces P_i hangt af van de andere processen uit de verzameling D (wegens de transitiviteit ook van alle processen uit V).
   d. Berekeningen gemaakt door processen die van het schuldige proces afhangen, worden ongedaan gemaakt. Al deze processen moeten volledig opnieuw uitgevoerd worden en beschikken terug over een volledige lijst van proceduredefinities (dat een proces opnieuw moet uitgevoerd worden, betekent dat zijn afstammelingen volledig verdwijnen.)
   e. De berekening die het schuldige proces gemaakt heeft, wordt ongedaan gemaakt en de proceduredefinitie die het gebruikt heeft, wordt uitgeschakeld.

tot dat een oplossing gevonden is.

Opmerkingen - Optimalisaties

1. In geval 3.b behoort het pas uitgevoerde proces tot de verdachten. Meestal zal het ook als schuldige aangewezen worden en zal het in de volgende iteratie opnieuw als uit te voeren proces gekozen worden. Dit proberen van een definitie, ontdekken van een conflict, toevoegen van nieuwe informatiestromen, proberen van een volgende definitie ... tot dat een geschikte definitie gevonden is of de definities uitgeput zijn, zullen we verder als één stap beschouwen.

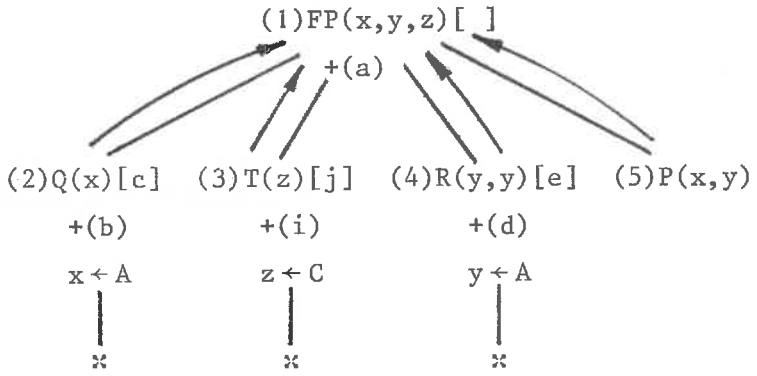
2. Wordt in de iteratie die op een conflict volgt het schuldige proces geselekteerd en heeft dat proces reeds alle definities geprobeerd, dan eindigt de nieuwe iteratie in een conflict van type 3.a. Ook dit kan men voorzien en men kan onmiddellijk op zoek gaan naar een definitieve schuldige (een proces dat over alternatieven beschikt). Tevens kan daardoor het aantal geïnduceerde informatiestromen iets kleiner zijn en moet men eventueel minder processen opnieuw uitvoeren (door zo snel mogelijk de definitieve schuldige aan te wijzen - zie sektie B.6.)
3. Beschikt een proces over geen alternatieve proceduredefinities en is het slechts van één enkel proces direct afhankelijk, dan kan het door dat proces opgeslorpt worden (sektie B.7).
4. Voor het bepalen van de verdachten (stap 4.a) is het vrij eenvoudig om van een kleinere verzameling te vertrekken :
   stap 3.a. : alleen de processen waarvan het geselekteerde proces direct afhangt.
   stap 3.b. : alleen het uitgevoerde proces en de processen die direct betrokken zijn bij de generatie van de tegenstrijdige bindingen.

B.10. EEN EENVOUDIG VOORBEELD

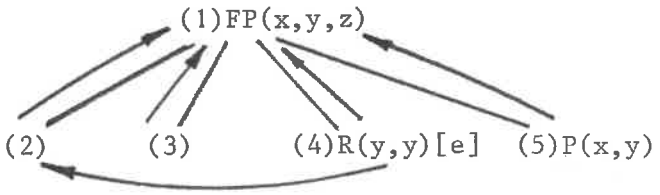
←FP(x,y,z)

a. FP(x,y,z) ← Q(x),T(z),R(y,y),P(x,y)
b. Q(A) ←
c. Q(B) ←
d. R(A,A) ←
e. R(x,y) ← SR(x,y)
f. SR(B,A) ←
g. P(B,A) ←
h. P(A,B) ←
i. T(C) ←
j. T(D) ←

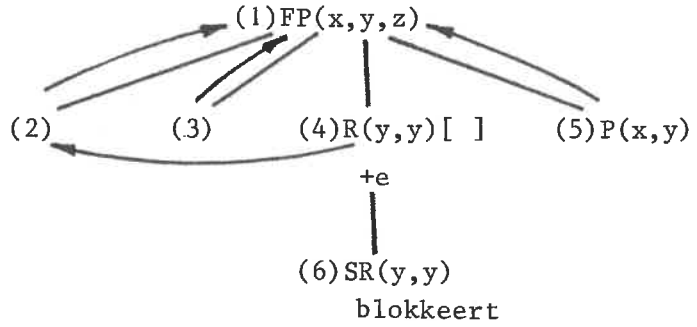
Zoals hoger vermeld nemen we aan dat de selektie der oproepen van links naar rechts gebeurd. Het initiële proces FP creëert 4 subprocessen. Nadat Q, T en R uitgevoerd zijn, hebben we :



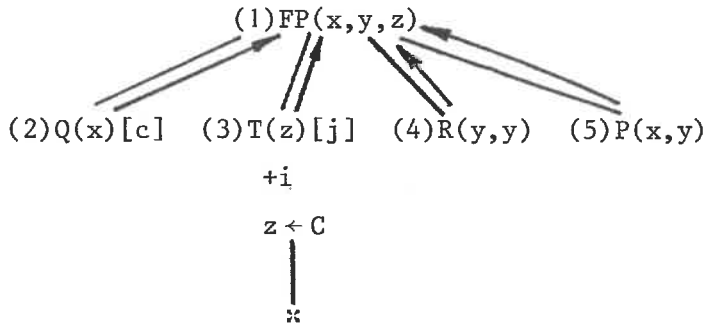
De pijlen duiden het afhankelijkheidsgraf aan. De lijsten [...] duiden aan welke alternatieve procedures er beschikbaar zijn. We zien dat proces (2) de binding x ← A genereert, proces (3) genereert z ← C en (4) genereert y ← A. Proces (1) genereert geen enkele binding. Nu moet proces (5): P(x,y) uitgevoerd worden. De procedure P(B,A) veroorzaakt een conflict met de binding x ← A afkomstig van proces (2). De procedure P(A,B) veroorzaakt een conflict met proces (4) (één stap - optimalisatie 1). Het proces P(x,y) hangt dus niet alleen af van (1) maar ook van (2) en (4) en heeft geen enkele oplossing. De informatie gecreëerd door de processen (1) (2) en (4) is onverenigbaar en één van die processen moet gewijzigd worden. Daar (2) en (4) van (1) afhangen komt (1) niet in aanmerking ((2) en (4) zijn de verdachten). We kiezen (4) (ook (2) is mogelijk) en de procedure (d) wordt afgewezen tengevolge van de informatie afkomstig van (1) en (2). Een informatiestroom tussen (2) (de andere verdachte) en (4) (de schuldige) wordt geïnduceerd.
Dit leidt tot :



Na verwijdering van de afhankelijkheid die omwille van de transitiviteit redundant is, krijgen we :



Nu blokkeert het proces (6). Dus ook procedure e van proces (4) wordt uitgeschakeld. Daar geen alternatieve procedure beschikbaar is, blokkeert proces (4) en moet proces (2) hernomen worden.



We zien dat de berekening verricht door proces (3) behouden blijft en dat proces (2) moet hernomen worden, alhoewel proces (2) (Q(x)) geen variabelen gemeenschappelijk heeft met het blokkerende proces R(y,y). De informatiestroom tussen die twee processen is een gevolg van het vroeger blokkeren van het proces P(x,y).

We willen hiermee illustreren dat elke poging om de sequentiële exploratietechniek te verbeteren, die geen rekening houdt met de geschiedenis van die berekening, in het bijzonder met vroegere blokkeringen, en enkel de huidige geblokkeerde toestand bestudeert (EN-boom met de substituties), tot mislukken gedoemd is. Het enige wat men kan bereiken is terugkeren tot het meest recente proces dat een variabele gebonden heeft. In dit voorbeeld is dit proces (3). Het hernemen van proces (3) neemt de oorzaak van het mislukken echter niet weg. Kijkt men alleen naar de EN-boom dan is er geen verschil tussen proces (2) en (3), beiden binden een variabele die in het geblokkeerde proces R(y,y) niet voorkomt. Nochtans leidt het herzien van proces (2) tot een oplossing terwijl het hernemen van proces (3) leidt tot een volkomen analoge mislukking !

C. VOORBEELDEN

C.1. KONINGINNENPROBLEEM - OPLOSSING MET PERMUTATIE

In een eerste voorbeeld bestuderen we het koninginnenprobleem zoals het gespecifieerd is in het vorige hoofdstuk. Nu echter nemen we aan dat steeds de linker oproep geselekteerd wordt. Dus wordt eerst een volledige opstelling, met een koningin in elke rij en een koningin in elke kolom, gegenereerd en pas daarna wordt nagegaan of er al dan niet twee koninginnen op dezelfde diagonaal staan. Met de naïeve sequentiële exploratiemethode gedraagt dit programma zich zeer slecht. Inderdaad, wordt een conflict gevonden, dan wordt systematisch een andere opstelling gegenereerd door het verplaatsen van de laatste, zo dit niet langer mogelijk is, van de voorlaatste, of ..., koningin. Was het conflict echter te wijten aan de posities van de eerste en de tweede koningin, dan betekent dit dat er (n-2) fakulteit opstellingen gegenereerd en gecontroleerd worden vooraleer de tweede koningin eindelijk verplaatst wordt en de oorzaak van het conflict opgeheven wordt.

Zoals we zullen zien zal de "intelligente" sequentiële exploratiemethode bij het ontdekken van een conflict tussen de i-de en de j-de koningin ($i<j$) onmiddellijk reageren met het verplaatsen van de j-de koningin. Om dit "intelligent" gedrag maximaal te benutten zullen we de procedure die de testen genereert lichtjes wijzigen. We dragen er zorg voor dat de testen uitgevoerd worden in de orde (1,2)(1,3)(2,3)(1,4)(2,4)(3,4)... in plaats van de orde (1,2),(1,3),(1,4),...,(2,3),(2,4)... . We zullen dus elke koningin vergelijken met haar voorgangers in plaats van ze te vergelijken met haar opvolgers. Dit zorgt ervoor dat het conflict tusen i en j ($i<j$) met j zo klein mogelijk, eerst ontdekt wordt en dat het aantal opstellingen dat uit de oplossingsruimte verdwijnt ((n-j)fakulteit) zo groot mogelijk is.

We zullen het programma iets vereenvoudigen door aan te nemen dat de lijst der rijnummers gegeven is. Tevens zullen we ons beperken tot een bord met dimensie 4.

Het programma

```
←Koninginnen(1.2.3.4.Nil,bord)
a. Koninginnen(rij,bord) ← Permutatie(rij,kolom)
                           Opstelling(rij,kolom,bord)
                           Controle(bord)
b. Permutatie(Nil,Nil) ←
c. Permutatie(x.y,u.v) ← Verwijder(u,x.y,w)
                         Permutatie(w,v)
d. Verwijder(x,x.y,y) ←
e. Verwijder(u,x.y,x.z) ← Verwijder(u,y,z)
f. Opstelling(Nil,Nil,Nil) ←
g. Opstelling(x.y,u.v,p(x,u).r) ← Opstelling(y,v,r)
h. Controle(bord) ← K(Nil,bord)
i. K(voor,Nil) ←
j. K(voor,p.q) ← Voorgangers(voor,p) K(p.voor,q)
k. Voorgangers(Nil,p) ←
1. Voorgangers(q.r,p) ← Voorgangers(r,p) Diagonaal(q,p)
```

De procedures voor Diagonaal bestaan uit een opsomming van paren die niet op dezelfde diagonaal staan.

Uitvoering

De uitvoering wordt geïllustreerd in fig. 4.C.1. Het initiële proces Koninginnen (O) creëert een proces Permutatie (afgekort P), een proces Opstelling (afgekort O) en een proces Controle (afgekort Co).

Slechts één procedure stemt overeen met het proces P en dit proces kan dus door het initiële proces (O) opgeslorpt worden.

Het proces Vr heeft twee passende procedures ter zijner beschikking. Opslorping is dus niet mogelijk en het wordt een zelfstandig proces (1). Het proces $P(w_0,v_0)$ beschikt over de procedures (b) en (c). Nochtans, de bindingen gegenereerd door toepassing van (b) zijn niet verenigbaar met de informatie afkomstig van proces (1). Het conflict induceert een informatiestroom vanuit proces (1). Deze informatiestroom wijzigt $P(w_0,v_0)$ tot $P(2.3.4.Nil,v_0)$ en elimineert procedure (b).

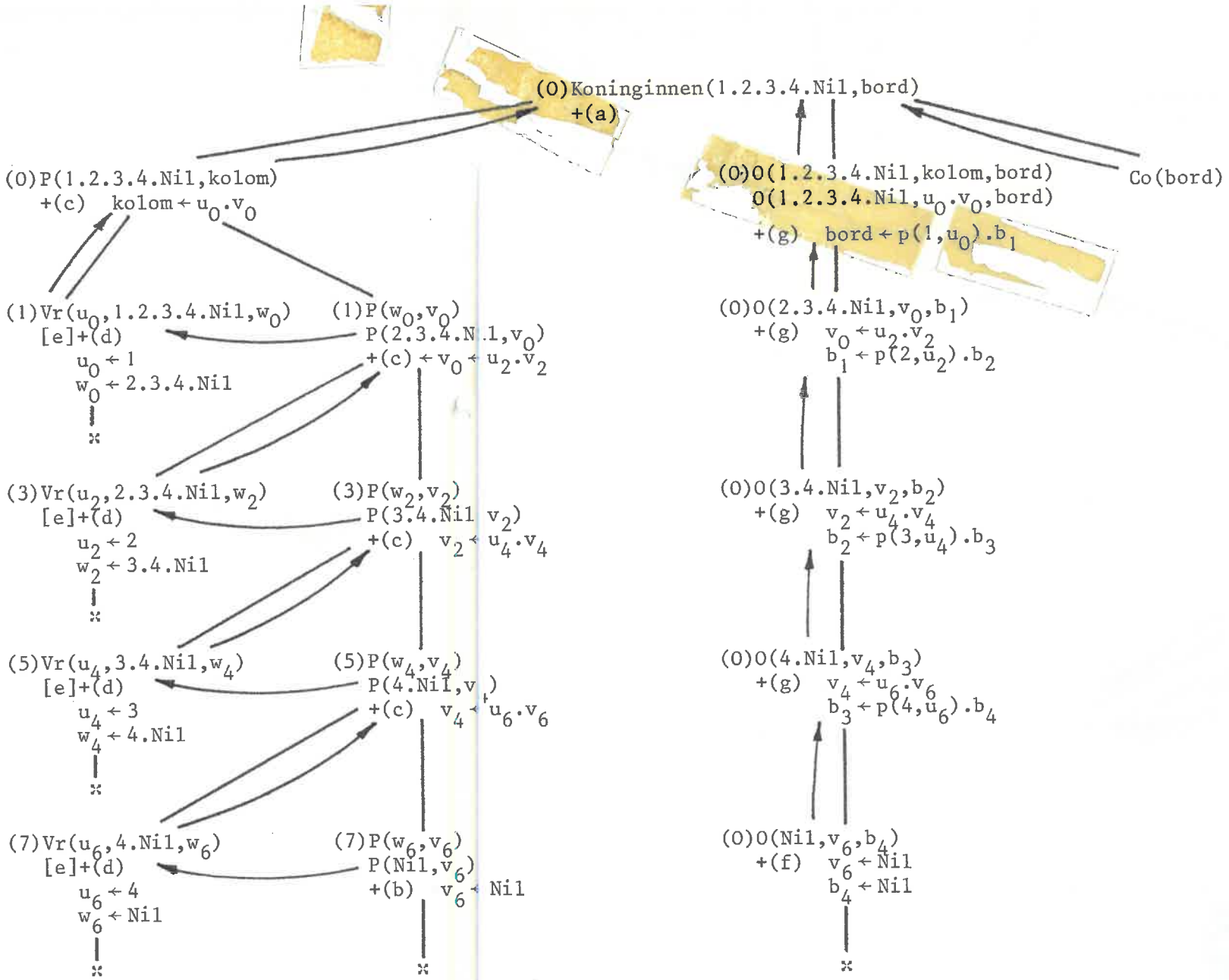


Fig. 4.C.1. : Uitvoering van de oproepen P en O. De zelfstandige informatiebronnen zijn (0), (1), (3), (5) en (7).

De informatiestroom van proces (0) naar $P(w_0,v_0)$ wordt geïnduceerd door de informatiestromen van (0) naar (1) en van (1) naar $P(w_0,v_0)$. $P(w_0,v_0)$ ontvangt dus alleen van proces (1) rechtstreekse informatie, daarenboven heeft het geen alternatieve procedures ter zijner beschikking. Het is dus geen zelfstandige informatiebron en wordt door (1) opgeslorpt. De verdere berekening van de permutatie verloopt analoog.

De informatie beschikbaar in bron (0) volstaat opdat de oproep Opstelling met slechts één procedure zou overeenstemmen. Dit proces wordt dus opgeslorpt door proces (0). Hetzelfde geldt voor alle afstammelingen. Bemerk dat proces (0) nu bindingen genereert voor de variabelen v_0, v_2, v_4 en v_6 die ook reeds gegenereerd werden door de bronnen (1) (3) (5) en (7). Daar deze laatsten tot (0) toegang hebben wordt hun informatie redundant. Juist omdat Opstelling de informatie over v_0, v_2, v_4 en v_6 niet overneemt, maar zelfstandig berekent is het mogelijk om Opstelling door (0) te laten opslorpen en deze berekening eens en voorgoed uit te voeren. (Strikt genomen bindt proces (0) u_0 aan bijvoorbeeld $u_8.v_8$ en wordt bij het nagaan van de consistentie gecontroleerd dat $u_8.v_8$ verenigbaar is met $u_2.v_2$. Eenvoudigheidshalve laten we proces (0) direct de binding $u_2.v_2$ genereren).

Proces(0) genereert nu de binding
bord←$p(1,u_0).p(2,u_2).p(3,u_4).p(4,u_6)$.Nil. Deze informatie is voldoende om de oproep Co en heel wat van zijn afstammelingen eveneens door (0) te laten opslorpen. De uitwerking van de oproep Co is gegeven in fig. 4.C.2.

Duiden we alleen de zelfstandige informatiebronnen aan met de voor de berekening belangrijke bindingen, en de nog uit te voeren oproepen, dan kan de staat van de berekening samengevat worden als in fig. 4.C.3. De uit te voeren oproep is $D(p(1,u_0),p(2,u_2))$. De informatie over u_0 en u_2 die toegankelijk is via bron (3) doet alle procedures voor D mislukken. (optimalisatie 1 - proces (3) wordt de enige verdachte). (3) moet dus proberen om andere informatie te genereren. Ook (5) en (7) moeten ongedaan gemaakt worden daar ze informatie afkomstig van (3), gebruikten. Het gaat hier om een conflict tussen de 1ste en de 2de koningin. Het conflict moet opgelost worden door het verplaatsen van de 2de koningin. Als gevolg daarvan moeten de posities van de 3de en de 4de koningin opnieuw berekend worden. Deze berekening is gegeven in fig. 4.C.4.

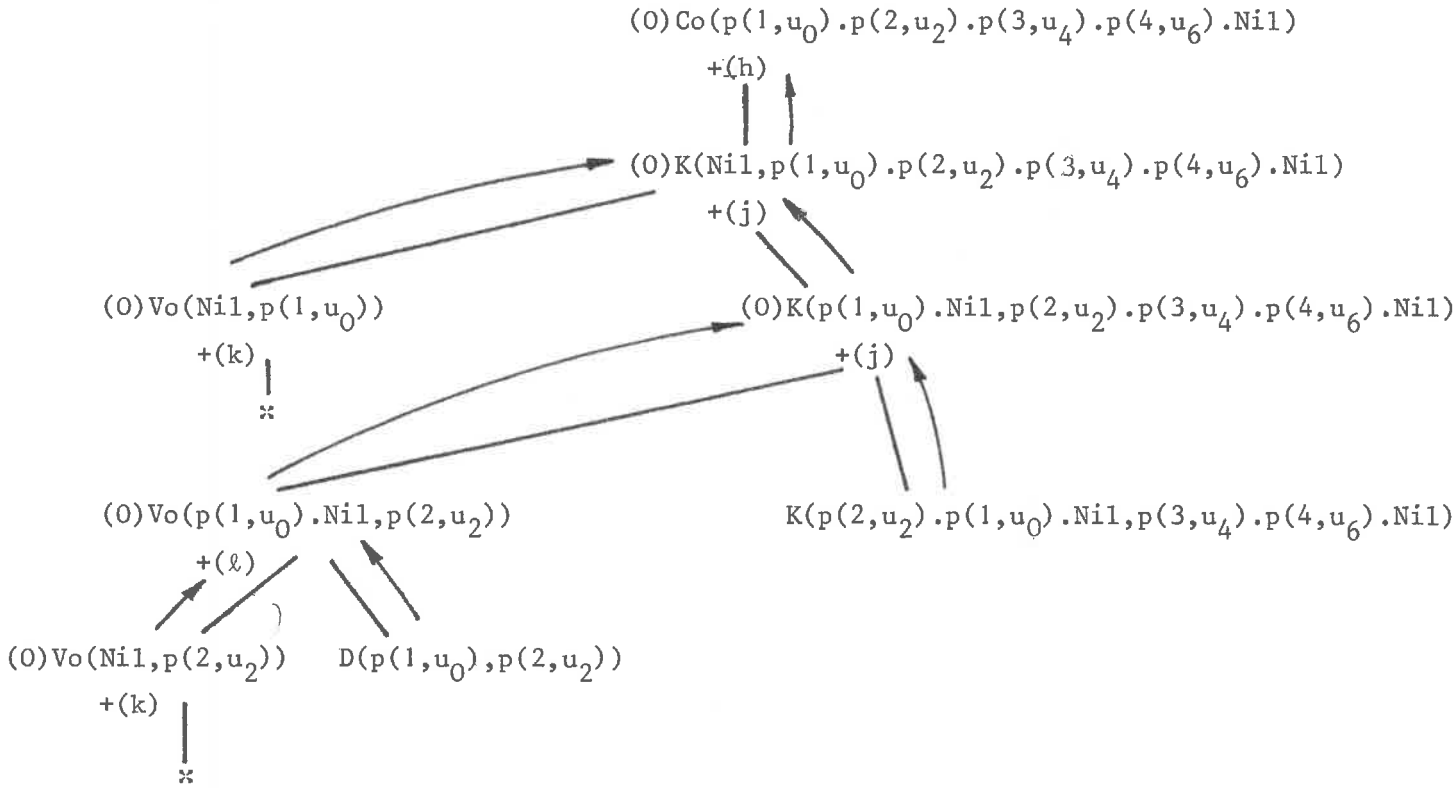


Fig. 4.C.2. : Uitvoering van de oproep Controle tot het punt waar de eerste test D gegenereerd wordt. Op de Diagonaal-oproepen na wordt alles opgeslorpt door proces (O) en dus slechts eenmaal uitgevoerd.

