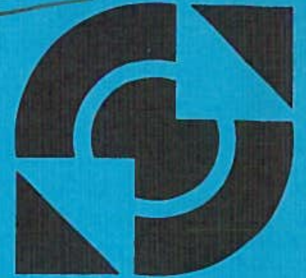
DIENSTEXEMPLAAR

M Fokkinga

Universiteit Twente

faculteit der
informatica



over het nut en de mogelijkheden van typering

voor het vak Structuur van Programmeertalen

door M.M. Fokkinga

vakcode: 211050
juni 1987
oplage: 160
prijs *f* 5,75

Inhoud

Samenvatting	0
1. Inleiding	1
2. Een definitie van typering	2
2.1 Een minitaal	2
2.2 De definitie	4
2.3 Voorbeelden van beoogde eigenschappen	8
3. Conventionele typeringen	11
3.1 De eenvoudige vorm	11
3.2 Recursieve typen	16
3.3 Subtypen	18
4. Tweede orde typeringen	22
4.1 De eenvoudigste vorm	24
4.2 Een ingewikkelder variant: de SVP typering	27
4.3 Typering als programmaspecificatie	33
5. Type-polymorfie	35
Aanhangsel A Een implementatie voor arrays	40
Aanhangsel B Type-equivalentie in de conventionele typering	42
Aanhangsel C Een unificatie algoritme	46
Literatuurverwijzingen	48

Druk: juni 1987

Samenvatting

In dit verhaal proberen we enig inzicht te verschaffen in het begrip "typering" zoals dat bij programmeertalen voorkomt. Het nut dat typering moet hebben staat daarbij voorop. Aan de hand van een minitaal en voorbeelden wordt de conventionele, zogenaamde eerste orde, typering beschreven; deze wordt gebruikt in talen zoals Pascal en Algol. Ook minder conventionele, en wel: hogere orde, typeringen en de zogenaamde type-polymorfie worden uitgelegd.

1. Inleiding

In dit verhaal proberen we enig inzicht te verschaffen in het begrip "typering" zoals dat bij programmeertalen voorkomt. We zullen daarbij de essentie beschrijven van de conventionele typering zoals die voorkomt bij talen zoals Algol 60, Algol 68, Pascal etc; en vervolgens ook de essentie aangeven van de typering in recentelijk ontworpen, en meestal nog experimentele, talen. In deze tweede versie van typering is het mogelijk om zg. "abstract data types" te programmeren (maar niet: te specificeren!).

In de inleiding op deel IV van "Programming Methodology" [Gries 1978] laat Gries in vijf bladzijden goed zien hoe weinig overeenstemming er in de vakliteratuur is over het begrip "type" en "typering". De verwarring, zin en onzin over die begrippen kan ik niet beter aangeven dan hij in die vijf bladzijden doet.

Een aspect van typeren komt in het geheel niet naar voren in het verhaal van Gries: het nut van typering wordt niet genoemd. En uiteindelijk zal zoiets als typering toch wel enig nut moeten hebben, om de aanwezigheid ervan te ondersteunen, en een vergelijking met "ongetypeerde" talen te kunnen doorstaan. We zullen een definitie van typering geven waarin dat nut een centrale rol speelt. Op grond van deze definitie kunnen de vragen, vermoedens en veronderstellingen van Gries behandeld worden.

Dankbetuiging. Ik ben Gerrit van der Hoeven erkentelijk voor leerzame gesprekken, en dank Thérèse ter Heide-Noll voor het type-werk.

2. Een definitie van typering

Aan de hand van een minitaal zullen we de definitie van typering geven. De definitie is op alle programmeertalen van toepassing; de minitaal dient eigenlijk alleen om voorbeelden te kunnen geven. Essentieel is dat de typering niet dient om de berekening, zoals uitgedrukt in een programma, te sturen. Daarom kiezen we een "ongetypeerde" minitaal; het nut van de typering --later aangebracht op deze minitaal-- kan dan pas aangetoond worden.

Het zal blijken dat onze definitie van typering nog vele realisatie toelaat. We zullen verscheidene realisaties noemen, en van ieder het nut aangeven.

2.1 Een minitaal

De minitaal zal niet wezenlijk verschillen van talen zoals Pascal en Algol 68. Om de definitie kort te houden zullen we de constructies erg veralgemenen. Er is bijvoorbeeld geen enkele beperking op het soort (het type?) van waarden dat door een funktie kan worden opgeleverd. Aldus hoeven we geen afzonderlijke syntactische en semantische categorieen te definieren voor de eenvoudige waarden, die wel als funktie-resultaat kunnen dienen, en de complexe waarden, die dat niet kunnen. De semantiek zullen we slechts informeel aangeven. Gezien de overeenkomsten met gewone programmeertalen, achten we misverstanden veroorzaakt door die informeelheid nauwelijks mogelijk.

Voor het gehele verhaal houden we ons aan de volgende notatie-afspraak:

x, y, z   varieren over identifiers,

e   varieert over expressies, (units in Algol 68).

In het algemeen hebben expressies een neveneffect. Een afzonderlijke categorie van statements is er daarom niet nodig. De taal is block-gestructureerd. De vorm van een block is

$\underline{\text{def}}\ x_1 = e_1, \ldots, x_n = e_n\ \underline{\text{in}}\ e\ \underline{\text{ni}}$

De geldigheid van de definitie $x_i = e_i$ strekt zich uit over $e_1, \ldots, e_n$ en e, met uitzondering van binnenblokken waarin opnieuw x_i wordt gedefinieerd. Twee bijzondere gevallen zijn de definities

$x = \underline{\text{newvar}}$

$x = (x_1, \ldots, x_n) \colon e$

In de eerste wordt door newvar een nieuwe stuk geheugen gereserveerd, en x is de variabele die naar dat stuk geheugen verwijst. Voorlopig leggen we niet vast wanneer het gereserveerde stuk geheugenruimte wordt vrij gegeven; m.a.w. we zeggen voorlopig niet of newvar net werkt als loc of als heap in Algol 68. In de tweede definitie is x een funktie met formele parameters $x_1,\dots,x_n$ en funktie romp e. (Let op, we spreken wel over funkties, omdat ze een waarde opleveren, maar i.h.a. hebben ze ook een neveneffect.)

We sommen nu alle vormen van expressies op, met voor ieder een kort kommentaar dat de semantiek in voldoende mate moet suggereren.

1	true, false	waarheidswaarden
2	e and e, e or e, not e	
3	0, 1, 2, ...	gehele getallen
4	e + e, e * e, ...	
5	$e_1; \dots; e_n$	sequentie
6	if e_1 then e_2 else e_3 fi	keuze
7	while e_1 do e_2 od	herhaling
8	deref e	dereferencing
9	$e_1 := e_2$	toekenning
10	newvar	variabele-creatie
11	$(x_1,\dots,x_n)$: e	routine tekst
12	e $(e_1,\dots,e_n)$	funktie-oproep
13	x	identifier
14	$\langle e_1,\dots, e_n \rangle$	record
15	e.1, ... , e.n	record selecties
16	def $x_1 = e_1, \dots, x_n = e_n$ in e ni	block

We maken ons niet druk over syntactische ambiguiteiten; middels haakjes kan alle dubbelzinnigheid vermeden worden. De semantiek van de funktie-oproep ligt vast doordat per definitie een oproep van een funktie $(x_1,\dots,x_n)$: e met argumenten $(e_1, .., e_n)$ equivalent is met het block def $x_1'=e_1,\dots,x_n'=e_n$ in e′ ni, waarbij in de funktie-tekst de $x_1,\dots,x_n$ systematisch hernoemd zijn in geheel nieuwe identifiers; dit om naamconfilicten te voorkomen.

2.2 De definitie

Beschouw nu eens de expressie true + 1. Er doen zich nu drie mogelijkheden voor:

1. evaluatie van deze expressie leidt tot een foutstop;
2. evaluatie eindigt met een of andere waarde, bijv. dezelfde waarde als waartoe 0 + 1 evalueert;
3. deze expressie is "illegaal", hij mag niet in beschouwing genomen worden.

Bij de eerste mogelijkheid dragen de waarden aangeduid door true en 1 kennelijk een kenmerk met zich mee die in de optellingsbewerking gebruikt wordt in een test of ze wel allebei als getal bedoeld waren. Deze testen noemen we domein-testen, naar [Tennent 1980]. Men spreekt ook wel van dynamische typering, omdat de waarden die in de loop van de berekening optreden "type"-informatie met zich mee dragen. Wij geven er de voorkeur aan die informatie "domein"informatie te noemen.

