Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O).

MC SYLLABUS 37

COLLOQUIUM CAPITA DATASTRUCTUREN

J.C. VAN VLIET (RED.)

MATHEMATISCH CENTRUM AMSTERDAM 1978

AMS(MOS) subject classification scheme (1970): 68-00, 68A05, 68A10, 68A20, 68A45, 68A50

Comp. Reviews: 4.34, 3.60, 3,63, 4.22, 4.33, 5.25, 5.27

ISBN. 90 6196 159 9

INHOUD

Inhoud i

Voorwoord viii

I DATASTRUCTUREN, CONCREET EN ABSTRACT;
EEN INLEIDING door R.P. van de Riet

1. Inleiding 1
1.1. Datastructuren 1
1.2. Programmeertalen 1
1.3. Nieuwe datastructuren 1
1.4. Implementatiedetails, voordelen 2
1.5. Implementatiedetails, nadelen 2
1.6. Afhankelijkheid 3
1.7. Ondeelbaarheid 3
1.8. Onoverzichtelijk 3
1.9. Abstractie 4
1.10. Hierarchie 4
1.11. Administratieve dataverwerking 5
1.12. Overzicht 5
2. Concrete datastructuren 5
2.1. Datastructuren in hardware 5
2.2. Concrete representatie 6
3. Abstracte datastructuren 7
3.1. Wat in plaats van hoe 7
3.2. Vastleggen van de semantiek 9
3.3. Verificatie van implementatie, gebruik en semantiek 11
4. Abstractie in databankorganisatie 11
4.1. Een "Theorie" van de werkelijkheid 12
4.2. "In de beperking toont zich de meester" 13
4.3. Het hierarchisch gestructureerde datamodel 14
4.4. Het DBTG model 14
4.5. Het relationeel gestructureerde model 15
4.6. De algebra van FA's 17
4.7. Abstractie in de vorm van aggregatie en generalisatie 19
4.8. Andere abstracties 23
5. Conclusie 23

Literatuur 24

II ABSTRACTE DATATYPEN door L.G.L.T. Meertens

1. Abstractie 27
2. Lijnen in de ontwikkeling 28
2.1. Records 28
2.2. Extensibiliteit 29
2.3. Correctheid van gegevensrepresentatie 30
2.4. Modulariteit 31
2.5. Inkapseling 32
2.6. Een stapje terug in de geschiedenis 33
3. De doorbraak 34
4. Axiomatische specificatie 35
5. De emancipatie van abstracte datatypen 37
Literatuur 40

III COMPLEXITEIT VAN VERZAMELINGENMANIPULATIE
door P. van Emde Boas

1. De relevantie van verzamelingenmanipulatie 43
2. De spelregels 45
3. De binaire hoop - een klassieke gegevensstructuur 47
4. De binomiale hoop 49
5. Een efficiënte prioriteitenwachtrij 55
6. Implementatie van de O (log log n) prioriteitenwachtrij 57
7. Recursieve implementatie van de prioriteitenwachtrij 59
Literatuur 62

IV PATTERN MATCHING IN SPRING by P. Klint

1. Introduction 65
2. An introduction to pattern matching 68
3. Models for pattern matching 75
3.1. An algebraic model of pattern matching 76
3.2. From model to implementation 78
3.3. An example of pattern design 80
3.4. A closer examination of alternation 81
References 83

V AXIOMATIEK VAN DATASTRUCTUREN

door H.J.M. Goeman, A. Ollongren & Th.P. van der Weide

1. Inleiding 85
2. Waarom een axiomatische benadering van datastructuren? 85
3. Gelaagde structuren 86
4. De axiomatisering van gelaagde structuren 88
5. Standaardmodellen 90
6. Bijzondere modellen 90
7. Enkele eigenschappen 93
8. Eindige grafen als onderliggende structuur 95
9. Definieerbare substructuren 97

Literatuur 98

VI DATASTRUCTUREN VOOR LINEAIRE RUIMTEN "TORRIX" door S.G. van der Meulen & M.Veldhorst

1. Inleiding en axiomatisch frame 99
1.1. Inleiding 99
1.2. Axiomatische abstractie 100
1.3. Lineaire ruimten 101
1.4. Basis-stellingen en representatie 102
1.5. Implementatie-criteria 103
1.6. Implementatie-taal 104
2. Abstractiestappen op een klassieke datastructuur 105
2.1. Het grondsysteem *scal* 105
2.2. Toekomstmuziek 106
2.3. Het nulnivo 107
2.4. Het operatie-nivo 108
2.5. Equivalentieklassen van *arrays* 109
2.6. Het torrix-nivo 110
3. Genereren en opereren 112
3.1. Operaties in T 112
3.2. Genereren 112
3.3. Toeschrijven en toekennen 113
3.4. Nul en een 115
3.5. Testen en trimmen 115
3.6. Selectors 116
3.7. Punt-operaties 117

3.8. Genererende- en assignerende operaties 118
3.9. Additieve operaties 118
3.10. Multiplicatieve operaties 119
3.11. Een voorbeeld 120
4. Nieuwe mogelijkheden 121
4.1. *vec* en *mat* als primitiva 121
4.2. Een gemiste kans 122
4.3. *vecrow*, *rows*, *sym* en *band* 122
4.4. Open einde 123
Literatuur 124

VII KNOWLEDGE REPRESENTATIONS IN ARTIFICIAL INTELLIGENCE
by H.J. Sint
1. Introduction 127
2. Two extremes: resolution based theorem proving and PLANNER 130
2.1. Logic with a theorem prover 131
2.2. The PLANNER formalism 134
3. Network representations 138
4. Production systems 147
5. Generalities 151
5.1. Declarative versus procedural representations 151
5.2. Redundancy 151
5.3. Canonical form 152
5.4. Task dependency 152
5.5. Learning 153
References 153

VIII DATATYPEN BEZIEN VANUIT DE RECURSIETHEORIE
door J.A. Bergstra
1. Inleiding 157
2. Dynamische recursie-theorie 158
2.1. Processen 158
2.2. Automaten 158
2.3. $D^{\leq}_{\mathbb{N}}$ 159
2.3.1. Zij PR = PR ($\mathbb{N}$, $\mathbb{N}$) 159
2.3.2. Definitie. $D^{\leq}_{\mathbb{N}}$ = <PR/$\equiv$, $\leq$, +, 0> 159
2.3.3. Functionele processen 160

2.3.4. Open problemen 160
2.3.5. Operatoren 161
2.4. D_{Σ}^{FC} 162
2.4.1. Een subrecursieve variant van $D_{\mathbb{N}}^{\leq}$ 162
2.4.2. Definitie. $D_{\Sigma}^{FC} = \langle RP_{\Sigma}/\equiv_{FC}, \leq_{FC}, +, 0\rangle$ 163
2.4.3. Resultaten t.a.v. D_{Σ}^{FC} 163
2.4.4. Open problemen t.a.v. D_{Σ}^{FC} 165
2.5. D_{Σ}^{EFC} 166
3. Datatypen 166
3.1. Datatypen uit de complexiteitstheorie 166
3.2. Algebraïsche datatypen 167
3.3. Niet-deterministische datatypen 168
4. Autonome processen 169
Literatuur 170

IX PROCEDURELE DATASTRUCTUREN door L.G.L.T. Meertens
1. Inleiding 171
2. Conventies 172
3. Rijen 175
4. Kettingbreuken 181
5. Files 184
6. Besluit 185
Literatuur 186

X FILE-OPTIMALISERING DOOR MIGRATIE VAN RECORDS
door J. van Leeuwen
1. Inleiding 187
2. Bijsturing in seriële bestanden 189
2.1. De FC of "frequency count" regel 190
2.2. De FD of "frequency difference" regel 191
2.3. De LD of "limited difference" regel 192
2.4. De WMC of "wait, move and clear" regel 192
2.5. De WM of "wait and move" regel 193
2.6. De MTF of "move to front" regel 193
2.7. De T of "transposition" regel 195
2.8. Andere "permutatie"-regels 195
2.9. Vertraagde permutatie-regels 196
3. Bijsturing in binaire zoekbomen 197

3.1. De SE of "simple exchange" regel 198
3.2. De DE of "double exchange" regel 198
3.3. De MTR of "move to root" regel 199
3.4. De MT of "monotone tree" regel 199
3.5. De LSR of "limited single rotation" regel 200
3.6. De LDR of "limited double rotation" regel 200
4. Bijsturing in andere structuren en aanverwante problemen 201
Literatuur 203

XI ENKELE UITGANGSPUNTEN VOOR DATAMANIPULATIE
door J.H. ter Bekke

1. Samenvatting 207
2. Definitie van gegevens 207
2.1. Basisbegrippen 208
2.2. Semantiek 209
2.2.1. Subjecten 209
2.2.2. Samenhangende subjecten 211
2.2.3. Geordende subjecten 212
3. Manipulatie van gegevens 214
3.1. Basisbegrippen 215
3.2. Selektie 216
3.2.1. Subjecten 216
3.2.2. Samenhangende subjecten 217
3.2.3. Geordende subjecten 220
Literatuur 223

VOORWOORD

In deze syllabus zijn de voordrachten gebundeld die gehouden zijn in het Colloquium Capita Datastructuren. Dit colloquium werd in het academisch jaar 1977/1978 door de Afdeling Informatica van het Mathematisch Centrum georganiseerd.

De serie voordrachten geeft een overzicht van een aantal recente ontwikkelingen op het gebied van datastructuren. Deze syllabus bevat zowel overzichtsartikelen als verslagen van nieuw onderzoek.

J.C. van Vliet.

DATASTRUCTUREN
CONCREET EN ABSTRACT
EEN INLEIDING

R.P. VAN DE RIET
Vrije Universiteit, Amsterdam

1. INLEIDING

1.1. Datastructuren

Datastructuren vormen het fundament van de informatica. Immers, in de informatica gaat het om opslaan en bewerken van informatie. Het is nodig dat die informatie op de een of andere manier wordt gerepresenteerd in de vorm van datastructuren.

Voor de representatie heb je op het laagste niveau materiaal nodig (hardware) in de vorm van adresseerbare bits en woorden in een computergeheugen en in de vorm van (vaak associatief) bereikbare bytes op magnetische tapes en schijven. Hier vind je datastructuren opgeslagen in de meest concrete vorm.

1.2. Programmeertalen

Deze stellen ons in staat om af te zien van sommige fysische bijzonderheden, zoals adressen, zodat we in staat zijn booleans, karakters, integers, reals, arrays, structures, sets, etc., in onze programma's te gebruiken. Wel moeten we verdacht zijn op problemen met overflow (van integers, reals, en ook van karakters en sets) en we krijgen te maken met wijzers in meer ingewikkelde constructies.

1.3. Nieuwe datastructuren

Met de hierboven genoemde hulpmiddelen kunnen we nieuwe gecompliceerde datastructuren maken, zoals (meervoudig) gelinkte (geschakelde) lijsten, bomen en netwerken. Datastructuren van nog weer grotere complexiteit verkrijgen we door lijsten en bomen te gebruiken voor implementatie van een stapel, een queue, een naamlijst of een verzameling (met operaties als

voegtoe, laatweg, zoekop, vereniging, doorsnede, etc.). En op een ander gebied maken we zulke gecompliceerde datastructuren als random-access files, index-sequentiële files, inverted files, etc.

1.4. Implementatiedetails, voordelen

Om met deze datastructuren te kunnen werken moet men als gebruiker zeer goed op de hoogte zijn van de details van de implementatie. Dit heeft soms voordelen. Zo kon professor van Wijngaarden, die een demonstratie voorbereidde van de Electrologica - MC X1 computer, waarin hij zoveel mogelijk decimalen van π wilde laten printen, toen hij hoorde dat een zeker bit kapot was (steeds 0 of steeds 1), eerst testen of dit bit, dat de zoveelste binaire digit van π moest representeren, toevallig de juiste waarde had (door π te printen en te vergelijken met een gepubliceerde versie) of de verkeerde waarde aangaf in welk geval van Wijngaarden de correcte decimalen van $-\pi$ liet printen. Dit dankzij kennis over het simpele 1-complement systeem waarin getallen op de X1 werden opgeborgen.

Zo konden Frank Teer en Ad König PASCAL en FORTRAN aan elkaar knopen op zo'n manier dat in FORTRAN gecreëerde cellen in de PASCAL ruimte werden opgeborgen door gebruik te maken van de truc om de run-time heap-wijzer van PASCAL door te spelen naar FORTRAN.

1.5. Implementatiedetails, nadelen

Meestal heeft het echter nadelen dat de gebruiker op de hoogte moet zijn van details van de implementatie. Immers, in de eerste plaats lijkt de datastructuur veel ingewikkelder dan hij in wezen is. Bijvoorbeeld, een stapel wordt gekenmerkt door de operaties stapel en ontstapel, een begrip dat met een enkel voorbeeld aan iedereen duidelijk gemaakt kan worden; in plaats daarvan moet een gebruiker meestal op de hoogte zijn van: een stapelwijzer, die ofwel de eerste lege, of de laatst bezette plaats aanwijst, en een array van elementen die gedeclareerd moet worden op een moment dat de lengte nog niet bekend is. Of hij moet op de hoogte zijn van een gelinkte lijst van cellen waarvan de voorste het laatst gevulde stapelelement is en de laatste naar niets wijst; tevens moet hij op de hoogte zijn van zulke zaken als "new" (voor het creëren van een lege cel) en "dispose" (voor het verwijderen van een gebruikte cel).

Een ander (wellicht extreem) voorbeeld: In een databank wordt een telefoonboek bewaard. Om een telefoonnummer te weten te komen moet een

gebruiker de opdracht kunnen geven: "geef me het telefoonnummer van meneer Jansen in Amsterdam", de databank stelt vervolgens de vraag welk adres meneer Jansen heeft en vervolgens wordt het telefoonnummer, indien aanwezig, bekend gemaakt (zie bijvoorbeeld CODD [6]). Stel nu dat elke gebruiker een programma zou moeten schrijven waarin hij rekening moet houden met de meervoudige geïndiceerdheid van het bestand en met de zeer efficiënt uitgedachte tekstcompressie; het vermoeden lijkt gewettigd dat in zo'n geval er slechts weinig gebruik van de databank zal worden gemaakt.

1.6. Afhankelijkheid

Een ander nadeel van het bekend zijn met alle details van de implementatie van een datastructuur is dat programma's hiervan afhankelijk worden, in plaats van afhankelijk van de operaties van de datastructuur. Een verandering van de implementatie van de datastructuur heeft dan ook gevolgen voor die programma's. Bijvoorbeeld, een programma, dat stiekem gebruik maakt van de waarde van de stapelwijzer, moet veranderen wanneer de stapel als een lineaire lijst wordt geïmplementeerd. Dit is een reden dat veel programma's niet meer werken na wijziging van een bedrijfssysteem of van een vertaler.

1.7. Ondeelbaarheid

Een ander nadeel is dat het moeilijk is om een groot project in hanteerbare stukken te verdelen en deze door afzonderlijke teams van programmeurs simultaan te laten uitvoeren, immers als het ene team afhankelijk is van de beslissingen van een ander team over details van de implementatie van een datastructuur, dan is het onmogelijk dat de teams simultaan werken. Gegarandeerd dat er onenigheid ontstaat (typische conversatie: "vorige maand hebben jullie gezegd dat je met een stapelwijzer zou werken, nu blijkt dat je het met een lijsttechniek hebt gedaan en kunnen wij onze programma's weer wijzigen").

1.8. Onoverzichtelijk

Tenslotte, het allerbelangrijkste nadeel is wel dat door alle details van de implementatie programma's onoverzichtelijk worden en zich slecht lenen voor correctheidsanalyse. Stel dat we een programma willen maken dat met behulp van een stapel een arithmetische expressie vertaalt van infix-

naar postfixnotatie. Om de correctheid van dit programma te bewijzen, zal het nodig zijn gebruik te maken van de eigenschappen van een stapel. Als je die eigenschappen in het volgende axioma kunt vastleggen:

ontstapel(stapel(i)) = i

dan bestaat er een goede kans dat het bewijs van correctheid inderdaad geleverd kan worden. Wanneer je daarentegen de details van de implementatie van de stapel er bij moet betrekken, is het bewijs een lastige zaak.

1.9. Abstractie

Redenen als hier genoemd hebben diverse onderzoekers aangespoord om programmeertalen te ontwerpen waarin de mogelijkheid tot abstractie veel beter is ingebouwd dan in de huidige programmeertalen. In CLU van BARBARA LISKOV [20,21], bijvoorbeeld, is het mogelijk een stapel te implementeren, waarbij alleen de operaties stapel, ontstapel en creëer stapel gebruikt kunnen worden. De wijze waarop de stapel is geïmplementeerd is verborgen voor de rest van het programma. De compiler kan rechtstreekse referentie naar interne variabelen van buitenaf voorkomen.

1.10. Hierarchie

We zien een zekere hierarchie van concrete datastructuren naar abstracte:
De meest concrete laag is de electronica.
De volgende laag is machinecode, waarin geabstraheerd wordt van poorten etc. Het is niet mogelijk bepaalde poorten te openen of te sluiten, wel is het mogelijk met bits en adressen te manipuleren.
De volgende laag is assemblercode, waarin geabstraheerd wordt van machineadressen en het, zeker op een computer met virtueel geheugen, niet of nauwelijks mogelijk is individuele bits aan en uit te zetten.
De volgende laag is hogere programmeertaal, waarin geabstraheerd wordt van bits, bytes en woorden, ja zelfs van adressen; hier is het mogelijk zulke datastructuren als stapels, lijsten en bomen te implementeren.
Bij de volgende laag, en in CLU blijf je in dezelfde taal bezig, wordt geabstraheerd van implementatiedetails en zijn het alleen de pure syntactische en semantische eigenschappen die gebruikt worden.

1.11. Administratieve dataverwerking

Werd hierboven een ontwikkeling "van concreet naar abstract" geschilderd op het terrein van programmatuur dat vaak met de term "software engineering" wordt aangeduid, ook op het gebied van de administratieve dataverwerking is een dergelijke ontwikkeling gaande, zij het dat daar een argument als "correctheid" niet zo'n grote rol speelt, maar meer het argument van "eenvoud voor de gebruiker". Ook is van belang een ontwikkeling te signaleren tot meer abstractie in datastructuren, teneinde de wiskundige structuur beter te kunnen analyseren.

1.12. Overzicht

In hoofdstuk 2 zullen we enige aandacht aan "concrete" datastructuren geven, in hoofdstuk 3 zullen we de "abstracte" datastructuren onder de loupe nemen, zeer summier voor zover het de zeer interessante ontwikkeling van "abstracte data types" betreft, omdat deze in een ander deel van dit colloquium uitvoerig aan de orde komen. Iets uitvoeriger zullen we in hoofdstuk 4 abstracties op het gebied van databanken behandelen. Tenslotte volgen enkele concluderende opmerkingen in hoofdstuk 5.
Het geheel is een beetje kaleidoscopisch geworden, maar dat brengt een veelzijdig onderwerp als datastructuren met zich mee.

2. CONCRETE DATASTRUCTUREN

Hoe worden datastructuren, bits, bytes, integers, reals, arrays, lijsten, structures, bomen, stapels, etc., conctreet gerepresenteerd? In dit hoofdstuk gaan we hier kort op in.

2.1. Datastructuren in hardware

In de eerste computers werden gehele getallen gerepresenteerd in de vorm van individueel adresseerbare woorden. Spoedig kon men ook op vele computers tekst op snelle wijze verwerken omdat er adresseerbare bytes kwamen. Nieuwe computers zijn ontwikkeld, zoals CDC Star-100 en Amdahl 470 V/6 (zie RAMAMOORTHY [26]) voor het werken met vectoren en matrices. Voor lijstmanipulaties wordt op het MIT een speciale LISP machine ontwikkeld. Computers met ingebouwde stapelmechanismen kennen we reeds lang; voorbeelden zijn de Electrologica X8 en de Burroughs 5000-6000 serie.

Een voorbeeld van een computer die op een bepaald micro-niveau met bits als adresseerbare eenheid opereert is de Burroughs 1700. Dat microprogrammering ons in staat stelt om volledig te abstraheren van bits, woorden, etc., wordt aangetoond door een microgeprogrammeerde computer die de hogere programmeertaal APL als machinetaal heeft (HASSITT [13]), daarin zijn bijvoorbeeld arrays elementaire objecten.

In een andere richting gaan computers met associatieve geheugens. Dit zijn geheugens waarin men tabellen kan opslaan die op inhoud worden geadresseerd. Bijvoorbeeld, stel dat het telefoonboek in zo'n geheugen wordt opgeborgen, dan is de naam van de abonnee (aangenomen dat hij uniek is) voldoende om het telefoonnummer te vinden en dat in min of meer één geheugencyclus. Een interessant overzichtsartikel over dit soort computers geven YAU en FUNG [33]. In een recent artikel beschrijven OZKARAHAN, SCHUSTER en SEVCIK [25] het RAP systeem, ontwikkeld op de Universiteit van Toronto. Hier worden een aantal cellen, zelf microcomputers, aangesloten op een snel en groot ronddraaiend geheugen waarin relaties in de vorm van tabellen staan opgeslagen. Elke cel is toegewezen aan een bepaalde sector van dit geheugen. De cellen kunnen nu ieder afzonderlijk hun sectoren lezen en onderzoeken op het voorkomen van bepaalde kenmerken. Er kunnen markeringsbits worden gezet om deelverzamelingen in de tabellen aan te wijzen. Lezen en schrijven van de cellen in hun rondwentelende sectoren gebeurt simultaan, waarbij een gelezen tuple eerst wordt bewerkt (veranderd, gemerkt, weggelaten, etc.) alvorens weggeschreven te worden. Het invoegen en weglaten van tuples kan zo tamelijk eenvoudig plaatsvinden. In [19] wordt een soortgelijk ronddraaiend associatief geheugen beschreven in het RARES systeem.

2.2. Concrete representatie

Over datastructuren concreet gerepresenteerd willen we kort zijn omdat we kunnen verwijzen naar de werken van KNUTH [15,16]. Interessante nieuwe ontwikkelingen op het gebied van complexiteit van datastructuren worden beschreven door VAN LEEUWEN [18]. Hier vindt men referenties naar AVL-trees, B-trees, 2-3 trees, 1-2 trees, HB-trees, etc. BAYER beschrijft in [2] zogenaamde Prefix B-trees: In de uiteinden van min of meer gebalanceerde binaire bomen worden de records opgeslagen, terwijl beginletters van de identifiers van de records in de vertakkingen staan, zodat een optimale zoekstrategie mogelijk is. In een recent boek van GHOSH [11] worden

datastructuren geanalyseerd, die nodig zijn om gegevens in een databank efficiënt op te bergen. Men zie ook het overzichtsartikel van SENKO [40].

3. ABSTRACTE DATASTRUCTUREN

We zien dat het abstractieproces (= afzien van details) in diverse richtingen wordt toegepast. We zullen hier kort ingaan op abstracte datatypen.

3.1. Wat in plaats van hoe

Bij het gebruiken van een datastructuur, zoals een stapel, willen we slechts gebruik maken van kennis over wat de datastructuur doet, terwijl we bij voorkeur niet willen weten hoe hij functioneert. In CLU, een programmeertaal ontwikkeld op het MIT door BARBARA LISKOV e.a. [20,21], worden clusters gebruikt om abstracte datatypen te implementeren. Een cluster is volledig afgesloten van zijn omgeving en kan ook apart gecompileerd worden. Het is onmogelijk om als gebruiker van een cluster gebruik te maken van enig object dat binnen de cluster wordt gebruikt, behalve als die objecten m.b.v. een ingewikkeld parameter-mechanisme naar buiten worden doorgegeven. Het is zelfs zo, dat het type van een object dat van binnen naar buiten of van buiten naar binnen wordt getransporteerd, verandert.

Voorbeeld van een cluster is die voor een stapel (uit [20]):

```
stack: cluster (element_type: type) is push, pop, empty:

    rep(type_param: type) = (tp: integer;
                             e_type: type;
                             stk:array[1..] of type_param);

    create
        s: rep(element_type);
        s.tp := 0;
        s.e_type := element_type;
        return s;
    end

    push: operation (s:rep, v: s.e_type);
          s.tp := s.tp+1;
          s.stk[s.tp] := v;
          return;
```

```
    end
    pop: operation (s:rep) returns s.e_type;
         if s.tp=0 then error;
         s.tp := s.tp-1;
         return s.stk[s.tp+1];
    end
    empty: operation (s:rep) returns boolean;
           return s.tp = 0;
    end
```

Van buiten kunnen we de stapel gebruiken op de volgende manier:
declareer en creëer een stapel:

```
s: stack (char);
```

hetgeen een (lege) stapel voor karakters *(char)* creëert. We kunnen nu vragen of de stapel leeg is:

```
if stack $ empty (s) then ... else ...
```

Ook kunnen we karakters stapelen:

```
stack $ push (s,'t');
```

of de stapel legen m.b.v.:

```
ch := stack $ pop(s);
```

Het is syntactisch onmogelijk om bij de stapelwijzer te komen of bij het array *stk*.

De boven aangegeven cluster vertoont veel overeenkomst met het begrip class in SIMULA 67 [7]. In het bijzonder wat betreft de mogelijkheid meerdere clusters simultaan te laten leven; bijvoorbeeld door de declaratie:

```
s,t: stack (char);
w: stack (integer);
```

ontstaan drie stapels, twee voor karakters en één voor integers. Belangrijke verschillen met de SIMULA classes zijn:
Ten eerste, met een SIMULA class kunnen willekeurige processen worden

gedefinieerd, terwijl de enige activiteit die een CLU cluster kan ontplooien is: bestaan.
Ten tweede, in SIMULA kunnen details over de interne representatie van de datastructuur niet verborgen blijven voor de omgeving, omdat procedures en variabelen gedefinieerd binnen de class bereikbaar zijn. Bijvoorbeeld in het geval van de definitie van een stapel, zoals boven aangegeven in CLU, zouden array *stk* en stapelwijzer *tp* beschikbaar zijn onder de namen *s.stk* en *s.tp*. Dit is in CLU niet het geval omdat, zoals gezegd, *s* buiten de cluster het type "stack" heeft en binnen de cluster een structuur, bestaande uit integer, type en array. Buiten de cluster zijn alleen de expliciet aangegeven procedures bereikbaar: *push*, *pop* en *empty*.

3.2. Vastleggen van de semantiek

Abstraheren van de implementatie van een datastructuur betekent dat de betekenis (= semantiek) van de datastructuur vastgelegd moet worden op een wijze die anders is dan een implementatie in termen van meer primitieve datastructuren.

Hier zijn diverse methoden voor bedacht. Een operationele methode wordt gebruikt in VDL (Vienna Definition Language, zie b.v. WEGNER [32]), hier wordt de betekenis van een taalconstructie vastgelegd in de vorm van toestanden en veranderingen van toestanden m.b.v. definities die zelf weer geïnterpreteerd kunnen worden door een interpretator. Zo kan, bijvoorbeeld, een stapel als volgt gedefinieerd worden:

$$is\_stack = (\langle s\_stack: is\_stack\rangle, \langle s\_element: is\_element\rangle) \vee is\_\Omega$$

met de betekenis: Aan de voorwaarde *is_stack* wordt voldaan door objecten die ofwel leeg (Ω) zijn, of waarvan de *s_stack* component aan dezelfde voorwaarde voldoet en de *s_element* component aan de voorwaarde *is_element* voldoet. De operaties *push* en *pop* kunnen als volgt gedefinieerd worden:

$$push(x,y) = \mu_0(\langle s\_stack: x\rangle, \langle s\_element: y\rangle)$$
$$pop(x) = s\_stack(x).$$

Hierin is μ_0 de constructie operator die van de stack *x* en het element *y* één boom maakt met twee takken. Deze takken worden aangeduid met *s_stack*

en *s_element*, en hun waarden zijn respectievelijk (de bomen) *x* en *y*. Het resultaat van *pop* is: als *x* geen *s_stack* heeft dan Ω (zoiets als nil) en anders de *s_stack* tak van *x*.

ZILLES [34] introduceert een algebraïsche methode: Uit het syntactische deel van de specificatie wordt de verzameling van alle mogelijke eindige zinnen gevormd welke de woorden van een woordalgebra vormen. Vervolgens worden axioma's gegeven die specificeren wanneer twee woorden equivalent zijn; voor de stapeloperaties is een axioma bijvoorbeeld:

pop(push(s,i)) = s.

Alle woorden waarvan niet kan worden bewezen dat ze equivalent zijn, worden geacht verschillend te zijn.

GUTTAG [12] geeft een interessant uitgewerkt voorbeeld van het gebruik van axioma's voor een "symbol table". Dat gedeelte dat een stapel definieert halen we hier naar voren: Als syntactische (operationele) definitie van stack vinden we:

NEWSTACK: → Stack
*PUSH : Stack * Integer → Stack*
POP : Stack → Stack
TOP : Stack → Integer
IS_NEWSTACK?: Stack → Boolean.

Hierin stellen *Stack*, *Integer* en *Boolean* de verzameling van stapels, van integers en van booleans voor. Met deze definitie kunnen alle mogelijke zinnen worden gevormd, zoals:

TOP(POP(PUSH(PUSH(NEWSTACK,1),2)))

en

TOP(PUSH(NEWSTACK,1))

die onder gebruikmaking van de volgende axioma's als equivalent kunnen worden beschouwd:

(1) *IS_NEWSTACK?(NEWSTACK) = true*
(2) *IS_NEWSTACK?(PUSH(stk,int)) = false*
(3) *POP(PUSH(stk,int)) = stk*

(4) *TOP(PUSH(stk,int)) = int*

(5) *POP(NEWSTACK) = error*

(6) *TOP(NEWSTACK) = error.*

Inderdaad, passen we regel 3 toe op de eerste zin, dan verschijnt de tweede zin; passen we vervolgens regel 4 toe, dan zien we dat beide zinnen equivalent zijn aan "1".

De rol van *error* in bovenstaande axioma's is dat, zo gauw een of andere parameter x_i in een functie $f(x_1,...,x_n)$ gelijk is aan *error*, dan is de functie equivalent met *error*.

Met bovenstaande axioma's is Guttag dan in staat diverse eigenschappen aan te tonen van de "symbol table", die m.b.v. een stapel is gedefinieerd. Problemen betreffende consistentie en volledigheid worden door Guttag aangestipt.

3.3. Verificatie van implementatie, gebruik en semantiek

Naar mijn mening ontbreekt op dit moment een programmeertaal waarin enerzijds de details van de implementatie van een datastructuur (noodzakelijkerwijs) beschreven staan en waar anderzijds de volledige semantische definitie van die datastructuur is aangegeven. Het zou dan mogelijk zijn om a) automatisch te verifiëren of die implementatie aan de semantische definitie voldoet, zoals in het "program verifier project" van het Stanford Artificial Intelligence Laboratory (LUCKHAM [22]), en b) na te gaan of de toepassing van de datastructuur op correcte wijze geschiedt. Een aanzet tot een dergelijke programmeertaal zou EUCLID (LAMPSON [17]) kunnen zijn, ontworpen om verifieerbare systeemprogramma's te maken.

4. ABSTRACTIE IN DATABANKORGANISATIE

Een catalogus van een bibliotheek is veel handiger om te gebruiken wanneer we willen weten of een bepaald boek er in staat, dan de verzameling boeken zelf; een telefoonboek is veel handiger te gebruiken om een nummer te zoeken dan de verzameling telefoonabonnees; vandaar dat de mens reeds eeuwen lang representaties van de werkelijkheid, in de vorm van boeken, kaarten en nu magnetische tapes en schijven, gebruikt in plaats van die werkelijkheid zelf. Databanken zijn bekende hulpmiddelen voor die representaties.

Wanneer je over abstracties praat dan moet je het zowel over abstracties van de werkelijkheid hebben als over abstracties van de datastructuren nodig om een databank te maken.

4.1. Een "Theorie" van de werkelijkheid

Als je een abstractie van de werkelijkheid wilt maken moet je eerst een model van de werkelijkheid hebben. We zien maar af van de filosofische vraag of er een objectieve werkelijkheid bestaat. In zijn boek "Theory of Databases" voert SUNDGREN [31] het begrip "frame of reference" $R_p(t)$ in, wat de kennis op een bepaald tijdstip t voorstelt van een persoon P. Hij voert een zogenaamde "consciousness function" c(k,P,t) in met waarden tussen 0 en 1 die als het ware een graad van bewustheid aangeeft van een bepaald kennisconcept k voor een persoon P op tijdstip t. c(k,P,t) = 0 iff $k \notin R_p(t)$, terwijl c(k,P,t) = 1 de situatie aangeeft dat P zich ogenblikkelijk bewust is van kennisconcept k op tijdstip t.

Sundgren definieert data als representatie van een stukje werkelijkheid m.b.v. een ander stukje werkelijkheid, of ook: data is de materialisatie van informatie, waarbij informatie weer equivalent is aan kennis, d.w.z. zulke zaken k waarvoor c(k,P,t) > 0.

Een infologisch model van een databank is een formele beschrijving van een representatie van een stuk werkelijkheid in die databank.

Sundgren definieert de betekenis, semantische inhoud, van een databericht dm, als het primaire conceptuele bericht pm en de informatie-inhoud van dm is een verzameling conceptuele berichten {sm}, waarbij elke sm van dm en $R_p(t)$ afleidbaar is en waarbij geen enkele sm bevat is in $R_p(t)$. Dit is ook op te schrijven in de vorm

$$\{sm\} = R_p(t+\Delta t) \setminus R_p(t),$$

aangenomen dat dm en dm alleen door P is herkend gedurende het tijdsinterval $<t,t+\Delta t>$. Een geabstraheerd model als dit moet dan het homomorfe beeld zijn van een werkelijkheid.

Ik neem aan dat u, geachte lezer, nu de draad van het verhaal wat kwijt bent en uzelf afvraagt waar dit formularium goed voor is. U treft dit verhaal aan in hoofdstuk 1 van Sundgren's boek en u zult een referentie ernaar in de overige zeven hoofdstukken tevergeefs zoeken. Op een dergelijke vrijblijvende wijze over abstractie praten staat natuurlijk erg

geleerd, maar is tamelijk zinloos. Helaas is deze behandeling van de stof kenmerkend voor Sundgren's boek, zodat we bij hem niet veel verder komen voor wat betreft abstracties van de werkelijkheid.

4.2. "In de beperking toont zich de meester"

Het ware wellicht ook beter geweest om ons te beperken tot een klein stukje van de werkelijkheid. In het volgende kiezen we een heel erg klein stukje werkelijkheid om aan de hand daarvan te kijken hoe er geabstraheerd wordt zodat op zinvolle wijze een databank gemaakt kan worden.

Beschouw een onderwijssituatie met studenten, vakken en leraren. Verder wil je voor het komende semester vastleggen welke studenten bepaalde vakken volgen en welke leraren die vakken geven. We abstraheren op dit moment dus van een groot aantal details, zoals welk lesmateriaal gebruikt zal worden, in welk lokaal het vak gegeven wordt, etc. Wel willen we naam en adres van student en leraar noteren.

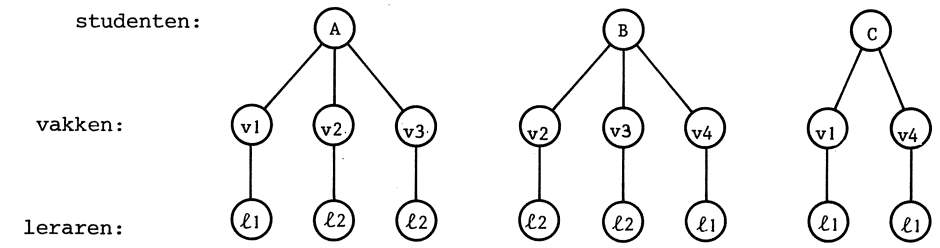
Vanuit het gezichtspunt van de student is een mogelijke aanpak aangegeven in de volgende declaratie:

```
type student = record
                    naam: alpha;
                    adres: alpha;
                    vakken: array[1..] of vak
               end;
       vak = record
                    naam: alpha;
                    leraar: record
                                 naam: alpha;
                                 adres: alpha
                            end
               end
```

Een concreet voorbeeld is:

student A kiest vakken v1, v2, v3 met leraren $\ell 1$, $\ell 2$, $\ell 2$
student B kiest vakken v2, v3, v4 met leraren $\ell 2$, $\ell 2$, $\ell 1$
student C kiest vakken v1, v4 met leraren $\ell 1$ en $\ell 1$.

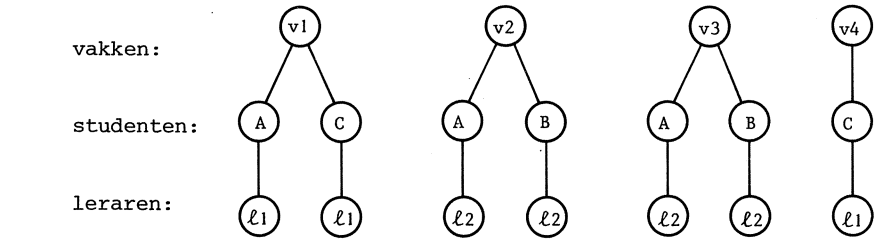
In een plaatje:



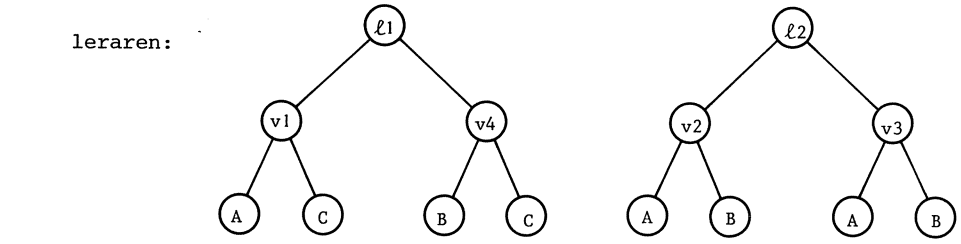
4.3. Het hierarchisch gestructureerde datamodel

Een voorbeeld hiervan is het IMS systeem. De fysische opslagstructuur is vrijwel identiek aan de hier getekende voorstelling in de vorm van een boom. Dat veel gegevens herhaald voorkomen (redundantie) moeten we voor lief nemen, zo komen naam en adres van de leraren liefst vier keer voor.

We kunnen ook andere modellen kiezen, wederom hierarchisch gestructureerd:



of:

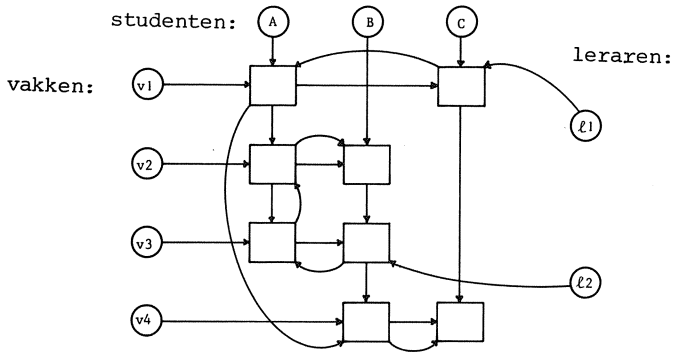


Ook op deze wijze moeten er redundante gegevens worden opgeborgen.

4.4. Het DBTG model

Het DBTG model is gebaseerd op netwerken (in grafentheoretische zin meer algemener dan de bomen in de boven besproken hierarchische structuur). Dit datamodel van CODASYL [8], grondslag van vele databankorganisaties, is hierop gebaseerd. In een plaatje komt de netwerkstructuur het best tot

uiting:



De doosjes zijn slechts verbindingselementen, ze zouden in principe nog informatie kunnen bevatten over de specifieke combinatie vak, student en leraar, bijvoorbeeld het cijfer dat de student voor dit vak bij die leraar zal behalen.

We merken op dat elk doosje in drie lijsten zit: met horizontale wijzers die vertrekken vanuit een vak, met verticale wijzers die vertrekken vanuit een student, en met gebogen wijzers die vertrekken vanuit een leraar. Informatie over studenten, vakken en leraren wordt nu zonder redundantie opgeborgen. Daarvoor in de plaats zijn wijzers gekomen en de gebruiker van een databank gebaseerd op dit netwerkmodel, moet goed op de hoogte zijn van het gebruik van wijzers. Willen we de datastructuur uitbreiden met bijvoorbeeld gegevens over lokaal en aanvangstijd, dan is dit niet eenvoudig.

4.5. Het relationeel gestructureerde model

Dit model voor een databankorganisatie is geïnspireerd op wiskundige relaties, voor het eerst ingevoerd door CODD [5]. Uitgangspunt is de idee van tuples in een deelverzameling van een Cartesisch product van verzamelingen, ook domeinen genoemd. Die deelverzameling wordt relatie genoemd. In ons voorbeeld hebben we de volgende verzamelingen:

SN(studentnamen) = {jan,piet,klaas}
SA(studentadressen) = {uil,spui}
VN(vaknamen) = {alg,calc,top,meetk}
LN(leraarsnamen) = {gros,kaas}
LA(leraarsadressen) = {lin,haar}.

De relaties zijn deelverzamelingen van S = SN * SA, V = VN, L = LN * LA en T = SN * V * LN, die we ook noteren met S = S(SN,SA), V = V(VN), L = L(LN,LA) en T = T(SN,V,LN).

S =

SN	SA
jan	uil
piet	spui
klaas	uil

L =

LN	LA
gros	lin
kaas	haar

T =

SN	V	LN
jan	alg	gros
jan	calc	kaas
jan	top	kaas
piet	calc	kaas
piet	top	kaas
piet	meetk	gros
klaas	alg	gros
klaas	meetk	gros

We zien dat adressen van leraren en studenten niet meer redundant voorkomen.

In bovenstaand model maakten we gebruik van de idee dat een student, voorgesteld door een tuple in S, geïdentificeerd wordt door zijn naam, evenzo voor een leraar. De basis hiervoor is het begrip functionele afhankelijkheid (FA) tussen twee domeinen A en B, dat we als volgt kunnen definiëren: Stel de werkelijkheid laat zich beschrijven door tuples uit A * B en bij elke A-waarde behoort in die werkelijkheid slechts één B-waarde, dan zeggen we B is functioneel afhankelijk van A, als volgt genoteerd: $A \rightarrow B$. Wil de databank, in de vorm van een relatie R, een goede representatie van die werkelijkheid geven, dan moet (op elk moment) gelden dat tuples uit de relatie R met gelijke A-waarde ook gelijke B-waarde bezitten:

$$\forall_{t,s}: t \in R \wedge s \in R \Rightarrow (t_A = s_A \Rightarrow t_B = s_B),$$

waar we met de index A en B de A- en B-waarde van het tuple aangeven.

Het begrip FA laat zich op triviale wijze uitbreiden tot het geval waarbij een aantal domeinwaarden $A_1, A_2, \ldots, A_n$ een andere domeinwaarde B uniek bepalen, hetgeen we noteren als

$$(A_1 + A_2 + \ldots + A_n) \rightarrow B.$$

Stel nu dat we een relatie R hebben met domeinen $D_1, \ldots, D_m$ (bijvoorbeeld R = S, D_1 = SN, D_2 = SA), en een deelverzameling α van deze domeinen heeft

de eigenschap $\alpha \rightarrow D_i$ voor alle i en bovendien α is zo klein mogelijk (d.w.z. als we één domein uit α weglaten, dan is er een (ander) domein D_j aan te wijzen, zodanig dat niet geldt $\alpha \rightarrow D_j$) dan heet α key van de relatie. De α-domeinwaarden identificeren in dit geval de tuples van de relatie R. Zo is, voor ons voorbeeld, {SN} key van S, {LN} key van L en {SN,V} key van T. De enige natuurlijke mogelijkheid om in een relatie, zoals hij concreet gerepresenteerd wordt in het computergeheugen, FA's aan te geven is d.m.v. key's. Zo heeft de relatie T, met {SN,V} als key, afgezien van triviale FA's als {SN} → SN, als enige FA: {SN,V} → LN, die in de databank is opgeborgen.

Het is vanzelfsprekend dat tussen de FA's van de databank en die van de werkelijkheid overeenstemming moet bestaan. Daarom is de keuze van domeinen en van key's in relaties van groot belang. Codd heeft verschillende definities gegeven van normaalvormen, die hierop neerkomen dat een relatie voldoet aan de (Boyce-Codd) normaalvorm indien inderdaad die overeenstemming bestaat. *) Als die overeenstemming niet bestaat dan is de kans op zogenaamde "update" problemen groot. Beschouw, bijvoorbeeld, de relatie T' = T'(SN,V,LN,LA). Willen we nu het adres LA van een leraar veranderen, dan moet dit op meerdere plaatsen in T' gebeuren. Ook geeft het toevoegen van een nieuwe leraar, die op dit moment nog geen studenten heeft toegewezen gekregen, problemen, als we aannemen dat de aparte relatie L niet wordt gehandhaafd. Evenzo raken we in dit geval het adres van leraar "gros" kwijt als hij in een bepaald semester geen les geeft.

Men zie in dit verband ook het recente artikel van TER BEKKE [36] over semantiek van gegevens in een databank.

4.6. De algebra van FA's

Om bij een gegeven werkelijkheid, in de vorm van een aantal domeinen en FA's tussen die domeinen, een geschikt aantal relaties te vinden is een interessant probleem. Zo stelt ARMSTRONG [1] een stel axioma's op waaraan FA's dienen te voldoen: (Laat α een verzameling domeinen voorstellen)

---

*) Men zie de standaardliteratuur (DATE [37], MARTIN [39] of TSICHRITZIS & LOCHOVSKY [38]) voor nadere details.

1. $\alpha \rightarrow \alpha$
2. $(\alpha \rightarrow \beta \wedge \beta \rightarrow \gamma) \Rightarrow \alpha \rightarrow \gamma$
3. $(\alpha \rightarrow \beta \wedge \alpha \subseteq \alpha' \wedge \beta' \subseteq \beta) \Rightarrow \alpha' \rightarrow \beta'$
4. $(\alpha \rightarrow \beta \wedge \alpha' \rightarrow \beta') \Rightarrow \alpha \cup \alpha' \rightarrow \beta \cup \beta'$.

Met deze axioma's kun je een gegeven verzameling F van FA's afsluiten tot een verzameling F^*. Vervolgens kun je kijken naar minimale bases die F^* voortbrengen; d.w.z. stel $\mathcal{B}$ is zulk een basis, dan moet gelden $\mathcal{B}^* = F^*$ en met minder dan $\mathcal{B}$ kun je niet toe.

BERNSTEIN [4] geeft algoritmen aan om uit te zoeken welke relaties en keys gekozen kunnen worden met de eigenschap dat de verzameling FA's afgeleid van de keys (zogenaamde "embedded functional dependencies") een basis vormen voor F^*. Hij bewijst dat de aldus, automatisch, geconstrueerde relaties aan de eisen voor de normaalvorm voldoen en in zekere zin minimaal zijn. SCHMID en SWENSON [28] bekritiseren de waarde van normaalvormen, en met name de rol van een FA.

Stel, in ons voorbeeld, dat een student twee adressen heeft met daarbij de indicatie oud of nieuw in een apart domein ON. Voor de relatie S' = S'(SN,SA,ON) is het nu onmogelijk met FA's aan te geven dat een student slechts één oud en één nieuw adres heeft. Ook is het onmogelijk het verband aan te geven tussen een student en zijn vakkenpakket. Bijvoorbeeld in de relatie S" = S"(SN,SA,ON,VN) kunnen we oud en nieuw adres en de keuze van zijn vakken onderbrengen, maar het is helaas onmogelijk met FA's aan te geven dat de student met unieke naam SN een aantal vakken heeft gekozen, maar dat er tussen adres en de keuze van de vakken geen enkel verband is.

FAGIN is hiermee in [10] verder gegaan en heeft het idee van meerwaardige afhankelijkheid (MA) ingevoerd. Stel je kunt de domeinen van een relatie R in drie groepen verdelen, α, β en γ; bijvoorbeeld voor S":

$$\alpha = \{SN\},\ \beta = \{SA,ON\},\ \gamma = \{VN\}.$$

Beschouw de verzameling $B_{a,c}$ bestaande uit alle β-waarden b waarvoor het tuple $\langle a,b,c\rangle \in R$. (Deze definitie is niet helemaal precies, omdat a, b en c in principe zelf weer tuples zijn, maar je kunt de definitie zonder veel moeite preciseren.) Dus bij een bepaalde waarde a en c van α en γ is $B_{a,c}$ de verzameling van alle bijbehorende β-waarden. Als nu geldt dat $B_{a,c}$ onafhankelijk is van c, dan zeggen we dat er een meervoudige afhankelijkheid (MA) bestaat tussen α en β. Notatie: $\alpha \twoheadrightarrow \beta$.

In ons voorbeeld S": of een student nu met algebra of meetkunde bezig is, zijn adressen zijn daarvan onafhankelijk: {SN} ↠ {SA,ON}. Tevens geldt: of je nu zijn oude adres of nieuwe adres neemt, de keuze van zijn vakken is daarvan onafhankelijk, dus: {SN} ↠ {VN}. Overigens is gemakkelijk in te zien dat als $\alpha \twoheadrightarrow \beta$ dat dan ook $\alpha \twoheadrightarrow \gamma$ (aangenomen α, β en γ hebben een lege doorsnede en hun vereniging bevat alle domeinen van R).
Fagin noteert $\alpha \twoheadrightarrow \beta \wedge \alpha \twoheadrightarrow \gamma$ daarom met $\alpha \twoheadrightarrow \beta|\gamma$.
Een niet bestaande MA is bijvoorbeeld {SN} ↠ {SA}, omdat de waarde van ON van invloed is op de waarde van SA.

Een interessant verband werd door Fagin aangetoond tussen het begrip MA en de algebraïsche begrippen projectie en join. De projectie Π_α van een relatie R is een nieuwe relatie, $\Pi_\alpha R$, waar slechts de domeinen, aangegeven in α, in voorkomen en de anderen zijn weggelaten. Bijvoorbeeld zijn S_A = S_A(SN,SA,ON) en S_V = S_V(SN,VN) projecties van S". De join van twee relaties R en S over gemeenschappelijke domeinen α, genoteerd als $R *_\alpha S$, bestaat uit alle mogelijke combinaties van tuples uit R en S met dezelfde waarden voor de α domeinen. Bijvoorbeeld is S" de join van S_A en S_V over SN.
Fagin toonde nu aan dat voor een relatie R een MA $\alpha \twoheadrightarrow \beta|\gamma$ dan en slechts dan geldt als $R = \Pi_{\alpha\cup\beta} R *_\alpha \Pi_{\alpha\cup\gamma} R$.

In [3] geven BEERI, FAGIN en HOWARD een uitgewerkte theorie over de axiomatisering van FA's en MA's. We merken op dat FA's en MA's gebruikt worden om te trachten er een stukje van de werkelijkheid mee te beschrijven. Een poging tot abstractie hiervan in termen van categorie-theorie komen we tegen bij RISSANEN [27]. Hij gaat vooral in op het onderwerp onafhankelijkheid tussen groepen domeinen.

4.7. Abstractie in de vorm van aggregatie en generalisatie

Hierboven werden pogingen beschreven om een stukje van de (gecompliceerde) werkelijkheid te vangen in wiskundige begrippen als FA's en MA's. Minder wiskundig, maar even interessant, is de wijze waarop SMITH & SMITH de werkelijkheid pogen te vangen d.m.v. abstractie. Zij ontbinden abstractie in twee onderling loodrechte componenten:

aggregatie en generalisatie.

Over aggregatie handelt [29], waar abstractie wordt gezien als het weglaten van alle details van een object, behalve die details die voor het begrip van een bepaald fenomeen van belang zijn. Zo zal het voor de relatie tussen

studenten, vakken en leraren weinig relevant zijn op welk adres de leraar woont. In deze relatie willen we de leraar als afzonderlijk object beschouwen, wellicht geïdentificeerd door zijn naam, maar geabstraheerd van nadere bijzonderheden als adres. De relaties tussen studenten, vakken en leraren zouden dan kunnen worden vastgelegd in objecten van het volgende type:

```
type student = aggregate [SN] {SN is key van student}
     SN: alpha {de naam van de student};
     SA: alpha {het adres van de student};
     VN: key vakken {VN moet key van vakken zijn}
     end;

     vak = aggregate [VN] {VN is key van vak}
     VN: alpha {de naam van het vak};
     LN: key leraar
     end;

     leraar = aggregate [LN] {LN is key van leraar}
     LN: alpha;
     LA: adres
     end;

var  vakken: collection of vak;
     leraren: collection of leraar;
```

Het is in deze opzet mogelijk om over de afzonderlijke vakken en leraren te beschikken. Merk op dat dit niet mogelijk was met de definitie in 4.2. Volgens die definitie konden wij alleen maar bij details van een leraar komen via een specifieke student A en een vak v1. Bijvoorbeeld, het adres van leraar $\ell 1$ is bereikbaar door (aangenomen v1 is het eerste vak van A):

```
A . vakken [1]. leraar . adres
```

In bovenstaande definitie kunnen leraren afzonderlijk worden aangegeven. Dat een implementatie in de vorm van relaties mogelijk is, in plaats van met wijzers, daarvoor zorgen aanwijzingen als "[*SN*]" en "*key*". In het laatste geval moet het aangegeven domein key zijn van het aangegeven object. Bijvoorbeeld:

VN: key vakken

in de definitie van student betekent dat VN key moet zijn in het object vakken. De reden dat hier *vakken* staat i.p.v. *vak* is dat het object een element moet zijn van de bestaande collectie vakken, in plaats van een volstrekt willekeurig opject van het type *vak*. Dit is weer van belang als we bij fysische implementatie een hashcode m.b.v. het totaal aantal verschillende objecten willen berekenen. Voor aldus geaggregeerde relaties worden eisen opgesteld ten aanzien van "update" handelingen, opdat de relatie "well-defined" is. Tussen Codd's normaalvorm en "well-definedness" blijkt geen simpele inclusierelatie te bestaan.

Werd in bovenstaande verhandeling de aggregatie als abstractie aangegeven in de vorm van Hoare's Cartesisch product [14], in een tweede verhaal [30] wordt een tweede type abstractie, namelijk generalisatie, geïnspireerd op de structuur van Hoare's "discriminated union". Hiermee wordt het volgende abstractieproces bedoeld:

Stel dat we studenten uit diverse faculteiten in een databank willen onderbrengen. De ene faculteit, α, wil gegevens zoals boven omschreven, een andere faculteit, γ, heeft naast vakken ook werkgroepen waaraan de student deel moet nemen, en een derde faculteit, β, heeft bovendien nog practica. Diverse soorten studenten dus, die veel gemeen hebben (naam, adres, vakken), maar ook behoorlijke verschillen vertonen (werkgroepen, practica). Het is zinvol om op een zeker niveau, namelijk het universiteitsniveau, van een gegeneraliseerde student te spreken. Smith & Smith stellen nu de volgende declaraties voor:

```
var gegen_student:
    generic
            fac = (α_student, β_student, γ_student)
      of
      aggregate [SN]
        SN: alpha {naam};
        SA: alpha {adres};
        fac: student_categorie {een van de drie waarden α_, β_ of
                                                    γ_student}
    end.
```

De structuur van α_, β_ en γ_studenten kan dan worden aangegeven als boven met student. De kreet *generic* betekent dat deze *gegen_student* een generalisatie is van de aangegeven objecten.

We zien hier een analogon met PASCAL's record-variant types. Een mogelijke notatie zou geweest kunnen zijn:

```
gegen_student: record
      SN: alpha;
      SA: alpha;
      case fac:(α_student, β_student, γ_student) of
      α_student: (vakken: - - );
      γ_student: (vakken: - -
                  werkgroepen: - - );
      β_student: (vakken: - -
                  werkgroepen: - -
                  practica: - - )
      end end;
```

Echter, dit is met opzet niet zo gekozen, omdat dan de mogelijkheid ontbreekt twee of meer orthogonale generalisaties aan te geven, bijvoorbeeld naast de facultaire generalisatie een generalisatie over de begrippen werk_student, beurs_student en pa's_student. Met de notatie van Smith & Smith is dit als volgt aan te geven:

```
var geg_student:

        generic
                fac = (α_student, β_student, γ_student);
                onderhoud = (werk_student, beurs_student, pa's_student)
        of
        aggregate [SN]

          SN: alpha;
          SA: alpha;
          fac: faculteits_categorie;
          onderhoud: onderhouds_categorie

        end;
```

Met PASCAL's notatie van in elkaar geneste varianten zouden we voor ons voorbeeld negen varianten moeten onderscheiden.

Smith & Smith geven een aardig twee-dimensionaal beeld van abstractie: in horizontale richting kan men zich de aggregatie voorstellen (uitbreiding van domeinen), in verticale richting de generalisatie (gelijksoortige objecttypen op elkaar gestapeld worden als een enkel objecttype beschouwd). Wederom worden voor "update" operaties preciese regels gegeven die er voor moeten zorgen dat de relaties "well-defined" blijven.

4.8. Andere abstracties

Nog vele andere onderzoekers hebben getracht en zullen trachten abstracties van de werkelijkheid, in de vorm van databanken en datastructuren, en abstracties van databanken in de vorm van wiskundige modellen, te beschrijven. We noemen DELLA VIGNA [9] en MILA MAJSTER [24] die grafentheoretisch bezig zijn; we noemen tenslotte MAIBAUM [23] die algebraïsch bezig is met Σ-algebra's, morphismen en categorieën.
[Wie had ooit gedacht dat zoiets alledaags als administratieve dataverwerking kon leiden tot zulke verheven zuiver-wiskundige zaken als algebra's en categorieën?]