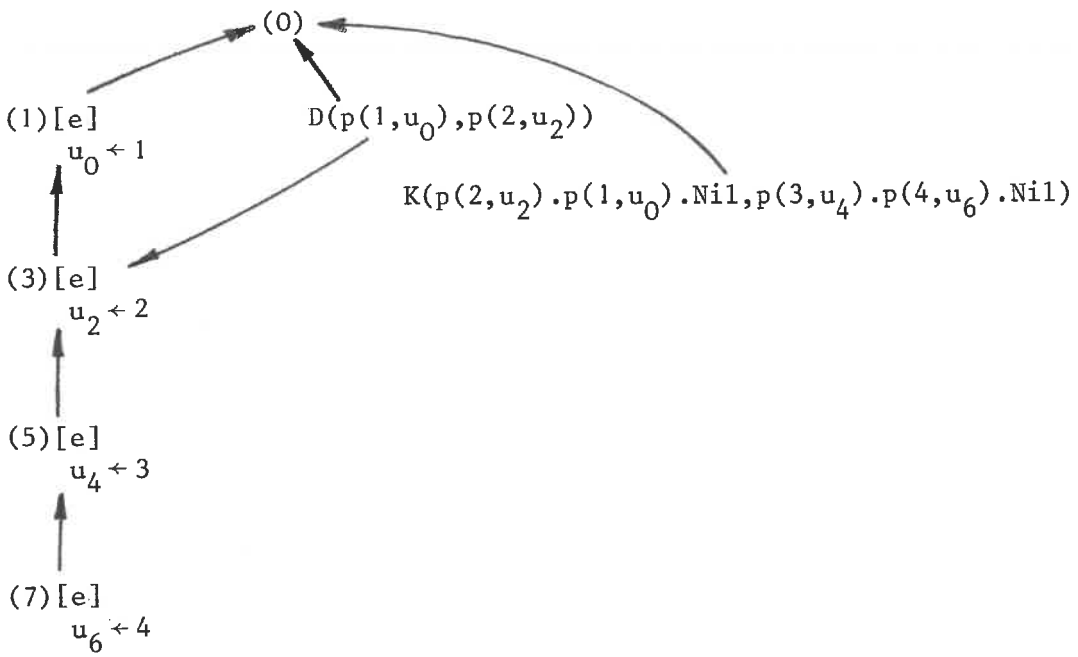


Fig.4.C.3. : De informatie toegankelijk via bron (3) doet de oproep D blokkeren. (Om de figuur te vereenvoudigen laten we de EN-boom weg maar behouden de soms redundante afhankelijkheid tussen zonen en vaders).

De oproepen behorend bij (3) waren Vr(u_2,2.3.4.Nil,w_2) en P(w_2,v_2), zij werden gecreëerd door een oproep behorend bij (1) en hebben nu dus toegang tot alle informatie in (1) en (0). Daar (e) de laatste procedure is die geschikt is om Vr uit te voeren wordt Vr door (1) opgeslorpt. Er ontstaat echter een nieuw zelfstandig proces namelijk Vr(u_2,3.4.Nil,w_3). Ook P(w_2,v_2) wordt nu door (1) opgeslorpt, alsook alle verdere permutatie-oproepen. Uit de bindingen $w_4 \leftarrow w_3$, $w_3 \leftarrow 4.Nil$, $w_4 \leftarrow x_5.y_5$, die toegankelijk zijn via de processen (8) en (9) kan men de bindingen $x_5 \leftarrow 4$ en $y_5 \leftarrow Nil$ afleiden. Deze afgeleide informatie die nodig is om conflicten te herkennen zou men kunnen onderbrengen in een pseudo proces dat van (8) en (9) afhangt. Om de voorstelling eenvoudig te houden zullen we dit niet doen.

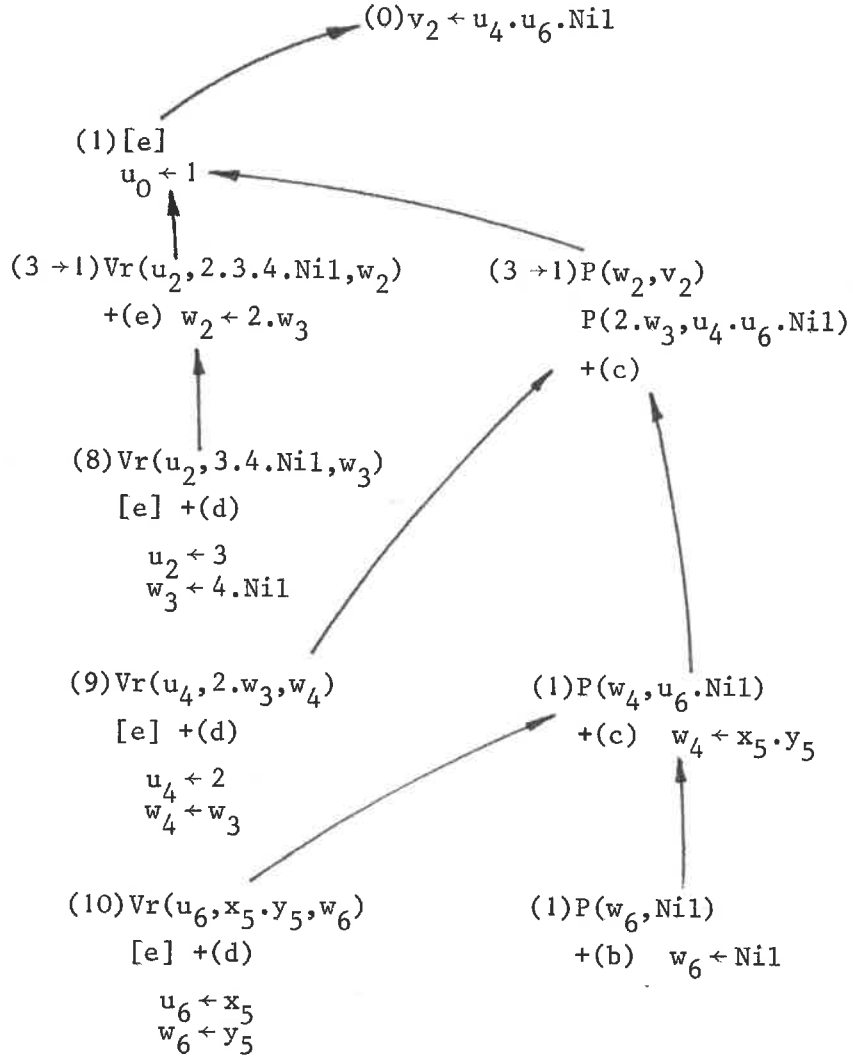


Fig. 4.C.4. : Een nieuwe start. $Vr(u_2,2.3.4.Nil,w_2)$ en $P(w_2,v_2)$ behorend tot proces (3) worden opnieuw uitgevoerd en worden nu door proces (1) opgeslorpt aangeduid als (3→1)).

De nieuwe struktuur, die samengevat is in fig. 4.C.5 is totaal verschillend van de oorspronkelijke (fig. 4.C.3). Doordat bron (0) nu over meer informatie beschikte was het mogelijk om alle permutatie-oproepen te laten opslorpen. Daardoor gebeurt het plaatsen van de 2de, 3de en 4de koningin onafhankelijk van elkaar (de processen (8), (9) en (10)). Dit is mogelijk omdat de procedure Verwijder een positie selekteert in een lijst; welk element er in die positie staat is van geen belang. We kunnen stellen dat het programma "bijgeleerd" heeft.

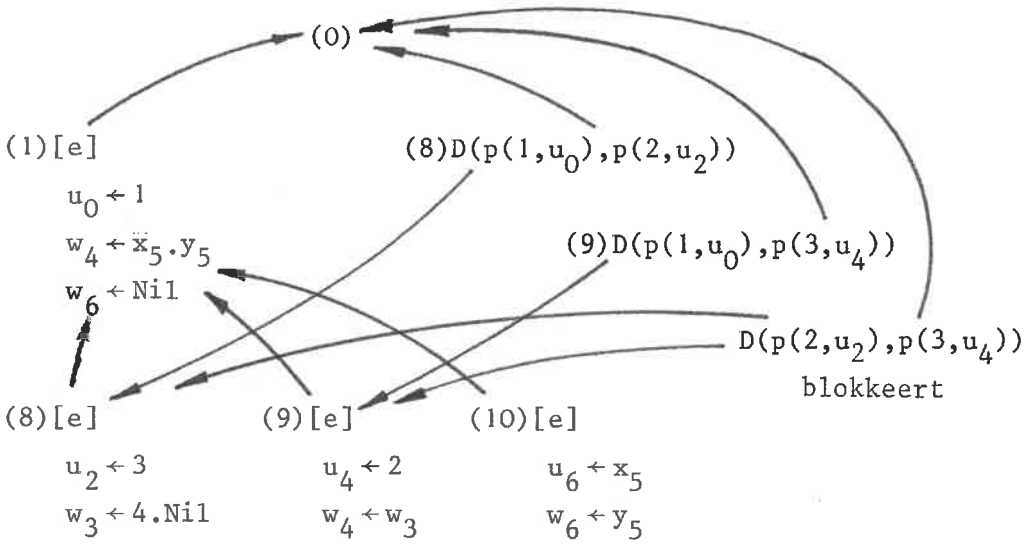


Fig. 4.C.5. : Door toedoen van de processen (8) en (9) worden alle procedures uitgeschakeld waarover D(p(2,u_2),p(3,u_4) beschikt. Dit veroorzaakt een conflict tussen de processen (8) en (9).

De informatie beschikbaar via (8) elimineert op één na alle procedures voor D(p(1,u_0),p(2,u_2)). Deze oproep wordt dus door (8) opgeslorpt. De oproep K(p(2,u_2).p(1,u_0).Nil,p(3,u_4).p(4,u_6).Nil) wordt natuurlijk door (0) opgeslorpt en genereert de verschillende diagonaaltesten. D(p(1,u_0),p(3,u_4)) wordt door (9) opgeslorpt, D(p(2,u_2),p(3,u_4)) echter, blokkeert. De informatie toegankelijk via (8) en (9) elimineert alle procedures. Er is dus een conflict tussen de onafhankelijke informatiebronnen (8) en (9). Deze bronnen zijn niet langer onafhankelijk en de ermee geassocieerde oproepen kunnen niet langer beschouwd worden als gelijktijdig uitvoerbaar. We moeten nu beslissen of we een informatiestroom creëren van (8) naar (9) of van (9) naar (8), in andere woorden :

of de oproepen geassocieerd met (8) ofwel voor ofwel na deze van (9) moeten uitgevoerd worden. Bemerk dat bij de naïeve sequentiële exploratiemethode deze beslissing veel vroeger moet genomen worden en dus op veel minder informatie gebaseerd is. Nagaan welke keuze tot de meest efficiënte uitvoering leidt is moeilijk en, in systemen waar elke door de gebruiker opgelegde controle-informatie ontbreekt (stellingbewijzers) erg belangrijk. We stellen ons vertrouwen in de selektiestrategie en dit betekent dat (9), als laatst geselekteerde opnieuw moet uitgevoerd worden. (9) wordt dus meteen direct afhankelijk van (8). Bemerk dat proces (10) dat de 4de koningin plaatst, niet beïnvloed wordt.
(In het algemeen geval zouden de processen die de 5de, 6de, 7de en 8ste koningin plaatsen, behouden blijven!). Ook de D-test op de 1ste en 2de koningin blijft behouden.

De nieuwe start is beschreven in fig. 4.C.6. De oproep Vr die hernomen wordt, heeft nu toegang tot de informatie van bron (8), en daar (e) de enige overblijvende procedure is, wordt de oproep door (8) opgeslorpt. Opnieuw blokkeert de uitvoering op $D(p(2,u_2),p(3,u_4))$. Proces (11) moet herzien worden. $Vr(u_4,4.Nil,y_5)$ uitvoeren met procedure (e) laat aan (8) toe om deze oproep op te slorpen en creëert een procedure-oproep Vr met Nil als tweede argument. Geen enkele procedure stemt daarmee overeen en $Vr(u_4,4.Nil,y_5)$ heeft dus geen oplossing. Het volledige proces (8) moet dus herzien worden.

In fig. 4.C.7 wordt de nieuwe poging beschreven. Bemerk dat het proces (10) nog steeds bewaard blijft. Alle D-testen worden hernomen : ze slagen nu behalve de laatste die de 3de koningin met de 4de vergelijkt. Het conflict is terug te brengen tot de informatie toegankelijk via processen (13) en (10). Proces (10) moet dus herzien worden. Het heeft nu toegang tot de binding $w_6 \leftarrow$Nil die door proces (1) geproduceerd werd. Deze informatie maakt procedure (e) ongeschikt en laat (1) toe om (10) op te slorpen. (13) dus moet herzien worden : fig. 4.C.8.

De test $D(p(1,u_0),p(3,u_4))$ blokkeert nu. Hij hangt van (19) af. (19) moet dus uitgevoerd worden met procedure (e). Dit creëert de oproep $Vr(u_4,w_{31},y_6)$. Oplossingen voor deze oproep komen in conflict met de binding $w_{31} \leftarrow$Nil die behoort bij (12). (19) wordt van (12) afhankelijk en is meteen onoplosbaar. (12) moet dus herzien worden.

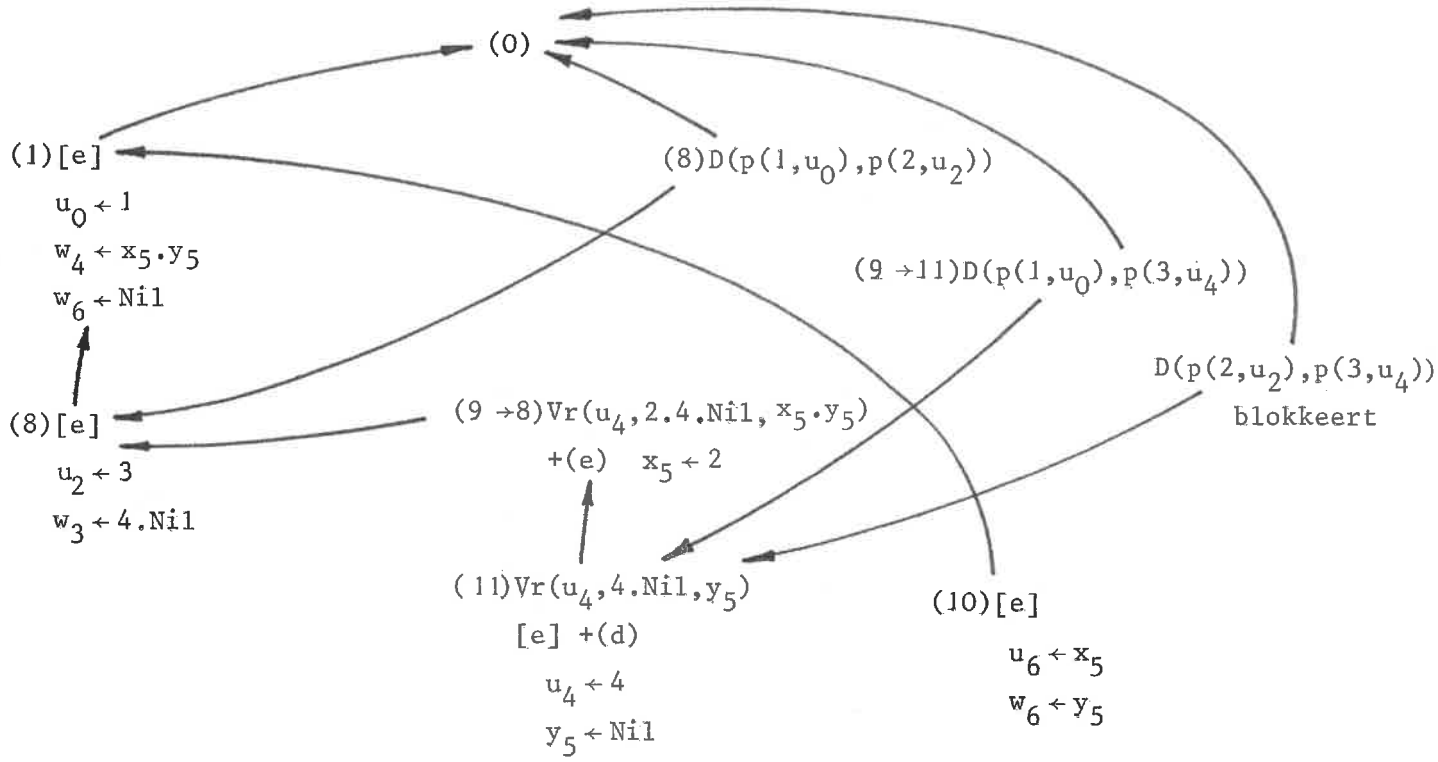


Fig. 4.C.6. : De 3de koningin wordt verplaatst, een nieuw conflict ontstaat.

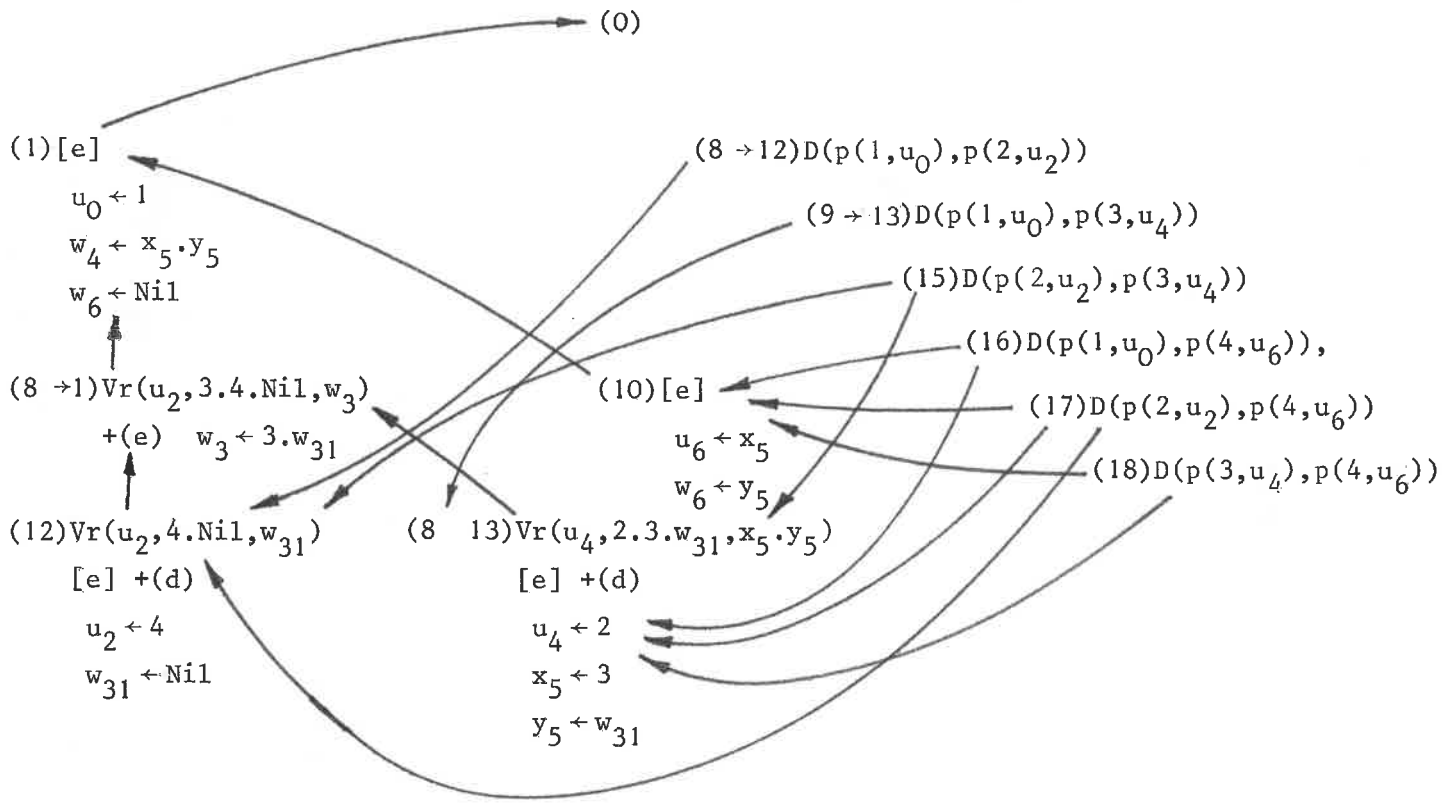


Fig. 4.C.7. : Nieuwe D-testen worden gegenereerd, zij slagen allemaal behalve de laatste : uit $u_6 \leftarrow x_5$, $x_5 \leftarrow 3$ volgt $u_6 \leftarrow 3$ en dit is niet verenigbaar met $u_4 \leftarrow 2$.

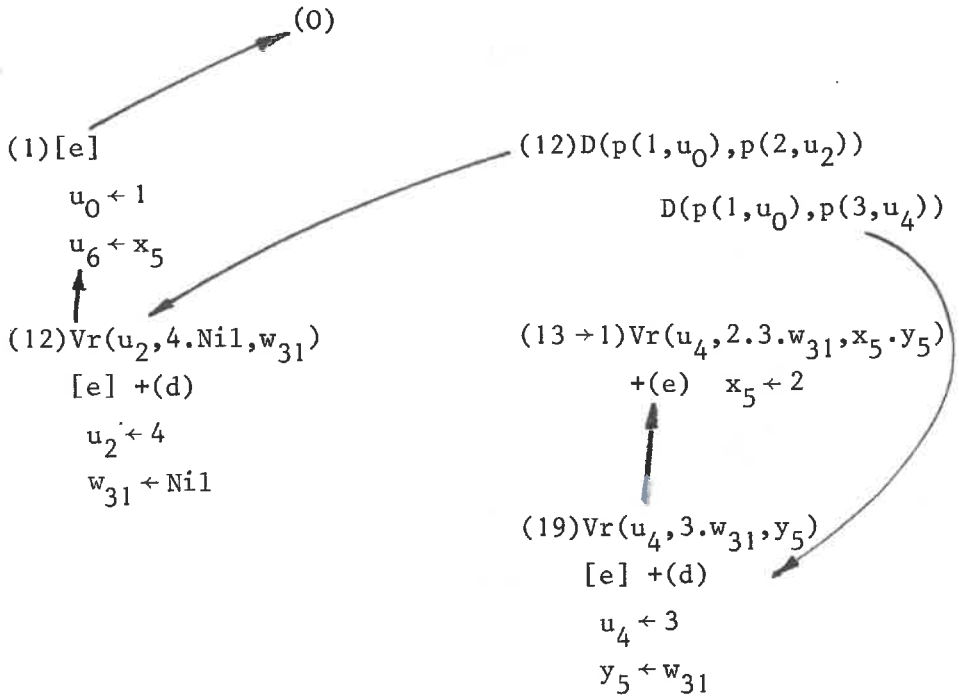


Fig. 4.C.8. : Een conflict tussen 1ste en 3de koningin

Het gebruik van (e) leidt tot de oproep Vr(u_2,Nil,w_{31}). Deze oproep heeft geen oplossing en (12) blokkeert. Dit veroorzaakt het herzien van (1) : fig. 4.C.9. Alle permutatie-oproepen worden door proces (0) opgeslorpt en verdwijnen voor goed uit de berekening. De vier processen die een koningin plaatsen zijn nu onafhankelijk van elkaar. De struktuur is samengevat in fig. 4.C.10.