Bij de tweede mogelijkheid spreekt men wel van ongetypeerde talen. Niet alleen de programmateksten, maar ook de waarden zijn ongetypeerd. Deze ongetypeerdheid van waarden is heel gebruikelijk: in talen zoals Pascal kun je aan de bitstring, aangeduid door 1, niet meer zien of die bitstring door een integer-expressie werd opgeleverd, dan wel een boolean-expressie. In feite kan iedere bitstring op welhaast iedere manier gebruikt worden. Uiteindelijk zijn de bitstringen louter representaties van abstracte begrippen die de taalontwerper, of de programmeur, in z'n hoofd heeft. Wanneer de programmeur de representatie maar goed kent, kan het best handig zijn om eenzelfde bitstring nu eens op deze, en dan weer op die manier te interpreteren. Dat gebeurt bijvoorbeeld veelvuldig in Pascal met behulp van de variantrecord: de waarden van het type

```
    record case boolean of true: (i :integer);
                            false: (p :pointer) end
```

kunnen naar believe als integer of als pointer gebruikt worden. Zoiets brengt wel gevaren met zich mee. Als de representatie-keuzen veranderen, dan kan het zijn dat true + 1 niet meer tot dezelfde waarde als 0 + 1 (= 1) evalueert, maar tot bijv. 1 + 1 (=2). Een heel programma e kan dus onder een representatie gelijkwaardig zijn met 1, terwijl het onder een andere representatie gelijkwaardig is met 2. Zulk soort programma's moesten verboden worden. En dat is precies de derde mogelijkheid!

Bij de derde mogelijkheid wensen we de expressie true + 1 niet te beschouwen, bijvoorbeeld om redenen zoals hierboven aangeduid: de waarde hangt te veel

af van de toevallige representatie-keuzen. Het middel bij uitstek om precies de verzameling van "wel te beschouwen" expressies te definieren, is typering. De "wel te beschouwen" expressies heten goed-getypeerd, of goed typeerbaar. Typen zijn louter hulpmiddelen bij de typering: het zijn kenmerken toegewezen aan expressies (niet aan de waarden aangeduid door expressies!). Het is niet noodzakelijk dat typen ook letterlijk in de programmatekst genoemd worden, zoals we zullen zien. We geven deze visie in een definitie weer.

Definitie Typering is een algoritmische filter op expressies, zo dat de doorgelaten expressies een beoogde semantische eigenschap hebben; het filterproces is geformuleerd door aan de expressie en al zijn subexpressies kenmerken toe te wijzen, typen genaamd, en voor de kenmerken van een expressie eisen te stellen in termen van de kenmerken van de direkte onderdelen van die expressie.

Zodadelijk in deel 2.3 zullen we verscheidene "beoogde semantische eigenschappen" noemen. Merk nu al op dat per definitie typen louter hulpmiddelen zijn voor het filterproces. Het zal heel vaak zo zijn dat typen met een grammatica te definieren zijn, maar dat is niet noodzakelijk. (Als typen niet middels een grammatica te definieren zijn, bijvoorbeeld omdat het "oneindige" objecten zijn, dan is er misschien wel een grammatica voor notaties voor die typen.) Misschien dat de toewijzing van typen aan alle subexpressies gemakkelijker verloopt wanneer de programmeur daar enige ondersteuning toe biedt, door nl. bij definities en parameters typen uitdrukkelijk te vermelden*). Als dat al zo is, dan zal, volgens onze definitie van typering, het systematisch wegschrappen van al die typen geen effect mogen hebben op de semantiek. Met andere woorden, typen zijn semantisch gezien redundant. Ze dienen slechts om de goed-getypeerde, i.e. doorgelaten ofwel "legale", expressies te karakteriseren. Deze karakterisering is algoritmisch, d.w.z. in eindige tijd mechanisch uitvoerbaar. Men zegt ook wel "statisch" of "compile-time"; dit om te benadrukken dat volledige kennis van de waarden die tijdens "run-time", dus "dynamisch", optreden niet nodig is.

Bij de expressie true + 1 had de toewijzing van typen aan de subexpressies als volgt kunnen zijn. De subexpressie true krijgt "boolean", en 1 krijgt "integer" toegewezen, en het geheel "integer". De eis die aan de typen van e_1 en e_2 gesteld wordt in een samenstelling e_1+e_2 is: beide moeten "integer"

---

*) Zoiets is zeker nodig als de korrekte toewijzing van typen anders niet beslisbaar zou zijn.

zijn. Aan deze eis is niet voldaan bij de toewijzing die we net als voorbeeld noemden. Met deze nog niet gemotiveerde en slechts vaag geformuleerde, keuzen is true + 1 dus niet goed-getypeerd. We zullen straks uitgewerktere voorbeelden van mogelijke typeringen geven.

Tenslotte merken we op dat onze definitie extreme gevallen toelaat. Enerzijds kan de filter zo nauw zijn, dat alle expressies weggefilterd worden: voor iedere goed-getypeerde expressie --dat zijn er dus geen-- geldt dan zeker de beoogde eigenschap. Anderzijds kunnen we aan iedere expressie louter zichzelf als kenmerk toewijzen, en het gehele filterproces formuleren aan de hand van het kenmerk van het gehele programma. Aldus is willekeurige beslisbare eigenschap van expressies te nemen als het filter-criterium. Vermoedelijk zijn beide extrema niet te vermijden bij een algemene definitie van typering.

Naast de beoogde semantische eigenschappen zal de typering ook andere --meest syntactische-- eigenschappen mogelijk maken, doordat iedere expressie een type wordt toegewezen. We noemen er twee.

1. Generische identificatie. Twee of meer definities van eenzelfde naam zijn tegelijkertijd geldig; welke definitie door een toegepast voorkomen van die naam geidentificeerd wordt, wordt dan door de typen van de kontekst van dat voorkomen bepaald. Bijvoorbeeld: zowel optelling als concatenatie kunnen in Algol 68 door + worden aangeduid. In "3+x" wordt de optelling bedoeld, in "'3'+x" de concatenatie. De typen maken hier dus een handige notatie mogelijk. In het eigenlijke programma, dat door de compiler hieruit ge(re)construeerd wordt, zijn typen wederom redundant.

2. Coercies, de automatische invoeging van operaties zo dat de tekst, die zonder deze ingevoegde operaties niet type-korrekt zou zijn, het met deze operaties wel is. Bijvoorbeeld, in de meeste talen mag een "integer variablele" geschreven worden waar een "integer waarde" vereist wordt. De ingevoegde operatie is de "dereferencing", die van een variabele zijn waarde oplevert. Zie deel 3.3. Na invoeging van die operaties, die at compile-time gebeurt, zijn typen wederom redundant.

Het is natuurlijk de vraag of deze syntactische eigenschappen van de taal, mogelijk gemaakt dankzij de typering, de beoogde semantische eigenschappen niet verstoren. Het is voor de verantwoording van de taalontwerper om dat aan te tonen. Veel van de mogelijke beoogde semantische eigenschappen die we in deel 2.3 zullen noemen worden niet verstoord door generische identificatie en coercies.

Afspraak Wanneer we in het vervolg praten over "t-waarden", dan bedoelen we:

waarden aanduidbaar door expressies van type t.

En met de notatie

e^t (of e:t)

geven we aan dat de expressie e van type t is; in deze notatie is de kontekst van e --ten onrechte!-- niet vermeld.

2.3 Voorbeelden van beoogde eigenschappen

De beoogde semantische eigenschappen van expressies, waaraan de goed-getypeerde expressies voldoen, kunnen "prettig" zijn voor de programmeur, voor de compiler-schrijver, of voor beide (of voor geen van beide). Het is de ontwerper van de typering die de eigenschap uitkiest. Een goed ontwerper moet een zinvolle semantische eigenschap nastreven. Pas dan is de typering nuttig.

Het is opmerkelijk hoe weinig pogingen er zijn gedaan bij de ontwerpen van talen zoals Algol 60, Algol 68, Pascal en Ada om het nut van de typering precies te beschrijven. Het Algol 68 Revised Report zegt niets over het nut van de modes afzonderlijk, maar wel over de taal als geheel: "Algol 68 has been designed in such a way that most syntactical and many other errors can be detected easily before they lead to calamitous results" [van Wijngaarden 1976]. De DoD vereisten die tot Ada hebben geleid vereisen wel "strong typing", maar zeggen niets over het nut dat strong typing zou moeten hebben:

> "3A. Strong typing. The language shall be strongly typed. The type of each variable, array, record, expression, function and parameter shall be determinable during translation",

[Steelman 1978]. In de 'Rationale for Ada' worden deze drie redenen genoemd voor de introductie van typen:

- factorization of properties, maintainability
- abstraction, hiding of implementation details
- reliability

[Ichbiah et al 1979]. Maar het probleem bij al dit soort beweringen is dat ze niet precies genoeg geformuleerd zijn om bewezen of weerlegd te kunnen worden. Geen wonder dat er bij de ontwerpen van de typeringen zoveel rare, onhandige, verkeerde of te beperkende keuzen zijn gemaakt. Alvorens tot een uitwerking van de typering over te gaan, zal je eerst een goed idee van de beoogde semantische eigenschappen moeten hebben. Wij sommen nu een aantal mogelijkheden op.

1. Van iedere (goed-getypeerde) expressie zal de evaluatie (al dan niet met een foutstop) termineren. Zo'n eigenschap is inderdaad middels typering te garanderen. Voor de lambda-calculus geeft [Coppo 1978] een typering met slechts twee typen; die kun je dan aanduiden met 0 en 1. Het probleem is om zoveel mogelijk expressies waarvan de evaluatie termineert, goed-getypeerd te laten zijn. We zullen dit niet voor onze minitaal uitwerken.