5. CONCLUSIE

In het voorgaande is aangegeven dat er tussen abstractie van datastructuren en abstractie van de werkelijkheid in de vorm van databanken belangrijke verschillen zijn: in het eerste proces speelt het streven naar hanteerbaarheid, correctheid en verifieerbaarheid van gecompliceerde programma's een grote rol, terwijl het er in het tweede geval om gaat de gecompliceerde werkelijkheid zo goed mogelijk te vangen in de databank. In beide gevallen gaat het echter om klaarheid van begrippen die soms tot doel hebben dat een eenvoudige gebruiker er ook mee om kan gaan (zie ZLOOF [35] en CODD [6]) en een andere keer dat de onderliggende structuren met behulp van de wiskunde worden blootgelegd.

LITERATUUR

[1] ARMSTRONG, W.W., *Dependency structures of data base relationships,* Proceedings of IFIP 74, North Holland, 1974, pp. 580-583.

[2] BAYER, R. & K. UNTERAUER, *Prefix B-Trees,* ACM Transactions on Database Systems, Vol. 2, No 1 (March 1977), pp. 11-26.

[3] BEERI, C., R. FAGIN & J.H. HOWARD, *A complete Axiomatization for Functional and Multivalued Dependencies in Database Relations,* To appear in Proc. 1977 ACM SIGMOD conference. Report RJ 1977 (27927) 4/1/77, IBM Research Division, San José, Cal.

[4] BERNSTEIN, P.A., *Synthesizing Third Normal Form Relations from Functional Dependencies,* ACM Transactions on Database Systems, Vol. 1, No 4 (Dec. 1976), pp. 277-298.

[5] CODD, E.F., *A Relational Model of Data for Large Shared Data Banks,* Comm. ACM Vol. 13, No 6 (June 1970), pp. 377-387.

[6] CODD, E.F., *Seven steps to rendez-vous with the casual user,* Report RJ 1333 (# 20842) IBM Research Division, San José, Cal., 1974.

[7] DAHL, O.J., K. NYGAARD & B. MYHRHAUG, *The Simula 67 Common Base Language,* Norwegian Computing Centre, Oslo 1968.

[8] Data Base Task Group of CODASYL Programming Language Committee, Report April 1971.

[9] DELLA VIGNA, P.L., *Data structures and graph grammars,* Proceedings of the 1st ECI conference, K. SAMELSON (ed.), Springer, 1976, pp. 783-793.

[10] FAGIN, R., *Multivalued Dependencies and a New Normal Form for Relational Databases,* ACM Transactions on Database Systems, Vol. 2, No 3, pp. 262-278.

[11] GHOSH, S.P., *Data base organization for data management,* Academic Press, 1977.

[12] GUTTAG, JOHN, *Abstract Data types and the Development of Data Structures,* Comm. ACM Vol. 20, No 6 (June 1977), pp. 396-404.

[13] HASSITT, A., J.W. LAGESCHULTE & L.E. LYON, *Implementation of a High Level Language Machine,* Comm. ACM Vol. 16, No 4 (April 1973), pp. 199-212.

[14] HOARE, C.A.R., *Notes on datastructuring,* in: APIC Studies in Data Processing No 8: *Structured Programming,* Academic Press, New York 1972, pp. 83-174.

[15] KNUTH, D.E., *The Art of Computer Programming I: Fundamental Algorithms,* Addison-Wesley, Reading Mass. (1969).

[16] KNUTH, D.E., *The Art of Computer Programming III: Sorting and Searching,* Addison-Wesley, Reading Mass. (1973).

[17] LAMPSON, B.W., J.J. HORNING, R.L. LONDON, J.G. MITCHELL & G.L. POPEK, *Report on the Programming Language Euclid,* SIGPLAN Notices, Vol. 12, No 2 (Febr. 1977).

[18] VAN LEEUWEN, J., *The complexity of data organization,* in: K.R. APT, J.W. DE BAKKER (eds.), *Foundations of Computer Science II,* Mathematical Centre Tracts 81, Mathematisch Centrum, 1976.

[19] LIN, C.S., D.C.P. SMITH & J.M. SMITH, *A Rotating Associative Memory,* ACM Transactions on Database Systems, Vol. 1, No 1 (March 1976), pp. 53-65.

[20] LISKOV, BARBARA & STEPHEN ZILLES, *Programming with abstract data types,* SIGPLAN Notices Vol. 9, No 4 (April 1974), pp. 50-58.

[21] LISKOV, BARBARA, ALAN SNYDER, RUSSELL ATKINSON & CRAIG SCHAFFERT, *Abstract Mechanisms in CLU,* Comm. ACM Vol. 20, No 8 (August 1977), pp. 564-576.

[22] LUCKHAM, D.C., *Program Verification and Verification-Oriented programming,* Information Processing 77, B. GILCHRIST (ed.), IFIP, North Holland, 1977, pp. 783-794.

[23] MAIBAUM, T.S., *Mathematical Semantics and model for data bases,* Information Processing 77, B. GILCHRIST (ed.), IFIP, North Holland 1977, pp. 133-138.

[24] MAJSTER, MILA E., *A model for Data Structures,* Proceedings of the 1st ECI conference, K. SAMELSON (ed.), Springer, 1976.

[25] OZKARAHAN, E.A., S.A. SCHUSTER & K.C. SEVCIK, *Performance Evaluation of a Relational Associative Processor,* ACM Transactions on Database Systems Vol. 2, No 2 (June 1977), pp. 175-195.

[26] RAMAMOORTHY, C.V. & H.F. LI, *Pipeline Architecture,* Computing Surveys, Vol. 9, No 1 (March 1977), pp. 61-102.

[27] RISSANEN, J., *Independent Components in Relations,* ACM Transactions on Database Systems, Vol. 2, No. 4 (December 1977), pp. 317-325.

[28] SCHMID, H.A. & J.R. SWENSON, *On the Semantics of the Relational Data Model,* Proc. of ACM SIGMOD Conf., San José, Cal., May 1975, pp. 211-223.

[29] SMITH, J.M. & D.C.P. SMITH, *Database Abstractions: Aggregation,* Comm. ACM Vol. 20 No 6 (June 1977), pp. 405-413.

[30] SMITH, J.M. & D.C.P. SMITH, *Database Abstractions: Aggregation and Generalization,* ACM Transactions on Database Systems, Vol. 2, No 2 (June 1977), pp. 105-133.

[31] SUNDGREN, BO, *Theory of Databases,* Petrocelli Charter, New York, 1975.

[32] WEGNER, P., *The Vienna Definition Language,* Computing Surveys Vol. 4, No 1 (March 1972), pp. 5-63.

[33] YAN, S.S. & H.S. FUNG, *Associative Processor Architecture - A Survey,* Computing Surveys, Vol. 9, No 1 (March 1977), pp. 3-28.

[34] ZILLES, S.N., *Algebraic Specifications of Data Types,* Computation Structures Group Memo 119, Laboratory for Computer Science MIT, March 1975.

[35] ZLOOF, MOSHI. M. & S. PETER DE JONG, *The system for Business Automation (SBA): Programming Language,* Comm. ACM, Vol. 20, No 6 (June 1977), pp. 385-395.

[36] BEKKE, J.H. TER, *Semantiek van gegevens als uitgangspunt voor manipulatie van gegevens,* Informatie 19, No 9 (Sept. 1977) pp. 491-497.

[37] DATE, C.J., *An Introduction to Data Base Management Systems,* (second printing) Addison-Wesley, 1977.

[38] TSICHRITZIS, D.C. & F.H. LOCHOVSKY, *Data Base Management Systems,* Academic Press (1977).

[39] MARTIN, J., *Computer Data Base Organization,* Prentice Hall, 1975.

[40] SENKO, M.E., *Data structures and data accessing in data base systems: past, present, future,* IBM Systems Journal, No 3, 1977.

ABSTRACTE DATATYPEN

L.G.L.T. MEERTENS
Mathematisch Centrum

1. ABSTRACTIE

Tegenwoordig wordt algemeen ingezien dat kenmerkend voor hogere programmeertalen het vermogen is tot abstraheren van implementatiedetails, zoals gegevensrepresentatie, geheugenbeheer, besturing. Waar bij lagere programmeertalen het hanteren van zulke abstracties vooral berust op zelfdiscipline van de programmeur, bieden hogere programmeertalen hetzij de abstracties kant en klaar aan, hetzij de instrumenten waarmee de noodzakelijke discipline gecreëerd kan worden. De ontwikkeling van programmeertalen is nauw verbonden met het groeien van het inzicht dat abstractiemechanismen wezenlijk zijn voor de taak van het programmeren.

Op een presenteerblaadje aangedragen abstracties vervullen een belangrijk, maar toch altijd beperkt, deel van de abstractiebehoefte. Veel van het recente werk op programmeertaalgebied is dan ook gericht op algemenere mechanismen waarmee de programmeur naar behoefte zijn eigen abstracties kan scheppen. Een aantal lijnen in deze ontwikkeling blijken nu samen te komen in het begrip "abstract datatype". Een werkelijke synthese is nog niet bereikt, en deze voordracht geeft dan ook geen afgerond beeld. Evenmin kan in dit beperkte bestek een werkelijk overzicht worden geboden van de vele punten die daarbij aan de orde zijn; het volgende is een meer afstandelijke beschouwing om althans de hoofdlijnen te vinden.

"Abstractie" op zich is een nogal abstract, en daardoor misschien wat mistig, begrip. Hoe kan zoiets als een programmeertaal de programmeur helpen abstracties te scheppen? Het antwoord is: in wezen op dezelfde wijze als waarop een natuurlijke taal dat mogelijk maakt. Abstraheren is het afzien van details. Door de daad van abstractie verdwijnen die details niet in een of andere objectieve zin, maar ze verdwijnen uit ons bewustzijn. Een taal maakt het mogelijk abstracties te hanteren door ze met etiketten te benoemen, zoals "Fikkie", of, op een wat hoger abstractie-

niveau, "hond". Wezenlijk voor de ontwikkeling van de menselijke cultuur en samenleving is het vermogen nieuwe abstracties te creëren van vaak onvoorstelbaar hoog niveau en daarbij horende termen in omloop te brengen, zoals "model", "machine", "Eurocommunisme" of "abstract datatype".

Om weer in de programmeerwereld terug te keren: twee abstracties die daar een rol kunnen spelen zijn "stelsel van lineaire vergelijkingen" en het "oplossen" van zo'n stelsel. Kortom, een (abstract) object en een operatie daarop. Om met een computer een stelsel van lineaire vergelijkingen op te lossen zijn nodig: een representatiekeuze (datastructuur) voor het object *en* een uitwerking in elementaire, op die datastructuur toepasbare, operaties (procedure) van een gekozen oplossingsmethode (algoritme). De beschrijving, in een programmeertaal, van datastructuur en procedure tezamen, vormt een programma.

Het is opvallend dat voor operaties al door de oudste programmeertalen een abstractie*mechanisme* geboden wordt (de subroutine), maar dat het zo lang geduurd heeft voordat voor objecten wezenlijk meer ter beschikking kwam dan kant-en-klare abstracties (datastructuren zoals array).

2. LIJNEN IN DE ONTWIKKELING

2.1. Records

Het oudste voorstel voor een abstractiemechanisme voor datatypen is het *records*-voorstel van HOARE [1] uit 1965. Dit voorstel heeft een grote invloed gehad op de verdere ontwikkeling van programmeertalen. Het idee zelf is van een verbluffende simpelheid: door een definitie van een "recordklasse" wordt een nieuw datatype geïntroduceerd van samengestelde objecten; ieder object is een eenvoudig conglomeraat van andere objecten, de "velden", die hun eigen datatype hebben. Arrays zijn ook een conglomeraat van objecten, maar alle van hetzelfde type. Bovendien kunnen de velden van een record ieder alleen door hun eigen selector bereikt worden, in plaats van door een aritmetische en dynamisch bepaalde index. Het aantal velden is dan ook vast.

De reden waarom dit als een *mechanisme* bestempeld kan worden, is dat de recordklassedefinitie een "etiket" introduceert voor een nieuw datatype. Zo'n etiket maakt het bovendien mogelijk recursieve definities op te schrijven, wat de uitdrukkingsmacht verhoogt. Vormen van recordklassedefinities hebben sindsdien een plaats gekregen in alle respectabele hogere program-

meertalen, zoals ALGOL W, ALGOL 68 of PASCAL. Een bijzondere vorm is te vinden in SIMULA 67, en daarop zal later dieper worden ingegaan.

2.2. Extensibiliteit

Een veel algemenere - in feite *te* algemene - benadering is die van de *extensibiliteit* (SCHUMAN [2]). Een extensibele (uitbreidbare) taal is een programmeertaal waarin de geoefende programmeur naar behoefte nieuwe syntactische constructies en de bijbehorende semantiek kan definiëren, dus zijn eigen probleemgerichte programmeertaal kan maken met de door hem gewenste abstracties. De pogingen zulke talen te definiëren zijn op weinig uitgelopen. In feite was dit project te ambitieus; het hield een onderschatting in van de problemen van het ontwerpen van programmeertalen. Een programmeertaal is immers meer dan een losse verzameling constructies die naar believen ingeperkt of uitgebreid kan worden.

Dat wil niet zeggen dat er niets mogelijk is. Zo wordt ALGOL 68, hoewel niet als extensibele taal ontworpen, door sommigen toch als zodanig beschouwd. Dit is te danken aan een combinatie van drie elementen:

a) de aanwezigheid van records (in de ALGOL-68-terminologie: structured values) en van een mode-definition om hun datatypen een etiket te verschaffen;
b) de "orthogonaliteit", waardoor zulke mode-definitions voor willekeurige datatypen gebruikt kunnen worden;
c) het ontwerpen van ALGOL 68 als een kerntaal plus een "standard-prelude" van toepassingsgerichte definities, waartoe het abstractiemechanisme van de operation-definition in de kerntaal is opgenomen.

Hierdoor kan de gebruiker van ALGOL 68 zijn eigen definities maken voor bijvoorbeeld een datatype *matrix*, en schrijven:

```
begin matrix a, b, c;
      read ((a,b,c)); print (a+b×c)
end.
```

Dit vereist dat zo'n programmaatje wordt ingebed in een groter geheel, als volgt:

```
begin mode matrix = [1:n, 1:n] real;
      op + = (matrix a,b) matrix : ...;
      op × = (matrix a,b) matrix : ...;

      <programmaatje>
end.
```

De facto is hiermee een uitbreiding van ALGOL 68 met het datatype *matrix* gecreëerd.

Een aantal dingen zijn minder gelukkig. Een is dat de definitie van *matrix* met de operaties + en × geen pakket vormt. De enige bijdrage die de programmeur daartoe kan leveren, is het textueel bijeenplaatsen van de definities. Een ander is dat de bereikte abstractie maar betrekkelijk is. Bij de gegeven definitie is het heel zinvol in het programmaatje een toekenning *a:= b* te schrijven. Maar stel dat het om dunbezaaide matrices gaat, en dat de programmeur, om ruimte te besparen, gedefinieerd heeft:

```
mode matrix = struct (int i,j, real el, ref matrix next).
```

Dan wordt de betekenis van zo'n toekenning direct heel anders. Eigenlijk zou := voor zo gerepresenteerde matrices opnieuw gedefinieerd moeten worden, maar ALGOL 68 laat dat niet toe.

Een taal die wel vanuit de gedachte van extensibiliteit is ontworpen, EL 1 (WEGBREIT [3]) (EL = Extensible Language), biedt ongeveer langs dezelfde lijnen als ALGOL 68 een extensie-mechanisme. In EL 1 zijn de gebreken van deze aanpak deels opgevangen; zo is het wel mogelijk de toekenning passend te herdefiniëren.

2.3. Correctheid van gegevensrepresentatie

In HOARE [4] wordt de betekenis van SIMULA-classes behandeld voor het bewijs van de correctheid van gegevensrepresentaties. De SIMULA-class is ontwikkeld voor de communicatie tussen samenwerkende processen. Hiertoe wordt een "klassiek" record, waarvan de velden overeenkomen met de geschaarde variabelen, uitgebreid met een aantal velden die operaties op die variabelen bevatten, en bovendien een initialiserende operatie. Hoare wijst er nu op dat dit een ontbinding van het correctheidsbewijs voor een datatype mogelijk maakt in (a) het correctheidsbewijs voor de gegevensrepresentatie en (b) het correctheidsbewijs voor het programma in termen van de *abstracte* eigenschappen van het datatype.

Om bijvoorbeeld een queue van integers te definiëren, schrijven we in SIMULA 67

```
class intq;
begin integer front, tail, length;
      integer array cell [1:large];

      procedure queue (i); integer i;
      if length = large then error else
      begin tail := if tail = large then 1 else tail+1;
            cell [tail] := i; length := length+1 end;

      integer procedure unqueue;
      if length = 0 then error else
      begin unqueue := cell [front];
            front := if front = large then 1 else front+1;
            length := length - 1 end;

      front := 1; tail := large; length := 0
end.
```

Indien een variabele x gedeclareerd is door *ref (intq) x*, en op x worden alleen de operaties *queue* en *unqueue* toegepast, dan gedraagt x zich als een keurige queue. We kunnen bijvoorbeeld schrijven:

```
i := x•unqueue; if i > 0 then x•queue (i-1).
```

In feite bepleit Hoare dus het gebruik van de SIMULA-class als abstractiemechanisme om voor objecten nieuwe datatypen te creëren. Een duidelijke tekortkoming is echter het feit dat niets de programmeur let om bijvoorbeeld te schrijven: *x•length := x•unqueue*. Hiermee wordt een wezenlijke invariant van de representatie verstoord. Dit doet afbreuk aan de abstractie. Daarnaast geldt als tekortkoming dat deze methode alleen kan worden toegepast als alle operaties op het nieuwe datatype monadisch zijn; een datatype als *complex* kan zo dus niet zinvol worden ingevoerd.

2.4. Modulariteit

Tussen de verschillende componenten van een systeem bestaat interactie, maar niet iedere component heeft rechtstreeks interactie met iedere

andere component. In een groot systeem zijn altijd deelsystemen aan te wijzen die slechts op beperkte wijze in contact treden met de rest van het systeem. Dit kan door de ontwerper van het systeem bewust bevorderd worden: hij kan het systeem opdelen in *modulen*, waarbij een gegeven moduul desgewenst vervangen kan worden door een passend ander moduul. Hierbij betekent "passend": hetzelfde gedrag vertonend ten opzichte van de buitenwereld van het moduul; met andere woorden, van de inwendige werking wordt geabstraheerd. Een specificatie van de voor "passen" noodzakelijke eigenschappen wordt *interface* genoemd. Dit zegt op zich nog niets over de juiste methodiek van opdeling, al zal het duidelijk zijn dat in het algemeen ernaar gestreefd moet worden de interface (en dus het aantal interacties over de grens van het moduul heen) overzichtelijk klein te houden.

De interacties waar het bij het programmeren om gaat, worden bepaald door het *toepassen* van *gedefinieerde* etiketten (identifiers, operatoren e.d.). De SIMULA-class-definitie kan in die zin beschouwd worden als een moduul dat, in het voorbeeld van 2.4, de etiketten *queue* en *unqueue* (maar helaas ook *front*, *tail*, *length* en *cell*) "exporteert".

Een redelijk uitgewerkt voorstel voor een abstractiemechanisme voor modulen volgens deze benadering is voor het eerst gegeven in SCHUMAN [5]. Een mogelijke definitie voor een moduul zou kunnen zijn:

```
module matrices =
begin mode matrix = [1:n,1:n] real;
      op + = (matrix a, b) matrix : ...;
      op × = (matrix a, b) matrix : ...;
      skip
end.
```

Hiermee is slechts een van de bezwaren ondervangen die in paragraaf 2.2 genoemd zijn bij het gebruik van ALGOL 68 als extensibele taal. Van de in 2.3 genoemde tekortkomingen bij SIMULA-classes is de beperking tot monadische operaties vervallen.

2.5. Inkapseling

De reden dat modulen op zich nog tekort schieten als abstractiemechanisme, is dat de interface bepaald wordt door de in het moduul geplaatste definities. Voor zover deze alleen bedoeld zijn om de inwendige werking te specificeren, zouden ze niet naar buiten mogen doordringen, dus in het

moduul *ingekapseld* moeten worden. Voorstellen om de ongewenste export van etiketten te verhinderen door ze bij hun definities tot "hidden" of "implicit" te verklaren (of door alleen die etiketten te exporteren die van een uitdrukkelijke exportvergunning voorzien zijn) zijn al in een vroeg stadium geopperd en ook in SCHUMAN [5] te vinden. Dat dit niet voldoende is, is opgemerkt in MORRIS [6]: zodra we, volgens deze aanpak, het *etiket* van een datatype exporteren, dringen ook *implementatiedetails* naar buiten. Het is alles of niets. Zijn oplossing is het datatype aan de buitenkant van het moduul als een nieuw, primitief type te beschouwen.

Dit sluit aan bij eerder werk van MEALY [7], BALZER [8], ROSS [9] en MORRIS [10] zelf. De vernieuwing van Morris is dat hij een gedachte die steeds meer toegespitst was op een *dynamische* oplossing voor het veiligheidsprobleem in bijvoorbeeld bedrijfssystemen, overzet in een algemeen *linguistisch* concept, het datatype. In feite zijn we hier beland bij een volledig abstractiemechanisme voor datatypen. Merkwaardig genoeg heeft deze publicatie geen aantoonbare invloed gehad op wat als de werkelijke "doorbraak" beschouwd kan worden, mogelijk doordat er geen syntactische notatie en voorbeelden in gegeven worden.

2.6. Een stapje terug in de geschiedenis

Alvorens verder te gaan, is het aardig nog een oudere publicatie (WILKES [11]) te noemen. Hoewel dit artikel uit 1968 "geïsoleerd" staat - geen van de andere publicaties verwijst ernaar -, bevat het toch al de kern van de gedachte, maar dan meer toegepast op (de methodiek van) programmeertaal*ontwerp* dan op programmeertalen zelf. Wilkes merkt op dat er bij een taal als ALGOL 60 slechts een klein gedeelte is van de syntaxis dat afhankelijk is van het type van de objecten. Hij stelt dan voor te komen tot een *volledige* scheiding van enerzijds "uitwendige" syntaxis, die een volledige taal beschrijft, maar *abstract* in de zin dat daardoor op generlei wijze de aard van de objecten of hun implementatie wordt vastgelegd, en anderzijds "inwendige" syntax, waarin uitsluitend dat laatste gebeurt. Als voorbeeld wordt een recept gegeven om forten klein te krijgen. Hierbij wordt verondersteld dat in een buitenblok een *flag F* gedeclareerd is. Met iets gemoderniseerde notatie:

```
begin gun A;
      while flying F
      do load A;
         fire A
      od
end.
```

De uitwendige taal ligt eenmalig vast, maar de inwendige taal kan per toepassingsgebied opnieuw gekozen worden. Voor numerieke toepassing ligt het meer voor de hand het type *real* op te nemen dan het type *gun*. Wilkes doet geen voorstel hoe de inwendige taal (anders dan bijvoorbeeld in machinecode) geïmplementeerd zou kunnen worden, maar suggereert wel tot slot dat langs deze lijnen een oplossing gevonden zou kunnen worden voor het probleem een voldoende bedreven programmeur in staat te stellen een taal uit te breiden door nieuwe datatypen toe te voegen.

3. DE DOORBRAAK

Als hier het artikel van LISKOV & ZILLES [12] tot "doorbraak" bestempeld wordt, is dat voornamelijk vanwege de invoering en heldere omschrijving van het begrip "abstract datatype". Een abstract datatype definieert in hun visie een klasse van abstracte objecten die *volledig wordt gekarakteriseerd* door de voor deze objecten beschikbare operaties. Deze gedachte wordt vervolgens gevangen in een taalconcept, de "cluster", waaromheen een taal CLU is gebouwd. Zónder overdrijving kan gesteld worden dat dit een oplossing is langs de door Wilkes aangegeven, maar niet opgepakte, lijnen.

Een cluster is een moduul dat een datatype definieert door een representatie te geven en operaties in termen van de representatie. Van buitenaf is de representatie echter volledig ontoegankelijk. Dit laatste is het belangrijkste verschil met de SIMULA-class, waardoor de cluster een volwaardig abstractiemechanisme wordt. (Daarnaast zijn er verschillen van secundair belang: een cluster-definitie kan geparametriseerd worden met een type, en de operaties zijn niet aan een instantie van een object, maar aan de cluster-definitie gebonden.)

Het is niet moeilijk kritiek uit te oefenen op deze grofstoffelijke uitwerking van een pure gedachte. In sommige opzichten is dit zelfs een stap terug ten opzichte van de in MORRIS [6] aangegeven oplossing. Toch zou deze kritiek in zoverre niet geheel gerechtvaardigd zijn, dat een

aantal overblijvende problemen nu eenmaal buitengewoon moeilijk zijn, en dat een bevredigende integratie van de sindsdien in de literatuur gegeven fragmentarische oplossingen nog niet mogelijk is gebleken. Hierop wordt later nog ingegaan.

Belangrijker is kritiek op de opvatting dat abstracte datatypen door een speciaal taalconcept als bijvoorbeeld een cluster belichaamd zouden moeten worden. Wat is nu eigenlijk het abstracte aan een abstract datatype? Een abstract datatype is een *door de gebruiker gedefinieerd* datatype. Een datatype in een programmeertaal is gekarakteriseerd door de operaties die beschikbaar zijn voor de objecten van dat type. Daarnaast is uiteraard een implementatie nodig. Het bezwaar van eerdere abstractiemechanismen voor door de gebruiker gedefinieerde datatypen was dat onbedoeld details van de door hem gegeven implementatie, zoals de gekozen datastructuur, naar buiten dringen. Dit uit zich dan in een "teveel" aan beschikbare operaties, operaties die niet in de bedoelde karakterisering zitten, waardoor ook niet het bedoelde datatype wordt bereikt. Als wij bijvoorbeeld definiëren

mode queue = [*1 : large*] *int*,

krijgen we ongevraagd de beschikking, voor een variabele *queue x*, over een element als *x*[*7*]. De bedoeling van "abstracte" datatypen is een abstractiemechanisme te verschaffen dat voldoende krachtig is om een *zelfde* abstractieniveau te bereiken als bij voorgebakken datatypen, met inkapseling van de niet voor buitenstaanders bedoelde implementatiedetails. Hieruit volgt dat de naamgeving "abstract datatype" niet de gelukkigste is: een abstract datatype is op zich niet abstracter dan welk ander datatype ook. De wezenlijke vernieuwing is de inkapseling van de voor de implementatie gebruikte datastructuur. Bij programmeertaalontwerp is het in het algemeen niet verstandig voor een zo simpel en wezenlijk concept een hele constructie met toeters en bellen te verzinnen; veel beter is het een algemener toepasbare constructie, zoals modulen, van een uitdrukkingsmogelijkheid voor het nieuwe concept te voorzien. Zo zouden we kunnen schrijven:

mode primitive queue = [*1 : large*] *int*.

4. AXIOMATISCHE SPECIFICATIE

Een nieuwe lijn op het terrein van abstracte datatypen is de axiomatische

karakterisering van een algebraïsche structuur voor een datatype (GUTTAG [13]). De gedachte hierachter is dat de algoritmische of operationele karakterisering door een implementatie niet de beste zou zijn, doordat deze vaak te willekeurig is, teveel details bevat waarvan weer geabstraheerd moet worden, en moeilijk te begrijpen is omdat algoritmen in het algemeen vaak moeilijk te doorgronden zijn.

Het is niet zo duidelijk waarom een axiomatische specificatie speciaal voor *abstracte* datatypen van belang zou zijn. Daarbij is er immers nog althans een expliciete operationele karakterisering. Bij voorgebakken datatypen ontbreekt deze meestal, althans in de taaldefinitie, of wordt beschreven in natuurlijke taal. Voorzover een compiler als taaldefinitie beschouwd mag worden, levert deze een heel wat ontoegankelijker beschrijving op.

Bovendien is het nog maar de vraag of axiomatische specificatie altijd de meest duidelijke is. Zelfs wiskundigen, die van axioma's toch meer kaas gegeten zullen hebben dan de meeste programmeurs, zullen moeite hebben een adequaat *en* duidelijk axiomastelsel voor bijvoorbeeld verzamelingen uit hun mouw te schudden. Het gaat in feite om het algemene probleem hoe de semantiek van een datatype te definiëren.

Door te kijken naar de gevallen waar de methode het meest succesvol blijkt, valt er toch een aardige gedachte voor mode-definities uit te halen, die ook aansluit bij werk van HOARE [14] en VON HENKE [15]. Een voorbeeld is misschien het duidelijkst:

tree ::= nil|atom(val : item)| pair (left, right: tree).

Dit is een compacte notatie voor wat in de notatie van Guttag zou luiden:

```
NIL      :        → Tree
ATOM     :  Item → Tree
VAL      :  Tree → Item
PAIR     :  Tree × Tree → Tree
LEFT     :  Tree → Tree
RIGHT    :  Tree → Tree
IS_NIL?  :  Tree → Boolean
IS_ATOM? :  Tree → Boolean
IS_PAIR? :  Tree → Boolean
```

```
IS_NIL?(NIL) = true
IS_NIL?(ATOM (i)) = false
IS_NIL?(PAIR(t1,t2)) = false
IS_ATOM?(NIL) = false
IS_ATOM?(ATOM (i)) = true
IS_ATOM?(PAIR(t1,t2)) = false
IS_PAIR?(NIL) = false
IS_PAIR(ATOM (i)) = false
IS_PAIR(PAIR(t1,t2)) = true
VAL(NIL) = error
VAL(ATOM (i)) = i
VAL(PAIR(t1,t2)) = error
LEFT(NIL) = error
LEFT(ATOM (i)) = error
LEFT(PAIR(t1,t2)) = t1
RIGHT(NIL) = error
RIGHT(ATOM (i)) = error
RIGHT(PAIR(t1,t2)) = t2.
```

Het aardige is dat deze manier van definiëren een unificatie biedt van drie concepten:

(i) records, als in

complex ::= pair(re,im:real);

(ii) (disjuncte) unies, als in

arithmetic ::= i(val:int) | r(val:real);

(iii) scalairen à la PASCAL, als in

colour ::= red|blue|green.

Bovendien kan door een compiler uit zo'n definitie rechtstreeks een (mogelijk niet optimaal efficiënte) implementatie worden geconstrueerd. Het is echter beslist niet zo dat alle datatypen zich op deze wijze laten vangen.

5. DE EMANCIPATIE VAN ABSTRACTE DATATYPEN

Langs ieder van de lijnen die elkaar ontmoeten in het begrip "abstract datatype" vinden nog verdere ontwikkelingen plaats. In hoeverre deze ontwikkelingen divergeren valt nog nauwelijks te bezien. Voor de verder geïnteresseerde lezer worden hier enkele vermeld:

- bij correctheid: de (zwevende) programmeertaal Alphard (WULF, LONDON & SHAW [16]);
- bij modulariteit: het geavanceerd gebruik van definitiemodulen (SWIERSTRA [17]);
- bij inkapseling: nauwgezette interface-bepaling (KOSTER [18]).

Opvallend is dat van het extensibiliteitsfront nog weinig nieuws te melden valt. Als we het abstracte datatype door een extensibiliteitsbril bekijken, zien we dat een klein maar belangrijk gedeelte van het programmeertaalontwerp in handen van de gebruiker gesteld is. Juist omdat het om een redelijk begrensd deel gaat, is er enige vooruitgang geboekt op een terrein dat in zijn algemeenheid hopeloos ingewikkeld is. Bij de meeste benaderingen is echter de onafhankelijkheid van de voor de implementatie gekozen representatie bewerkstelligd door een uniforme notatie - meestal de procedure-aanroep - voor operaties op een object van een abstract type. Hierdoor dreigen abstracte datatypen alsnog achtergesteld te blijven bij tevoren gegeven, in de taaldefinitie ingebouwde, typen, die hun eigen notaties hebben.

Deeloplossingen zijn de mogelijkheid de assignment zelf te definiëren, of om "generators" (CLU) of "iterators" (Alphard) te definiëren. Zo wordt het mogelijk een for-statement te schrijven die bijvoorbeeld de elementen uit een object van een nieuw type *set* afloopt. Voor een werkelijke emancipatie van abstracte datatypen is echter meer nodig.

De ontwikkelde deeloplossingen wijzen hierbij een weg waarlangs nog veel te bereiken valt. De hoofdgedachte is dat het aantal begrippen in een programmeertaal met voorgebakken notatie en semantiek, zoals toekenning of iteratie, weliswaar groot kan zijn (voor ALGOL 68 een honderdtal), maar toch noodzakelijkerwijze eindig. Maar een deel daarvan is op een of andere wijze verbonden met het type-begrip. Door ieder daarvan, een voor een, open te breken voor door de gebruiker gedefinieerde datatypen, kunnen de privileges van de traditionele datatypen gemeengoed worden.

Een poging tot volledige uitwerking van deze gedachte zou te ver voeren. Moge daarom een voorbeeld volstaan: de vector (een-dimensionale array). Voor zo'n datatype kunnen we onderkennen: het type van de elementen, het indextype (traditioneel een segment van de integers), het creëren van een datastructuur bij een definitie, het toekennen van een waarde aan een element en het opvragen van de waarde.

We kunnen nu afspreken dat we een type [*index*] *item* beschrijven als

een object

struct (repr: <t>, assign: proc (<t>, index, item),
val: proc (<t>, index) item).

Hierin staat *<t>* voor het type van de datastructuur die voor de representatie gebruikt wordt. (Zie ook GRIES & GEHANI [19].)

Een stukje programma als

```
[index] item a;
...
a[i] := a[j]
```

wordt dan (ongeveer) geïnterpreteerd als

```
<t> a;
...
assign (a,i,val(a,j)).
```

Op zo'n manier kunnen strings als index toegelaten worden, of kan voor dunbezaaide vectoren een ruimtebesparende datastructuur gekozen worden. Een standaarddefinitie voor index-typen die een segment van de integers zijn kan in de taaldefinitie zijn opgenomen.

Wil de hier ontvouwde gedachte toekomst hebben, dan zal het noodzakelijk zijn de exotische vormenrijkdom waarmee sommige bestaande programmeertalen beladen zijn, wat uit te dunnen. Het ontwerp van een programmeertaal die ruimte biedt aan werkelijk geëmancipeerde abstracte datatypen zal echter ook dan niet simpel zijn. Een aantal lastige problemen is opgesomd in MEERTENS [20]. Het ziet er wel naar uit dat de meeste daarvan op een redelijk aanvaardbare manier zijn op te lossen.

LITERATUUR

[1] HOARE, C.A.R., *Record handling,* ALGOL Bulletin 21.3.6 (1965), *eveneens in: Programming Languages,* F. Genuys (ed.), Academic Press, New York, 1968.

[2] SCHUMAN, S.A. (ed.), Proc. Int. Symp. on Extensible Languages, SIGPLAN Notices 6 (December 1971)/IBM-France Centre Scientifique de Grenoble Report no. FF2.0143 (1971).

[3] WEGBREIT, B., *The treatment of data types in ELl,* Comm. ACM 17 (1974) 251-264.

[4] HOARE, C.A.R., *Proof of correctness of data representations,* Acta Informatica 1 (1972) 271-281.

[5] SCHUMANN, S.A., *Toward modular programming in high-level languages,* ALGOL Bulletin 37.4.1 (1974) 30-53.

[6] MORRIS, J.H., *Types are not sets,* SIGPLAN Symposium on Principles of Programming Languages (1973) 120-140.

[7] MEALY, G., *Another look at data,* Proc. AFIPS 31 (1967) 525-534.

[8] BALZER, R.M., *Dataless programming,* Proc. AFIPS 31 (1967) 557-566.

[9] ROSS, D.T., *Uniform referents: an essential property for a software engineering language,* in: *Software Engineering* 1, J.T. Tou (ed.), Academic Press, New York, 1970.

[10] MORRIS, J.H., *Protection in programming languages,* Comm. ACM 16 (1973) 15-21.

[11] WILKES, M.V., *The outer and inner syntax of a programming language,* Computer J. 11 (1968) 260-263.

[12] LISKOV, B. & S. ZILLES, *Programming with abstract data types,* Proc. Symp. on Very High Level Languages, SIGPLAN Notices 9 (April 1974) 50-59.

[13] GUTTAG, J.V., *Abstract data types and the development of data structures,* Comm. ACM 20 (1977) 396-404.

[14] HOARE, C.A.R., *Recursive data structures,* Stanford University Report CS-73-400 (1973).

[15] VON HENKE, F.W., *On generating programs from data types: an approach to automatic programming,* Proc. Conf. on Proving and Improving Programs, IRIA (1975) 57-70.

[16] WULF, W.A., R. LONDON & M. SHAW, *An introduction to the construction and verification of Alphard programs,* IEEE Transactions on Software Engineering 2 (1976) 253-264.

[17] SWIERSTRA, S.D., *Abstract data types, definition modules, extensions to definition modules,* TH Twente, Vakgroep Informatica (1977).

[18] KOSTER, C.H.A., *Visibility and types,* Proc. Conf. on Data: Abstraction, Definition and Structure, SIGPLAN Notices 11. Special issue (1976) 179-190.

[19] GRIES, D. & N. GEHANI, *Some ideas on data types in high-level languages,* Comm. ACM 20 (1977) 414-420.

[20] MEERTENS, L.G.L.T., *From abstract variable to concrete representation,* New Directions in Algorithmic Languages 1976, IRIA (1977) 107-133.

COMPLEXITEIT VAN VERZAMELINGENMANIPULATIE

P. VAN EMDE BOAS
Rijksuniversiteit Amsterdam

1. DE RELEVANTIE VAN VERZAMELINGENMANIPULATIE

Het verzamelingenbegrip waarop de gehele hedendaagse wiskunde gefundeerd is heeft binnen de programmeertalen nooit een overeenkomstige positie verworven. De set-constructie uit PASCAL waarmede de gebruiker via een omweg enkele bitmanipulatie-procedures in handen krijgt verdient nauwelijks de naam verzamelingenmanipulatie - in het bijzonder niet omdat het aantal elementen in het universum waarbinnen de verzamelingen leven in het algemeen begrensd wordt door de woordlengte van de computer waarop het PASCAL systeem geïmplementeerd is. Zo is het op de CYBER niet mogelijk set-of-char als type te gebruiken.

Als er sprake is van verzamelingenmanipulatie binnen een programma verloopt dit altijd via gegevensstructuren, die zelf weer zijn opgebouwd met behulp van arrays en/of wijzers.

Stellen we ons de vraag waarom verzamelingenmanipulatie belangrijk is, dan kunnen we deze met twee redenen beantwoorden. In de eerste plaats kunnen we opmerken dat verzamelingenmanipulatie een onderdeel vormt van de implementatie van vele hedendaagse efficiënte combinatorische algoritmen, waarbij de efficiëntie van deze manipulaties niet zelden de "bottle-neck" vormt die een nog efficiëntere oplossing verhindert [7]. In de tweede plaats kom je, al overwegende wat er in een typisch administratieve toepassing als het bijwerken van een gegevensbestand gebeurt, al snel tot de overtuiging dat wij hier te maken hebben met een verzameling gelijkvormige gegevens waarop verzamelingsoperaties zinvol zijn.

In ALGOL 68 terminologie kan men zich de records uit het klantenbestand van de firma Paschalis & Zn als volgt gedefinieerd denken:

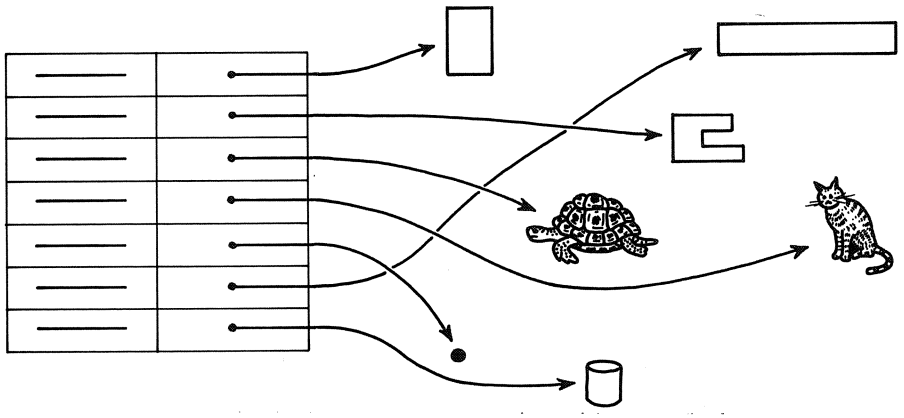
```
mode klant = struct (
             string naam, adres, stad,
             int leeftijd, bool geslacht, rhooms,
             real saldo, maximaalcrediet, achterstallige betalingen, boete,
                 rentepercentage
                 :
             );
```

Het blijkt dat de genoemde firma er beter aan doet om, naast de genoemde velden, een extra veld toe te voegen van het type *int* of *string*. De bedoeling van dit veld is dat de klanten op grond van de waarde van dit veld éénduidig geïdentificeerd kunnen worden. Men gebruikt voor dit veld de term *primary key*. De behoefte aan een dergelijke identificatiesleutel is nog sterker als informatie binnen verschillende bestanden met elkaar vergeleken dient te worden. Het conflict tussen enkele omroepverenigingen en de Dienst Luister- en Kijkgelden van de PTT is tot deze problematiek terug te voeren, evenals de discussie omtrent de wenselijkheid van het invoeren van een persoonsnummer.

De voordelen van het gebruik van een primary key binnen een gegevensbestand zijn vele. De sleutel heeft meestal een vaste omvang, zulks in tegenstelling tot de records zelve. De sleutels zijn lineair geordend, hetzij door de ordening der getallen, hetzij lexicografisch. Het wordt mogelijk een catalogus van het bestand te vormen, die er als volgt uit ziet:

```
# mode pk = int or string #
  mode referentie = struct (pk code, ref klant wijzer);
  mode catalogus = flex [1:0] referentie;
```

Als het bestand erg groot is kan de catalogus nog in het geheugen passen, terwijl het bestand zelve in het achtergrondgeheugen is opgeslagen. Het idee is uitgebeeld in fig. 0.



catalogus in kerngeheugen gegevens in achtergrondgeheugen

Fig. 0. Catalogus en gegevens

Mogelijke operaties die op een bestand dienen te worden uitgevoerd zijn:

geef het aantal klanten,
verwijder een klant,
zoek een klant,
voeg een nieuwe klant toe,
combineer twee bestanden,
zoek alle klanten met zekere eigenschappen.

Afgezien van de laatste operatie zijn al deze opdrachten via de catalogus op te vatten als opdrachten voor verzamelingenmanipulatie, waarbij het universum de verzameling van mogelijke sleutelwaarden is en de verzameling bestaat uit de in het bestand optredende sleutels. We komen derhalve tot de conclusie dat verzamelingenmanipulatie relevant is voor administratieve toepassingen.

2. DE SPELREGELS

We zullen ons bij onze beschouwingen beperken tot eindige verzamelingen. In het bijzonder is het universum U waarbinnen de gemanipuleerde verzamelingen leven, eindig. Het aantal elementen van U duiden we aan met u. Het universum is lineair geordend door de relatie $\leq$. De deelverzamelingen die

we in onze toepassing tegen kunnen komen hebben een grootte die begrensd wordt door een getal n. Daarnaast zijn we door de hoeveelheid beschikbaar geheugen niet in staat om meer dan N elementen te verwerken. Uiteraard geldt in het algemeen

$$n \leq N \leq u,$$

maar, in de toepassingen die we vandaag bespreken zullen deze drie grootheden gelijk zijn.

We beschouwen de in tabel 1 gegeven verzameling van instructies, ingedeeld naar het aantal betrokken verzamelingen en het feit of de ordening er al dan niet een rol bij speelt. De betekenis van de meeste instructies is duidelijk op grond van hun naam.

	Geen ordening	Ordening	
Een verz.	Member	Min	Max
	Insert	Extract min	Extract max
	Delete	All min	All max
	Cardinality	Predecessor	Successor
Meerdere verz.	Find	Split	
	Union	Concatenate	

tabel 1. Het instructierepertoire

Predecessor (i) berekent het grootste element in de verzameling dat $\leq$ i is. Analoog voor successor. Find (i) levert de verzameling op die het element i bevat. Split (A,i,B) verdeelt de verzameling A in twee stukken:

$$A := \{x \in A \mid x \leq i\}, \quad B := \{x \in A \mid x > i\}.$$

Concatenate (A,B,C) verenigt A en B in C mits $A \leq B$.

Als machinemodel voor het bepalen van de complexiteit van algoritmen gebruiken we de RAM [1] met uniforme tijdmaat. Het geheugen meten we in termen van RAM woorden. Bij deze laatste maat dienen we ons te onthouden

van getruceerde coderingen. Operaties als * en / en bitoperaties zijn verboden. Indien de aritmetische operaties optelling en aftrekking verboden zijn voor geheugenadressen spreken we van een adresmachine [6].

Een probleem uit de verzamelingenmanipulatie heeft nu de volgende vorm. Men geeft een deelcollectie van het repertoire uit tabel 1, gecombineerd met waarden voor $n \leq N \leq u$. Gevraagd een goede, bruikbare, efficiënte of optimale gegevensstructuur waarop dit repertoire kan worden uitgevoerd.
Op het eerste gezicht lijkt het of het manipuleren van verzamelingen reële getallen buiten ons model valt. Dit kan men echter opvangen door de aanname dat $n,N << u$.

Ik beperk mij in het vervolg van de voordracht tot een tweetal recente ontwikkelingen. Ik wil U allereerst kennis laten maken met de binomiale hoop die ontwikkeld is door VUILLEMIN [12] en BROWN [2]. Dit is op dit moment de meest efficiënte structuur voor het repertoire: cardinality, insert, delete, min, union. Ter kennismaking bespreken we eerst de klassieke binaire hoop.

In het tweede gedeelte van de voordracht wil ik het hebben over lopend onderzoek betreffende een $O(u)$-space implementatie van een eerder door mij ontwikkelde gegevensstructuur waarop het gehele repertoire van éénverzamelingsinstructies kan worden uitgevoerd in tijd $O(\log \log u)$ per verwerkt element.
Een uitgebreidere, Engelstalige voordracht over dit onderwerp is te vinden in [11].

Programmateksten in deze voordracht zijn afkomstig van schrijftafelwerk. Voor de correctheid ervan wordt derhalve geen garantie gegeven.

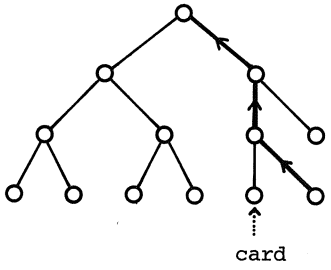
3. DE BINAIRE HOOP - EEN KLASSIEKE GEGEVENSSTRUCTUUR

De binaire hoop is de gegevensstructuur waarop de heapsort algoritme gebaseerd is. De structuur bestaat uit een binaire boom die handig is opgeborgen in een array $a[1{:}n]$. De zonen van een element $a[i]$ zijn de elementen $a[2*i]$ en $a[2*i+1]$. Binnen een hoop geldt dat het element opgeborgen bij de vader niet groter is dan de elementen opgeborgen bij de zonen. In het bijzonder is het kleinste element te vinden bij de wortel $a[1]$. Een globale teller *card* houdt bij hoeveel elementen thans in het array liggen opgeslagen.

Op grond van deze beschrijving is het direct duidelijk dat de

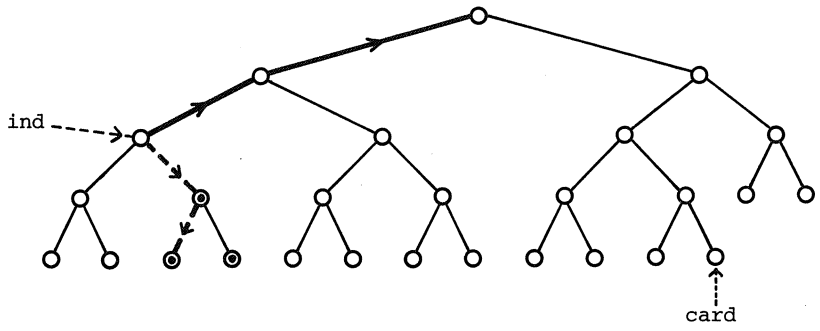
operaties min en cardinality in constante tijd kunnen worden uitgevoerd. De operaties insert en delete vragen tijd $O(\log n)$, waarbij we ervan uitgaan dat de positie van het te verwijderen element via een wijzer gegeven is. Aan deze conditie is automatisch voldaan in het geval van de extract min instructie. De onderstaande programma's geven een beschrijving van de insert en delete instructie.

```
proc insert = (elt x) void:
     (card +:= 1; int n := card;
     while int j = n÷2; (j=0|false|a[j] > x)
        do a[n] := a[j]; n := j od; a[n] := x);
```



Programma 1: Invoegen van element in hoop.

```
proc delete = (int ind) elt:
   if ind = card then card -:= 1; a[ind]
elif ind < card then elt x = a[ind], y = a[card]; card -:= 1;
   if x > y
      then int n := ind; #upwards search#
           while int j = n÷2; (j=0|false|a[j] > y)
              do a[n] := a[j]; n := j od;
               a[n] := y
      else int n := ind; #downwards search#
           while int j = index smallest son (n);
                 (j > card|false|a[j] < y)
              do a[n] := a[j]; n := j od;
               a[n] := y
      fi; x
fi;
```



Programma 2: Verwijdering element uit hoop.

We zien dat bij insert een element op de eerste vrije plaats in het array wordt ingevoegd, waarna het op weg naar de wortel met zijn vader stuivertje verwisselt zolang zijn vader groter is. Bij de delete-opdracht wordt het laatste element uit het array op de plaats van het te verwijderen element ingevoegd; hierna is, afhankelijk van de grootte van dit element, een opwaartse of neerwaartse reeks verwisselingen noodzakelijk. Bij de neerwaartse gang is ter bepaling van de kleinste zoon een extra vergelijking nodig. Een van de aantrekkelijke eigenschappen van het heapsort programma is dat het vrijwel geen extra geheugen vraagt. Een ongesorteerd array wordt binnen zichzelf tot hoop omgeordend door voor i, lopende van 2 tot n, het element $a[i]$ in te voegen. Hierna kunnen we $n-1$ keer het minimale element verwijderen en opslaan op de vrijgekomen plaats en na afloop is het array gesorteerd van groot naar klein.

De heap is in principe asymmetrisch. Men kan niet tegelijkertijd min en max efficiënt uitvoeren. Dit kan eventueel met een paar gekoppelde binaire hopen maar hierop wil ik thans niet nader ingaan.

Een ander gemis is het ontbreken van de union-instructie. Dit gemis is opgeheven in de binomiale hoop.

4. DE BINOMIALE HOOP

De binomiale hoop is geconcipieerd door VUILLEMIN [12]. Zijn idee bestaat er uit dat men een bos van hoop-gesorteerde bomen gebruikt. Deze bomen zijn van een speciaal type, bekend onder de naam *binomiale boom*.

Deze zijn inductief gedefinieerd als aangegeven in fig. 1. In fig. 2 geven we als voorbeeld hoe B_3 er uit ziet. Fig. 3 toont een alternatieve inductieve beschrijving van de bomen B_k.

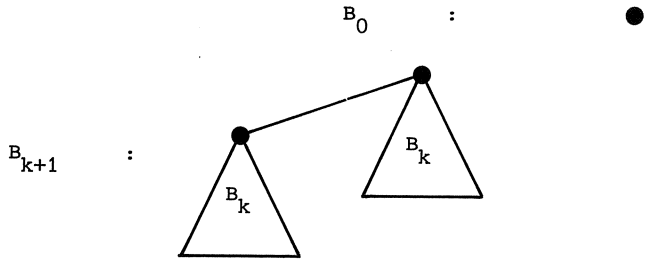


fig. 1. Definitie binomiale bomen

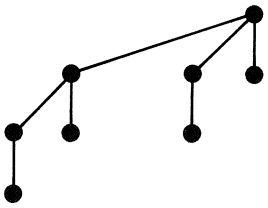


fig. 2. De boom B_3

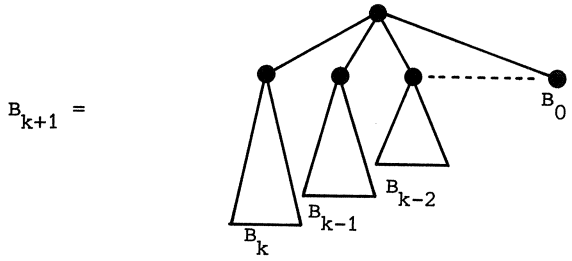


fig. 3. een alternatieve decompositie van binomiale bomen

Een binomiale boom B_k bevat 2^k elementen. Via de binaire schrijfwijze van een getal n kunnen we voor iedere natuurlijke n een bos van binomiale bomen van ongelijke omvang vinden dat precies n elementen bevat. Dit bos F_n bevat een B_k dan en slechts dan als het k-e cijfer in de binaire schrijfwijze van n een één is.

Leggen we de extra eis op dat iedere boom B_k de ordening van een hoop

heeft (element van de vader niet groter dan dat van de zonen) dan is de beschrijving van de binomiale hoop volledig.

Bij de implementatie zal men in het algemeen een extra knoop toevoegen, de "kop van de staart", die de verschillende optredende B_k's als zonen heeft. Binnen deze knoop kan men bovendien het aantal elementen en het kleinste element aflezen.

In fig. 4 vindt U een binomiale hoop van 11 elementen.

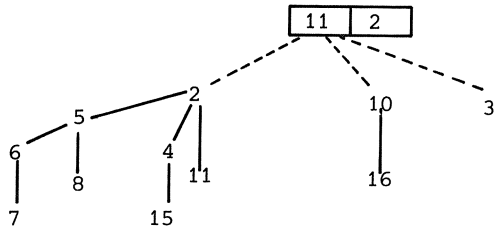


fig. 4. Voorbeeld van een F_{11}

Merk op dat het aantal kanten in F_n gelijk is aan n - $\nu(n)$, waarbij $\nu(n)$ het aantal enen in de binaire schrijfwijze van n is. Omdat de min direct afleesbaar is en de insert gezien kan worden als een speciaal geval van de union (vereniging F_n met een F_1) zijn union en delete de interessante opdrachten. Merk allereerst op dat het mogelijk is in constante tijd (onafhankelijk van k) twee B_k's samen te voegen tot een B_{k+1}, waarbij de B_k met de kleinste wortel bovenaan komt te staan. De union laat zich hiermede beschrijven als een soort binaire optelling, zoals is geïllustreerd in fig. 5.

Door het optreden van een carry-B_k kunnen soms drie B_k's tegelijk aanwezig zijn, waarvan in de volgende stap twee worden samengevoegd tot een B_{k+1}; de keuze van de achterblijver is voor de implementator. Analyse leert dat de tijd nodig voor een union evenredig is met het aantal te creëren nieuwe kanten. Hieruit volgt dat per union/insert instructie tijd $\mathcal{O}$(log n)

$F_{11} + F_7 \rightarrow F_{18}$:

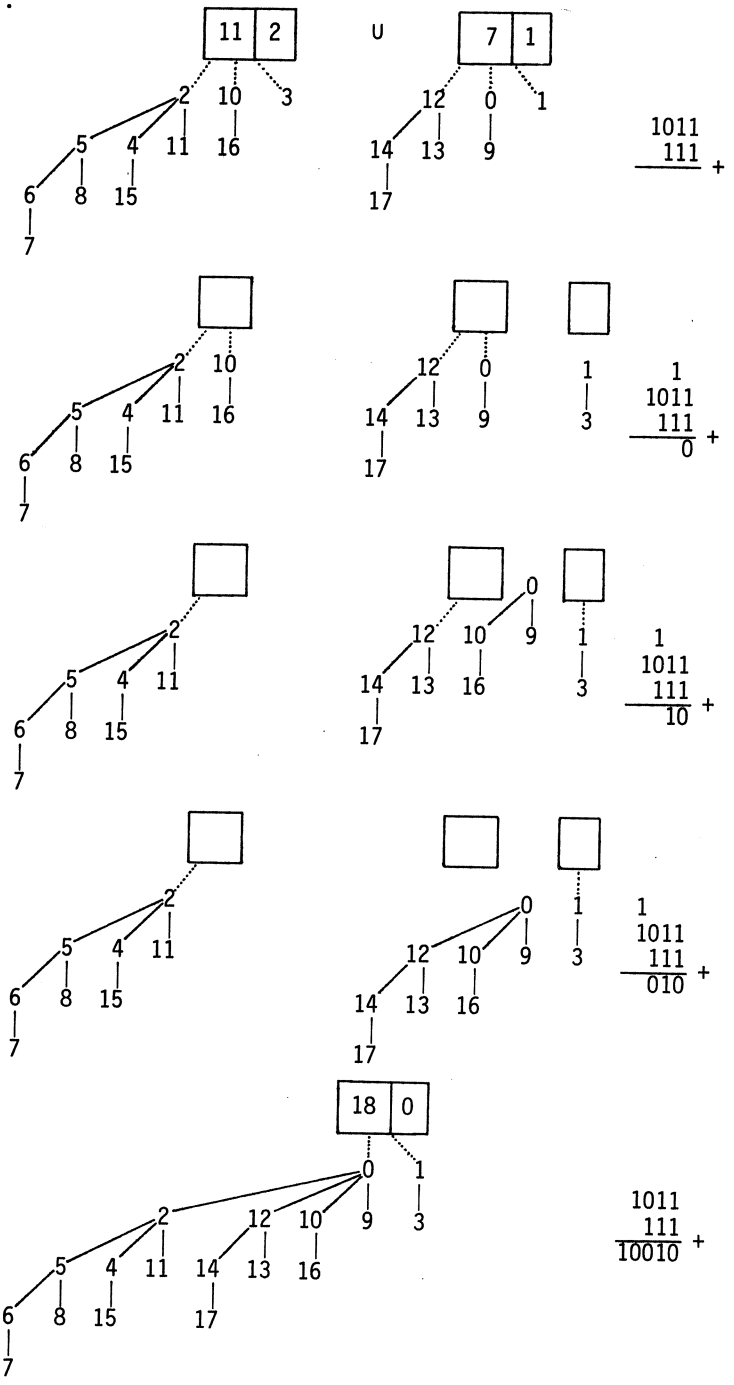


fig. 5. Vereniging van een F_{11} en een F_7

voldoende is. In het geval van m inserts in een hoop van n elementen blijkt de nodige tijd $O(m + \log n)$ te zijn. Dit op grond van de relatie:

$$\text{aantal nieuwe kanten} = ((n+m)-\nu(n+m)) - (n-\nu(n)) \leq m + \log n.$$

Hieruit volgt een gemiddelde inserttijd $O(1)$ in het geval dat veel toevoegingen plaatsvinden.