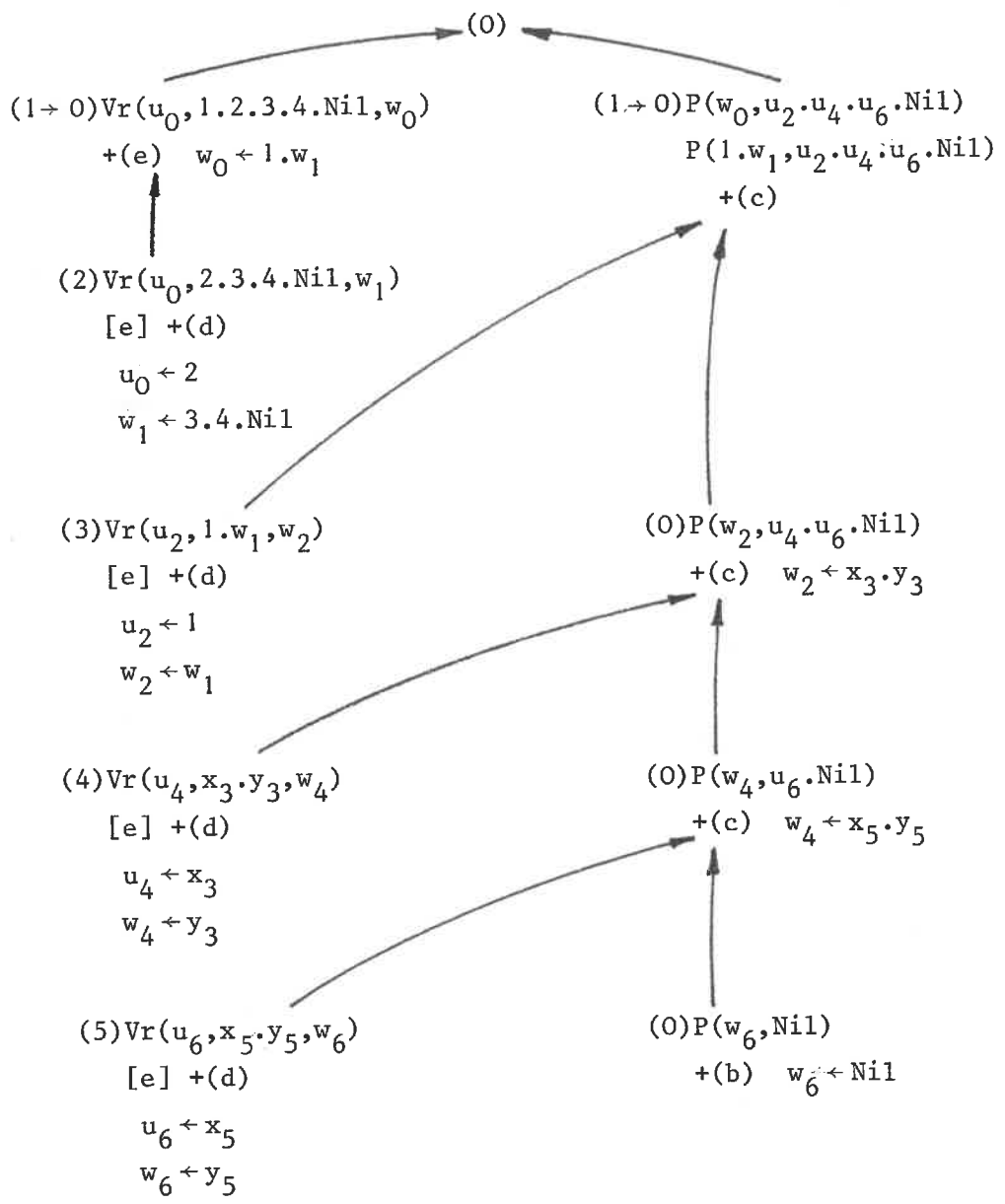


Fig. 4.C.9. : De oproepen $Vr(u_0,1.2.3.4.Nil,w_0)$ en $P(w_0,v_0)$ met $v_0 \leftarrow u_2.u_4.u_6.Nil$ worden opnieuw uitgevoerd.

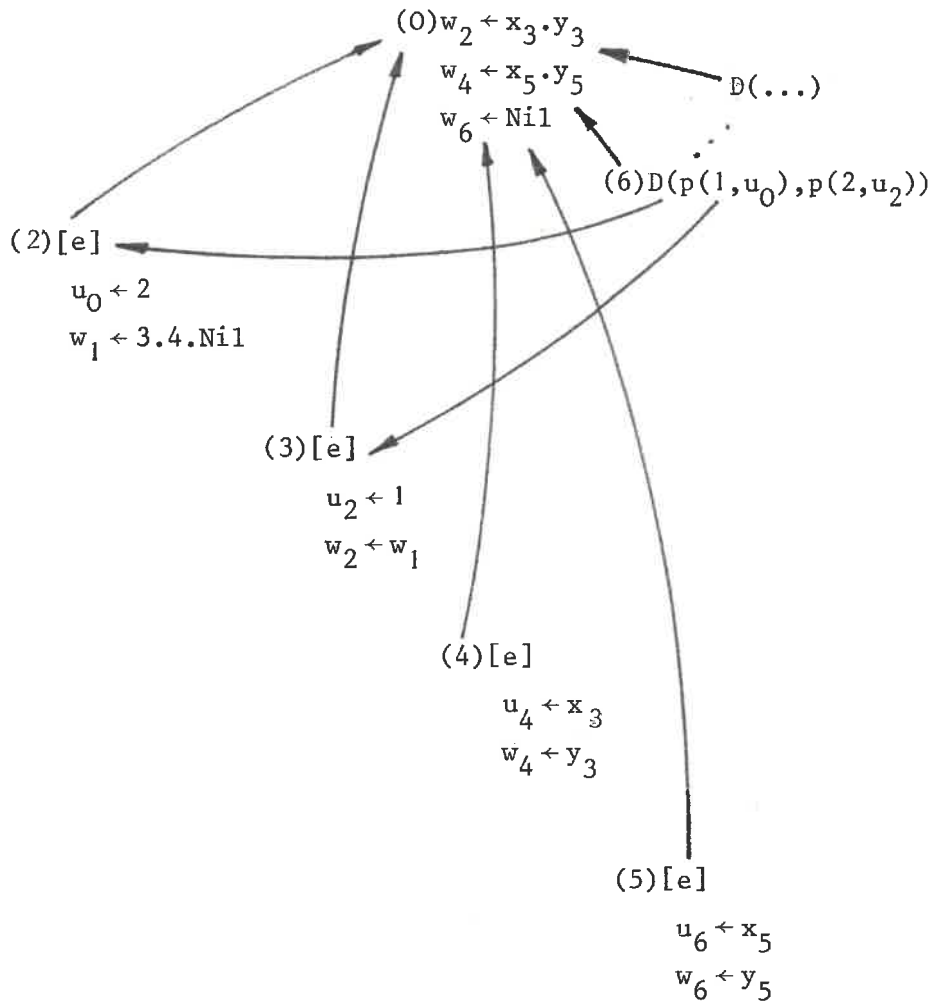


Fig. 4.C.10. : De informatie $u_0 \leftarrow 2$ en $u_2 \leftarrow 1$ elimineert alle procedures waarover (6) beschikt.

De test D(p(1,u_0),p(2,u_2)) blokkeert. Er is dus een conflict tussen de informatie in (2) en (3). We creëeren een informatiestroom van (2) naar (3). De processen (4) en (5) die respectievelijk de 3de en 4de koningin plaatsen blijven behouden. Gebruik makend van de informatie in (2) en (0) wordt de oproep in (3) nu : Vr(u_2,1.3.4.Nil,$x_3.y_3$). De oproep wordt door (1) opgeslorpt en geeft aanleiding tot de substitutie $x_3 \leftarrow 1$ en de oproep Vr(u_2,3.4.Nil,y_3). Deze laatse oproep wordt het nieuwe zelfstandige proces (7).

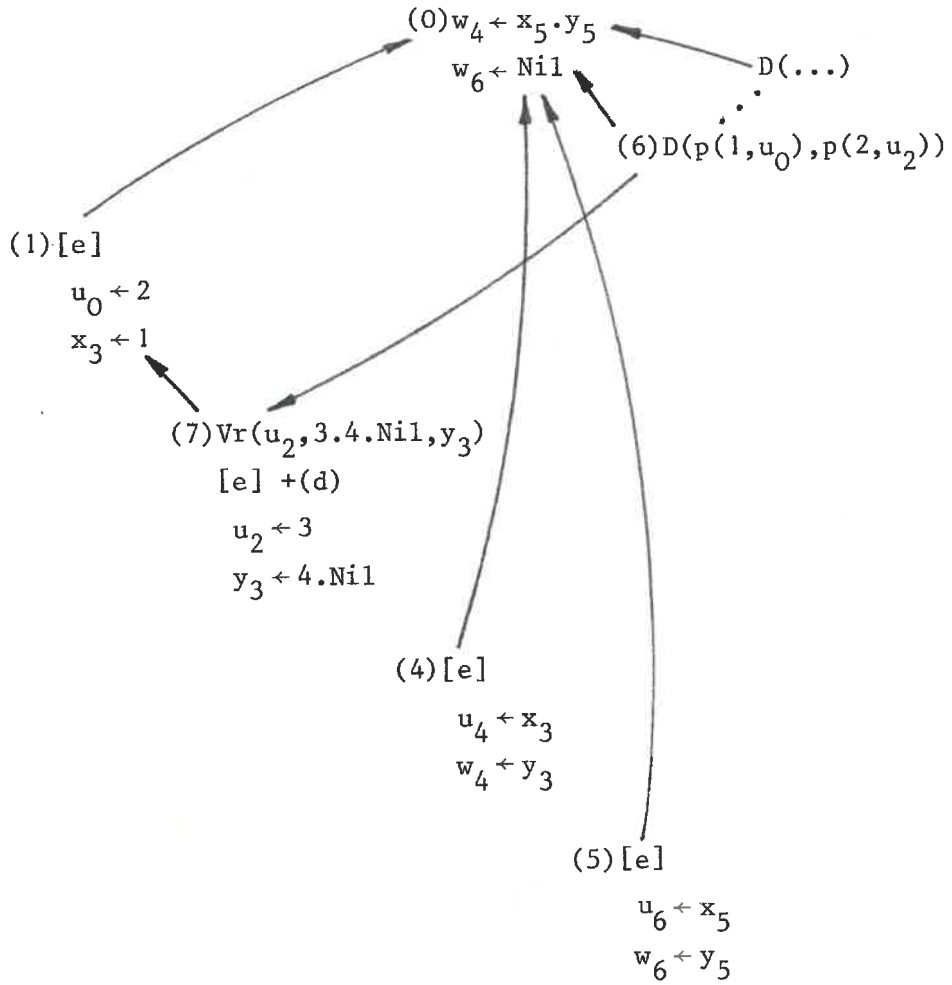


Fig. 4.C.11. : Ook de bindingen $u_0 \leftarrow 2$ en $u_2 \leftarrow 3$ brengen geen oplossing voor D(p(1,u_0),p(2,u_2)).

Als gevolg van het eerste conflict gebeurt het plaatsen van de 1ste en de 2de koningin niet langer onafhankelijk van elkaar. Opnieuw blokkeert de oproep D(p(1,u_0),p(2,u_2)) (fig. 4.C.11) en (7) moet herzien worden. Daar procedure (e) de laatste is kan de oproep Vr(u_2,3.4.Nil,y_3) door (1) opgeslorpt worden. De uitvoering geeft aanleiding tot de substitutie $y_3 \leftarrow 3.y_{31}$ en de oproep Vr(u_2,4.Nil,y_{31}). We krijgen fig. 4.C.12.

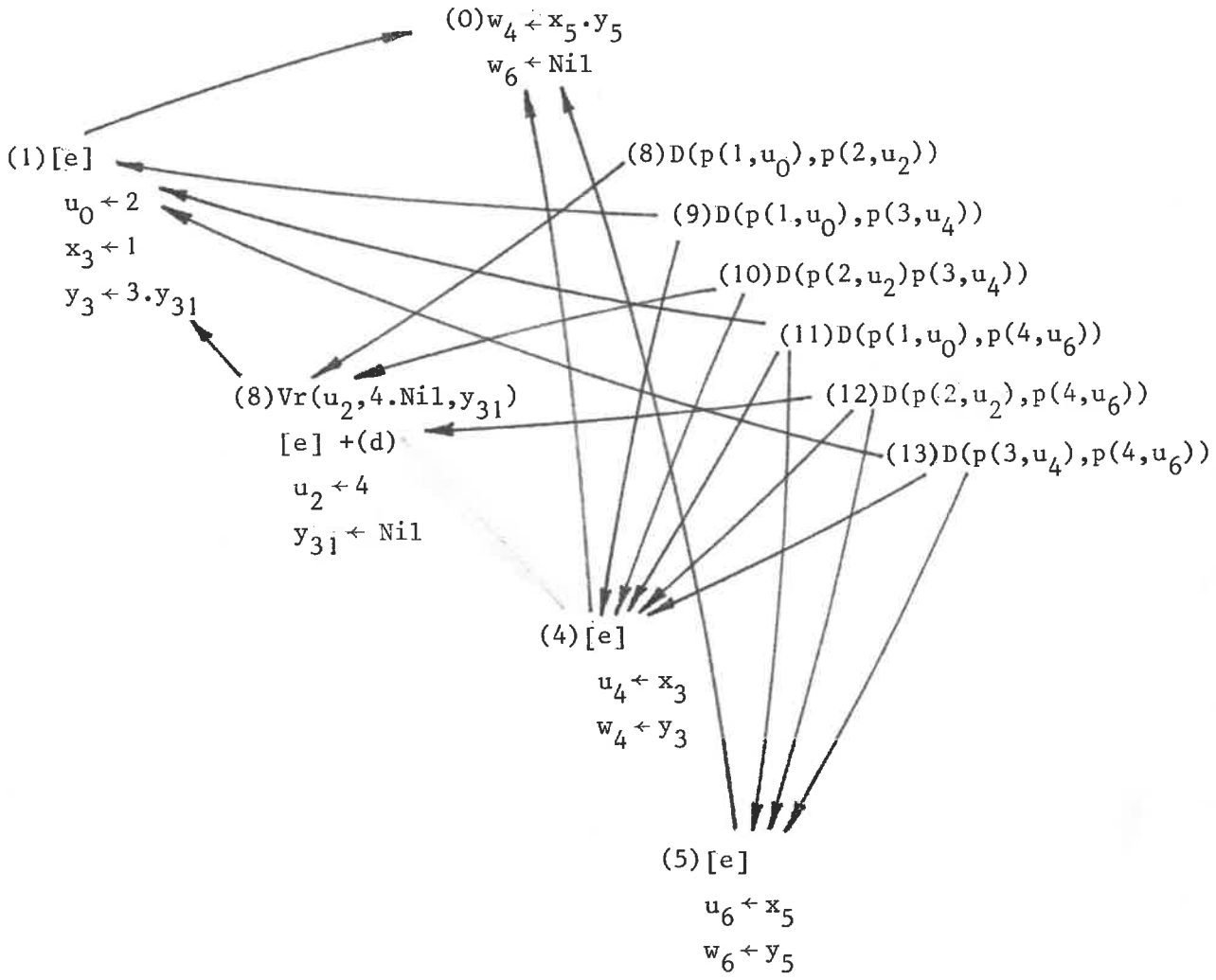


Fig. 4.C.12. : Een oplossing.

Deze maal slagen alle testen en vinden we de oplossing p(1,2).p(2,4).p(3,1),p(4,3).Nil.

Besluit

We zien dat grote stukken van de berekening slechts eenmaal uitgevoerd worden, meer bepaald de oproep Opstelling en het grootste deel van de oproep Controle. De berekening spitst zich toe op het berekenen van een permutatie enerzijds en het uitvoeren van de diagonaaltesten anderzijds. Waar initieel het berekenen van de permutatie een sequentieel proces is, slaagt het systeem er naderhand in om dit proces op te splitsen

in parallelle processen (de oproepen Verwijder) die elk een koningin plaatsen (eerst voor de 2de, 3de en 4de koningin, later voor alle 4 de koninginnen). Slechts naarmate de Diagonaaltesten conflicten ontdekken ontstaan er informatiestromen tussen deze processen en wordt het parallellisme terug ingeperkt. Bemerk dat zodra een proces herzien wordt, de processen die ervan afhangen volledig moeten herzien worden en terug starten als onafhankelijke processen (vb. wanneer de eerste koningin herplaatst werd, moest de volledige permutatie-berekening opnieuw uitgevoerd worden, maar we kregen vier onafhankelijke processen).

C.5. KONINGINNENPROBLEEM - OPLOSSING DOOR HET UITBREIDEN VAN EEN PARTIELE OPSTELLING

Een meer conventionele oplossing van het koninginnenprobleem bestaat erin dat men een partiële oplossing, bestaande uit m koninginnen die in vrede samenleven op de m eerste rijen, stap per stap uitbreidt totdat men een volledige oplossing heeft. Een uitbreiding bestaat erin dat men een koningin plaatst op de m+1ste rij en op een kolom waar ze niet in conflict komt met de m reeds aanwezige koninginnen. Is het uitbreiden van een partiële oplossing niet mogelijk, dan moet die partiële oplossing gewijzigd worden. De sequentiële exploratietechniek zorgt ervoor dat dit op een systematische wijze gebeurt.

We stellen een partiële oplossing voor door een struktuur m:q met m het aantal koninginnen en q de lijst met hun posities. De ledige partiële oplossing wordt dus voorgesteld door 0:Nil. Een positie stellen we terug voor door p(r,k) met r het rijnummer en k het kolomnummer.

Het programma :

a. Oplossing(bord) ← O(0:Nil,bord)
b. O(4:bord,bord) ←
c. O(m:q,bord) ← Voeg_toe(m:q,m_1:r) O(m_1:r,bord)
d. Voeg_toe(m:q,m_1:p(m_1,k).q) ← m_1=m+1 genkolom(k) Vrede(p(m_1,k),q)
e. genkolom(1) ←
f. genkolom(2) ←
g. genkolom(3) ←
h. genkolom(4) ←
i. Vrede(p,Nil) ←
j. Vrede(p,q.ℓ) ← Vrede(p,ℓ) OK(p,q)

De relatie OK bestaat uit paren koninginnen die elkaar niet uitsluiten.

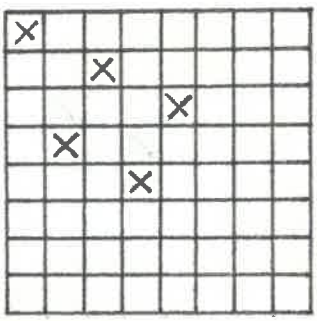
Het begin van de uitvoering is gegeven in fig. 4.C.13. We merken op dat bijna alle oproepen door het initiële proces opgeslorpt worden. Alleen de oproepen genkolom en OK blijven als zelfstandige processen bestaan. Dit betekent dat het programma zich in feite herleidt tot de volgende probleemstelling (genkolom wordt afgekort als GK).

$\leftarrow GK(k_1)\ GK(k_2)\ OK(p(2,k_2),p(1,k_1))$
$GK(k_3)\ OK(p(3,k_3),p(1,k_1))\ OK(p(3,k_3),p(2,k_2))$
$GK(k_4)\ OK(p(4,k_4),p(1,k_1))\ OK(p(4,k_4),p(2,k_2))\ OK(p(4,k_4),p(3,k_3))$.

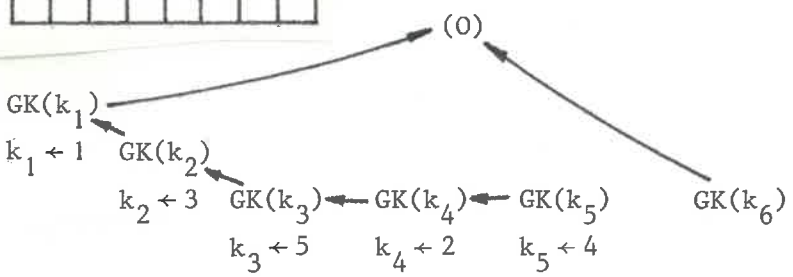
Met de naïeve sequentiële exploratiemethode en dezelfde selektieregel zouden de diverse oproepen O, Voeg_toe, Plus en Vrede vele malen uitgevoerd worden. Hier worden ze slechts eenmaal uitgevoerd.

Beperken we ons tot deze laatste probleemstelling, dan zal de verbeterde exploratietechniek weinig afwijken van de naïeve. Weliswaar zijn initieel de verschillende processen genkolom onafhankelijk van elkaar, het ontdekken van conflicten door de processen OK leidt echter spoedig tot een totale ordening tussen de processen genkolom.

Nochtans zijn bepaalde optimalisaties mogelijk, maar deze komen slechts voor bij grotere dimensies. We geven een voorbeeld voor het 8-koninginnenprobleem :



Deze partiële oplossing bestaat uit 5 koninginnen, ze moet uitgebreid worden met een 6de koningin. We veronderstellen dat de generators $GK(k_1)...GK(k_5)$ totaal geordend zijn. De afhankelijkheid tussen de processen is dus als volgt :



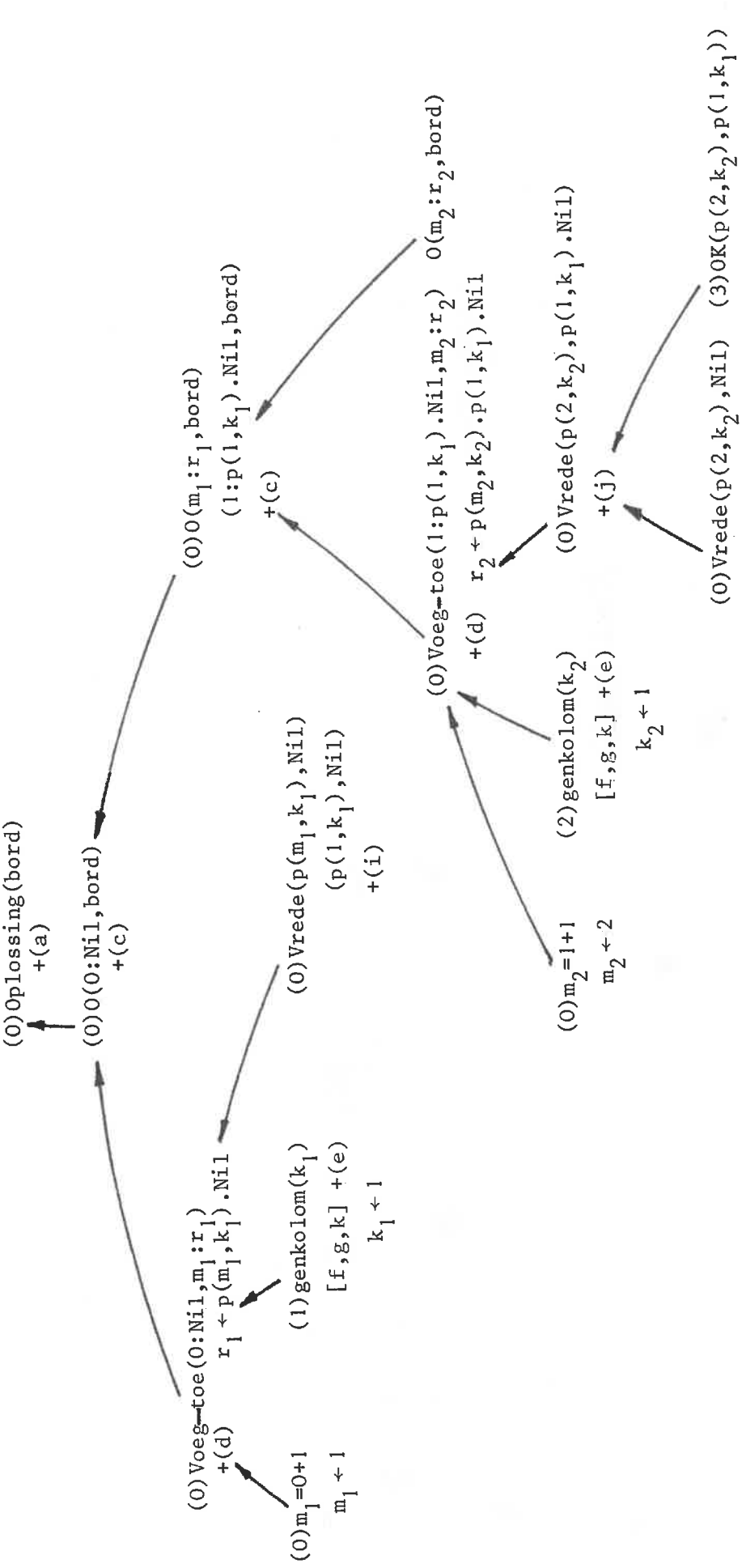
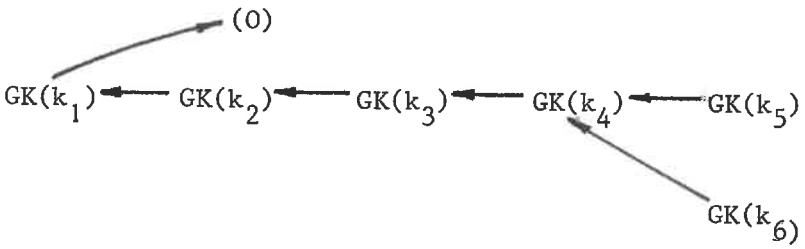


Fig. 4.C.13. : Alleen genkolom en OK blijven als zelfstandige processen bestaan.