2. Van iedere (goed getypeerde) expressie zullen alle domeintesten gedurende

de evaluatie met succes verlopen. Dus de evaluatie van een expressie kan niet tot een (bepaald soort) foutstop leiden. Dit is prettig voor de programmeur, maar ook voor de implementator van de taal omdat de domein-informatie van waarden kennelijk overbodig is. Vermoedelijk staat deze eigenschap voor ogen bij veel ontwerpen van typeringen. Maar wanneer de programmeur niet zelf nieuwe typen kan definieren, is deze eigenschap nog van weinig waarde. Want allerlei abstracte begrippen zullen in onze minitaal m.b.v. gehele getallen en waarheidswaarden gerepresenteerd moeten worden. De test of twee waarden wel beide gehele getallen zijn, garandeert in het geheel niet dat ze ook beide eenzelfde soort abstract begrip voorstellen.

3. Een semantische equivalentie van (goed-getypeerde) expressies is onafhankelijk van de representatie, gebruikt bij de implementatie. Dus als expressies e en 0 dezelfde waarde aanduiden, dan zal dat ook zo zijn en blijven wanneer we de implementatie iets wijzigen door nl. true en false andere waarden te laten aanduiden. Deze eigenschap is al genoemd in deel 2.2 bij mogelijkheid 3. Een precieze uitwerking wordt gegeven in [Fokkinga 1981a]. Zonder precieze formuleringen en bewijzen te geven wordt deze eigenschap ook in de praktijk wel nagestreefd met de gebruikelijke typeringen.

4. De grootte van de representatie van waarden, aangeduid door (goed getypeerde) expressies, is berekenbaar. Deze semantische eigenschap vergemakkelijkt de constructie van een implementatie, het vergemakkelijkt geenszins de taak van de programmeur. In Pascal is deze eigenschap nagestreefd: het type van arrays bevat ook de grenzen van het indexbereik. In Algol 68 behoren die grenzen niet tot het type.

5. De waarde aangeduid door (goed-getypeerde) expressies hangt niet af van het initiele geheugen. M.a.w. er kan geen gebruik gemaakt worden van niet geinitialiseerde variabelen. In onze minitaal zijn "aliassen" mogelijk: verschillende namen voor dezelfde variabele, bv. na def x = newvar in def y=x in ...ni ni . In een taal zonder "aliasing" lijkt de eigenschap af te dwingen, als we het type van een subexpressie een drietal laten zijn, te weten

- de verzameling gedeclareerde maar niet geinitialiseerde variabelen,
- de verzameling gedeclareerde en alreeds geinitialiseerde variabelen, en
- de verzameling gedeclareerde en in die expressie te initialiseren variabelen.

Iets dergelijks wordt in [Dijkstra 1976] uitgewerkt voor een minitaal zonder procedures. Ook in Aleph [Grune 1975] en CDL [Koster 1975] geldt deze eigenschap.

6. De waarde van een (goed-getypeerde) expressie zal voldoen aan de --als type gegeven-- specificatie. Er zijn typeringen ontwikkeld waarin inderdaad alle wiskundige eigenschappen uitdrukbaar zijn, en volgens welke alleen die expressies getypeerd zijn, die voldoen aan hun specificatie. In deel 4.3 komen we hierop terug.

3. Conventionele typeringen

3.1 De eenvoudigste vorm

In talen zoals Pascal en Algol 68 worden typen middels een grammatica gedefinieerd. Ze worden ook letterlijk in de programmatekst opgenomen, om de toewijzing van typen aan alle subexpressies te vergemakkelijken, en daarmee ook de algoritmische filter. In onderstaand voorbeeld laten we zien dat het aangeven van het type van gedefinieerde identifiers niet noodzakelijk is.

De typen worden gedefinieerd door de volgende grammatica.

$$t ::= \underline{int} \mid \underline{bool} \mid \ldots \mid (t_1, \ldots, t_n \rightarrow t) \mid \langle t_1, \ldots, t_n \rangle \mid \underline{ref}\ t \mid \underline{void}$$

De vorm $(t_1, \ldots, t_n \rightarrow t)$ is een funktie-type; de argumenten moeten type $t_1, \ldots, t_n$ hebben en het resultaat heeft dan type t. De vorm $\langle t_1, \ldots, t_n \rangle$ is een record-type; de componenten hebben achtereenvolgens type $t_1, \ldots, t_n$. De vorm ref t is het type van variabelen die waarden van type t kunnen aannemen. En void is het type van expressies die louter een neveneffect hebben, en geen waarde opleveren, zoals bijvoorbeeld een assignment en een herhaling. Kiezen we echter de semantiek van de minitaal zo dat een assignment en een herhaling wel een waarde opleveren, dan is het type void eigenlijk niet nodig. Om uitgebreide verhalen te voorkomen zullen we zoveel mogelijk proberen om deze aspecten te omzeilen; we laten het aan de lezer over om zelf aanvullingen te geven voor zijn favoriete keuzen.

De eisen die aan de typen van subexpressies gesteld worden, in conventionele typeringen, sommen we in onderstaande tabel op. Links staat de ontleding van expressie e in z'n subexpressies; de superscripten geven het type aan. Rechts staat de eis die gesteld wordt aan de typen.

nr	e^t	eisen
1	$\underline{true}^{t1}$, $\underline{false}^{t2}$	$t1 = t2 = t = \underline{bool}$
2	$e1^{t1}$ $\underline{and}$ $e2^{t2}$, ...	$t1 = t2 = t = \underline{bool}$
3	0^{t0}, 1^{t1}, ...	$t0 = t1 = t = \underline{int}$
4	$e1^{t1} + e2^{t2}$,...	$t1 = t2 = t = \underline{int}$
5	$(e1^{t1}; e2^{t2}; \ldots en^{tn})$	wordt overgelaten aan de lezer

6	$\underline{\text{it}}\ e1^{t1}\ \underline{\text{then}}\ e2^{t2}\ \underline{\text{else}}\ e3^{t3}$	$t1 = \underline{\text{bool}},\ t2=t3=t$
7	$\underline{\text{while}}\ e1^{t1}\ \underline{\text{do}}\ e2^{t2}\ \underline{\text{od}}$	$t1 = \underline{\text{bool}}$, overgelaten aan de lezer
8	$\underline{\text{deref}}\ e1^{t1}$	$t1 = \underline{\text{ref}}\ t$
9	$e1^{t1} := e2^{t2}$	$t1 = \underline{\text{ref}}\ t2$, overgelaten aan de lezer
10	$\underline{\text{newvar}}^{t1}$	$t1 = t = \underline{\text{ref}}\ t'$ voor een of andere t'
11	$(x1^{t1},..,xn^{tn}):\ e0^{t0}$	$(t1,..,tn \rightarrow t0) = t$
12	$e0^{t0}(e1^{t1},..,en^{tn})$	$t0 = (t1,..,tn \rightarrow t)$
13	x^{t1}	$t1 = t$
14	$\langle e1^{t1},..,en^{tn}\rangle$	$\langle t1,..,tn\rangle = t$
15	$e0^{t0}.3$	$t0=\langle t1,..,tn\rangle$ en $t3 = t$ voor een of andere $t1,..,tn$
16	$\underline{\text{def}}\ ...xi^{txi} = ei^{ti}..\ \underline{\text{in}}\ e0^{t0}\ \underline{\text{ni}}$	$txi = ti$ (voor alle i), $t0 = t$

We hebben niet gezegd hoe de typen worden gevonden; alleen aan welke eisen ze moeten voldoen. Het ligt voor de hand dat in het geldigheidsbereik van een definitie $x^t=e^t$ alle voorkomens van x precies het type t krijgen toegewezen. Dit is in de conventionele typeringen het geval. Als dan ook nog bij iedere definitie en bij iedere formele parameter (tesamen: bij iedere introductie) het type van de identifiers expliciet vermeld wordt, dan is door de eisen het type van iedere subexpressie eenduidig bepaald. Dit is met inductie naar de opbouw van expressies te bewijzen.

Een voorbeeld. Beschouw het volgende programmafragment.

```
def succ = (x): x + 1
  , double = (f,y): f(f(y))
in ...double (succ,...)....
```

Volgens bovenstaande eisen is er geen andere type-toewijzing mogelijk dan de volgende.

- x krijgt het type int, op grond van eis 4
- succ krijgt het type (int → int), op grond van eis 11 en 16,
- f krijgt het type van succ, op grond van de aanroep van double en eisen 11, 12, 16
- y krijgt het type int,

double krijgt dus uiteindelijk type ((int → int), int → int).

Dat programmafragment is dus goed-getypeerd [handgeschreven: goed te typeren], en dus geldt de beoogde semantische eigenschap. Die hebben we weliswaar niet expliciet geformuleerd, en evenmin

hebben we bewezen dat de conventionele typering zoals hierboven geschetst die eigenschap waar maakt. Maar te denken valt bijv. aan de afwezigheid van foutstoppen doordat een boolean bij een integer opgeteld zou worden.