De delete-instructie veronderstelt opnieuw dat de plaats van het te verwijderen element via een wijzer gegeven is. In het geval van een extract min hoeven we slechts $O(\log n)$ wortels te onderzoeken om de kleinste te vinden. We gebruiken het feit dat verwijdering van een willekeurige knoop uit een B_k een F_{2^k-1} achterlaat. Voor de wortel van de boom is dit evident op grond van fig. 3; fig. 6 schetst een bewijs voor het algemene geval.

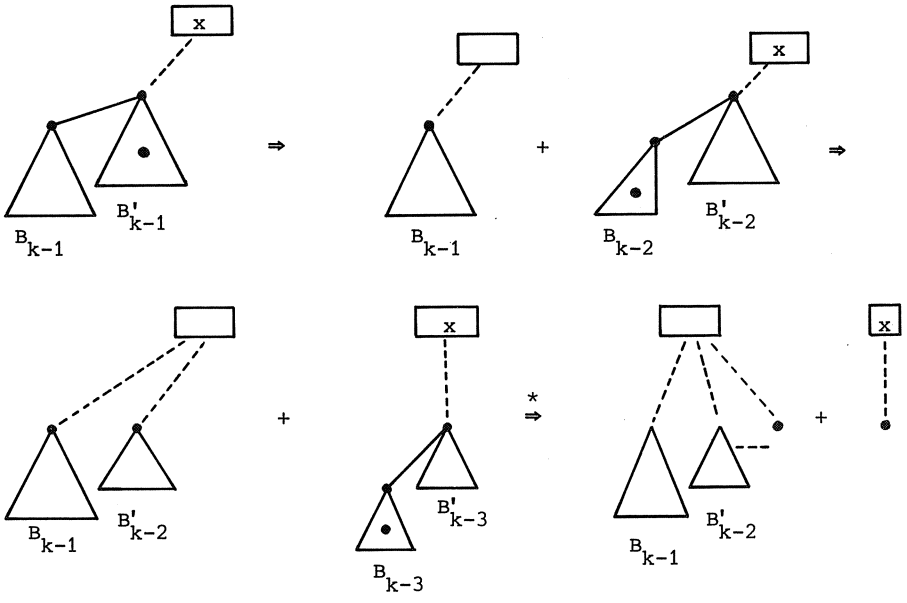


fig. 6. Verwijdering van een element uit een B_k laat een F_{2^k-1} achter

De delete-instructie bestaat nu uit het ontleden van de B_k die het te verwijderen element omvat tot een F_{2^k-1}. Hierna wordt deze F_{2^k-1} verenigd met het restant van de gegeven hoop. Beide acties vereisen tijd $O(\log n)$.

Merk op dat het onmogelijk is om een hoop-actie als in §3 uit te voeren op een B_k. Deze actie zou tijd $O(\log^2 n)$ vragen omdat het aantal zonen van een knoop tot log n kan toenemen.

MARK BROWN [2] heeft laten zien dat de binomiale hoop kan worden geïmplementeerd met niet meer dan twee wijzers per knoop. Het systeem is weergegeven in fig. 7.

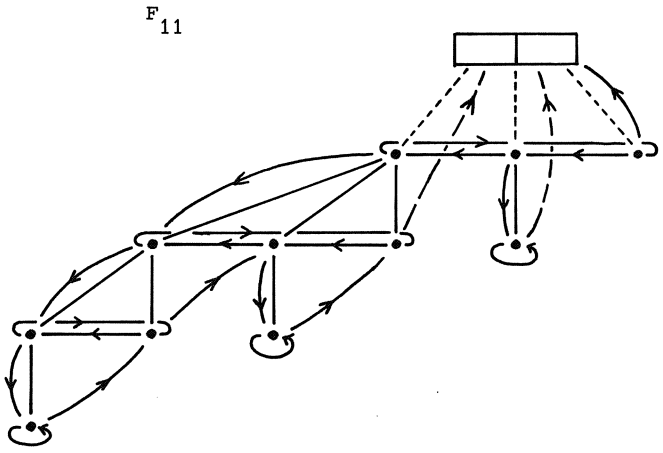


fig.7. Implementatie van een F_{11} met twee wijzers per knoop.

Daarnaast bevat het rapport van Mark Brown een uitgebreide analyse van alle in een uitdrukking voor de rekentijd optredende grootheden. Hij komt tot de conclusie dat van alle gegevensstructuren die het genoemde instructierepertoire implementeren de binomiale hoop verreweg de meest efficiënte is. (De asymptotische orde $O(\log n)$ wordt ook gehaald door 2 - 3 bomen (AHU [1]) en "leftist trees" (KNUTH [4]).)

5. EEN EFFICIËNTE PRIORITEITENWACHTRIJ

Onder een prioriteitenwachtrij versta ik in deze voordracht een gegevensstructuur waarop alle éénverzamelingsoperaties uit tabel 1 efficiënt kunnen worden uitgevoerd. Normaliter wordt de term gebruikt in het geval van het repertoire: insert, delete en min.

Bekende gegevensstructuren voor het volledige repertoire zijn de AVL-bomen (zie [4]) en de 2 - 3 bomen [1]. Al deze structuren vragen $O(n)$ geheugen en tijd $O(\log n)$.

We zijn geïnteresseerd in het geval dat $n \approx u$, d.w.z. het geval waarin de verzameling in principe kan uitgroeien tot het gehele universum.
Het idee waarop de gegevensstructuur waarover ik het verder wil hebben gebaseerd is, is in wezen eenvoudig. Het universum $\{0,\ldots,u-1\}$ wordt verdeeld in p melkwegstelsels elk van omvang q, waarbij $p*q = u$. Het melkwegstelsel G_t bestaat uit de elementen $\{t*q,\ldots,t*q + (q-1)\}$ voor $0 \leq t < p$.
De gegevensstructuur is opgebouwd uit structuren voor ieder van de melkwegstelsels en een structuur voor de cluster waarvan de melkwegstelsels lid zijn. Ieder melkwegstelsel bevat die verzamelingselementen die er toe behoren; de cluster bevat die melkwegstelsels die minstens één verzamelingselement bevatten. Deze hiërarchische indeling is schematisch aangegeven in fig. 8.

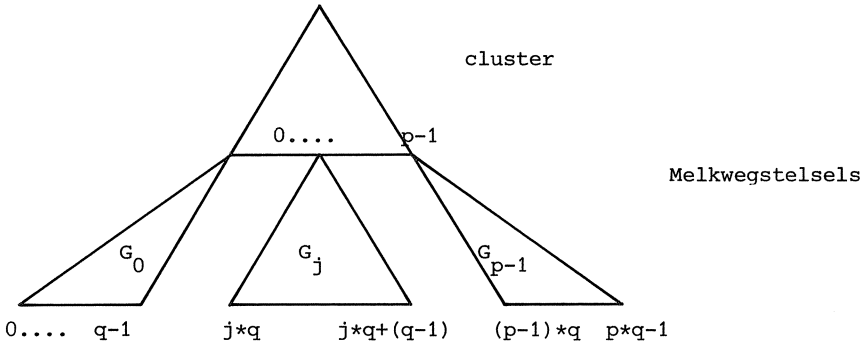


fig.8. Ontleding van het Universum

Iedere operatie op het universum blijkt uiteen te vallen in een eindig

aantal soortgelijke operaties op de melkwegen en de cluster. Hierbij blijkt het van belang om voor een gegeven element x de omvattende melkweg en de plaats van x binnen zijn melkweg snel te kunnen bepalen, evenals de omgekeerde bewerking. We gebruiken hiervoor de operatoren *head*, *tail* en *conc*. D.w.z., als t = *head* x en r = *tail* x dan is x element r in melkweg G_t en tevens geldt $x = t$ *conc* r.
Merk op dat moet gelden

head $x = x$ *over* q,
tail $x = x$ *mod* q en
t *conc* $u = t*q + u$.

Deze formules gebruiken echter verboden operaties als vermenigvuldiging en deling (of in het geval dat we alles binair schrijven en q een macht van 2 is - bitoperaties).

Het is verleidelijk om onze RAM met dit type instructies uit te breiden, maar de theorie leert ons dat hiermede een wezenlijke uitbreiding gegeven wordt aan de rekencapaciteit (HARTMANIS & SIMON [3]), zodat de relevantie van onze implementatie in twijfel kan worden getrokken.

Ik wil van enkele instructies aangeven hoe de decompositie er uit ziet:

```
proc insert un = (elt x) void:
    begin y := head x; t := tail x;
      if empty gal(y)
         then insert gal(y,t); insert cl(y)
         else insert gal(y,t)
      fi
    end;
proc min un = elt : (y := min cl; y conc min gal(y));
proc succ un = (elt x) elt : (y := head x; t := tail x;
       if t > max gal(y) then z := succ cl(y+1); z conc min gal(z)
          else y conc succ gal(y,t) fi);
```

Inspectie van de programma's leert dat iedere actie hoogstens één clusteractie en één melkweg-actie vergt. Dit levert ons in het geval dat p = q = $\sqrt{n}$ bij recursief gebruik van de decompositie een recurrente betrekking op van het type

$$T(n) \leq 2*T(\sqrt{n}) + C,$$

waaruit volgt dat $T(n) = O(\log n)$.

Hiermede hebben we de bestaande structuren niet weten te verbeteren. Kijken we echter opnieuw naar de programma's dan zien we dat in het geval dat twee operaties worden gebruikt, een van beide altijd een speciaal geval is (insertie van een eerste element; verwijdering van een laatste element, etc.). Indien we een uniek element in de structuur gewoon ergens apart opbergen en de rest van de structuur zolang ongebruikt laten kunnen deze speciale acties in constante tijd worden uitgevoerd. De problemen treden weer op bij de overgang 1 → 2 en terug, maar ook daarbij blijken de extra werkzaamheden slechts constante tijd te vergen.

De recurrente betrekking wordt nu:

$$T(n) \leq T(\sqrt{n}) + C,$$

waaruit volgt

$$T(n) = O(\log \log n)$$

en dit is een wezenlijke verbetering van de asymptotische groei der complexiteit. Rest nog slechts één probleem: hoe implementeren we het idee. De rest van de voordracht zal aan dit probleem gewijd zijn.

6. IMPLEMENTATIE VAN DE $O(\log \log n)$ PRIORITEITENWACHTRIJ

Een eerste implementatie van het boven geschetste idee is eind 1974 gegeven in [8]. Afgezien van de voor schrijftafelprogramma's gebruikelijke fouten vertoont de aldaar ontwikkelde implementatie twee zwakheden. Van de ordeningsinstructies uit tabel 1 worden alleen de min, extract min en all min geïmplementeerd, waarbij de oplossing voor het algemene geval als opgave aan de lezer werd gelaten. Verder ontbreekt iedere beschrijving van de implementatie van de nodige adresberekeningen; gesuggereerd wordt dit te doen met bitmanipulatie maar zoals eerder opgemerkt zijn dat verboden instructies.

Een betere implementatie, gebaseerd op een PASCAL versie die echt gewerkt heeft, werd gepresenteerd op het FOCS 16 congres (oktober 1975) en

is later verschenen in [10]. Gekozen was voor adresberekening via wijzers. Dit betekende dat de structuur de vorm aanneemt van een binaire boom, waarbij de bladeren de elementen voorstellen en de interne knopen corresponderen met deeluniversa uit de hiërarchische decompositie. De wortel correspondeert met het universum, de $\sqrt{u}$ knopen op het middenniveau zijn de melkwegstelsels van de eerste orde etc. Ieder deeluniversum valt in de boom uiteen in een linker- en rechterhelft en bevat dus dubbele informatie over wat er links en rechts gebeurt. Bovendien was het nodig om informatie over de positie van het deeluniversum in het grotere verband op te slaan in een extra veld. Dit betekende vijf velden per knoop. Daarnaast was voor iedere knoop een lijstje van log log n wijzers aanwezig om voor deze knoop de relevante subuniversa te kunnen localiseren. Een schets hiervan is gegeven in fig. 9.

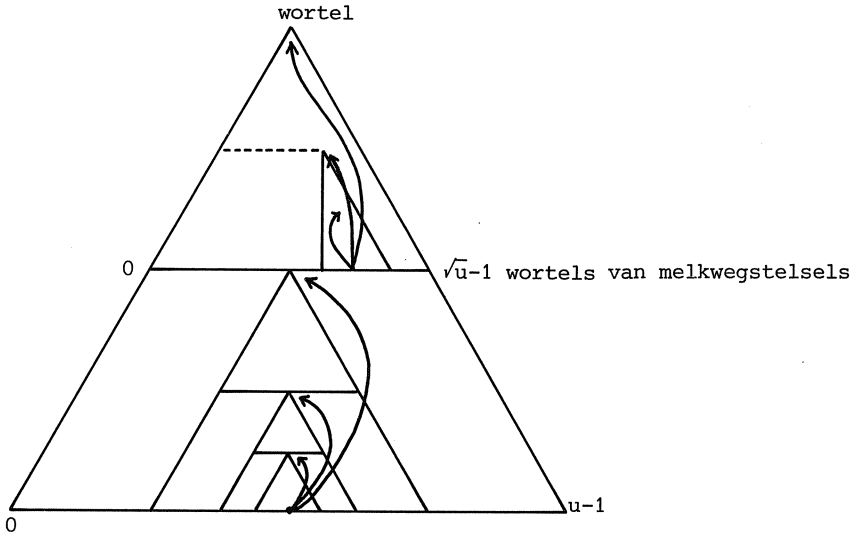


fig.9. Wijzers in de PASCAL implementatie

De consequentie van deze wijzers is dat de structuur meer dan lineair geheugen vergt; om exact te zijn O(u log log u) geheugenplaatsen. Het bleek mogelijk de structuur te initialiseren in tijd O(u log log u) zonder gebruik van verboden instructies.

Alle operaties werden weergegeven door recursieve procedures van een halve bladzijde; de gehele structuur is ongetwijfeld een van de meest gecompliceerde gegevensstructuren uit de complexiteitstheorie, en de beloning

voor het eerste correctheidsbewijs voor de programma's die ik op het FOCS congres heb uitgeloofd is tot op de huidige dag niet geclaimd, afgezien van een geslaagde poging van Knuth om het probleem op te lossen door het overbodig te maken, maar daarover later meer.

Het opmerkelijke is dat ondanks het meer dan lineaire geheugengebruik het mogelijk is om de structuur te gebruiken voor een O(u)-space, O(log log u)-time implementatie [9]. Hiertoe keren we terug tot het oorspronkelijke idee van de decompositie, maar in plaats van een recursieve toepassing combineren we twee verschillende implementaties. Voor de cluster gebruiken we de O(log log n)-time, O(n log log n)-space implementatie. Voor de melkwegstelsels kiezen we een O(m)-space, O(m)-time implementatie, bv. een ongesorteerde lijst. Kiezen we nu m = log log n en u = n·m = n log log n, dan zien we dat we ons doel bereikt hebben; zie fig. 10.

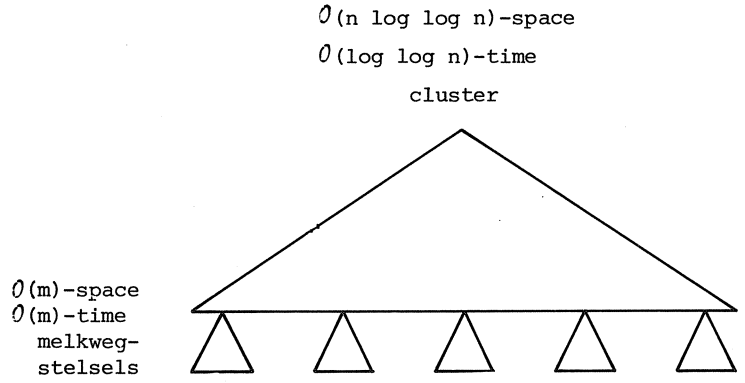


fig.10. O(u)-space implementatie

Immers we hebben voor tijd en geheugen de schattingen:

Tijd: $O(\log\log n) + O(m) = O(\log\log n) = O(\log\log u)$

Geheugen: $O(n \log\log n) + n \cdot O(m) = O(n \log\log n) = O(u)$.

7. RECURSIEVE IMPLEMENTATIE VAN DE PRIORITEITENWACHTRIJ

Maart 1977 ontving ik van Knuth een exemplaar van diens classroom notes, waarin hij mijn structuur behandelde [5]. In deze brief schetste hij een SIMULA implementatie die gebruik maakte van een recursieve gegevens-

structuur. Dit ligt, mede gezien de recursieve aanpak van de operaties, voor de hand, en het valt te verwachten dat het correctheidsbewijs op deze manier aanmerkelijk eenvoudiger zal worden.

De structuur ziet er in ALGOL 68 uitgedrukt als volgt uit:

```
mode deque = struct(int card,min,max,
                    ref deque top,
                    ref [ ] deque bottoms);
```

Merk op dat van de tweezijdigheid in de hiervoor besproken structuur niets over is. De splitsing in linker- en rechterhelft die gesuggereerd wordt door de plaatjes is kennelijk geheel overbodig.

Op grond van deze vorm komt men tot een geheugengebruik dat voldoet aan de volgende recurrente betrekking

$$S(u) \leq C + (\sqrt{u} + 1) \cdot S(\sqrt{u}),$$

waaruit men kan afleiden dat $S(u) = O(u)$, (volgens KNUTH [5] zelfs $O(u)$ bits!).

Voor het gebruik van het array *bottoms* dat de melkwegen voorstelt zijn adresberekeningen nodig. Knuth gebruikt hiervoor verboden instructies, zij het in de context waar hij denkt aan een concrete implementatie op een echte computer. We zullen zo zien hoe we deze instructies kunnen vermijden.

Ik wil U kennis laten maken met de vorm van de insert-instructie in een recursieve implementatie. We gebruiken voor de triviale inserties een procedure *first insert*, gedefinieerd door:

```
proc first insert = (ref deque queue, int arg) void:
     (card of queue := 1; min of queue := max of queue := arg);
```

De insert-procedure zelf krijgt een extra parameter *order* die de nog resterende recursiediepte van de structuur aangeeft. Voor *order = 0* is de bodem van de recursie bereikt en worden alle operaties geacht triviaal te zijn.

```
proc insert = (ref deque queue, int arg, order) void:
  if order = 0 then # bottom case of recursion;
                    trivial actions for small universe #
               .....
```

```
else case card of queue +1 in
  (first insert (queue,arg); goto done),
  (# set up internal structure for existing elt #
     int old = min of queue;
     int old tail = order tail old, old gal = order head old;
  ref deque cl = top of queue, bot = (bottoms of queue) [old gal];
     first insert (cl,old gal); first insert (bot, old tail) )
  esac; # now we are ready for new insert #
  int new tail = order tail arg, new gal = order head arg;
ref deque cl = top of queue, bot = (bottoms of queue) [new gal];
  if card of bot /= 0
      then insert (bot, newtail, order-1)
      else first insert (bot, newtail);
           insert(top, new gal, order-1)
  fi; # now adjust min/max and card #
    if arg < min of queue then min of queue := arg
  elif arg > max of queue then max of queue := arg fi;
    card of queue +:= 1; done: skip fi;
```

Programma 4: Insertie in een recursieve implementatie

In *insert* speelt de verboden instructie *conc* nog geen rol, maar die komt bij *delete* en *successor* om de hoek kijken. Merk verder op dat de instructies *head*, *tail* en *conc* de orde van de aanroep als extra parameter nodig hebben.

De oplossing om van de verboden operaties af te komen is als vanouds het gebruik van functietabellen. Weliswaar hebben we log log u verschillende tabellen nodig, maar de lengtes hiervan zijn respectievelijk $u, \sqrt{u}, \sqrt{\sqrt{u}}, \ldots, 4$ en de som van deze lengtes is $O(u)$. Men krijgt dus operaties zoals

```
op tail = (int order, arg) : tails[order][arg],
proc conc = (int order, head, tail) : tail + concs[order][head] .
```

Resteert de vraag hoe deze tabellen geïnitialiseerd dienen te worden. Dit blijkt via eenvoudig telwerk in tijd $O(u)$ te kunnen; zie programma 5. Eenzelfde hoeveelheid tijd is nodig om de recursieve structuur te initialiseren.

```
int l := 2;
  for k to max order do
    int s := -1;
      for i from 0 tot l - 1 do
       concs[k][i] := s + 1;
        for j from 0 to l - 1 do
          heads[k][s +:= 1] := i; tails[k][s] := j
       od od;
     l := s + 1
   od;
```

Programma 5: Berekening adrestabellen

Merk op dat een rijtje twee dimensionale arrays t.b.v. *conc* niet alleen onnodig is maar zelfs verboden - adressering in een tweedimensionaal array vereist de vermenigvuldiging die we juist trachten te elimineren.

Er blijven nog twee vragen ter beantwoording over. Allereerst: hoe ziet de recursieve structuur er uit als de recursie wordt uitgewerkt? Is hiervoor een nette regelrechte beschrijving denkbaar? Daarnaast kunnen we opmerken dat de in §6 beschreven implementaties op basis van wijzers legale implementaties op een adresmachine zijn, mits ook de argumenten via wijzers worden doorgegeven. De implementatie uit §7 is dat niet, getuige de aritmetiek gebruikt voor het uitrekenen van array-indices.

Bestaat er een nette $O(u)$-space implementatie op een adresmachine?

LITERATUUR

[1] AHO, A.V., J.E. HOPCROFT & J.D. ULLMAN, *The design and analysis of computer algorithms*, Addison Wesley 1974.

[2] BROWN, M.R., *Implementation and analysis of binomial queue algorithms*, SIAM J. on Computing (to appear).

[3] HARTMANIS, J. & J. SIMON, *On the structure of feasible computations*, *Advances in Computers*, Vol. 14, M. Rubinoff & N.C. Yovits (eds), Academic Press, New York, 1976.

[4] KNUTH, D.E., *The art of computer programming*, Vol. 3, *Sorting and searching*, Addison Wesley 1973.

[5] KNUTH, D.E., *Notes on the Van Emde Boas construction of priority deques; An instructive use of recursion,* Classroom Notes Stanford, March 1977.

[6] TARJAN, R.E., *Reference machines require non-linear time to maintain disjoint sets,* ACM STOC 9 Symposium, May 2-4, 1977, p. 18-29.

[7] TARJAN, R.E., *Complexity of combinatorial algorithms,* Report Stanford CS 609, 1977.

[8] VAN EMDE BOAS, P., *An O(n log log n) on-line algorithm for the insert-extract min problem,* Report TR 74-221 Cornell Dept. CS, December 1974.

[9] VAN EMDE BOAS, P., *Preserving order in a forest in less than logarithmic time and linear space,* Information Processing Letters 6, p.80-82, June 1977.

[10] VAN EMDE BOAS, P., R. KAAS & E. ZIJLSTRA, *Design and implementation of an efficient priority queue,* Math. Systems Theory 10, 99-127, February 1977.

[11] VAN EMDE BOAS, P., *Developments in data structures,* in J.K. Lenstra e.a. (eds), Interfaces between Computer Science and Operations Research, Math. Centre Tracts 99, pp. 33-61 (1978).

[12] VUILLEMIN, J., *A data structure for manipulating priority queues,* CACM 21 (1978), pp. 309-314.

PATTERN MATCHING IN SPRING

P. KLINT
Mathematisch Centrum

1. INTRODUCTION

In the early days, computers were mainly used for numeric computing. That is the reason why those first programmable data manipulators were called 'computers'. Nowadays, the vast majority of all applications does hardly involve any calculating at all. But one relict of the past remains: until someone invents a better term we have to call those applications 'non-numeric'.

Pattern matching or pattern directed scanning belongs to the subarea of non-numeric computer applications that deals with certain recognition processes. In general a structure description ('pattern') is given to a scanning or recognition device to determine whether a given element from a given universe satisfies that structure description or not. In the former case the pattern match is said to succeed, in the latter case it is said to fail. The universe may consist of linear character strings, data structures, line drawings, photographs and the like. It seems to be agreed upon that one speaks of picture recognition or pattern recognition if the elements in the universe are more-dimensional, and to reserve the names pattern matching and pattern directed scanning for universes with linear elements. In the sequel I will concentrate on the universe that consists of linear character strings.

Several programming languages and programming systems contain features related with pattern matching in the sense that a structure description of legal or expected strings is given. To gain insight in those features and to identify certain problems, I will now briefly comment on some of these systems.

Lexical scanner- or parser-generators transform a grammar (the structure description) for a certain language into a program that recognizes that language. In most cases the input grammars are restricted to certain types (regular expressions, LR(1)) to allow the use of more efficient parsing algorithms. The generation of the recognition program tends to be time or space consuming, which makes it impractical to regenerate the recognition program after changes have been made to the original input grammar (a desirable property for the parsing of extensible languages).

Macro processors are fed with a set of definitions of macros with a certain delimiter structure (the structure description). Next the input stream is literally copied to an output stream, with the exception that 'macro calls' are recognized and replaced as indicated by their definition. Most (all?) macro processors use a horrible notation for the description of delimiter structure or macro definition. Such a notation is probably induced by the fact that those macro processors use character-by-character interpretation of macro definitions.

Most string manipulation languages, such as COMIT[1], AMBIT[2], SNOBOL4[3] and SL5[4] contain pattern matching features in some form. In all older languages, pattern matching is based on the COMIT replacement rule: a part of the 'work space' or 'subject string' is matched and replaced. As a consequence, these languages tend to have very powerful pattern matching facilities, but are rather weak in string synthesis. This discrepancy is treated in detail in [5]. SL5 uses recursive coroutines to model a pattern matching algorithm. Various models for pattern matching are discussed in section 3..

Some of the above approaches have been very successful in a specific problem area, but lack generality, provide an awkward notation, are too complex (obtrusive heuristics of the SNOBOL4 scanner) or are general but inefficient.

The programming language SPRING, which is currently under development at the Mathematical Centre, is being designed with the following purposes:

- The language will be used to write a production quality document processing system.

- The input specifications for the document processing system may be very complex since mathematical notation, tabular material, block diagrams must be handled. Some of these applications require the definition and recognition of sublanguages for input specifications, e.g.,

  min from i=0 to 10 f sub i

  may be used to describe $\min_{i=0}^{10} f_i$.

  As a consequence, SPRING must allow a concise description of such languages.

- Powerful output operations are needed to produce complex two-dimensional layouts. The SPRING string manipulation primitives are based on the concept of three-dimensional 'block's [6], which can be concatenated and sliced in all directions.

Given this anticipated use of the SPRING programming language, some design goals can be formulated. Some of these are an attempt to learn from past experience. At this moment it is not yet clear which new pitfalls have been introduced.

- General purpose pattern matching and string manipulation facilities.

- Pattern matching and string synthesis primitives of 'equal' complexity.

- Uniform treatment of patterns and procedures. A pattern can be considered as a procedure executed on a pattern matching machine. Patterns can have local variables and arguments.

In the following sections attention is focussed on the pattern matching aspects of SPRING and especially on the data structures which have been considered for the implementation of patterns. Section 2 contains an introduction to pattern matching and some examples are given. In section 3 several methods for the formal characterization of patterns are compared. One specific characterization is examined more closely and implementation problems are discussed.

2. AN INTRODUCTION TO PATTERN MATCHING

For those who are not familiar with pattern matching, this section contains an overview of pattern matching primitives and some examples. To keep the experts awake, I will use the SPRING notation which may be less familiar to them. Some of the primitives discussed are exclusive to the SPRING programming language.

A pattern match is initiated by the match operator (?), which has a string (or text file) as left operand ('the subject') and a pattern as right operand. The simplest form of a pattern is a string. For example,

```
'abc' ? 'ab'
```

determines whether the subject contains the string 'ab'. The match operator can fail or succeed and delivers a value. In case of failure, the subject is delivered as value. In the case of success the assembled scan value is delivered as result. In a few moments we will see how this value is constructed.

More complex patterns are build by means of subsequentiation (--) and alternation (|) In the pattern $P_1 \mid P_2$, P_1 is applied first. If P_1 fails, then P_2 is applied and the success or failure of $P_1 \mid P_2$ as a whole is determined by the success or failure of P_2. If P_1 succeeds, $P_1 \mid P_2$ as a whole succeeds, but the alternative P_2 is remembered. If in a later stage of the same pattern match, failure occurs, previous untried alternatives are attempted and in this way P_2 may be tried.

In the case of sequentiation (or pattern concatenation) P_1 -- P_2, P_1 is applied first. If P_1 fails then P_1 -- P_2 fails. If P_1 succeeds, then P_2 is applied next. Success of P_2 implies success of P_1 -- P_2 as a whole. On failure of P_2, possible untried alternatives of P_1 are attempted and P_2 is applied again, until a successful application of P_2 (P_1 -- P_2 succeeds) or there are no more untried alternatives of P_1 (P_1 -- P_2 fails).

An example is:

```
('d' | 'l') -- 'ea' -- ('n' | 'r' | 'd')
```

which will match 'dean', 'dear', 'dead', 'lean', 'lear' and 'lead', in this order. *

It was already mentioned that a successful pattern match delivers an assembled scan value as result. This value is the concatenation of contributions made during the pattern match. Two types of contributions exist. Internal contributions (=P, where P is a pattern valued expression) contribute a value which is equal to the part of the subject that is matched by P, i.e.,

```
'b' ? =('a' | 'b' | 'c')
```

contributes 'b'. External contributions (/S, where S is a string valued expression) just contribute the value S, i.e.,

```
'b' ? ('a' | 'b' | 'c')--/'xyz'
```

---

* Monadic operators have precedence over dyadic operators. Dyadic operators are (in order of decreasing priority):

```
--
|
=: !=: /: !/:
?
:=
```

contributes 'xyz' and

```
'b' ? =('a' | 'b' | 'c') -- /'xyz'
```

contributes 'bxyz'.

Closely related with the above contribution operators are the subject assignment and contribution assignment operators. The subject assignment (P=:V, P a pattern valued expression, V a variable) assigns the part of the subject that is matched by P to V, i.e.,

```
'b' ? (('a' | 'b' | 'c') =: x)
```

assigns 'b' to x. The contribution assignment (P/:V, P a pattern, V a variable) assigns the contributions made by P to V and does not add these contributions to the overall scan value, i.e.,

```
'b' ? (=('a' | 'b' | 'c') -- /'xyz') /: x
```

assigns 'bxyz' to x, but does not deliver a scan value.

A rather delicate problem arises, if one considers the precise moment at which such assignments are effectuated: immediately after the successful application of the left hand operand of the assignment operator (independent of the success or failure of the overall pattern match), or when the whole pattern match succeeds. The current version of SPRING provides two forms of all operators, for which this problem exists. For example, assignment operators can be conditional (=: and /:) or immediate (!=: and !/:). This distinction between immediate and conditional evaluation is not satisfactory, since in many cases the evaluation order is not reflected properly by the program text. This can be seen in

```
'abc' ? ('a'=:x) -- ('b'!=:x) -- ('c'=:x)
```

where the values 'b', 'a' and 'c' are assigned to x in this order. Other models, such as immediate evaluation whose effects are undone on failure of the pattern match, may lead to tremendous implementation

problems (how does one reverse interactive terminal i/o?) or logical inconsistencies. This problem is not solved correctly by any programming language I am aware of.

To complete this overview, four points are now discussed: unevaluated expressions, actions, built-in functions and patterns with arguments and local variables.

Unevaluated expressions (*P) provide a means to delay the evaluation of a pattern, until it is actually needed during a pattern match. This is useful for the definition of recursive patterns and for the definition of patterns which are context dependent. For example:

```
p := ('b' -- *p | 'a')
```

defines a pattern that matches 'a', 'ba', 'bba', ...

Conditional and immediate actions (@E and !@E, E arbitrary expression) are pieces of program which are evaluated but their value is discarded, i.e.

```
'ab' ? 'a' -- @print('an action') -- 'b'
```

will print a message.

Several built-in functions are available to solve frequently occurring problems:

span(S) matches a span of characters in S.

any(S) matches a single character in S.

len(n) matches a string of n characters.

break(S) matches a string of characters not in S, followed by a character in S.

arb() matches a string of zero or more characters, in the sense that it matches first zero characters, then one character and so on.

rpt(p) applies pattern p as often as possible and succeeds on the first failure of p.

pos(n) succeeds if the pattern match has proceeded to position n of the subject.

rpos(n) as above, but n is counted from the right of the subject.

tab(n) continues the pattern match at position n of the subject.

rtab(n) as above, but n is counted from the right end of the subject.

reverse(S) reverses the string S.

replace(S_1, S_2, S_3) replaces all occurrences in S_1 of characters in S_2 by the corresponding character in S_3.

Two examples may illustrate the use of the primitives mentioned so far. A simple grammar for the recognition of identifiers is: *

```
(1)  empty  := '';
(2)  letter := 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ';
(3)  digit  := '0123456789';
(4)  letgit := letter || digit;     # || denotes string concatenation #
(5)  id     := any(letter) -- (span(letgit) | empty);
```

---

* In the following examples, line numbers are added to simplify the description. These are, however, not part of the actual programs.

A simple minded procedure for the conversion of integers to roman numerals is (see [7]):

```
(1) proc roman(n){
(2)   var t;
(3)   if succeed n := n ? =rtab(1) -- len(1)=: t then
(4)   return(
(5)     replace(roman(n), 'IVXLCDM', 'XLCDM**')||
(6)     ('0,1I,2II,3III,4IV,5V,6VI,7VII,8VIII,9IX,' ? t--=break(','))
(7)   ) fi
(8) };
```

In line (3) the number n is split in a left part which is assigned to n, and a rightmost digit which is assigned to t. In line (5) the left part n is converted to roman and next (roman) multiplied by ten. In line (6) the rightmost digit is converted to roman and appended to the result.

To make procedures and patterns as similar as possible, patterns may have arguments and local variables. This is illustrated in the following pattern that recognizes palindromes:

```
pat{var head; arb()!=:head--(len(1) | '')--*reverse(head)--rpos(0)}
```

A final example illustrates the power of integrated pattern matching and more dimensional string concatenation. The following grammar recognizes parenthesized list expressions and converts these expressions into a two-dimensional representation of the list:

```
(1)   blank  := span(' ') | '';
(2)   list   := pat{var result, atom;
(3)         ( span(letter) =: result |
(4)          '(' -- rpt(*list/:atom -- @(result := result| < atom)) --
(5)             @(result := it('-') == result) -- ')'
(6)         ) -- blank -- /(' ' || ('|' == result) || ' ')
(7)         };
```

Figure 2.1. gives an example of the result of application of this pattern. In line (3) of the above program, an atomic list is recognized and the associated name is assigned to 'result'. In line (4), a parenthesized list with sublists is recognized. In each repetition the next sublist is concatenated to the result with all top sides aligned (this is done by |<). In line (5), a row of dashes (equal to the width of the result, this is done by it('-')) is placed above the result. Finally, in line (6) a bar is placed above the result (this is done by ==) and a blank is appended to the left and the right of the result, before this value is contributed.

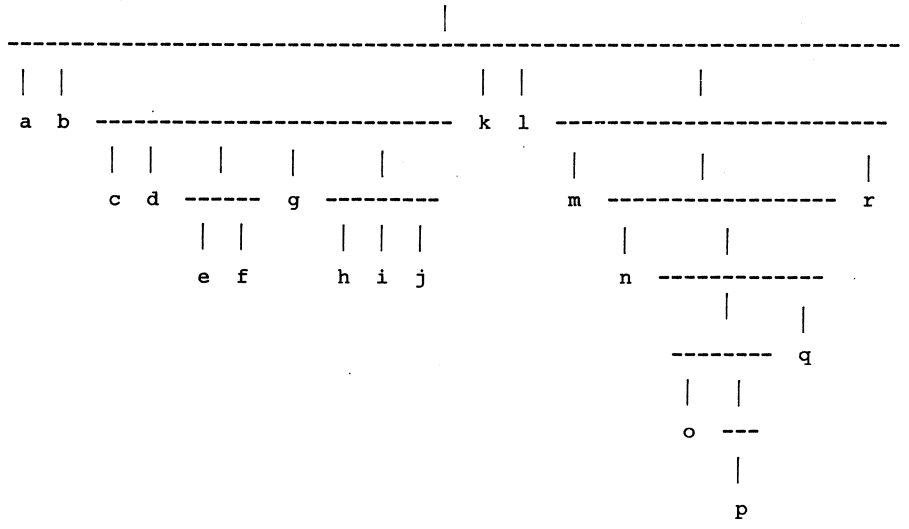


Fig. 2.1. The result of applying the pattern list to
'(a b (c d (e f) g (h i j)) k l (m (n ((o (p)) q)) r))'.

3. MODELS FOR PATTERN MATCHING

At least three models have been used to describe patterns:

SET: patterns are sets of strings

COR: patterns are collections of recursive coroutines

ALG: patterns are algebraic transformations on (subject, cursor) values.

Model SET is the oldest and was used during the design of SNOBOL3[8]. It is possible to associate a set of strings with every pattern constructed from simple strings by sequentiation and alternation. One may argue that the non-commutativity of alternation invalidates the model. More severe problems arise, however, with a number of built-in functions. It is extremely difficult, if not impossible, to associate a unique set of strings with patterns which contain primitives such as POS(n), TAB(n), BREAK(s), since the strings that are matched by such primitives depend on the context in which these strings occur.

Model COR is the most recent one and is used in the SL5 programming language. Several languages (SNOBOL4 [9], TMG [10]) refer to some sort of 'backtrack process' that is used to implement a pattern matching algorithm. SL5 has formalized this notion and provides language primitives to implement and guide a backtrack process, which is completely under control of the programmer. This approach has many advantages and the primitives can be used in many situations where a guided search is involved. It seems, however, that the generality of the SL5 model leads to a less efficient implementation of the pattern matching ('string scanning' in SL5 terminology) algorithm. I realize that the efficiency argument is often abused and that the advantage of a smaller programming effort and simplicity often outweighs the disadvantage of inefficiency. But given the goals and scope of the SPRING project, it seemed wise to choose a somewhat less general model that would more certainly lead to an efficient implementation.

Model ALG is used as basis for the SPRING pattern matching algorithm and will now be discussed in more detail.

3.1. An algebraic model of pattern matching

In this model (based on [11] and [12]), a pattern P is a function P(S, c), where S is a string, called the subject string and c is an integer indexing S, called the pre-cursor position. The value of P(S,c) is a sequence of integers, indexing S, the elements of which are called the post-cursor positions. Cursor positions can take on the values 0, 1, ..., $|S|$, where $|S|$ denotes the length of the subject S. Examples:

P = 'ab' | 'aab' | 'a'; S = 'aab'
  P(S, 0) = {3, 1}
  P(S, 1) = {3, 2}
  P(S, 2) = {} ({} denotes the empty sequence)
  P(S, 3) = {}

LEN(n)(S, c) = if c + n ≤ |S| then {c + n} else {} fi*
POS(n)(S, c) = if c = n then {c} else {} fi
RPOS(n)(S, c) = if n = |S| - c then {c} else {} fi
TAB(n)(S, c) = if n ≥ c ∧ n ≤ |S| then {n} else {} fi
RTAB(n)(S, c) = if |S| - n ≥ c ∧ n ≤ |S| then {|S| - n}
  else {} fi

In the sequel we will need an extended definition of patterns, which operates on sequences of cursor positions:

$$P(S, \{c_1, c_2, \ldots, c_n\}) \equiv$$

$$P(S, c_1) \cup P(S, c_2) \cup \ldots \cup P(s, c_n)$$

---

* Built-in functions with names in lower case (i.e. len) are modelled by functions in upper case (i.e. LEN).

where ∪ denotes concatenation of integer sequences.

With the aid of this machinery we can now define sequentiation and alternation:

$$(P \text{ -- } Q)(S, c) \equiv Q(S, P(S, c))$$
$$(P \quad Q)\ (S, c) \equiv P(S, c) \quad Q(S, c)$$

A number of properties based on the above definitions can be derived:

P1. Sequentiation is associative:
$(P_1 \text{ -- } P_2) \text{ -- } P_3 = P_1 \text{ -- } (P_2 \text{ -- } P_3)$

P2. Alternation is associative:
$(P_1 \mid P_2) \mid P_3 = P_1 \mid (P_2 \mid P_3)$

P3. Sequentiation distributes over alternation from the right:
$(P_1 \mid P_2) \text{ -- } P = (P_1 \text{ -- } P) \mid (P_2 \text{ -- } P)$

Let a monic pattern be defined as a pattern which has for every subject and pre-cursor position at most one post-cursor position. If we assume that a set of built-in monic patterns (primitives) exists, the following holds:

P4. If p is a primitive and P_1 and P_2 are arbitrary patterns:
$p \text{ -- } (P_1 \mid P_2) = p \text{ -- } P_1 \mid p \text{ -- } P_2$

P5. $P_1 \text{ -- } P_2$ is monic if P_1 and P_2 are monic.

P6. Any pattern formed by sequentiation and alternation of primitives can be written in the canonical form:
$p_{11} \text{ -- } \cdots \text{ -- } p_{1n_1} \mid \cdots \mid p_{m1} \text{ -- } \cdots \text{ -- } p_{mn_m}$

where p_{ij} is primitive, $m \geq 1$ and each $n_i \geq 1$ and all indices are finite.

3.2. From model to implementation

After this abstract definition of patterns we come down to the ground again and try to figure out which physical representations are best suited for the representation of patterns. We will consider in turn: trees, directed graphs with subsequent and alternate linkage, and directed graphs with subsequent linkage only.

The most obvious representation of a pattern like $(P_1 \text{ -- } P_2 \mid P_3) \text{ -- } P_4$ is a tree such as in figure 3.1..

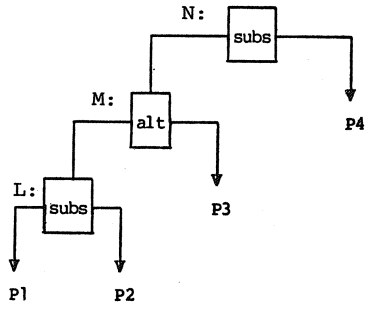


Fig. 3.1. Tree representation of $(P_1 \text{ -- } P_2 \mid P_3) \text{ -- } P_4$.

This representation has the disadvantage that during pattern matching a considerable amount of tree walking must be done. If $P_1 \text{ -- } P_2$ succeeds in the above example, the next primitive to be tried (P_4) is only found after the walk (L, M, N).

The next idea is to add threads to the tree which determine the match order of the nodes. But such a threading alone turns out to be sufficient to guide the pattern match and hence the tree structure can be forgotten altogether. The result is a path diagram such as in figure 3.2..

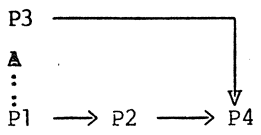


Fig. 3.2. Path diagram for $(P_1 \text{ -- } P_2 \mid P_3) \text{ -- } P_4$.

Subsequents are indicated by solid arrows, alternates by dotted arrows. Each node has an associated primitive which is a monic pattern. An s-vacancy is a node without subsequent, an a-vacancy is a node without alternate.

The subsequentiation of two path diagrams D_1 and D_2 is found by drawing a solid arrow from every s-vacancy in D_1 to the root of D_2. The alternation of two path diagrams D_1 and D_2 is found by starting in the root of D_1, following the chain of alternates and connecting the first a-vacancy with the root of D_2 by means of a dotted arrow.

In the next section we will see that path diagrams form a useful tool for the design of patterns and can also be used during the minimization of the time or space complexity of pattern components. But how can we be sure that such an optimized path diagram has the same formal properties as the original unoptimized path diagram? To settle this question we define a reverse mapping from a path diagram back to a pattern. Such a <u>derived pattern</u> D(n) of node n is inductively defined as:

D(n) = p(n) -- D(s) | D(a)	if both a and s exist
= p(n) -- D(s)	if only s exists
= p(n) | D(a)	if only a exists
= p(n)	if neither a or s exist

where s and a denote the subsequent and alternate of node n, and p(n) denotes the primitive associated with n.

3.3. An example of pattern design

The primitive ARB can be recursively defined as:

```
ARB = NULL | LEN(1) -- ARB
```

which corresponds to the path diagram in figure 3.3.. Note that the primitive associated with null nodes succeeds immediately. Null nodes are used to make connections in path diagrams.

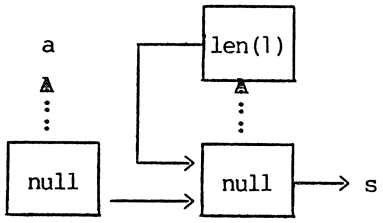


Fig. 3.3. Path diagram for ARB.

Now suppose we want to optimize this diagram by removing the first NULL node and attaching the alternate to the LEN(1) node (figure 3.4).

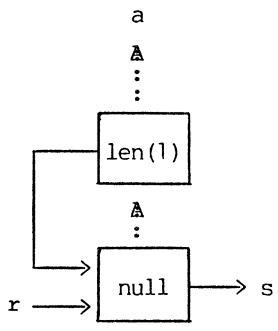


Fig 3.4. Optimized path diagram for ARB.

Does this path diagram implement ARB? The derived pattern of the optimized diagram is

(1) D(r) = p(NULL) -- D(s) | LEN(1) -- D(r) | D(a)

If we suppose that the diagram is correct, the expected value

D(r) = ARB -- D(s) | D(a)

may be substituted in (1), giving

D(r) = D(s) | LEN(1) -- (ARB -- D(s) | D(a)) | D(a)
= D(s) | (LEN(1) -- ARB -- D(s) | LEN(1) -- D(a)) | D(a)

which leads to the contradiction:

ARB -- D(s) | D(a) = ARB -- D(s) | LEN(1) -- D(a)

Hence the optimized path diagram does not implement ARB correctly.

3.4. A closer examination of alternation

The directed graph representation of patterns leads to an efficient pattern matching algorithm. However, some observations can be made which lead to improvement.

Empirical evidence shows that the number of occurrences of the subsequentiation operator heavily dominates the number of occurrences of the alternation operator. As a result, the majority of alternate links in the directed graph will be empty. This suggests to use a directed graph representation with subsequent linkage only and to introduce a new primitive operator to indicate alternation explicitly. A considerable amount of space for pattern storage is saved by this method.

The next idea is to make alternation right associative. Consider the pattern $P_1 \mid P_2 \mid P_3$ which can be written in prefix form as:

ALT(ALT(P_1, P_2), P_3)

Execution time and space on the pattern matching history stack can be saved if this is changed into:

$$ALT(P_1, ALT(P_2, P_3))$$

In this way only one untried alternative needs to be remembered at a time.

Figures 3.5. and 3.6. give some measurements of the static (i.e. number of static occurrences in patterns) and dynamic (i.e. dynamic usage count during pattern matching) frequency of some primitive pattern components. Measurements were done during a self compilation of the SPRING compiler. The low static frequency of alternation motivated the removal of explicit alternation linkage in the representation of patterns.

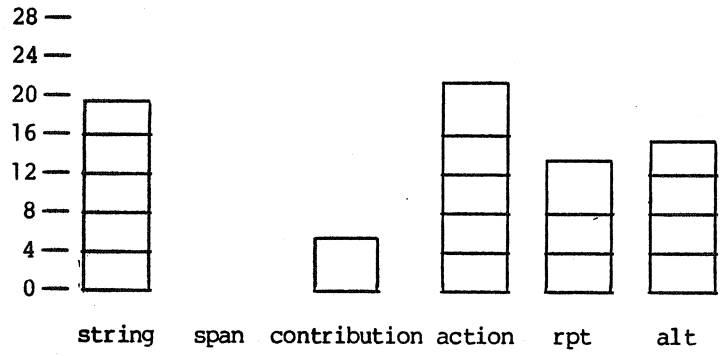


Fig. 3.5. Static frequency of some primitive patterns (in %).

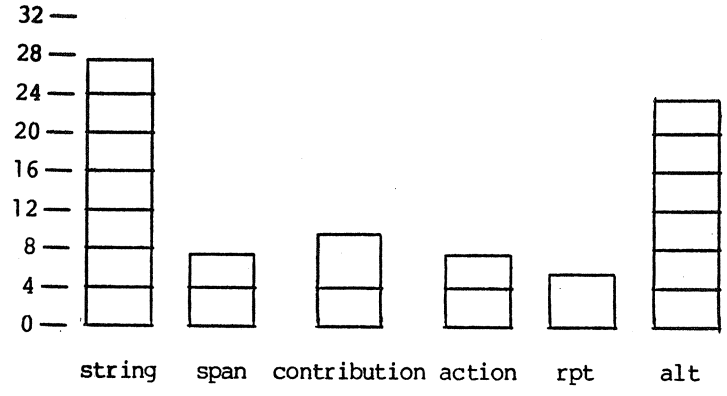


Fig. 3.6. Dynamic frequency of some primitive patterns (in %).

REFERENCES

[1] YNGVE, V.H., *Computer programming with COMIT II,* MIT press, 1972.

[2] CHRISTENSEN, C., *Examples of symbol manipulation in the AMBIT programming language,* Proceedings ACM 20th National Conference, 1965, 247-261.

[3] GRISWOLD, R.E., J.F. POAGE & I.P. POLONSKY, *The SNOBOL4 programming language,* Second edition, Prentice-Hall, Englewood Cliffs, N.J.

[4] GRISWOLD, R.E. & D.R. HANSON, *An overview of the SL5 programming language,* SL5 project document S5LD1b, The University of Arizona, Tucson, Arizona, October 9, 1976.

[5] DOYLE, J.N., *A generalized facility for the analysis and synthesis of strings, and a procedure based model of an implementation,* S4D48, The University of Arizona, Tucson, Arizona, February 11, 1975.

[6] GIMPEL, J.F., *Blocks - a new datatype for SNOBOL4,* CACM 15 (1972)6, 438-447.

[7] GIMPEL, J.F., *Algorithms in SNOBOL4,* Wiley and sons, 1976.

[8] FARBER, D.J., R.E. GRISWOLD & I.P. POLONSKY, *SNOBOL a string manipulation language,* JACM 11 (1964)1, 21-30.

[9] McCLURE, R.M., *TMG - a syntax directed compiler,* ACM 20th National Conference, 1965, 262-274.

[10] McILROY, M.D., *A manual for the tmg compiler-writing language,* Bell Laboratories, Murray Hill, New Jersey, 1972.

[11] GIMPEL, J.F., *A theory of discrete patterns and their implementation in SNOBOL4,* CACM 16 (1973)2, 91-100.

[12] STEWART, G.F., *An algebraic model for string patterns,* Second Symposium on Principles of Programming Languages, Palo Alto, January 20-22 1975, 167-184.

AXIOMATIEK VAN DATASTRUCTUREN

H.J.M. GOEMAN, A. OLLONGREN, TH.P. VAN DER WEIDE
Rijksuniversiteit Leiden

1. INLEIDING

In de informatica zijn informatie en bewerking van informatie objecten van onderzoek. Deze worden beschreven in wiskundige structuren, waarvoor termen als datastructuren en informatiestructuren gebruikt worden. Aanvankelijk werden datastructuren beschreven met behulp van natuurlijke talen, en een minimum aan formele middelen. Maar omdat zulke structuren gecompliceerd bleken, werd meer en meer de behoefte gevoeld ze formeel te beschrijven, en kwamen wiskundige technieken meer in aanmerking. We zullen in deze voordracht een axiomatisering van datastructuren invoeren en motiveren en we zullen laten zien dat deze een universeel karakter heeft, dat wil zeggen, dat tal van bekende datastructuren ermee beschreven kunnen worden. Van vroegere studies, die van invloed zijn geweest op deze bijdrage noemen wij slechts de Weense Definitiemethode en het werk van A. OLLONGREN ([1]), en van H.D. EHRICH ([2]). Een eerder stadium van dit onderzoek, waaraan ook werd deelgenomen door J.A. Bergstra en G.A. Terpstra, resulteerde in [3]. Verder zijn resultaten te vinden in [4] en [5].

2. WAAROM EEN AXIOMATISCHE BENADERING VAN DATASTRUCTUREN?

Aan het begrip datastructuur kunnen we twee belangrijke facetten onderscheiden:

(1) een datastructuur is een geheugenmedium: het dient voor de opslag van gegevens;
(2) een datastructuur is gestructureerd: het bestaat uit componenten die op de een of andere manier bijeengenomen zijn.

Beide facetten van het begrip datastructuur kunnen door middel van axioma's gekarakteriseerd worden.

Omdat alle typen datastructuren het karakter van een geheugenmedium hebben, is een gemeenschappelijke axiomatische karakterisering van dit facet mogelijk.

Voor de onderliggende structuur bestaat daarentegen een grote verscheidenheid van mogelijkheden. Het lijkt dus meer voor de hand te liggen voor verschillende klassen van datastructuren verschillende axiomatiseringen van dit facet te ontwerpen. Toch kan men ook ten aanzien van dit facet streven naar een zeer algemene karakterisering, waarbinnen allerlei klassen van datastructuren gedefinieerd kunnen worden.

Als voordelen van een axiomatische karakterisering worden veelal genoemd:

(1) gemeenschappelijke eigenschappen van datastructuren worden benadrukt (b.v. de toekenning!);
(2) een bewijstheorie voor eigenschappen van (klassen van) datastructuren wordt mogelijk (vgl. inductie-eigenschappen);
(3) een axiomatische karakterisering is een representatie-onafhankelijke karakterisering, zonder overbodige overspecificatie.

In onze axiomatiek zijn de beide genoemde facetten duidelijk onderscheiden. We hebben ons vooral met het geheugenfacet beziggehouden (het begrip toekenning speelt daarbij een belangrijke rol).

Het is nog niet helemaal duidelijk of een algemene axiomatische karakterisering van het structuurfacet, dan wel een specifieke benadering voor klassen van datastructuren afzonderlijk, de voorkeur verdient. Beide mogelijkheden worden nader toegelicht.

3. GELAAGDE STRUCTUREN

Wanneer we objecten uit de reële wereld willen beschrijven, kiezen we eerst voor een verzameling aspecten die we aan een object wensen te onderscheiden. Wanneer een object (of geheugen) aspecten heeft, en dus niet elementair is, zal het een gestructureerd karakter hebben en is het opgebouwd met behulp van andere objecten. Zo'n niet-elementair object wordt beschreven door het waarderen van zijn aspecten. De waarde van een aspect beschouwen we ook als een object. Merk op dat de beschrijving van een object hierdoor een gelaagd karakter krijgt. Om objecten te beschouwen is het nodig over operaties te beschikken. Voor het gemak zullen we slechts één operatie toestaan: de toekenning. Opdat we zullen kunnen rekenen met

objecten, zullen we eisen dat elk object constructief te beschrijven is. Het zal duidelijk zijn, dat het in deze opzet mogelijk is een object in kleinere eenheden te ontbinden.

We hebben de opzet zo algemeen en omvattend mogelijk gekozen. Dit geeft ons vele mogelijkheden tot classificatie, dit is het invoeren van deelklassen van grootheden met gelijksoortige eigenschappen.

Het gelaagde karakter van objecten levert nog andere mogelijkheden. Door groepen van aspecten als een nieuw aspect te beschouwen zien we tijdelijk af van details van een object. De details worden aldus naar een hoger niveau verplaatst, waar we uiteraard hetzelfde proces kunnen herhalen.

We zullen in het vervolg de aspecten die we aan objecten onderscheiden, benoemen met de namen uit een verzameling S. We noemen de elementen van S selectoren. De verzameling van abstracte objecten noemen we O. We veronderstellen dat O een nulelement Ω bevat. De operator om uit abstracte objecten nieuwe abstracte objecten te construeren noemen we ν. Het type van ν is: $O \times S \times O \rightarrow O$. De ν-operator geeft aan een aspect van een object een nieuwe waarde. Anders gezegd: de ν-operator herdefinieert een geheugen op een bepaalde plaats. Krijgt het aspect de waarde Ω, dan betekent dat, dat het aspect niet meer aanwezig is in het abstracte object.

Een aantal eigenschappen die voor de hand liggen, zijn:
- er zijn andere geheugens dan Ω
- een geheugen is op elke plaats uniek gevuld (desnoods met Ω)
- het nulelement en de elementaire objecten zijn overal leeg (met Ω gevuld)
- het resultaat van herdefiniëren van een geheugen op een bepaalde plaats zal zijn, dat het geheugen op die plaats het erin gestopte geheugen bevat
- de verschillende aspecten aan een object kunnen onafhankelijk van elkaar gewijzigd worden.

De al eerder aangestipte elementaire objecten worden beschreven door lege objecten $\neq \Omega$:
- als twee objecten dezelfde aspecten hebben, dan zijn ze gelijk, of beiden leeg
- elementaire geheugens kunnen alleen op een triviale manier geconstrueerd worden
- niet elementaire geheugens kunnen in eindig veel stappen met de ν-operator, vanuit Ω, geconstrueerd worden.

4. DE AXIOMATISERING VAN GELAAGDE STRUCTUREN

In de formele beschouwing kijken we naar structuren $\mathcal{O}$ van de vorm

$$\mathcal{O} = \langle O,S,\nu,\Omega\rangle$$

waarin:
- de elementen van O heten objecten
- de elementen van S heten selectoren
- Ω is een constante in O
- ν, de toekenningsoperator, is van type $O \times S \times O \to O$.