$GK(k_6)$ start als onafhankelijk proces en beschouwt de waarden 1 tot 8.

1 brengt $GK(k_6)$ in conflict met $GK(k_1)$
2 brengt $GK(k_6)$ in conflict met $GK(k_4)$
3 brengt $GK(k_6)$ in conflict met $GK(k_2)$
4 brengt $GK(k_6)$ in conflict met $GK(k_4)$
5 brengt $GK(k_6)$ in conflict met $GK(k_3)$
6 brengt $GK(k_6)$ in conflict met $GK(k_1)$
7 brengt $GK(k_6)$ in conflict met $GK(k_2)$
8 brengt $GK(k_6)$ in conflict met $GK(k_3)$

Na vereenvoudiging wordt het afhankelijkheidsgraf :



$GK(k_6)$ faalt en is direct afhankelijk van $GK(k_4)$. De waarden $k_5 \leftarrow 5,6,7$ of 8 worden dus niet overwogen. Het systeem ontdekt onmiddellijk dat k_4 moet gewijzigd worden.

Inderdaad, de onmogelijkheid om de 6de koningin te plaatsen is enkel te wijten aan de eerste vier koninginnen en de partiële oplossing bestaande uit de eerste vier koninginnen moet gewijzigd worden.

Bemerk het belang van de controle : deze optimalisatie is slechs mogelijk omdat de nieuwe koningin eerst vergeleken wordt met de eerste, dan met de tweede,...

D. EEN VERDERE VERFIJNING

D.1. PRINCIPE

Tot nu toe kreeg een proces (i) toegang tot de informatie die behoort bij alle processen (de verzameling V) waarvan het afhangt. Werd een proces (i) uitgevoerd met een procedure (d) dan waren de berekende bindingen afhankelijk van alle processen in de verzameling V. Evenzo werd het uitschakelen van een proceduredefinitie toegeschreven aan alle processen uit dezelfde verzameling V. Werd een proces uit de verzameling V herzien, dan gingen alle door proces (i) uitgevoerde berekeningen verloren. De door middel van procedure (d) berekende bindingen waren niet langer geldig en alle uitgeschakelde proceduredefinities kwamen terug ter beschikking.

Door het verder beperken van de informatie-uitwisseling kunnen soms bepaalde door proces (i) berekende resultaten gered worden. Dit is het geval wanneer de informatie waarop ze gebaseerd waren, onveranderd gebleven is.

Een proces (i) krijgt nu enkel toegang tot de informatie die bij zijn voorouders behoort. Zolang die voorouders niet herzien worden, blijven de bindingen die proces (i) na uitvoering met procedure (d) berekend heeft, geldig. (Het herzien van een voorouder heeft de volledige verdwijning van proces (i) tot gevolg). Natuurlijk is het mogelijk dat deze bindingen in strijd komen met de bindingen die door andere processen berekend worden. Op deze conflictsituatie komen we verder terug.

Tevens houden we voor elke uitgeschakelde proceduredefinitie bij welke processen (informatiebronnen) en precies verantwoordelijk zijn voor het uitschakelen van die definitie. Ook deze verzameling (W) zal gewoonlijk kleiner zijn dan de verzameling V van processen waarvan het uitgevoerde proces afhangt. Slechts wanneer een proces uit de verzameling W herzien wordt, komt de uitgeschakelde definitie terug ter beschikking.

We krijgen dus twee soorten informatiestromen. Enerzijds een totale hiërarchische informatiestroom van vader op zoon (langs de takken van de EN-boom) en anderzijds een beperkte zijdelingse informatiestroom tussen processen in verschillende takken van de EN-boom. Deze laatste informatiestroom is beperkt omdat alleen bepaald wordt welke processen medeverantwoordelijk zijn voor het uitschakelen van proceduredefinities.

Samen bepalen beide soorten informatiestromen het afhankelijkheidsgraf.

Is een proces (j) medeverantwoordelijk voor het uitschakelen van een proceduredefinitie (d), dan is het onnodig om bij te houden dat proces (k) voorouder van proces (j) eveneens medeverantwoordelijk is. Deze laatste informatie is redundant; inderdaad, het herzien van proces (k) heeft het herzien (zelfs verdwijnen) van proces (j) tot gevolg en procedure (d) komt dus in elk geval terug ter beschikking.

D.2. ALGORITME 2

Herhaal

1. Selekteer een proces (oproep)
2. Voer dit proces uit. Het proces beschikt alleen over de informatie (bindingen) die behoort bij de voorouders van dit proces.
3. a. Indien geen enkele beschikbare proceduredefinitie met de oproep overeenstemt (rekening houdende met de informatie waarover het proces beschikt), dan is er een conflict. Verantwoordelijk voor het conflict zijn (een verzameling W)
   - de vader (voorouders niet nodig, leidt tot redundantie)
   - de processen die medeverantwoordelijk zijn voor de uitgeschakelde proceduredefinities.

   b. Indien, na uitvoering van het proces, de globale informatie inconsistent is (de nieuwe bindingen zijn in strijd met reeds bestaande bindingen, dan is er een conflict. Verantwoordelijk voor het conflict zijn : (een verzameling W)
   - het pas uitgevoerde proces (voorouders niet nodig - redundantie)
   - processen waarvan de informatie tot de strijdige binding bijdraagt (de strijdige binding kan afgeleid zijn uit bindingen die bij verschillende processen behoren) (voorouders van die processen niet nodig - redundantie).
4. Indien geen conflict, dan verder gaan, anders :
   a. Uit de verzameling W wordt een verzameling "verdachten" afgeleid. Het zijn de processen die volgens de afhankelijkheidsrelatie "laatst" zijn in de verzameling W.

b. Tussen de verdachten wordt een schuldige gekozen.

c. Alle andere tot W behorende processen zijn medeverantwoordelijk voor de uitschakeling van de proceduredefinitie die het schuldige proces gebruikt heeft. Dit wordt genoteerd (nieuwe zijdelingse informatiestromen - redundante informatie kan eventueel verwijderd worden).

d. De door het schuldige proces berekende bindingen verdwijnen; tevens verdwijnen alle processen die van het schuldige proces afstammen.

e. Alle proceduredefinities die mede door het schuldige proces (of zijn afstammelingen) uitgeschakeld werden, komen terug ter beschikking.

tot dat een oplossing gevonden is.

Opmerkingen

1. Wordt in geval 3.b. het pas uitgevoerde proces als schuldige aangewezen en in de volgende iteratie opnieuw geselekteerd, dan hebben we, zoals in Algoritme 1, de situatie waarbij een na een de verschillende definities geprobeerd worden tot ze alle uitgeschakeld zijn, of tot een goede gevonden is. Deze opeenvolgende iteraties kan men als één stap beschouwen.
2. Wordt in de iteratie, die op een conflict volgt, het schuldige proces geselekteerd en heeft dat proces geen andere proceduredefinities ter zijner beschikking, dan komt men onvermijdelijk in een conflict van type 3.a. Men kan dit voorzien en men kan onmiddellijk een definitieve schuldige aanwijzen. (een proces dat over alternatieven beschikt). Zoals bij Algoritme 1 kan dit het aantal geïnduceerde informatiestromen kleiner maken (Slechts de afstammelingen van de definitieve schuldige moeten verdwijnen.)
3. Is de informatie die via de voorouders toegankelijk is, zodanig dat een proces slechts over één geschikte definitie beschikt (alleen direct afhankelijk van de vader), dan kan het door de vader opgeslorpt worden.
4. Dit algoritme is consistent met Algoritme 1. Heeft men, vóór het uitvoeren van een iteratie, met beide algoritmes hetzelfde afhankelijkheidsgraf en overal dezelfde beschikbare proceduredefinities, dan geldt :

- De verzameling V (Algoritme 1) kan uit de verzameling W (Algoritme 2) afgeleid worden door de verzameling W uit te breiden met alle processen waarvan een proces uit W afhangt (sluiting van de verzameling onder de afhankelijkheidsrelatie).
- De verzameling van de verdachten is onder beide algoritmen dezelfde (de "laatsten" in de partiële orde die door de afhankelijkheidsrelatie bepaald is).
- Met beide algoritmen kan men dus dezelfde schuldige kiezen.
- Met Algoritme 1 worden meer berekeningen ongedaan gemaakt. Door het uitvoeren van een aantal iteraties in de gepaste volgorde kan met Algoritme 1 de toestand bekomen worden die bestaat na het beëindigen van de iteratie in Algoritme 2 (zelfde afhankelijkheidsgraf en dezelfde beschikbare proceduredefinities). Inderdaad :
  . Proceduredefinities die behoren bij processen die van het schuldige proces afhangen maar er niet van afstammen komen in Algoritme 1 terug beschikbaar. De informatie waarover Algoritme 2 beschikt laat toe te bepalen welke definities nog steeds ongeschikt zijn. Door het uitvoeren van de gepaste iteraties kan Algoritme 2 de ongeschikte definities opnieuw ontdekken.
  . Bindingen gegenereerd door processen die van het schuldige proces afhangen maar er niet van afstammen, zijn verdwenen. Met Algoritme 2 werden die bindingen onafhankelijk van de informatie in het schuldige proces berekend en blijft de berekening dus geldig (de controle van de globale consistentie is wel complexer). Door het uitvoeren van de gepaste iteratie kan Algorithme 1 die binding opnieuw berekenen.

D.3. VOORBEELDEN

1. $\leftarrow P(x_1)D_1(x_1,x_2)D_2(x_1,x_3)D_1(x_2,x_3)D_3(x_1,x_4)D_2(x_2,x_4)D_1(x_3,x_4)$

a1. $P(1)\leftarrow$	b1. $D_1(1,3)\leftarrow$	c1. $D_2(1,2)\leftarrow$	d1. $D_3(1,2)\leftarrow$
a2. $P(2)\leftarrow$	b2. $D_1(1,4)\leftarrow$	c2. $D_2(1,4)\leftarrow$	d2. $D_3(1,3)\leftarrow$
a3. $P(3)\leftarrow$	b3. $D_1(2,4)\leftarrow$	c3. $D_2(2,1)\leftarrow$	d3. $D_3(2,1)\leftarrow$
a4. $P(4)\leftarrow$	b4. $D_1(3,1)\leftarrow$	c4. $D_2(2,3)\leftarrow$	d4. $D_3(2,3)\leftarrow$
	b5. $D_1(4,1)\leftarrow$	c5. $D_2(3,2)\leftarrow$	d5. $D_3(2,4)\leftarrow$
	b6. $D_1(4,2)\leftarrow$	c6. $D_2(3,4)\leftarrow$	d6. $D_3(3,1)\leftarrow$
		c7. $D_2(4,1)\leftarrow$	d7. $D_3(3,2)\leftarrow$
		c8. $D_2(4,3)\leftarrow$	d8. $D_3(3,4)\leftarrow$
			d9. $D_3(4,2)\leftarrow$
			d10. $D_3(4,3)\leftarrow$

Het betreft hier een voorbeeld uit een speciale groep van programma's waarin alle relaties gedefinieerd zijn door een expliciete opsomming van hun elementen. Aan het efficient oplossen van dergelijke problemen werd reeds heel wat onderzoek besteed [Mackworth 77, Freuder 78]; we komen er in de volgende sektie op terug.

Het betreft hier opnieuw een oplossing voor het 4-Koninginnenprobleem. x_i is het kolomnummer van de koningin op de ide rij. De relatie P definieert alle mogelijke kolomnummers. De relatie $D_i(x_k,x_\ell)$ met $\ell=k+i$ beschrijft welke combinaties er toegelaten zijn tussen de koninginnen op de kde en de ℓde rij.

De eerste stappen van de uitvoering zijn beschreven in de figuren 4.D.1 en 4.D.2. Op de lijst der alternatieve proceduredefinities duiden we aan door welke processen de procedures uitgeschakeld zijn. De procedures voor proces (4) ($D_1(x_2,x_3)$) worden alle uitgeschakeld, deels door proces (2) en deels door proces (3). Daaruit volgt een conflict tussen de processen (2) en (3). We lossen het op door proces (2) de door proces (3) gebruikte procedure c1 te laten uitschakelen. (3) maakt nu gebruik van c2 (fig. 4.D.2.) en (4) beschikt terug over b4. Proces (4) blokkeert opnieuw en er ontstaat een nieuw conflict tussen de processen (2) en (3). (Daar er reeds een informatiestroom bestaat van proces (2) naar proces (3) moeten we nu noodzakelijkerwijze proces (3) als schuldige aanwijzen). Proces (2) schakelt dus ook c2 uit. Daar Proces (3) over geen andere procedures beschikt ontstaat er een nieuw conflict, nu tussen de processen (1) en (2). Deze maal besluiten we dat proces (2) schuldig is en proces (1) schakelt de procedure b1 uit. Proces (2) moet nu uitgevoerd worden met behulp van b2 terwijl proces (3) terug beschikt over c1 en c2 en proces (4) over b1 - b6.

Door de informatie-overdracht te verfijnen kunnen we nu niet alleen het juiste herstartpunt bepalen maar kunnen we eveneens de verzameling der uit te proberen proceduredefinities kleiner maken.

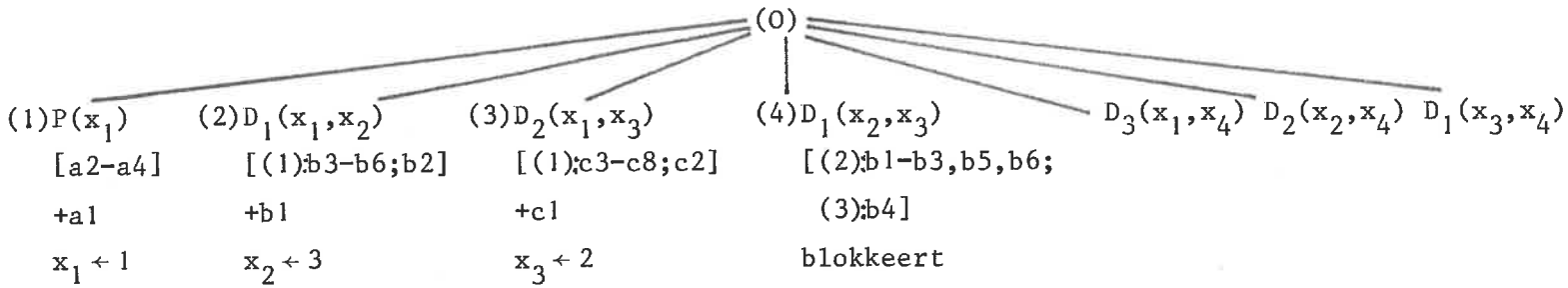


Fig. 4.D.1. : Wanneer proces (2) gebruik maakt van de procedures b3-b6 wordt x_1 gebonden aan een waarde die strijdig is met de binding $x_1 \leftarrow 1$ van proces (1). Proces (2) wordt als schuldige aangewezen en de procedures b3-b6 worden door proces (1) uitgeschakeld. Hetzelfde gebeurt met de procedures c3-c8 van proces (3). Bij proces (4) worden alle procedures uitgeschakeld. Dit leidt tot een conflict tussen de processen (2) en (3).

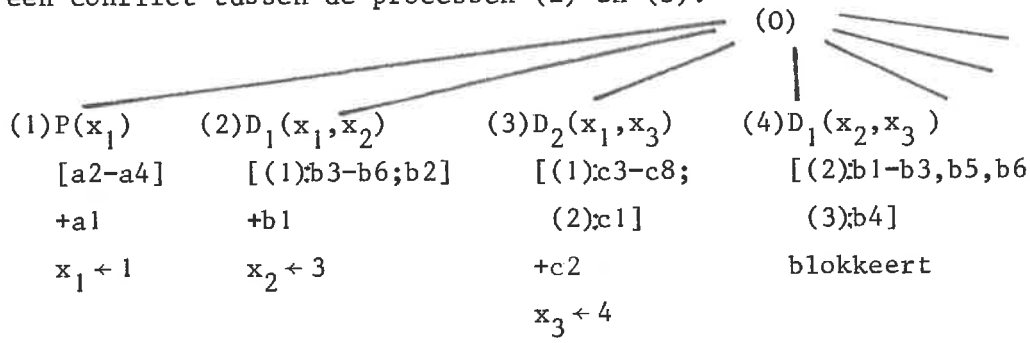


Fig. 4.D.2. : De door proces (3) gebruikte procedure c1 wordt door proces (2) uitgeschakeld. Proces (4) kan nu terug beschikken over procedure b4. Proces (1) gebruikt nu procedure c2; proces (4) blokkeert opnieuw.

2. We hernemen ook even het permutatieprogramma voor het oplossen van het koninginnenprobleem. De uitvoering is geïllustreerd in fig. 4.D.3. De procedures waarover proces (15) beschikt worden alle uitgeschakeld, de ene door proces (1), de andere door de binding $u_2 \leftarrow 2$ die uit de informatie toegankelijk via (1) en (4) kan afgeleid worden. Processen (1) en (4) zijn de verdachten. Proces (4) wordt als schuldige aangewezen en proces (1) schakelt dus de door proces (4) gebruikte procedure (d) uit. Het proces (6) kan terug beschikken over de procedure (b) (Perm(Nil,Nil)←). Op dit ogenblik echter, en dus zeker ook op het ogenblik dat (6) het gebruik van (b) zal overwegen, heeft (6) toegang tot de informatie $v_2 \leftarrow u_4.v_4$ van proces (0). Dit betekent dat procedure (b) definitief kan uitgeschakeld worden en dat de permutatie-oproep slechts éénmaal moet uitgevoerd worden en ten gepaste tijde door proces (0) zal opgeslorpt worden. De berekening herleidt zich dus tot het uitvoeren van de oproepen Verwijder en Diagonaal. We kunnen de wisselwerking tussen de Verwijder en Diagonaal processen evengoed nagaan door het bestuderen van een eenvoudiger probleem :

$$\leftarrow Vr(x_1,1.2.3.4.Nil,u_1.v_1),Vr(x_2,u_1.v_1,u_2.v_2),Vr(x_3,u_2.v_2,u_3.v_3),$$
$$Vr(x_4,u_3.v_3,Nil),D_1(x_1,x_2),D_2(x_1,x_3),D_1(x_2,x_3),D_3(x_1,x_4),D_2(x_2,x_4),$$
$$D_3(x_3,x_4).$$

Hierin is x_i de kolomcompositie van de koningin op de $i$de rij en zijn de procedures D_i gedefinieerd zoals in het vorige voorbeeld. Enkele stappen van de uitvoering zijn geïllustreerd in de figuren 4.D.4 tot 4.D.7. Bemerk dat er in feite een totale ordening ontstaat tussen de Verwijder-processen en dat men bij een conflict tussen x_i en x_k $(k>i)$ in feite op zoek gaat naar de x_j $(j \leqslant k)$ met zo groot mogelijke index die kan verplaatst worden. Het verplaatsen van x_j heeft echter geen enkele rechtstreekse invloed op de processen $Vr(x_\ell,...)$ met $\ell > j$. De door die processen uitgevoerde berekeningen blijven behouden, alleen komen een aantal alternatieve proceduredefinities die uitgeschakeld waren, terug ter beschikking.

In fig. 4.D.8 en fig. 4.D.9 geven wij een deel van de oplossingsruimten van het 8-koninginnenprobleem volgens beide oplossingsmethodes. Het conventionele programma dat een partiële oplossing uitbreidt bevat met de intelligente sequentiële exploratietechniek 75 knooppunten (fig. 4.D.8). Met de naïeve exploratie zijn er 137 knooppunten. Voor een equivalente oplossingsruimte bevat het permutatieprogramma 74 knooppunten.

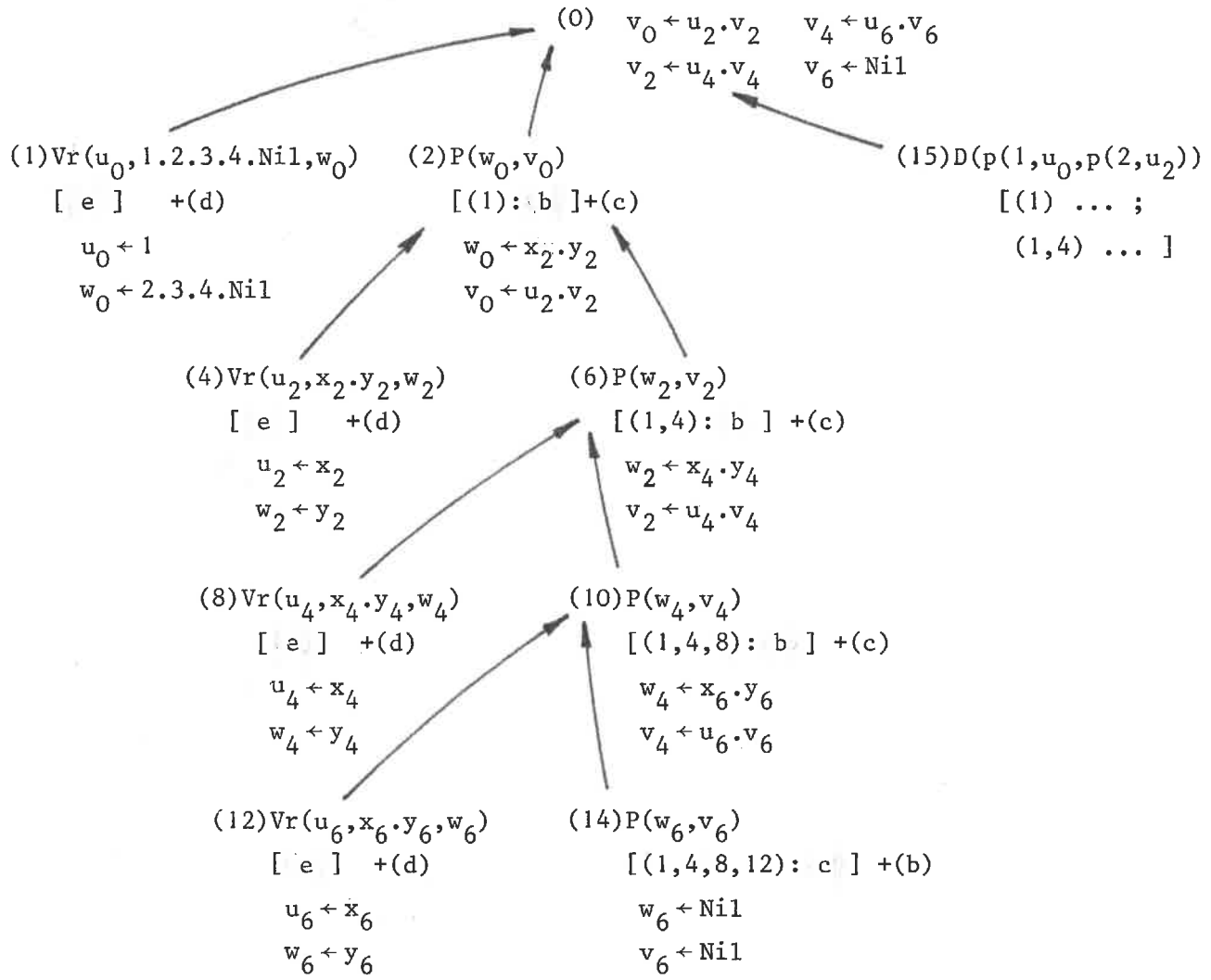


Fig. 4.D.3. : Uit $u_2 \leftarrow x_2$, $w_0 \leftarrow x_2.y_2$ en $w_0 \leftarrow 2.3.4.Nil$ volgt $u_2 \leftarrow 2$. Samen met $u_0 \leftarrow 1$ schakelt deze informatie alle procedures uit waarover D(p(1,u_0),p(2,u_2)) beschikt.