Wanneer de aanroep "double(succ,...)" niet binnen gezichtsbereik was geweest, dan hadden we expliciet een type bij f moeten vermelden, om het type van double te kunnen bepalen. Een alternatief zou zijn om niet het type van f te vermelden, maar een zg. "voorbeeld expressie" waaruit dat type wel bepaald kan worden. Bij ingewikkelde typen kunnen dergelijke voorbeeld expressies soms veel korter te formuleren zijn. De definitie van double zou dan kunnen luiden:

def double = (f eg succ, y): f(f(y))

Opmerking. Het lijkt nu misschien eenvoudig om de taal uit te breiden met type-definities. Dan moet de typering, d.w.z. de tabel met eisen voor de type-toewijzing, ook uitgebreid worden. Dat gebeurt heel algemeen in Hoofdstuk 4 en 5 zo dat de beoogde semantische eigenschappen behouden blijven (vermoedelijk; een formeel bewijs is er nog niet in alle gevallen). Conventionele uitbreidingen, waarvoor sommige beoogde semantische eigenschappen zeker niet behouden blijven, worden behandeld in Aanhangsel B; het zijn de type-equivalentie-regels van Algol 68 en van Pascal.

Een nuttige datastructurering die ogenschijnlijk niet in de minitaal aanwezig is, is de zogenaamde onderscheiden vereniging. Men kan aantonen dat met de SVPtypering van deel 4.2 die structurering zelf te programmeren is. Wij zullen hem nu echter naast de groepvorming en funktievorming als nieuw taalelement aan de minitaal toevoegen, en bijbehorende typeringsregels geven.

De onderscheiden vereniging lijkt op de union van Algol 68 en op de variant record van Pascal, maar verschilt er wel van. Een onderscheiden vereniging komt overeen met de vereniging uit de verzamelingenleer, maar aan ieder lid uit een onderscheiden vereniging is te zien tot welk der summmanden hij eigenlijk behoort. Als expressievormen kiezen wij:

$$\underline{in}_1(e), \ldots, \underline{in}_n(e), (\underline{case}\ e\ \underline{in}\ f_1,\ldots,f_n)$$

Het resultaat van $\underline{in}_i(v)$ is: v als lid van de i-de summand in een onderscheiden vereniging. Voor een e die geschreven kan worden als $\underline{in}_i(v)$ is $(\underline{case}\ e\ \underline{in}\ f_1,\ldots,f_n)$ gelijk aan $f_i(v)$. De case-constructie test dus eerst tot welke summand de waarde van e behoort, en past dan de bijbehorende funktie toe.

Een voorbeeld van het gebruik is de linear search. Zij a, n, en x globaal gegeven, gevraagd wordt een funktie f(m) die nagaat of er een i is zo dat $m \leq i \leq n$ en a(i) = x. Als resultaat kiezen we zo'n i, indien die bestaat, en false anders. Het programma ziet er dan als volgt uit.

```
def f = (m): if m=n+1   then in2(false) else
             if a(m)=x  then in1(m) else f(m+1)
in
    .....
    (case f(1)
       in (i): print ("index gevonden, namelijk:", i)
        , (b): print ("zo'n index bestaat niet")
    )
    .....
ni
```

We zullen verderop, in deel 3.2 en hoofdstuk 5, zien dat de onderscheiden vereniging ook heel nuttig is bij recursieve datastructureringen.

We geven nu de typen en de typeringseisen die bij de onderscheiden vereniging genomen kunnen worden. Als typesamenstellingswijze definieren we

t ::= ...| (t1 +...+tn)

De tabel met eisen breiden we als volgt uit.

nr	e^t	eisen
	<u>in</u>$_i$ ($e0^{t0}$)	t = (t1 +...+t0+...+tn) voor een of andere n, t1,...,t_{i-1},t_{i+1},...,tn
	(<u>case</u> $e0^{t0}$ <u>in</u> $e1^{t1}$,...en^{tn})	er zijn t1',...,tn' zodat voor alle i ti = (ti' → t) en t0 = (t1' +...+tn')

Het is nu niet moeilijk om na te gaan dat het voorbeeldprogramma met de linear search goedtypeerbaar is: de identifier f krijgt het type (<u>int</u> → (<u>int</u> + <u>bool</u>)) toegewezen, en de subexpressies <u>in</u>$_1$(m), <u>in</u>$_2$(<u>false</u>) en f(m+1) krijgen ieder het type (<u>int</u> + <u>bool</u>).

3.2 Recursieve typen

De typering die we net hebben getoond garandeert weliswaar enige semantische eigenschappen, maar maakt ook verscheidene semantisch zinvolle expressies illegaal. Potentieel oneindige datastructuren lijken bijvoorbeeld onmogelijk, en ook sommig ander gebruik van waarden wordt bemoeilijkt. Voor een deel kan aan deze bezwaren tegemoet gekomen worden door oneindige typen te beschouwen. Bijvoorbeeld het oneindige type

void + <int, void + <int, void + <int,... >>>

Hierbij hebben we verondersteld dat t + t′ een type is, wanneer t en t′ dat ook zijn; t + t′ is dan de zogenaamde "onderscheiden vereniging", te vergelijken met de union van Algol 68 en de variant-record van Pascal.

In een programmatekst kunnen natuurlijk geen oneindige typen voluit uitgeschreven worden. Recursie is dan het middel om eindige notaties voor oneindige typen te hebben. Bovengenoemd type zou bijvoorbeeld genoteerd kunnen worden als

(rec z. void + <int, z>)

of als intseq where intseq = void + <int, intseq>.

De introductie van oneindige typen, en dus ook van recursieve notaties van typen, geeft wel enige complicaties. Er moet voor gezorgd worden dat het beslisbaar is of twee notaties eenzelfde type aanduiden. Bovengenoemd oneindige type kan bijvoorbeeld op de volgende twee manieren worden genoteerd:

intseq = void + <int, intseq>

intseq′= void + <int, void + <int, intseq′>>

De introductie van oneindige typen noodzaakt des te meer tot een diepe bezinning op het nut ervan. Niet alle voorbeelden van "beoogde semantische eigenschappen" worden door deze typering waargemaakt. Bijvoorbeeld de berekenbaarheid van de grootte van de representatie der aangeduide waarden gaat verloren.

Volledigheidshalve vermelden we nog hoe bij bovenstaand type intseq de funkties kop, staart, isleeg, opKopVan en de waarde legeRij geprogrammeerd worden. Bij iedere identifier-introductie zullen we zijn type vermelden.

```
kop : intseq  → int
    = (s: intseq): case s in (x:void): error,(g:<int,intseq>): g.1
staart: intseq  → intseq
      = (s: intseq): case s in (x:void): error, (g:<int, intseq>): g.2
isLeeg: intseq → bool
      = (s : intseq): case s in (x:void): true, (g:<int, intseq>): false
opKopVan: (int, intseq → intseq)
        = (i:int, s: intseq): in_2 (<i, s>)
legeRij: intseq
       = in_1 (empty)
```

Hierbij hebben we verondersteld dat empty een expressie van type void is, en dat error willekeurig type heeft.

3.3 Subtypen

Soms is er op de verzameling typen een subtype-relatie $\leq$ gedefinieerd. Het is mij nog niet geheel duidelijk wat het nut van zo'n subtype-relatie is. Enerzijds lijkt die een geschikt middel te zijn om de plaatsing van coercies (impliciet geprogrammeerde waardetransformaties) te definieren. Anderzijds lijkt die louter een verfijning van de typering, en dus van de beoogde semantische eigenschappen, mogelijk te maken. De karakteristieke wetten voor de subtype-relatie die wij hieronder geven lijken op beide gevallen van toepassing te zijn.

Bekende subtype-relaties zijn

$\underline{int} \leq \underline{real}$,

$1..10 \leq \underline{int}$,

$(\underline{int} \rightarrow \underline{int}) \leq (\underline{int} \rightarrow \underline{real})$.

Het effect van $t \leq t'$ op de typeringsregels is dat daar waar een t'-expressie hoort te staan, ook een t-expressie geschreven mag worden. Bijvoorbeeld, 3 kan op de plaats van een real-expressie staan, en succ (van type $(\underline{int} \rightarrow \underline{int})$) mag geschreven worden waar een $(\underline{int} \rightarrow \underline{real})$ verwacht wordt. De typeringstabel van deel 3.1 wordt dus uitgebreid met de regel:

nr	e^t	eisen
17	$e^{t'}$	$t' \leq t$

Semantisch is het effect van deze regel als volgt:
De expressie $e^{t'}$ --als expressie van type t-- is per definitie gelijkwaardig met $c_{t',t}(e^{t'})$, waarbij

$c_{t',t}$ van type $(t' \rightarrow t)$

de coercie van t' naar t is. Let wel, voor sommige subtype-relaties is iedere $c_{t',t}$ de identiteitsfunktie. Bijvoorbeeld, het valt te verwachten dat bij $1..10 \leq \underline{int}$ de coercie $c_{1..10,\underline{int}}$ de waarde onveranderd laat.

De heel algemeen geformuleerde regel nr 17 van de typeringstabel heeft tot gevolg dat er syntactische ambiguiteiten optreden. Immers, als real-expressie kan 3 + 5 op twee verschillende manieren ontleed worden, en bijgevolg zou de semantiek ook ambigu kunnen zijn: enerzijds is 3 + 5 gelijkwaardig met $c(3) +_r c(5)$ en anderzijds met $c(3 +_i 5)$, waarbij $c = c_{\underline{int}, \underline{real}}$ en $+_i : (\underline{int}, \underline{int} \rightarrow \underline{int})$, $+_r : (\underline{real}, \underline{real} \rightarrow \underline{real})$. Dergelijke syntactische ambiguiteiten zijn niet erg, en zelfs heel prettig omdat ze overspecificatie

vermijden, mits ze maar niet tot semantische ambiguiteiten leiden. Deze aanpak van coercies wordt uitgewerkt in [Reynolds 1981].