Verder eisen we dat $\mathcal{O}$ aan de volgende axioma's voldoet: (hoofdletters staan voor objecten, griekse letters voor selectoren) *)

A1: $\exists_A[A \neq \Omega]$

A2: $\exists_B[A = \nu(A,\alpha,B)]$

A3: $\nu(A,\alpha,B) = \nu(A,\alpha,C) \Rightarrow B = C.$

Met behulp van A2 en A3 vinden we:

er is een unieke B, zodat $\nu(A,\beta,B) = A$.

Deze unieke B zullen we aangeven met $A(\beta)$, het resultaat van applicatie in A via β.

A4: $\Omega(\alpha) = \Omega.$

We voeren empty(A) in als afkorting voor $\forall_\alpha [\ A(\alpha) = \Omega\ ]$, en el(A) voor empty(A) $\wedge$ $A \neq \Omega$.

A5: $\nu(\nu(A,\beta,B),\beta,B) = \nu(A,\beta,B)$

---

*) In alle formules zal $\Rightarrow$ steeds de laagste prioriteit hebben.

A6: $\beta \neq \gamma \Rightarrow \nu(\nu(A,\beta,B),\gamma,C) = \nu(\nu(A,\gamma,C),\beta,B)$

A7: $\forall_\gamma [A(\gamma) = B(\gamma)] \Rightarrow A = B \vee (\text{empty}(A) \wedge \text{empty}(B))$
(extensionaliteit)

A8: $\text{el}(A) \wedge A = \nu(B,\gamma,C) \Rightarrow A = B \wedge C = \Omega$

A9: Als $\Phi(A)$ een eigenschap is voor objecten, zodat:
- $\Phi(A)$ geldt voor alle A met empty(A)
- $\Phi(A) \wedge \Phi(B) \Rightarrow \Phi(\nu(A,\beta,B))$ voor alle A,β,B met $A(\beta) = \Omega \wedge B \neq \Omega$,

dan volgt:

$$\forall_A [\ \Phi(A)\ ]$$

(ν-inductie).

Om voor de hand liggende redenen zullen we in het vervolg soms $\alpha \in A$ schrijven i.p.v. $A(\alpha) \neq \Omega$.
Verder voeren we een relatie in voor objecten, d.m.v.:

$$A \leq B \equiv \forall_{\alpha \in A} [A(\alpha) = B(\alpha)] \wedge (\text{empty}(B) \Rightarrow (A = \Omega \vee A = B))$$

$$A < B \equiv A \leq B \wedge \neg(B \leq A).$$

Uit deze axioma's zijn de volgende stellingen af te leiden:

STELLING: (diepte-inductie)

Als $\Phi(A)$ *een eigenschap is voor objecten, zodat:*
- $\Phi(A)$ *geldt voor alle* A *met* empty(A)
- $\forall_\alpha [\ \Phi(A(\alpha))\ ] \Rightarrow \Phi(A)$ *voor alle* A

dan volgt: $\forall_A [\ \Phi(A)\ ]$.

STELLING 2: (domein-inductie)

Als $\Phi(A)$ *een eigenschap is voor objecten, zodat:*

$$\forall_{B<A} [\ \Phi(B)\ ] \Rightarrow \Phi(A) \quad \textit{voor alle A}$$

dan volgt: $\forall_A [\ \Phi(A)\ ]$.

Beide stellingen zullen we hier niet bewijzen.
Wel merken we op dat ν-inductie bewijsbaar is uit A1 t/m A8, diepte- en domein-inductie (vgl. [3]).

5. STANDAARDMODELLEN

Laat E een niet-lege verzameling, en $\Omega \notin E$. Laat S een verzameling. Dan construeren we het standaardmodel $\mathcal{O}_S^E$ als volgt.

We definiëren: $O := \bigcup_{i \geq 0} O_i$

waarin: $O_0 := E \cup \{\Omega\}$

$O_{i+1} := \{f: S \to \bigcup_{k=0}^{i} O_k \mid f(\alpha) \neq \Omega$ voor tenminste één, maar niet meer dan eindig veel selectoren $\alpha\}$.

Om de ν-operator in te voeren, zullen we eerst met elke $A \in O$ een functie $\hat{A}$ associëren volgens:

$$\hat{A} = \begin{cases} \lambda\alpha\cdot\Omega & \text{als } A \in O_0, \\ A & \text{anders .} \end{cases}$$

We definiëren nu voor $A,B \in O$ en $\beta \in S$

$$\nu(A,\beta,B) = \begin{cases} A & \text{als } A \in O_0 \text{ en } B = \Omega \\ \Omega & \text{als } A \notin O_0 \text{ en } B = \Omega \text{ en } \forall_{\alpha \neq \beta}[A(\alpha) = \Omega] \\ \lambda\alpha\cdot[\textit{if } \alpha=\beta \textit{ then } B \textit{ else } \hat{A}(\alpha)] & \text{anders} \end{cases}$$

Het is eenvoudig in te zien dat dit model voldoet aan A1 t/m A9. Bijgevolg zijn de axioma's A1 t/m A9 consistent.

6. BIJZONDERE MODELLEN

Door bijzondere keuzen te maken voor de verzameling S van selectoren, en voor de verzameling E van elementaire objecten, vinden we een aantal bijzondere modellen $\mathcal{O}_S^E$.

Allereerst kiezen we voor S een singleton, zeg $S = \{s\}$. Een aldus verkregen standaardmodel kunnen we zien als een stapelconcept. Stapeling en ontstapeling kunnen uitgevoerd worden met behulp van respectievelijk de ν-operatie en de applicatie-operatie. Dankzij de diepte van objecten, en het gemis van een zogenaamde "higher-level-application" verloopt het opslaan en terughalen van informatie via de bodemlaag, zoals bij de stapel. We kunnen de objecten in zo'n model zien als representanten van de natuurlijke getallen. Als demonstratie zullen we voor dit model een programmeertaal ontwikkelen.

De programma's mogen alleen variabelen van het type object inspecteren en wijzigen. Programmavariabelen worden aangegeven met de symbolen X,Y,... Het ligt voor de hand om de volgende typen opdrachten te kiezen als formaat voor primitieve opdrachten:

$X := \nu(\Omega,s,X)$,

$X := X(s)$.

Deze operaties komen respectievelijk overeen met X := X+1 als $X \neq \Omega$, en X := X÷1. Met behulp van de *if*-*fi*-constructie en de *do*-*od*-constructie bouwen we grotere programma's uit kleinere (zie DIJKSTRA [6]). Als elementaire test zouden we, ingeval E een singleton is, willen kiezen: $X = \Omega$. Ingeval E een grotere verzameling is, kiezen we liever de beide elementaire testen: empty(X) en $X \doteq Y$ ($X \doteq Y$ is waar d.e.s.d. als empty(X) $\wedge$ empty(Y) $\wedge$ X = Y). Merk op dat in dit geval $X \doteq \Omega$ equivalent is met $X = \Omega$.

Op de eerste plaats willen we laten zien dat het programma

do $X \neq \Omega \rightarrow X := X(s)$ *od*

voor alle beginwaarden van X eindigt.
Dit is triviaal, als we weten dat de beginwaarde van X voldoet aan empty(X). Veronderstel nu: het programma eindigt bij beginwaarde A, voor zekere $A \neq \Omega$. Het is dan eenvoudig in te zien, dat het programma dan ook eindigt bij de beginwaarde $\nu(\Omega,s,A)$.
Met behulp van ν-inductie, en de overweging dat $\nu(\Omega,s,A) = \nu(B,s,A)$ voor alle waarden van B, volgt het gestelde.

Vervolgens geven we een voorbeeld van een programma, dat, in termen van natuurlijke getallen, de volgende aktie realiseert:

X,Y := Y *div* 2,0.

Een programma hiervoor is:

```
X,Y := e,Y(s);     {e is een elementair object}
do  Y ≠ Ω → X,Y := ν(Ω,s,X), Y(s)(s)  od.
```

De correctheid van dit programma kan worden ingezien, door als doelrelatie

te kiezen

$$(A - 2*X) \div 1 = 0 \wedge Y = 0 .$$

Deze doelrelatie moet worden gerealiseerd vanuit de beginrelatie

$$Y = A .$$

Als verzwakking van de doelrelatie kiezen we de invariant:

$$(A - 2*X) \div 1 = Y .$$

Een algemener stapelconcept wordt verkregen door voor S een grotere eindige verzameling te kiezen.
Laat $|S| = n$. We kunnen nu spreken van n-aire dieptebomen. Ook kunnen we een zo verkregen standaardmodel zien als een multilevel array (gelijksoortig aan ROSENBERG-THATCHER [11]).

In het bijzondere geval dat S precies twee elementen bevat ontstaat een klasse van datastructuren die veel overeenkomsten vertoont met de symbolische expressies (S-expressies) van J. McCarthy. De S-expressies zijn de enige datastructuren die in de door McCarthy ontwikkelde taal Lisp voorkomen. S-expressies zijn opgebouwd uit de symbolen .,(,) en een tevoren gegeven (oneindige) verzameling van atomaire symbolen (waaronder het symbool NIL) en worden als volgt gedefinieerd:

a) atomaire symbolen zijn S-expressies (atomaire S-expressies)
b) als X en Y S-expressies zijn, dan ook (X.Y).

Elementaire funkties over S-expressies zijn:

1) het predicaat atom [X] dat aangeeft of een S-expressie X een atomair element is of niet,
2) het predicaat eq[X,Y] dat alleen is gedefinieerd als beide S-expressies X en Y atomair zijn en dat over de identiteit van atomaire expressies beslist,
3) de operaties car[X] en cdr[X], gedefinieerd als X niet atomair is en wel zo dat car[(X.Y)] = X en cdr[(X.Y)] = Y,
4) de operatie cons[X,Y] gedefinieerd voor alle S-expressies X en Y en wel zo dat cons[X,Y] = (X.Y).

De overeenkomstige operaties in O_S^E als voor E de verzameling van atomaire expressies, uitgezonderd NIL wordt gekozen en voor S de verzameling van selectoren {left, right} zouden zijn:

1) het predicaat empty(X),
2) het predicaat = voor gelijkheid tussen objecten,
3) de applicaties X(left) en X(right),
4) de operatie $\nu(\nu(\Omega,\text{left},X), \text{right},Y)$.

Merk op dat het nul object Ω een niet geheel identieke rol vervult als NIL, met het gevolg dat al deze operaties totaal gedefinieerd zijn.

Andere modellen worden verkregen door voor S de vrije halfgroep $<\Sigma^*,*,\epsilon>$ te kiezen over een gekozen verameling Σ.
Zo'n model bevat als een deelstructuur de Weense Objecten ([1]). Daartoe voeren we in het predicaat W(A):

$$W(A) \equiv \forall_\alpha\, [A(\alpha) \neq \Omega \Rightarrow (el(A(\alpha)) \wedge \forall_\beta\, [A(\alpha*\beta) = \Omega])].$$

Bovendien bevat dit model de objecten, zoals deze door EHRICH ([2]) ingevoerd werden, voorzover ze een eindig domein hebben. Dit kan worden ingezien met behulp van het volgende predicaat E(A):

$$E(A) \equiv \forall_\alpha\, [empty(A(\alpha))].$$

Tenslotte bevat dit model de associatieve geheugens (verzamelingen van associons, zie M. REM [7]). Immers, een associon is niets anders dan een tupel van identifiers.

7. ENKELE EIGENSCHAPPEN

Enkele gevolgtrekkingen uit de axioma's zijn:

Eig.1: $\nu(A,\beta,B)\ (\beta) = B$. Deze eigenschap is slechts een andere schrijfwijze voor axioma A5.

Eig.2: $\alpha \neq \beta \Rightarrow \nu(A,\beta,B)\ (\alpha) = A(\alpha)$.

bewijs: $\nu(\nu(A,\beta,B), \alpha, A(\alpha))$

$= \nu(\nu(A,\alpha,A(\alpha)), \beta, B)$ (vanwege A6)

$= \nu(A,\beta,B)$ (vanwege definitie van $A(\alpha)$)

M.b.v. de definitie van $\nu(A,\beta,B)\ (\alpha)$ volgt het gestelde. □

Eig.3: $\nu(\nu(A,\beta,C),\ \beta,\ B) = \nu(A,\beta,B)$ tenzij $el(A) \wedge B = \Omega \wedge C \neq \Omega$ (in dat geval geldt $\nu(\nu(A,\beta,C),\ \beta,\ B) = \Omega \neq A = \nu(A,\beta,B)$).

bewijs: Laat $L = \nu(\nu(A,\beta,C),\ \beta,\ B)$
en $R = \nu(A,\ \beta,\ B)$.
M.b.v. Eig.1 en Eig.2 volgt: $\forall_\alpha\ [\ L(\alpha) = R(\alpha)\ ]$
Vanwege extensionaliteit (A7) volgt dus

$$L = R \vee (empty(L) \wedge empty(R)).$$

Veronderstel $L \neq R$, dan volgt $empty(L) \wedge empty(R)$ en dus $el(L) \vee el(R)$. Als $el(R)$, dan volgt $el(A) \wedge B = \Omega$ wegens A8. Als $el(L)$, dan volgt $el(A) \wedge B = \Omega$ door tweemaal toepassen van A8. Dus zeker $el(A) \wedge B = \Omega$. Omdat $L \neq R$ volgt $C \neq B$ wegens A5, en dus $C \neq \Omega$. □

Eig.4: $\exists_A\ [el(A)]$.

bewijs: Met behulp van ν-inductie volgt: $\forall_B\ [B \neq \Omega \Rightarrow \exists_A\ [el(A)]]$.
Dan levert A1 het gestelde. □

Ook kunnen binnen de structuur operaties ingevoerd worden. We geven een tweetal voorbeelden:

Domeinvereniging: $\forall_{A,B}\ \exists_C\ \forall_\alpha\ [\ C(\alpha) \neq \Omega \iff (A(\alpha) \neq \Omega \vee B(\alpha) \neq \Omega)\ ]$

bewijs: ν-inductie naar A. Als $empty(A)$, dan kies $C = B$. Stel nu dat A de eigenschap heeft, en $A(\delta) = \Omega$, $D \neq \Omega$ en $E = \nu(A,\delta,D)$. Laat B een object, en laat C' vervolgens een object zodat $\forall_\alpha\ [\ C'(\alpha) \neq \Omega \iff (A(\alpha) \neq \Omega \vee B(\alpha) \neq \Omega)\ ]$. Kies $C = \nu(C,\delta,D)$. Dan volgt:

$$\forall_\alpha\ [\ C(\alpha) \neq \Omega \iff (A(\alpha) \neq \Omega \vee B(\alpha) \neq \Omega)\ ].$$ □

Projectie: $\forall_A \exists_B \forall_\beta\ [\ B(\beta) \neq \Omega \iff \exists_\alpha\ [\ A(\alpha)(\beta) \neq \Omega\ ]]$

bewijs: slaan we over. □

Binnen de verzameling objecten kunnen deelklassen gekarakteriseerd worden door een predicaat. De volgende stelling kan in zo'n geval hulp bieden om eigenschappen van zo'n deelstructuur te bewijzen.

STELLING: (Raamstelling)

Laat Ψ *een eigenschap voor objecten zodat:*

- $\text{empty}(A) \Rightarrow \Psi(A)$
- $\Psi(A) \wedge \neg \text{emtpy}(A) \Rightarrow \exists_{\alpha}[A(\alpha) \neq \Omega \wedge \Psi(\nu(A,\alpha,\Omega))]$

en laat Φ *een eigenschap voor objecten zodat:*

- $\text{empty}(A) \Rightarrow \Phi(A)$
- $(\Psi(A) \wedge C = \nu(A,\beta,B) \wedge A < C \wedge \Psi(C)) \Rightarrow (\Phi(A) \Rightarrow \Phi(C))$,

dan geldt: $\forall_A [\ \Psi(A) \Rightarrow \Phi(A)\ ]$.

Het bewijs van deze stelling zullen we eveneens achterwege laten.

8. EINDIGE GRAFEN ALS ONDERLIGGENDE STRUCTUUR

Vele datastructuren hebben als onderliggende structuur niet alleen een verzameling waarvan de elementen aangeduid worden als knopen, maar ook een verzameling, waarvan de elementen aangeduid worden als takken. Bovendien kunnen, eventueel onder zekere restricties, knopen en takken worden toegevoegd of verwijderd. Zie bijvoorbeeld MAJSTER [8].

We zullen daarom nu een axiomatische beschrijving van dergelijke onderliggende structuren geven.

Als onderliggende structuren worden toegestaan structuren van de vorm

$$< S, \text{node}, \text{from}, \text{to} >$$

waarin:

- node is een predicaat over S
- from is een partiële functie van type $S \rightarrow S$
- to is een partiële functie van type $S \rightarrow S$

Allereerst voeren we in het predicaat edge, d.m.v.

$$\text{edge}(\alpha) \equiv \neg \text{node}(\alpha) .$$

We eisen de volgende axioma's:

S1: $\text{from}(\sigma) = \beta \Rightarrow \text{edge}(\sigma) \wedge \text{node}(\beta)$

S2: $\text{to}(\sigma) = \beta \Rightarrow \text{edge}(\sigma) \wedge \text{node}(\beta)$

S3: $\text{edge}(\sigma) \Rightarrow \exists_{\alpha,\beta} [\ \text{from}(\sigma) = \alpha \wedge \text{to}(\sigma) = \beta\ ]$

S4: $\text{node}(\alpha) \wedge \text{node}(\beta) \Rightarrow \exists_{\sigma} [\ \text{from}(\sigma) = \alpha \wedge \text{to}(\sigma) = \beta\ ]$

S5: $(\text{from}(\sigma) = \text{from}(\tau) \wedge \text{to}(\sigma) = \text{to}(\tau)) \Rightarrow \sigma = \tau.$

Als $\text{node}(\alpha) \wedge \text{node}(\beta)$ geldt, dan geven we de unieke tak σ, zodat:

$$\text{from}(\sigma) = \alpha \wedge \text{to}(\sigma) = \beta,$$

aan met $< \alpha, \beta >$.

De aldus verkregen structuren geven we aan met $\mathcal{O}_G$.
Binnen de aldus opgebouwde structuren $\mathcal{O}_G$ geven we een paar interessante deelklassen van objecten d.m.v. predicaten aan, de cirkelobjecten en de lijnobjecten.

Allereerst beschrijven we objecten, die alleen gevuld zijn in takken als ze ook in het bijbehorende begin- en eindpunt gevuld zijn, d.m.v.

$$\text{pointed}(A) \equiv \forall_{\alpha,\beta}[\, <\alpha,\beta> \in A \Rightarrow (\alpha \in A \wedge \beta \in A)\,].$$

In objecten kunnen we begin- en eindpunten onderscheiden d.m.v.:

$$\text{entry}(A,\alpha) \equiv \text{node}(\alpha) \wedge \alpha \in A \wedge \forall_\beta\, [\, <\beta,\alpha> \notin A\,]$$
$$\text{exit}(A,\alpha) \equiv \text{node}(\alpha) \wedge \alpha \in A \wedge \forall_\beta\, [\, <\alpha,\beta> \notin A\,] .$$

Dan kunnen we nu beschrijven, wanneer een object louter uit cykels is opgebouwd, d.m.v.:

$$\text{cycling}(A) \equiv \text{pointed}(A) \wedge \forall_\alpha\, [\, \neg\, \text{entry}(A,\alpha) \wedge \neg\, \text{exit}(A,\alpha)].$$

Een cirkel is een object dat uit één cykel is opgebouwd, dus

$$\text{cycle}(A) \equiv \text{cycling}(A) \wedge \forall_{B<A}\, [\, \text{cycling}(B) \Rightarrow \text{empty}(B)\,] .$$

Een lijnobject (keten) kunnen we nu eenvoudig beschrijven d.m.v.

$$\text{chain}(A) \equiv \text{empty}(A) \vee$$
$$\exists_{\alpha,\beta}\, [\, <\alpha,\beta> \notin A \wedge \forall_{B\neq\Omega}\, [\, \text{cycle}(\nu(A,<\alpha,\beta>,B))\,]\,] .$$

In de volgende paragraaf volgen andere voorbeelden van te definiëren deelstructuren.
Het is niet moeilijk aan te tonen dat elke niet-lege keten een unieke beginknoop, en een unieke eindknoop heeft.
Een belangrijk hulpmiddel bij het gebruik van ketens kan de volgende stelling zijn, de z.g. keteninductie.

STELLING: (keteninductie)

Als $\Phi(A)$ *een eigenschap is voor objecten, zodat:*

- $\text{empty}(A) \Rightarrow \Phi(A)$
- $\Phi(A) \Rightarrow \Phi(\nu(A,\beta,B))$ *voor alle* A,β,B
  met $\text{chain}(A) \wedge A(\beta) = \Omega \wedge B \neq \Omega \wedge \text{chain}(\nu(A,\beta,B))$

dan volgt:

$$\forall_A [\ \text{chain}(A) \Rightarrow \Phi(A)\ ] .$$

bewijs: Pas de raamstelling uit 7. toe. □

9. DEFINIEERBARE SUBSTRUCTUREN

Naast de cirkel en de keten zijn nog andere deelklassen definieerbaar binnen O_G.
We geven als voorbeeld nog de bomen:

$$\text{tree}(A) \equiv \text{empty}(A) \vee \Big(\text{pointed}(A) \wedge \exists!_{\alpha \in A} [\ \text{entry}(A,\alpha)\ ] \wedge \forall_{\alpha,\beta,\gamma} [\ \langle\beta,\alpha\rangle \in A \wedge \langle\gamma,\alpha\rangle \in A \Rightarrow \beta = \gamma\ ] \wedge \forall_{B<A} [\ \neg\, \text{cycling}(A)\ ] \Big) .$$

Het zal duidelijk zijn dat deze (formele) manier van definiëren een ingewikkelde beschrijving kan vereisen.
Vaak zal het handiger zijn een inductieve definitie te gebruiken. Het is echter de vraag of een inductief beschreven subklasse ook door een formule uit de 1e-orde taal beschreven kan worden.
In praktische situaties, bijvoorbeeld de mode-declaration in Algol 68 ([9]) en het record-concept in Pascal ([10]), lijkt dit steeds het geval.
Men zal daarom gebruik mogen maken van inductieve definities, in die situaties waarin het mogelijk is een formele weg, zoals aangegeven bij ketens en bomen, te bewandelen.

LITERATUUR

[1] OLLONGREN, A., *Definition of programming languages by interpreting automata,* Acad. Press, APIC Series no. 11 (1974).

[2] EHRICH, H.D., *Outline of an algebraic theory of structured objects,* Automata, Languages and Programming, Third International Colloquium, Edinburgh University Press (1976).

[3] BERGSTRA, J.A., H.J.M. GOEMAN, A. OLLONGREN, G.A. TERPSTRA & TH.P. VAN DER WEIDE, *Axioms for multilevel objects,* Revision of Report no. 76-7, Inst. Toegep. Wsk. en Inform., Rijksuniversiteit Leiden (1976).

[4] BERGSTRA, J.A., A. OLLONGREN & TH.P. VAN DER WEIDE, *Rational objects,* in: Fundamentals of Computation Theory, Lecture Notes in Computer Science 56 (1977) 33-38.

[5] BERGSTRA, J.A., A. OLLONGREN & TH.P. VAN DER WEIDE, *Multilevel set objects,* Report no. 77-10, Inst. Toegep. Wsk. en Inform., Rijksuniversiteit Leiden (1977).

[6] DIJKSTRA, E.W., *A discipline of programming,* Prentice Hall Inc., Englewood Cliffs, N.J. (1976).

[7] REM, M., *Associons and the closure statement,* Mathematical Centre Tracts 76, Amsterdam (1976).

[8] MAJSTER, M.E., *Extended directed graphs, a formalism for structured data and data structures,* Acta Informatica 8 (1977) 37-59.

[9] WIJNGAARDEN, A. VAN, e.a. (eds), *Revised report on the algorithmic language ALGOL 68,* Acta Informatica 5 (1975) 1-236.

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

[11] ROSENBERG, A.L. & J.W. THATCHER, *What is a multilevel array,* IBM Journal of Research and Development, vol. 19, No. 2 (1975) 163-169.

DATASTRUCTUREN VOOR LINEAIRE RUIMTEN 'TORRIX'

S.G. VAN DER MEULEN, M. VELDHORST
Rijksuniversiteit Utrecht

1. INLEIDING EN AXIOMATISCH FRAME

1.1. Inleiding

Een axiomatisch systeem generaliseert door weglating en specificeert door aanvaarding van bepaalde eigenschappen (relaties, functies, operaties etc.). Een praktische betekenis van zo'n abstract systeem ligt in de verscheidenheid van interpretaties die het toelaat - het kan fungeren als een hard en betrouwbaar frame voor alle mogelijke verzamelingen objecten waarvan de eigenschappen voldoen aan de gestelde axioma's. Zo kunnen polynomen en formele machtreeksen (ook met negatieve exponenten), een grote verscheidenheid van systemen met meerdere vrijheidsgraden, waarnemingsreeksen, statistische steekproeven en diverse configuraties uit de operationele wiskunde evenzeer worden geïnterpreteerd als vectoren in een n-dimensionale ruimte, als de meer traditionele interpretaties in het toch al zo omvangrijke gebied van lineaire stelsels vergelijkingen.

Axiomatische methoden worden meestal gehanteerd in een mathematische en/of logische context waarin theorieën worden getoetst door het bewijzen van stellingen. Er is verder een groeiend besef dat axiomatische abstractie ook een belangrijke rol kan spelen in het ontwerpen van programmeertalen en in het bijzonder in de beschrijving van datastructuren inclusief de op hun gedefinieerde operaties die in feite de datastructuur karakteriseren {14}.

Wij willen hier nagaan in hoeverre en op welke wijze een bestaand, bekend en ook belangrijk abstract algebraïsch systeem kan worden overgedragen op een vooral praktisch bruikbaar algoritmisch systeem. Met "overdragen" wordt hier uitdrukkelijk bedoeld: "implementeren tot op gebruikersnivo". In deze onderneming spelen bepaalde aspecten van "software-engineering" dan ook een

niet te verwaarlozen rol, zoals: een redelijke efficiëntie, een "user's-interface" zonder rode knoppen en gemakkelijke toegankelijkheid ook voor niet-informatici.

1.2. Axiomatische abstractie

Het aangeduide toepassingsgebied vinden we in de lineaire algebra waarvan vooral de moderne ontwikkeling wordt gekenmerkt door een aantal opvallende abstractie-stappen (zie ook {06}, {12} en {13}):

- van het lichaam der reële of complexe getallen (**R** of **C**) naar een willekeurig (mogelijk niet-commutatief, al of niet geordend, misschien ook eindig) ander grondlichaam F;
- van vectoren als rijtjes van n getallen (uit **R** of **C**) naar meer autonome, in principe zelfs coördinaten-vrije, objecten waarvan een lineaire afhankelijkheidsrelatie dan (in het eindig-dimensionale geval) voert naar het n-rijtje van elementen uit F als een van de mogelijke representaties;
- een daarmee samenhangende accentverschuiving van de coördinaat-gebonden coëfficienten-matrix naar het concept van de lineaire transformatie als centrale operator in een vectorruimte;
- het loslaten van één bepaalde vaste dimensie voor een subtieler systeem van naast, over en binnen elkaar gelegen (deel)ruimten, hun transformaties en hun projecties;
- verdergaande generalisaties van het grondlichaam F naar een ander algebraïsch grondsysteem, bijvoorbeeld naar een ring R (van vectorruimte naar vectormodule).

De algoritmische relevantie van dit omvangrijke toepassingsgebied behoeft geen betoog: eliminatiemethoden en velerlei andere iteratieve en recurrente processen spelen een belangrijke rol, programma's voor het oplossen van lineaire stelsels zijn nog in harde machinecode geschreven en de "array" is ongetwijfeld de meest klassieke van alle datastructuren.

Des te opvallender is het dat de hier geschetste ontwikkeling in de moderne wiskunde, niet of nauwelijks heeft geleid tot bijvoorbeeld een kritische bezinning op de datastructuren waarmee wordt gewerkt (en vaak op zeer grote schaal). In feite is het zelfs zo, dat de bestaande standaardlibraries voor vector-matrix bewerkingen op geen enkele wijze in hun structuur de toch bepaald niet verborgen gebleven groei van het wiskundig fundament weerspiegelen.

1.3. Lineaire ruimten

We gaan uit van een systeem S van scalairen $\alpha,\beta,\gamma,\text{---}\epsilon S$. Dit grond-systeem kan een lichaam F, of een ring R, of een nog ander algebraïsch relevant systeem zijn, bijvoorbeeld de verzameling N der natuurlijke getallen. We nemen minimaal aan dat in S een commutatieve optelling "+" is gedefinieerd en, althans partiëel, de bijbehorende inverse operatie "-" (S is minimaal een commutatieve additieve monoïde). S kan al dan niet geordend zijn en zeer wel ook eindig. Een vermenigvuldiging in S hoeft niet commutatief te zijn, wel wordt de distributiviteit over de optelling voorondersteld.

Ter vermijding van nodeloos langdradige uiteenzettingen, geven we de axioma's van een lineaire vectorruimte V over S alsof S een commutatief lichaam is (en voor 4 bovendien geordend). We nemen dus aan dat de lezer wel weet hoe en waar de gegeven regels moeten worden aangepast, mogelijk weggelaten.

De axioma's voor de vectoren $u,v,w,\text{---}\epsilon V$ zijn dan:

1) V is een commutatieve additieve (dwz. abelse) groep:

$u+(v+w)=(u+v)+w$ (associativiteit)

$u+v = v+u$ (commutativiteit)

$u+o = u$ (existentie unieke nulvector o)

$u-u = o$ (existentie unieke inverse $-u$)

2) De scalairen van S werken als lineaire operatoren op V:

$\alpha(u+v) = \alpha u+\alpha v$ (distributiviteit over vectoren)

$(\alpha+\beta)u = \alpha u+\beta u$ (distributiviteit over scalairen)

$(\alpha\beta)u = \alpha(\beta u)$ (associativiteit)

$1u = u$ (scalaire 1 = identiteitsoperator)

3) Op V zijn meer algemene lineaire transformaties $L: V\rightarrow V$ gedefinieerd, zodat:

$L(\alpha u+\beta v) = \alpha Lu+\beta Lv$ (lineariteit)

Voor lineaire transformaties A en B is een som A+B met een nul-transformatie O gedefiniëerd, zodat:

$(A+B)u = Au+Bu$ (distributiviteit)

$Ou = o$ (nul-transformatie)

Voor lineaire transformaties A en B is een product (functionele compositie) AB met eenheidstransformatie I gedefinieerd, zodat:

$(AB)u = A(Bu)$, $(AB)C = A(BC)$ (associativiteit)

$A(B+C) = AB+AC$, $(A+B)C = AC+BC$ (distributiviteit)

$AO = OA = O$, $AI = IA = A$ (nul en identiteit)

4) Indien S geordend is, kan V worden gedefinieerd als een inproduct-ruimte door een operatie $<,>$: $V \times V \rightarrow S$ met de eigenschappen:

$<u,v> = <v,u>$ (commutativiteit)

$<\alpha u+\beta v,w> = \alpha<u,w>+\beta<v,w>$ (distributiviteit)

$<u,u>\geq 0$, $<u,u>=0 \Leftrightarrow u=o$ (normeerbaarheid)

1.4. Basis-stellingen en representatie

Via de begrippen lineaire onafhankelijkheid ($\sum^{n}\alpha_i u_i = o \Leftrightarrow \alpha_i = 0$ voor alle $i=1,--,n$), lineair onafhankelijke basis $B \subset V$ (iedere eindige deelrij in B is lin. onafh. en elke $v \in V$ is te schrijven als $v=\sum \phi_i u_i$ met $u_i \in B$), eindig-dimensionale vectorruimte (B is eindig), komen we op het begrip dimensie (iedere basis in een eindig-dimensionale vectorruimte bevat evenveel vectoren $n=\dim V_n$). Een en ander culmineert in de fundamentele stelling:

Iedere n-dimensionale vectorruimte V_n
over (een lichaam) S, is isomorf met S^n

Het directe gevolg is dat iedere vector $v \in V_n$ geschreven kan worden als $v=(\phi_1,---,\phi_n)=\sum \phi_i e_i$ op de basis (e_i) met $e_i=(0,--,\underset{\uparrow\, i}{1},---,0)$ en dat iedere lineaire transformatie

$A: V_n \rightarrow V_m$

geschreven kan worden als een mxn matrix $A=(\alpha_{ij})$, $\alpha_{ij} \in S$, $i=1,--,m$, $j=1,--,n$.

Deze representatie geeft voor de axioma's 1 t/m 4 de volgende rekenregels:

1) $u \pm v \Rightarrow (\upsilon_1,---,\upsilon_n)\pm(\phi_1,---,\phi_n) = (\upsilon_1\pm\phi_1,---,\upsilon_n\pm\phi_n)$

2) $\alpha u \Rightarrow \alpha(\upsilon_1,---,\upsilon_n) = (\alpha\upsilon_1,---,\alpha\upsilon_n)$

3) $v=Au \Rightarrow (\phi_h) = (\sum_i^n \alpha_{hi}\upsilon_i)$

$A\pm B \Rightarrow (\alpha_{hk})\pm(\beta_{hk}) = (\alpha_{hk}\pm\beta_{hk})$

$AB \Rightarrow (\alpha_{hi})\times(\beta_{ik}) = (\sum_i^n \alpha_{hi}\beta_{ik})$

4) $<u,v> \Rightarrow <(\upsilon_i),(\phi_i)> = \sum_i^n \upsilon_i\phi_i$

indien S is **C**, wordt door $<u,v> \overset{def}{=} \sum \upsilon_i\overline{\phi_i}$ bereikt dat het inproduct $<u,v> \in$ **R** (geordend), zodat de normeerbaarheid kan worden gered ten koste van de commutativiteit: $<u,v>=\overline{<v,u>}$.

Nogal verrassend laat de isomorfiestelling een - wiskundig overigens triviale - uitbreiding van de rijtjes-representatie toe, die voor een implementatie van V duidelijke voordelen heeft.

1.5. Implementatie-criteria

Voor een enigszins adequate implementatie van dit veel-omvattende abstracte systeem is minimaal vereist:

I De mogelijkheid om het scalaire grondsysteem S zonder irrelevante specificaties te kunnen hanteren.

Anders gezegd: de vectorruimte V heeft er geen boodschap aan, wat voor een lichaam (resp. ring of ander passend systeem) S nu is. Direct voor de hand liggende realisaties van S zijn **R** (de reële getallen), **C** (de complexe getallen), **Q** (de rationale getallen), **Z** (de gehele getallen), $\mathbf{Z}_n$ (de restklassen modulo n, i.h.b. met n=p priem), **N** (de natuurlijke getallen) etc. Vanuit het standpunt van de wiskunde is dit een zeer beperkte en ook nogal eenzijdige bloemlezing.

II De mogelijkheid om vectoren en hun transformaties (matrices) te behandelen als autonome (dwz. niet-afgeleide) objecten.

Onze kijk op V is dus veeleer gegeven door de axioma's zoals geformuleerd in 1.3 dan door de isomorfe formulering van 1.4. Deze laatste is voor ons eigenlijk meer een kijk in de keuken. We weten allemaal dat 1.4 inderdaad de ingrediënten etaleert en dat onze gerechten ook zo worden bereid - waar mogelijk echter, prefereren we het menu à la 1.3.

III De mogelijkheid om gemakkelijk om te gaan met (deel)ruimten van verschillende dimensie, al dan niet overlappend.

Deze eis betekent in 1.3-terminologie dat we in staat willen zijn om te werken met arrays van tenminste dynamisch bepaalde, maar bij voorkeur ook variabele lengte. Een veel pregnantere parafrase is dat we in onze operaties (som, inproduct etc.) vectoren uit verschillende (deel)ruimten willen kunnen combineren.

IV De mogelijkheid de beschikbare geheugenruimte efficiënt te gebruiken zonder zelf te moeten passen en meten.

Dit criterium is natuurlijk van totaal andere orde dan I, II en III. Onmathematisch als het is, kan het echter niet buiten beschouwing blijven: grote lineaire systemen stellen bij uitstek het probleem van de geheugeneconomie. Ons standpunt is dat van een moderne implementatie minimaal verwacht mag worden dat het zelf bijhoudt welke informatie nog wel, en welke

niet meer actueel is en de (opnieuw) beschikbare geheugenruimte steeds zo weet te verkavelen dat optimaal aan de vraag kan worden voldaan. De technische term is natuurlijk "garbage collection". We zijn echter tevens van mening dat een eenvoudig spel ook met eenvoudige technieken gespeeld moet kunnen worden. Het is dan ook belangrijk de grens tussen meer en minder eenvoudig, duidelijk te kunnen markeren (zie pag. 111 en 114).

1.6. Implementatie-taal

In principe gaat de keuze voor de realisering van 1.5 tussen een zelfstandige programmeertaal, uitbreiding of aanpassing van een bestaande programmeertaal of realisering geheel binnen het kader van een bestaande programmeertaal.

De eerste mogelijkheid was (en is nog steeds) zeer verleidelijk omdat zij de meest compromisloze werkwijze toestaat. Een stevige afkeer om de wereld van nòg weer een programmeertaal te voorzien, kan iemand doen uitzien naar alternatieven.

Wanneer een niet al te obscure programmeertaal de derde mogelijkheid toelaat, kan de tweede buiten beschouwing blijven. Voor ALGOL68 is dit min of meer het geval. De implementatie (een "library-prelude" binnen ALGOL68) heet TORRIX = vecTOR/matRIX - of ook: een kleine Galliër die om de dooie dood niet bang is voor de Romeinen.

Historisch is dit relaas ietwat primitief. Een allereerste aanzet was al een uitgewerkt voorbeeld in A68-oud {09}, later aangepast in {10}. Ook nieuwe ideeën werden meestal in ALGOL68 geformuleerd - een zelfstandige programmeertaal werd nooit serieus overwogen. De ontwikkeling van TORRIX was in feite steeds het aftasten van de mogelijkheden van de moedertaal. Hoewel ALGOL68 op veel essentiële punten verrassend bruikbaar is gebleken en - voor ons doel - ook volstrekt superieur aan enige andere ons bekende taal, werd TORRIX beslist geen compromisloze implementatie. Zodat eigenlijk nu pas, nu het project min of meer is afgerond, duidelijk is wat voor taalmiddelen je minimaal nodig hebt om zoiets ten volle te realiseren.

Op enkele van deze zaken zullen we verder ingaan voor zover ze direct betrekking hebben op de TORRIX-datastructuren en de daarop gedefinieerde operaties. De volledige TORRIX-programmatuur, alsmede een uitvoerige theoretische motivering en een gedetailleerde "usersmanual", vindt men in {18}. Een ietwat afwijkende uitwerking (van vroegere datum) is beschreven in {17}.

2. ABSTRACTIESTAPPEN OP EEN KLASSIEKE DATASTRUCTUUR

2.1. Het grondsysteem *scal*

Het scalaire systeem S voor V is, inclusief alle operaties +, −, ×, /, = etc. (voor zover van toepassing voor een specifieke keuze van S), gegeven door de mode *scal*. TORRIX is dus geënt op een mode/operator-pakket overeenkomstig de standard-prelude voor *real* in {22}.

Uiteraard is *real* ook een zeer voor de hand liggende keuze voor *scal*. Men bedenke echter dat *real* een discrete benadering is voor **R**; een nauwkeuriger benadering geeft bijv. *mode scal = long real*. Een voor de numerieke praktijk lang niet onbelangrijk neveneffect van het werken met zo'n niet-gespecificeerde *scal* is dan ook met name de mogelijkheid dezelfde source-text te gebruiken voor verschillende numerieke precisies en/of getalvoorstellingen.

Het is minder aantrekkelijk een vectorruimte V over **C** te definiëren door *mode scal = compl*. Immers, in vrijwel alle praktijk-problemen waarin complexificatie optreedt, worden essentieel reële- naast complexe vectoren en matrices gebruikt. Men moet zijn behoefte aan elegantie ook weer niet zo ver drijven, dat de gebruiker wordt gedwongen een substantieel deel van z'n geheugen ongebruikt te laten omdat het voor ongebruikte imaginaire nullen gereserveerd moest worden. De TORRIX-oplossing is een mode *coscal* naast *scal*.

Uiteraard is

```
mode coscal = struct(scal re,im)
```

zodat *coscal* samenvalt met *L compl* voor de keuze *mode scal = L real*. De ALGOL68 mode-equivalencing doet hier precies wat we graag willen. Het is overigens wel een spijtige zaak dat het ALGOL68 *ref*-concept niet op een iets andere wijze is uitgewerkt. Er is dan namelijk een veel fraaiere oplossing mogelijk voor **C** naast **R**; we komen hierop terug in 4.2.

Zonder noemenswaardige (dwz. anders dan technische) problemen kunnen we ad libitum algebraïsche grondsystemen definiëren, zoals:

```
mode rat    = struct(int  whole,numer,denom)   # subset van Q #
mode quater = struct(real re,i,j,k)    # de quaternionen #
mode modulon = struct(int  rest)       # de restklassen modulo n #
mode quadr  = struct(rat  r,s)         # het kwadratisch lichaam r+s√d #
```

Een voor interval-numerici wellicht interessante *scal* is nog:

mode triple = struct(real lower,middle,upper).

2.2. Toekomstmuziek

TORRIX definieert een mode/operator-pakket door mode-declaraties voor zekere datastructuren (vectoren en matrices) en operator-declaraties op deze modes. De datastructuren zelf liggen hierbij geheel open - dwz. ze zijn onbeschermd toegankelijk voor de gebruiker die er dus ook met zijn eigen gereedschap op kan prutsen. Ook de operatoren kan hij een andere betekenis geven dan de bedoeling was. Nu hoeft men geen compassie te hebben voor de programmeur die zijn eigen speelgoed stuk maakt, maar de argeloze gebruiker verdient wel degelijk bescherming tegen eventuele capriolen van dergelijke artiesten. In ALGOL68 kunnen we niet méér doen dan enkele echt gevaarlijke modes en operatoren (in {18} aangegeven met het taboe-teken "†") "weg te declareren".

In {14} vinden we een overzicht van nieuwe ideeën voor de constructie van mode/operator-pakketten ("classes", "modules", "clusters" etc). Het zal duidelijk zijn dat sommige van de meer geavanceerde benaderingen, hun abstractie-faciliteiten en hun veiligheidsvoorzieningen, bijzonder interessant zijn voor een TORRIX-systeem. We denken daarbij dan niet in eerste instantie aan de taboe-modes en -operatoren, of aan betrekkelijke kleinigheden zoals bij *modulon* en *quadr* (zie 2.1), die beide niet kunnen bestaan zonder een parametrische constante (n resp. $\sqrt{d}$).

Het gaat om groter wild. De *scal* in een TORRIX-systeem fungeert als een parametrische mode: er zijn evenveel TORRIX-systemen als mode/operator-pakketten voor *scal*. Nu moge het voor de praktijk geen onoverkomelijk bezwaar zijn dat voor iedere nieuwe *scal* tenminste één (pre)compilatie nodig is van een over die *scal* gedefinieerde TORRIX. We gaan hier echter enigszins anders tegenaan kijken als we ons realiseren dat het grondsysteem *S* heel best ook zelf weer een vector-module kan zijn, althans in een aantal operaties. Anders gezegd: het kan heel best zijn dat we voor de definitie van een *scal* ook weer een TORRIX-systeem nodig hebben.

Dit is al enigszins het geval voor *mode scal = quater* die zich additief gedraagt als een 4-dimensionale vector. Ook bij de uitbreiding van TORRIX met *scal* (*V* over R) naar TORRIX met *coscal* (*V* over R+C) hebben we te maken met een stuk TORRIX "dat zichzelf herhaalt".

Nog veel duidelijker doet zich de situatie voor als we voor S een polynomenring P nemen. We hebben dan een vector-module V over P, waarin P een vectorruimte is over bijvoorbeeld **Q**. In de ALGOL68-implementatie van TORRIX lukt dit best: men compileert eerst P over **Q** (met *mode scal = rat*) en vervolgens V over P (met *mode scal = poly*, na een overgangsdeclaratie *mode poly = vec*).

We willen natuurlijk iets dat minder primitief, en tevens veel economischer is in gegenereerde code. Dat kan dan weinig anders zijn dan een mode/operator-pakket dat op een of andere manier "over de mode in recursie" kan gaan.

In ons voorbeeld: bij de optelling van vectoren in V_n (over P) worden n scalaire optellingen gedaan. Dat zijn dan n optellingen van polynomen in P_m (over **Q**), die ieder voor zich weer m scalaire optellingen doen in **Q**. Op beide nivo's wordt - in termen van *scal* - echter precies dezelfde operatie gedaan (nl. een vector-optelling). De ALGOL68-code is in beide nivo's identiek, zodat we beter over twee "incarnaties" van dezelfde routine kunnen spreken.

"Cluster-recursie" (een mode/operator-pakket gaat over de mode in recursie) is, voor zover ons bekend, nog niet als zodanig onderzocht. Een van de problemen is dat niet alle operaties over *scal* code-invariant zullen zijn. Zo is het product van twee polynomen een convolutieproduct dat een polynoom teruggeeft, terwijl een product van vectoren in V (over P) iets heel anders kan zijn (inclusief ongedefinieerd).

De conceptuele organisatie van de moderne algebra (vooral zichtbaar in de categorieën-theorie) is een sterke indicatie dat recursieve mode/operator-pakketten krachtige instrumenten kunnen zijn voor abstracte type-manipulatie. Ook los van de strikt wiskundige context kan het idee van dergelijke pakketten vruchtbaar blijken - we zouden daarbij kunnen denken aan de algebraïsche (relationele) modellen voor databases.

2.3. Het nulnivo

Uit de isomorfiestelling $V_n \cong S^n$ volgt nog niet eenduidig de algoritmische representatie van de objecten in V. Er zijn tenminste twee mogelijkheden: enkel- en dubbel-geïndiceerde [1] arrays, of procedures met één of twee

---

[1] Het foutieve taalgebruik "1- en 2-dimensionale arrays" is in de context van lineaire algebra uiterst misleidend.

integer parameters. De array-representatie is onontkoombaar in alle gevallen waarin de primaire *scal*s afkomstig zijn van input (in verreweg de meeste gevallen dus). De procedure-representatie kan meer voor de hand liggen wanneer het object door een gesloten uitdrukking is gedefinieerd (nulvector, eenheidsmatrix, Hilbertmatrix etc). In principe zijn ook mengvormen mogelijk. We zouden dus kunnen komen tot de tamelijk algemene:

```
mode object1 = union([]scal, proc(int)scal);
mode object2 = union([,]scal, proc(int,int)scal,
                     [][]scal, proc(int)proc(int)scal,
                     []proc(int)scal, proc(int)[]scal
                    )
```

Tegen deze aanpak kan een technisch en een conceptueel bezwaar worden aangetekend. Het technische bezwaar is, dat men (althans in ALGOL68) een *union* niet kan "slicen" noch parametrizeren. Men kan natuurlijk op *object1* en *object2* selectors definiëren (operatoren dus), maar de overhead is dan aanzienlijk. Het conceptuele bezwaar is dat men een grotere algemeenheid (bovendien met veel minder overhead) kan verkrijgen op een hiërarchisch hoger definitie-nivo.

Het definitie-nulnivo voor TORRIX is dan ook heel simpel:

een mode *array1* voor []*scal*
een mode *array2* voor [,]*scal*
} in de beschrijving samengevat onder de naam "*array*".

Niet simpel is uiteraard dat *scal* een willekeurige S kan zijn.

2.4. Het operatie-nivo

Iedere *array* realiseert een functie $[1:n]\rightarrow S$ resp. $[1:m,1:n]\rightarrow S$ en representeert als zodanig een object in V wanneer we de operatoren kiezen overeenkomstig de rekenregels 1 t/m 4 van 1.4. Impliciet daarin is, dat de domeinen [1:n] resp. [1:m,1:n] "compatibel" zijn - dwz. in de additieve operaties moeten ze gelijk zijn, terwijl in de multiplicatieve operaties het aantal "kolommen" links overeen moet komen met het aantal "rijen" of elementen rechts. Deze eis brengt in feite niets anders tot uitdrukking dan dat de operanden in passende ruimten V_n en V_m moeten liggen.

De in 1.5 geformuleerde criteria betekenen dat we eigenlijk wel van deze beperkingen af willen. Immers, II stelt dat we V los van de *array*-representatie willen kunnen zien, en III verlangt dat we objecten uit verschillende

deelruimten willen kunnen combineren. Om dit goed te doen zullen we de datastructuren voor vectoren en matrices met enige behoedzaamheid moeten kiezen. De uiteindelijke structuur hangt namelijk sterk af van de toepassing. We kunnen hierbij denken aan de vele manieren waarop een V-object "dun-bezet" kan zijn: nulvector en -matrix, eenheids-vector, driehoeksmatrix, bandmatrix etc. De *array* speelt dan vaak geen andere rol dan die van subobject. Teneinde de *array* in deze rol op technisch eenvoudige wijze te kunnen hanteren, kiezen we voor het operatie-nivo niet de *array* zelf, maar een object dat ernaar verwijst:

mode vec = ref array1 of ook: *mode vec = ref ()scal*
mode mat = ref array2 of ook: *mode mat = ref(,)scal*

Door de operatie-keuze zullen *vec* en *mat* zich in alle opzichten gedragen als "vector" en "matrix", hoewel ze niet méér zijn dan de simpelste basisstructuren daarvoor. Uiteindelijke modes *vector* en *matrix* (en meer specifiek: *band*, *triang*, *sparse*, *linsys* etc) kunnen, op soms gecompliceerde wijze, uit *vec*s en *mat*s zijn samengesteld.

2.5. Equivalentieklassen van *arrays*

Teneinde *vec*s en *mat*s uit willekeurige (deel)ruimten met elkaar te kunnen combineren, moeten we voor hun *array*s een equivalentierelatie definiëren. Voor de hand ligt, dit als volgt te doen:

$$\left.\begin{matrix} u = v \\ A = B \end{matrix}\right\} \Leftrightarrow_{\text{def}} \left\{ \begin{matrix} \text{alle elementen in de doorsnee van de domeinen} \\ \text{zijn gelijk; alle elementen daarbuiten zijn nul.} \end{matrix}\right.$$

Door nu *vec* en *mat* te interpreteren als equivalentieklassen van *array*s (i.p.v. als één specifieke *array*) en de rekenregels 1 t/m 4 overeenkomstig uit te breiden, bereiken we zo'n beetje alles wat we op het eenvoudigste operatie-nivo willen.

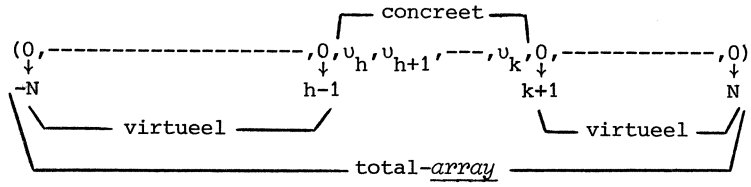
We denken ons iedere concrete *array* [1:n]→S of [1:m,1:n]→S als te zijn ingebed in een "total-*array*" [-N:N]→S resp. [-N:N,-N:N]→S, waarin N de maximaal-toelaatbare *array*-subscript is. Uiteraard kan zo'n total-*array* nooit in het geheugen bestaan (N kan bijv. 2^{15}-1, of 2^{18}-1 zijn). Trouwens, alle elementen buiten het concrete *array*-domein [1:n] resp. [1:m,1:n] zijn per definitie nul, en het laatste wat je daarmee moet doen, is ze in het geheugen opslaan. Het idee van de total-*array* werkt natuurlijk de andere kant op: in principe kiezen we voor elke *vec* of *mat* de meest economische repre-

sentant van z'n equivalentieklasse - en dat is de kortste. Voor de nulvector en nulmatrix zijn dit lege *arrays* (!), voor een eenheidsvector een *array1* met precies één element (=1), etc.

We noemen de nullen buiten de concrete *array*: virtuele nullen. De uitbreiding van de rekenregels 1 t/m 4 komt er eenvoudig op neer dat we alle virtuele nullen ahw. laten meedoen (algoritmisch wordt hier uiteraard heel wat onder tafel gewerkt). Aldus ontstaat een praktisch zeer bruikbaar ruimtemodel T: een "bijna-oneindig"-dimensionale vectorruimte waarin iedere V als deelruimte is vervat:

$$V \subset T \qquad \text{en} \qquad \dim T = 2N + 1$$

De uitbreiding van V tot T impliceert ook dat de concrete *array*-grenzen geheel vrij worden: $(h{:}k)$*scal* definieert evenzeer een vector in T als bijv. $(1{:}n)$*scal*:



Ook de slice $u(p{:}q$ *at* $p)$ van een gegeven *array* (bijv. concreet voor $1{:}n$ met $1 \leq p$ en $q \leq n$) krijgt een natuurlijke interpretatie: het is de projectie van u op de deelruimte in T opgespannen door de eenheidsvectoren $e_p, \ldots, e_q$. Dit blijft gelden voor $p=q$ en zelfs $p<q$ (nulvector).

We noemen -N=*mindex* en N=*maxdex*. De basis-definities van het systeem T zijn nu:

```
mode array1 = (mindex:maxdex)scal;
mode array2 = (mindex:maxdex,mindex:maxdex)scal;
mode vec    =  ref array1;
mode mat    =  ref array2;
```

2.6. Het torrix-nivo

De *ref* voor de *array* brengt tot uitdrukking dat *vec* en *mat* equivalentieklassen van *arrays* zijn en het ligt voor de hand dat we concrete *arrays* van verschillende omvang willen kunnen assigneren aan een *vec*- resp. *mat*-variabele. Het is opmerkelijk dat dit zonder meer kan:

de mode van een *vec*-variabele is *ref ref array1*
de mode van een *mat*-variabele is *ref ref array2*

Bij een toekenning wordt dan een *ref array* geassigneerd. Achter een *ref* wordt een mode in ALGOL68 "formal", dwz. *array*-grenzen spelen daar geen rol meer: een *ref array* is een *ref*[]*scal* resp. *ref*[,]*scal* en iedere concrete *array* is hier acceptabel. Dit is ook precies wat we willen hebben: een *vec* of *mat* vertegenwoordigt een hele equivalentie-klasse en niet één bepaalde *array*. Zoals "*array*" staat voor "[]*scal*" resp. "[,]*scal*", zo schrijven we vaak "*torrix*" voor "*ref array*" (dwz. voor "*vec*" resp. "*mat*"). Evenmin als "*array*" is "*torrix*" echter een union - strikt genomen is het ook geen mode-indicatie, maar slechts een woord in dit verhaal.

Bij toekenning van een *torrix* aan een *torrix*-variabele wordt de *torrix* (rechts van :=) gecopieerd naar de *ref torrix*-destination (links van :=). Nu is de copie van een *torrix* de copie van een *ref* en niet van de onderliggende *array*. Ook dit is een veelal zeer gewenste gang van zaken - waar mogelijk willen we het maken van copieën natuurlijk vermijden. Anderzijds willen we wèl in staat zijn om een slice van een *array* (kolom, rij, diagonaal, projectie) apart te benoemen: met name subscript-reductie van *ref array2* naar *ref array1* kan belangrijke tijd winnen. We hebben echter wel behoefte aan een operator *copy* waarmee eventueel een copie kan worden afgedwongen (zie verder 3).

We onderscheiden aldus:

het nul-nivo:	concrete *array*s,
het operatie-nivo:	*vec* resp. *mat* (samengevat als *torrix*) verwijzend naar total-*array*s,
het torrix-nivo:	*torrix*-variabelen (*ref vec* resp. *ref mat*) het nivo waarop total-*array*s (concrete *array*s van verschillende omvang) kunnen worden gemanipuleerd.

We merken en passant op, dat we voor het werken met "flexibele rijen" het ALGOL68-"*flex*"concept klaarblijkelijk helemaal niet nodig hebben en eigenlijk ook niet kunnen gebruiken (vanwege de operationele restricties voor "transient names"). Dit doet vermoeden dat "*flex*" een niet alleen ongelukkig, maar ook overbodig concept is (voor verdere details zie {16}).

Voor het torrix-nivo is essentieel een garbage-collector nodig. Het operatienivo gebruikt de *heap*-generator alleen om syntactische redenen - de heap wordt er echter zuiver als stack bespeeld. De grens tussen meer en minder eenvoudig (zie pag. 104) ligt dus precies bij het torrix-nivo.

3. GENEREREN EN OPEREREN

3.1. Operaties in *T*

We willen dat operaties een datastructuur karakteriseren en niet omgekeerd. Zodra echter zo'n structuur vastligt - dwz. een implementatiemodel is gekozen zoals *T* voor *V* - kan een wisselwerking optreden. Hier moeten we op onze hoede zijn omdat extrinsieke operaties een systeem aardig kunnen vertroebelen. Zo is ordening een nogal voor de hand liggende operatie op een concrete *array*, maar niet op een vector in *V* - ook niet als *S* een ordening toestaat. Coördinaat-ordening is namelijk helemaal geen *V*-operatie. Wanneer we hem niettemin in *T* zouden opnemen, verzieken we het idee van de total-*array* volledig.

Toch zijn er heel wat zinnige (zelfs onmisbare) *T*-operaties die in het mathematisch epos over *V* ontbreken zonder er echter mee in strijd te zijn. We denken hierbij aan het genereren van een concrete-*array*, het assigneren van een *T*-object aan een *T*-variabele, het laten verdwijnen, overschrijven of copiëren van informatie en voorts het afvragen van allerlei condities als onderdeel van diverse algoritmische stuurconstructies.