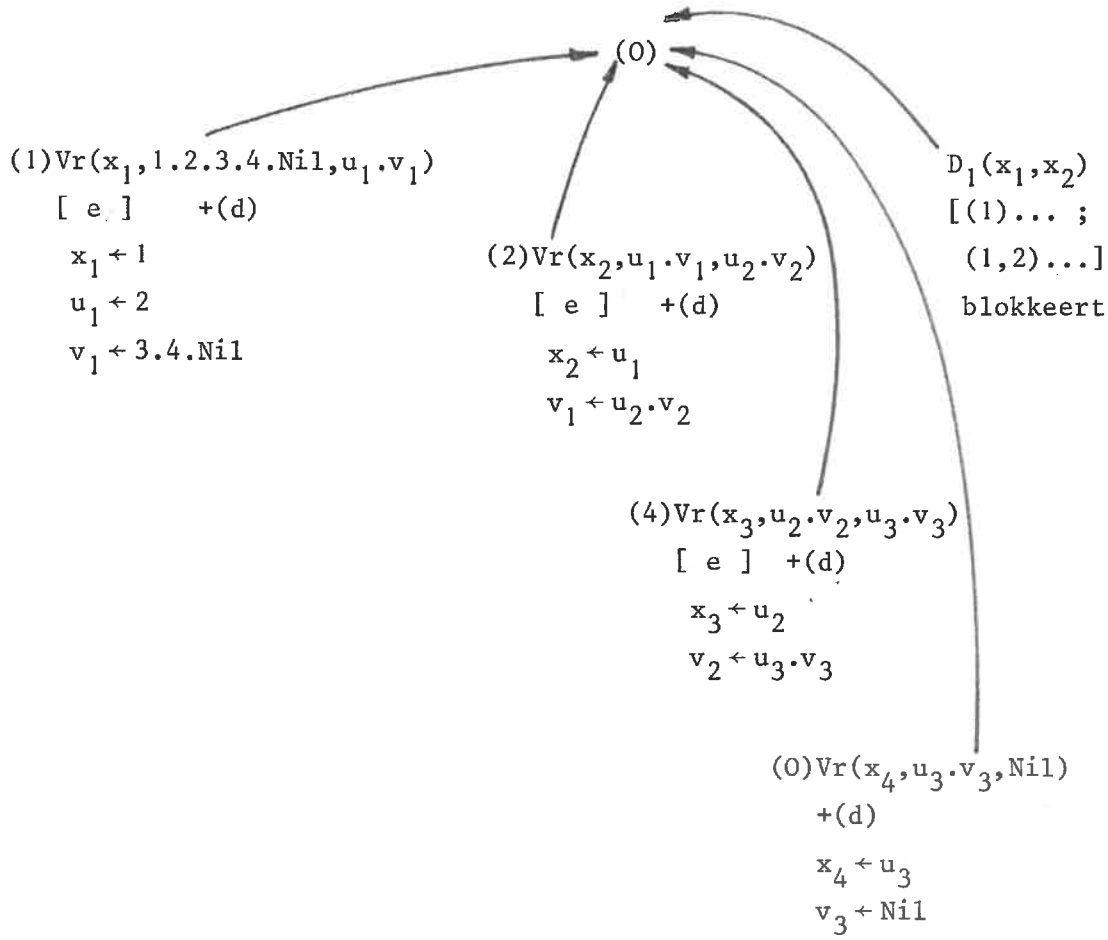


Fig. 4.D.4. : De oproep $Vr(x_4,u_3.v_3,Nil)$ beschikt slechts over één gepaste procedure-definitie en wordt door proces (0) opgeslorpt. De informatie $x_1 \leftarrow 1$ samen met de uit $u_1 \leftarrow 2$ en $x_2 \leftarrow u_1$ afgeleide informatie $x_2 \leftarrow 2$ elimineert alle procedures waarover $D_1(x_1,x_2)$ beschikt. In dit conflict waarbij de processen (0) (1) en (2) betrokken zijn wordt proces (2) de schuldige. Proces (1) (ook (0) maar dit is redundant) schakelt de procedure (d) die door proces (2) gebruikt werd, uit.

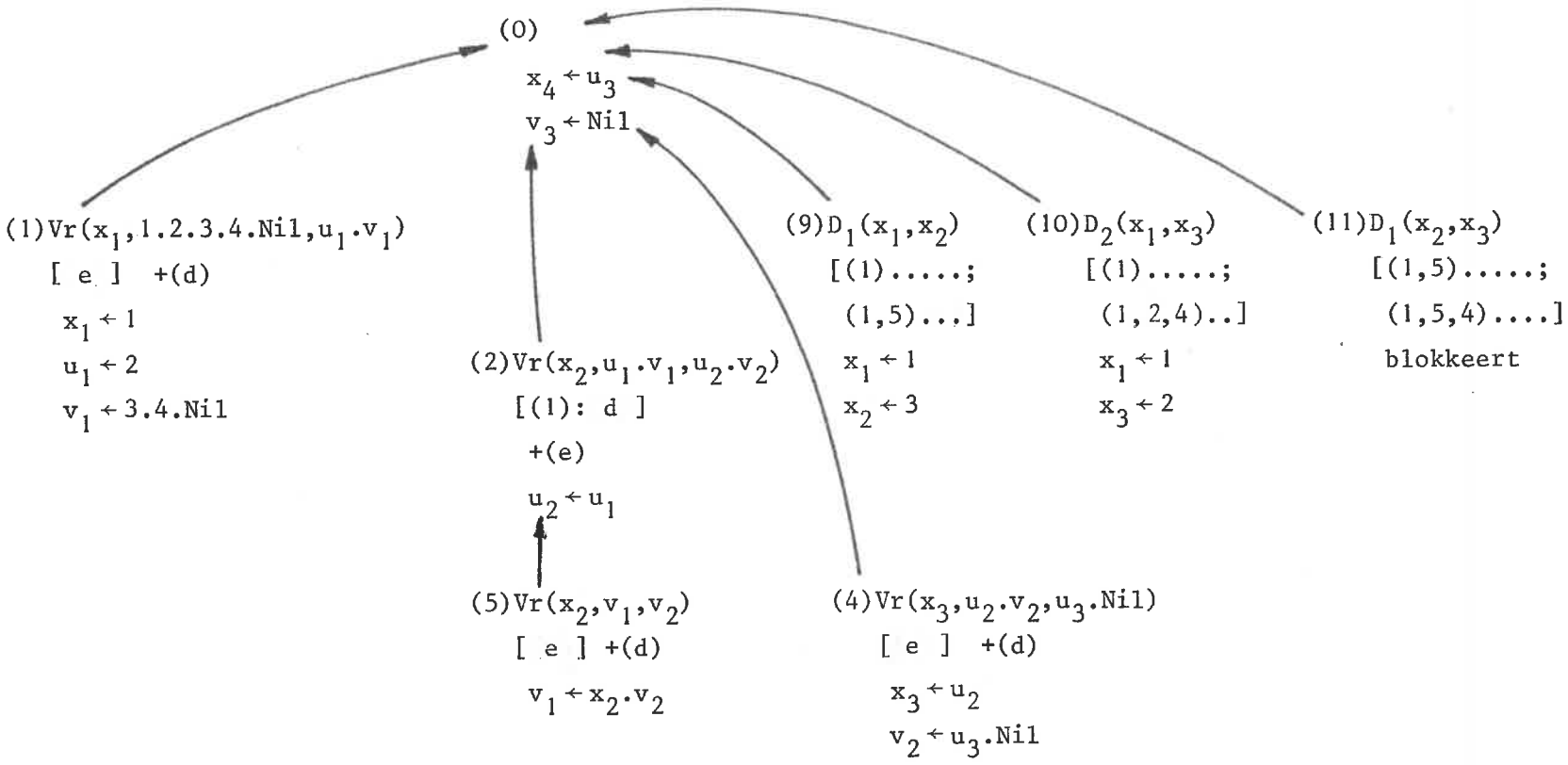


Fig. 4.D.5. : Nu blokkeert de oproep $D_1(x_2,x_3)$. De processen (0), (1) (5) en (4) zijn bij het conflict betrokken. In deze verzameling zijn (5) en (4) de laatsten en dus de verdachten. Volgens de geldende selektieregel wordt (4) de schuldige. Bemerk dat we de oorzaak voor het elimineren van de procedures van (11) zoeken bij de "eerste" producenten van x_2 en x_3, dus niet bij (9) en (10). De herziening van (4) heeft voor gevolg dat (10) en (11) terug beschikken over de procedures die met medewerking van (4) uitgeschakeld werden. De door (4) gebruikte procedure (d) wordt door (1) en (5) uitgeschakeld.

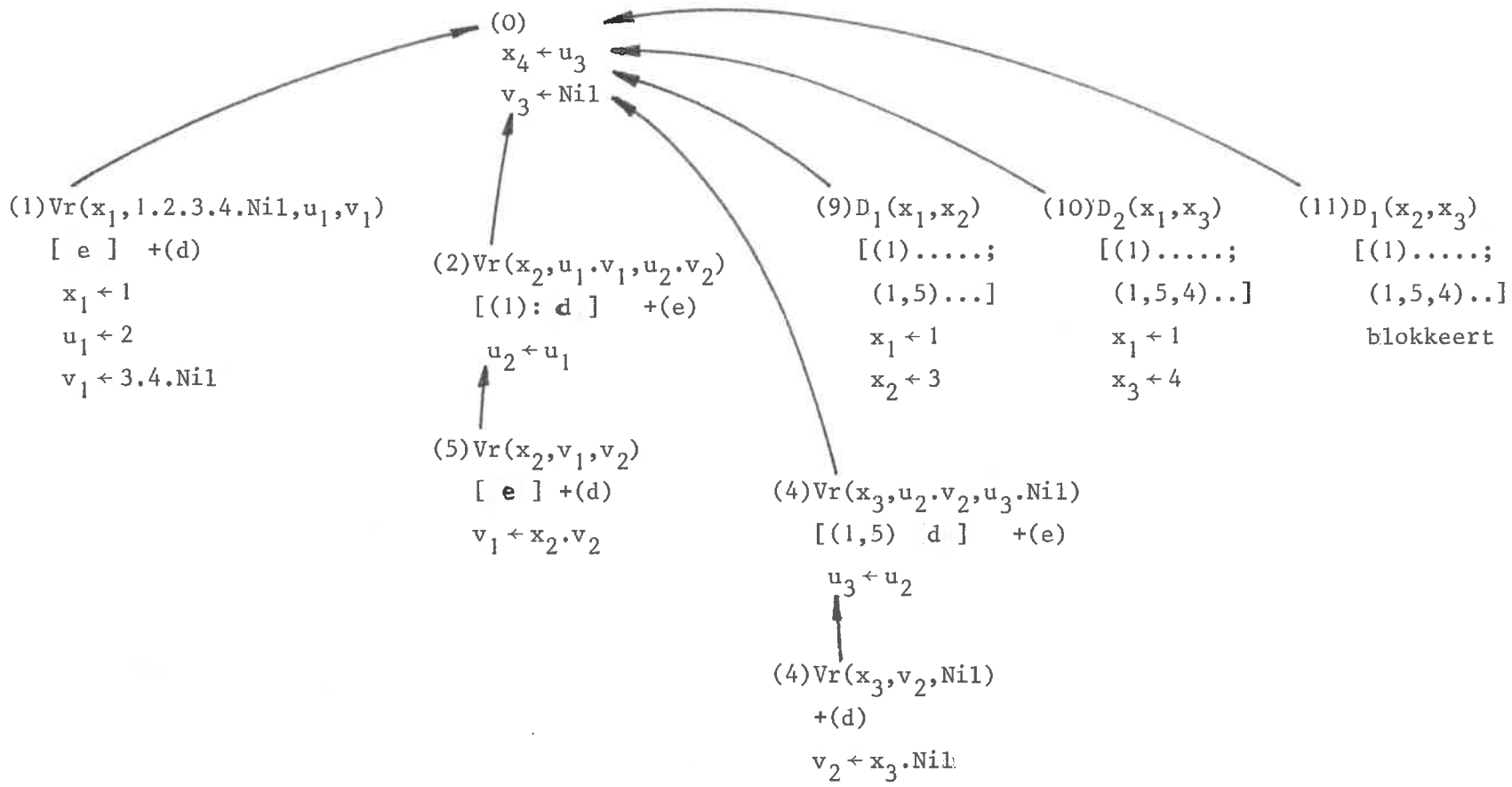


Fig. 4.D.6. : Daar de oproep $Vr(x_3,v_2,Nil)$ slechts over één gepaste procedure beschikt wordt hij door proces (4) opgeslorpt. Uit de bindingen van (1), (5) en (4) kan men $x_3 \leftarrow 4$ afleiden. Dit is in strijd met de binding van proces (10). In dit nieuwe conflict wordt proces (10) als schuldige aangewezen. Gebruik van een andere beschikbare definitie leidt tot $x_1 \leftarrow 1$ en $x_3 \leftarrow 4$. Proces (11) blokkeert opnieuw. De oorzaak ligt bij de processen (0) (1) (5) en (4). Proces (4) is schuldig en zijn procedure (d) wordt door (1) en (5) uitgeschakeld. Daar (4) over geen alternatieven beschikt ontstaat een nieuw conflict tussen (0) (1) en (5). Proces (5) wordt de definitieve schuldige. Bemerk dat het niet nodig is om de door proces (4) uitgevoerde berekening te herzien!

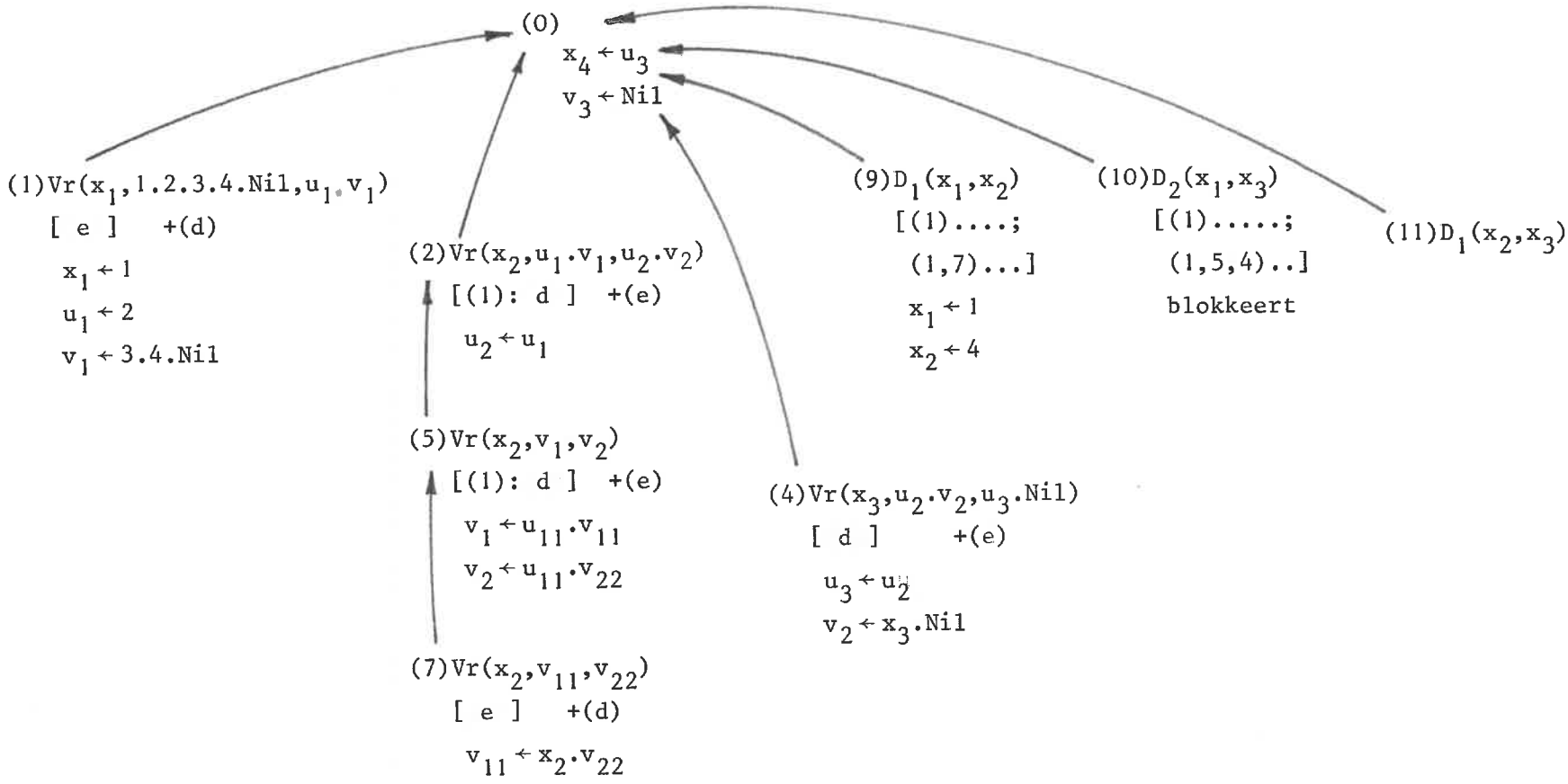


Fig. 4.D.7. : De uitvoering van proces (5) met behulp van procedure (e) leidt tot het ontstaan van proces (7). Uit de informatie van de processen (1), (2), (5), (4) en (7) kan men $x_2 \leftarrow 4$ en $x_3 \leftarrow 3$ afleiden. Deze informatie is in strijd met de informatie van zowel processen (9) als (10). Deze laatste processen worden als schuldige aangewezen. Deze maal zal $D_2(x_1, x_3)$ blokkeren.

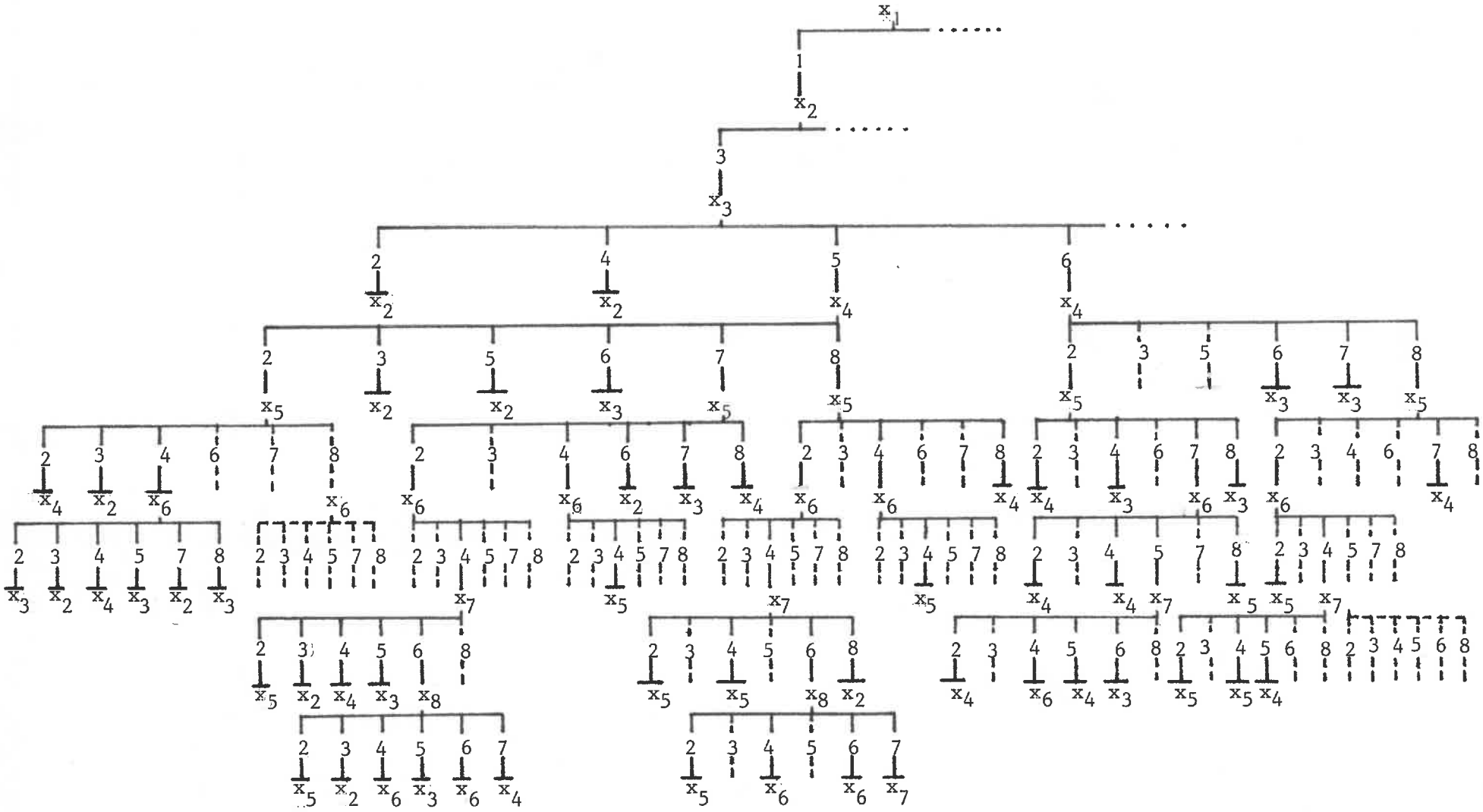


Fig. 4.D.8. : Een deel van de oplossingsruimte van het 8-koninginnenprobleem
$\leftarrow P(x_1), D_1(x_1,x_2), D_2(x_1,x_3), D_1(x_2,x_3), D_3(x_1,x_4)$ ... $D_7(x_1,x_8)$, ... $D_1(x_7,x_8)$.
In streeplijn vindt men de knooppunten die alleen door de naïeve sequentiële exploratietechniek bezocht worden.
Er werd aangeduid aan welke variabelen de conflicten te wijten zijn (vb. : $x_3 \leftarrow 2$ komt in conflict met x_2).