Stel nu eens dat $\underline{int} \leq \underline{real} \leq \underline{compl}$ en dat er dus conversies $c_{\underline{int},\underline{real}}$ en $c_{\underline{real},\underline{compl}}$ zijn. Dan is er ook een conversie $c=c_{(\underline{real} \to \underline{real}),(\underline{int} \to \underline{compl})}$, namelijk

$$c = (f: (\underline{real} \to \underline{real})): (x:\text{int}): c_{\underline{real},\underline{compl}}(f\ (c_{\underline{int},\underline{real}}(x)))$$

Dus een funktie sin:$(\underline{real} \to \underline{real})$ kan ook (dank zij deze coercie) als een funktie $:(\underline{int} \to \underline{compl})$ gebruikt worden. In het algemeen geformuleerd krijgen we als wet:

$(t_1',\ldots,t_n' \to t') \leq (t_1,\ldots,t_n \to t)$ indien
$t_1 \leq t_1'$ en ... en $t_n \leq t_n'$ en $t' \leq t$.

In woorden: de sybtype-relatie is monotoon in het resultaat en antimonotoon in de argumenten.

Op dezelfde manier vinden we ook voor de andere type-samenstellingswijzen wetmatigeheden in de sub-type relatie. Bijvoorbeeld voor de record-samenstellingen:

$\langle t_1,\ldots,t_n\rangle \leq \langle t_n',\ldots,t_n'\rangle$ indien
$t_1 \leq t_1'$ en ... en $t_n \leq t_n'$

Maar hier is ook een verfijnig van mogelijk. Stel namelijk dat het record-type de vorm

$\langle x_1:t_1,\ldots, x_n:t_n\rangle$

heeft, waarbij $x_1,\ldots, x_n$ verschillende identifiers zijn, de zogenaamde veld-identifiers. Deze zouden net als in Pascal en Algol 68 in plaats van de weinig suggestieve selectoren 1, 2, ... gebruikt kunnen worden. Dan kunnen we ook definiëren

$\langle x_1:t_1,\ldots,x_n:t_n\rangle \leq \langle x_{i1}:t_{i1},\ldots,x_{ik}:t_{ik}\rangle$
indien
$\{i1,\ldots,ik\} \subseteq \{1,\ldots,n\}$ en $t_{i1} \leq t_{i1}'$ en ... en $t_{ik} \leq t_{ik}'$.

De bijbehorende coercie laat komponenten vervallen en past vervolgens komponentsgewijze de betreffende coercies toe. Zo is het mogelijk om daar waar een

<age: int, salary: real >

gevraagd wordt, een

<age: 0..100, sex: bool, salary: int, year of birth: int>

te schrijven; de selectie wordt automatisch tussengevoegd.

Voor de type-samenstellingswijze ref t ligt hetvolgende voor de hand:

ref t $\leq$ t

Dus daar waar een int wordt verwacht, mag ook een ref int staan; de coercie $C_{\text{ref } t,\ t}$ = deref wordt automatisch ingevoegd.

Beschouw nu eens een toekenning ... := 3. Als bestemmingsvariabele wordt een ref int verwacht, maar dan moet een ref real natuurlijk ook mogen. Dus terwijl int $\leq$ real, zouden we willen ref real $\leq$ ref int. Dit geeft echter ongewenste interactie met de al gegeven wet ref t $\leq$ t. Want dan zou gelden

ref real $\leq$ ref int $\leq$ int,

zodat daar waar een int hoort te staan, ook een ref real geschreven mag worden; dit is ongewenst. Toch kunnen we onze wens met behulp van de subtype-relatie modelleren. Dat gaat als volgt:

(i) breidt de typen uit met t ::= ... | acc t

(ii) definieer

1. ref t $\leq$ acc t
2. acc t′ $\leq$ acc t indien t $\leq$ t′

(let op de antimonotonie; dit is onze wens!)

(iii) herdefinieer de typeringseis voor de toekenning:

nr	e^t	eisen
9′	$e1^{t1} := e2^{t2}$	t1 = acc t2 en ...

Acceptoren --aanduidbaar door expressies met type acc t-- zijn "write-only variabelen", dus bv. zuivere uitvoer-variabelen (denk aan het gebruik van een funktie-identifier als variabele, in Pascal). References, ofwel variabelen, kunnen op twee manieren gebruikt worden:

- volgens **ref** t $\leq$ **acc** t als acceptor voor "t-waarden",
- volgens **ref** t $\leq$ t als "t-waarden".

Bovenstaand idee van acceptoren is ontleend aan [Reynolds 1981], en komt ook naar voren in [Swierstra 1981].

4. Tweede orde typeringen

We hebben laten zien dat dat middels typering een klasse van expressies bepaald wordt waarvoor een beoogde semantische eigenschap geldt. Als er per definitie geen goed-getypeerde expressies zijn, is trivialiter iedere eigenschap gegarandeerd voor iedere expressie uit die klasse. De kunst is echter om zoveel mogelijk expressies door te laten. De vraag rijst derhalve of de tot nu toe behandelde typeringen niet te beperkend zijn. En het antwoord luidt zonder meer: ja! We geven hiervoor twee argumenten.

Ten eerste is het onmogelijk om met conventionele typeringen, zoals gekarakteriseerd in de tabel van deel 3.1, hetvolgende programmafragment te typeren.

```
def double = (f,x): f(f(x))
in ... double(succ, 3) ... double(sqrt, 3.14)...
```

Natuurlijk is er wel een programma mogelijk dat conventioneel getypeerd kan worden, en hiermee duidelijk gelijkwaardig is, nl.

```
def double1 = (f,x): f(f(x))
  ; double2 = (f,x): f(f(x))
in ...double1(succ, 3) ... double2(sqrt, 3.14)...
```

Dus alleen voor de typering moeten semantisch zinvolle programma's op een nogal domme wijze herschreven worden. Dit staat de vorming van program-o-theken met algemeen toepasbare programmatuur (sorteringsprocedures, linear search etc) in de weg.

Ten tweede merken we op dat het nut van typering, d.w.z. de beoogde semantische eigenschappen, soms geformuleerd zijn aan de hand van de typen. Bijvoorbeeld: domeintesten zijn gedurende run-time overbodig; m.n. int-operaties zoals de + zullen nooit toegepast kunnen worden op waarden opgeleverd door bool-expressies. Maar aan zo'n eigenschap heb je niet veel, als er alleen de elementaire typen int en bool zijn, en de programmeur niet in staat is z'n eigen elementaire typen te definieren. Immers, erg veel abstracte begrippen zullen op de een of andere manier met getallen en waarheidswaarden gerepresenteerd moeten worden. Denk bijvoorbeeld aan huisnummers, of kleuren, of geboorte-jaren. En je zou graag hebben dat run-time domeintesten t.a.v. huisnummers, kleuren, en jaarnummers overbodig zijn. Dus moeten daarvoor ook nieuwe typen gedefinieerd kunnen worden; zonder nieuwe typen kun je alleen garanderen dat getallen en waarheidswaarden niet verkeerd gebruikt worden.

We zullen in dit hoofdstuk typeringen behandelen die aan deze bezwaren tegemoet komen. De typeringen heten van "tweede orde", omdat er nu identifiers gedefinieerd kunnen worden (via definities of parameterisaties), die in andere definities in het type optreden. De benaming "polymorfe typering" wordt ook wel gebruikt; "polymorf" = "veel-vormig".

In deel 4.1 behandelen we de eenvoudigste vorm; in deel 4.2 een ingewikkelder variant. Het zal blijken dat in de ingewikkelder variant zg. "abstract data types" programmeerbaar zijn. In deel 4.3 schetsen we kort hoe in --andere varianten van-- tweede orde typeringen programma-specificaties mogelijk zijn. Door deze typeringen kan dus programma-korrektheid worden afgdwongen, zie "beoogde semantische eigenschap nr 6" in deel 2.3.

Notatie. Naast int, bool, ... komen er nu ook andere type-identifiers. Deze zullen we soms met gewone, en soms met onderstreepte identifiers noteren. De onderstrepingen dienen louter ter verhoging van de leesbaarheid; zij hebben verder geen betekenis.

4.1 De eenvoudigste vorm

In de eenvoudigste vorm van tweede orde typering, zie [Reynolds 1974], kunnen typen wel als parameter worden doorgegeven, en worden gedefinieerd, maar overigens niet als expressie optreden. Dus funkties kunnen geen type opleveren, en typen kunnen niet een component zijn van een record, enzovoort.
We leggen de typering uit aan de hand van een voorbeeld.

Beschouw hetvolgende programmafragment.

```
def double = (z: tp, f: (z → z), x: z):  f(f(x))
in  ... double (int, succ, 3) ... double(real, sqrt, 3.14)...
```

Het is nu nodig om de typen van de formele parameters expliciet te vermelden. Anders is niet duidelijk dat het type van f afhangt van de eerste parameter; type tp wordt gebruikt als "het type van typen".