In feite maken we hier dus een essentieel onderscheid tussen een mathematisch systeem zoals een vectorruimte *V* en zijn implementatie in een algoritmische discipline *T*. In dit onderscheid blijft gelden dat $V \subset T$, maar dan in de striktere betekenis dat alle objecten in *V* realiseerbaar zijn in *T* en dat alle operaties van *V* in *T* geïmplementeerd (of implementeerbaar) zijn, maar ook omgekeerd dat *T* geen objecten of operaties bevat die geen zinnige interpretatie hebben in *V*. Uiteraard was dit een tamelijk informele babbel over een relatie die best eens verder onderzocht zou mogen worden.

Een gedetailleerde en volledige behandeling van alle *T*-objecten en hun operaties vindt men in {18}. We volstaan hier met een globaal overzicht en een wat diepergaande discussie van slechts enkele meer principiële zaken.

3.2. Genereren

Een concrete *array* wordt gegenereerd via een *proc(int)vec* respectievelijk *proc(int,int)mat*. Bijvoorbeeld *genvec(n)* genereert een *[1:n]scal* en retourneert de *vec*. Evenzo genereert *genmat(m,n)* een *[1:m,1:n]scal* resulterend in een *mat*. Een *[h:k]scal* kunnen we maken door de constructie *genvec(k-h+1)[at h]*, een vierkante matrix *[1:n,1:n]scal* ook door *gensquare(n)*.

De *gen*-procedures reserveren wel ruimte voor *arrays*, maar initialiseren deze niet. We hadden ook direct kunnen genereren bijv. door *heap[1:n]scal* resp. *heap[1:m,1:n]scal*. In de *gen*-procedures zijn echter controles en extra-voorzieningen ingebouwd, zodat hun gebruik veiliger is. Voor de meeste toepassingen is het namelijk aan te bevelen het gigantische (2N+1 dimensionale) *T*-bereik voor het genereren van *arrays* wat in te perken, bijvoorbeeld tot het domein [1:n,1:n]. Dit kan door de aanroep *setgendex(1,n)*.

Een belangrijke vorm van genereren is copiëren. Zo copieert *copy u* de gegeven vector *u* en *copy a* de gegeven matrix *a*. Een copie van de j^{e} kolom van *a* wordt verkregen door *copy a[,j]*. Voor TORRIX-intern hebben we de dyadische operator *span* die een concrete *array* genereert waar de concrete *arrays* van z'n beide operanden in passen: zowel *array2(a)* als *array2(b)* passen in *array2(a span b)*.

We zullen verderop (zie 3.8) een onderscheid maken tussen niet-genererende operaties, die geen enkele aanroep van een *gen*-procedure bevatten, en genererende operaties, die er tenminste één plegen. Er zijn ook operaties die, afhankelijk van de situatie, genererend of niet-genererend kunnen zijn.

3.3. Toeschrijven en toekennen

In een "ascription" wordt een object gehecht aan een identifier, in een "assignation" wordt het gecopieerd naar een locatie (die bijv. via een *ref* aan een identifier kan hangen). Typische TORRIX-toepassingen van "ascription" zijn:

```
vec vecu = genvec(n); vec vecv = genvec(m);
mat mata = genmat(m,n); mat matb = gensquare(n);
vec mataj = mata[ ,j]; vec copymataj = copy mata[ ,j];
```

Al deze operaties (formeel zijn het declaraties) definiëren een identifier op operatie-nivo; ze zijn op één na genererend. Alleen door de declaratie *vec mataj = mata[,j]* wordt een reeds bestaand object (de j^{e} kolom van *mata*, een *vec* dus) aan "*mataj*" gehecht. De hele geschiedenis speelt zich af op operatie-nivo: *vecu*, *vecv*, *mata*, *matb*, *mataj* en *copymataj* zijn allen *ref array* identifiers.

Aan identifiers op operatie-nivo kunnen concrete *arrays* worden geassigneerd. Hier gelden dus strikte compatibiliteits-eisen: alle grenzen links en rechts moeten kloppen: *vecu:=mata[,j]* kan, maar *vecu:=vecv* kan alleen

als *m=n*. Merk op dat *mataj:=vecu* in principe een nieuwe *array* assigneert aan de j^{e} kolom van *mata* (deze was immers door ascription gehecht aan *mataj*).

Een totaal andere situatie treffen we aan op torrix-nivo. Door declaraties van *torrix*-variabelen zoals:

loc[1] *vec u,v,w;*
loc mat a,b,c;

worden geen concrete *array*s gegenereerd, maar slechts nieuwe *ref array*s. Een belangrijk gevolg is, dat men aan *u, v, w,-- a, b, c,--* iedere *torrix* (*ref array*) kan assigneren, ongeacht de omvang van z'n concrete *array*. Deze wordt immers niet gecopieerd - de toekenning vindt plaats op het bovenste *ref*-nivo en dat is *ref torrix*:

loc vec u:=genvec(n); loc vec v:=genvec(m); loc vec w;
loc mat a:=genmat(m,n); loc mat b:=gensquare(n); loc mat c;

Na deze declaraties is *u:=v* evenzeer mogelijk als bijv. *v:=a[,j]*. Wel moet men zich er goed van bewust zijn dat in geen van deze assignaties een concrete *array* wordt gecopieerd en dat men na *v:=a[,j]* door assignaties zoals *v[i]:=s* en ook *v[]:=u* iets verandert in de j^{e} kolom van *a*. Wil men dit niet, dan had men moeten assigneren *v:=copy a[,j]*. Merk op, dat *v[]:=u* weer op operatie-nivo assigneert en hetzelfde effect heeft als *vec(v):=u*.

Op het torrix-nivo veronderstellen we uiteraard een goed-functionerende garbage-collector. Zolang men - al toeschrijvend en toekennend - niet uitgaat boven het operatie-nivo, kan men eventueel zonder (zie pag. 104).

Waar de ALGOL68 assignatie faalt, heeft TORRIX nog de toekenningsoperator *into*; bijvoorbeeld:

0 into vecu; 1 into u; 0 into a[i,];
mat david = hilbert into gensquare(n);

Hier is *hilbert* een *proc(int,int)scal* die de Hilbert-matrix levert (zie 3.7).

Een zeer nuttige operator is nog *=:=* die twee concrete *array*s verwisselt. Ook hier geldt de eis dat de concrete grenzen overeenstemmen. Door *a[i,]=:= a[h,]* worden de i^{e} en h^{e} rij van *a* verwisseld. Een operatie *=:=*, evenals *:=* gedefinieerd voor alle *ref amode*, missen we node in ALGOL68 {16}.

---

1) Hoewel optional in ALGOL 68, verdient het aanbeveling "*loc*" altijd te schrijven - zeker in een context waarin twee verschillende *ref*-nivo's naast elkaar worden gebruikt.

3.4. Nul en een

Standaard in TORRIX zijn:

```
vec zerovec = heap[maxdex:mindex]scal;
mat zeromat = heap[maxdex:mindex,
                   maxdex:mindex]scal;
```

Een "ultra-flat" descriptor met grenzen *maxdex:mindex* genereert uiteraard een lege concrete *array* (in optima-forma dus een nulvector o resp. nulmatrix O). De grenzen zijn zo gekozen teneinde de reken-operaties in T optimaal te laten verlopen. Het is een plezierige omstandigheid dat, hoewel *zerovec* en *zeromat* syntactisch links van := mogen voorkomen, er niets anders aan geassigneerd kan worden dan een lege *array* met ultra-flat descriptor.

Minder veilig (en daarom ook niet standaard in TORRIX) is de als volgt gedeclareerde eenheidsvector:

```
vec e = 1 into genvec(1)    # dit is e1 #
```

Grappig is dat deze ene declaratie goed is voor alle 2N+1 eenheidsvectoren in T: *e(at k)* produceert e_k. Helaas is *e* niet bestand tegen sabotage van het soort *e(1):=0*.

3.5. Testen en trimmen

Er zijn een aantal tests (ondervragings-operaties) die min of meer betrekking hebben op de concrete *array* van een gegeven *torrix*. Sommigen accentueren de betekenis van het total-*array* concept. De monadische en dyadische *lwb*, *upb*, *size* en *square* doen precies wat ze suggereren; *size u* retourneert de dimensie van de kleinste ruimte V op een basis e_p, e_{p+1},---, e_q waarin u ligt, *square a* vraagt of de matrix *a* vierkant is. De operatie *u fitsin v* test of het concrete domein van *u* past in dat van *v*; idem *a fitsin b*.

Interessanter zijn *zero* en =. De test *zero u* retourneert dan en slechts dan *true* als *u is zerovec*; idem voor *zero a*. Dit impliceert dat bijv. *zero (0 into vecu)* de waarde *false* retourneert, ook al bestaat *vecu* nu uit louter (n) nullen. Of u=o, moet men dat ook vragen door *u=zerovec*. De achtergrond zal duidelijk zijn: = test of de total-*array*s links en rechts gelijk zijn, dwz. of de vergeleken *vec*s equivalent zijn volgens de definitie van 2.5. De *zero*-test informeert naar een heel specifieke concrete *array*.

Met = hangt de trim-operatie samen. Wanneer een concrete *array* "aan de buitenkant" (links en/of rechts) nullen heeft, kan deze worden ingekort zonder dat de *vec* verandert. Dit is precies wat *trim* *u* doet. Merk op dat, na *u:= copy v* de test *u=trim v* altijd *true* zal zijn, zelfs als de concrete *array* van *v* werd gereduceerd tot die van *zerovec*. Het spreekt vanzelf dat *trim* alleen werkt op torrix-nivo.

De operatoren *sigma*, *sigmabs*, *max*, *maxabs*, *min* en *minabs* retourneren resp. de som, het maximum of het minimum van de (absolute waarden van de) elementen van de total-*array*s. In het bijzonder is dus *sigma zerovec* = 0 en *sigma e* = 1. Ietwat onverwacht misschien is *minabs u* = 0 voor alle waarden van *u*. De operatoren *max*, *maxabs*, *min* en *minabs* zijn ook dyadisch gedefinieerd, maar beperken zich dan tot de concrete *array*s. Zo is *k minabs u* het minimum van de absolute waarden van de elementen van de concrete *array* van *u*; aan *k* wordt de (kleinste) subscript van dit minimum geassigneerd.

3.6. Selectors

Op concrete *array*s (en hun *ref*s) staat het hele arsenaal van ALGOL68-slicers ter beschikking: "[*i*]", "[*i*,*j*]", "[*i*,]", "[,*j*]", "[*h*:*k* *at* *h*]" etc. Ze worden echter undefined zodra een subscript buiten de concrete grenzen komt. Zolang we alleen met *vec*s en *mat*s werken levert dit nog geen probleem - met *lwb* en *upb* blijven we wel binnen de perken. Dit kan echter drastisch veranderen zodra *vec*s en *mat*s sub-objecten worden van gecompliceerdere datastructuren.

Wanneer we willen dat het total-*array* concept volledig werkt en een nul retourneert (ipv. een foutmelding) zodra we buiten de concrete grenzen selecteren, moeten we selectie-operatoren definiëren. Een alternatieve notatie voor "u_i" is "u.i" of ook "i.u". Omdat de punt niet als operator-symbol beschikbaar is, behelpen we ons met *?* als selector. We hebben nu *u?i* en *i?u* als total-*array* extensie van *u*[*i*]. Voorts zijn *a?i*, *j?a* en *j?a?i* de extensies van resp. *a*[*i*,], *a*[,*j*] en *a*[*i*,*j*]. We mogen voor *j?a?i* ook schrijven *(i?j)?a* of *a?(i?j)*. We gebruiken verder de operator *//* voor projectie: *u?(h//k)* en *(h//k)?u* zijn de total-*array* extensies voor *u*[*h*:*k* *at* *h*]. Een vorm van *?* waaraan ook kan worden geassigneerd is, vanwege de moeilijk in de hand te houden zij-effecten, niet goed realiseerbaar in ALGOL68. De selectors *?* en *//* zijn geïmplementeerd, voornamelijk met het oog op latere (op *vec* en *mat* gebaseerde) datastructuren. Zie ook 4.3 en 4.4.

De operatoren *trnsp* en *diag* doen precies wat we er redelijkerwijs van mogen verwachten: *(trnsp a)[i,j] is a[j,i]* en *(diag a)[i] is a[i,i]*; en ook *(k diag a)[i] is a[i,i+k]*. De operaties *col* en *row* retourneren een *mat* (met één kolom resp. rij) zodat *(k col u)[i,k] is u[i]* en *(h row u)[h,i] is u[i]*. De operatoren *trnsp*, *diag*, *col* en *row* doen niets anders dan de *array* van hun operand voorzien van een nieuwe descriptor - er worden geen elementen gecopieerd. Een eenheidsmatrix kunnen we nu genereren door:

```
mat identy = 0 into gensquare(n); 1 into diag identy;
```

Natuurlijk wordt *(k diag a) is zerovec* zodra *k* buiten de concrete *array* selecteert.

Hoewel we er ons redelijk uit hebben kunnen redden, schiet ALGOL68 het duidelijkst tekort op het punt van total-*array* selectie. Zo ergens, dan voelt men hier de behoefte aan beter gereedschap (zie ook {14}).

3.7. Punt-operaties

Veel reken-operaties in *V* zijn "punt-operaties", dwz: op alle (overeenkomstige) elementen van de operand(en) wordt dezelfde operatie gedaan en deze operaties zijn onafhankelijk van elkaar. Er is dus geen enkele reden om ze in een bepaalde volgorde te doen en daardoor kan er (op machinenivo) veel (soms zeer veel) aan worden geoptimaliseerd. Als er n of n^2 processors beschikbaar zijn, mogen ze ook allemaal tegelijk. In ALGOL68 moeten we een Hilbert-matrix als volgt definiëren:

```
proc hilbert = (int n)mat:
     (mat dave = gensquare(n);
      for i to n
       do for j to n
            do dave[i,j]:=1/(i+j) od
       od; dave
     )
```

Wat we natuurlijk nodig hebben is een collateral-loop-clause, die geen volgorde-eisen stelt aan de elaboraties van z'n do-part; bijvoorbeeld:

```
proc hilbert = (int n)mat:
     with i,j thru mat dave=gensquare(n); dave
       do dave[i,j]:=1/(i+j) od
```

Het is eenvoudig te realiseren en zinvol om een dergelijke constructie, zoals hierboven is gedaan, als waarde de yield van zijn REFETY-ROWS-thru-part te laten retourneren. Voor verdere discussie zie {16}.

3.8. Genererende- en assignerende operaties

In wiskundige notatie betekent "a□b" het door de operatie □: $A \times B \rightarrow C$ aan $a \in A$ en $b \in B$ toegevoegde element $c \in C$. Van enig "zij-effect" op a of b is hierbij geen sprake: "a□b" definieert een (nieuw) element in C. Vertaald naar T betekent dit een genererende operatie: a en b blijven wat ze zijn, "a□b" genereert een nieuw object c. Wanneer nu C=*scal* behoeven we ons nergens nerveus over te maken, maar de behuizing van de concrete *array* van een *vec* en zeker van een *mat* kan in omvangrijke stelsels wel degelijk zorgen baren.

Met x, y en z duiden we hier en verderop een (*ref*) *torrix* aan en met s een (*ref*) *scal*. Een veel gewenste operatie zoals $y{:=}y{+}x$ is in geheugengebruik onzinnig oneconomisch wanneer x *fitsin* y. Immers, het is klaarblijkelijk de bedoeling dat x bij y wordt opgeteld en daarvoor hoef je niet eerst een nieuwe *torrix* te genereren als de concrete *array* van y groot genoeg is voor die van x.

Voor een operatie $y{:=}y\square x$, met als voorwaarde x *fitsin* y, schrijven we $y\square{<}x$; de "<" geeft de richting van de assignatie aan. Bij sommige operaties speelt de voorwaarde geen rol, zoals bij $y{:=}y \times s$ resp. $y{:=}s \times y$ waarvoor we altijd kunnen schrijven $y \times {<} s$ resp. $s \times {>} y$, ongeacht de omvang van de concrete *array* van y.

De "□<"operaties spelen zich af op operatie-nivo. Op torrix-nivo kan de voorwaarde x *fitsin* y een interessantere rol vervullen. De assignatie $y{:=}y\square x$ kan, ongeacht de omvang van $y\square x$, immers altijd zodra y een *ref torrix* is. Na de assignatie is de oorspronkelijke concrete *array* van y voer voor de garbage collector: y verwijst nu naar de nieuw gegenereerde $y\square x$. Dit genereren willen we echter wèl voorkomen als x *fitsin* y. Voor deze samengestelde wens schrijven we $y\square{:=}x$. Deze operatie doet dus $y\square{<}x$ zolang x *fitsin* y en schakelt over op $y{:=}y\square x$ zodra de voorwaarde niet meer vervuld is.

3.9. Additieve operaties

Voor het gemak schrijven we "±" voor "+" of "−". We nemen "±<" als additieve basis-operatie:

```
op ±< = (mat a,b)mat:
          if b fitsin a
        then with i,j thru b
               do a[i,j]±:=b[i,j] od; a
        else torrixerror("bah"); skip
          fi
```

Voor "±" gebruiken we x *inspan* y die de concrete *array* van x assigneert in x *span* y (zie pag. 113), buiten het domein van x aangevuld met concrete nullen. We definiëren nu "±" en "±:=" als volgt:

```
op ±  = (mat a,b)mat: (a inspan b)±<b;
op ±:= = (ref mat a, mat b)ref mat:
          if b fitsin a
        then a±<b; a
        else a:=a±b
          fi
```

Toepassingen zijn:

w:=u+v; c:=a-b	doen wat ze pretenderen zonder enige voorwaarde of restrictie op concrete grenzen;
a[i,] -< sxa[h,]	trekt sx de h$^{e}_{-}$ rij van a af van de i$^{e}_{-}$ rij;
diag a -< vecu	trekt *vecu* af van de hoofddiagonaal van a;
vec uv = u+v[at n+1]	is de concatenatie van u en v;
u+:=v[at n+1]	plakt de concrete *array* van v achter die van u;
a+:=(n+1)col v	breidt a uit met een copie van v als (n+1)$^{e}_{-}$ kolom;
etc.	

3.10. Multiplicatieve operaties

De eenvoudigste multiplicatieve operaties zijn sxx en xxs die natuurlijk op hetzelfde uitkomen als S commutatief is. De assignerende versies zijn sx>x en xx<s.

Interessanter zijn de "somproducten" Vx$V\rightarrow S$: scalaire accumulaties van producten van *array*-elementen zoals het inproduct <u,v>, in TORRIX-notatie u<>v, en het convolutieproduct u><v. In $u<>v=\sum v_i \phi_i$ lopen de indices dezelfde kant op, in $u><v$ lopen ze "tegen elkaar in". Bij de compositionele substitutie van polynomen - (u∘v)(s)=u(v(s)) waarin u en v polynomen voorstellen - wordt u∘v berekend met behulp van convolutieproducten.

Voor een lineaire transformatie Au en een matrix-product AxB worden somproducten *(a?i)xu* en *(a?i)x(j?a)* gebruikt die, zolang *S* niet complex is, identiek zijn met "<>". Het Hornerproduct voor u(s) - u weer opgevat als een polynoom - kunnen we ook beschouwen als een somproduct *VxS→S*, notatie *uos*.

Over de producten *axu* (lineaire transformatie) en *axb* (matrix product) valt na het voorafgaande weinig meer te zeggen; ze implementeren de in 1.4 gegeven rekenregels, maar zijn gedefinieerd voor alle concrete *arrays* (hoe gek ook gekozen). Naast *axu* (matrix maal kolom levert kolom), hebben we ook *vxa* (rij maal matrix levert rij). Merk op dat *col vecu* x *row vecv* een matrix levert waarvan het (i,j)^{e} element *(vecu?i)x(vecv?j)* is.

Bijzondere vermenigvuldigingsoperaties zijn nog *uxxv*: het zg. Cauchy-product van polynomen, met de definiërende eigenschap dat *(uxxv)os = (uos)x(vos)* en de polynomiale compositie *uov*, gedefinieerd door *(uov)os = uo(vos)*.

3.11. Een voorbeeld

Onderstaande *proc(mat)vec cholesky* ontbindt C in LXL^{T}, waarin L een beneden-driehoek is. C moet symmetrisch zijn en z'n boven-driehoek (incl. diagonaal) verandert niet. De result-*vec* p[1:n] bevat de reciproke diagonaalelementen van L. Als p[0]=det(C)=0, was C singulier en *upb* p vertelt hoever de ontbinding ging. *S* is R, *sqrt s* is gedefinieerd; *eps*, *zero* en *one* zijn globale constanten.

```
proc cholesky = (mat c)vec:
  if square c and not zero c
then mat a=c[at 1,at 1]; loc int n:=upb(a);
    scal norm=maxabs(diag a)xeps;
     vec p=genvec(n+1)[at 0]; ref scal det=p[0]:=one;
     for i to n while vec ai=a[i, ];
                    for j to i-1
                     do (ai[j]-:=ai<>a[j,:j-1])x:=p[j] od;
                   scal lii=ai[i]-ai<>ai[:i-1];
                   (detx:=if lii > norm then lii else zero fi)>zero
      do n:=i; p[i):=1/sqrt(lii) od;
      p[:n at lwb(c)-1]     # bounds corresponding to given c #
else zerovec
  fi
```

4. NIEUWE MOGELIJKHEDEN

4.1. *vec* en *mat* als primitiva

Het in 2 en 3 geschetste systeem biedt met *scal*, *vec* en *mat* een volledige implementatie van *V* over *S*. Wanneer de kritieke operaties (bijv. door handcodering op machinenivo) worden geoptimaliseerd, zal TORRIX in principe kunnen concurreren met de meest efficiënte traditionelere systemen. Het idee van de equivalentieklassen van *array*s (de total-*array*) staat een efficiënte implementatie op geen enkele wijze in de weg, de behandeling van de nul-vector en -matrix helpt optimalisering. Belangrijker is dat het de weg opent naar een systematische behandeling van ruimte-economische problemen die zich met name voordoen bij dunbezette stelsels ("sparse matrices").

Doordat we objecten uit verschillende (deel)ruimten vrij met elkaar kunnen combineren (er zijn geen compatibiliteitseisen), kunnen we *vec* en *mat* zonder voorzorgen - zonder zorgen zelfs - als primitiva nemen voor complexere datastructuren. Een belangrijk punt hierin is, dat een willekeurige verzameling (rij, keten, boom) van *vec*s resp. *mat*s zich vanzelf weer zal gedragen als een vector resp. matrix in *V* zolang we ons bij de operaties op dergelijke structuren maar laten leiden door de lineariteit van *V*.

Zij $A=\{A_1,---,A_p\}$ en $u=\{u_1,----,u_q\}$ - willekeurig gestructureerde verzamelingen - dan is bijvoorbeeld $Au=\{A_1u_1,A_1u_2,---,A_1u_q,A_2u_1,------,A_pu_{q-1},A_pu_q\}$ een lineaire transformatie. Bij de opbouw van **A**, **u** en **Au** in een keten-structuur, weren we uiteraard *zerovec* en *zeromat* (met de *zero*-operator) en hanteren we eventueel *trim*. Merk op, dat overlappende domeinen alleen vervelend zijn voor de selectors *?* en *//*, maar verder op geen enkele wijze roet in het eten gooien. In een keten-structuur kan een sommering $A+A_{p+1}$ heel simpel $\{A_1,---,A_p,A_{p+1}\}$ zijn, en een sommering **A+B** de samenvoeging van de ketens.

Dit is duidelijk een veel te algemene benadering van het "sparse object" probleem en alleen van principieel belang omdat eruit blijkt hoe belangrijk het is dat we de *V*-operaties hebben gedefinieerd voor alle mogelijke *array*-grenzen en hoe essentieel onze keuze van de nul-objecten was. Bovendien blijkt er ook uit hoe vrij we zijn in de keuze van de verzamelingsstructuur.

In de praktijk zijn dunbezette vectoren veel minder belangrijk dan dunbezette matrices. In het bijzonder de driehoeks-, (scheef)symmetrische- en bandmatrix treden veelvuldig op in de numerieke wiskunde. In het volgende laten we, zeer summier, zien hoe je met *vec*s en *mat*s als primitiva kunt opereren in samengestelde structuren.

4.2. Een gemiste kans

De meest voor de hand liggende definitie van een complexe *vec* en *mat* is ongetwijfeld:

```
mode covec = struct(vec re,im);
mode comat = struct(mat re,im);
```

Na *loc covec cu* hebben we dan met *re of cu := u* en *im of cu := zerovec* een keurige (nog) reële *covec* in handen. De beide concrete *arrays* van zo'n *covec* mogen immers niet alleen op verschillende plaatsen in het geheugen liggen, maar kunnen ook ongelijk zijn van lengte. Na *re of cu := genvec(p); im of cu := genvec(q)(at p+k)* is *cu* gedeeltelijk reëel, gedeeltelijk nul en gedeeltelijk zuiver imaginair. Alle gewenste operaties op zo'n *covec* laten zich zonder problemen definiëren, maar ze zijn helaas onverenigbaar met de ALGOL68-definitie van *compl*:

```
mode compl = struct(real re,im)
```

die immers de geheugen-locatie van de *re* en de *im* in een *compl* niet vrijlaat. En dan te bedenken dat een iets ruimere uitwerking van het *ref*-concept:

```
mode compl = struct(refex real re,im)
```[1)]

de bovengedefinieerde *covec* en *comat* zeer wel mogelijk zou hebben gemaakt. We hebben ons in TORRIX moeten schikken in de veel fantasielozere:

```
mode covec = ref()coscal; mode comat = ref(,)coscal.
```

4.3. *vecrow, rows, sym* en *band*

Een belangrijke eerste stap in de richting van een mode voor bepaalde klassen van dunbezette matrices is:

```
mode vecrow = ref()vec        # dus ref()ref()scal #
```

Hoewel een *vecrow* best plezierig kan zijn om eens een rijtje *vec*s mee te administreren, is zijn primaire functie een matrix te representeren. Dat is dan nog op verschillende manieren mogelijk: *vecrow* kan worden geïnterpreteerd als een matrix van rijen (*rows*), een symmetrische matrix (*sym*) of een matrix van diagonalen (*band*) etc.

---

1) Voor *refex* zie {16} en {15}.

We definiëren op en met *vecrow* allereerst een (vrij groot) aantal basis-operaties zoals *into*, *span*, ±, ±:=, *trim*, *max*, *sigma* etc. en ook selectors *?* en zelfs *!* (zodat aan *vecrow!i* op torrix-nivo een andere *vec* kan worden geassigneerd). De *vecrow* refereert bij dit alles met ijzeren consequentie naar een total-*vecarray*. Merk op dat we met *vecrow* definitief op torrix-nivo zitten. Een *vecrow*-object wordt in principe als een matrix geïnterpreteerd, maar zonder nog te specificeren hoe. Hierdoor fungeert *vecrow* als een soort tussen-structuur voor modes die "kleur bekennen":

```
mode rows = struct(vecrow †rows);
mode sym  = struct(vecrow †sym) ;
mode band = struct(vecrow †band);
```

Om ongelukken te voorkomen zijn de veldselectors taboe - elk van deze modes geeft immers een heel specifieke matrix-interpretatie van z'n *vecrow*. Er is echter een monadische *?*, zodat *?rows*, *?sym* en *?band* de structuren (zonder weak dereferencing) terugbrengen op *vecrow*-nivo. Alle basisoperaties (zie boven) worden dus met behulp van *?* expliciet via *vecrow*s gespeeld.

Daar waar de specifieke interpretatie van de *vecrow* als rijen-matrix (eventueel kolommen-matrix), symmetrische- resp. band-matrix een rol gaat spelen - ihb. dus in de multiplicatieve operaties zoals de lineaire transformatie Au - zijn de operaties gebonden aan de modes *rows*, *sym* en *band*. Het spreekt vanzelf dat alle operaties op *vecrow* en z'n diverse kleuren, in de kortst mogelijk keren worden teruggespeeld naar de primitiva *vec* en *mat*.

Tenslotte merken we nog op dat *vecrow* t.a.v. de additieve operaties eventueel ook gebruikt kan worden voor de representatie van een dunbezette vector. Een *rows* kan goed worden gebruikt voor een driehoeksmatrix waarvan de *vecrow* dan gegenereerd kan worden door bijvoorbeeld:

```
proc gentriang = (int n)vecrow:
    with i thru vecrow triang = genvecarray(1,n)
      do triang(i):=genvec(i) od
```

4.4. Open einde

Aldus kan men in de stemming geraken voor een science-fiction droom over super-modes *vector* en *matrix*, die ieder voor zich een *union* zijn van alles wat men bijeen kan declareren aan zinnige structuren voor de objecten van V, inclusief *proc(int)scal*, *proc(int,int)scal*, *proc(int)vec* etc. Selectors *?* en *!* zorgen voor de isomorfie met S^n en wel zo, dat bijv. *u!i:=s* voor

iedere i tot gevolg heeft dat *u?i* deze *s* weer oplevert. Verder implementeren we uiteraard alle operaties +, −, × etc. met en zonder "<" en ":=".

Hierbij doet zich de voor taal-ontwerpers wel interessante vraag voor, wat voor soort mode/operator-pakket organisatie je hiervoor nodig hebt - bij voorkeur ad libitum uitbreidbaar en ontvankelijk voor recursie over de mode. ALGOL68 is hier een ontoereikend instrument.

Een meer nuchtere vraag is natuurlijk waar (voorbij *vecrow*, *rows*, *sym*, *band* en eventueel nog (,)*mat*, *matlist* en *mattree*) de grens van de praktische relevantie ligt. Wij hebben voorlopig geen antwoord. Literatuur-studie (o.a. {01}, {02} en {21}) doet vermoeden dat men in de numerieke praktijk op het punt van vector- en matrix-representatie nog maar nauwelijks uit de FORTRAN-luier is. Het blijft dan ook vooralsnog een open vraag in wat voor behoeften een modernere wieg-met-TORRIX-kussens kan voorzien.

We hebben er wel enig vertrouwen in, dat het total-*array* idee algoritmische technieken gunstig kan beïnvloeden en dat het idee van een minimaal gespecificeerde *scal* vruchtbaar kan zijn. Een en ander is in elk geval technisch handig gebleken.

LITERATUUR

{01} BARKER, V.A. (ed.), *Sparse matrix techniques*, Lecture Notes in Mathematics, Springer Verlag, 1977.

{02} BUNCH, J.R. & D.J. ROSE, *Sparse matrix computations*, Academic Press, New York, 1976.

{03} DAHL, O.-J., E.W. DIJKSTRA & C.A.R. HOARE, *Structured programming*, Academic Press, New York, 1972.

{04} DEKKER, T.J., *ALGOL60 procedures in numerical algebra, part 1*, Mathematical Centre Tract 22, Amsterdam, 1968.

{05} GUTTAG, J.V., *Abstract data types and the development of data structures*, Comm. ACM 20 (1977) 396-404.

{06} HALMOS, P.R., *Finite-dimensional vector spaces*, 2nd edition, P. van Nostrand Company, 1958.

{07} IVERSON, K.E., *A programming language*, John Wiley & Sons, 1962.

{08} KOSTER, C.H.A., *Visibility and types,* Proc. Conf. on Data: Abstraction, Definition and Structure, SIGPLAN Notices 11. Special issue (1976) 179-190.

{09} LINDSEY, C.H. & S.G. VAN DER MEULEN, *Informal introduction to ALGOL68,* (thans verouderd en uit roulatie genomen), North Holland Publ. Company, Amsterdam, London, 1971.

{10} LINDSEY, C.H. & S.G. VAN DER MEULEN, *Informal introduction to ALGOL68 REVISED,* North Holland Publ. Company, Amsterdam, London, 1977.

{11} LISKOV, B. & S. ZILLES , *Programming with abstract data types,* Proc. Symp. on Very High Level Languages, SIGPLAN Notices 9 (April 1974) 50-59.

{12} MCLANE, S. & G. BIRKHOFF, *Algebra,* McMillan Company Collier, London, 1967.

{13} MCLANE, S., *Categories for the working mathematician,* Springer Verlag, New York, Heidelberg, Berlin, 1971.

{14} MEERTENS, L.G.L.T., *Abstracte datatypen,* deze syllabus, pp. 27-41.

{15} MEULEN, S.G. VAN DER, *Refers, a generalization of address and access-algorithm,* (in voorbereiding).

{16} MEULEN, S.G. VAN DER, *ALGOL68 Might-have-beens,* Proc. of the Strathclyde ALGOL68 Conference, SIGPLAN Notices 12 (June 1977) 1-18.

{17} MEULEN, S.G. VAN DER & P. KÜHLING, *Programmieren in ALGOL68 II Sprachdefinition, Transput und spezielle Anwendungen,* Walter de Gruyter, Berlin, New York, 1977.

{18} MEULEN, S.G. VAN DER & M. VELDHORST, *TORRIX a programming language for operations on vectors and matrices over arbitrary fields and of variable size,* Mathematical Centre Tracts 86 en 87 (1978) 2 dln.

{19} PAUL, G. & M.W. WILSON, *The VECTRAN language,* IBM Palo Alto Scientific Center, Palo Alto, California, 1975.

{20} SHAW, M. & W.A. WULF, *Abstraction and verification in Alphard: defining and specifying iteration and generators,* CACM 20 (Aug. 1977) 553-563.

{21} WILKINSON, J.H. & C. REINSCH, *Handbook for automatic computation*, vol. II Linear Algebra, Springer Verlag, Berlin, 1971.

{22} WIJNGAARDEN, A. VAN et al, *Revised report on the algorithmic language ALGOL68*, Mathematical Centre, Tract 50, Amsterdam, 1976.

KNOWLEDGE REPRESENTATIONS IN ARTIFICIAL INTELLIGENCE

H.J. SINT
Mathematisch Centrum

1. INTRODUCTION

Once upon a time there was a program called the General Problem Solver [1]. But it's name was soon shortened to GPS and programs to solve specific problems are still being written. Another time, one single, unifying principle was invented to prove theorems [2]. The principle was sound and elegant, but at this moment computers still don't prove interesting theorems. Also in a very remote past, people were writing programs to translate a text from one language to another (say, newspapers from Russian to English). First they thought syntax was the bottleneck, then they and their successors who worked on projects less and less ambitious began to suspect that semantics had something to do with it as well, later still, especially when psychologists came in who tried to make computers understand simple stories in order to model human language understanding, a third issue came up: 'pragmatics' or 'world knowledge': stories cannot be understood, i.e. reasonable questions about them cannot be answered, without access to all kinds of simple facts. Look at the following 'story' (and in this paper the word 'story' will always refer to something of comparable length and complexity):

> 'Yesterday I went by train to Rotterdam. When the guard came, I discovered I had lost my ticket. Well, I seem to have honest looks, for he didn't fine me'.

Without knowing

- that traveling by train is not free,
- that paying for it is done by buying a piece of cardboard,
- that people sometimes try to travel without paying,

the reasonable question 'why should the guard want to fine the narrator' cannot be answered.
In vision knowledge plays a role too: the back of a chair which is otherwise hidden behind a table is more easily recognized as part of a chair than, say, the back of a lion in the same situation as part of a lion.

Not all information needed to understand a story or analyse a scene is present in the story or the scene itself and so the performance of a story understander or a scene analyser critically depends on the amount of data it can successfully deal with. With this observation focus was changed in these areas from writing general to writing special purpose programs, and the question how to organize the data the program uses, became as important as the question what operations on them were needed. This tendency began to show in other areas of Artificial Intelligence as well. The disappointment about the failure of resolution based theorem provers has certainly played a role here, and probably also the fact that the only AI programs that can successfully compete with humans are specialists that deal with a limited amount of data on one subject. An early example of such a successfull AI program is DENDRAL [3] that assisted chemists in deriving molecular structures from their mass and NMR spectra.

Let's try to be a little more precise about the subject of this paper. First, it concerns Artificial Intelligence (AI), which to my opinion can best be described as the branch of computer science that bothers with problems for which no general solution is known, either because no algorithm exists (parsing Dutch sentences to obtain some unambiguous representation) or because using programs based on available algorithms would require an unfeasible amount of time and/or space (theorem proving).
The terms 'knowledge' and 'knowledge representation' are widely used in AI, in connection not only with natural language but with all kinds of specialized programs. A much more mundane and nearly accurate paraphrase of the question 'how to represent knowledge' is 'how to organize data in an AI program'. The use of the word data has however the objection that it is always associated with passive objects manipulated by an active program and in AI it is not unusual to organize 'data' as a set of pro-

cedures to be executed when certain circumstances arise. Hence, a term more general than 'data' is needed, but I doubt if 'knowledge' with all its psychological associations is a fortunate choice.

The core problem of knowledge representation is a search problem. In the complicated domains AI programs try to deal with, many facts can be represented only implicitly by lack of space and some data will not be available at the time the knowledge base is constructed. Hence, some deduction mechanism is needed. For example, a natural language program with knowledge about animals and their eating habits must be able to answer the question 'do seagulls eat elephants' and 'do elephants eat seagulls' both with 'no'; for instance by using the facts that small animals don't eat large ones and the fact that elephants are vegetarians, respectively. One could oppose that deduction here is not at all necessary because with, say, 100 species of animals, construction of a bit matrix telling who eats whom is completely feasible, (and realistic natural language programs certainly won't incorporate more), but then the program could never deal with 'Clyde is an elephant. Does Clyde eat seagulls?'. Finding some representation and defining some deduction process is in itself not extremely difficult; the difficulty lies in keeping the process from drowning in loads and loads of irrelevant facts it deduces in its search for the relevant one. The problem is somewhat different from that in mathematical theorem proving, where in general only relevant theorems are offered to the prover but it has to perform a many step deduction on them; while in knowledge based programs the deduction is most of the time shallow but the relevant inputs to the deduction process have to be selected from a large set.

The borderline between knowledge based AI programs and database management systems is far from clear and in some cases seems to be a matter of using techniques historically ascribed to one of the two areas (for example, the program described in [4] that retrieves references from a large bibliography and formats them according to the wishes of the user). In general, AI programs are more experimental and database managers more performance oriented.

In the next chapters I will describe four ways to organize knowledge based programs: as a resolution based theorem prover, as a

PLANNER program, (these two are out of date now but were milestones in the development of the field), as a network with a manipulator, and as a production system. There is a final chapter naming some more general, as yet unsolved problems.

Nearly all examples I will give are based on natural language understanding. This has two reasons. First, natural language understanding has played a central role in all discussions on knowledge representation. Many programs in AI are concerned with natural language, and then one should not only think of story understanders but also of programs with more practical task domains (data retrieval programs, advise givers) with a natural language user interface. Second, when taking examples from natural language all human readers will immediately understand what is expected of the program, which makes discussion a lot easier. Moreover, the examples are meant to be simple illustrations of the techniques described, and are much less sophisticated than their counterparts in real AI programs.

One general and maybe superfluous remark to end this introductory section: all programs mentioned incorporate an amount of knowledge small compared to the amount human beings can deal with, even about one topic. Incorporation of ten times as much knowledge is in general not possible without seriously affecting the program's performance.

2. TWO EXTREMES: RESOLUTION BASED THEOREM PROVING AND PLANNER

Two extreme ways to organize a knowledge based program are:

- There is a knowledge base containing the data to be handled, and there is a set of general procedures which are completely independent of the contents (though of course not of the form) of the knowledge base. This organization is called declarative; resolution based theorem proving is the standard example.

- For each fact to be incorporated in the knowledge base, a statement is written in some special programming language. This kind of organization is called procedural; PLANNER is the oldest example.

These two extreme approaches share two properties: they are abandoned by now as models of knowledge representation (I'm not talking about mathematical theorem proving) and they have been very influential. I will give a short overview of both.

2.1. Logic with a theorem prover

One can organize knowledge as a set of logical formulas considered to be true; first order predicate calculus is used mainly. A request can be presented to such a program as a theorem to be proven; an answer can be extracted from instantiations of variables made during the proof process.

This organization was much favoured when resolution was just invented [2] and faith in the power of theorem proving was large, but it has been decreasing in popularity since. I will illustrate the process of resolution with an example. (People not familiar with predicate calculus might want to skip this one).

The knowledge base contains the next four statements:

1. All elephants are grey.
2. All elephants are mammals.
3. All mammals have lungs.
4. All polar bears are white.

The question will be

5. 'Given that Clyde is an elephant, what is his colour?'.

A straightforward translation into predicate calculus is:

1. $\forall x$ (ELEPHANT(x) => VALUE(colour(x), grey))
2. $\forall y$ (ELEPHANT(y) => MAMMAL(y))
3. $\forall z$ (MAMMAL(z) => HAS-LUNGS(z))

4. ∀w (POLAR-BEAR(w) => VALUE(colour(w), white))

5. ELEPHANT(Clyde) => ∃t (VALUE(colour(Clyde), t)

The last formula states that if Clyde is an elephant, he has some colour.

Resolution theorem proving uses a different representation, Conjunctive Normal Form. In this representation, each formula is a disjunction of possibly negated literals, existential quantifiers are removed (by a trick I won't explain here), all remaining variables are implicitly universally quantified. Furthermore, resolution is a proof-by-contradiction method: the negation of the theorem to be proved is added to the database. Conversion to CNF yields 6 formulas:

1. ~ELEPHANT(x) v VALUE(colour(x), grey)

2. ~ELEPHANT(y) v MAMMAL(y)

3. ~MAMMAL(z) v HAS-LUNGS(z)

4. ~POLAR-BEAR(w) v VALUE(colour(w), white)

5. ELEPHANT(Clyde)

6. ~VALUE(colour(Clyde),t)

where ~ is the negation and v the disjunction sign.

The shortest proof proceeds as follows.

The instantiation of 1

~ELEPHANT(Clyde) v VALUE(colour(Clyde), grey)

combined with 5

ELEPHANT(Clyde)

yields

7. VALUE(colour(Clyde), grey).

The instantiation of 1 and the 'resolving' of the literals ELEPHANT(Clyde) and ~ELEPHANT(Clyde) is one step.

Instantiating 6 to

~VALUE(colour(Clyde), grey)

and combining it with 7

VALUE(colour(Clyde), grey)

yields the desired contradiction.

The instantiation of the unknown t in formula 6 was 'grey', and this is the answer to the question. Note that the proof process can make several false steps; it can combine 1 and 4 to yield a new formula MAMMAL(Clyde) and then this new one and 2, yielding HAS-LUNGS(Clyde). Alternatively, it can combine 4 and 6 (instantiating t to 'white' and w to Clyde, yielding POLAR-BEAR(Clyde).

Using logic as a representation has large advantages especially from a theoretical viewpoint: its foundations are well established, many of its variants are complete (i.e. any request that can be answered will theoretically be answered correctly), and it is very good in modelling some awkward language constructs. Existential quantification is one of them; representing and using knowledge like 'Every disease has some cure' with its implicit dependence of the cure upon the disease, is easier in predicate calculus than in any other representation I know of. More subtle points include the representation of beliefs and wants and problems about intension and extension of objects. (MONTAGUE, [5]). Moreover, the fact that a theorem prover can be written independently of the formulas it handles, and hence doesn't commit the programmer to any special application, is very attractive.

Its main disadvantage for use in AI is its lack of selectivity. Theorem provers cannot decide which formulas are relevant to the request and which are not. Their deductive power is considerable and so they generate and use many formulas; when the knowledge base (the set of formulas considered to be true) has a non trivial size, a fast combinatoric explosion is the sad result. Although it is certainly true that combinatoric explosion is a problem with all representations, it seems at the moment to be more successfully attacked by other paradigms. A second disadvantage (which in the long run could be the more serious) is that theorem provers are too strict. It is often important to be able to state some general rule and yet to handle exceptions, or to handle incomplete or fuzzy knowledge, stating that 'if x, then there is some evidence that y'. A program shouldn't break down when along with the knowledge that elephants are grey, the knowledge is entered that Clyde is an elephant, and that he is white (a theorem prover will from that moment on be able to prove anything at all, at least if it knows as well that grey $\neq$ white, and that elephants cannot have two colours at the same time). One doesn't want to be forced to remove the rule that

elephants are grey either. The exception can be handled by modifying the formula stating that all elephants are grey to one stating that all elephants which are not equal to Clyde are grey. However, doing such a modification automatically is impossible. It would require that the program checks for each new formula whether it clashes with the database, i.e. it should try to prove its negation. If the formula does clash, the clash will 'eventually' be detected; if it doesn't there is no guarantee that the prover will halt. As a consequence there can be no guarantee that a newly entered fact does not clash with the database. This makes automatic extension rather dangerous, how easy translating new facts to their predicate calculus equivalent may be.

2.2. The PLANNER formalism

The inventor of the procedural formalism is Hewitt, who designed the programming language PLANNER to support it [6]. The design of PLANNER was more or less a challenge of resolution theorem proving, which after four years still failed to meet the high expectations it had roused.

PLANNER is a language to prove theorems, which are offered to a program as a goal statement, which is a PLANNER construct itself. Among its basic facilities are:

- Functions THASSERT and THERASE who, when executed, add their argument to and delete it from the database, respectively. (N.B.: there is no database separate from the PLANNER program; one has to execute a PLANNER THASSERT statement to get anything into the database at all).

- Two kinds of theorems: Consequent theorems, superficially described as 'to establish some goal X, try to establish Y', and antecedent theorems, equally superficially described as 'when X is asserted, then assert Y'.

- A pattern matcher which evokes theorems on a goal when that goal matches the 'X' in a consequent theorem, and on the database when an assertion in it matches the 'X' in an antecedent theorem.

- An automatic backtracking facility. The Y in a consequent theorem can have several alternatives; if such an alternative fails for some reason, the environment which existed at the time it was first tried is restored and the next alternative is tried.

- A whole machinery to help guiding the search process, including the possibility to recommend the use of certain theorems in certain situations, and to restrict the search depth.

PLANNER was never fully implemented. A subset of it, having all basic facilities but being much less fancy than the original proposal, was implemented by Winograd e.a. (called MICROPLANNER). This subset was used in the program SHRDLU which could manipulate a simple block world receiving instructions in English [7].

The same example as above, but this time in PLANNER notation:

1. 'All elephants are grey', formulated as a consequent theorem:

```
(THCONSE(x)(colour $?x grey)
           (THGOAL(ELEPHANT $?x)))
```

'to proof something is grey, proof it is an elephant'.
$?x denotes that x is a variable, to be bound by pattern matching.

2. 'All elephants are mammals', formulated as an antecedent theorem:

```
(THANTE(x)(ELEPHANT $?x)
       (THASSERT(MAMMAL $?x)))
```

'if x is an elephant, assert it is a mammal'.

3. 'All mammals have lungs':

```
(THANTE(x)(MAMMAL $?x)
       (THASSERT(HAS-LUNGS $?x)))
```

4. 'All polar-bears are white', again a consequent theorem:

```
(THCONSE(x)(colour $?x white)
        (THGOAL(POLAR-BEAR $?x)))
```

We assert that Clyde is an elephant and ask to find his colour:

5. (THASSERT(ELEPHANT Clyde))

6. (THPROG(y)(THGOAL(colour Clyde $?y)))

'Find a y, such that y is the colour of Clyde.'

PLANNER evaluates the goal statement and first searches the database for an assertion that matches the goal, i.e. '(colour Clyde $?y)'. This fails. Then, it searches the consequent theorems to see if there is one that matches the goal, and finds 4. As a result, x in 4 is bound to Clyde, y in 6 is bound to white, and a new goal is set up: (POLAR-BEAR Clyde). This goal fails, and hence the bindings for x and y are undone and PLANNER looks for an alternative for the original goal. It finds theorem 1, binds x in 1 to Clyde, binds y in 6 to grey, sets up the goal (ELEPHANT Clyde), finds it in the database, hence succeeds and returns grey as a value. When to use consequent and when to use antecedent theorems can be specified by the user, for instance - 'exhaust all consequent theorems before using any antecedent theorem', which is in fact

the strategy followed here.

There are some important differences between this approach and the resolution theorem proving one. First, the user can guide the search process in a number of different ways, which diminishes the risk of running into a combinatorial explosion. This may not be very clear in the example, but was in practice (i.e. in MICROPLANNER) a great improvement. Second, exceptions are much less of a problem in PLANNER. When trying to prove something, the database is checked always before any theorem is tried, and hence asserting that Clyde is white will do the trick. Because theorems are not proved by looking for a contradiction, there is less risk of proving unprovable things as a consequence of an inconsistency in the database. Third, PLANNER makes a distinction between consequent and antecedent theorems. Look again at our example. If the program wants to find out if some animal is grey, then checking if it is an elephant seems a sensible approach, as being grey is a sufficiently rare attribute. On the other hand, if the program wants to find out if some object is a mammal then checking to see if it is an elephant doesn't seem very sensible because there will be lots of different mammals around. Hence the first statement is incorporated as an antecedent theorem, and the second as a consequent theorem.

There is a problem here: suppose we assert the statement that Fred isn't grey, and then try to prove he is not an elephant. Although this follows from the second statement, PLANNER will be at a loss here. Including the antecedent theorem

```
(THANTE(x)(NOT(COLOUR $?x grey))
          (ASSERT(NOT(ELEPHANT $?x))))
```

would solve the problem but is hardly satisfying. No adequate solution for this problem has been found.

A second problem with PLANNER is the automatic backtrack facility. (For a discussion, see [8]). The environment that existed at the time a failing alternative was tried first is restored, thus destroying all information about the cause of the failure. If several alternatives will in the end all fail for the same reason, there is no way in PLANNER to prevent trying them all.

It became soon clear that PLANNER had a lot of defects and would be unable to really solve the representation problem, I suppose that's the reason it was never implemented. Nevertheless, its design evoked a lot of discussion and some of the ideas in it, especially the 'pattern directed invocation of procedures', proved powerful enough to be used in many programming languages developed afterwards.

3. NETWORK REPRESENTATIONS

QUILLIAN [9] was the first to propose a graph with nodes representing concepts and links representing relations between them as a model for human memory. He hoped that he could use this scheme to adequately represent anything that is needed to understand natural language. Of course he failed, but his 'semantic memory' bred a large and diverse offspring, still mostly used for natural language understanding (though not always, in [10] foundations for a program to process electronic mail within a computer network are described; knowledge about commands is represented in a network). I think there are three reasons why network representations are so popular; two of them are more or less valid but the third is not. This last, fallacious reason for their popularity is, that representing concepts as nodes and relations as links seems very natural and understandable - at least to humans looking at a nice picture of a network. As a consequence, authors proposing network schemes get easily carried away. Both WOODS [11] and BRACHMANN [10] have, in papers called 'What's in a link' and 'What's in a concept', heavily criticized the lack of specification in many proposals of what exactly nodes and links are to stand for, and of the operations needed to manipulate the nets they form. One cannot simply connect the concepts father - isaac - abraham - bible or number - eyes - two - person, and hope a program will work out all by itself what to do with these connections. A choice must be made between either having many different kinds of links and nodes, and general procedures to handle each kind, or having few types but attaching special procedures to individual concepts and relations. I will return to this issue.

A better reason for their popularity is that they can easily model property inheritance, a point which has bothered people in natural language understanding considerably. In the previous chapter, a lot of

work was needed to find the colour of the elephant Clyde. Many other properties can also be deduced from the fact that Clyde is an elephant: he has four paws, has bones in his inside to keep him in the right shape, likes peanuts etc. Storing all these properties with the concept for 'Clyde' doesn't seem sensible; when another elephant comes along they all would have to be stored a second time. On the other hand, when storing each property with the appropriate superset there should be an easier way to retrieve them.

A network like that in fig 1 solves this problem. In this network there are two kinds of links, links labeled 'IS-A' pointing from a set to its superset, and links labeled otherwise that assign a value (the value they are pointing at) to a property (their label) for all members in the

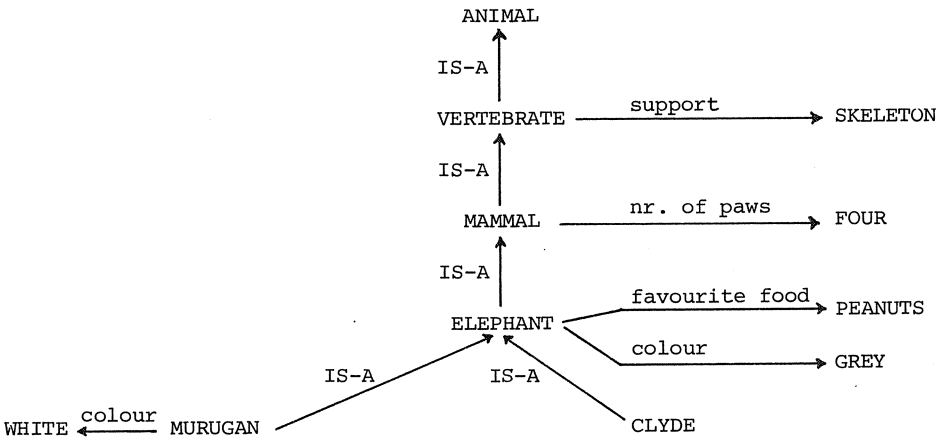


Fig. 1

set they are pointing from. (Of course, the net manipulator must be programmed to discriminate between the two kinds of links; the difference comes in no way natural). When searching the value of some property associated with some concept, the manipulator locates the node for the concept, checks its links to see if one is labeled with the property looked for, if not, looks if it has an outgoing 'IS-A'-link and repeats the check with the node at its other end etc, until the desired property is found or the root is reached. Exceptions are free with this paradigm as can be seen in the elephant net containing a second elephant Murugan which is white. This is nice indeed, though there are some difficulties (nodes can have more than one outgoing 'IS-A'-link for instance, the statement 'Murugan is an albino' would be less straightforward as not

all albinos are elephants. Also, the expressive power of such a net is rather limited; adding the fact that 'Clyde is older than Murugan' is not possible in this simple paradigm. (Adding a link from Clyde labeled AGE and pointing to '> AGE(Murugan)' or something equivalent is certainly not the right way to do it).
There is a more subtle problem with the albino example. Let's assume the clashing colour property of Murugan is detected somehow. This can be done fairly easily but includes a lot of work; a nice idea is to have a program do these kind of clash detection in the time it is not working on any user request, as is suggested in [12]. (From this paper I borrowed the elephant Clyde by the way). Then, the program can either store Murugan as an exceptional (because white) elephant, or as an exceptional (because grey) albino. A way to force the program to make the right choice, is to distinguish between 'defining' and 'asserted' properties , see for instance [11]. The idea is that properties of the first kind are essential for any object to be classified as a member of the set to which the property is attached, while properties of the second kind are more or less incidental. As a consequence, exceptions on asserted properties are allowed but exceptions on defining properties are not. (This distinction exactly parallels the distinction in philosophy between analytic and synthetic judgments. Its usefulness is denied by Wittgenstein, who finds in AI adherents in Schank and Winograd).

Finally, networks can reduce the search problem mentioned as a weak point for theorem provers. Links are supposed to connect concepts that bear upon one another, and once an entrypoint suitable for a certain request has been found in a network, one can hope to completely bypass nodes that don't bear upon the question at all. A method that can facilitate guiding the search process at many levels of specification is the use of 'slots' (Minsky's term). The basic idea was introduced by FILLMORE [13], who proposed to center the representation of an English sentence around its main verb, and to associate with each verb a fixed set of linktypes, roughly corresponding to the cases that are associated with that verb in highly inflected languages like Latin or, a better example maybe, Finnish. Labels for these linktypes are typically 'agent', 'direct object', 'indirect object', 'place where from', 'instrument', 'time until', etc. Though a considerable number of linktypes is needed (maybe 20 or something), hardly ever more than 5 will be associated with

one verb.

In the prototype representation of a verb these links point to empty places, slots that have to be filled in an instantiation. When a sentence comes along, a representation is build trying to fill the slots of its main verb. Note that not all slots have to be filled always; 'John walks' is OK but 'John sells' is not though 'John sells his house' is OK again. A question leads to building an incomplete representation that can be matched against the database. An obvious refinement of this scheme (providing a link to the concept of property inheritance) is to restrict the value that a filler for some slot can take to the members of some class- one could say, introducing type checking. To say it once more, procedures must of course be written to handle nodes and links in the action representations, and these are not those that handle property inheritance. An example that will hopefully lead to many questions, some of which will be answered in the sequel:
I specify 'walking' as an action with four slots.

- An animal agent,
- A starting place,
- A finishing place,
- A distance.

The agent is obligatory, the others are optional. (I could have chosen more or different slots, but I don't want to make my example too elaborate). There are four pieces of network in the knowledge base relevant to the example. Three are shown in fig. 2a, the fourth is the elephant net of fig. 1.

Attached properties are not shown here. The specification that distance should be a number is not nice. Instead of giving a better solution, I will for the moment be satisfied by declaring that the parsing process can recognize '10 km' as a distance measure and convert it to meters, so that distances can be compared as numbers. The statement '10000 is a number' is even less nice. What you want here is a recipe to recognize numbers, but such a thing doesn't fit in this sample paradigm. When the sentence 'Clyde walks 10 km to the lake' comes along, the

parser will build the instantiation of 'walking' shown in fig. 2b, using the template shown before and checking for value restrictions. (The only real value restriction in this example is on the agent; all others follow from the specified cases, they are reduced to syntactical restrictions. 'The book walks to the lake' can be parsed with 'the book' as agent; 'Clyde walks in due time' however can not be parsed because there is no slot for 'in due time').