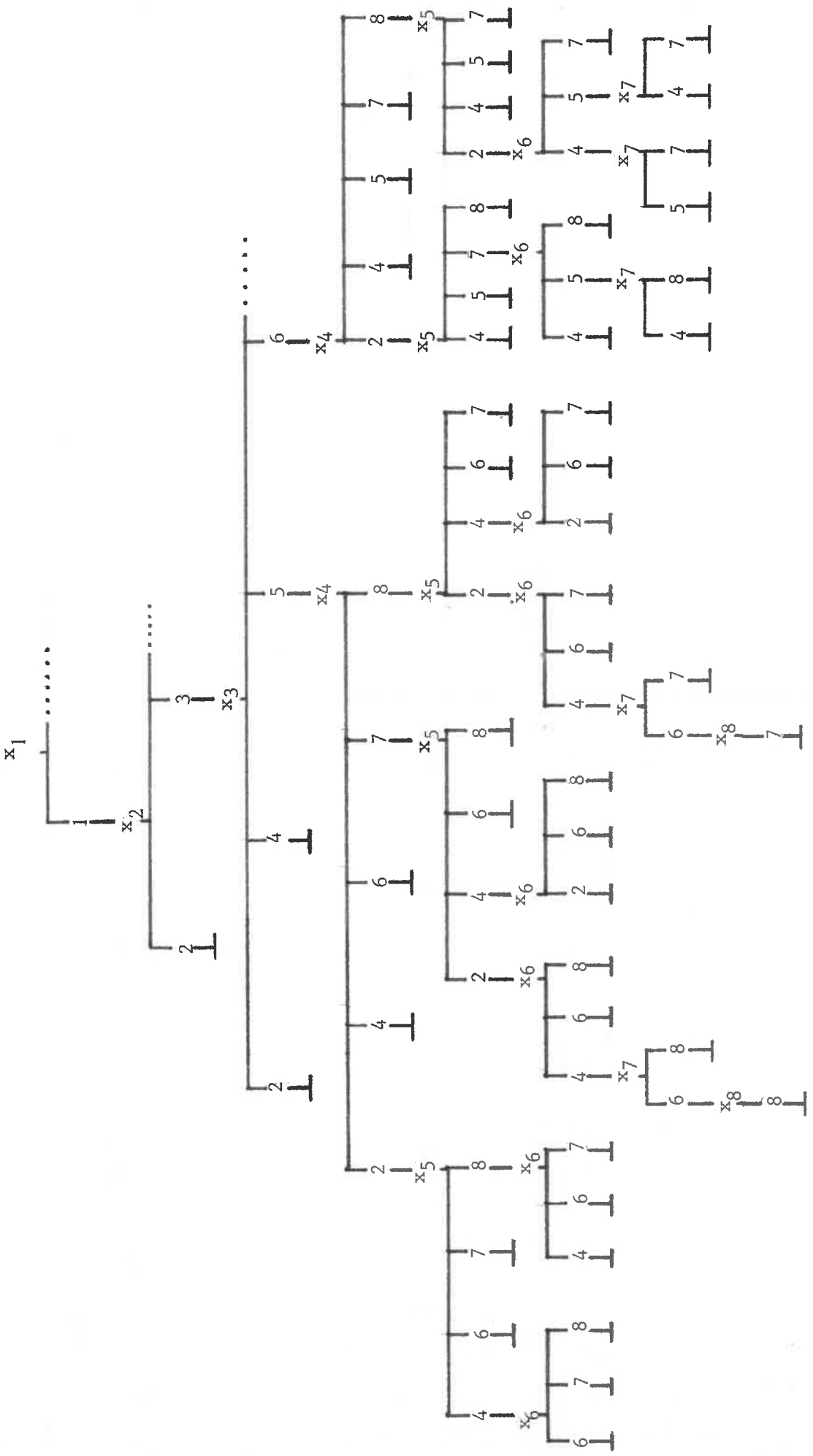


Fig. 4.D.9. : Deel van de oplossingsruimte van het 8-koninginnenprobleem met het permutatieprogramma.

E. DISKUSSIE

De tekortkomingen van de naïeve sequentiële exploratiemethode zijn reeds lang bekend. In het bijzonder werden de combinatorische problemen bestudeerd waar de relaties expliciet gedefinieerd zijn. Alhoewel de tijd nodig voor het oplossen van dergelijke problemen onvermijdelijk exponentieel toeneemt met de complexiteit van het probleem, toch werd geprobeerd om betere methodes te ontwikkelen en de groep problemen die praktisch oplosbaar zijn uit te breiden. We willen hier een tweetal recente methodes aanhalen en vergelijken met onze 'intelligente' sequentiële exploratiemethode.

E.1. DE METHODE VAN MACKWORTH [Mackworth 77]

Als we Mackworth's formulering in Horn-uitdrukkingen omzetten krijgen we :

$$P(x_1,x_2,\ldots,x_n) \leftarrow D_1(x_1),D_2(x_2),\ldots,D_n(x_n),P_1(x_1),\ldots,P_n(x_n),$$
$$P_{12}(x_1,x_2)P_{13}(x_1,x_3),\ldots,P_{n-1,n}(x_{n-1},x_n)$$

Hierin definieert D_i de eindige verzameling waarden die x_i kan aannemen. De unaire predikaten P_i definiëren beperkingen waaraan de x_i moeten voldoen. De binaire predikaten P_{ij} definiëren beperkingen waaraan de paren x_i, x_j moeten voldoen. Zoals reeds gezegd zijn alle relaties gedefinieerd door een opsomming van hun elementen. De elementen van een relatie $P(x_1,x_2,\ldots,x_n)$ die aan alle voorwaarden D_i, P_i en P_{ij} voldoen worden gezocht. De methode van Mackworth bestaat erin dat hij, vooraleer de (naïeve) sequentiële exploratietechniek te gebruiken, de relaties D_i,P_i en P_{ij} probeert te verfijnen door het elimineren van de elementen die niet kunnen uitgebreid worden tot een n-tal dat aan de relatie $P(x_1,\ldots,x_n)$ voldoet. Men kan relaties D_i', P_i' en P_{ij}' als volgt definiëren :

$$D_i'(x_i) \leftarrow P(x_1,\ldots,x_i,\ldots,x_n)$$

$$P_i'(x_i) \leftarrow P(x_1,\ldots,x_i,\ldots,x_n)$$

en

$$P_{ij}'(x_i,x_j) \leftarrow P(x_1,\ldots,x_i,\ldots,x_j,\ldots,x_n)$$

De relatie $P(x_1,\ldots,x_n)$ wordt niet gewijzigd door in haar definitie D_i te vervangen door D_i^1, P_i door P_i^1 en P_{ij} door P_{ij}^1. Echter, omdat de relaties minder elementen bevatten, verloopt de sequentiële exploratie efficiënter. Mackworth probeert die relaties $D_i^!$, $P_i^!$ en $P_{ij}^!$ te benaderen. Daartoe onderzoekt hij een drietal duidelijke gebreken van de sequentiële exploratietechniek en probeert ze te verhelpen door het verfijnen van de relaties D_i, P_i en P_{ij}.

Mackworth neemt aan dat achtereenvolgens $D_1,D_2,\ldots,D_n$ gebruikt worden om $x_1,x_2,\ldots,x_n$ te berekenen en dat de berekening onderbroken wordt van zodra aan een beperking P_i of P_{ij} niet voldaan is. Dan neemt de berekening een nieuwe start vanaf de variabele x_i met hoogste index wiens domein D_i nog niet uitgeput is.

Met deze veronderstellingen onderscheidt hij drie duidelijke gebreken :

a. "Knooppuntconsistentie" ("node-consistency")

Het domein D_i van de variabele x_i bevat waarden die niet voldoen aan de beperking P_i. Voor elke aanvaardbare kombinatie van $x_1,x_2,\ldots,x_{i-1}$ zullen deze waarden aan x_i toegekend worden en zal het conflict met P_i herontdekt worden. Met naïeve sequentiële exploratie wordt het conflict efficiënt opgelost maar het kan gemakkelijk vermeden worden door de relatie D_i te vervangen door een relatie $D_i^!$ die als volgt kan gedefinieerd worden :

$$D_i^!(x_i) \leftarrow D_i(x_i)\ P_i(x_i)$$

Meteen wordt het controleren van de beperking P_i overbodig.

b. "Boogconsistentie" ("arc-consistency")

Voor bepaalde waarden van x_i is er geen enkele waarde van x_j waarvoor voldaan is aan $P_{ij}(x_i,x_j)$. Bij naïeve sequentiële exploratie kan het wegwerken van het conflict zeer onefficiënt gebeuren. Inderdaad, x_i wordt slechts gewijzigd nadat alle mogelijke combinaties van $x_{i+1},\ldots,x_j$ werden beschouwd. Bovendien kan het conflict opnieuw optreden bij elke nieuwe aanvaardbare combinatie van $x_1,\ldots,x_{i-1}$. Mackworth's oplossing bestaat opnieuw in het verfijnen van D_i :

$$D_i^!(x_i) \leftarrow D_i(x_i)\ P_{ij}(x_i,x_j)\ D_j(x_j)$$

(Ook D_j kan verfijnd worden : $D_j^!(x_j) \leftarrow D_j(x_j)\ P_{ij}(x_i,x_j)\ D_i(x_i)$)

c. "Padconsistentie" ("path-consistency")

Bepaalde waarden van x_i en x_j voldoen aan $P_{ij}(x_i,x_j)$ maar er is geen enkele x_k waarvoor zowel aan $P_{ik}(x_i,x_k)$ en $P_{jk}(x_j,x_k)$ voldaan is. Bij naïeve sequentiële exploratie wordt het conflict ontdekt nadat alle waarden van x_k beschouwd werden. x_j wordt echter pas gewijzigd nadat alle mogelijke combinaties van $x_{j+1},\ldots,x_k$ beschouwd werden. Ook dit conflict wordt dus zeer onefficiënt aangepakt en het kan opnieuw verschijnen voor alle aanvaardbare combinaties van $x_1,\ldots,x_{i-1}$. Hier past Mackworth de relatie P_{ij} aan :

$$P'_{ij}(x_i,x_j) \leftarrow P_{ij}(x_i,x_j), D_k(x_k)\ P_{ik}(x_i,x_k)\ P_{jk}(x_j,x_k)$$

De aanpassing van P_{ij} is het gevolg van een "pad" van x_i over x_k naar x_j.

De verfijningen van de relaties onder a. en b. zijn vrij eenvoudig, deze onder c. is complexer en vraagt heel wat rekenwerk.

Nu kunnen we nagaan hoe de intelligente sequentiële exploratietechniek (Algoritme 2) op deze drie conflictsituaties reageert.

a. $\leftarrow D_i(x_i)P_i(x_i)$ ...

Alle procedures waarover $P_i(x_i)$ beschikt worden door $D_i(x_i)$ uitgeschakeld. Het blokkeren van P_i is dus uitsluitend aan D_i te wijten en de door D_i gebruikte procedure wordt definitief uitgeschakeld. In andere woorden, de betreffende waarden voor x_i worden slechts eenmaal beschouwd en worden definitief uitgeschakeld. We bereiken dus hetzelfde resultaat als Mackworth.

b. $\leftarrow \ldots D_i(x_i)\ldots D_j(x_j)\ P_{ij}(x_i,x_j)$

Alle procedures waarover $P_{ij}(x_i,x_j)$ beschikt worden door D_i uitgeschakeld. Het gevolg is terug dat de door D_i gebruikte procedure definitief uitgeschakeld wordt. De symmetrische situatie waarbij voor bepaalde waarden van x_j niet aan P_{ij} kan voldaan worden zal door een eenvoudig systeem niet zo goed afgehandeld worden. Meestal zal het blokkeren van P_{ij} zowel aan D_i als aan D_j toegeschreven worden en zal de door D_j gebruikte procedure door D_i uitgeschakeld worden. Zodra $D_i(x_i)$ herzien wordt komt de betreffende waarde voor x_j terug beschikbaar. Het conflict kan meermaals optreden, maar wordt efficiënt behandeld.

c. $\leftarrow \ldots D_i(x_i) \ldots D_j(x_j)\ P_{ij}(x_i,x_j) \ldots D_k(x_k)\ P_{ik}(x_i,x_k), \ldots, P_{jk}(x_j,x_k)$

Een aantal van de procedures waarover P_{ik} beschikt zullen door D_i uitgeschakeld worden en komen pas opnieuw aan bod nadat D_i herzien wordt. Analoog worden een aantal van de procedures waarover P_{jk} beschikt door D_j uitgeschakeld. Wat de overblijvende procedures betreft kunnen zich twee gevallen voordoen :

1. De rest van de procedures waarover P_{ik} beschikt worden door D_k uitgeschakeld. Dit betekent dat D_i en D_k samen alle procedures waarover P_{ik} beschikt, uitschakelen. Er is dus een conflict tussen D_i en D_k en de door D_k gebruikte procedure wordt door D_i uitgeschakeld. Ze komt pas terug aan bod nadat D_i herzien wordt.
2. Slaagt de test op P_{ik}, dan blokkeert P_{jk}. In dit geval zorgen D_j en D_k samen voor de uitschakeling van alle procedures waarover P_{jk} beschikt. Het gevolg is dat de door D_k gebruikte procedure door D_j uitgeschakeld wordt.

Alle procedures waarover D_k beschikt worden dus ofwel door D_i ofwel door D_j uitgeschakeld. Dit leidt tot een conflict tussen D_i en D_j en de door D_j gebruikte procedure wordt door D_i uitgeschakeld. Meteen komen de door D_j uitgeschakelde procedures terug ter beschikking. Dit effekt fekt wordt minder scherp wanneer tussen de testen P_{ik} en P_{jk} een derde test $P_{\ell k}$ komt en deze tot gevolg heeft dat $D_\ell(x_\ell)$ bepaalde waarden van x_k uitschakelt. Toch kunnen we stellen dat de conflicten vrij behoorlijk opgevangen worden. Dit is vooral te danken aan het feit dat bij een nieuwe start (bv. wijzigen van x_j) slechts een gedeelte van het domein van de variabelen met hogere index (x_k) opnieuw moet onderzocht worden.

We kunnen besluiten dat de intelligente sequentiële exploratietechniek de conflicten van type a en b goed en deze van type c vrij behoorlijk opvangt. De methode behandelt slechts de conflicten die werkelijk optreden terwijl een herhaling van hetzelfde conflict niet uitgesloten is. De methode van Mackworth gaat op zoek naar alle potentiële conflicten van de drie hogergenoemde types en rekent er een en voorgoed mee af. Vooral het controleren van padconsistentie is een erg kostelijke operatie en leidt soms tot niets omdat het gestelde probleem padconsistent is.

Beschouwen we bijvoorbeeld het n-koninginnenprobleem. Zij x_i de kolompositie van de koningin op de ide rij dan voldoet de formulering

$$\leftarrow D(x_1) \ldots D(x_n)\ P_{12}(x_1,x_2) \ldots P_{1n}(x_1,x_n) \ldots P_{n-1\ n}(x_{n-1},x_n)$$

waarin D het domein {1,...,n} van de waarden x_i en $P_{ij}(x_i,x_j)$ de toegelaten combinaties x_i-x_j beschrijft, voor n=8 aan alle door Mackworth gestelde consistentie-eisen.

Mackworth's methode is dus slechts nuttig voor problemen met een groot aantal padinconsistenties. In alle geval is het steeds voordelig om, nadat de probleemstelling padconsistent gemaakt is, de intelligente sequentiële exploratie te gebruiken in plaats van de naïeve. Zelfs in problemen die heel wat padinconsistenties bevatten kan aan het nut van Mackworth's methode getwijfeld worden wanneer het probleem vele oplossingen heeft en men slechts in één oplossing belang stelt. De winst aan efficiëntie bij de sequentiële exploratie weegt vermoedelijk niet op tegen het vele werk dat nodig is om het probleem padconsistent te maken. Wellicht is het voordeliger om onmiddellijk met de intelligente sequentiële exploratie te starten en opportunistisch naar een eerste oplossing te zoeken.

E.2. DE METHODE VAN FREUDER [Freuder 78]

Freuder heeft niet alleen unaire en binaire beperkingen maar ook beperkingen van hogere orde. Dus naast P_i en P_{ij} ook P_{ijk},... . Opnieuw wordt gezocht naar de n-tallen $x_1,\ldots,x_n$ die aan alle beperkingen voldoen. De methode van Freuder kan als volgt uitgelegd worden : (Deze logische formulering maakt de correctheid van de methode triviaal!).
We definiëren de volgende relaties :

$$Q_i(x_i) \leftarrow P_i(x_i)$$
$$Q_{ij}(x_i,x_j) \leftarrow P_{ij}(x_i,x_j), Q_i(x_i), Q_j(x_j)$$
$$Q_{ijk}(x_i,x_j,x_k) \leftarrow P_{ijk}(x_i,x_j,x_k), Q_{ij}(x_i,x_j), Q_{ik}(x_i,x_k)\ Q_{jk}(x_j,x_k)$$

$$\begin{aligned} Q_{12\ldots n}(x_1,\ldots,x_n) \leftarrow\ & P_{12\ldots n}(x_1,\ldots,x_n) \\ & Q_{23\ldots n}(x_2,x_3,\ldots,x_n) \\ & Q_{13\ldots n}(x_1,x_3,\ldots,x_n) \\ & \vdots \\ & Q_{12\ldots n-1}(x_1,x_2,\ldots,x_{n-1}) \end{aligned}$$

(indien de beperking $P_{i,\ldots,k}$ niet bestaat verdwijnt ze uit het rechterlid). Het is duidelijk dat de relatie $Q_{1\ldots n}(x_1,\ldots,x_n)$ de gevraagde relatie definieert. Wanneer we systematisch elke oproep van Q door de definitie vervangen bekomen we terug de oorspronkelijke beperkingen (dezelfde beperking zal meermaals voorkomen). Dit logisch programma op de conventionele manier uitvoeren is natuurlijk zeer onefficiënt. Eenzelfde deelprobleem verschijnt meermaals en wordt dus meermaals opgelost. Freuder echter vertrekt niet van de probleemstelling maar van de gegevens. Hij berekent eerst alle elementen van de relaties Q_i en gebruikt ze dan om de relaties Q_{ij} te berekenen. Op hun beurt worden de relaties Q_{ij} gebruikt om de relaties Q_{ijk} te berekenen. Dit kan zo doorgaan totdat de relatie $Q_{1\ldots n}$ bekomen wordt. Hij past echter nog een belangrijke optimisatie toe. Een paar x_i-x_j is slechts interessant als het kan uitgebreid worden tot een oplossing $x_1\ldots x_n$. Van zodra hij ontdekt dat dit niet mogelijk is doet Freuder het uit de relatie Q_{ij} verdwijnen. Hetzelfde geldt voor alle andere hulprelaties. Het principe is als volgt : Eens de relatie Q_{ijk} berekend gebruikt hij deze relatie om de relatie Q_{ij} (ook Q_{ik} en Q_{jk}) te verfijnen. Hij definieert :

$$Q'_{ij}(x_i,x_j) \leftarrow Q_{ij}(x_i,x_j)\ Q_{ijk}(x_i,x_j,x_k).$$

Vervangen we hierin Q_{ijk} door zijn definitie, dan bekomen

$$Q'_{ij}(x_i,x_j) \leftarrow Q_{ij}(x_i,x_j)P_{ijk}(x_i,x_j,x_k)Q_{jk}(x_j,x_k)Q_{ik}(x_i,x_k)$$

Een verdere vervanging van definities leidt tot :

$$Q'_{ij}(x_i,x_j) \leftarrow P_i(x_i),P_j(x_j)P_k(x_k)P_{ij}(x_i,x_j)P_{ik}(x_i,x_k)P_{jk}(x_j,x_k)P_{ijk}(x_i,x_j,x_k)$$

Dit is niets anders dan de eis tot padconsistentie. Deze uitwerking maakt ook duidelijk dat de vervanging van Q_{ij} door Q'_{ij} aan de relatie $Q_{1,2,\ldots,n}(x_1,\ldots,x_n)$ geen nieuwe eisen oplegt.

Deze optimisatie wordt systematisch doorgevoerd en telkens een nieuwe relatie (k argumenten) berekend of een bestaande relatie verfijnd werd wordt ze gebruikt om alle relaties in k-1 argumenten te verfijnen en om de relaties in k+1 argumenten te (her) berekenen. Zoals we gezien hebben resulteert het gebruik van de relaties in 3 argumenten om de relaties in 2 argumenten te verfijnen, in padconsistentie. Voor relaties in meer argumenten resulteert het in meer complexe vormen van consistentie.

Gaan we even na wat de methode precies doet bij het 8-koninginnenprobleem. De beperkingen $P_i(x_i)$ bepalen de mogelijke kolomposities en de beperkingen $P_{ij}(x_i,x_j)$ bepalen de mogelijke paren x_i-x_j. Men begint met het berekenen van de relaties Q_{ijk} (dit zijn niet minder dan 56 relaties en elke relatie bevat ruim 100 tot ruim 200 elementen!). Tevens wordt padconsistentie gecontroleerd. Dit betekent in feite dat men nagaat of elke configuratie x_i-x_j kan uitgebreid worden met een koningin in willekeurige kolom k (voor alle k verschillend van i en j). Daar dit het geval is leidt dit tot geen enkele verfijning. Daarna worden de relaties $Q_{ijk\ell}$ berekend (niet minder dan 70 relaties). De pogingen om de relaties Q_{ijk} te verfijnen zal nu wel tot uitschakeling van enkele elementen leiden (Er zijn configuraties (x_i,x_j,x_k) zó dat voor bepaalde ℓ geen enkele configuratie (x_i,x_j,x_k,x_ℓ) aanvaardbaar is). We zien dat men systematisch alle mogelijke manieren berekent om 3, 4, 5 ... 8 koninginnen te plaatsen. Bovendien, van zodra men ontdekt dat een bepaalde configuratie niet met een willekeurige kolom kan uitgebreid worden doet men ze verdwijnen. De methode zoekt alle oplossingen in parallel en het voorbeeld suggereert dat ze zeer veel geheugen vraagt. Dit maakt ze alvast weinig aantrekkelijk wanneer het probleem vele oplossingen heeft en men slechts in één oplossing belang stelt. In dergelijke gevallen lijkt de intelligente sequentiële exploratie verkieselijk.

Het is leerrijk om de formulering van Freuder te vergelijken met de volgende (we veronderstellen alleen binaire beperkingen P_{ij}) :

$$Q_{12}(x_1,x_2) \leftarrow P_{12}(x_1,x_2)$$
$$Q_{123}(x_1,x_2,x_3) \leftarrow Q_{12}(x_1,x_2)\ P_{13}(x_1,x_3)\ P_{23}(x_2,x_3)$$
$$Q_{1234}(x_1,x_2,x_3,x_4) \leftarrow Q_{123}(x_1,x_2,x_3)\ P_{14}(x_1,x_4)\ \ldots\ P_{34}(x_3,x_4)$$

$$Q_{12\ldots n}(x_1,\ldots,x_n)\ \ Q_{1\ldots n-1}(x_1,\ldots,x_{n-1})\ P_{1n}(x_1,x_n)\ \ldots\ P_{n-1n}(x_{n-1},x_n)$$

Vervangen we in de definitie van $Q_{1,\ldots,n}$ alle oproepen Q door hun definitie, dan bekomen we terug de originele probleemstelling. Bereken we achtereenvolgens de relaties $Q_{12},\ldots,Q_{123}$, dan bekomen we, eveneens in parallel alle oplossingen. De oplossingsruimte is identiek dezelfde als bij de naïeve sequentiële exploratie. Wat betreft de complexiteit kan de berekening van een relatie $Q_{1\ldots m}(x_1,\ldots,x_n)$ best de vergelijking doorstaan met de be-

rekening van een relatie $Q_{i1,\ldots,im}(x_{i1},\ldots,x_{im})$ bij Freuder. (Zelfde aantal maar eenvoudiger termen in het rechterlid, zeker voor m klein t.o.v. n bevatten de relaties P_{ij} minder elementen dan de relaties $Q_{i1,\ldots in-1}$) Freuder berekent echter niet een maar $\frac{n}{(n-m)!m!}$ dergelijke relaties! Om dit extra werk goed te maken moet het verfijningsproces de relaties wel drastisch beperken. In problemen zoals het 8-koninginnenprobleem dat padconsistent is komt het verfijningsproces eerder laat op gang en kan betwijfeld worden of het extra werk wel goedgemaakt wordt. Vergeten we daarenboven niet dat deze vergelijking de naïeve sequentiële exploratie betreft. Ook de intelligente sequentiële exploratie vangt, zoals we toonden, padinconsistenties vrij behoorlijk op. In plaats van alle potentiële conflicten systematisch op te zoeken en uit te schakelen behandelt de intelligente sequentiële exploratie de conflicten zoals ze zich voordoen. Bij complexe conflicten maakt de context de conclusies minder diepgaand en daardoor is het niet onmogelijk dat eenzelfde conflict terug verschijnt, maar toch wordt steeds voorkomen dat men een conflict probeert op te lossen door het veranderen van totaal irrelevante variabelen. Tevens wordt voorkomen dat eenzelfde waarde terug toegekend wordt (dezelfde procedure terug gebruikt wordt) wanneer alle factoren die tot de uitschakeling van die waarde bijdroegen nog steeds aanwezig zijn. Op deze manier wordt de irriterende "dwaze" gedrag van de naïeve sequentiële exploratie voorkomen.