De romp van double is een conventionele expressie; hij moet ook aan de conventionele typeringseisen voldoen. Daarbij wordt z beschouwd als een nieuw "primitief" type, ongerelateerd aan andere type-identifiers of -constanten. Aldus is gegarandeerd dat de type-korrektheid van de romp behouden blijft wanneer z door willekeurig type wordt vervangen! Gegeven dat f: (z → z) en x: z, is het duidelijk dat de romp type-korrekt is en z als type heeft. Het type van double luidt in deze tweede orde typering:

$(z \in \underline{tp}, (z \to z), z \;\to\; z)$.

Hierbij is z een lokale naam voor het eerste argument, die in het tweede en derde argument en in het resultaat gebruikt wordt. Een andere rol dan lokale benaming vervult z niet. Dus andere notaties voor hetzelfde type zijn bijvoorbeeld

$(x \in \underline{tp}, (x \to x), x \;\to\; x)$, en

$(y \in \underline{tp}, (y \to y), y \;\to\; y)$

etcetera.

De aanroepen van double zijn geen conventionele expressies; er staat nu een type-argument en dat kon in alle vorige taalvarianten niet. De typeringseisen m.b.t. dergelijke argumenten zijn als volgt. Beschouw de aanroep double(int, succ, 3). Volgens het type van double moet het eerste argument een type zijn; dat klopt. Nu wordt dit type-argument, int dus, letterlijk gesubstitueerd in de overige parametertypen en resultaat-type van double.

Dat levert

```
...(int → int), int → int)
```

De rest van de aanroep --weer een conventionele expressie-- wordt nu ten aanzien van dit nieuw verkregen type getoetst. Inderdaad is succ: (int → int), en is 3: int, zodat de aanroep als geheel type int heeft. We zullen kortheidshalve de typeringstabel van deel 2.3 niet formeel geven; hopelijk is bovenstaande uiteenzetting duidelijk genoeg.

Ten aanzien van type-definities merken we het volgende op.

We hebben al gesteld in Hoofdstuk 2 dat de oproep f(e1,...,en), waarbij f=(x1,...,xn):e, semantisch gelijkwaardig is met het blok

```
def x1 = e1,...,xn = en in e ni.
```

Wanneer we die gelijkwaardigheid ook per definitie stellen voor wat de typering betreft, dan liggen nu ook de typeringsregels voor type-definities geheel vast. Bijvoorbeeld,

```
(def count: tp = int
  , succ: (count → count) = (x:int):x+1
  , zero: count = 0
  in (----succ (succ(zero))----)...count...ni)...int...
```

is goed-getypeerd vanwege de gelijkwaardigheid met

```
f (int, (x:int): x+1, 0)...int...
waarbij
f: (z ∈ tp, (z → z), z →  ...z...)
 = (count: tp, succ: (count → count), zero: count):
   (----succ (succ (zero))----)...count...
```

Dus binnen in en ni, en binnen de linkerleden der definities is count een type-identifier die qua typering staat voor een geheel nieuwe type, ongerelateerd aan int !

Los hiervan kunnen we ook de definitievorm

```
def z : tp == t in e ni
```

toevoegen aan de taal, met als typeringsregel dat binnen e de type-identifier z louter een afkorting is van t. Vooral wanneer de type-expressie t groot is, is dit een handige notatie. Maar bedenk wel dat deze typedefinitie geen representatiekeuze (te weten: het type t waarvoor z staat) afschermt. (Zie structurele equivalentie van typen, in Aangangsel B.)

Aangezien typen nu als argument (en parameter) kunnen optreden, lijkt het erop dat ze noodzakelijk "run-time" een rol spelen. Dat zou in strijd zijn met onze definitie van typering, waar we stellen dat ze louter dienen om de "goed-getypeerde" van de "niet goed-getypeerde" expressies te onderscheiden, en semantisch in het geheel geen invloed hebben. Gelukkig zijn ze inderdaad semantisch overbodig: in iedere goed-getypeerde expressie waarvan het type als geheel geen tp bevat, en ook de gebruikte globale identifiers een type zonder tp hebben, kunnen alle type-argumenten geelimineerd worden zodat er een conventioneel getypeerde expressie overblijft, die gelijkwaardig is met de oorspronkelijke. Deze eliminatie is algoritmisch; d.w.z. kan door de compiler --voorafgaande aan run-time-- gedaan worden. (Hierbij is verondersteld dat er geen recursie op typen is toegestaan).

We zouden nu de beoogde semantische eigenschappen kunnen formuleren, en bewijzen dat zij door deze typering worden afgedwongen. Het voert ons te ver om dat hier te doen. Bovendien is dat nog steeds -anno 1982-- onderwerp van onderzoek.

Daarnaast kun je de winst van deze typering ten opzichte van de conventionele typering onderzoeken. Een formulering daarvan wordt gegeven in [Fortune et al 1980]. Het komt erop neer dat --onder bepaalde voorwaarden-- een veel grotere klasse van funkties uitdrukbaar is in de tweede orde getypeerde taal, dan in de eerste orde getypeerde taal.

4.2 Een ingewikkelder variant: de SVPtypering

We geven een veralgemening van de zojuist gegeven eenvoudigste vorm van tweede orde typering. Nu kunnen typen ook als resultaat van een funktie, en als komponent van een record optreden. Er is in feite geen onderscheid meer tussen typen en expressies; dat wil zeggen, typen zijn "gewone expressies van type tp". (In het bijzonder is tp zelf een type, dus een expressie van type tp, ofwel tp: tp).

Het is overigens de vraag of deze typering nog wel terecht "van tweede orde" genoemd kan worden. Het lijkt erop dat zij beter "orde-loos" genoemd kan worden. Overigens is deze typering, die ik in het vervolg met "SVPtypering" zal aandui-den, naar mijn weten nog niet in de literatuur bestudeerd; de eerste formulering is [Fokkinga 1981 b].

De SVPtypering is een uitbreiding van de tweede orde typering, zodat de definitie

def double = (z: tp, f: (z → z), x: z): f(f(x))

nog steeds mogelijk is. Dat typen nu ook als komponenten van een record kunnen optreden, laten we weer aan de hand van een voorbeeld zien. Daartoe gaan we "arrays" programmeren. Die zaten immers nog niet in de taal.

Bij een gegeven type elt en een gegeven int-waarde bestaat
"het abstracte datatype (afgekort: ADT) der arrays over type elt en ter lengte van de gegeven int-waarde"
uit de volgende dingen:

- een type, noem het even repr, dat het type is van de array-waarden en dus de representatie van array-waarden geeft;
- een funktie van type (repr, int, elt → repr), die de selectieve verandering ("bijstelling", "updating") van een array verzorgt;
- een funktie van type (repr, int → elt), die de subscriptie op een array is;
- een funktie van type ((int → elt) → repr), die bij gegeven f: (int → elt) een array-waarde maakt met de waarden f(1), f(2),..., f(lth) (waarbij lth de gegeven int-waarde is);

enzovoorts. Dus een "array-ADT" bestaat uit een record met bovengenoemde kompo-nenten. De funktie die zo'n array-ADT oplevert wordt in de SVPtypering als volgt getypeerd en gedefinieerd.

```
def arrayADT                                  {maak array-ADT}
   : (elt ∈ tp, int  →
            < repr ∈ tp                       {representatie van arraywaarden}
            , (repr, int, elt → repr)         {bijstelling}
            , (repr, int    → elt)            {subscriptie}
            , ((int → elt) →  repr)           {maak arraywaarde}
            .
            .
            .
            >)
   = (elt: tp, lth: int):
            --------?? romp van arrayADT ??--------
```

Uit het type van arrayADT is te zien dat arrayADT een funktie is.
In dat type is elt louter een lokale naamgeving voor de eerste parameter; systematische vervanging van elt door een andere, nieuwe, identifier laat de betekenis onveranderd. Het resultaat-type is een record. Daarin is repr louter een lokale naamgeving voor de eerste komponent; systematische hernoeming van repr door een andere, nieuwe, identifier is dus geoorloofd.
In de definierende funktietekst voor arrayADT hadden we de eerste parameter ook anders dan "elt" kunnen noemen, bijvoorbeeld xxx. De romp van arrayADT had dan van type

< repr ∈ tp, (repr, int, xxx → repr), ... >

moeten zijn; d.w.z. het resultaattype van het funktietype van arrayADT, waarin elt door xxx is vervangen. Er zijn nog verscheidene keuzen mogelijk om de romp van arrayADT uit te detailleren. In aanhangsel A zullen we twee uitwerkingen geven.

Beschouw nu de aanroep

arrayADT(real, 10)

Deze duidt een implementatie aan voor array-ADT's met elementtype real en ter lengte 10. Het type van deze aanroep is

< repr ∈ tp, (repr, int, real → repr), ... >

ofwel: < z ∈ tp, (z, int, real → z), ... >

Let erop dat de argumentexpressies in het resultaattype van arrayADT gesubstitueerd zijn. De eerste parameter is in het type van arrayADT elt genoemd; daarvoor in de plaats staat nu de argumentexpressie real. De tweede parameter van

arrayADT is niet benoemd, en wordt dus niet gebruikt in het resultaattype; daarom komt toevallig de argumentexpressie 10 niet te voorschijn in het type van de aanroep. (Als we zoals in Pascal subrangetypen hadden gehad, hadden we kunnen definieren:

```
def arrayADT'
    : ( elt ∊ tp, lth ∊ int
           < repr   tp
           , (repr, ⌊1..lth⌋, elt → repr)
           .
           .
```

En dan zou de expressie 10 wel in het type van arrayADT (real, 10) te voorschijn komen.)