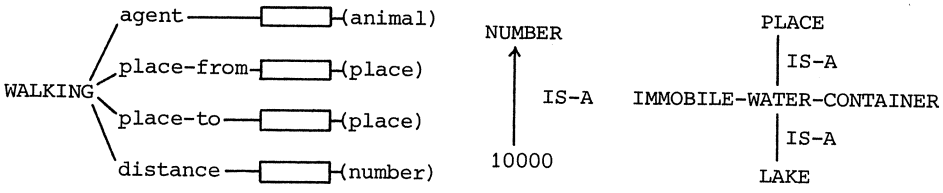


Fig. 2a

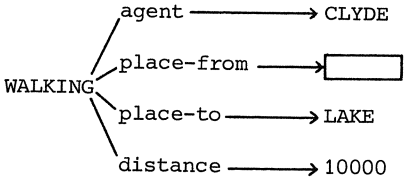


Fig. 2b

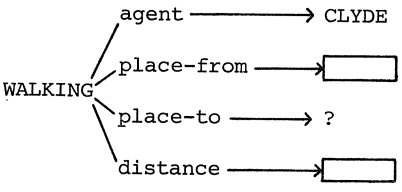


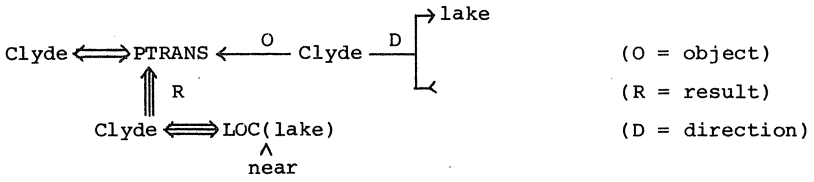
Fig. 2c

Fig. 2

Three of the four slots are replaced by links ('normal' pointers this time) to other concepts in the net.

The question 'where did Clyde walk to?' will subsequently lead to the pattern shown in fig. 2c, the slot for which a filler is asked is denoted by a question mark. Matching this against the knowledge base will lead to an answer 'to the lake'. (Let's hope the base is not too large, or that there is some administration immediately giving all pointers to instantiations of walking).

Until now we have considered network schemes which can be called declarative; networks have different kinds of nodes and links, and the network manipulator consists of a set of general procedures to handle each kind, without commitment to special concepts. Although my unsophisticated examples won't inspire much trust in this approach (there is for instance in the last example no way of answering the question 'where is Clyde' after having entered the fact that he walked to the lake), there are programs using it and performing well. An example is the SAM program [14] which uses several declarative network schemes.
At the heart of this program is Schank's Conceptual Dependency, a representation centered around 11 basic actions to which all others are reduced. 'Walking' is in this representation an instance of the primitive action PTRANS (=physical transfer), and 'Clyde walks to the lake' would be represented as



The representation includes a causal link to the change of place of the object of a PTRANS. Note that there are six kinds of links involved in this example. To make inferences following less directly from the semantics of the action, SAM uses the Conceptual Memory of RIEGER [15], again a declarative network approach. This scheme allows for instance to deduce from 'John's cold improved because I gave him an apple' that John has eaten the apple. At the level of 'world knowledge' SAM uses scripts - descriptions in Conceptual Dependency terms of stereotyped situations. To handle the train story in the introduction, a train journey script would be used which would include descriptions of getting to the station, buying a ticket, looking up the right platform, getting on to the train, showing the ticket to the guard, etc. SAM can work in an inference mode, and is then able to answer questions about small stories, or in a paraphrase mode, and can then rephrase the story, including all details left out in the original but specified in the scripts used.

Some of the problems and restrictions with networks of these types are modelling existential quantification, negation and logical disjunction. Of these, negation (this elephant is not grey) seems to be the most serious; I have a feeling that the other two would still go unnoticed at the present state of the art if philosophers hadn't been dealing with them for so long. Deficiencies that seem to me more serious, like the ability to handle complicated temporal relations (just look at the previous sentence for an example) are silently understood to be too difficult to handle at present.

One of the most influential papers on knowledge representation has been [16] by MINSKY. In his proposal knowledge is represented as a collection of so called frames; each frame represents some concept ('room' or 'birthday party' or 'science fiction novel' or 'wedge'). Associated with each frame are indications about its use, like the situation in which it is likely to be applicable, things that can be expected next (with the frame for 'door' information like 'keep corridor and room frame ready' could be associated). Furthermore, a frame has a collection of slots, and with each slot information is associated about what kind of filler (often another frame) can be expected, what should happen if no filler comes up (reject the frame or assume some default) and what has to be done if an unexpected filler comes up. Minsky wrote his paper from the viewpoint of a psychologist and not from that of a programmer, (which saved him the trouble of specifying all kinds of tiresome details).

Frames can be used in many different ways. To give some idea, I will give two examples. First, let's look once more at the 'Clyde walks 10 km to the lake' example. A frame for walking is shown in figure 3. Again there are four slots. Variables prefixed with question marks can be bound to some value in an instantiation of the frame. A variable without the ?-prefix refers to that value. The slot for the agent specifies a value restriction: the agent must be an animal, and a default: if nothing further is known about the agent, he is assumed to be human. The slot for the place where-from also specifies a default: the place where the agent was before. (The value restriction is here implicit in the case tag of the node as 'place where-from'). Slots are often filled by other frames. A frame for 'distance' is also shown in figure

3, allowing alternative specifications, for instance '10 km', 'half an hour', ' a great distance'. With some nodes procedure calls are associated. 'Walking' has three; one to assert that agent X is no longer at Y, one to assert that agent X is now at Z, and one to check if the distance is walkable. (A program may be expected to protest if 40.000 km is specified). Procedure calls are only executed if all their arguments have received values. (So, if Clyde is under a tree, and then the fact that he walks away is entered, his where-abouts are afterwards unspecified).
The declarative approach is abandoned here; at least some of the procedures to handle the network are specific to concepts represented.

Second example. Suppose a story is entered starting with the sentence 'Clyde is an elephant 10 cm tall'. The elephant frame contains a slot for height, specifying 3 m as a default and accepting anything between 1 and 5 m. However, instead of rejecting 10 cm as a filler immediately, the instantiation of the elephant frame is passed on to a

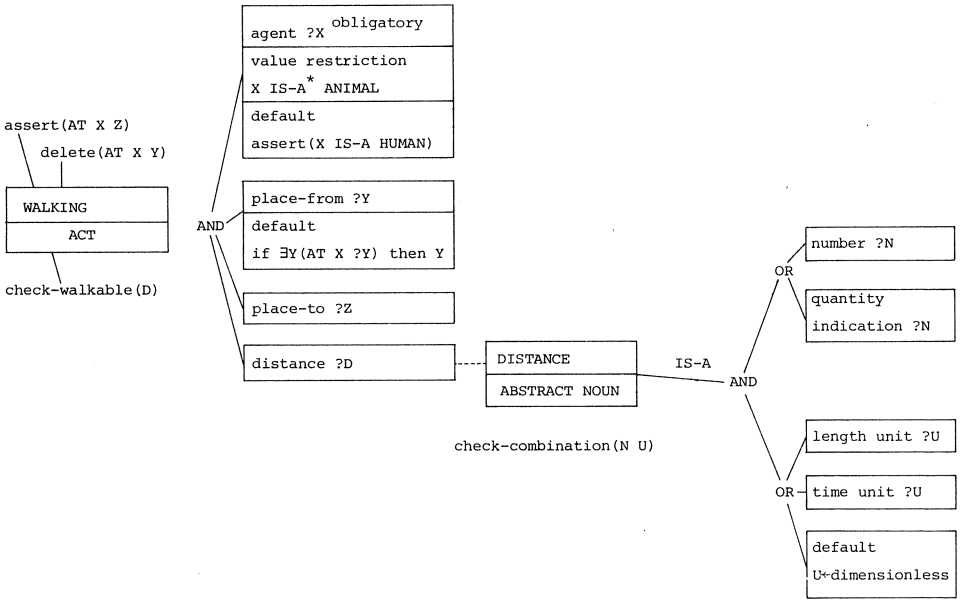


Fig. 3

special program part called 'the complaint department'. This program part has the task of patching up inconsistencies and could come up in this case with the suggestion 'Assume Clyde is a toy elephant'. Only when this suggestion is not justified by the sequel of the story, the program will reject the small size. Some inconsistencies like a black albino will be rejected immediately, probably asking the parser to try and find another representation for the incoming sentence.

Several AI programs based on frame representations are written, for scene recognition as well as for natural language. An attempt to integrate the frame approach and the procedural approach is done in the KRL (Knowledge Representation Language) project. For an overview of this programming language, see [17], for results with a first implementation (KRL0), see [18]. In KRL frame representations can be built and several kinds of procedures can be attached to them, including equivalents of the PLANNER antecedent and consequent theorems. Interesting features of the language are the possibility to give multiple descriptions of an object (kissing as a physical or kissing as a social act, a lake as a water container or as a recreation site), and to classify an object by comparing its description with a stereotype. Future versions of KRL will include sophisticated scheduling and resource allocation facilities.

Frames greatly help to guide inference; once a proper frame is selected a lot of (not explicitly represented) information is available. In this respect they are close to other, independently proposed mechanisms like scripts [14]. Other inferences derive from the value restrictions on the fillers of a slot, as could be seen in the 'walking' example, specifying being human as a default for the agent. In principle frames can guide induction as well. When confronted with an unknown situation ('I was looking out of the window to the beautiful landscape rapidly passing me') one can try to find a match between information gathered about the situation and the top level slots of some frame, and so be able to name the situation. This is very hard however if the number of frames becomes large and there is no knowledge at all about what to expect; the search problem pops up again.

4. PRODUCTION SYSTEMS

In this chapter a way of representing knowledge will be considered which, for a change, is hardly ever used for natural language understanding: production systems. A production system (PS) is a general scheme for information processing. It consists of a set of rules with two sides which specify a condition and an action respectively. A question is presented to a PS as a datastructure, which is very inappropriately called the database. I will use this term, but remember that the rules form the knowledge base of the program while the database is a kind of scratchpad containing intermediate results gathered in the process of fulfilling some request. Under control of an interpreter the following basic loop is executed:

- A rule is selected whose conditions are successfully matched against the database; the matching process is governed by the interpreter which also decides what should be done in case none or more than one rule applies.

- The action specified in the right hand side of the rule is performed; in general, this action will modify the database. 'Stop execution' is a possible action.

In the most 'pure' form of a PS, both condition and action part of a rule are simple literals, the condition is satisfied if it occurs literally in the database and the action consists of simply replacing the matched part with the other side of the rule. In this form, PS's were proposed by Post and can typically serve to implement a parser for a formal language (and untypically to code any Turing Machine).

The use of PS's in AI has considerably diverted from this pure form. The condition part can for instance amount to searching for some graph structure in the database (DENDRAL, [3]) or matching the database against a pattern with free variables that are bound during the pattern match (HEARSAY, [20]). The action part can be a complete program. Finally, the control structure is sometimes modified - for instance, by hav-

ing in each cycle only a limited set of active rules that can be selected instead of testing them all. For an overview of production systems, see [19].

A simple minded example: the next set of rules can serve to identify an animal species some of whose characteristics are offered as an input. They are given in an easily understandable notation rather than in terms of an existing pattern matching language. The part left of the ==> sign is the precondition, the statement to the right is the action.

```
colour   ?x   ==>  if      x = grey   then (elephant mouse wolf)
                     elif  x = brown  then (giraffe lion)
                     elif  x = white  then (polar-bear)
                   fi

size     ?x   ==>  if      x = small  then (cat mouse)
                     elif  x = large  then (elephant giraffe)
                   fi

food     ?x   ==>  if      x = meat   then (lion cat wolf polar-bear)
                     elif  x = plants then (mouse horse elephant giraffe)
                   fi

domestic ?x   ==>  if      x = wild   then (elephant wolf lion giraffe
                                            polar-bear mouse)
                     elif  x = tame   then (cat horse)
                   fi
```

The rule interpreter for this set works as follows. The precondition of each rule in turn is matched against the database; it succeeds if the database contains the first keyword in the precondition. In that case, the next word in the database is bound to the variable x, and the if-statement in the right hand side is executed. This statement delivers a result: a new set of words, or empty if all tests fail. This result replaces the matched part in the database. If no rule applies anymore, the rule interpreter counts occurrences of each word in the database and returns the word(s) that occur most frequently as answer.

For instance, the initial database 'colour grey size small' would be converted to 'elephant mouse wolf size small' by applying the first rule, and then to 'elephant mouse wolf cat mouse' by applying the second rule, yielding 'mouse' as an answer. The database 'colour red size small domestic tame food meat' will yield the answer 'cat', the colour-clue is not used.

The most important feature of a PS is the indirect but uniform communication between rules: no rule ever calls another directly, but all rules write into the one, common accessible database so that the effect of application of any rule can be inspected immediately by all others. Note that this is a control structure fundamentally different from that provided by Algol-like languages. The obvious weak point of PS's is the duty to test many rules at each cycle, how many depends on the control structure; it typically averages out to half the number of rules.

Indirectness of communication results in a very modular structure of the knowledge incorporated in a PS, and hence this knowledge can fairly easy be extended or modified; adding a new rule can be done without change to any of the others as it doesn't have to be explicitly fitted into the control structure (which of course doesn't mean that the behaviour of the system won't change). This provides a flexible (though sometimes opaque) way of programming, which can be useful in experimental surroundings; PS's are frequently used in modelling of psychological processes (see for instance [21]).

Much of the behaviour of a PS depends upon its control structure and the interrelation between rules. The architecture of the select-act cycle is an important feature; if for instance the rules are always tested in the same order and the first applicable rule is always selected, then system behaviour can be changed considerably by reordering the rules; a feature used by Newell for adapting his PS's until they showed the desired behaviour. On the other hand this feature reduces easy extensibility; adding a rule requires detailed knowledge about the others. Another possibility is to have mutual exclusive preconditions, which guarantees that only one rule can be selected at each cycle; but this imposes a restriction on new rules and hence also reduces extensibility. The most suitable control structure to allow easy extension is first

gathering all applicable rules and then selecting one out of this set; either an explicit or an implicit ordering can be used. This however doubles search effort. PS's with a non trivial number of rules often use a specialized control structure in order to reduce testing overhead. A possibility sometimes used is to group rules together, and have at each moment only one active group. Only rules within this group can be selected. This however violates the uniformness of PS structure. Having the rule interpreter make a preselection of rules to be tested is a better solution, and so is taking care to test preconditions shared among more than one rule only once.

One can simulate a direct call from one rule to another by having the calling rule put some specific marker in the database which serves to inhibit application of all rules with exception of the called one. This should however be considered bad programming practice in a PS, (the equivalent of 'globals considered harmful' in this environment, where all communication is by means of the global database, is 'calls considered harmful') Such tricks affect the modularity of a PS and make flow of control completely opaque. As a consequence, application of PS's is virtually restricted to areas where flow of control is simple and where the knowledge to be incorporated in the rules can be divided into nearly independent chunks. Production systems are typically used for performance oriented, knowledge based specialists like DENDRAL [3] (with a knowledge base on a small subject from chemistry) or MYCIN [22] (with a knowledge base on bacterial blood diseases). Recognition tasks (scene analysis, speech understanding) also provide suitable domains, knowledge in these areas is generally incomplete, the database can be used to gather hypotheses and all pieces of evidence in favour of or against them. HEARSAY [20] is a comparatively successful speech understanding system with a PS-like organization.

A PS can be completely declarative, i.e. both sides are passive data manipulated by the interpreter, or completely procedural, for instance when the precondition side is a pattern containing variables to be bound, and the action side a body of executable code using the values found in the pattern match. A PS is then in fact a set of procedures, called when the pattern which forms their argument matches the database.

5. GENERALITIES

5.1. Declarative versus procedural representations

There have been violent disagreements in the past about which approach is the best for representing knowledge: the declarative or the procedural. The declarative approach with its clear separation of knowledge base and knowledge manipulator is certainly the more appealing one, especially when the representation used is as uniform as in the case of logic. Having a special procedure which is called each time the concept 'rain' turns up is ugly and ad hoc in comparison.
Adherents of the procedural approach maintain that separating knowledge from the way it is used and looking for uniform representations with uniform processes defined on them is running into a blind alley, as the knowledge needed to handle the complicated task domains of AI is not uniform and cannot be separated from its use. The discussion has died down by now. It seems clear that some knowledge can be best represented as a datastructure (Linnaeus' classification of animals) while other knowledge seems to be procedural in nature. An extreme example of the latter is a skill like driving a car. One can of course account for the relationship between the movements of a driver and the behaviour of the car by looking at the way cars are constructed, but this relationship explains nothing about driving.
Some practical issues remain; for instance datastructures are much easier to extend than sets of procedures. When a mixed approach is used, many decisions must be made where to put things: in a procedure or in the database, and not even a start has been made to find suitable criteria.

5.2. Redundancy

As remarked before, it is impossible to explicitly represent all knowledge in a knowledge base of non trivial size. This doesn't mean however, that the knowledge represented explicitly cannot contain some redundancy; if some fact is deducible but has a high probability to be often needed as input for further deduction, storing it explicitly seems sensible. (Example from [17] : If it is known that Mary is a plumber and that all plumbers are human; the fact that Mary is human can be deduced.

Being human however is an attribute much used in reasoning about some object.) Once the decision is made to store some redundant information explicitly, the question arises how much and which. Moreover, should the programmer decide about that (which seems sensible to me for the time being) or should the program upon deduction of some fact decide to keep it or to throw it away? This is a point where more experience is needed.

5.3. Canonical form

Each representation contains some basic building blocks. For these one can choose few, low level primitives and reduce to them all notions represented, or higher level, more diverse primitives. The first choice has the advantage that similarity between different concepts immediately shows in the representation, and, more important, that if reduction to these primitives can be done in a unique way, deciding if two expressions are equivalent has become trivial. For example, the two sentences 'Jan heeft een hond Fikkie' and 'Fikkie is a dog and his master is Jan' would be recognized as different ways of expressing the same fact. Schank is a great advocate of this approach, it is the main philosophy behind his Conceptual Dependency. The disadvantage is that the representation is a lot bulkier. Furthermore one can have severe doubts (as for instance WOODS [11] argues convincingly) if a unique reduction procedure for something as complicated as natural language exists.

5.4. Task dependency

Division of knowledge based programs according to their task domain can be done in several ways.
First, one can look at the kind of input and then distinguish programs for speech recognition, for natural language understanding (input by means of a keyboard), for scene recognition, or (the simplest kind) programs accepting only some formal language.
Then, one can make a division according to the contents of the knowledge base: does the program know about blocks, about drugs and bacteria, about traffic accidents, about samples from the moon, etc.
Finally, they can be divided according to their output: should the program answer questions, should it react on commands by simulating some

action or by undertaking real action, should it silently accept an input stream and only react on certain conditions (tasks like watching traffic in a tunnel), etc.
At the moment people merely select some input, task domain and output ('A program to simulate the manipulation of blocks on commands given in English' , [7]) and choose the representation(s) which seem(s) the best fit for this sequence. There is little systematic investigation into the dependency between each of these sub-tasks of a program and the representation(s) used - should there be one, uniform representation whatever the program accepts or does, should the representation depend on the contents of the knowledge base alone, should it depend on the output but not on the input and contents? Again, more experience is needed before these questions can be answered.

5.5. Learning

In the long run AI programs will have to be able to learn - and this doesn't mean storing new deductions made or adapting coefficients of a heuristic function, but building their own knowledge structures. Finding an adequate knowledge representation is one (but not the only) key problem which has to be solved before the problem of learning, which has been abandoned as too difficult for the present state of the art, can be taken up again.

REFERENCES

[1] NEWELL, A., J.C. SHAW & H.A. SIMON, *Report on a general problem solving program for a computer,* in: Information processing: proceedings of the international conference on information processing, Paris: UNESCO, (1960).

[2] ROBINSON, J.A., *A machine oriented logic based on the resolution principle,* JACM 12, (1965).

[3] FEIGENBAUM, E.A., et al., *On generality and problem solving - a case study involving the DENDRAL-program,* in: Machine Intelligence 6 (eds Meltzer & Michie), Edinburgh University Press, (1971).

[4] SANDEWALL, E., *Some observations on conceptual programming,* in: Machine Intelligence 8 (eds Elcock & Michie), Ellis Horwood Ltd., (1977).

[5] MONTAGUE, R., *Formal philosophy: Selected papers of R. Montague,* (ed. Thomason), Yale University Press, (1974).

[6] HEWITT, C., PLANNER: *A language for proving theorems in Robots,* Proceedings of IJCAI 1 (eds Walker & Norton), (1969).

[7] WINOGRAD, T., *Procedures as a representation for data in a computer program for understanding natural language,* Ph.D. Thesis, MIT, Cambridge Mass, (1971).

[8] SUSSMAN, G.J. & D.V. McDERMOTT, *Why Conniving is better than Planning,* MIT AI-memo 255A, (1972).

[9] QUILLIAN, M.R., *Semantic memory,* in: Semantic Information Processing (ed Minsky), MIT-Press, Cambridge Mass, (1968)

[10] BRACHMANN, R.J., *What's in a concept: Structural foundations for semantic networks,* BBN-report 3433 (1976).

[11] WOODS, W.A., *What's in a link, foundations of semantic networks,* in: Representation and understanding (eds Bobrow & Collins), Academic Press, (1975).

[12] FAHLMANN, S.E., *A system for representing and using real world knowledge,* MIT, AI-memo 331, (1975).

[13] FILLMORE, C.J., *The case for case,* in: Universals in linguistic theory (eds Bach & Harms), Holt, Chicago Ill.,(1968).

[14] SCHANK, R.C. & R.P. ABELSON, *Scripts, plans and knowledge,* Proceedings of IJCAI 4, (1975).

[15] RIEGER, C., *Conceptual memory,* in: Conceptual Information Processing (ed. Schank), North Holland Publishing Cy., Amsterdam (1975).

[16] MINSKY, M., *A framework for representing knowledge,* in: The psychology of computer vision (ed. Winston), McGraw-Hill, New York, (1975).

[17] BOBROW, D.G. & T. WINOGRAD, *An overview of KRL, a knowledge representation language,* Cognitive Science, V.1, No.1, (1977).

[18] BOBROW, T.G., T. WINOGRAD, et al, *Experience with KRL-0, One cycle of a knowledge representation language,* Proceedings of IJCAI 5, (1977).

[19] DAVIS, R. & J. KING, *An overview of production systems*, in: Machine Intelligence 8 (eds Elcock & Michie), Ellis Horwood, Ltd, (1977).

[20] LESSER, V.R. & L.D. ERMAN, *A retrospective view of the Hearsay-II architecture*, Proceedings of IJCAI 5, (1977).

[21] NEWELL, A. & H.A. SIMON, *Human problem solving*, Prentice Hall (1972).

[22] DAVIS, R., B.G. BUCHANAN & E.H. SHORTLIFFE, *Production rules as a representation for a knowledge based consultation program*, Artificial Intelligence 8, (1977).

DATATYPEN BEZIEN VANUIT DE RECURSIETHEORIE

J.A. BERGSTRA
Rijksuniversiteit Leiden

1. INLEIDING

In deze voordracht willen wij verslag uitbrengen van ons onderzoek van het afgelopen jaar naar de mogelijkheden om met gebruikmaking van het begrippenkader van de recursietheorie inzicht te krijgen in de logische achtergronden van datatypen, datastructuren en implementaties daarvan. De hoofdlijnen kunnen als volgt worden samengevat:

A) Na vervanging van 'functie' door 'proces' als centraal begrip in de (gewone) recursietheorie ontstaat een zogenaamde dynamische recursietheorie (DRT). Deze heeft veel gemeen met de gewone recursietheorie maar staat veel dichter bij 'information processing'.

B) Een natuurlijke subrecursieve variant van deze DRT krijgt men door slechts recursieve (in de zin van berekenbare) processen te beschouwen en processen tot elkaar te herleiden middels eindige automaten. In dit systeem wordt 'proces' een natuurlijke generalisatie van bijv. de Turing-tape. Tot op zekere hoogte kan men stellen: datatype $\equiv$ proces.

C) De identificatie datatype $\equiv$ proces gaat voorbij aan de mogelijkheid van datatypen met een niet-deterministisch karakter. Dit geeft aanleiding tot de definitie: Datatype $\equiv$ drietal $<L_I, L_O, C>$ waarin C een collectie van processen is met inputtaal L_I en outputtaal L_O.

D) 'Datatype' is een 'logisch' begrip. 'Datastructuur' daarentegen niet. In deze opzet wordt 'datastructuur' een per definitie niet precies te omschrijven begrip.

D) DRT vergelijkt geheugenstructuren algoritmisch, op dezelfde wijze als gewone RT informatieverzamelingen algoritmisch vergelijkt.

2. DYNAMISCHE RECURSIE-THEORIE

2.1. Processen

Processen communiceren met de buitenwereld W m.b.v. een inputtaal Σ_I en een outputtaal Σ_O. Proces P is aanvankelijk in toestand P_ε. De mogelijke toestanden van P worden genoteerd met P_σ, $\sigma \in \Sigma_I^*$. Om precies te zijn: P_σ is steeds een functie van $\Sigma_I * \Sigma_I^*$ naar Σ_O. P_ε heet de grafiek van P. $P_\sigma(\tau) = k$ betekent: Zij P in toestand P_σ en laat achtereenvolgens input $(\tau)_1, (\tau)_2 \ldots (\tau)_\ell$ ($\ell = \mathrm{lth}(\tau)$) aan P gegeven worden, dan geeft P als laatste output k. Kennelijk geldt: $P_\sigma(\tau) = P_\varepsilon(\sigma*\tau)$.
We willen nu berekeningen definiëren met processen als argumenten. De 'regel' voor het gebruik van een proces P is als volgt: Zij P in toestand P_σ, geven we P input $a \in \Sigma_I$ dan wordt output $P_\sigma(a)$ gegeven. De nieuwe toestand van P is $P_{\sigma*a}$. Het verschil met het gebruik van een orakel ($: \mathbb{N} \to \mathbb{N}$ of $\mathbb{N} \to \{0,1\}$, of $\Sigma^* \to \{0,1\}$) zit in de toestandsverandering. Bij een proces moet men liever denken aan bijv. de Turing-tape. De collectie van processen (in hun begintoestand) met inputalfabet Σ_I en outputalfabet Σ_O noteren we in het vervolg met $PR(\Sigma_I, \Sigma_O)$.

2.2. Automaten

Zij $\Sigma_I, \Sigma_O, \Sigma_I^1, \Sigma_O^1 \ldots \Sigma_I^n, \Sigma_O^n$ een n+1 - tal paren, T, van in- en outputalfabetten. Een automaat M van type T is een 3n+5 - tal

$$M = \langle S, s_0, \delta, (IN, OUT), (OUT^1, s^1, IN^1) \ldots (OUT^n, s^n, IN^n) \rangle.$$

Hierbij is:

S de verzameling van toestanden, $s_0 \in S$ de begintoestand
(Zij $K = S \times \Sigma_I \times \Sigma_O^1 \times \ldots \times \Sigma_O^n$),
$\delta: K \to S$ de transitiefunctie,
$OUT: K \to \Sigma_O \cup \{NO\}$ de outputfunctie,
$IN^i: K \to \Sigma_I^i \cup \{NO\}$ de inputfunctie voor het i-de proces,
IN het inputregister,
OUT^i het outputregister voor het i-de proces,
s^i de beginwaarde van OUT^i.

NO is een symbool $\notin \Sigma_O \cup \Sigma_I^1 \cup \ldots \cup \Sigma_I^n$. IN kan elementen uit Σ_I bevatten, de OUT^i zijn registers voor Σ_O^i.

Zij $P \in PR(\Sigma_I,\Sigma_O)$, $Q^i \in PR(\Sigma_I^i,\Sigma_O^i)$. Onderstaande beschrijving bevat een informele definitie van

$$P = M(\vec{Q})$$

(M simuleert P m.b.v. $Q^1 \ldots Q^n$).

We gaan uit van de Q^i in hun begintoestand en M in state s_0. Zodra een input a aan het systeem (lees: M) wordt gegeven krijgt IN de waarde a. Het argument $k \in K$ voor δ, OUT en de IN^i is steeds $<s,y,x_1,\ldots,x_n>$ waarbij s de toestand is, y de waarde is van IN en x_i de waarde van OUT^i. M kan nu op de bekende wijze van toestand naar toestand lopen. Wanneer OUT(k) = NO wordt er geen output gegeven, zo ook wordt er, wanneer $IN^i(k)$ = NO geen input aan het proces Q^i gegeven.

Is $OUT(k) = b \in \Sigma_O$ dan wordt b als output gegeven. Feitelijk resulteert dit in een verandering van IN. IN krijgt de waarde van de nieuwe input. Dit alles wordt geacht in een stap te geschieden. Op soortgelijke wijze resulteert $IN^i(k) = b \in \Sigma_I^i$ in een verandering van OUT^i. OUT^i wordt de output van Q^i op input b. (Hierbij verandert Q^i van toestand.)

2.3. $D_{\mathbb{N}}^{\leq}$

2.3.1. Zij PR = PR(ℕ,ℕ).

DEFINITIE. $P \leq \vec{Q}$ als er een automaat M is, van het goede type, zodat:
i) $P = M(\vec{Q})$;
ii) S_M is een recursieve deelverzameling van ℕ.
iii) δ, OUT en de IN^i zijn recursieve functies.

Het is eenvoudig in te zien dat $P \leq Q$ een transitieve relatie levert. We kunnen nu, juist als in de gewone recursietheorie, de door $\leq$ voortgebrachte equivalentierelatie uitdelen.

2.3.2. DEFINITIE. $D_{\mathbb{N}}^{\leq} = <PR/\equiv,\leq,+,0>$.

Hierbij is $\leq$ de op $PR/\equiv$ geïnduceerde partiële ordening. 0 is de laagste graad (bijv. van de constante processen, i.e. de processen welke altijd eenzelfde output geven). Noteren we de eq. klasse van P met d(P) (degree of P) dan is d(P) + d(Q) = d(P+Q) waarbij P + Q als volgt algoritmisch gedefinieerd kan worden:

$$P+Q_{s_1\ldots s_k}(a) = \begin{cases} \text{als } k \text{ even dan } 0 \\ \text{als } k \text{ oneven en } s_k = 0 \text{ dan } P(a) \\ \text{als } k \text{ oneven en } s_k \neq 0 \text{ dan } Q(a). \end{cases}$$

OPMERKING. P + Q is niet noodzakelijk het supremum van P en Q. Bijv. hoeft $P+P \equiv P$ niet te gelden. Het supremum $P \vee Q$ kunnen we als volgt definiëren:

$$(P \vee Q)_{s^1 * \sigma}(a) = \begin{cases} \text{als } s^1 = 0 \text{ dan } P(a) \\ \text{anders } Q(a). \end{cases}$$

2.3.3. Functionele processen

DEFINITIE. Het proces P heet functioneel wanneer voor zekere functie $f: \mathbb{N} \to \mathbb{N}$ geldt (voor alle σ)

$$P_\sigma(a) = f(a).$$

Notatie voor P: P^f.

FEIT. $P^f \leq P^g \iff f \leq g$ (in de gewone zin).

DEFINITIE. P heet EF ('eventually functional') als voor elke $h: \mathbb{N} \to \mathbb{N}$ er een k is zodat

$$P_{h(1)\ \ldots\ h(k)}$$

functioneel is.

FEIT. Stel $f < g$ dan is er een Q met:
i) $P^f < Q < P^g$;
ii) $Q \equiv R$ impliceert R niet EF.

CONCLUSIE. $D^{\leq}_{\mathbb{N}}$ levert een echte verfijning op van de gewone graden van onoplosbaarheid.

DEFINITIE. P is van minimale graad (MIN(P)) als
i) $0 < P$;
ii) $R \leq P \Rightarrow (R \equiv 0 \vee R \equiv P)$.

STELLING. *Er zijn processen van minimale graad.*

2.3.4. Open problemen

1) Hoeveel minimale graden zijn er?
2) Bestaat er een minimale graad beneden elke graad $\neq 0$?

3) Bevat $D^{\leq}_{\mathbb{N}}$ mod(P) voor gegeven P $\in$ PR een minimale graad ('minimal cover' van P)?

4) Is $d = d_1 + d_2$ een relatie die uitdrukbaar is in de eerste orde taal van de struktuur $\langle PR/\equiv, \leq, 0\rangle$?

2.3.5. Operatoren

DEFINITIE. Een operator F is een functie: PR $\to$ PR die, gezien als operatie op grafieken, continu is (in de producttopologie op $\mathbb{N} \to \mathbb{N}$, na codering van grafieken in zulke functies).

Operatoren kunnen gebruikt worden door automaten M^O die

i) voorzien zijn van registers voor processen;

ii) expliciete definitie van processen toestaan in de vorm van het volgende assignment:

$$Q_i \Leftarrow F(M^O_1(Q_{i_1} \ldots Q_{i_\ell}, \vec{F})).$$

Hierbij zijn de Q_j processen in het j-de register. Het effect van dit assignment is tweeledig:

- Q_i krijgt de waarde F(P) met $P = M^O_1(\vec{Q}_{i_k}, \vec{F})$,
- $Q_{i_1} \ldots Q_{i_\ell}$ krijgen de waarde $\lambda\sigma\cdot 0$ (deze processen zijn a.h.w. verbruikt).

Het is nu duidelijk hoe de relatie

$$P = M^O(\vec{Q}, \vec{F})$$

kan worden gedefinieerd.

Tenslotte definiëren we

$$F \leq \vec{G}, \vec{Q} \text{ voor operatoren } F, \vec{G}$$

door:

$$\exists M^O \; \forall P \; F(P) = M^O(P, \vec{Q}, \vec{G}).$$

Dit leidt tot een gradenstructuur op de collectie van operatoren.

We kunnen twee soorten van operatoren onderscheiden:

- operatoren recursief in een proces.
- operatoren niet recursief in enig proces.

Van de tweede categorie is de hieronder beschreven operator IN een (universeel) voorbeeld.

IN(P) wordt bepaald door de volgende eigenschappen:

$$\begin{cases} IN(P)_{\sigma * 0} = IN(P)_{\varepsilon} \\ IN(P)_{s_1+1,\ldots,s_k+1}(s+1) = P_{s_1,\ldots,s_k}(s). \end{cases}$$

(IN(P) wordt door 0 geïnitialiseerd, IN(P) $\vdash$ (IN-{0})* $\equiv$ P).

PROBLEMEN.

5) Zijn er operatoren van minimale graad?
6) Zit er tussen twee proces-graden een operator-graad?
7) Hoeveel operator-graden zijn er?

2.4. D_{Σ}^{FC} .

2.4.1. Een subrecursieve variant van $D_{\mathbb{N}}^{\leq}$. Zij Σ een aftelbaar oneindig alafbet.

DEFINITIE. RP_{Σ} is de collectie van processen P met input-taal Σ_I^P, outputtaal Σ_O^P zodat:

i) Σ_I^P, Σ_O^P eindig deel van Σ
ii) na codering van Σ_I^P en Σ_O^P in $\mathbb{N}$ is P recursief in de zin van 2.3.1 (P$\equiv$0).

De gedachte is dat voor de praktijk interessante processen uit PR zich in RP_{Σ} bevinden. (gerechtvaardigd zolang er geen nondeterminisme in het spel is).

DEFINITIE. Zij P, $\vec{Q}$ in RP_{Σ}

$$P \leq_{FC} \vec{Q}$$

als $P = M(\vec{Q})$ geldt voor een automaat M met eindig veel toestanden. (P is 'finite conrol reducible to' $\vec{Q}$).

2.4.2. DEFINITIE. $D_\Sigma^{FC} = RP_\Sigma/\equiv_{FC},\leq_{FC},+,0>$.

OPMERKINGEN.

i) + en $\vee$ gedragen zich als bij $D_{IN}^{\leq}$;

ii) operatoren kunnen op soortgelijke wijze behandeld worden;

iii) functionele processen hebben nu graad 0. Hun rol wordt overgenomen door de initialiseerbare processen (die in $D_{IN}^{\leq}$ alle functioneel zijn).

DEFINITIES.

INIT(P): P is initialiseerbaar ≡ voor zekere $a \in \Sigma_I^P$ geldt $\forall\sigma \in (\Sigma_I^P - \{a\})^*\ P_{\sigma * a} = P_\varepsilon$.

INIT(d): graad d bevat een initialiseerbaar proces.

MIN(d) : d is minimale graad ($\neq 0$).

MIN(P) : P is van minimale graad.

MON(P) : Er bestaat een ordening $\preccurlyeq$ op Σ_O^P zodat voor elke $\sigma,a,b: P_\sigma(a) \preccurlyeq P_{\sigma * a}(b)$ (P heet monotoon).

MON(d) : d bevat monotoon proces.

L(P) : P is lineair ≡ Σ_I^P bevat één element.

L(d) : d bevat een lineair proces.

RED(P) : P is recudibel ≡ er zijn Q en R met $Q < P$, $R < P$ en $P \equiv Q + R$.

2.4.3. RESULTATEN t.a.v. D_Σ^{FC}.

STELLING 1.

i) *Er is een maximale graad M;*

ii) *Bij elk proces is er een equivalent proces met twee inputsymbolen en twee outputsymbolen.*

BEWIJS (aanduiding).

i) M is de graad van de Turing-tape TT. TT beschrijven we informeel als volgt:

Σ_I = {READ, WRITE-0, WRITE-1, MOVE-LEFT, MOVE-RIGHT}

Σ_O = {0,1,BL,BLANK}

TT gedraagt zich als een Turing-tape die aanvankelijk geheel met BL gevuld is. Op READ komen antwoorden 0, 1 of BL. Op de andere instructies (inputs) volgt antwoord BLANK. Vanzelfsprekend is elk recursief proces met behulp van TT en 'finite control' te simuleren.

ii) Codeer input- en outputsymbolen in binaire strings.

STELLING 2. *Er is een proces van minimale graad.*

BEWIJS (aanduiding). Zij

Σ_I = {UP,DOWN,BOTTOM}

Σ_O = {0,1}

$$A_\sigma(k) = \begin{cases} 0 & \text{als } k = \text{BOTTOM en } \sigma = \text{UP}^t * \text{DOWN}^t \text{ voor zekere } t \in \mathbb{N} \\ 1 & \text{anders.} \end{cases}$$

Met enige moeite bewijst men nu:

i) $A \not\equiv_{FC} 0$;

ii) $B \leq_{FC} A^n \wedge B \not\equiv_{FC} 0 \Rightarrow A \leq_{FC} 0$.

De gedachte achter ii) is als volgt. Stel B voldoet aan de condities, B = M(A). Dan verschilt B op oneindig veel plaatsen van $M(P_1)$, P_1 het constant 1 proces. Dus zijn er willekeurig lange runs σ waarop $M(A)(\sigma)$ van $M(P_1)(\sigma)$ verschilt. Dit kunnen we nu gebruiken om een run te construeren waarbij vijf fasen te onderscheiden zijn: aanloop/repeterend gedeelte/ gedeelte waarin A de eerste DOWN instructie krijgt/volgend repeterend gedeelte/uitloopgedeelte waarin A een signaal (0) geeft dat op e.o.a. wijze door M(A) wordt doorgegeven.

Gegeven deze situatie is het niet moeilijk meer om $A \leq B$ aan te tonen.

STELLING 3.

i) TT *is niet van lineaire graad;*

ii) *er is een lineair proces* L *zodat* $TT \equiv L + L + L + L$.

BEWIJS.

i) Op betrekkelijk eenvoudige wijze is aan te tonen dat het predicaat $P_M^L(s,k) \iff$ 'M in state s stopt op $L_0 k$' beslisbaar is. Dit zou niet mogelijk zijn wanneer TT m.b.v. L simuleerbaar is. (Hier is L een willekeurig lineair proces.)

ii) Neem L als volgt: $\Sigma_I = \{0\}$, $\Sigma_O = \{0,1\}$. L geeft eerst een 0, dan een 1, dan weer een 0, dan twee 1-en, dan weer een 0, vervolgens drie 1-en enz.
Met enig werk blijkt het mogelijk te zijn om een counter, CO, m.b.v. L + L te simuleren. Daar CO + CO ≡ TT geldt TT ≡ (L+L) + (L+L).

STELLING 4.

i) *Een minimale graad is monotoon;*

ii) *Een initialiseerbare graad* > 0 *is niet monotoon;*

iii) *Een lineaire graad* > 0 *is niet monotoon;*

iv) *De som van monotone graden is weer monotoon.*

CORROLARIUM. M *is niet de som van (eindig veel) minimale graden.*

BEWIJSSCHETS.

i) Zij $P \not\equiv 0$. Er is een uniforme constructie voor $Q \leq P$ zodat $Q \not\equiv 0$ en Q monotoon.

ii)...iv) Berusten er op dat monotone processen tijdens een run slechts eindig veel keren niet-triviale informatie kunnen geven.

2.4.4. Open problemen (t.a.v. D_Σ^{FC})

8) Hoeveel minimale processen zijn er?

9) Is er onder elke graad $\not\equiv 0$ een minimale graad?

10) heeft elke graad $\not\equiv$ TT een 'minimal cover'?

11) Is er een paar graden A en B zonder infimum?

12) Is de elementaire theorie van D_Σ^{FC} beslisbaar?

13) Bestaan er niet-minimale irreducibele graden?

14) (Indien JA op 13) Is elke graad de som van eindig veel irreducibele graden? (Hoe is het met M in het bijzonder?)

2.5. D_Σ^{EFC}

Het is mogelijk om met een variant $\leq_{EFC}$, (extend finite control), van $\leq_{FC}$ processen met oneindige in- en outputalfabetten tot elkaar te herleiden. Hiertoe voert men een eindig aantal extra registers voor symbolen in. Het gebruik ervan is beperkt tot:

i) een registerinhoud als OUT-put of IN^i-put geven;

ii) inhouden van twee registers op gelijkheid testen;

iii) assignment van de inhoud van IN en de OUT^i aan de nieuwe registers.

Zij f nu een vaste bijectie: $\mathbb{N} \to \Sigma$; RP'_Σ is de collectie van processen P met:

i) Σ_I^P, Σ_O^P zijn recursieve deelverzamelingen van Σ;

ii) P is recursief.

(In i) en ii) moet men recursief lezen m.b.t. de identificatie van Σ en $\mathbb{N}$ midels f.)

FEIT 1. $P,Q \in RP_\Sigma$, $P \leq_{EFC} Q \Rightarrow P \leq_{FC} Q$.

FEIT 2. $P \in RP_\Sigma$, $Q \in RP'_\Sigma$, $Q \leq_{EFC} P \Rightarrow \exists R \in RP_\Sigma\ Q \equiv_{EFC} R$.

Uit deze feiten concluderen wij dat $\leq_{EFC}$ een zeer natuurlijke uitbreiding van $\leq_{FC}$ naar RP'_Σ is.

3. DATATYPEN

Duidelijk is dat vele datatypen behorende bij abstracte machinemodellen passen in D_Σ^{FC}. Dit deel is gewijd aan enkele opmerkingen over de beschrijving van andere datatypen.

3.1. Datatypen uit de complexiteitstheorie

Zij Δ een vast eindig alfabet. Een 'probleem' is een deelverzameling A van Δ^*.

Voor problemen A en B is het bekend (COOK) wat men moet verstaan onder: A is in polynomiale tijd reduceerbaar tot B (in de zin van Turing-reduceerbaar met B als orakel). Notatie $A \leq_T^P B$.

We willen dit nu in termen van processen beschrijven. Allereerst moeten we bij A,B passende processen $\tilde{A},\tilde{B}$ vinden.

DEFINITIE. $\tilde{A}$. $\Sigma_I^{\tilde{A}} = \Delta \cup \{\#\}$

$\Sigma_O^{\tilde{A}} = \{0,1,BL\}$.

Voorts:

$$\tilde{A}_{\sigma * \#} = \tilde{A}_{\varepsilon},\ \tilde{A}_{\sigma}(k) = BL \text{ als } k \neq \#$$

$$\tilde{A}_{\sigma}(\#)_{\sigma \in \Delta^*} = \begin{cases} 0 & \text{als } \sigma \in A \\ 1 & \text{anders.} \end{cases}$$

Informeel gesproken: $\tilde{A}$ is een initialiseerbare karakteristieke functie van A.

STELLING. *Er bestaat een proces* P *in* RP_Σ *met de volgende eigenschap:*

$$A \leq_T^P B \iff \tilde{A} \leq_{FC} \tilde{B},\ P.$$

BEWIJSSCHETS. We geven een summiere beschrijving van P. P leest eerst een rij $\sigma \in \{0,1\}^*$ in die de exponent exp bepaalt, gevolgd door een #. Daarna is P in toestand $P_{\sigma_{exp}}$. Vervolgens leest P weer een string $\tau \in \{0,1\}^*$ gevolgd door #. Deze string staat nu op een tape waarover een leeskop kan heen- en weerlopen en lezen. Zij ℓ de lengte van τ. Nu gedraagt P zich als een Turing-machine met twee tapes, de zojuist beschreven leestape met het input-woord τ, en een gewone lees- en schrijftape, oorspronkelijk overal gevuld met BL. Echter, wanneer meer dan ℓ^{exp} instructies zijn uitgevoerd levert P nog slechts output BLANK. Wordt # ingelezen dan keert P terug naar toestand $P_{\sigma_{exp}}$.

OPMERKING. Een soortgelijk resultaat is te vinden voor LOGSPACE reducibility.

PROBLEEM.

15) Bestaat er P zodat $\tilde{A} \leq_{FC} \tilde{B}, P \iff \tilde{A}$ reduceerbaar tot $\tilde{B}$ in lineaire (kwadratische) tijd?

3.2. Algebraïsche datatypen

Beschouw de algebraïsche structuur $\langle \mathbb{N}, SUC, 0, = \rangle$. We willen er relevante processen bij definiëren, analoog aan het effect van de ALGOL-60 declaratie integer i,j,k; Deze declaratie roept in onze situatie een proces P in het leven met (informeel genoteerd)

$$\Sigma_I^P = \{i \Leftarrow 0,\ j \Leftarrow 0,\ k \Leftarrow 0,\ i \Leftarrow SUC(i),\ j \Leftarrow SUC(j),\ k \Leftarrow SUC(k),\ i = j?\}$$
$$\Sigma_O^P = \{0,1,BL\}.$$

Hierbij komt 0 of 1 als output op i = j? inputs en BL als output in de andere gevallen. Het is duidelijk hoe P zich dient te gedragen.

Voorts is duidelijk dat de overgang van een algebraïsche structuur naar een bijbehorend proces van heel wat parameters afhangt. Binnen D_Σ^{FC} kunnen de resultaten van verschillende keuzen kwalitatief met elkaar worden vergeleken.

3.3. Niet-deterministische datatypen

De identificatie datatype ≡ proces houdt geen rekening met niet-deterministische verschijnselen. Vandaar onderstaande algemene definitie.

DEFINITIE. Een datatype is een drietal

$$D = \langle \Sigma_I, \Sigma_O, C_D \rangle,$$

waarbij $C_D \subseteq PR(\Sigma_I, \Sigma_O)$ niet leeg. (Meestal bevat C_D slechts recursieve processen.)

D heet deterministisch als C_D een singleton is. De elementen van C_D zijn implementaties van D.

DEFINITIE. D_1 is relatief implementeerbaar t.o.v. D_2 ($D_1 \leq D_2$) wanneer er een (extended) finite control machine $M_{(E)FC}$ is zodat geldt:

$$\forall P \in C_{D_2}[M_{(E)FC}(P) \in C_{D_1}].$$

OPMERKING. Kiest men de Σ_I, Σ_O weer deel van een vaste Σ dan genereert ≤ op de zo verkregen collectie van niet-deterministische datatypen een structuur soortgelijk aan de Medvedev lattice [1].

Het is niet moeilijk om voorbeelden D_1 en D_2 te vinden met D_1 deterministisch en D_2 niet-deterministisch zodat $D_1 \leq D_2$ geldt.
Bijvoorbeeld: D_2 registreert eindige deelverzamelingen van een gegeven verzameling E en heeft ter beschikking de operaties INSERT(V,e), DELETE(V,e), EMPTY(V)? en de niet-deterministische instructie $x_E \Leftarrow SEL(V)$ die tot gevolg heeft dat x_E de waarde krijgt van enig element in V (mits V niet leeg). Voor D_1 kan men nu nemen: weer de eindige deelverzamelingen

van E ditmaal zonder SEL maar met UNION en INTERSECTION. Deze laatste twee kunnen m.b.v. de in D_2 beschikbare operaties worden gerealiseerd op uniforme wijze (i.e. onafhankelijk van de preciese implementatie van SEL).

PROBLEEM.

16) Geef een uitwerking van bovenstaand voorbeeld (o.i.d.) met: een axiomatische beschrijving van D_1 en van D_2 en een correctheidsbewijs van de (relatieve) implementatie.

4. AUTONOME PROCESSEN

In een rekenmachine kan een functie TIME gerealiseerd worden. Deze hangt van 'de tijd' af, meer dan van de inputhistorie. Er is alle reden om TIME als een (autonoom) datatype op te vatten. In elk geval kunnen we TIME zien als een autonoom proces. Deze sectie is gewijd aan het precies invoeren van autonome processen en recursie er op. Intuïtief gesproken krijgt een autonoom proces zo nu en dan een input en geeft het zo nu en dan een output. Wanneer we nu gediscretiseerde tijd $t_0,t_1,t_2,t_3,\ldots$ aannemen dan is het echter natuurlijk om te veronderstellen dat een AP elk tijdstip input krijgt en output geeft. We kunnen een speciaal symbool BLANK reserveren om lege (i.e. niet bedoelde) in- en output aan te duiden. Het blijkt dat PR en RP_Σ nu als collecties van autonome (recursieve autonome) processen kunnen worden opgevat. We zullen ze noteren met APR en ARP_Σ. Op verrassend eenvoudige wijze kunnen we reduceerbaarheid op APR en ARP_Σ definiëren. Namelijk door de definitie van automaat M zodanig te wijzigen dat de mogelijkheden OUT(k) = NO en $IN^i(k)$ = NO komen te vervallen (of, wat hetzelfde is, NO te vervangen door BLANK). Bovenstaande geeft aanleiding tot het vormen van de structuren

$$AD_\omega^{\leq} = \langle ARP/\equiv,\leq,+,0\rangle$$
$$AD_\Sigma^{FC} = \langle ARP_\Sigma/\equiv_{FC},\leq_{FC},+,0\rangle.$$

Het is onze overtuiging dat zich binnen $AD_\omega^{\leq}$ en AD_Σ^{FC} belangwekkende zaken afspelen.

PROBLEMEN.

17) Structuur van $AD_\omega^{\leq}$.

18) Structuur van AD_Σ^{FC}.

19) Vind natuurlijke inbeddingen van D_{Σ}^{FC} in AD_{Σ}^{FC}.

20) Vind binnen AD_{Σ}^{FC} geheugenmechanismen welke intrinsiek van autonome aard zijn.

LITERATUUR

[1] HARTLEY ROGERS Jr., *The theory of recursive functions and effective computability*, McGraw Hill.

PROCEDURELE DATASTRUCTUREN

L.G.L.T. MEERTENS
Mathematisch Centrum

1. INLEIDING

Een datastructuur is een methode om gegevens in een automaat te representeren, zodat bepaalde operaties op die gegevens redelijk efficient kunnen worden uitgevoerd door manipulaties met de representatie.

Bovenstaande definitie legt, terecht, geen beperkingen op aan de mogelijke representatiemethoden. Toch bestaat de onwillekeurige neiging om daarbij te denken aan het opslaan van de gegevens in geheugencellen, waarbij door "pointers" een toegangsstructuur wordt aangebracht. De bedoeling van het onderhavige verhaal is aandacht te vragen voor de mogelijkheid gegevens te representeren met behulp van procedures.

Procedurele datastructuren bieden voor- en nadelen. Zoals altijd bij de keuze van een datastructuur, moeten deze voor een bepaalde toepassing tegen elkaar worden afgewogen. Onder de voordelen vallen te noemen: een eenvoudige wijze om sommige objecten te representeren die bij "conventionele" representatie een onbeperkt geheugenbeslag vergen; de mogelijkheid bewerkingen min of meer automatisch uit te stellen tot ze werkelijk nodig zijn; de mogelijkheid een mengeling te scheppen met andere representaties op zo'n manier dat eerder gedefinieerde operaties niet aangepast behoeven te worden.

Hetzelfde onderwerp is behandeld in REYNOLDS [1]; hier zijn wat meer, en ook uiteenlopender, voorbeelden gegeven. Bovendien wordt bij de behandeling, zij het oppervlakkig, aandacht geschonken aan een zekere formele gelijkenis met conventionele datastructuren, en wordt meer nadruk gelegd op enkele subtiele kwesties.

2. CONVENTIES

Als voertuig om enkele algoritmische gedachten over te dragen wordt hier een variatie op ALGOL 68 gebruikt. De afwijkingen bestaan hoofdzakelijk uit een opheffing van de scope-beperkingen - wezenlijk voor het hier beschrevene - en een voornamelijk uit gemakzucht ingevoerde afkorting om uitstel van executie aan te geven.

Een valkuil voor beginnende beoefenaren van de ALGOL-68-kunde wordt gevormd door de scope-beperkingen. De ontdekking dat procedures een formeel gelijkberechtigd data-type zijn, en dat dus het resultaat van een bewerking een procedure kan zijn, opent geheel nieuwe perspectieven, die pas bij nadere bestudering van de taal weer geheel worden afgesloten. Zo lijkt het een ogenblik dat het mogelijk is een operator te maken die bij twee functies f en g het compositum $f \circ g$ oplevert, en wel als volgt:

```
mode fun = proc (real) real;

op o = (fun f, g) fun:
       (real x) real: f (g (x)).
```

Helaas, deze vlieger gaat niet op, op grond van een aantal zinsneden in het definierend rapport, alle van de vorm "it is required that ... be not newer in scope than ...". Deze beperkingen zijn, in tegenstelling tot een aantal andere beperkingen die het voorschrijven van een zinledige bewerking onderscheppen, louter ingevoerd om een bepaalde implementatietechniek mogelijk te maken. Als al deze zinsneden worden geschrapt, blijft een goed gedefinieerde taal over, die alleen afwijkt van ALGOL 68 doordat de betekenis van programma's in meer gevallen gedefinieerd is (maar nooit anders gedefinieerd). Omdat het voor ons doel juist zeer goed uitkomt naar believen nieuwe procedures te construeren, wordt in het vervolg dan ook aangenomen dat alle scope-beperkingen zijn opgeheven.

Een andere manier om deze restricties te omzeilen is de taaluitbreiding met "partial parametrization", als beschreven in LINDSEY [2]. Dit voorstel suggereert meteen een implementatietechniek die ook bij de hier gebruikte aanpak toepasbaar is. In enkele woorden geschetst komt deze hierop neer dat de waarde van de bevroren parameters (in bovenstaand voorbeeld f en g) "ergens" (bijv. op de "heap") wordt weggeborgen, en dat de representatie van

de procedure behalve een adres van de code ook een pointer naar dat waardenpakket bevat. Een dergelijke representatie staat wel bekend als een "closure". Om de implementatie algoritmisch weer te geven, kunnen we iets schrijven als:

```
mode fun = union (proc (real) real, closure);

mode closure = struct (fun f, g,
                       proc (fun, fun, real) real a);

op call = (fun p, real x) real:
          case p in
               (proc (real) real f): f (x),
               (closure c): (a of c) (f of c, g of c, x)
          esac;

proc o = (fun f, g, real x) real: f call (g call x).
```

waarna het effect van

```
fun lnsin = ln o sin; lnsin (11 / 7)
```

ongeveer overeenkomt met

```
closure lnsin = (ln, sin, o); lnsin call (11 / 7).
```

We nemen aan dat wat hier is uitgeschreven, of iets met hetzelfde effect, impliciet geschiedt. (In practisch voorkomende gevallen van niet zeer grote ingewikkeldheid is het overigens nog wel doenlijk een programma in de hier gebezigde ALGOL-68-varieteit met de hand om te zetten in standaard-ALGOL-68, maar naarmate het aantal verschillende toepassingen stijgt ontstaan meer soorten closures, die in een grote union vereend moeten worden en daar in de operatie *call* weer met conformity-clauses uitgepulkt moeten worden - een weinig appetijtelijke bezigheid.) In zekere zin vormen de closures impliciete conventionele datastructuren, waarbij de correcte toepassing van operaties op de velden gewaarborgd wordt door de ontstaansgeschiedenis, zo-

dat conformity-tests overbodig zijn.

Verwant is ook het own-begrip uit ALGOL 60. Zonder scope-beperking kunnen we bijv. schrijven:

```
proc int fac = (int n:= -1, f;
                int: f:= if (n+:=1) = 0 then 1 else n * f fi).
```

Achtereenvolgende aanroepen van *fac* leveren 0!, 1!, 2!, ... In tegenstelling tot de situatie bij ALGOL 60 is de semantiek hier goed gedefinieerd; bovendien is initialisatie mogelijk. Dit voorbeeld laat overigens zien dat er ook een band is met de class uit SIMULA 67.

Zij *m* de mode *proc (x) y*, en zij *e* een uitdrukking van de mode *m*. Dan is *e'* = *(x a) y: (e) (a)* eveneens van de mode *m*, en voor iedere waarde van *a* is *(e') (a)* gelijk aan *(e) (a)*. Toch behoeven *e* en *e'* niet hetzelfde gedrag te vertonen; het is namelijk denkbaar dat de elaboratie (executie) van *e* bepaalde (neven-)effecten teweeg brengt, wat bij *e'* zeker niet het geval is. Naar zal blijken, is er nogal eens behoefte aan dat (op zich gewenste) effecten van *e* pas bij de aanroep in werking treden. Daarom wordt een afkorting ingevoerd: in een context waar een uitdrukking van de mode *m* vereist is, zal *: e* de uitdrukking *e'* aanduiden (met dien verstande dat voor de parameter *a* zonodig een andere identifier gekozen wordt, om niet met de identifiers in *e* in conflict te geraken). Bij uitbreiding zal voor de parameterloze-proceduremode *proc y* de afkorting *: e* de uitdrukking *y: e* beduiden.

Voorbeeld:

```
fun sc = if random < .5 then sin else cos fi;

fun sc2 = : if random < .5 then sin else cos fi.
```

Na de eerste declaratie is *sc*, afhankelijk van de uitkomst van *random*, of verder *sin*, of verder *cos*. Bij de tweede declaratie wordt *random* nog niet aangeroepen; verdere aanroepen *sc2 (x)* leveren, in een grillige afwisseling, *sin (x)* dan wel *cos (x)* als waarde op.