Tot slot mogen we niet vergeten dat de sequentiële exploratietechniek ruimer toepasbaar is. Relaties kunnen impliciet gedefinieerd zijn onder de vorm van procedures. Denken we maar aan de oplossing van het koninginnenprobleem door middel van het permutatieprogramma. Daar drukten we, door middel van een procedure, uit dat $x_1,\ldots,x_n$ en permutatie moeten zijn van de getallen $1,\ldots,n$. Dit leidt tot een drastische beperking van het aantal kandidaatoplossingen. Een dergelijke beperking is niet mogelijk bij de methodes van Mackworth en Freuder.

F. HET VINDEN VAN ALLE OPLOSSINGEN

Wil men alle oplossingen vinden, dan mag men het zoekproces niet stopzetten nadat een eerste oplossing gevonden is. Men kan een nieuwe oplossing vinden door de huidige oplossing te verwerpen : door te stellen dat er een (kunstmatig) conflict is tussen alle processen die tot de huidige

oplossing bijdragen. Het systeem zal dan, rekening houdende met de bestaande partiële orde, een "schuldige" aanwijzen. De door dit schuldige proces gebruikte procedure wordt door alle andere processen uitgeschakeld. Het conflict zal zich natuurlijk herhalen tot dat de aangewezen "schuldige" over alternatieve procedures beschikt. In feite is men alzo bezig om al de processen die tot een oplossing bijdragen op een kunstmatige wijze totaal te ordenen.

Dit heeft grote nadelen wanneer het probleem in onafhankelijke deelproblemen uiteenvalt. Bijvoorbeeld wanneer er twee onafhankelijke oproepen P(x) en Q(y) ontstaan. Veronderstel dat Q m oplossingen heeft. Het kunstmatige conflict heeft tot gevolg dat P tussenkomt in het uitschakelen van de procedures waarover Q beschikt. Eens alle m oplossingen afgehandeld zal P als schuldige aangewezen worden, zal voor P een nieuwe oplossing gezocht worden en zal het systeem opnieuw beginnen met het zoeken van de oplossingen voor Q. Daar Q niets met P te maken heeft zal het systeem dezelfde m oplossingen vinden. Heeft P n oplossingen dan zal het systeem n keren de m oplossingen van Q berekenen.

We kunnen dit vermijden door het deelprobleem Q afzonderlijk te beschouwen en eerst alle oplossingen voor dit deelprobleem te berekenen. Daar we nu alleen het deelprobleem Q beschouwen, blijft het kunstmatige conflict beperkt tot de processen die tot Q behoren. (Dit sluit niet uit dat, bij het zoeken van nieuwe oplossingen, er echte conflicten ontstaan met informatie uit processen die niet tot Q behoren). Werden de m oplossingen van Q gevonden, dan kan men Q vervangen door één enkel proces Q wiens oplossingen gegeven zijn door m lemma's (één lemma per oplossing). Deze vervanging blijft geldig zolang de processen wiens informatie gebruikt werd tijdens het zoeken van de oplossingen van Q, niet herzien worden. Zoeken we nu alle oplossingen van een ruimer probleem dat zowel P als Q omvat, dan begint men met Q' als schuldige aan te wijzen, Q' doet beroep op al zijn lemma's en dit levert m oplossingen op. Daarna wordt P als schuldige aangewezen. Voor P wordt een andere oplossing gezocht en opnieuw wordt Q' opgelost door middel van de m lemma's. Op deze wijze vindt men op een efficiënte wijze alle oplossingen voor P en Q. Wordt in een volgende stap een proces R als schuldige aangewezen, en werd de informatie van proces R tijdens het afleiden van de m oplossingen van Q gebruikt, dan worden de lemma's ongeldig en moet Q terug door zijn oorspronkelijke formulering vervangen worden.

Een eenvoudig voorbeeld

(a) P(ℓ,p(x,y)) ← Split(ℓ,ℓ1,ℓ2) Member(x,ℓ1) Member(y,ℓ2)

(b) Split(Nil,Nil,Nil) ←

(c) Split(x.Nil,x.Nil,Nil) ←

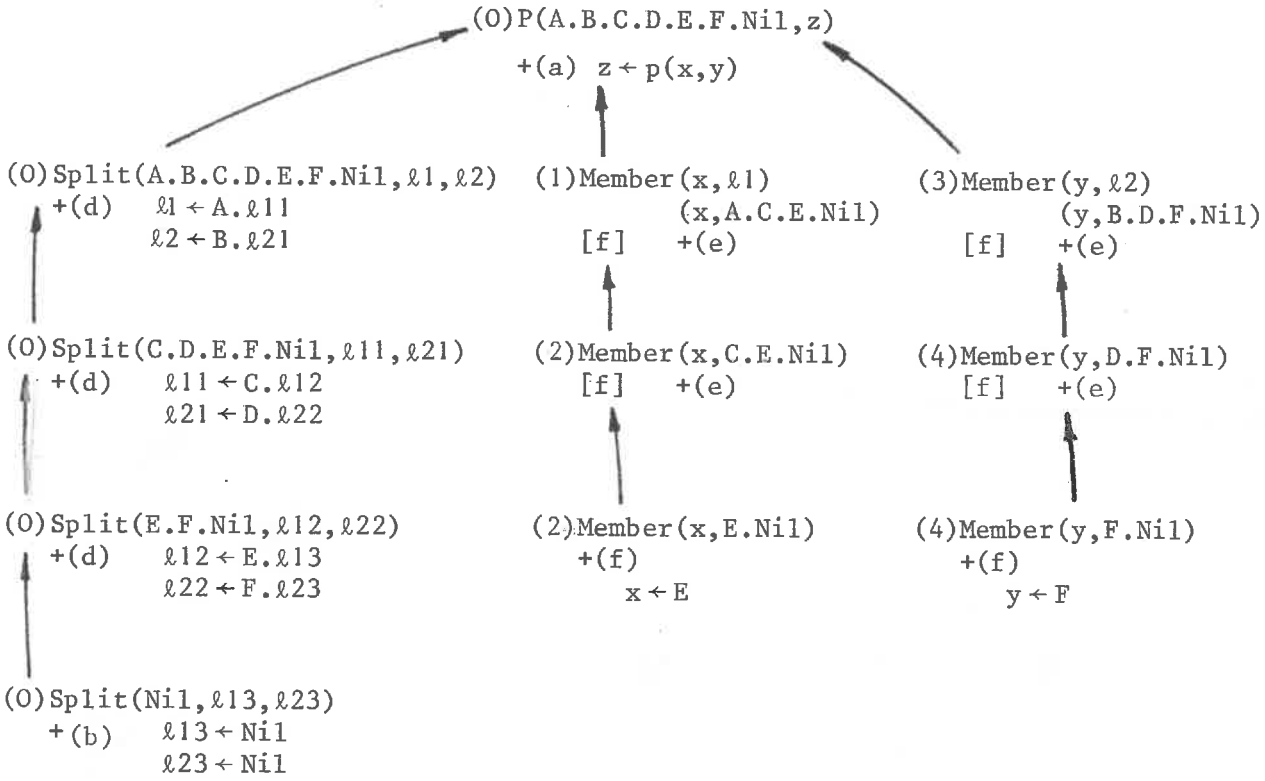
(d) Split(x.y.ℓ,x.ℓ1,y.ℓ2) ← Split(ℓ,ℓ1,ℓ2)

(e) Member(x,y.z) ← Member(x,z)

(f) Member(x,x.y) ←

←P(A.B.C.D.E.F.Nil,z)

Een eerste oplossing :



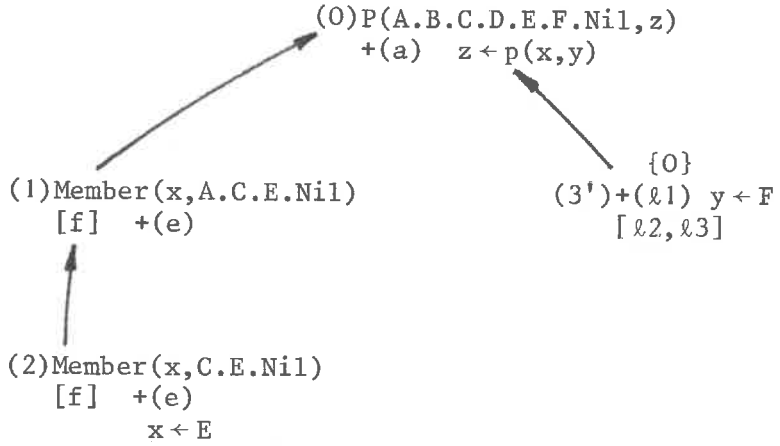
De bestaande partiële orde laat toe om een totale orde te kiezen die aanleiding geeft tot de opeenvolgende probleemstellingen :

←Member(x,A.C.E.Nil) Member(y,B.D.F.Nil)

←Member(y,B.D.F.Nil)

terwijl een andere totale orde uiteindelijk aanleiding geeft tot ←Member(x,A.C.E.Nil). De huidige oplossingen van beide deelproblemen zijn onafhankelijk van elkaar en beide deelproblemen hebben eventueel meerdere oplossingen (er zijn alternatieve procedures beschikbaar).

We beschouwen het deelprobleem Member(y,B.D.F.Nil) afzonderlijk. Het heeft drie oplossingen of "lemma's" : (ℓ1):y←F (ℓ2):y←D en (ℓ3):y←B. De geldigheid van deze oplossingen hangt alleen van proces (0) af. We kunnen nu het probleem herleiden tot : (de lijst {...} duidt aan van welke processen de geldigheid der lemma's afhangt.



Om alle oplossingen te vinden wijzen we eerst proces (3') als schuldige aan. Dit resulteert in de oplossingen z←p(E,F), z←p(E,D) en z←p(E,B). Daarna wordt proces (2) als schuldige aangewezen. Het gebruik van de lemma's resulteert nu in z←p(C,F), z←p(C,D) en z←p(C,B). Daarna wordt proces (1) als schuldige aangewezen. De lemma's zijn nog steeds geldig en we vinden z←p(A,F), z←p(A,D) en z←p(A,B). In de volgende stap wordt proces (0) als schuldige aangewezen, meteen worden de lemma's ongeldig (daar het hier gaat om het vaderproces verdwijnen de deelproblemen volledig). Daar het proces (0) over geen alternatieve procedures beschikt eindigt de berekening.

Een tweede voorbeeld

We beschouwen een programma dat alle binaire bomen met een bepaald profiel berekent. Het profiel stellen we voor door een lijst van elementen, een binaire boom met deelbomen x en y door x⁂y.
Programma :

(a) Tree(x.Nil,x) ←
(b) Tree(x.y.z,u⁂v) ←Append(s,t,x.y.z) Tree(s,u) Tree(t,v)
(c) Append(Nil,x,x) ←
(d) Append(x.u,v,x.w) ←Append(u,v,w)

←Tree(A.B.C.D.E.F.Nil,z)

De berekening van een eerste oplossing is gegeven in fig. 4.F.1. Bij het uitvoeren van de oproep Append(s1,t1,A.B.C.D.E.F.Nil) hebben we de definities zodanig gemanipuleerd dat de lijst opgesplitst werd in twee lijsten van drie elementen. De oproep Tree(s1,u1) hebben we niet uitgewerkt. De uitwerking is analoog als deze van Tree(t1,v1). De gevonden oplossing is : z←(A×(B×C)) × (D×(E×F)).

Bestuderen we de struktuur van de oplossing, dan hebben we (21) : Tree(s18,u18) en (22) : Tree(t18,v18) als onafhankelijke problemen, ze beschikken echter niet over alternatieve procedures en het loont dus niet de moeite om afzonderlijk hun oplossingen te berekenen. We moeten dus meeromvattende deelproblemen beschouwen. Dit brengt ons bij (17) : Tree(s1,u14) en (18) : Tree(t14,v14). (18) heeft eventueel meerdere oplossingen (proces (20)), (17) heeft echter slechts één oplossing en dit maakt het onnodig om (18) afzonderlijk op te lossen. In een volgende stap komen (5) Tree(s1,u1) en (14) Tree(t1,v1) in aanmerking. Het deelprobleem Tree(t1,v1) heeft eventueel meerdere oplossingen (de processen (16) en (20)), zo ook (5).

Het kan dus lonend zijn om alle oplossingen van het deelprobleem Tree(t1,v1) te berekenen. (Het is in principe niet uitgesloten dat we bij het zoeken van andere oplossingen in conflict komen met een proces die voorkomt in de oplossingen van (5). Dit beperkt dan de geldigheidsduur van de lemma's). De huidige oplossing (21) bestaat uit t1←x14.y14.y18.Nil, v1←x14×(y14×y18). De oplossing hangt af van proces (4). Het creëren van een kunstmatig conflict zal leiden tot het herzien van proces (20). Het gebruik van procedure(d) zal echter niet tot een nieuwe oplossing leiden. Daarna wordt proces (16) herzien. Dit leidt tot een tweede oplossing (ℓ2) t1←x14.y14.y18.Nil,v1←(x14×y14)×y18. Dit is meteen de laatste oplossing. De lemma's blijven geldig zolang proces (4) niet herzien wordt.

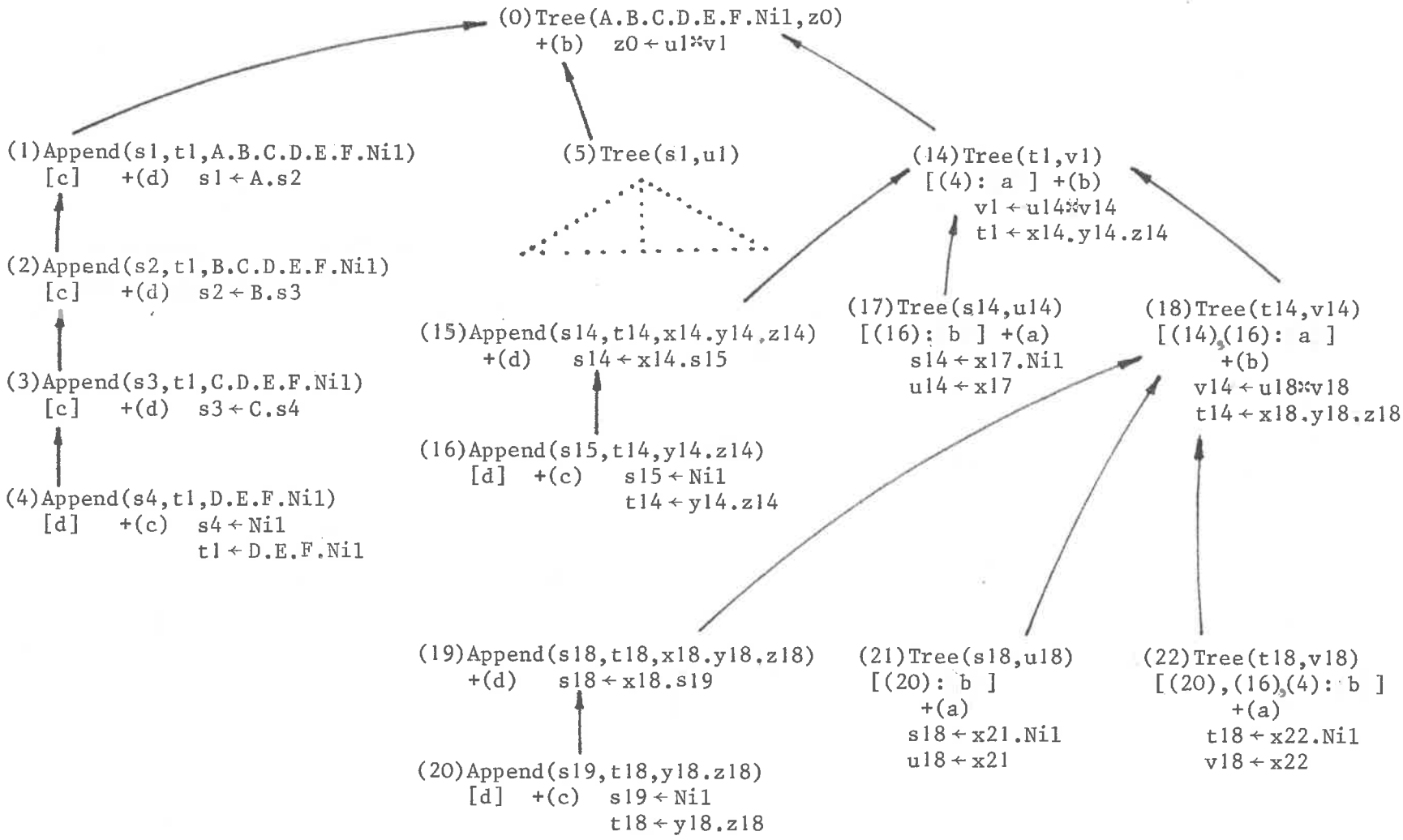
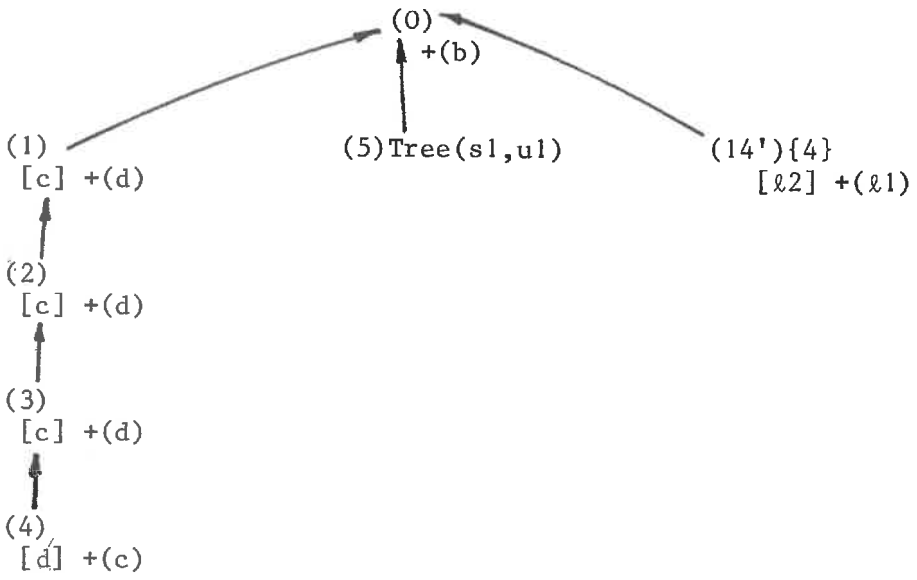


Fig. 4.F.1. : Een eerste oplossing voor Tree.

Het probleem herleidt zich nu tot :



Gezien er niet meer onafhankelijke deelproblemen zijn zoeken we nu de oplossingen van het globale probleem. Een eerste oplossing bestaat uit $z0 \leftarrow (A \times (B \times C)) \times (D \times (E \times F))$. Het aanwijzen van (14') als schuldige leidt tot het gebruik van het tweede lemma en tot de oplossing $z0 \leftarrow (A \times (B \times C)) \times ((D \times E) \times F)$. Daarna werden de processen uit Tree(s1,u1) als schuldige aangewezen. Dit leidt tot een nieuwe oplossing van dit deelprobleem. Deze wordt terug met beide lemma's gecombineerd en resulteert in $z0 \leftarrow ((A \times B) \times C) \times (D \times (E \times F))$ en $z0 \leftarrow ((A \times B) \times C) \times ((D \times E) \times F)$. Daar Tree(s1,u1) geen andere oplossingen meer heeft wordt nu proces (4) als schuldige aangewezen en moet dus herzien worden. Meteen worden de lemma's ℓ1 en ℓ2 ongeldig en moet terug het volledige deelprobleem Tree(t1,v1) beschouwd worden. Telkens een nieuwe oplossing gevonden is voor het globale probleem (na herziening van (4)) of voor een deelprobleem kan men natuurlijk proberen om onafhankelijke deelproblemen te isoleren om alzo het proces van het vinden van alle oplossingen (voor het globale- of voor een deelprobleem) te versnellen.

Bij het zoeken van alle oplossingen voor problemen die in onafhankelijke deelproblemen uiteenvallen kan deze beperkte vorm van parallelisme (parallellisme in de zin dat men eerst alle oplossingen berekent van welgekozen deelproblemen en dan pas één voor één de oplossingen van het globale probleem) aanleiding geven tot een aanzienlijke versnelling van het zoekproces.

G. NABESCHOUWINGEN

Onze belangstelling voor de problemen van de naïeve sequentiële exploratietechniek werd gewekt door onze contacten met Robert Kowalski. Deze laatste had, voor enkele M Sc studenten, een project opgezet dat zich tot doel stelde de sequentiële exploratietechniek voor programma's in de logika der Horn-uitdrukkingen te verbeteren ([Hill 76],[Warner 77]). Het probleem was reeds lang gesteld [Sussman & McDermott 72] maar de pogingen om eraan te verhelpen waren tot dan toe weinig bevredigend (o.a. Sussman's HACKER systeem [Sussman 73]). Ook dit project had weinig resultaat. Naderhand werd de auteur er zich van bewust dat het project faalde omdat men zich blindstaarde op de staat van de berekening en men geen rekening hield met de vroeger opgetreden conflicten (zie ons voorbeeld van sektie B).

Langzamerhand realiseerde de auteur zich dat de sleutel tot het probleem erin bestaat de berekening op te splitsen in deeltaken die, zoveel als mogelijk, onafhankelijk van elkaar uitgevoerd worden (Een eerste minder algemene versie staat in [Bruynooghe 78]).

Alhoewel deze idee hier enkel uitgewerkt is voor programma's in de logika der Horn-uitdrukkingen, toch is ze onzes inziens algemeen toepasbaar in elke situatie waarin de sequentiële exploratietechniek toegepast wordt en kan men de resultaten van ander recent werk op deze idee projecteren. Men kan de idee onderkennen in het recente werk van de groep rond Sussman : [Stallman & Sussman 76], [Sussman 77], [Doyle 78a], [Doyle 78b], [McAllester 78]. Dit onderzoek had plaats in een totaal ander verband (analyse van elektrische schakelingen) en slechts zeer recent probeerde men de resultaten te veralgemenen. Ook het zeer theoretische werk [Cox 77] in verband met stellingbewijzers kan op deze idee geprojecteerd worden, alhoewel het voor ons onduidelijk is hoe dat systeem informatie over de reeds opgetreden conflicten bijhoudt.

Een goed overzicht van ander, niet zo nauw verwant, werk vindt men in [Sussman 77] en [Doyle 78b].

V. BESLUIT

Nadat in een eerste hoofdstuk de belangrijkste aspekten van het "logisch programmeren" toegelicht werden, hebben we in dit proefschrift onderzocht hoe de uitvoering van logische programma's beter beheerst kan worden.

In het tweede hoofdstuk werd bestudeerd hoe het geheugenbeheer van de bestaande PROLOG-vertolkers verbeterd kan worden. Er werd aangetoond dat een door de auteur ontwikkelde "kopieermethode" minstens evenwaardig is met een gelijktijdig door David Warren ontwikkelde methode. Deze laatste methode is op de "gedeelde strukturen" ("structure sharing") techniek gebaseerd. Een techniek die bij automatische stellingbewijzers algemeen als de beste aanvaard wordt. Ons merkwaardig resultaat is te danken aan het uitbuiten van de belangrijke geheugenbesparingen die het gebruik van de "sequentiële exploratie" ("backtracking") toelaat.

Het grote onderscheid tussen PROLOG en de logika der Horn-uitdrukkingen bestaat uit de controle, Logika als dusdanig bevat geen controle. In de wiskunde wordt, bij het bewijzen van stellingen, de controle ("welke afleidingsregels te gebruiken") overgelaten aan de intuïtie van de wiskundige. Een systeem ontwikkelen dat over deze intuïtie beschikt en daarenboven voldoende efficiënt is om als vertolker voor een programmeertaal gebruikt te worden, is momenteel nog niet mogelijk. Men aanvaardt vrij algemeen dat controle onontbeerlijk is.

Terwijl de logika reeds een oude en goed gefundeerde wetenschap is kan dit van de controle niet gezegd worden. Er is nog veel diskussie over de vraag welke soort(en) controle gewenst is. De laatste twee hoofdstukken vormen een bijdrage tot deze diskussie.