Beschouw nu de eerste komponent van die implementatie van arrays:
arrayADT(real, 10).1
Deze is van type tp, en geeft de representatie van array-waarden weer. Aan de hand van het type van arrayADT, of van arrayADT(real, 10), kan er niets meer worden afgeleid over deze eerste komponent. Dus de manier waarop array-waarden worden gerepresenteerd is (buiten de romp van arrayADT) onbekend!

De tweede komponent,
arrayADT(real, 10).2
is de bijstelling. Deze is van type
"(repr, int, real → repr)
waarbij repr staat voor de eerste komponent"
Precies geformuleerd luidt het type:
(arrayADT(real, 10).1, int, real → arrayADT(real, 10).1)
M.a.w. we hebben in de 2de komponent van het type van arrayADT(real,10) de lokale naam voor de eerste komponent vervangen door de globale aanduiding van de eerste komponent.

Array-waarden worden gewijzigd door de zojuist behandelde bijstelling; ze worden uit bekende soorten waarden geschapen door arrayADT(real, 10).4, dat is de "maak array" funktie. Deze heeft als type
((int → real) → arrayADT(real, 10).1)

Aan de hand van voorbeelden zijn nu de typeringsregels behandeld voor aanroepen en selecties. Die regels zullen we zodadelijk nog samenvatten. Maar allereerst nog een voorbeeld voor een record. Beschouw daartoe de expressie

$\langle$ int, 3, 3 $\rangle$

Het type hiervan is niet uniek bepaald. Ieder van de volgende vier typen is een type voor die expressie:

$\langle$ z $\in$ tp, z, z $\rangle$,

$\langle$ z $\in$ tp, int, z $\rangle$,

$\langle$ z $\in$ tp, z, int $\rangle$,

$\langle$ z $\in$ tp, int, int $\rangle$ ofwel $\langle$ tp, int, int $\rangle$.

Tengevolge hiervan kunnen we in het algemeen niet meer spreken over "het type van een expressie", maar slechts over "een type van een expressie".

Samengevat komt de typering hierop neer:

- door in een resultaattype van de funktie de parameternamen te vervangen door de argumentexpressies, verkrijgt men een type voor de aanroep;
- door in de i-de komponent van een recordtype van de geselecteerde expressie e de komponentnamen te vervangen door desbetreffende selecties op e, verkrijgt men een type voor de selectie e.i;
- een recordexpressie $\langle e_1,\ldots,e_n \rangle$ is van type $\langle x_1 \in t_1,\ldots,x_n \in t_n \rangle$ indien voor i=1,...,n de komponent e_i van type "t_i met substitutie van e_1 voor x_1, van e_2 voor x_2,... en van e_{i-1} voor x_{i-1}" is;
- voor het overige gelden er dezelfde regels als bij de conventionele typering. Met name moet ieder argument van een funktieaanroep een type hebben dat --behoudens lokale naamgeving-- gelijk is aan het betreffende parametertype, en ieder toegepast voorkomen van een identifier krijgt het type dat bij de introductie (= definitie, parameter) aan die identifier wordt toegewezen.

Opdat de typering algoritmisch is, moet de gelijkheid van typen, die bij aanroepen wordt geeist, beslisbaar zijn. Dat is zeker zo als die gelijkheid opgevat wordt als "gelijkheid van expressies". Als "gelijkheid van de waarden aangeduid door expressies" beslisbaar zou zijn, dan hadden we die gelijkheid wel kunnen eisen. In dat geval zouden bijvoorbeeld

arrayADT (real, 10).1

arrayADT (real, 9+1).1

arrayADT (real, if true then 10 else 9).1

gelijke typen zijn. In onze SVPtypering zijn het verschillende expressies, en

dus ook verschillende typen.
Er zijn typeringen waar dit anders ligt; met name de Automath typeringen [de Bruijn 1980] en Intuitionistische Typetheorie [Martin-Lof 1975].

Dus, in de SVPtypering, zijn verschillende expressies van type tp ook verschillende typen. Derhalve zijn na def real-10-array: tp = arrayADT (real, 10).1
de typen

real-10-array , en
arrayADT (real, 10).1

verschillend. Wat --voor praktisch gebruik-- wel handig is, is een afkortingsmechanisme zo dat qua typering de gedefinieerde identifiers en de definierende expressie onderling gewisseld mogen worden. Zo'n definitie zullen we schrijven met het == -teken i.p.v. het = -teken. Dan zijn na

def real-10-array: tp == arrayADT (real, 10).1
, ten: int == 10
, real-ten-array : tp == arrayADT (real, ten).1

de volgende typen gelijk:

arrayADT (real, 10).1
arrayADT (real, ten).1
real-10-array
real-ten-array.

Het zojuist behandelde voorbeeld van het abstracte datatype der arrays --zie ook Aanhangsel A-- laat zien dat de typering krachtig genoeg is om soms dat te definieren wat in conventionele talen al vooraf in de taaldefinitie ingebouwd moet worden. In feite zijn ook de ingebouwde datatypen van gehele getallen en waarheidswaarden niet nodig. Het is mogelijk om zonder gebruik te maken van de expressievormen e + e, e * e, e and e, if e then e else e, etcetera, en zonder de expressievormen 0, 1, 2, ... abstracte data typen

integerADT en
boolADT

te definieren, die de optelling, vermenigvuldiging, conjunctie, test met vertakking, etcetera, en een paar getallen als komponent hebben. De eigenlijke taaldefinitie schrompelt daarmee ineen. Het nadeel daarvan is alleen dat de schrijfwijzen onaanvaardbaar ongemakkelijk worden.
Dit zou verholpen kunnen worden door een goed notatiesysteem te bedenken voor expressies. Bijvoorbeeld een systeem volgens welke ook infix, postfix of distfix (een gemengde vorm) funkties gedefinieerd kunnen worden.

We geven tot slot nog twee interessante voorbeelden.
Allereerst laten we zien hoe gebruik gemaakt kan worden van het feit dat tp van type tp is:

```
def nullADT: < repr ∈ tp, repr > = < tp, tp >
```

Hier wordt het abstracte data type van de triviale verzameling gedefinieerd. Vergelijk dit met Algol 68 waar empty de enige waarde van de mode void is. In feite kan er niet veel met nullADT.2 gedaan worden. Het dient hoogstens als "iets wat er verder niet toe doet".

Nu het tweede voorbeeld. Beschouw allereerst nogmaals de typen

```
(elt ∈ tp, int → --- elt --- ) en
<repr ∈ tp, --- repr --- >.
```

In beide wordt lokaal aan een type een naam gegeven, en verderop gebruikt. Het is soms zinvol en nodig om parameters of komponenten te benoemen die géén type zijn. Bijvoorbeeld, een funktie die alle array-waarden sommeert kan gedefinieerd worden door

```
def f : (x∈ int, a: arrayADT(real, x).1 → real)
      = (x: int, a: arrayADT(real, x).1):
          "de som van sub(a,1) t/m sub(a,x)
          waarbij sub == arrayADT(real, x).3"
```

In het type van f wordt de eerste parameter --van type int-- benoemd en verderop ook gebruikt.

4.3 Typering als programmaspecificatie

Er zijn andere varianten van de tweede orde typering waarmee programmaspecificaties als type geformuleerd kunnen worden. Dit is o.a. het geval in Intuitionistische Typetheorie [Martin-Lof 1975], [Nordstrom 1981], in de talen PL/CV2, PL/CV3 [Constable 1983], en in de Automath-talen [de Bruijn 1980]. Ik geef hier een oppervlakkige uiteenzetting en enig commentaar. Tot een precieze behandeling voel ik me (anno 1982) niet in staat; de details zijn voor mij ook nog duister.

Stel dat elementaire beweringen, zoals gelijkheid van natuurlijke getallen, al in de typen kunnen worden uitgedrukt. Het is dan mogelijk om ook samengestelde beweringen zoals

$\underline{A}x\in P.\ P'$ en $\underline{E}x\in P.\ P'$

als tweede orde typen te formuleren, namelijk als

$(x\in t \rightarrow t')$ resp. $\langle x\in t,\ t'\rangle$

waarbij t en t' al de formuleringen-als-type zijn van de beweringen P en P'. Deze formulering wordt gerechtvaardigd door het feit dat de constructieve interpretatie van $\underline{A}x\in P.\ P'$ luidt:

er is een algoritme dat iedere constructie van een x die aan P voldoet, omzet in een constructie van P'.

Dergelijke algoritmen zijn dus precies de programma's van type $(x\in t \rightarrow t')$. Net zo luidt de constructieve interpretatie van $\underline{E}x\in P.\ P'$

er is een tweetal bestaande uit een x die aan P voldoet en een constructie --in termen van x-- van P'.

Het zijn dus precies de expressies van type $\langle x\in t,\ t'\rangle$ die hiermee overeenkomen.