3. RIJEN

Een rij *s* kan wiskundig beschouwd worden als een functie *s*: N -> V, waarbij N = {0, 1, 2, ...} voor de verzameling der natuurlijke getallen staat, en V een willekeurige verzameling is. Tot nader order zal in de voorbeelden steeds de keuze N voor V dienst doen. De representatie die zich voor zulke rijen natuurlijk aandient, wordt gegeven door de declaratie

```
mode seq1 = proc (int) int.
```

Een voorbeeld, voor de rij (1, 2, 3, ...), wordt geboden door

```
seq1 pnat = (int i) int: i + 1.
```

Er is een complicatie: wie of wat garandeert ons dat de procedure *s* "repliceerbaar" is, met andere woorden, dat de uitkomst van een aanroep *s (i)* onafhankelijk is van de voorgeschiedenis? Om ook voor het algemene geval, waarbij *s* een geheugen heeft, een rij met *s* te verbinden, identificeren we *s* met de successieve parameterwaarden in aanroepen van *YIELD*, voortgebracht door het oneindige proces

```
for i from 0 do YIELD (s (i)) od.
```

We willen nu eerst eens proberen operaties *head*, *tail* en *join* te definieren, waarbij, voor $s = (s_0, s_1, s_2, \ldots)$, *head* $s = s_0$, *tail* $s = (s_1, s_2, \ldots)$ en *(head s) join (tail s) = s*:

```
op head = (seq1 s) int: s(0);

op tail = (seq1 s) seq1: (int i) int: s (i + 1);

op join = (int h, seq1 t) seq1:
                (int i) int: if i = 0 then h else t (i - 1) fi.
```

(Merk op dat hier de aangekondigde opheffing van de scope-beperking ge-

bruikt wordt.)

Deze operaties kunnen nu weer gebruikt worden om nieuwe operaties te helpen bouwen, als in

```
op + = (seq1 s1, s2) seq1 :
          : (head s1 + head s2) join (tail s1 + tail s2);

op constseq = (int c) seq1:
                  : c join (constseq c).
```

Het uitstel van executie is hier essentieel; zonder zulk uitstel zouden bijv. bij de rij-optelling bij voorbaat alle elementen uit beide rijen paarsgewijze samengenomen worden, wat niet alleen voorbarig is, maar bovendien niet eindigt. De definitie van *constseq* kan natuurlijk veel "efficienter" geschreven worden als

```
op constseq = (int c) seq1: (int i) int: c,
```

maar de eerdere definitie is wat illustratiever. Voortbouwend kunnen we komen tot

```
op cum = (seq1 s) seq1:
          : (constseq head s) + (0 join (cum tail s)),
```

hetgeen de gecumuleerde rij $(s_0, s_0 + s_1, s_0 + s_1 + s_2, \ldots)$ oplevert. Nu beschrijft *cum constseq 1* dezelfde rij als *pnat*. Een ander lichaam voor *cum* zou kunnen zijn:

```
: (head s) join ((constseq head s) + (cum tail s)),
```

maar dit werkt alleen goed voor repliceerbare *s*.

Een zekere overeenkomst met lineaire lijsten, benadrukt door de schrijfwijze met operaties als *head*, *tail* en *join*, zal duidelijk zijn. Zelfs is *constseq c* te vergelijken met een circulaire lijst, een lijst die in zijn eigen staart bijt. Deze overeenkomst brengt ons ertoe na te gaan of deze overeenkomst nog sterker tot uitdrukking kan worden gebracht, door naar analogie met

```
mode list = ref struct (int head, list tail)
```

rijen te representeren als

```
mode seq2 = proc struct (int head, seq2 tail).
```

Ook hier speelt de kwestie van repliceerbaarheid. De door een *seq2 s* voorgestelde rij is die, voortgebracht door het proces

```
seq2 v:= s;
do struct (int head, seq2 tail) ht:= v;
   YIELD (head of ht); v:= tail of vt
od.
```

De operaties *head* en *tail* lijken bij deze representatie triviaal:

```
op head = (seq1 s) int: head of s;

op tail = (seq1 s) seq1: tail of s.
```

Toch schuilt hier een addertje, zoals aan het licht zal treden wanneer we de grassprietjes uiteen buigen. Eerst echter nog een definitie van *join*, die op zich probleemloos is:

```
op join = (int h, seq2 t) seq2: : (h, t).
```

Het is niet moeilijk in te zien dat *head (h join t) = h* en *tail (h join t) = t*. Slechter is het gesteld met de eis *(head s) join (tail s) = s*, waaraan voor niet repliceerbare *s* in het algemeen niet voldaan wordt. Het blijkt helaas dat dit ook niet lukt met andere definities.

Is het mogelijk rijen in de representatie *seq1* over te voeren in de representatie *seq2*, en omgekeerd? Voor de hand liggende definities zijn:

```
op s1tos2 = (seq1 s1) seq2: : (head s1) join (s1tos2 tail s1);

op s2tos1 = (seq2 s2) seq1: : (head s2) join (s2tos1 tail s2).
```

Het zal duidelijk zijn dat repliceerbaarheid van *s1* resp. *s2* ook de repliceerbaarheid van *s1tos2 s1* resp. *s2tos1 s2* inhoudt. Met iets meer moeite valt in te zien dat ook in het algemene geval *s1tos2 s1*, als proces beschouwd, ekwivalent is met *s1*. Het blijkt echter niet zo, dat in het algemeen *s2tos1 s2* = *s2*. Door een ingewikkelder definitie van de operatie *s2tos1* valt dit wel te bereiken. Het blijkt bovendien mogelijk die operatie zo te definieren dat *s2tos1 s2* repliceerbaar is, zelfs als *s2* zelf dat niet is!

Die oplossing zal hier niet worden geschetst, wel een methode om uit een niet repliceerbare *seq1 s* een repliceerbare versie te maken.

```
op rep = (seq1 s) seq1:
        begin seq1 mem:= (int j) int: -1;
              (int i) int:
              begin int si:= mem (i);
                    if si = -1
                    then si:= s (i);
                         seq1 mem0 = mem;
                         mem:= (int j) int:
                               if j = i then si else mem0 (j) fi
                    fi;
                    si
              end
        end.
```

In deze definitie houdt de variabele *mem* bij voor welke posities de rijwaarde al is opgevraagd, en wat die rijwaarde was. Is de vraag nog niet eerder gesteld, dan levert de in *mem* onthouden procedure het onnatuurlijke resultaat -1 af. Het trekken van een copie via *mem0* is noodzakelijk; anders zou, na de allereerste keer, de aanroep van *mem* niet meer eindigen.

We zullen nu enige tijd stil blijven staan bij het algemene, niet repliceerbare geval. Het is duidelijk dat de essentie dan gevat wordt door het corresponderende oneindige proces. Daarom bekijken we

```
mode seq = proc int.
```

Het proces is nu, voor een *seq s*,

do YIELD (s) od.

Een voorbeeld van een rij in deze representatie zijn we in de inleiding al tegengekomen, in de gedaante van de procedure *fac*. Het zal duidelijk zijn dat het begrip "repliceerbaarheid" hier zijn zin verliest. Na de teleurstellende ervaring bij *seq2*, zouden we verwachten dat operaties *head*, *tail* en *join*, met *(head s) join (tail s) = s*, hier nog veel minder te realiseren zijn. Er is echter een oplossing die een heel eind in de goede richting komt. Hierbij nemen *head* en *tail* de curieuze vorm aan:

op head = (seq s) int: s;

op tail = (seq s) seq: s.

De operatie *head* is wel te begrijpen, maar *tail* vereist enige uitleg. Een lichaam als *(s, s)* zou meer voor de hand liggen. De gedachte achter bovenstaande definitie is, dat we erop rekenen dat een operatie *tail s* steeds vooraf gegaan zal worden door *head s*, zodat de rij al "opgeschoven" is. De operatie *join* is iets lastiger te construeren:

```
op join = (proc int h, proc seq t) seq:
          begin seq v:= : (int hh = h; v:= t; h);
                : v
          end.
```

Deze definitie behoeft enige toelichting. Bij de representatie *seq* moeten we erg op onze tellen passen: door bij wijze van spreken alleen al naar een *s* te kijken, verandert deze. Uitstel van executie wordt dan essentieel. Om die reden is bij de formele parameters van *join* een extra *proc* geplaatst, wat wat meer armslag geeft bij het maken van operanden. We hebben nu

(: head s) join (: tail s) = s.

Gebruik makend van deze definities construeren we:

```
op + = (seq s1, s2) seq:
       (: (head s1 + head s2)) join (: (tail s1 + tail s2));

op constseq = (int c) seq:
              (: c) join (: constseq c);

op cum = (seq s) seq:
         (: constseq head s) + ((: 0) join (: cum tail s)).
```

(Voor de ALGOL-68-kenner: eigenlijk is de definitie van + niet toegelaten in hetzelfde bereik als de standaardoperatie *proc (int, int) int +*, wegens het "firmly related" zijn van de operand-modes.)

De overeenkomst met de definitie van *cum* voor de *seq1*-representatie is duidelijk. Overigens kan het wel eenvoudiger:

```
op cum = (seq s) seq:
         (int c:= 0; : c +:= s).
```

Dit laatste voorbeeld is illustratief voor de flexibiliteit van procedurele datastructuren. Voor de geinteresseerde lezer nog een klein puzzeltje. Gegeven

```
op y = (int p, seq) seq:
       : if int h = r; h mod p = 0 then y (p, r) else h fi;

op z = (seq s) seq:
       (int h; (: h:= head s) join (: z  (h y tail s))),
```

en de vraag is: welke rij wordt berekend door

```
z cum ((: 2) join (: constseq 1))?
```

4. KETTINGBREUKEN

De voorstelling van reele getallen als kettingbreuken, als in

$$\pi = 3 + 1 / (7 + 1 / (15 + 1 / (1 + 1 / (292 + \ldots)))),$$

leent zich bij uitstek voor toepassing van procedurele datastructuren. Een uitgebreidere behandeling dan hier wordt gegeven is te vinden in BEELER & al. [3], item 101. Het navolgende is goeddeels daaraan ontleend.

De schrijfwijze $x = p + q / x'$ laat zien hoe kettingbreuken geidentificeerd kunnen worden met rijen van paren gehele getallen. Bij de "gewone" kettingbreuken is steeds de waarde van q gelijk aan 1, maar het is nuttig ook andere waarden toe te laten.

Iedere representatie kent operaties die daarbij speciaal goedkoop zijn. Bij de decimale voorstelling is vermenigvuldiging of deling door 10 bijna gratis, maar bepaling van de omgekeerde duur. Bij de voorstelling van rationale getallen als breuk is bepaling van de omgekeerde weer goedkoop. Dat geldt ook voor kettingbreuken, want $1/x = 0 + 1 / x$. Het gebruikelijke argument voor decimale of binaire aritmetiek is het gemak van optelling, vermenigvuldiging en dergelijke. Bij rationale getallen hebben teller en noemer vaak tezeer de neiging om bij deze operaties te groeien om rationale aritmetiek praktisch bruikbaar te doen zijn.

Het is niet algemeen bekend dat aritmetiek op kettingbreuken een praktische mogelijkheid is. Met behulp van procedurele datastructuren is dit echter haast een peuleschil. Voordelen die daarbij door geen der andere representaties geboden worden, zijn de onbegrensde aritmetische precisie, zonder afronding of wat dan ook, waarbij de "termen" van het resultaat een voor een afgeleverd worden, in een eindeloos proces, totdat bijv. iemand die tevreden is met de bereikte precisie het proces de nek omdraait. Het tevoren specificeren van de precisie is dan ook overbodig.

Voordat we het over die aritmetiek zullen hebben, gaan we eerst als vingeroefening π berekenen. We definieren de procedurele representatie van een kettingbreuk door

```
mode pq = struct (int p, q),
mode cf = proc pq.
```

Beschouw nu de uitdrukking z(x) = (ax + b) / (cx + d). Dit zullen we afkorten tot z(x) = (a b / c d) (x). In het bijzonder geldt: (1 0 / 0 1) (x) = x. Als we stellen: x = p + q / x', dan leren enkele elementaire algebraische bewerkingen ons dat z(x) = (ap+b aq / cp+d cq) (x'). Hierbij zien we hoe z(x) een term uit de ontwikkeling van x kan "absorberen". Als we z(x) zelf als kettingbreuk zien, met z(x) = s + t / z'(x), dan vinden we

z'(x) = (tc td / a-sc b-sd) (x).

Op deze wijze kunnen we termen van z(x) "afstoten".

Dit kunnen we gebruiken om een algemene in een gewone kettingbreuk om te zetten. Voor een breuk z = s + 1 / z' geldt immers: s = entier(z) (aangenomen dat z' > 1). Aangezien, voor c > 0, d ≥ 0, z = z(x) monotoon is in x voor x > 0, geldt dat z in het interval (a / c, b / d) ligt, en dus, mits ook a, b ≥ 0, s in het interval [a ÷ c, b ÷ d] ligt. Zolang dit interval nog niet tot een punt is ingekrompen, weten we s niet en moeten we termen van x blijven absorberen. Vroeg of laat raakt s echter bekend en kan er een term worden afgestoten. Dit alles leidt tot

```
op reg = (cf x) cf:
        begin int a:= 1, b:= 0, c:= 0, d:= 1;
              : begin while a ÷ c ≠ b ÷ d
                      do pq h = x; int p = p of h, q = q of h;
                         (a, b, c, d):=
                         (a * p + b, a * q, c * p + d, c * q)
                      od;
                      int s = a ÷ c;
                      (a, b, c, d):= (c, d, a - s * c, b - s * d);
                      (s, 1)
                end
        end.
```

Op haast dezelfde wijze kunnen we een kettingbreuk in een decimale ontwikkeling omzetten. Nu moeten we echter z = s + z' / 10 gebruiken, maar opnieuw is s = entier(z):

```
op dec = (cf x) seq:
        begin int a:= 1, b:= 0, c:= 0, d:= 1;
              : begin while a ÷ c ≠ b ÷ d
                      do pq h = x; int p = p of h, q = q of h;
                         (a, b, c, d):=
                         (a * p + b, a * q, c * p + d, c * q)
                      od;
                      int s = a ÷ c;
                      (a, b, c, d):=
                      (10 * (a - s * c), 10 * (b - s * d), c, d);
                      s
                end
        end.
```

Nu kunnen we uit de kettingbreukontwikkeling

$$\arctan x = 0 + x \,/\, (1 + x^2 \,/\, (3 + 4x^2 \,/\, (5 + \ldots))),$$

en dus

$$\pi = 0 + 4 \,/\, (1 + 1 \,/\, (3 + 4 \,/\, (5 + 9 \,/\, (7 + 16 \,/\, (9 + \ldots)))))$$

de decimale ontwikkeling van π vinden, met het proces:

```
begin cf pi = (int k:= -1;
                  : if (k+:=1) = 0 then (0, 4) else (2*k-1, k*k) fi
               );
      seq dpi = dec pi;
      do YIELD (dpi) od
end.
```

Implementatie in het primitieve taaltje *dc* uit het UNIX-systeem van de PDP 11 leverde in 1054 seconden 555 decimalen op, waarop het proces, vermoedelijk wegens geheugenruimtegebrek, ten gronde ging. De 35 decimalen die Ludolf van Ceulen in zijn aan π gewijde leven wist te berekenen waren er binnen 8 seconden uit. De orde van het proces zal wel iets als $n^2 \log n$ zijn.

Wat de aritmetiek betreft moge verder een schets volstaan. We beschouwen de uitdrukking

$$z(x, y) = (axy + bx + cy + d) / (exy + fx + gy + h).$$

Door geschikte keuzen voor a tot en met h kunnen hieruit onder meer uitdrukkingen voor $x + y$, $x - y$, xy en x / y gevormd worden, zodat deze gebruikelijke aritmetische operaties onder een hoedje te vangen zijn. Net als boven kunnen termen van x en y geabsorbeerd worden en termen van $z(x, y)$ afgestoten. Voor een algemene correcte behandeling, vooral als het resultaat in standaardvorm moet zijn, is wel enige zorg vereist, maar het is zeker doenlijk.

Een aardig idee is het volgende. De methode van Newton, toegepast op het vinden van algebraische wortels (niet noodzakelijk vierkantswortels), vervangt een benadering door een betere benadering. Als we dit zien als een transformatie op een kettingbreuk, is het duidelijk dat het invoeren van incorrecte termen niet helpt. Maar tegen de tijd dat we aan de incorrecte termen toezijn, zijn de corresponderende correcte termen allang geproduceerd! Het aantal correcte uitvoertermen is namelijk ongeveer het dubbele van de invoertermen. Door nu de uitvoer terug te koppelen en in plaats van de invoer te stellen, wordt meteen het juiste resultaat geproduceerd. Alleen moet er nog even voor een goede start gezorgd worden.

Een probleem dat nog niet goed onder de knie is, is bijv. hoe met kettingbreuken een grootheid als sin cos 1 te berekenen. Het probleem hierbij is dat de termen in de kettingbreukontwikkeling van sin x een functie zijn van x, en dus op hun beurt zelf weer kettingbreuken worden.

5. FILES

Van de eigenschappen van het UNIX-systeem die het een goede naam onder zijn gebruikers bezorgd hebben, is de uniforme file-behandeling misschien wel het belangrijkste. Dit kan, in termen van procedurele datastructuren, bereikt worden door als representatie te nemen:

```
mode file = struct (proc (char) void put, proc char get).
```

Deze techniek is ook beschreven in STOY & STRACHEY [4]. Om van bijv. een string een read-only file te maken, kunnen we een operatie definieren als

```
op stof = (string s) file:
          begin int i = lwb s - 1;
                (: undefined,
                 : if (i+:=1) > upb s then eof else s[i] fi)
          end.
```

Van de *standout* file uit de ALGOL-68-transput wordt eenvoudig een write-only file gemaakt door

```
file so = ((char c) void: put (standout, c), : undefined).
```

De beschrijving van het ALGOL-68-transputsysteem zelf kan wat realistischer op implementatie worden afgestemd door een ALGOL-68-*file*, juist op die punten waar de onderliggende representatie een rol zou gaan spelen, te beschrijven met behulp van procedures voor de operaties daarop. Dit is nader uitgewerkt in VAN VLIET [5]. Dit geeft niet alleen een flexibeler model, maar bovendien een efficientere implementatie, doordat geen tijd hoeft te worden besteed aan het uitzoeken van welk van de uiteenlopende mogelijkheden het geval is.

6. BESLUIT

Voor procedures als datastructuren zijn hier slechts enkele toepassingen geschetst. Naast de mogelijkheid van andere toepassingen op het gebied van de numerieke wiskunde of de systeemprogrammering, mag de bruikbaarheid op het gebied met de ongelukkige naam "kunstmatige intelligentie" niet onvermeld blijven. Het gebrek aan aandacht voor procedurele datastructuren, met name bij het ontwerp van programmeertalen, is niet in overeenstemming met hun nut en komt vermoedelijk voort uit onbekendheid met hun mogelijkheden, wat weer te verklaren valt uit dat gebrek aan aandacht.

LITERATUUR

[1] REYNOLDS, J.C., *User-defined data types and procedural data structures as complementary approaches to data abstraction,* New Directions in Algorithmic Languages 1975, 154 - 165, S.A. Schuman (ed.), IRIA, 1976.

[2] LINDSEY, C.H., *Specification of partial parametrization proposal,* ALGOL Bulletin 39.3.1 (1976) 6 - 9.

[3] BEELER, M., & al., *HAKMEM,* Artificial Intelligence Memo No. 239, MIT, 1972.

[4] STOY, J.E. & C. STRACHEY, *OS-6, an experimental operating system for a small computer; Part 2: input/output and filing system,* The Computer Journal 15 (1972) 195-203.

[5] VAN VLIET, J.C., *Towards an implementation-oriented definition of the ALGOL 68 transput,* Mathematical Centre Report IW 90, 1977.

FILE-OPTIMALISERING DOOR MIGRATIE VAN RECORDS

J. VAN LEEUWEN
Rijksuniversiteit Utrecht

1. INLEIDING

Een aantal in de praktijk veel gebruikte gegevensstructuren laten nog heel wat speelruimte in de eigenlijke, onderlinge plaatsing van records. Dit geldt stellig voor ongeordende files (zoals sequential files of piles), maar ook voor geordende files die records volgens een zoekboom aaneen linken. Men kan de nog aanwezige vrijheid in record plaatsing benutten door de meest gevraagde records zo ver mogelijk "vooraan" in de file te plaatsen, en zo een grote(re) zoektijd alleen te riskeren voor weinig opgevraagde records. Dit heeft men traditioneel natuurlijk vaak gedaan, op basis van een bekende of verwachte waarschijnlijkheidsdistributie p voor de records in een file. De bepaling van p op grond van een waargenomen file-verkeer duidde men vroeger (zo in de 50-er jaren) aan als het maken van een "activity profile".

De typische omstandigheid laat zich illustreren voor sequential files, waarbij we nu eerst maar een vaste record-lengte d aannemen. Als de opvraag-frequentie van de i^{de} record p_i bedraagt, dan is de gemiddelde zoektijd per record in een file van n records evenredig met

$$(1.1) \qquad \sum_{1}^{n} p_i * i * d.$$

Al in de vroege praktijkjaren wist men dat (1.1) minimaal is als men records rangschikt zo dat

$$(1.2) \qquad p_1 \geq p_2 \geq \ldots \geq p_n.$$

BOOTH [3] vindt het al heel gewoon. Indien men een variabele record-lengte toelaat en de i^{de} record een lengte d_i heeft, dan wordt de gemiddelde zoek-

tijd evenredig met

$$(1.3) \qquad \sum_{1}^{n} p_i(d_1+\ldots+d_i).$$

Het vinden van een criterium voor optimale plaatsing is nu weinig lastiger, en men kan bewijzen (SMITH [32], zie KNUTH [16] pp. 400-401):

STELLING. *De gemiddelde zoektijd* (1.3) *is minimaal dan en slechts dan als records zo gerangschikt zijn dat* $\frac{p_1}{d_1} \geq \frac{p_2}{d_2} \geq \ldots \geq \frac{p_n}{d_n}$.

Indien men records opgeslagen heeft in een indexed-sequential file en een binaire zoekboom als file-directory gebruiken wil, dan zal men die boom in eerste instantie zo willen kiezen dat

$$(1.4) \qquad \sum_{1}^{n} p_i * \ell_i$$

minimaal is, met ℓ_i de diepte van de knoop die naar de i^{de} record wijst. Het lijkt erop dat die boom met de algoritme van Huffman te construeren is (zie KNUTH [16] of VAN LEEUWEN [34]), maar daarin worden records naar noodzaak vrijelijk omgeplaatst en wordt dus geen rekening gehouden met het feit dat de rangschikking van records al door de ordening der primary keys van te voren vastligt. Men kan een optimale boom met voorgeschreven recordvolgorde toch nog in $O(n \log n)$ bepalen met een diepe algoritme van HU en TUCKER ([13], ook HU [12] en KNUTH [16]), laatst nog weer iets gemodificeerd door GARCIA en WACHS [9]. In de meest algemene vorm moet men ook de frequentie van niet-succesvolle zoekacties in aanmerking nemen, en rekenen met waarschijnlijkheden q_i dat men de file betreedt met een key die tussen de i^{de} en $(i+1)^{ste}$ record valt. KNUTH [15] geeft een $O(n^2)$ algoritme die zulk een werkelijk algemene, optimale zoekboom bij bekende p_i en q_i construeert. Wie deze algoritmen voor realistische files toch te duur vindt kan tegenwoordig kiezen uit diverse $O(n)$ algoritmen die een "bijna-optimale" zoekboom leveren waarmee men in de praktijk ook al goede resultaten krijgt (FREDMAN [8], MEHLHORN [20], ook VAN LEEUWEN [34]).

Een beduidend nadeel van deze optimale structuren is dat files zelden statisch zijn, en geen zijn optimaliteit behoudt indien voortdurend willekeurige records worden ingevoegd of weggelaten. Het is dan al niet eenvoudig file-directories efficiënt te houden als de p_i's gewoon gelijk zijn, en nog worden geregeld nieuwe types gebalanceerde bomen ontworpen die al

of niet beter functioneren dan bestaande "dynamische" datastructuren (zie OTTMAN, SIX en WOOD [27], OTTMAN [26] en VAISHNAVI, KRIEGEL en WOOD [33] voor enige recente activiteiten). Ons doel hier is na te gaan welke extra bijsturing men zou wensen als ook de p_i-waarden verschillend zijn, en binnen de vrijheid van balancering (of een andere bindende eis) men dié plaatsing van records of wijzers naar records wil bereiken die nog het dichtst bij een optimale zoekboom komt en blijft.

Een zeker even lastig probleem dat de optimaliteit van een eenmaal gevonden structuur bedreigt treedt op indien geen betrouwbare informatie over de p_i voorhanden is, en men pas tijdens de looptijd van een file het activity-profile te weten komt. Nog weer moeilijker wordt het als de p_i's niet "constant" zijn, maar voortdurend wijzigen volgens een onvoorspelbaar patroon van opvraging door de users. Het liefst zou men tijdens de transacties records willen laten "migreren" en de plaatsing in de file (of althans de plaatsing van wijzers in het file-directory) zo laten aanpassen dat de meest gevraagde records zich toch steeds weer "vooraan" bevinden. Files die op deze wijze "dynamisch" zijn heten traditioneel "self-organizing", "self-optimizing", "self-modifying" of "intelligent".

We zullen onderzoeken welke methodes van "zelf-optimalisering" in de aangeduide zin zoal bekend zijn, eventueel in combinatie met een meer algemener dynamisch gebruik van de file. Natuurlijk moet een praktische optimaliserings-algoritme niet te kostbaar zijn, en stellig niet steeds weer een optimale structuur van voren af aan opbouwen zodra een verstoring optreedt. Men kan dus niet hopen op een voortdurend behoud van de optimale vorm, maar wel op gestadige convergentie daar naar toe indien een huidig opvraagpatroon zich zou stabiliseren.

2. BIJSTURING IN SERIËLE BESTANDEN

Het is niet verwonderlijk dat diverse mogelijkheden voor zelf-optimalisering al meer dan 10 jaar geleden te berde gebracht werden, in beschouwingen over record-plaatsing in de toen al ruim toegepaste sequential files en serieel te doorlopen tabellen. Voor zover mij bekend heeft McCABE [18] voor het eerst expliciete "relocation schemes" aangegeven, maar de kans is groot dat die in de praktijk al enige jaren bekend waren. Later zijn enkele van die schemes nog wel weer eens onafhankelijk door anderen voorgesteld en geanalyseerd. Een typisch schema van McCabe (de MTF of "move to front" regel) werd door HENDRICKS [11] hervonden, en door BURVILLE en KINGMAN [6]

verder bestudeerd, als methode voor de rangschikking van boeken: "When a book is demanded, it is removed and replaced (before the next demand) at the left hand end, the other books being moved to the right as necessary to make room for it". Een wat praktischer schema ook al door McCabe voorgesteld (de "transposition" regel) maakt een minder radicale aanpassing per keer: "... the record to which a query refers is relocated by interchanging it with the record preceding it in the file. (If the queried record happens to be the first in the file, then its location is unchanged)". McCabe vermoedde dat dit schema asymptotisch beter was dan zijn MTF-regel, maar dat werd pas in 1975 door RIVEST [28] bewezen.

Natuurlijk zijn er wel meer "regels" voor seriële bestanden bedacht en soms ook op bruikbaarheid getest. Er blijken voornamelijk twee principes ten grondslag te liggen:

(i) het gebruik van een of andere vorm van telling van de opvraag-frequentie,

(ii) het gebruik van standaard permutaties van de record-volgorde (waarbij typisch na opvraging van record j de file wordt gepermuteerd volgens π_j).

De "duurte" van een record-plaatsing $r_{i_1}, r_{i_2}, \ldots, r_{i_n}$ meten we met een expressie zoals (1.1). Indien r_i op tijdstip t een opvraag-frequentie $p_i(t)$ heeft en $v_i(t)$ vergelijkingen van keys nodig zijn om hem te localiseren, dan zal de gemiddelde zoektijd voor een record in de file evenredig zijn met

$$(2.1) \qquad D(t) = \sum_{1}^{n} p_i(t) * v_i(t).$$

We nemen aan dat $p_i(t) \to p_i$ voor $t \to \infty$ en hopen dat door toepassing van een vorm van zelf-optimalisering D(t) naar D convergeert, waarbij D de duurte van een optimale record-plaatsing bij limiet-frequenties p_i is. We zullen nu nagaan hoe principes (i) en (ii) zijn verwerkt tot regels die dit doel op al of niet praktische wijze bereiken.

2.1. De FC of "frequency count" regel

Elke record r_i wordt van een extra field c_i voorzien waarin wordt bijgehouden hoeveel maal r_i tot nu werd opgevraagd, en men handhaaft een rangschikking zodat te allen tijde

$$(2.2) \qquad c_{i_1} \geq c_{i_2} \geq \ldots \geq c_{i_n}.$$

Indien een volgend query naar r_{i_j} vraagt, dan wordt zijn teller bijgewerkt

$$c_{i_j} := c_{i_j} + 1$$

en zo nodig (als nu $c_{i_j} > c_{i_{j-1}}$) een verwisseling tot stand gebracht met $r_{i_{j-1}}, r_{i_{j-2}}, \ldots$ om relatie (2.2) voor de tellers van opvolgende records te herstellen.

Ondanks de zekere convergentie is deze methode toch niet erg bruikbaar. Nog afgezien van het feit dat de regelmatig noodzakelijke verschuiving van mogelijk méérdere records niet erg wenselijk is, noemt KNUTH [16] (p.398) al dat in de praktijk de tellers spoedig uit elke redelijke field-grootte zullen barsten. Da navolgende schema's zijn er op uit om de telling binnen de perken te houden.

2.2. De FD of "frequency difference" regel

Wederom wordt record r_i van een extra field d_i voorzien, maar daarin telt men nu het verschil tussen de frequency count en die van zijn opvolger in de huidige plaatsing:

$$d_{i_j} = c_{i_j} - c_{i_{j+1}} \tag{2.3}$$

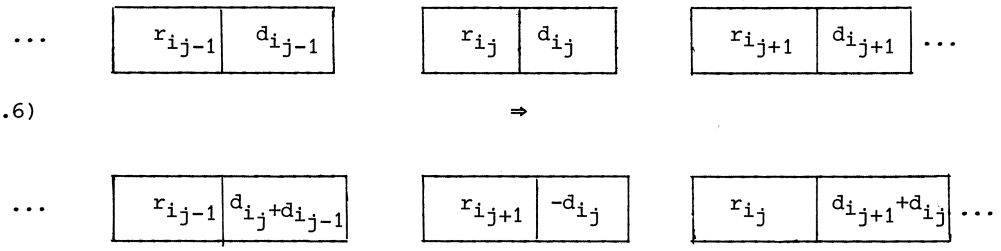
(met $c_{i_{n+1}} = "c_{n+1}" = 0$). Men handhaaft nu een zodanige rangschikking dat te allen tijde

$$d_{i_j} \geq 0. \tag{2.4}$$

Indien record $r_{i_{j+1}}$ wordt opgevraagd, dan moeten nu maar liefst 2 fields worden bijgewerkt:

$$\begin{aligned} d_{i_{j+1}} &:= d_{i_{j+1}} + 1 \\ d_{i_j} &:= d_{i_j} - 1 \end{aligned} \tag{2.5}$$

maar gelukkig kan maar één daarvan (d_{i_j}) de te handhaven invariant (2.4) in gevaar brengen. Als men r_{i_j} met zijn voorgangers wil verwisselen, dan is het niet nodig de $c_{i_{j+1}}, c_{i_j}, \ldots$ expliciet te kennen (al zou het wel mogelijk zijn na het serieel doorlopen van de file):



(2.6) ⇒

en dit zou nog enkele stappen naar "links" moeten doorgaan om (2.4) te herstellen.

De bijwerking van de record-plaatsing is natuurlijk dezelfde als in de FC-regel, maar de verwachting is dat de d-fields minder snel zullen overlopen dan de c-fields en dus langer met minder bits volstaan kan worden. Mede in aanmerking nemend dat de hoeveelheid werk bij schuiven iets is toegenomen, blijft het een te "onbegrensde" techniek om met vertrouwen te implementeren.

2.3. De LD of "limited difference" regel

Deze door BITNER [2] voorgestelde variant van de FD-regel gebruikt een of andere vaste grens g, en werkt een veld gewoon niet bij als het in absolute waarde boven g zou komen. De LD-regel zal voor vaste g niet noodzakelijk convergeren, maar doet dat wel voor $g \to \infty$. BITNER [2] toont experimenteel dat die convergentie dan ook snel verloopt.

2.4. De WMC of "wait, move and clear" regel

In deze ook door Bitner voorgestelde regel wordt weer in elk record een gewone frequency-count bijgehouden, maar geen relocaties vinden plaats tot de frequency-count van een juist opgevraagde record r_i een zekere grens g overschrijdt. Dan worden de records volgens een of andere "move"-regel omgeplaatst en alle c-fields weer op 0 gezet ("cleared"), alvorens het fileverkeer weer voortgaat. Als "move"-regel zou men sortering op het dan bekende c-field kunnen overwegen, maar dat is wel kostbaar voor een grote file. BITNER [2] onderzoekt het gedrag als we een eenvoudige "optimaliserende" permutatie-regel gebruiken en bewijst:

STELLING. *Voor* $g \geq 1$ *is de asymptotische duurte van de file onder de "wait g, move and clear" regel kleiner dan die onder de gebruikte permutatie-regel alleen.*

(Het idee is dat een permutatie-regel ook uitwisseling naar voren zou verrichten als eens records van lage frequentie worden opgevraagd, maar de WMC-regel in ieder geval wacht tot een moment dat zich een record als duidelijk "meest gevraagde" heeft gemanifesteerd en dan hém naar voren wisselt.) BITNER [2] bewees ook nog dat de "wait g, move and clear" regel convergeert naar D als $g \to \infty$ (maar niet erg snel). De WMC-regel zal goed bruikbaar zijn als de $p_i(t)$ blijven variëren.

2.5. De WM of "wait and move" regel

Dit is een variant van de WMC-regel waarin wederom gewacht wordt tot het c-field van een of andere r_i een grenswaarde g overschrijdt en dan pas een standaard permutatie-regel wordt toegepast, maar nu uitsluitend het c-field van deze r_i weer op 0 gezet wordt en de overige records dus hun huidige telling behouden. BITNER [2] toont dat de "wait g and move" regel altijd convergeert, maar niet noodzakelijk naar D voor $g \to \infty$!

We zullen dan nu eens kijken naar enkele traditionele technieken die uitdrukkelijk niet van telling gebruik maken. Zoals gezegd voeren diverse van deze technieken al terug tot McCABE [18].

2.6. De MTF of "move to front" regel

Hierin wordt steeds de nieuw opgevraagde record van zijn plaats verwijderd en aan de "kop" van de file geplaatst. Dit is eenvoudig te implementeren als records dubbel-gelinkt in een indexed-sequential file staan (wel ten koste van toch weer tenminste één extra field per record), maar vereist anders dat alle voorgangers van de opgevraagde record een plaats naar rechts verschoven moeten worden om ruimte aan de kop te maken. McCABE [18] bewees al dat de gemiddelde zoektijd voor een file onder de MTF-regel naar een bepaalde uitdrukking in de p_i's convergeert, maar na een onafhankelijk bewijs van hetzelfde resultaat tonen dan BURVILLE en KINGMAN [6] voorzover mij bekend voor het eerst dat die limiet tussen D en 2D-1 ligt: "The moral... is that, if the popularity of the books (records) is known, they should be shelved in that (i.e. the optimal) order, but that if not, the mechanism described here at most doubles the mean search time". Bedenk overigens wel dat de record-plaatsing zelf onstabiel blijft. Het resultaat laat zich op de volgende manier eenvoudig bewijzen (zie ook RIVEST [28]).

Zij p(i,j) de "limiet-waarde" voor de waarschijnlijkheid dat record r_i vóór record r_j in de file voorkomt. Maar r_i kan bij de MTF-regel alleen vóór r_j voorkomen als er ooit een verzoek voor r_i ná een verzoek voor r_j gedaan werd, dan gevolgd door nog eens willekeurig k keren dat een record ongelijk r_i en r_j (steeds met kans $1-p_i-p_j$) opgevraagd wordt. Dus

$$(2.7) \qquad p(i,j) = p_i * \sum_{k\geq 0} (1-p_i-p_j)^k = \frac{p_i}{p_i+p_j}$$

en er volgt dat

$$(2.8) \qquad D(t) \to \tilde{D}_{MTF} = \sum_{j=1}^{n} p_j * (1+ \sum_{\substack{i=1 \\ i\neq j}}^{n} p(i,j)) = 1 + 2 \sum_{1\leq i<j\leq n} \frac{p_i p_j}{p_i+p_j} .$$

Aannemend dat $p_1 \geq p_2 \geq \ldots \geq p_n$ en dus dat $D = \sum_{i=1}^{n} p_i * i$ volgens (1.2), en gebruikend dat

$$(2.9) \qquad \frac{1}{2} p_j \leq \frac{p_i p_j}{p_i+p_j} < p_j \qquad \text{voor } i < j$$

en

$$(2.10) \qquad \sum_{1\leq i<j\leq n} p_j = \sum_{j=1}^{n} p_i*(j-1) = D - \sum_{j=1}^{n} p_j = D - 1,$$

dan volgt direct dat

$$(2.11) \qquad D \leq \tilde{D} < 2D - 1,$$

het gezochte resultaat.

Het is opmerkelijk dat de MTF-regel de enige door KNUTH [16] besproken techniek voor een zelf-organiserende file is, mogelijk door de eenvoud en goede resultaten voor gelinkte lijsten: "Computational experiments involving actual compiler symbol tables indicate that the self-organizing method works even better than the above formulas predict ...". Voor niet-gelinkte, maar eenvoudig seriële tabellen brengt de MTF-regel echter wel veel werk per keer. McCABE [18] (p.616) schrijft al: "The method of relocation we have discussed is probably impractical for most files, because the number of records to be relocated would be excessive", en komt dan met een voorstel voor een meer praktische regel.

2.7. De T of "transposition" regel

Hierin wordt steeds de nieuw opgevraagde record met zijn directe voorganger (als die er is) verwisseld. McCabe bewijst dat ook nu D(t) naar een zekere limiet $\tilde{D}_T$ moet convergeren, maar kan er nog geen expliciete formule voor vinden (dat lukte later eerst aan RIVEST [28]). McCabe vermoedde dat de T-regel in ieder geval op den duur een betere file-organisatie zou onderhouden dan de MTF-regel. Dit werd pas in 1975 door RIVEST [28] ook aangetoond:

STELLING. $D \leq \tilde{D}_T < \tilde{D}_{MTF} < 2D-1$, *tenzij* $n = 2$ *of* $p_i = p_j$ *voor alle* i *en* j.

Het lijkt er dus op dat de T-regel, ook door de mogelijk geringere hoeveelheid werk per keer, beter voldoen zal dan de MTF-regel. McCABE [18] (p.617) heeft er hoge verwachtingen van: "It is the author's belief that transposition relocation can be made the basis of a practical and effective activity organization for files held in core memory of a computer, or held on discs". Dit is vast waar voor serieel te doorlopen tabellen, maar veel minder duidelijk voor gelinkte lijsten. BITNER [2] toont voor enkele waarschijnlijkheidsdistributies en experimenteel dat de MTF-regel veel sneller naar zijn "goede" limiet convergeert dan de T-regel, en dat men dus in de praktijk niet onverdeeld gunstig over de T-regel oordelen kan.

2.8. Andere "permutatie"-regels

RIVEST [28] merkt op dat de MTF- en T-regels voorbeelden zijn uit een algemene klasse van technieken die de file permuteren. In elke dergelijke techniek is er voor elke i een standaard permutatie π_i, en steeds als een of andere r_j opgevraagd is worden de records in de file volgens de bijbehorende π_j omgeschikt. Een "beste" permutatie-regel is een zodanige regel $R = \{\pi_1,\ldots,\pi_n\}$ zodat voor elke waarschijnlijksdistributie p_i en elke andere permutatie-regel R' geldt:

$$\tilde{D}_R \leq \tilde{D}_{R'}. \qquad (2.12)$$

Het is niet bekend of er een "beste" permutatie-regel is die inderdaad de "beste" is voor elke keuze der p_i's, maar RIVEST [28] vermoedt dat de T-regel een voorbeeld is. YAO (zie RIVEST [28] of BITNER [2]) heeft aangetoond dat de T-regel in wezen de enige beste permutatie-regel kan zijn, als er überhaupt een beste regel onafhankelijk van de specifieke p_i-ver-

deling bestaat! RIVEST [28] toont wel iets over de noodzakelijke vorm der permutaties in een beste regel:

STELLING. *De* π_i *in een beste permutatie-regel* R *moet de records in posities* i+1 *tot* n *vastlaten en de record in positie* i *naar een positie* i' *met* i' < i *opschuiven.*

Men kan uit deze stelling in ieder geval een bevestiging der intuïtie lezen welke soort permutaties optimaliserend zijn, maar het laat ook de MTF-regel toe waarvan we in ieder geval weten dat hij niet de (asymptotisch) "beste" is.

2.9. Vertraagde permutatie-regels

Men gebruikt weliswaar een standaard permutatie-regel maar voert de voorgeschreven omschikking der records slechts om de M stappen uit, voor een overwogen keuze van M. (Als verdere generalisatie zou men de keuze van M nog weer in de tijd kunnen variëren). Dit idee voert ook al weer tot McCABE [18] terug, die daarmee de hoeveelheid werk voor de anders bij elke stap nodige omschikkingen wat wil reduceren: "The integer M would be chosen so that the amount of time spent in relocating records would not be excessive". Het is duidelijk dat dit misschien voor de MTF-regel belangrijk is, maar niet voor de T-regel. McCabe toont (officieel alleen voor de MTF- en T-regels) dat een vertraagde permutatie-regel naar dezelfde limiet $\tilde{D}$ convergeert als de onvertraagde versie van die regel, maar dan natuurlijk wel langzamer: "... one must consider... that the average positions will approach this (i.e. their) limiting value M times as slowly as it would if we relocated after every query". Dit laatste is waarschijnlijk de reden dat men het idee van vertraging voor zover mij bekend daarna nooit meer expliciet heeft opgevoerd.

Een zo lange rij van mogelijke schema's voor zelf-optimalisering (en zonder moeite kan men er nog bij bedenken) vraagt om een conclusie welke het beste is, maar daarvoor ontbreekt ten enen male (nog) voldoende experimentering in de praktijk. Het lijkt er erg op dat de T-regel te verkiezen is boven andere, maar dat wordt voorlopig niet bevestigd door het enig mij bekende vergelijkend onderzoek van BITNER [2]. Eenvoud en mate van convergentie naar een best mogelijk gedrag in aanmerking nemend, concludeert BITNER (voornamelijk op basis van experimenten) dat de LD-regel

er in de praktijk beter vanaf komt dan welke andere dan ook! Hij toont dat zelfs kleine g-waarden (in de orde van 5 of 10) al heel goede resultaten geven, en dat kost maar een paar bits per record.

3. BIJSTURING IN BINAIRE ZOEKBOMEN

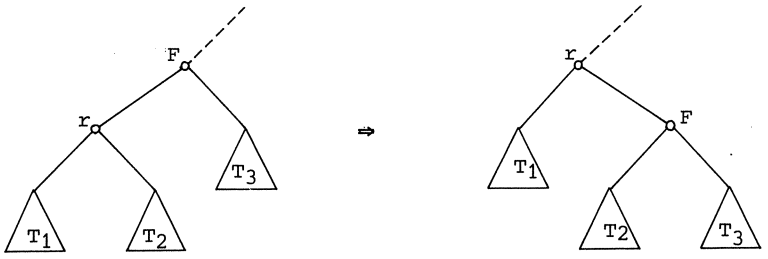
Het idee om ook in zoekbomen on-line optimalisering te verrichten lijkt pas veel later te zijn ontstaan dan voor sequentiële structuren. Eerst in het begin der 70-er jaren komt er wat literatuur waarin algoritmen (met name de Hu-Tucker algoritme) voor de constructie van optimale zoekbomen voor de praktijk van niet zoveel waarde worden geacht, omdat zoals tevoren de nodige p_i's niet goed genoeg bekend zijn maar ook omdat een optimale zoekboom meestal grillig wijzigt als records worden ingevoegd of weggelaten. Dan komen de ideeën voor "bijna-optimale" zoekbomen die goedkoper (typisch in $O(n)$ tijd) te maken zijn, maar zonder overdadig werk (typisch in slechts $O(\log n)$ tijd per transactie) in bijna-optimale vorm zijn te handhaven bij dynamisch file-gebruik (zie BRUNO en COFFMAN [5], WALKER en GOTLIEB [36], NIEVERGELT en REINGOLD [25], MEHLHORN [20], FREDMAN [8] en VAN LEEUWEN [34]). Er moet overigens gezegd worden dat men aanvankelijk de bijna-optimale bomen als statische structuur bestudeerde, en niet echt voorstelde hoe zij zich voor een dynamische file intern zouden moeten bijsturen. Dit werd eerst besproken door VAN LEEUWEN [34], maar omstreeks diezelfde tijd op concreter wijze uitgewerkt door ALLEN en MUNRO [1].

De "duurte" van een binaire zoekboom voor $r_1,\ldots,r_n$ laat zich weer meten met een expressie E(t) als in (1.4). We hopen ditmaal dat E(t) naar E convergeert voor $t \to \infty$, waarbij E de gemiddelde zoektijd is voor een record in de echt optimale binaire zoekboom bij limiet-waarschijnlijkheden p_i. We nemen doorgaans aan dat er records in alle knopen van de boom zitten, maar het is geenszins ongewoon dat in een file-directory de records uitsluitend aan de bladen zitten. We spreken steeds over "records", maar het is duidelijk dat in een praktische organisatie we alleen wijzers naar records in de knopen van de zoekboom hangen. Zoals tevoren zijn de bekende methoden van zelf-optimalisering in zoekbomen weer gebaseerd op een vorm van telling enerzijds en een migratie-regel anderzijds, waarbij men nu op een of andere wijze de "meest gewilde" records naar de top van de boom wil doen bewegen. Het is gunstig om de voorgestelde migratie-regels zonder tellers eerst te bestuderen.

3.1. De SE of "simple exchange" regel (ook wel "move up one" regel genoemd).

In deze door ALLEN en MUNRO [1] voorgestelde regel wordt in navolging van de T-regel steeds de nieuw opgevraagde record met zijn vader in de boom "verwisseld", op de volgende wijze:

(3.1)



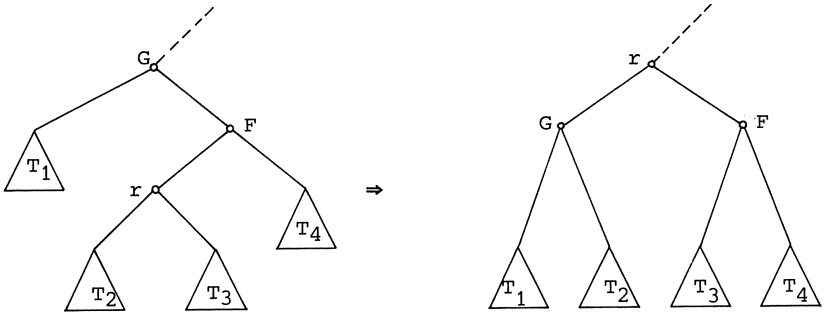
(Ga na dat we een binaire zoekboom behouden.)

Geheel in tegenstelling tot de T-regel voor seriële bestanden, verloopt de SE-regel mogelijk zeer belabberd. Allen en Munro bewijzen dat voor $p_i = p_j$ $(=\frac{1}{n})$ E(t) naar een waarde $E_{SE} \approx \sqrt{\pi n}$ convergeert, en dat is erg ver van de optimale $E \approx \log n$ in dit geval verwijderd.

3.2. De DE of "double exchange" regel

Hierin wordt een "simple exchange" als in (3.1) verricht als het pad bij F "recht door" loopt, maar een "dubbele" wisseling van de volgende vorm geschiedt als er bij F een "knik" zit:

(3.2)



(Ga weer na dat we een binaire zoekboom houden.)

Het idee is dat zo de zoeklengte voor de meeste records kleiner blijft dan na twee opvolgende simple exchanges. We zullen de techniek zo meteen weer ontmoeten als het al of niet roteren expliciet wordt afgewogen aan het verschil in opvraag-frequentie van de "opgetrokken" records (in T_2,T_3 en r) en de "neergeduwde" records (in T_1 en G). Voor de DE-regel zelf lijken geen resultaten bekend.

3.3. De MTR of "move to root" regel

In deze door ALLEN en MUNRO [1] ingevoerde regel worden steeds simple exchanges op de nieuw opgevraagde record herhaald tot die record in de wortel van de boom belandt, zo een soortgelijk effect bereikend als de MTF-regel. Voor de MTR-regel blijkt E(t) heel redelijk te convergeren naar een limiet E_{MTR}, die gegarandeerd maar hoogstens een (kleine) constante factor van E verschilt! Allen en Munro (zie ook MEHLHORN [22]) bewijzen

STELLING. $E \leq E_{MTR} < 1.3863\ (E + \log E) + 3$

Het gedrag van de MTR-regel is dus in voorkomende gevallen verrassend beter dan dat van de SE-regel, geheel in tegenstelling tot het verband tussen de analoge MTF- en T-regels in seriële bestanden. Allen en Munro tonen nog dat voor een random boom al na ongeveer 2n log n opvragingen de verwachte gemiddelde zoektijd onder de MTR-regel tot minder dan 1 van E_{MTR} genaderd is. De regel is eenvoudig te implementeren, maar het beste dat over de hoeveelheid werk per keer gezegd kan worden is dat die evenredig is met de oorspronkelijke zoektijd van de opgevraagde record.

We zullen nu voortgaan met enkele weer door BITNER [2] voorgestelde methoden, waarin al of niet begrensde tellers de beslissing om interne rotaties uit te voeren bepalen.

3.4. De MT of "monotone tree" regel

Elke record krijgt een extra c-field waarin het aantal opvragingen voor die record wordt geteld, en men handhaaft een zodanige plaatsing in de zoekboom dat langs elk pad vanaf de wortel de tellingen monotoon afnemen. De meest gevraagde record staat dus in ieder geval steeds aan de top. Als de verhoogde teller van een nieuw opgevraagde record de monotonie verstoort, dan worden rotaties (simple exchanges) met die record uitgevoerd en herhaald tot de monotonie weer hersteld is. (Ga na dat dit zo lukt.)

MEHLHORN [20] bewees al dat de MT-regel mogelijk tot heel slechte zoekbomen kan leiden, met E_{MT} meer dan een factor $\frac{n}{4 \log n}$ slechter dan E. De oorzaak ligt vooral in het ontbreken van enige waarborg voor balancering. BITNER [2] bewijst dat de "gemiddelde" monotone boom even duur is als een random boom.

Om enige balancering te garanderen moeten we selectiever zijn in de uit te voeren rotaties (zie ook VAN LEEUWEN [34] waar dit voor dynamische Huffman-bomen wordt onderzocht).

3.5. De LSR of "limited single rotation" regel

Dit werkt als de MTR-regel, maar simple exchanges vinden alleen plaats in die knopen langs het zoekpad waar het totaal aantal opvragingen van de "op te trekken" records groter is dan dat voor de "neer te duwen" records (let op T_1,T_3, r en F in (3.1)).

3.6. De LDR of "limited double rotation" regel

Deze ook door BITNER [2] voorgestelde regel werkt als de LSR-regel, maar nu wordt bij "knikpunten" niet een simple maar een double exchange overwogen (zie 3.2).

Zo kan men de lijst van zelf-optimaliserende technieken voor zoekbomen nog wel langer maken, maar de belangrijkste thans bekende varianten hebben we gehad. Wederom is het moeilijk te zeggen welke regel "beter" is, al vallen sommige zoals de SE- en MT-regels wel af. Simulaties van BITNER [2] tonen dat onder de regels met tellers de beste resultaten bereikt worden met de LDR-regel!

Geen der besproken methodes is overigens onderzocht voor volledige dynamische files, waarbij dus regelmatig ook nog eens records kunnen worden ingevoegd of weggelaten. Het is niet moeilijk om invoeg en weglaat routines erbij te bedenken, maar het is niet te voorspellen hoezeer de bijna-optimaliteit van de zoekboom daarbij steeds verstoord wordt. Het probleem om een dynamische bijna-optimale zoekboom-structuur te vinden werd voor zover mij bekend voor het eerst in VAN LEEUWEN [34] aangepakt. Voortbouwend op een door hem ontwikkelde "truc" (zie ook VAN LEEUWEN [35], sect. 1.9) gelukte het aan MEHLHORN [21] een bevredigende oplossing te vinden met behulp van een type van weight-balanced trees:

STELLING. *Er is een datastructuur voor volledig dynamische files met de volgende eigenschappen*

(i) *de gemiddelde zoektijd blijft binnen een vaste constante van de optimale E bij de huidige opvraag-frequenties,*

(ii) *de tijd voor interne bijsturing na elke opvraging is evenredig met de oorspronkelijke zoektijd voor de juist opgevraagde record,*

(iii) *de tijd voor invoeging van nieuwe records is zeker niet groter dan* $O(\log W)$, *waarin W het totaal aantal opvragingen tot nu toe is.*

Hiermee lijkt het laatste woord nog niet gezegd. Het zou interessant zijn te onderzoeken of er ook oplossingen zijn op basis van andere gebalanceerde structuren zoals 2-3 bomen.

4. BIJSTURING IN ANDERE STRUCTUREN EN AANVERWANTE PROBLEMEN

Het idee om on-line optimaliserende herordeningen aan te brengen langs het betreden pad in een bestand kan men in principe op elke opslagstructuur voor records of keys toepassen.

Zo kan men het opslaan van records met binaire keys in een "digital search tree" of "sequence tree" eens nader bekijken. Opvolgende bits van een key leveren steeds de te volgen richting door een dergelijke boom, en het record-adres wordt opgeslagen in de eerste "lege" knoop die je tegenkomt. Het is duidelijk dat de volgorde van binnenkomst van records hun plaatsing bepaalt, en dat is niet altijd de meest gunstige rangschikking voor een geringe gemiddelde zoektijd. In diverse artikelen sinds 1971 hebben MIYAKAWA, YUBA, SUGITO en HOSHI (zie [24]) de optimalisering van sequence trees bestudeerd. Zij vonden een voorstelbare, maar erg gecompliceerde methode om door efficiente "shifts" een willekeurige (consistente) recordplaatsing in een sequence tree naar een optimale plaatsing te transformeren. Zij geven ook algoritmen die een gegeven optimale sequence tree steeds weer in optimale vorm brengt bij de gebruikelijke dynamische acties op een file, zelfs bij het abrupt wijzigen van opvraag-frequenties. De algoritmen vergen typisch $O(n^2L)$ of $O(n^3L)$ tijd, met L de maximaal toelaatbare lengte der binaire key-codes. Het is voor zover mij bekend nog open om goedkopere algoritmen te vinden die een redelijke vorm van bijna-optimaliteit voor sequence trees handhaven. (Ga na dat de methodes van 3 niet zo maar opgaan.)

Evenzo kan men zelf-optimalisering in hash-tables proberen, maar dat is lastig en voor zover mij bekend niet vanuit een algemeen standpunt als het

onze nog ondernomen. Een bekende algoritme van BRENT [4] probeert in ieder geval bij collisions de in te voegen key zo "tussen" de andere te plaatsen dat de gemiddelde zoektijd zo min mogelijk toeneemt (zie ook KNUTH [16], p.525-526). RIVEST [29] geeft een algoritme om een statische verzameling van keys onder een willekeurig gegeven hash-functie zo te rangschikken dat de gemiddelde zoektijd voor een key in de hash-table minimaal is. Hij toont dat Brent's algoritme bij invoeging van een nieuwe record in een thans optimale plaatsing niet noodzakelijk weer een optimale hash-table geeft. Het probleem om dan wel een optimaliteit-behoudende invoeg-algoritme voor hash-tables te vinden laat RIVEST [29] nog voor een belangrijk deel open. Wederom kan men zich afvragen of er een vorm van bijna-optimaliteit voor hash-tables is die zich eenvoudiger (of althans goedkoper) handhaven laat.

Een stellig zo interessant probleem dat wel met zelf-organisatie verband houdt (maar dan in een uitgebreide betekenis) treft men vooral bij diskfiles, waar men regelmatig geconfronteerd wordt met een daling van de zoek-efficiëncy als meer en meer records door gebrek aan ruimte in de huidige plaatsing op de tracks naar track- of cylinder-overflow areas moeten gaan. Het moet als erg onpraktisch beschouwd worden om voortdurend feitelijke records te schuiven, en men treft vaak dat de diskfile in bedrijf blijft tot de performance "te veel terugloopt" en dan buiten user-time opnieuw wordt ingedeeld. Men reorganiseert de record-plaatsing en het file-directory typisch zo dat tracks half-vol zijn, om wat ruimte voor te verwachten expansie te hebben zonder nou een track grof onder te bezetten. Het is overigens niet ongebruikelijk om vrije ruimte wat grilliger door de file te verspreiden. Per track nu kan men een manier van zelf-organisering zoals de T-regel heel eenvoudig implementeren, zodat te zijner tijd de meest gevraagde records nog wel op de main tracks blijven (maar wie weet hoe het opvraag-patroon verandert ten gunste van de records die op overflow-tracks terechtgekomen zijn!). Men komt hier op een soort probleem dat welbekend is uit de studie van paging-strategieën in operating systems (we gaan hier nu niet verder op in). De vraag is wel wanneer men nu eigenlijk tot reorganisatie van een diskfile moet overgaan, en op grond van welk criterium men de meest gunstige "database reorganisation points" herkennen kan. EISEN en LEIBOWITZ [7] bezien al een verwante vraag, en proberen te beslissen of het voordeliger is periodiek te reorganiseren of steeds te wachten tot de som van toename in de zoektijd en hoeveelheid tijd voor reorganisatie een zekere grens te boven gaat. In een bepaald analytisch model vinden zij voordeel bij het laatste. SHNEIDERMAN [31] lijkt de eerste

die dit echt op diskfiles toepast. Een zeer interessante meer recente studie werd ondernomen door MARUYAMA en SMITH [17], die voor een type diskfiles aantonen dat er toch soms weinig voordeel is te halen met reorganisering.