In het derde hoofdstuk werd een controletaal ontwikkeld. Aan de basis van deze ontwikkeling ligt de associatie tussen procedure-oproepen en processen. Een netwerk van parallelle processen zorgt dan voor de uitvoering van een programma. De logika volstaat opdat deze parallelle processen het gewenste resultaat zouden berekenen. Echter, om een efficiënte uitvoering te bekomen, moet de samenwerking tussen deze processen georganiseerd worden. Hiertoe werd uitgegaan van twee belangrijke principes.

Enerzijds de luie evaluatie : een proces voert slechts berekeningen uit op uitdrukkelijk verzoek. Anderzijds het opleggen van beperkingen. Dit laatste is nodig omdat een procedure met verschillende input/output patronen kan gebruikt worden, maar niet alle patronen tot een efficiënte uitvoering leiden. Deze beperkingen zorgen voor de eigenlijke synchronisatie van de processen.

Hopelijk hebben de voorbeelden voldoende geïllustreerd dat deze concepten toelaten om de controle van een programma modulair en gestructureerd te ontwikkelen en dat het mogelijk is om vrij complexe algoritmen te bekomen met een verrassend eenvoudige logika. Dit betekent dat de expressieve mogelijkheden van programma's in de logika der Horn-uitdrukkingen in belangrijke mate uitgebreid worden.

De resulterende programma's kunnen beschouwd worden als een beschrijving op zeer hoog niveau die in theorie een parallelle verwerking toelaat. De eenvoud van de concepten, alsook de opsplitsing tussen de logische component en de controlecomponent laten toe om zeer betrouwbare programma's te bekomen. Het ware interessant om te onderzoeken in hoeverre dergelijke programma's, met behoud van hun correctheidseigenschappen (automatisch of half-automatisch) kunnen omgezet worden naar andere formalismen. In het bijzonder naar een formalisme waarvoor een implementatie beschikbaar is die parallelle verwerking toelaat. (vb. Concurrent Pascal). Voor logische programma's die volledig deterministisch zijn, waar elke variabele slechts één producent heeft en waar de communicatie tussen parallelle processen gebeurt via eenvoudige lijststrukturen, stelt dit wellicht geen onoverkomelijke problemen. (vb. Sorteeralgoritme - Zeef van Eratosthenes).

Ook in PROLOG kan het gebruik van beperkingen (konsumentenbeperking, val-beperking) erg nuttig zijn. Soms wenst men een procedure slechts voor welbepaalde input/output patronen te gebruiken. Eventueel kan dit de procedure sterk vereenvoudigen. Door deze input/output patronen expliciet te maken, kan een verkeerd gebruik voorkomen worden. Tevens wordt het eenvoudiger om zich te overtuigen van de praktische eindigheid van de programma's. Voor elke procedure moet men enkel rekening houden met de toegelaten input/output patronen. Voldoet een bepaalde oproep niet aan de beperkingen, dan kan bij de vertaling of de uitvoering een "controlefout" vastgesteld worden. Het expliciet maken van de beperking kan ook het gebruik van "/" verantwoorden.

Voorbeeld :

$Mengen(Nil_{val}, Nil_{val}, Nil) \leftarrow$

$Mengen(Nil_{val}, (x.y)_{val}, x.y) \leftarrow$

$Mengen((x.y)_{val}, Nil_{val}, x.y) \leftarrow$

$Mengen((x.\ell 1)_{val}, (x.\ell 2)_{val}, x.\ell 12) \leftarrow Mengen(\ell 1_{val}, \ell 2_{val}, \ell 12)$

$Mengen((x.\ell 1)_{val}, (y.\ell 2)_{val}, x.\ell 12) \leftarrow x_{val} < y_{val} \quad Mengen(\ell 1_{val}, (y.\ell 2)_{val}, \ell 12)$

$Mengen((x.\ell 1)_{val}, (y.\ell 2)_{val}, y.\ell 12) \quad y_{val} < x_{val} \quad Mengen((x.\ell 1)_{val}, \ell 2_{val}, \ell 12)$

Onder de beperking dat beide eerste argumenten volledig bepaald zijn, is het eenvoudig om zich ervan te overtuigen dat de verschillende procedures elkaar uitsluiten en dat de uitvoering van een oproep Mengen eindig is indien beide inputlijsten eindig zijn (inductie op de som van de lengte van beide inputlijsten). Het is dan ook verantwoord om het programma te schrijven als :

$Mengen(Nil_{val}, Nil_{val}, Nil) \leftarrow /$

$Mengen(Nil_{val}, (x.y)_{val}, x.y) \leftarrow /$

$Mengen((x.y)_{val}, Nil_{val}, x.y) \leftarrow /$

$Mengen((x.\ell 1)_{val}, (x.\ell 2)_{val}, x.\ell 12) \leftarrow /Mengen(\ell 1_{val}, \ell 2_{val}, \ell 12)$

$Mengen((x.\ell 1)_{val}, (y.\ell 2)_{val}, x.\ell 12) \leftarrow x_{val} < y_{val} / Mengen(\ell 1_{val}, (y.\ell 2)_{val}, \ell 12)$

$Mengen((x.\ell 1)_{val}, (y.\ell 2)_{val}, y.\ell 12) \leftarrow / Mengen((x.\ell 1)_{val}, \ell 2_{val}, \ell 12)$

Ook een correcte implementatie van "negation as failure" [Clark 78] kan bekomen worden :

$Not(x\downarrow) \leftarrow x\downarrow \; / \; fail$

$Not(x\downarrow) \leftarrow$

De konsumentenbeperking verhinderd dat componenten in het argument van Not gebonden worden.

Het gebruik als database taal is een ander mogelijk toepassingsgebied. Vrij recent werden relationele database talen ontwikkeld. Ze zijn aantrekkelijk omdat een relationele database conceptueel zeer eenvoudig is. Tot nu toe echter zijn de implementaties eerder inefficiënt en kennen ze weinig sukses. Vermoedelijk ligt de oorzaak bij het zuiver deklaratieve karakter van de relationele talen en bij het onvermogen om automatisch een geschikte "controle" te genereren. Horn-uitdrukkingen vormen een natuurlijke uitbreiding van het relationele database concept. Procedures

zijn immers relaties die expliciet (door middel van een opsomming van hun elementen zoals bij relationele databases) of impliciet (door middel van methodes (procedures) om de elementen van de relaties te berekenen) gedefinieerd zijn. Gegevens en programma's vormen een uniform geheel en Horn-uitdrukkingen vormen een universeel formalisme dat niet alleen geschikt is om de gegevensbank te ondervragen ("query") maar ook om de antwoorden verder te verwerken. Mits het voorzien van speciale procedures die toelaten om operaties uit te voeren op de elementen die aan een relatie voldoen, verkrijgen we een zeer interessante taal (deze speciale procedures zijn te vergelijken met ingebouwde funkties in o.a. SEQUEL). Dit zou toelaten om o.a. te schrijven :

z is som y in R(x,y) : z is de som van alle antwoorden y op de vraag R(x,y)

ℓ is lijst x in S(x,y) : ℓ is de lijst van alle antwoorden x op de vraag S(x,y)

Dit mechanisme vormt een eenvoudige uitbreiding van de procedure "Bag" die in hoofdstuk I sektie B.3. voorgesteld werd. Het laat toe om de antwoorden op een vraag verder te manipuleren.

Een gebruiker die complexe problemen oplost of erg bekommerd is om de efficiëntie van zijn logisch programma kan het uitbreiden met een controlecomponent.. Bij het opstellen van deze controle kan de gebruiker zowel met de semantiek van de vraag rekening houden als met de inwendige voorstelling van de gegevensbank (toegangspaden en dergelijke). Een wijziging van die inwendige voorstelling maakt alleen het herschrijven van de controle noodzakelijk, controle die de correctheid van het programma niet verandert maar alleen de efficiëntie beïnvloedt.

In het vierde hoofdstuk werd de intelligente sequentiële exploratietechniek voorgesteld. Het is een techniek die de controle minder kritisch maakt en het zoekproces in belangrijke mate kan versnellen in situaties waar verschillende alternatieven moeten geprobeerd worden vooraleer een oplossing gevonden is. Dit gebeurt door het beter benutten van de informatie over reeds vastgestelde mislukkingen en over reeds gevonden oplossingen.

Een belangrijke innovatie is dat de methode de gelijktijdigheid toelaat tussen verschillende stappen in de berekening. Ze is dus toepasbaar in situtaties waar de berekening gebeurt door middel van meerdere processoren.

De berekening wordt opgesplitst in gedistribueerde parallelle processen die de eigenlijke berekeningen uitvoeren en een centraal proces. Dit centrale proces controleert de consistentie van de partiële resultaten en dwingt processen die een verkeerde richting opgaan om hun berekeningen terug ongedaan te maken. Het centrale proces zorgt daarbij dat de volledigheid behouden blijft en dat alle eventuele oplossingen gevonden worden.

Er werd vastgesteld dat de methode een verrijking vormt van de bestaande algoritmen voor het oplossen van combinatorische problemen [Mackworth 77][Freuder 78]. Waar deze laatste methodes eerder gericht zijn op het verfijnen van de probleemspecifikatie door het elimineren van alle potentiële conflictgevallen en daarna ofwel beroep doen op de naïeve sequentiële exploratie ofwel in parallel alle oplossingen berekenen, gaat onze methode zeer opportunistisch (de conflicten opvangend die zich voordoen) op zoek naar een eerste oplossing. Dit maakt haar uitermate geschikt voor het oplossen van vele praktische problemen waar men slechts in één oplossing geïnteresseerd is. (Voorbeeld : het vertalen van microprogramma's in REGTRAL naar machinecode waarbij, rekening houdend met een aantal beperkingen, de adressen van de instrukties berekend moeten worden.

Alhoewel de methode hier enkel geformuleerd werd voor programma's in de logika der Horn-uitdrukkingen kunnen de basisideeën toegepast worden in elke situatie waarin een oplossingsruimte moet onderzocht worden. Het lijkt steeds voordelig om een probleem in deelproblemen op te splitsen en de interakties tussen deze deelproblemen te beperken tot de eis dat de verschillende deeloplossingen consistent blijven. Een onnatuurlijke totale orde tussen de verschillende stappen van een niet-deterministische berekening moet zoveel mogelijk vermeden worden.

In de laatste twee hoofdstukken zijn wij erin geslaagd de onnnatuurlijke totale ordening, die tussen de verschillende stappen van een berekening bestaat, te doorbreken. We hebben een partiële ordening ontwikkeld die een bepaalde berekeningsstap slechts van andere berekeningsstappen laat afhangen wanneer dit werkelijk noodzakelijk is.

VI. BIBLIOGRAFIE

Andreka H. en Nemeti I. [76]

The generalised completeness of Horn predicate-logic as a programming language. D.A.I. Research report No. 21, University of Edinburgh, March 1976.

Ashcroft E.A. en Wadge W.W. [77]

Lucid, a nonprocedural language with iteration. Comm. ACM Vol 20, No. 7 (July 1977) p. 519-526.

Backus J. [78]

Can programming be liberated from the von Neumann style ? a functional style and its algebra of programs. Comm. ACM Vol 21, No. 8 (August 1978) p. 613-641 (Turing award lecture).

Baker H.G. en Hewitt C. [77]

The incremental garbage collection of processes. Proceedings of the Symposium on Artificial Intelligence and Programming Languages, University of Rochester, August 15-17, 1977, p. 55-59.

Barth J.M [77]

Shifting garbage collection overhead to compile time. Comm. ACM Vol 20, No. 7 (July 1977) p. 513-518.

Battani G. en Meloni H. [73]

Interpreteur du langage de programmation PROLOG. Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Université d'Aix-Marseille, 1973.

Bergman M. en Kanoui H. [75]

Sycophante, système de calcul formel et d'intégration symbolique sur ordinateur. Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Université d'Aix-Marseille, 1975.

Bibel W. [76a]

Predicative programming, séminaires IRIA, Roquencourt, 1976, ook in Lect. Notes Comp. Sc., Vol 33, Springer, Berlin, New York, 1975, p. 274-283.

Bibel W. [76b]

Synthesis of strategic definitions and their control. Report 7610, T. Universität München, May 1976.

Bibel W. [76c]

A uniform approach to programming. Report 7633, T. Universität München, 1976.

Bibel W., Furbach U. en Schreiber J.F. [78]

Strategies for the synthesis of algorithms, T. Universität München, 1978.

Bobrow D.G. en Raphael B. [74]

New programming languages for artificial intelligence research. Computing Surveys, Vol 6, No. 3 (September 1974), p. 153-174.

Bobrow D.G. en Wegbreit B. [73]

A model and stack implementation of multiple environments. Comm. ACM, Vol 16, No. 10 (October 1973), p. 591-603.

Boyer R.S. en Moore J.S. [72]

The sharing of structure in theorem-proving programs. Machine Intelligence 7 eds. Meltzer B. en Michie D., Edinburgh University Press, p. 101-116.

Bruynooghe M. [76]

An interpreter for predicate logic programs - basic principles. Report CW 10, Afdeling Toegepaste Wiskunde en Programmatie, K.U.Leuven, Oktober 1976.

Bruynooghe M. [78]

Intelligent backtracking for an interpreter of Horn clause logic programs. Report CW 16, Afdeling Toegepaste Wiskunde en Programmatie, K.U.Leuven, September 1978.

Chang C.L. en Lee R.C.T. [73]

Symbolic logic and mechanical theorem proving. Academic Press, New York, 1973.

Clark K. [78]

Negation as failure. Logic and Data Bases. Eds. Minker J. en Gallaire H., Plenum Publishing Corporation, New York, 1978.

Clark K. en Darlington J. [78]

Algorithm classification through synthesis. Dept. of Computing and Control, Imperial College London, June 1978.

Clark K. en Kowalski R. [77]

Predicate logic as a programming language. Dept. of Computing and Control, Imperial College, London, March 1977.

Clark K. en Tärnlund S.A. [77]

A first order theory of data and programs. IFIP 77, ed. Gilchrist B., North Holland Publishing Company.

Colmerauer A. [75]

Les grammaires de métamorphose. Groupe d'Intélligence Artificielle, U.E.R. de Luminy, Université d'Aix-Marseille, Novembre 1975.

Comerauer A., Kanoui H., Roussel P. en Pasero R. [73]

Un système de communication homme-machine en français. Groupe d'Intélligence Artificielle, U.E.R. de Luminy, Université d'Aix-Marseille, 1973.

Cox P.T. [77]

Déduction plans : a graphical proof procedure for the first-order predicate calculus. Report CS-77-28, Dept. of Computer Science, University of Waterloo, October 1977.

Darvas F., Futó I. en Szeredi P. [76]

Some application of theorem-proving based machine intelligence in QSAR : automatic calculation of modular properties and automatic interpretation of structure-activity relationships. International symposium on QSAR, Suhl, GDR, October 1976.

Darvas F., Futó I. en Szeredi P. [78]
Logic-based program system for predicting drug interactions. Int. J. Bio-Medical Computing, Vol 9 (1978) p. 259-271.

de Kleer J., Doyle J., Steele G.L. Jr. en Sussman G.J. [77]
AMORD : explicit control of reasoning. Proceedings of the Symposium on Artificial Intelligence and Programming Languages, University of Rochester, August 15-17 1977, p. 116-125.

Deliyanni A., Kowalski R.A. [79]
Logic and semantic networks. Comm. ACM, Vol 22, No. 3 (March 1979), p. 184-192.

Doyle J. [78a]
A glimpse of truth maintenance. AI Memo 461a, MIT, November 1978.

Doyle J. [78b]
Truth maintenance systems for problem solving AI-TR-419, January 1978.

Fishman D.H. en Minker J. [75]
Π-representation : A clause representation for parallel search. Artificial Intelligence, Vol 6 (1975) p. 103-127.

Freuder E.C. [78]
Synthesizing constraint expressions. Comm. ACM, Vol 21, No. 11 (November 78) p. 958-966.

Friedman D.P. en Wise D.S. [76]
CONS should not evaluate its arguments. Automata, Languages and Programming. Eds. Michaelson S. en Milner R., Edinburgh University Press, 1976, p. 257-284.

Friedman D.P. en Wise D.S. [78]
A note on conditional expressions. Comm. ACM, Vol 21, No. 11 (November 1978), p. 931-933.

Futó I., Darvas F. en Szeredi P. [78]
The application of PROLOG to the development of QA and DBM systems. Logic and Data Bases, Eds. Minker J. en Gallaire H., Plenum Publishing Corporation, New York, 1978.

Green C.C. [69]
The application of theorem proving to question-answering systems. Report CS 138, Memo AI-96, Stanford University, June 1969.

Hansen P.B. [78]
Distributed processes : a concurrent programming concept. Comm. ACM, Vol 21, No. 11 (November 1978), p. 934-941.

Hayes P.J. [73]
Computation and deduction. Proceedings of 1973. Math. Foundations of Computer Science Symposium, Czechoslovakian Academy of Sciences.

Hayes P.J. [77]
In defence of logic. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, August 22-25 1977.

Henderson P. en Morris J.H. Jr. [76]
A lazy evaluation. Conf. Record Third ACM Symposium on Principles of Programming Languages, Atlanta, Ga, January 1976, p. 95-103.

Hewitt C. [72]
Description and theoretical analysis (using schemata) of PLANNER : a language for proving theorems and manipulating models in a robot (Ph. D.), AI TR-258, M.I.T. April 1972.

Hill A.J. [76]
A predicate logic database. Computing and Control dept., Imperial College, London, August 1976.

Hoare C.A.R. [78]
Communicating sequential processes. Comm. ACM, Vol 21, No. 8 (August 1978), p. 666-677.

Hodges W. [77]

Logic. Penguin Books 1977.

Hogger C. [78]

Derivation of logic programs. Ph. D. Imperial College, London 1978.

Kahn G. [74]

The semantics of a simple language for parallel programming. IFIP 74, North Holland Publishing Company 1974.

Kahn G. en McQueen D.B. [77]

Coroutines and networks of parallel processes. IFIP 77, ed. Gilchrist B., North Holland Publishing Company 1977, p. 993-998.

Kleene S.C. [67]

Mathematical Logic. John Willey & Sons 1967.

Kowalski R.A. [74a]

Predicate logic as a programming language. IFIP 74, North Holland, 1974, p. 569-574.

Kowalski R.A. [74b]

Logic for problem-solving. Memo No. 75, Department of Computational Logic, University of Edinburgh, 1974.

Kowalski R.A. [76]

Algorithm = logic + control. Imperial College, London, November 1976. (zal verschijnen in Comm. ACM).

Kowalski R.A. [78]

Logic for data description. Logic and Data Bases, eds. Minker J. en Gallaire H., Plenum Publishing Corporation, New York, 1978.

Kowalski R.A. [79]

Logic for problem solving. Dit boek zal binnenkort verschijnen.

Leavenworth B.M. en Sammet J.E. [74]

An overview of nonprocedural languages. Proceedings of a symposium on very high level languages, March 24-29 1974, Santa Monica, California.

Lichtman B.M. [75]

Features of very high level programming with PROLOG. Imperial College September 1975.

Loveland D.W. [78]

Automated theorem proving : a logical basis. North Holland 1978.

Mackworth A.K. [77]

Consistency in networks of relations. Artificial Intelligence Vol 8 (1977), p. 99-118.

Mårkusz Z. [77]

How to design variants of flats using programming language PROLOG based on mathematical logic. IFIP 77, ed. Gilchrist B., North Holland, p. 885-889.

McAllester D.A. [78]

A tree valued truth maintenance system. AI-Memo 473, MIT, May 1978.

McGabe F. [78]

IC-PROLOG. Imperial College, London, 1978.

Moore J.S. [73]

Computational logic : structure sharing and proofs of program properties, part 1. D.C.L Memo, No. 67, University of Edinburgh, 1973.

Morris F.L. [78]

A time and spare-efficient garbage compaction algorithm. Comm. ACM, Vol 21, No. 8 (August 1978), p. 662-665.

Pereira L.M. en Monteiro L.F. [78]

The semantics of parallelism and co-routining in logic programming. Laboratoria National de Engenharia Civil, Lisboa, 1978.

Peremans W. [72]

Toegepaste Logica I. Onderafdeling Wiskunde T.H. Eindhoven, 1972.

Robinson J.A. [65]

A machine-oriented logic based on the resolution principle. J. ACM Vol. 12, No. 1 (January 1965) p. 23-41.

Roussel P. [75]

PROLOG manuel d'utilisation. Groupe d'Intélligence Artificielle, U.E.R. de Luminy, Université d'Aix-Marseille, September 1975.

Schwarz J.S. [77]

Using annotations to make recursion equations behave. DAI Research Report, No. 43, Dept. of Artificial Intelligence, University of Edinburgh, 1977.

Stallman R.M. en Sussman G.J. [76]

Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. AI Memo 380, MIT, September 1976.

Sussman G.J. en Winograd T. [70]

MICRO-PLANNER reference manual. AI Memo 203, MIT July 1970.

Sussman G.J. en McDermott, D.V. [72]

Why conniving is better than planning. AI Memo 255 A, MIT April 1972.

Sussman G.J. [73]

A computational model of skill aquisition. MIT Dept. of Math. Ph. D. thesis, AI Lab. TR297, (August 1973) also published by American Elsevier Publishing Company, New York 1975.

Sussman G.J. [77]

Electrical design, a problem for artificial intelligence research. Proceedings of the Fifth International Joint Conference on Artificial Intelligence (August 1977).

Tärnlund S.A. [75a]

On logic programming. IEEE Workshop on Automated Theorem-Proving, Argonne National Laboratory, Argonne, Illinois, June 1975.

Tärnlund S.A. [75b]

An interpreter for the programming language predicate logic. Fourth International Joint Conference on Artificical Intelligence, Tbilisi, Georgia, USSR 3-8 september 1975; p. 601-608.

Tärnlund S.A. [78]

An axiomatic data base theory. Logic and Data Bases, eds. Mincker J. en Gallaire H., Plenum Publishing Corporation, New York 1978.

Trum P. en Winterstein G. [78]

Description, implementation and practical comparison of unification algorithms. Fachbereich Informatik, Universität Kaiserslautern 1978.

van Emden M.H. [76a]

Programming with resolution logic. Machine Representations of Knowledge, published as Machine Intelligence 8, eds. Elcock E.W. en Michie D., Ellis Harwood Ltd, 1976.

van Emden M.H. [76b]

Verification conditions as representations of programs. Automata, Languages and Programming, eds. Michaelson S. en Milner R., Edinburgh University Press 1976, p. 99-119.

van Emden M.H. [76c]

A proposal for an imperative complement to PROLOG. CS-76-39, Dept. of Computer Science, University of Waterloo, August 1976.

van Emden M.H. [77]

Computation and deductive information retrieval. CS-77-16, Dept. of Computer Science, University of Waterloo.

van Emden M.H. en Kowalski R.A. [76]

The semantics of predicate logic as a programming language. JACM Vol 23, No. 4 (October 1976), p. 733-742.

Vuillemin J.E. [73]

Proof techniques for recursive programs. Ph.D. thesis, Stanford University 1973.

Wadsworth C.P. [71]

Semantics and pragmatics of the lambda-calculus. Ph. D. thesis, University of Oxford 1971.

Warner N.M. [77]

Intelligent backtracking. MSC Theory Group. Project spring term 1977, Imperial College, London.

Warren D.H.D. [74]

Warplan : a system for generating plans. DCL Memo 76, University of Edinburgh, 1974.

Warren D.H.D. [77a]

Implementary PROLOG-compiling predicate logic programs, Volume 1 and 2. D.A.I. Research Report No. 39 and No. 40, University of Edinburgh 1977.

Warren D.H.D. [77b]

Logic programming and compiler writing. D.A.I. Research Report No. 44, University of Edinburgh 1977.

Warren D.H.D., Pereira L.M. en Pereira F. [77]

PROLOG - the language and its implementation compared with LISP. Proceedings of the Symposium on Artificial Intelligence and Programming Languages, University of Rochester, August 15-17 1977, p. 109-115.