Een aardige organisatie voor diskfiles (en/of hun directory) is de B-boom (McCREIGHT en BAYER [19]), die ten koste van wat random-access tijd een heel elegant beheer van track-space implementeert. HELD en STONEBRAKER [10] duiden aan dat het behouden van een dergelijke dynamische structuur in de praktijk toch meer overhead eist en wel eens minder ideaal zou kunnen zijn dan een standaard statische diskfile met geregelde reorganisaties. De B-boom is overigens een dynamische datastructuur in de traditionele zin, en voor zover mij bekend heeft men nog niet onderzocht hoe zo'n structuur is te optimaliseren en in optimale of bijna-optimale vorm is te onderhouden als men de opvraag-frequenties der onderscheiden records in aanmerking gaat nemen. De statisch optimale vorm moet in ieder geval zo veel mogelijk records boven in de boom hebben, zodat in de hoogste niveaus tracks zoveel mogelijk gevuld zijn (zie MILLER, PIPPENGER, ROSENBERG en SNIJDER [23], die ook een linear time algoritme aangeven voor constructie van een optimale B-boom als de rij van keys geordend ingeleverd wordt).

Tenslotte is het in dit kader nog interessant erop te wijzen dat ook in priority queues (van processen) een vorm van zelf-organisatie voorkomt (weer in uitgebreide zin), in het geval dat prioriteiten aan wijziging onderhevig kunnen zijn. JOHNSON [14] voert een symbolische "UPDATE"-primitive in, en toont dat in ieder geval in een heap een record niet eerst weggelaten en met gewijzigde prioriteit opnieuw ingevoegd hoeft te worden, maar direct van zijn huidige plaats naar een consistente locatie in de heap kan worden gewisseld. Als prioriteiten regelmatig wijzigen en dan meestal ook omhoog gaan, dan kan men overwegen een k-aire heap te gebruiken voor een $k > 2$ om de "hoogte" te drukken (JOHNSON [14]).

Het zal duidelijk zijn dat het onderzoek inzake schemas voor zelf-optimalisering van files nog lang niet ten einde is.

LITERATUUR

[1] ALLEN, B. & I. MUNRO, *Selforganising binary search trees*, Conf. record 17th Annual Symposium on Foundations of Computer Science, Houston, 1976, pp.166-172.

[2] BITNER, J.R., *Selfmodifying datastructures,* Proc. Conference on Theoretical Computer Science, Waterloo, 1977, p.36-42.

[3] BOOTH, A.D., *The efficiency of certain methods of information retrieval,* Inf. Control 1 (1958) 159-164.

[4] BRENT, R.P., *Reducing the retrieval time of scatter storage techniques,* CACM 16 (1973) 105-109.

[5] BRUNO, J. & E.G. COFFMAN Jr., *Nearly optimal binary search trees,* Proc. IFIP Congress 1971, North Holland Publ. Company, Amsterdam, 1972, pp. 99-103.

[6] BURVILLE, P.J. & J.F.C. KINGMAN, *On a model for storage and search,* J. Appl. Prob. 10 (1973) 697-701.

[7] EISEN, M. & M. LEIBOWITZ, *Replacement of randomly deteriorating equipment,* Manag. Sci. 9 (1963) 263-276).

[8] FREDMAN, M.L., *Two applications of a probabilistic search technique: sorting* X + Y *and building balanced search trees,* Proc. 7th Annual ACM Symposium on Theory of Computing, Albuquerque, 1975, pp. 240-244.

[9] GARCIA, A.M. & M.L. WACHS, *A new algorithm for minimum cost binary trees,* SIAM J. on Computing 6 (1977) 622-642.

[10] HELD, G. & M. STONEBRAKER, B-*trees re-examined,* CACM 21 (1978) 139-143.

[11] HENDRICKS, W.J., *The stationary distribution of an interesting Markov chain,* J. Appl. Prob. 9 (1972) 231-233.

[12] HU, T.C., *A new proof of the* T-C *algorithm,* SIAM J. Appl. Math. 25 (1973) 83-94.

[13] HU, T.C. & A.C. TUCKER, *Optimal computer search trees and variable-length alphabetic codes,* SIAM J. Appl. Math. 21 (1971) 514-532.

[14] JOHNSON, D.B., *Priority queues with update and finding minimum spanning trees,* Inf. Proc. Letters 4 (1975) 53-57.

[15] KNUTH, D.E., *Optimum binary search trees,* Acta Informatica 1 (1971) 14-25.

[16] KNUTH, D.E., *The art of computer programming,* vol. 3: *sorting and searching,* Addison-Wesley, Reading, Mass., 1973.

[17] MARUYAMA, K. & S.E. SMITH, *Optimal reorganisation of distributed space disk files,* CACM 19 (1976) 634-642.

[18] MCCABE, J., *On serial files with relocatable records,* Oper. Res. 12 (1965) 609-618.

[19] MCCREIGHT, E.M. & R. BAYER, *Organisation and maintenance of large ordered indexes,* Acta Informatica 1 (1972) 173-189.

[20] MEHLHORN, K., *Nearly optimal binary search trees,* Acta Informatica 5 (1975) 287-296.

[21] MEHLHORN, K., *Dynamic binary search,* Proc. 4th Colloq. on Automata, Languages and Programming (Turku), Springer Lect. Notes in Computer Science 52 (1977) 323-336.

[22] MEHLHORN, K., *Effiziente algorithmen,* Teubner Studienbücher 41 (Informatik), Teubner Verlag, Stuttgart, 1977.

[23] MILLER, R.E., N. PIPPENGER, A.L. ROSENBERG & L. SNIJDER, *Optimal 2-3 trees,* Proc. Conference on Theoretical Computer Science, Waterloo, 1977, pp. 30-35.

[24] MIYAKAWA, M., T. YUBA, Y. SUGITO & M. HOSHI, *Optimum sequence trees,* SIAM J. Computing 6 (1977) 201-234.

[25] NIEVERGELT, J. & E.M. REINGOLD, *Binary search trees of bounded balance,* Proc. 4th Annual ACM Symposium on Theory of Computing, Denver, 1972, pp. 137-142.

[26] OTTMAN, Th., *On log n solutions of the dictionary problem for one-sided height-balanced binary search trees,* EATCS Bulletin 1978, no. 4, pp. 20-25.

[27] OTTMAN, Th., H.W. SIX & D. WOOD, *New results in balanced search trees* (circa June 1977), CS Techn. Rep. 77-CS-15, Dept. of Applied Math., McMaster University, Hamilton (Ont.), 1977.

[28] RIVEST, R.L., *On selforganising sequential search heuristics,* CACM 19 (1976) 63-67.

[29] RIVEST, R.L., *Optimal arrangement of keys in a hash table,* Tech. Memo 73, Lab. for Computer Science, MIT, Cambridge, Mass., 1976.

[30] SCHAY, G. & F.W. DAVER, *A probabilistic model of a selforganising file system,* SIAM J. Appl. Math. 15 (1967) 874-888.

[31] SHNEIDERMAN, B., *Optimum data base reorganisation points,* CACM 16 (1973) 362-365.

[32] SMITH, W.E., cited in KNUTH [16] p. 401.

[33] VAISHNAVI, V.K., H.P. KRIEGEL & D. WOOD, *Height balanced 2-3 trees,* CS Techn. Rep. 78-CS-2, Dept. of Appl. Math., McMaster University, Hamilton (Ont.), 1978.

[34] VAN LEEUWEN, J., *On the construction of Huffman trees,* Proc. 3rd Colloq. on Automata, Languages and Programming (Edinburgh), Edinburgh University Press, 1976, pp. 382-410.

[35] VAN LEEUWEN, J., *The complexity of data-organisation,* in: K.R. APT en J.W. DE BAKKER (eds.), *Foundations of Computer Science* II (part 1), MC Tract 81, Mathematisch Centrum, Amsterdam, 1976, pp. 37-147.

[36] WALKER, W.A. & C.C. GOTLIEB, *A topdown algorithm for constructing nearly optimal lexicographic trees,* in: R.C. READ (ed.), *Graph theory and computing,* Acad. Press, New York, 1972, pp. 303-323.

ENKELE UITGANGSPUNTEN VOOR DATAMANIPULATIE

J.H. TER BEKKE
Ministerie van Verkeer en Waterstaat

1. SAMENVATTING

Een database bestaat uit een (eindige) kollektie samenhangende gegevens die voor meerdere toepassingen toegankelijk moet zijn. Om tot een goed database beheer te komen is het streven om in het model van een database zo weinig mogelijk vast te leggen over de fysieke representatie van en toegang tot de gegevens. Een goed uitgangspunt hiervoor vormt het relationele gegevensmodel. Dit model wordt nader uitgewerkt in het eerste deel van dit artikel.

Hoewel de gegevens (-waarden) in een database in de loop der tijd kunnen veranderen, blijft de betekenis (semantiek) van de gegevens in een database ongewijzigd. Deze semantiek vormt het uitgangspunt voor het ontwerp van een manipulatietaal voor databases. In het tweede deel van dit artikel worden elementen van een eenvoudige manipulatietaal voor relationele databases geïntroduceerd aan de hand van enkele voorbeelden. De resulterende algoritmen bestaan telkens uit een klein aantal eenvoudige opdrachten. Dit betekent dat de eenvoudige relationele gegevensstrukturen ook tot uitdrukking komen in de algoritmen die op deze strukturen zijn gedefinieerd.

2. DEFINITIE VAN GEGEVENS

Voor de vastlegging van de informatie-inhoud van een database is het aan te bevelen om in eerste instantie geen aandacht te schenken aan de (uiteindelijke) fysieke representatie van de gegevens; het zogenaamde streven naar data-independence [2]. Pas nadat we weten welke operaties op de gegevens moeten worden uitgevoerd, kunnen we op een zinvolle wijze een representatie kiezen waarmee een efficiënte implementatie van deze operaties mogelijk wordt gemaakt. Deze aanpak heeft als voordeel dat een wijziging

in de fysieke representatie van de gegevens (welke veroorzaakt kan worden door nieuwe toepassingen) geen konsekwenties hoeft te hebben voor de logische struktuur van de algoritmen welke op deze gegevensstrukturen zijn gedefinieerd.

Omdat de volgende faktoren geen bijdrage leveren tot de formele vastlegging van de gegevens in een database, spelen deze faktoren geen rol in een model van een database (deze faktoren zullen dus ook niet voorkomen in onze algoritmen):

- fysieke volgorde
- wijzers
- toegangspaden

Wanneer we bij het ontwerp van een database model geen gebruik maken van bovenstaande faktoren, dan blijken er slechts enkele wiskundige begrippen nodig te zijn. Deze basisbegrippen zijn echter niet voldoende voor een formele vastlegging van alle invariante eigenschappen van een database. Vandaar dat we naast bestaande wiskundige begrippen enkele nieuwe begrippen introduceren waarmee we ook deze eigenschappen van de gegevensstrukturen kunnen vastleggen.

2.1. Basisbegrippen

In het relationele gegevensmodel wordt een waardenverzameling een domein genoemd. De term enkelvoudig domein wordt gebruikt voor een verzameling elementaire (d.w.z. door de gebruiker opgevat als niet verder op te splitsen) waarden. Voorbeelden van enkelvoudige domeinen zijn *int* (de verzameling der gehele getallen) en *bool* (de verzameling der logische waarden). Een variabele v_1 die een waarde aanneemt uit domein D_1 wordt hier in dit artikel aangegeven met *var* $v_1 : D_1$.

Gegeven zij een eindige kollektie (niet noodzakelijk verschillende) enkelvoudige domeinen $D_1,\ldots,D_n$. Een relatie op deze enkelvoudige domeinen is een deelverzameling van het cartesische produkt $D_1 \times \ldots \times D_n$; n wordt de graad van de relatie genoemd. Wanneer *var* $v_i : D_i$ is gedefinieerd voor $i = 1,\ldots,n$ dan wordt in dit artikel een relatie R op de domeinen $D_1,\ldots,D_n$ aangegeven met *rel* $R.v_1,\ldots,v_n$.

Een element $(v_1,\ldots,v_n)$ van een n-aire relatie $R.v_1,\ldots,v_n$ wordt een n-tupel genoemd; het bestaat uit n waarden v_i met $v_i \in D_i$ voor $i = 1,\ldots,n$. Het aantal tupels in een relatie wordt de cardinaliteit van deze relatie

genoemd. Een variabele wordt in de context van een relatie ook wel een attribuut van deze relatie genoemd.

In het voorgaande is een relatie gedefinieerd als één deelverzameling van een cartesisch produkt. Bij databases wordt dit begrip echter in een ruimere betekenis gehanteerd. Omdat het tijdsaspekt ook een rol speelt worden verschillende deelverzamelingen van eenzelfde cartesisch produkt (welke door wijzigingsopdrachten in elkaar kunnen overgaan) gemakshalve beschouwd als verschillende toestanden van eenzelfde relatie. Wij zullen hier echter het onderscheid tussen deze verschillende interpretaties handhaven door bij databases voortaan te spreken van database relaties. Met dit laatste begrip kunnen we een database definiëren als een eindige kollektie van database relaties van diverse graden.

2.2. Semantiek

Bij niet-geïntegreerde gegevensverwerking kan een programmeur de gegevens rangschikken en struktureren op een manier zoals het hem gunstig lijkt voor zijn toepassing. In een geïntegreerde database omgeving is dit echter niet langer toegestaan. Omdat gegevens in zo'n omgeving door verschillende programma's mogen worden gebruikt, wordt een systematische aanpak voor de gegevensstrukturering vereist. Voordat we denken aan mogelijke toepassingen zullen we eerst de semantiek van de gegevens moeten vastleggen en in de gegevensstrukturen tot uitdrukking laten komen. Deze aanpak vinden we ook terug in het werk van CODD [2,5].

In onderstaande definities zullen we de volgende notaties gebruiken:

k_+: niet-lege kollektie van attributen $k_1,\ldots,k_n$
k_-: mogelijk lege kollektie van attributen $k_1,\ldots,k_n$
k'_-: gemodificeerde kollektie van attribuutwaarden k_-
s,fs,ts: attributen welke gedefinieerd zijn op hetzelfde domein
R_i: toestand van relatie *R* op tijdstip i.

2.2.1. Subjecten

Een consistent database model ontstaat niet enkel door gebruik te maken van het begrip database relatie. Codd heeft dit reeds enkele jaren geleden in een aantal artikelen opgemerkt [2,5] en hiervoor (als hulpmiddel voor de database ontwerper) het begrip funktionele afhankelijkheid geïntroduceerd. In essentie is dit begrip ingevoerd om de betekenissen van de te

representeren gegevens formeel vast te leggen. In dit artikel willen we ook dezelfde semantische eigenschappen vastleggen. Daarbij zullen we echter niet uitgaan van het begrip funktionele afhankelijkheid tussen attributen van een database relatie. In eerste instantie beperken we ons doelbewust tot de inhoud van een database relatie in zijn geheel. Pas in tweede instantie stellen we eisen aan attributen van database relaties. Deze aanpak komt voor een groot gedeelte overeen met de aanpak die Smith en Smith voorstelden in [7].

Iedere database relatie gebruiken we voor gegevens over een enkele soort van subjecten (hiermee introduceren we voor het eerst semantiek in ons gegevensmodel). Voorbeelden van subjecten die we in een database model kunnen tegenkomen zijn: artikelen, klanten, orders, studenten, docenten, afdelingen. Nadat we hebben vastgesteld welke soorten van subjecten voor onze toepassingen van belang zijn, gaan we aangeven welke attributen we nodig achten voor iedere soort van subjecten. Hierbij zullen we altijd een attribuut of kollektie van attributen reserveren voor identifikatiedoeleinden. Om verschillende subjecten van dezelfde soort (welke dus voorkomen in dezelfde database relatie) van elkaar te onderscheiden, zullen we aan ieder subject een uniek kenmerk toekennen. Deze unieke identifikatie wordt in de database literatuur gewoonlijk de (primaire) sleutel van een database relatie genoemd. Database relaties welke gekarakteriseerd worden door een unieke identifikatie van de tupels zullen we voortaan subject relaties noemen. Dit brengt ons tot de volgende definitie van subject relatie:

DEFINITIE. Relatie $S.s_+,a_-$ is een subject relatie in s_+ wanneer iedere $(s_+,a_-) \in S_i$ en $(s_+,a'_-) \in S_i$ impliceert $a'_- = a_-$.

In een deklaratie zal een subject relatie $S.s_+,a_-$ met sleutel s_+ worden aangegeven met $S.[s_+],a_-$.

Stel dat we gegevens willen vastleggen over studenten. Wanneer de variabelen: identiteitsnummer (*sn*), naam (*snaam*), woonplaats (*loc*) als volgt zijn gedefinieerd:

var sn: int
var snaam: string
var loc: string

dan kunnen we deze gegevens over studenten representeren in de volgende subject relatie:

rel STUDENT. [*sn*], *snaam, loc*

Zoals gewoonlijk in de database literatuur laten we de definities van de onderliggende domeinen in het vervolg van dit artikel achterwege.

Naast de betekenis die we toekennen aan iedere database relatie, moeten we ook de samenhang die er tussen diverse database relaties bestaat in ons model tot uitdrukking brengen. Deze samenhang vormt ook een onderdeel van de semantiek van de gegevens. In een netwerkmodel of hierarchisch model kunnen we deze samenhang vaak aangeven door middel van toegangspaden. In het relationele gegevensmodel echter onderkennen we geen toegangspaden zodat we in dit gegevensmodel deze samenhang op een andere manier (d.w.z. expliciet) moeten aangeven. Ook dit kan worden aangemerkt als een groot voordeel van het relationele model: alle vormen van samenhang komen op dezelfde wijze tot uitdrukking.

Een vorm van samenhang die we vaak tegenkomen kan worden aangegeven met de term subset invariant [9]. Deze vorm van samenhang treffen we aan bij niet-gelijksoortige subjecten. We zullen hiervan een drietal voorbeelden geven. Daarnaast is er een vorm van samenhang die we aantreffen bij gelijksoortige subjecten. Toepassingen hiervan vinden we vooral bij netwerktoepassingen (kritieke pad analyse, stuklijst toepassingen). Dit zullen we ook illustreren aan de hand van een aantal voorbeelden.

2.2.2. Samenhangende subjecten

Stel dat we in een database gegevens willen vastleggen over afdelingen in een universiteit. In het relationele gegevensmodel komt dit tot uitdrukking door de aanwezigheid van een subject relatie waarin gegevens voorkomen over afdelingen. Wanneer we bijvoorbeeld de volgende gegevens willen vastleggen voor iedere afdeling: afdelingscode (*an*), afdelingsnaam (*anaam*); dan hebben we dus te maken met de volgende subject relatie:

rel AFDELING. [*an*], *anaam*

Stel vervolgens dat we ook gegevens willen vastleggen over studenten in de universiteit. Wanneer we voor iedere student willen beschikken over de volgende gegevens: identiteitsnummer (*sn*), naam (*snaam*), woonplaats (*loc*) en studierichting (*an*), dan hebben we dus ook de volgende subject relatie in onze database:

rel STUDENT. [*sn*], *snaam, loc, an*

Iedere student volgt een studierichting die binnen de universiteit aanwezig is (waarbij een studierichting overeenkomt met een afdeling in de universiteit), dus is de verzameling van *an*-waarden in subject relatie *STUDENT* een deelverzameling van de verzameling van *an*-waarden in subject relatie *AFDELING*. We gebruiken hiervoor de volgende notatie:

inv STUDENT. an → AFDELING

We kunnen deze karakterisering ook als volgt definiëren:

DEFINITIE. Subject relatie $T.s_+,t_-$ is in s_+ samenhangend met subject relatie $S.[s_+],a_-$ wanneer iedere $(s_+,t_-) \in T_i$ impliceert $(s_+,a_-) \in S_i$.

Een tweede voorbeeld van samenhangende subjecten vinden we bij artikelen, orders en klanten. In dit geval is een order logisch verbonden met een artikel en een klant. Wanneer in dit geval de subject relaties als volgt zijn gedefinieerd:

rel ARTIKEL. [an], omschrijving, prijs, voorraad
rel ORDER. [an,kn], hoeveelheid
rel KLANT. [kn], knaam, loc

dan gelden hiervoor de volgende invarianten:

inv ORDER. an → ARTIKEL
inv ORDER. kn → KLANT

Een derde voorbeeld van samenhangende subjecten treffen we aan bij netwerkplanningen. In dit geval bevat de database gegevens over aktiviteiten (*rel AKT*) en voortgangsrelaties (*rel VOORTGANG* met sleutel (from-*an*,to-*an*)):

rel AKT. [an], uitvoerder, tijdsduur
rel VOORTGANG. [fan,tan], tijdsverschil

Hiervoor gelden de volgende invarianten:
inv VOORTGANG. fan → AKT
inv VOORTGANG. tan → AKT

In de volgende paragraaf laten we zien dat deze invarianten voor dit laatste voorbeeld onvoldoende zijn om de betekenis van de gegevens vast te leggen.

2.2.3. Geordende subjecten

Wanneer we een gegevensmodel definiëren voor assemblage toepassingen, dan moeten we eerst een relatie definiëren waarin gegevens over onderdelen

zijn vastgelegd. Dus in de eerste plaats hebben we te maken met de volgende subject relatie:

rel PART. [*pn*], *pnaam*, *prijs*

Vervolgens hebben we een subject relatie nodig waarin gegevens over de samenhang tussen verschillende onderdelen wordt vastgelegd. Onder andere moeten we in staat zijn om aan te geven dat een bepaald onderdeel *tpn* een direkte komponent is van een onderdeel *fpn* en daarin *mult* aantal keren voorkomt. Deze gegevens resulteren in de volgende subject relatie:

rel LINK. [*fpn*,*tpn*], *mult*

Vervolgens moeten we eisen dat geen enkel onderdeel zichzelf bevat als (direkte of indirekte) komponent. Met andere woorden voor de gegevensstrukturen moeten we een (logische) ordening introduceren. Deze consistentie-eisen kunnen we als volgt definiëren:

DEFINITIE. Relatie $S.[s],a\_$ is geordend in s door relatie $U.[fs,ts],u\_$ wanneer iedere $(s_0,s_1,u_{-1}) \in U_i$ en ... en $(s_{n-1},s_n,u_{-n}) \in U_i$ impliceert $(s_n,s_0,u_{-0}) \notin U_i$ en wanneer U samenhangt met S in fs en ts.

Voor het laatste voorbeeld noteren we dus de volgende eigenschappen:

inv LINK. *fpn* → *PART*	(*LINK* in *fpn* samenhangend met *PART*)
inv LINK. *tpn* → *PART*	(*LINK* in *tpn* samenhangend met *PART*)
inv LINK. *fpn*,*tpn* ↑ *PART*	(*LINK* in (*fpn*,*tpn*) niet-cyclisch op *PART*)

Bovenstaande invarianten tezamen definiëren dus een ordening op subject relatie *PART*. In dit geval wordt *LINK* de ordeningsrelatie genoemd. Een tupel (tpn,fpn,mult) wordt de inverse genoemd van een tupel (fpn,tpn,mult). Wanneer we dus *fpn* en *tpn* in relatie *LINK* met elkaar verwisselen, dan hebben we opnieuw een ordening van subjecten in *PART*.

Een tweede voorbeeld waarin associaties tussen subjecten van dezelfde soort voorkomen, treffen we aan bij kritieke pad analyse. Naast aktiviteiten treffen we hier subjecten aan welke voortgangsrelaties worden genoemd. In dit geval is het triviaal dat geen enkele aktiviteit zichzelf kan opvolgen. Dus ook hier hebben we te maken met een ordening van aktiviteiten. Wanneer de subject relaties als volgt zijn gedefinieerd:

rel AKT. [an], uitvoerder, tijdsduur
rel VOORTGANG. [fan,tan], tijdsverschil

dan zijn de invarianten:

inv VOORTGANG. fan → AKT
inv VOORTGANG. tan → AKT
inv VOORTGANG. fan,tan ↑ AKT

3. MANIPULATIE VAN GEGEVENS

Zoals we in het voorgaande deel van dit artikel hebben gezien, maken we bij het ontwerp van een database model gebruik van enkele bestaande wiskundige begrippen (waaronder het begrip relatie). Op grond hiervan zouden we misschien geneigd zijn om voor het ontwerpen van algoritmen ook gebruik te maken van bestaande wiskundige talen (zoals bijvoorbeeld de predikatenlogika). Het is dan ook niet zo verwonderlijk dat de eerste manipulatietalen voor relationele databases gebaseerd waren op begrippen uit de verzamelingsalgebra [4] of predikatenlogika [3].

Een nadeel van bestaande wiskundige talen voor gebruik bij databases is dat ze ons niet in staat stellen om gebruik te maken van semantische eigenschappen zoals deze bijvoorbeeld in het voorafgaande zijn geïntroduceerd. Hoewel andere manipulatietalen voor relationele databases, zoals Sequel [1] en Query-by-Example [11], niet gebaseerd zijn op bestaande wiskundige begrippen, gelden hiervoor in feite dezelfde nadelen.

Een ander nadeel ontstaat door het verschil in toepassingen. In de wiskunde gebruiken we meestal de existentiële- en universele-kwantor (de verzamelingsalgebra bevat operaties waarmee hetzelfde bereikt kan worden) omdat de toepassingen zich daarvoor lenen. Daarentegen zijn de selektiekriteria bij gegevensverwerking vaak aanzienlijk ingewikkelder dan in de wiskunde. De beschikbaarheid van alleen deze twee kwantoren zou hier leiden tot zeer vreemde en ondoorzichtige beschrijvingen.

We zullen met behulp van een aantal voorbeelden een datamanipulatie-taal introduceren waarbij de semantiek van gegevens als basis heeft gediend voor het ontwerp. Deze taal leent zich er dan ook voor om van de semantische eigenschappen van een database gebruik te maken. In een drietal artikelen [8,9,10] is ook uitvoerig over deze manipulatietaal voor databases gerapporteerd.

Om een vergelijking met andere wiskundige talen te vereenvoudigen, zullen we tevens een aantal voorbeelden opvoeren waarvan de alternatieve

formulering in de predikatenlogika is aan te geven. Het zijn de volgende problemen uit paragraaf 3.2:

PROBLEEM 4. Gebruik van de existentiële kwantor

$\{e.en \mid e \in E \wedge \exists_{a \in A} (e.loc = a.loc \wedge e.an = a.an)\}$

PROBLEEM 5. Gebruik van de universele kwantor

$\{e.loc \mid e \in E \wedge \forall_{a \in A} (e.loc \neq a.loc)\}$

Hiervoor kunnen we ook de existentiële kwantor gebruiken

$\{e.loc \mid e \in E \wedge \neg\exists_{a \in A} (e.loc = a.loc)\}$

PROBLEEM 6. Gebruik van twee kwantoren

$\{l.ln \mid l \in L \wedge \forall_{p \in P} \exists_{a \in A} (a.ln = l.ln \wedge a.pn = p.pn)\}$

PROBLEEM 7. Gebruik van drie kwantoren

$\{l.ln \mid l \in L \wedge \exists_{g \in G} \forall_{a \in A} \exists_{t \in T} (l.ln = t.ln \wedge g.gn = t.gn \wedge a.an = t.an)\}$

3.1. Basisbegrippen

Er is één eenvoudige taalkonstruktie die erg belangrijk is voor data-manipulatie. Deze constructie, die selektie-expressie wordt genoemd, is als volgt gedefinieerd (de accoladen { en } geven fakultatieve onderdelen van de expressie aan):

(*relatie* {*.doel*} {|*predikaat*} {*per subject*})

De selektie-expressie geeft aan dat tupels worden verkregen uit de database die:

- zijn afgeleid uit *relatie*
- attributen hebben zoals gespecificeerd in het *doel*
- voldoen aan het *predikaat*
- en zijn gepartitioneerd volgens *subject*.

In afwezigheid van:

.doel: zal alleen de aanwezigheid van tupels relevant zijn

| *predikaat*: zullen alle tupels worden geselekteerd

per subject: zal er geen partitionering plaatsvinden.

Deze taalkonstruktie neemt een centrale plaats in bij de formulering van algoritmen. Daarnaast zullen we andere taalelementen nodig hebben waarmee we gebruik kunnen maken van de semantiek van gegevens. Laatstgenoemde elementen zullen we echter introduceren aan de hand van voorbeelden.

3.2. Selektie

Bij de formulering van algoritmen zouden we ingewikkelde taalkonstrukties kunnen toestaan waardoor we in staat zouden zijn om ieder probleem te formuleren in één enkele opdracht. We hebben met opzet niet voor deze aanpak gekozen: ieder probleem moet worden opgesplitst in een klein aantal hanteerbare deelproblemen.

3.2.1. Subjecten

De eerste drie voorbeelden hebben betrekking op één enkele subject relatie. Subject relatie *S* bevat over studenten de volgende gegevens: identiteitsnummer (*sn*), naam (*snaam*), adres (*adres*), woonplaats (*loc*) en studierichting (*an*). Dus subject relatie *S* is als volgt gedefinieerd:

rel S.[sn], snaam, adres, loc, an

PROBLEEM 1. Selekteer de wiskunde studenten.
Gegevens over studenten kunnen we vinden in relatie *S*. Dus onafhankelijk van de gewenste attributen is onze selektie-expressie gericht op subject relatie *S*. Wanneer we geïnteresseerd zijn in de attributen: *sn,snaam* en *loc* dan bestaat het eerste deel van de selektie-expressie uit:

S.sn, snaam, loc

Deze gegevens zijn alleen gewenst voor studenten uit de wiskunde afdeling. Het predikaat ziet er dus als volgt uit:

an = "wiskunde"

De gewenste informatie wordt dus aangegeven met de volgende selektie-expressie:

(S.sn, snaam,loc | an = "wiskunde")

Tenslotte gebruiken we het woord *get* om aan te geven dat de geselekteerde informatie tevens het gewenste eindprodukt voorstelt:

(1.1) *get (S.sn,snaam,loc | an = "wiskunde")*

PROBLEEM 2. Geef alle studenten gegroepeerd volgens studierichting.
In dit voorbeeld moeten we alle tupels van subject relatie *S* onderbrengen in disjunkte klassen zodat met iedere afdeling een kollektie van tupels korrespondeert. We krijgen dus de volgende opdracht:

(2.1) *get* (*S.sn,snaam,loc,an per an*)

PROBLEEM 3. Geef de woonplaatsen van studenten.
In dit geval willen we informatie over woonplaatsen. Aangezien we niet beschikken over een subject relatie waarin gegevens over woonplaatsen zijn vastgelegd (en omdat een selektie-expressie niet impliciet een projektie tot gevolg heeft) zullen we eerst een tijdelijke relatie moeten creëren (3.1) alvorens we over kunnen gaan tot selektie (3.2). De algoritme bestaat dus uit de volgende opdrachten:

(3.1) *rel L.*[*loc*]*: (S.loc)*

(3.2) *get* (*L.loc*)

3.2.2. Samenhangende subjecten

We zullen nu een aantal voorbeelden geven waarin gegevens moeten worden geselekteerd uit een aantal samenhangende subject relaties. In het eerste voorbeeld speelt de sleutel van een subject relatie een belangrijke rol. De laatste twee voorbeelden uit deze serie laten zien hoe het gebruik van de (existentiële en universele) kwantoren uit de predikatenlogika kan worden vermeden. In deze voorbeelden laten we zien hoe in het algemeen vergelijkingen met verzamelingen kunnen worden omgezet in vergelijkingen met aantallen (cardinaliteiten).

Het eerste voorbeeld uit deze serie neemt de volgende subject relaties als uitgangspunt (subject relatie *A* bevat gegevens over afdelingen en subject relatie *E* bevat gegevens over employees):

rel A.[*an*]*, anaam, loc*
rel E.[*en*]*, enaam, loc, an*
inv E. an → A

PROBLEEM 4. Welke employees werken in hun woonplaats.
In dit probleem zijn we geïnteresseerd in gegevens over employees. Dus het einddoel bestaat voor iedere geselecteerde employee uit de volgende gegevens:

E.en, enaam, loc, an

Met iedere employee korrespondeert precies één afdeling (een gevolg van de subset invariant). We gebruiken de notatie *A*[*an*] om deze korresponderende afdeling aan te geven. Dus het predikaat ziet er als volgt uit:

loc = A[*an*]*. loc*

De algoritme bestaat dus uit de volgende opdracht:

(4.1) *get* (*E.en,enaam,loc,an* | *loc = A*[*an*]*.loc*)

PROBLEEM 5. In welke woonplaatsen is geen enkele afdeling gevestigd. Aangezien er geen subject relatie bestaat met gegevens over woonplaatsen, gaan we in (5.1) eerst een tijdelijke relatie creëren. Vervolgens breiden we deze subject relatie uit met een attribuut waarin het aantal korresponderende employees staat aangegeven (5.2). Tenslotte levert selektie (5.3) het gewenste resultaat:

(5.1) *rel L.*[*loc*]*: (E.loc)*

(5.2) *ext L.aantal: count* (*A per loc*)

(5.3) *get* (*L.loc* | *aantal = 0*)

Het volgende probleem heeft betrekking op subject relaties welke gegevens bevatten over leveranciers (*rel L*), produkten (*rel P*) en assortimenten (*rel A*). De definitie van de database luidt:

rel L.[*ln*]*, lnaam, adres*
rel P.[*pn*]*, pnaam, voorraad*
rel A.[*ln,pn*]*, prijs*
inv A.ln → L
inv A.pn → P

PROBLEEM 6. Selekteer de leveranciers die alle produkten in hun assortiment hebben.
Het gewenste resultaat kunnen we als volgt aangeven:

get (*L.ln,lnaam,adres* | *"alle produkten in het assortiment"*)

De voorwaarde *"alle produkten in het assortiment"* die we voor iedere leverancier *ln* moeten evalueren kunnen we ook als volgt noteren:

$A_{ln}.\ pn \supseteq P.pn$

waarbij A_{ln} de deelverzameling uit *A* aangeeft welke korrespondeert met leverancier *ln*.

Om deze voorwaarde te evalueren kunnen we gebruik maken van de semantiek van de gegevens, met name van de subset invarianten. We weten dat de produkten in de assortimenten uitsluitend betrekking hebben op bestaande produkten in subject relatie *P*, dus geldt:

$A_{ln}.pn \subseteq P.pn$

We mogen dus ook evalueren:

$A_{ln}.pn = P.pn$

Maar op dit moment hebben we nog niet het volle profijt gehad van de samenhang tussen de subject relaties. Alle verzamelingen in een database zijn eindig. We hebben al gezien dat $A_{ln}.pn \subseteq P.pn$; we mogen in plaats van verzamelingen dus ook aantallen met elkaar vergelijken:

count $(A_{ln}.pn)$ = *count* $(P.pn)$

Omdat iedere waarde voor *pn* hoogstens eenmaal voorkomt in een assortiment A_{ln} (want samen vormen *pn* en *ln* de sleutel van *A*), en omdat iedere waarde voor *pn* hoogstens eenmaal voorkomt in *P* (want *pn* is de sleutel van *P*), mogen we ook de volgende voorwaarde evalueren:

count (A_{ln}) = *count* (P)

Voor iedere leverancier moeten we het aantal produkten in het assortiment bepalen. We gaan hiervoor subject relatie *L* tijdelijk uitbreiden met een attribuut *asm*, dat als volgt is gedefinieerd:

ext L.asm: count (A per ln)

Daarnaast hebben we het aantal produkten nodig:

var pakket: count (P)

De volledige algoritme voor dit probleem ziet er dus als volgt uit:

(6.1) *ext L.asm: count (A per ln)*

(6.2) *var pakket: count (P)*

(6.3) *get (L.ln,lnaam,adres | asm = pakket)*

Het laatste voorbeeld uit deze serie heeft betrekking op subject relaties welke gegevens bevatten over leveranciers (*rel L*), artikelen (*rel A*), gebruikers (*rel G*), klanten (*rel K*) en leveranties (*rel T*):

rel L.[ln], lnaam, loc
rel A.[an], anaam, prijs
rel G.[gn], gnaam, loc
rel K.[ln,gn]
rel T.[ln,an,gn], hoeveelheid
inv K.ln → L
inv K.gn → G
inv T.ln,gn → K
inv T.an → A

PROBLEEM 7. Welke leveranciers hebben een klant waaraan ze alle artikelen leveren.
Voor de oplossing van dit probleem bepalen we eerst het aantal artikelen dat met iedere klant korrespondeert. Met andere woorden: we bepalen het aantal leveranties (d.w.z. tupels uit *T*) dat korrespondeert met een klant (d.w.z. tupel uit *K*). Indien dit aantal leveranties gelijk is aan het aantal artikelen, dan neemt de betreffende klant alle artikelen af van de betreffende leverancier; m.a.w. de leverancier voldoet aan de vraagstelling. De volledige algoritme voor dit probleem luidt dus:

(7.1) *ext K.aantal: count (T per ln,gn)*

(7.2) *var pakket: count (A)*

(7.3) *ext L.aantal: count (K | aantal = pakket per ln)*

(7.4) *get (L.ln,lnaam,loc | aantal > 0).*

In het voorgaande zijn een aantal voorbeelden gegeven waarbij de verzamelingsfunktie *count* werd gebruikt bij de vergelijking van gegevensverzamelingen. Het ligt voor de hand dat naast deze funktie ook andere verzamelingsfunkties mogen worden gebruikt. In het volgende zullen we ter illustratie hiervan een voorbeeld geven waarin de funktie *max* zal voorkomen. Deze funktie levert de maximale waarde in een kollektie van numerieke waarden; wanneer de kollektie leeg is dan de waarde *nil* (deze waarde heeft geen gevolgen voor een wijzigingsopdracht).

3.2.3. Geordende subjecten

Stel dat een projekt bestaat uit een aantal elkaar niet overlappende aktiviteiten. Tussen deze aktiviteiten bestaan zekere voortgangsrelaties

(welke ook wel precedence relations worden genoemd). Het gegevensmodel voor deze toepassing bestaat dus uit twee subject relaties. Een subject relatie wordt gebruikt om gegevens vast te leggen over aktiviteiten (*rel A*), terwijl de ander wordt gebruikt om gegevens vast te leggen over voortgangsrelaties (*rel P*). We hebben dus de volgende database:

rel A.[an], anaam, duur
rel P.[fan,tan], dmax, dmin
inv P.fan → A
inv P.tan → A
inv P.fan,tan ↑ A

In subject relatie *P* geeft *dmax* (respektievelijk *dmin*) de maximale (respektievelijk minimale) tijdsduur aan welke moet liggen tussen de aanvang van aktiviteit *fan* en de aanvang van aktiviteit *tan* (tevens geldt: $dmin \geq A[fan].duur$ en $dmax \geq A[fan].duur$).

Gegeven de minimale tijdsverschillen die er moeten liggen tussen de aanvangstijdstippen van de aktiviteiten (*dmin*) en de tijdsduren van de aktiviteiten (*duur*), kunnen we de kortst mogelijke periode berekenen welke nodig is voor de uitvoering van het gehele projekt. Het is bekend dat hierbij voor iedere aktiviteit een tijdsinterval bestaat waarbinnen de aktiviteit mag beginnen zonder dat de projektduur hierdoor wordt beïnvloed. Dit interval wordt bepaald door het vroegst mogelijke aanvangstijdstip (rekening houdend met alle voorafgaande aktiviteiten) en het laatst toelaatbare aanvangstijdstip (rekening houdend met alle opvolgende aktiviteiten). Kritieke aktiviteiten zijn de aktiviteiten waarvoor dit interval de lengte nul heeft.

PROBLEEM 8. Bepaal de kritieke aktiviteiten.
Voor iedere aktiviteit berekenen we eerst de kortste periode die nodig is voor uitvoering van alle voorgaande aktiviteiten. Op dezelfde wijze berekenen we de kortste periode die nodig is voor uitvoering van de aktiviteit plus alle opvolgende aktiviteiten. Wanneer voor een aktiviteit de som van deze twee tijdsduren gelijk is aan de minimale projektduur, dan hebben we te maken met een kritieke aktiviteit.
Het vroegst mogelijke begintijdstip (*est*) wordt berekend in (8.1) en (8.2). Merk op dat de waarde van attribuut $A[fan].est$ in (8.2) nodig is voor de berekening van $A[tan].est$. We moeten de wijzigingen (*upd*) dus aanbrengen in de volgorde zoals is aangegeven in subject relatie *P* (*per ↑ tan*).

Alle aktiviteiten moeten tenminste *duur* tijdseenheden beginnen voor het eindtijdstip van het projekt. De laatst toegestane aanvangstijdstippen (*sbc*) worden dus bepaald in (8.3) en (8.4). Hierbij hebben we in (8.4) de inverse volgorde nodig voor de bepaling van *sbc* (*per* ↓ *fan*).
De minimale periode die nodig is voor de uitvoering van het gehele projekt (*spp*) wordt berekend in (8.5). Kritieke aktiviteiten worden dus verkregen in (8.6).

(8.1) *ext A.est: int*(0)

(8.2) *upd A.est: max*(*P.dmin*+*A*[*fan*].*est per* ↑ *tan*)

(8.3) *ext A.sbc: int*(*duur*)

(8.4) *upd A.sbc: max*(*P.dmin*+*A*[*tan*].*sbc per* ↓ *fan*)

(8.5) *var spp: max*(*P.est*+*sbc*)

(8.6) *get* (*A.an,anaam* | *est*+*sbc* = *spp*)

Stel dat we gegevens willen vastleggen over onderdelen die van belang zijn voor een assemblage proces. Sommige onderdelen worden hierbij als elementaire onderdelen beschouwd omdat ze moeten worden ingekocht. Andere onderdelen worden geassembleerd met behulp van elementaire onderdelen. In een relationele database voor deze toepassing spelen twee samenhangende subject relaties een rol. Een subject relatie zal gegevens bevatten over onderdelen, d.w.z. elementaire en samengestelde onderdelen (*rel P*). Daarnaast hebben we een subject relatie nodig voor de vastlegging van gegevens over de samenhang tussen onderdelen (*rel L*). Geen enkel onderdeel kan zichzelf bevatten als komponent. In de database wordt aan deze eis voldaan wanneer geldt dat subject relatie *P* wordt geordend door subject relatie *L*. We krijgen dus het volgende database model:

rel P.[*pn*], *pnaam, voorraad, oqty*
rel L.[*fpn,tpn*], *mult*
inv L.fpn → *P*
inv L.tpn → *P*
inv L.fpn,tpn ↑ *P*

Voor ieder onderdeel is de bestelde hoeveelheid vastgelegd in attribuut *oqty*. Om aan deze bestellingen te voldoen kan het nodig zijn dat samengestelde onderdelen worden geassembleerd. Wanneer het aantal aanwezige elementaire onderdelen niet voldoende is voor dit assemblage proces, moet er

worden ingekocht.

PROBLEEM 9. Welke elementaire onderdelen moeten worden ingekocht.
Voor ieder onderdeel is de benodigde hoeveelheid (*nodig*) opgebouwd uit een tweetal komponenten:
- de bestelde hoeveelheid (*oqty*)
- de hoeveelheid die nodig is om omvattende onderdelen te assembleren (*fqty*).

We kunnen dit als volgt formuleren:

def P.nodig: oqty + fqty.

Om aan de vraag van een onderdeel *fpn* te kunnen voldoen zullen we dus:

*mult * P[fpn].nodig*

onderdelen *tpn* nodig hebben.
Gebruik van de verzamelingsfunktie *total* resulteert in (9.3). Tenslotte vinden we de elementaire onderdelen in (9.4) door gebruik te maken van de verzamelingsfunktie *empty* (boolean funktie met waarde: *collectie = leeg*).
De algoritme voor dit probleem luidt dan als volgt:

(9.1) *ext P.fqty: int*(0)

(9.2) *def P.nodig: oqty + fqty*

(9.3) *upd P.fqty: total (L.mult*P[fpn].nodig per ↑ tpn)*

(9.4) *ext P.elem: empty (L per fpn)*

(9.5) *get (P.pn,pnaam,nodig-voorraad | elem and nodig > voorraad).*

LITERATUUR

[1] CHAMBERLIN, D.D. & R.F. BOYCE, Sequel: *A structured English Query Language*, Proc. 1974 ACM SIGMOD Workshop on Data Description, Access and Control, 249-264.

[2] CODD, E.F., *A Relational Model of Data for Large Shared Data Banks*, Comm. ACM 13, 6 (June 1970), 377-387.

[3] CODD, E.F., *A Data Base Sublanguage Founded on the Relational Calculus*, Proc. 1971 ACM SIGFIDET Workshop, 35-68.

[4] CODD, E.F., *Relational Completeness of Data Base Sublanguages*, Courant Computer Science Symposia Vol. 6: *Data Base Systems*, Prentice-Hall, New York, 1977, 65-98.

[5] CODD, E.F., *Further Normalization of the Data Base Relational Model,* Courant Computer Science Symposia Vol. 6: *Data Base Systems,* Prentice-Hall, New York, 1972, 33-64.

[6] DATE, C.J., *An Introduction to Database Systems,* Addison-Wesley, 1977 (2nd edition).

[7] SMITH, J.M. & D.C.P. SMITH, *Database Abstractions: Aggregation,* Comm. ACM 20, 6 (June 1977), 405-413.

[8] TER BEKKE, J.H., *An Alternative Formulation of Queries on a Relational Database,* Technical Note, Technological University Eindhoven, Department of Mathematics, July 1975.

[9] TER BEKKE, J.H., *A Data Manipulation Language for Relational Data Structures,* Systems for Large Data Bases (ed. P.C. Lockemann and E.J. Neuhold), North-Holland, Amsterdam, 1976, 159-168.

[10] TER BEKKE, J.H., *Semantiek van gegevens als uitgangspunt voor manipulatie van gegevens,* Informatie 19, 9 (September 1977), 491-497.

[11] ZLOOF, M.M., *Query by Example,* Proc. NCC 44 (May 1975).

UITGAVEN IN DE SERIE MC SYLLABUS

Onderstaande uitgaven zijn verkrijgbaar bij het Mathematisch Centrum, 2e Boerhaavestraat 49 te Amsterdam-1005, tel. 020-947272.

MCS 1.1 F. GÖBEL & J. VAN DE LUNE, *Leergang Besliskunde, deel 1: Wiskundige basiskennis*, 1965. ISBN 90 6196 014 2.

MCS 1.2 J. HEMELRIJK & J. KRIENS, *Leergang Besliskunde, deel 2: Kansberekening*, 1965. ISBN 90 6196 015 0.

MCS 1.3 J. HEMELRIJK & J. KRIENS, *Leergang Besliskunde, deel 3: Statistiek*, 1966. ISBN 90 6196 016 9.

MCS 1.4 G. DE LEVE & W. MOLENAAR, *Leergang Besliskunde, deel 4: Markovketens en wachttijden*, 1966. ISBN 90 6196 017 7.

MCS 1.5 J. KRIENS & G. DE LEVE, *Leergang Besliskunde, deel 5: Inleiding tot de mathematische besliskunde*, 1966. ISBN 90 6196 018 5.

MCS 1.6a B. DORHOUT & J. KRIENS, *Leergang Besliskunde, deel 6a: Wiskundige programmering 1*, 1968. ISBN 90 6196 032 0.

MCS 1.6b B. DORHOUT, J. KRIENS & J.TH. VAN LIESHOUT, *Leergang Besliskunde, deel 6b: Wiskundige programmering 2*, 1977. ISBN 90 6196 150 5.

MCS 1.7a G. DE LEVE, *Leergang Besliskunde, deel 7a: Dynamische programmering 1*, 1968. ISBN 90 6196 033 9.

MCS 1.7b G. DE LEVE & H.C. TIJMS, *Leergang Besliskunde, deel 7b: Dynamische programmering 2*, 1970. ISBN 90 6196 055 X.

MCS 1.7c G. DE LEVE & H.C. TIJMS, *Leergang Besliskunde, deel 7c: Dynamische programmering 3*, 1971. ISBN 90 6196 066 5.

MCS 1.8 J. KRIENS, F. GÖBEL & W. MOLENAAR, *Leergang Besliskunde, deel 8: Minimaxmethode, netwerkplanning, simulatie*, 1968. ISBN 90 6196 034 7.

MCS 2.1 G.J.R. FÖRCH, P.J. VAN DER HOUWEN & R.P. VAN DE RIET, *Colloquium Stabiliteit van differentieschema's, deel 1*, 1967. ISBN 90 6196 023 1.

MCS 2.2 L. DEKKER, T.J. DEKKER, P.J. VAN DER HOUWEN & M.N. SPIJKER, *Colloquium Stabiliteit van differentieschema's, deel 2*, 1968. ISBN 90 6196 035 5.

MCS 3.1 H.A. LAUWERIER, *Randwaardeproblemen, deel 1*, 1967. ISBN 90 6196 024 X.

MCS 3.2 H.A. LAUWERIER, *Randwaardeproblemen, deel 2*, 1968. ISBN 90 6196 036 3.

MCS 3.3 H.A. LAUWERIER, *Randwaardeproblemen, deel 3*, 1968. ISBN 90 6196 043 6.

MCS 4 H.A. LAUWERIER, *Representaties van groepen*, 1968. ISBN 90 6196 037 1.

MCS 5 J.H. VAN LINT, J.J. SEIDEL & P.C. BAAYEN, *Colloquium Discrete wiskunde*, 1968.
ISBN 90 6196 044 4.

MCS 6 K.K. KOKSMA, *Cursus ALGOL 60*, 1969. ISBN 90 6196 045 2.

MCS 7.1 *Colloquium Moderne rekenmachines, deel 1*, 1969. ISBN 90 6196 046 0.

MCS 7.2 *Colloquium Moderne rekenmachines, deel 2*, 1969. ISBN 90 6196 047 9.

MCS 8 H. BAVINCK & J. GRASMAN, *Relaxatietrillingen*, 1969.
ISBN 90 6196 056 8.

MCS 9.1 T.M.T. COOLEN, G.J.R. FÖRCH, E.M. DE JAGER & H.G.J. PIJLS, *Elliptische differentiaalvergelijkingen, deel 1*, 1970.
ISBN 90 6196 048 7.

MCS 9.2 W.P. VAN DEN BRINK, T.M.T. COOLEN, B. DIJKHUIS, P.P.N. DE GROEN, P.J. VAN DER HOUWEN, E.M. DE JAGER, N.M. TEMME & R.J. DE VOGELAERE, *Colloquium Elliptische differentiaalvergelijkingen, deel 2*, 1970.
ISBN 90 6196 049 5.

MCS 10 J. FABIUS & W.R. VAN ZWET, *Grondbegrippen van de waarschijnlijkheidsrekening*, 1970. ISBN 90 6196 057 6.

MCS 11 H. BART, M.A. KAASHOEK, H.G.J. PIJLS, W.J. DE SCHIPPER & J. DE VRIES, *Colloquium Halfalgebra's en positieve operatoren*, 1971.
ISBN 90 6196 067 3.

MCS 12 T.J. DEKKER, *Numerieke algebra*, 1971. ISBN 90 6196 068 1.

MCS 13 F.E.J. KRUSEMAN ARETZ, *Programmeren voor rekenautomaten; De MC ALGOL 60 vertaler voor de EL X8*, 1971. ISBN 90 6196 069 X.

MCS 14 H. BAVINCK, W. GAUTSCHI & G.M. WILLEMS, *Colloquium Approximatietheorie*, 1971. ISBN 90 6196 070 3.

MCS 15.1 T.J. DEKKER, P.W. HEMKER & P.J. VAN DER HOUWEN, *Colloquium Stijve differentiaalvergelijkingen, deel 1*, 1972. ISBN 90 6196 078 9.

MCS 15.2 P.A. BEENTJES, K. DEKKER, H.C. HEMKER, S.P.N. VAN KAMPEN & G.M. WILLEMS, *Colloquium Stijve differentiaalvergelijkingen, deel 2*, 1973. ISBN 90 6196 079 7.

MCS 15.3 P.A. BEENTJES, K. DEKKER, P.W. HEMKER & M. VAN VELDHUIZEN, *Colloquium Stijve differentiaalvergelijkingen, deel 3*, 1975.
ISBN 90 6196 118 1.

MCS 16.1 L. GEURTS, *Cursus Programmeren, deel 1: De elementen van het programmeren*, 1973. ISBN 90 6196 080 0.

MCS 16.2 L. GEURTS, *Cursus Programmeren, deel 2: De programmeertaal ALGOL 60*, 1973. ISBN 90 6196 087 8.

MCS 17.1 P.S. STOBBE, *Lineaire algebra, deel 1*, 1974. ISBN 90 6196 090 8.

MCS 17.2 P.S. STOBBE, *Lineaire algebra, deel 2*, 1974. ISBN 90 6196 091 6.

MCS 17.3 N.M. TEMME, *Lineaire algebra , deel 3*, 1976. ISBN 90 6196 123 8.

MCS 18 F. VAN DER BLIJ, H. FREUDENTHAL, J.J. DE IONGH, J.J. SEIDEL & A. VAN WIJNGAARDEN, *Een kwart eeuw wiskunde 1946-1971, Syllabus van de Vakantiecursus 1971*, 1974. ISBN 90 6196 092 4.

MCS 19 A. HORDIJK, R. POTHARST & J.Th. RUNNENBURG, *Optimaal stoppen van Markovketens*, 1974. ISBN 90 6196 093 2.

MCS 20 T.M.T. COOLEN, P.W. HEMKER, P.J. VAN DER HOUWEN & E. SLAGT, *ALGOL 60 procedures voor begin- en randwaardeproblemen*, 1976. ISBN 90 6196 094 0.

MCS 21 J.W. DE BAKKER (red.), *Colloquium Programmacorrectheid*, 1975. ISBN 90 6196 103 3.

MCS 22 R. HELMERS, F.H. RUYMGAART, M.C.A. VAN ZUYLEN & J. OOSTERHOFF, *Asymptotische methoden in de toetsingstheorie; Toepassingen van naburigheid*, 1976. ISBN 90 6196 104 1.

MCS 23.1 J.W. DE ROEVER (red.), *Colloquium Onderwerpen uit de biomathematica, deel 1*, 1976. ISBN 90 6196 105 X.

MCS 23.2 J.W. DE ROEVER (red.), *Colloquium Onderwerpen uit de biomathematica, deel 2*, 1976. ISBN 90 6196 115 7.

MCS 24.1 P.J. VAN DER HOUWEN, *Numerieke integratie van differentiaalvergelijkingen, deel 1: Eenstapsmethoden*, 1974. ISBN 90 6196 106 8.

MCS 25 *Colloquium Structuur van programmeertalen*, 1976. ISBN 90 6196 116 5.

MCS 26.1 N.M. TEMME (ed.), *Nonlinear analysis, volume 1*, 1976. ISBN 90 6196 117 3.

MCS 26.2 N.M. TEMME (ed.), *Nonlinear analysis, volume 2*, 1976. ISBN 90 6196 121 1.

MCS 27 M. BAKKER, P.W. HEMKER, P.J. VAN DER HOUWEN, S.J. POLAK & M. VAN VELDHUIZEN, *Colloquium Discretiseringsmethoden*, 1976. ISBN 90 6196 124 6.

MCS 28 O. DIEKMANN, N.M. TEMME (EDS), *Nonlinear Diffusion Problems*, 1976. ISBN 90 6196 126 2.

MCS 29.1 J.C.P. BUS (red.), *Colloquium Numerieke programmatuur, deel 1A, deel 1B*, 1976. ISBN 90 6196 128 9.

MCS 29.2 H.J.J. TE RIELE (red.), *Colloquium Numerieke programmatuur, deel 2*, 1976. ISBN 144 0.

* MCS 30 P. GROENEBOOM, R. HELMERS, J. OOSTERHOFF & R. POTHARST, *Efficiency begrippen in de statistiek*, 1977. ISBN 90 6196 149 1.

MCS 31 J.H. VAN LINT (red.), *Inleiding in de coderingstheorie*, 1976. ISBN 90 6196 136 X.

MCS 32 L. GEURTS (red.), *Colloquium Bedrijfssystemen*, 1976. ISBN 90 6196 137 8.

MCS 33 P.J. VAN DER HOUWEN, *Differentieschema's voor de berekening van waterstanden in zeeën en rivieren*, 1977. ISBN 90 6196 138 6.

MCS 34 J. HEMELRIJK, *Oriënterende cursus mathematische statistiek*, ISBN 90 6196 139 4.

MCS 35 P.J.W. TEN HAGEN (red.), *Colloquium Computer Graphics*, 1977. ISBN 90 6196 142 4.

MCS 36 J.M. AARTS, J. DE VRIES, *Colloquium Topologische Dynamische Systemen*, 1977. ISBN 90 6196 143 2.

MCS 37 J.C. van Vliet (red.), *Colloquium Capita Datastructuren*, . ISBN 90 6196 159 9.

* MCS 38 T.H. Koornwinder (ed.), *Representations of locally compact groups with applications,* . ISBN 90 6196 161 0.

MCS 39 O.J. Vrieze & G.L. Wanrooij, *Colloquium Stochastische spelen,* 1978. ISBN 906196 167 X.

De met een * gemerkte uitgaven moeten nog verschijnen.