In feite komt het erop neer dat constructieve bewijzen, in tegenstelling tot de bewijzen in de klassieke logica, algoritmen zijn en dus in beginsel mechanisch uitvoerbaar. Algoritmen zijn bewijzen, en de typen zijn de stellingen; korrektheid ten aanzien van specificaties is goed-getypeerdheid van expressies. Maar pas op, er kleven mijns inziens mogelijk twee nadelen aan deze aanpak van programmatuurkorrektheid.

Ten eerste moet het hele korrektheidsbewijs in feite in het programma gecodeerd worden, terwijl een groot gedeelte daarvan voor het uiteindelijke doel niet van belang is. Bijvoorbeeld, de bewering "een naar grootte geordend paar getallen" zou als volgt kunnen luiden:

$\underline{E}\, x \in \underline{int},\, y \in \underline{int}.\ \underline{E}\, m \in \underline{nat}.\ x + m = y$

ofwel $\langle x \in \underline{int},\, y \in \underline{int},\ \langle m \in \underline{nat},\ x + m = y \rangle\rangle$.

Welnu, behalve het opleveren van de twee getallen, moet er dus ook het bewijs van de geordendheid geprogrammeerd worden, en dat is voor de berekening zelf onnodig. Natuurlijk kun je daar waar je dat wenst, de specificatie enorm afzwakken. De bewering "een paar getallen" luidt

$\underline{E}\, x \in \underline{int},\, y \in \underline{int}.\ \underline{true}$

ofwel $\langle x \in \underline{int},\, y \in \underline{int},\, \underline{true} \rangle$

Het bewijs van true is triviaal, de expressie bestaande uit een punt, of zoiets, is van type true. Maar het blijft vooralsnog de vraag of dit in alle gevallen de bewijslast tot nul reduceert.

Ten tweede is er dit nadeel. Potentieel niet-terminerende progmramma's mogen er niet zijn, omdat potentieel niet-eindigende constructies geen bewijs zijn. Dus de algemene vorm van recursie is niet geoorloofd, een veel beperkter vorm van recursie wel. Nu wordt vaak een programma geschreven waarvan men op grond van bijvoorbeeld de te verwachten invoer de terminatie inziet. Dergelijke programma's worden onmogelijk. Dit lijkt een onoverkomelijk nadeel te zijn.

Het is voorts opmerkelijk dat in dergelijke talen de typering $\underline{tp}: \underline{tp}$ niet geldt. In plaats daarvan zijn er typen $\underline{tp}_0, \underline{tp}_1, \underline{tp}_2, \ldots$ met $\underline{tp}_i : \underline{tp}_{i+1}$ voor alle i. De noodzaak hiervan is mij nog niet precies duidelijk. Te meer daar er wel andere semantische interpretaties zijn van typen, waarbij $\underline{tp}: \underline{tp}$ niet tot een tegenspraak leidt. Dit is het geval als we typen interpreteren als "closures", zie [Scott 1976].

5. Type polymorfie

Een andere uitbreiding van de conventionele typering is de "type polymorfie", zoals beschreven door [Milner 1978]. We zullen de voor- en nadelen, en het verband met de eenvoudigste vorm van tweede orde typering uiteenzetten.

Het voordeel ten opzichte van de 2de orde typering, en tevens het meest in het oog springend, is het feit dat typen bijna nooit vermeld hoeven worden bij identifier-introducties (parameters en declaraties), hoewel het altijd mag, en dat het polymorfe karakter van funkties (d.w.z. de toepasbaarheid op argumenten van velerlei typen) niet expliciet vermeld en bestuurd hoeft te worden.

Laten we allereerst de typering aan de hand van een voorbeeld schetsen. Beschouw nogmaals

```
def double = (f, x) : f(f(x))
in  ---  double(succ,...) --- double(..., 3.14) ---
```

De conventionele typeringseisen blijven op een uitzondering na onveranderd gehandhaafd.
De uitzondering betreft het type van de toegepaste voorkomens van een door def gedefinieerde identifier, zoals double. Voor parameters, zoals f en x, geldt dat alle voorkomens hetzelfde type krijgen. Als we met a, b, c en d de nog onbekende typen aanduiden van f, x, f(x) en f(f(x)), dan eist de conventionele typering dat

$a = (b \to c)$ {uit $f^a(x^b)$: c}, en
$a = (c \to d)$ {uit $f^a(\ldots)^c$: d}.

Dit tweetal vergelijkingen in de onbekenden a, b, c en d (en over de typen
$t ::= int \mid bool \mid (t_1,\ldots,t_n) \mid \ldots$)
is gelijkwaardig met

$a = (b \to b)$,
$d = c = b$.

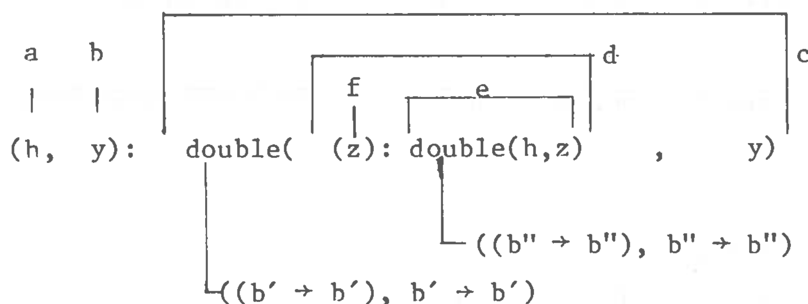
Dus zijn de typen van f, x en double aan elkaar gerelateerd als $(b \to b)$, b en $((b \to b), b \to b)$, waarbij b een nog vrij te kiezen type is.[†] In plaats van b dan eenduidig te bepalen, blijft hij als zogenaamde generische type-variabele in het type van double staan. Voor ieder toegepast voorkomen van double mag voor b een andere keuze gedaan worden; dit is het enige verschil met de conventionele typering. Bij de eerste aanroep van double dwingen de eisen nu af dat b = int, en bij de twee aantoep dat b = real.

[†] Het type van double is nu: "$((b \to b), b \to b)$ met b generisch instantieerbaar".

Iets ingewikkelder wordt het bij de definitie

def quadruple = (h,y): double((z): double(h,z), y)

De twee voorkomens van double krijgen achtereenvolgens als type $((b' \to b'), b' \to b')$ en $((b'' \to b''), b'' \to b'')$, waarbij b' en b'' nieuwe type-variabelen zijn. De conventionele eisen bepalen nu of deze expressie goed-getypeerd kan worden, en wat het type van het geheel is -- eventueel uitgedrukt in type-variabelen--. We werken dit nu uit. Stel dat de typen van de subexpressies als volgt benoemd worden:



De geeiste gelijkheden luiden dan

$((b' \to b'), b' \to b') = (d, b \to c),$

$((b'' \to b''), b'' \to b'') = (a, f \to e),$

$d = (f \to e).$

Dit stelsel is gelijkwaardig met

$b' = b = c = f = e = b'',$

$a = d = (b \to b).$

Dus het type van het geheel is van de vorm

$((b \to b), b \to b)$

waarbij b niet beperkt is, en dus generisch geinstantieerd mag worden bij toegepaste voorkomens van quadruple.

Opmerking. Er zit een addertje onder het gras bij de bepaling of een type-variabele generisch is, en dus geinstantieerd mag worden. Beschouw de funktie-expressie

(f): def g = f in --g(--)-- g(--) --- ni

Het is duidelijk dat als f type a krijgt, g dat ook krijgt. Maar bij de aanroepen van g mag er natuurlijk niet zo maar een vrije keuze voor a gedaan worden. Anders zou op deze manier iedere expressie tot een goedtypeerbare zijn om te zetten. Dus in het type van g is a niet generisch, omdat hij ook al in het type van een parameter voorkomt. (Het kan wel zijn dat de conventionele eisen

afdwingen dat a van zekere vorm is, bijvoorbeeld (a' → a"). Dat geldt dan natuurlijk ook voor het type van g.)

Opmerking. Het is beslisbaar of een gegeven stel typevergelijkingen oplosbaar is, en er is een algoritme dat "de minst beperkende" oplossing berekent in een tijdsduur en met geheugenruimte die lineair afhangen van het aantal symbolen in het stel vergelijkingen. Zie [Martelli, Montanari 1982] en Aanhangsel C.

In de praktijk blijkt dat heel veel zinvolle programma's middels de type polymorfie goed-typeerbaar zijn. Maar niet alle programma's. In feite komt de klasse van polymorf-typeerbare programma's overeen met een deel van de goed-typeerbare programma's volgens de eenvoudige 2de orde typering. Die overeenkomst is als volgt. Stel dat

def x = ...

getypeerd wordt met t waarin a en b generische typevariabelen zijn. Dan hadden we met de 2de orde typering kunnen schrijven:

def x': (a ∈ tp, b ∈ tp → t) = (a: tp, b: tp): ....

Ieder toegepast voorkomen van x moeten we dan schrijven als x'(t1,t2) voor die conventionele typen t1 en t2 (d.w.z. zonder tp erin) die in de polymorfe typering uiteindelijk ook voor a en b gekozen zouden zijn.

Uit bovenstaande blijkt tevens wat het verlies is van de type-polymorfie ten opzichte van 2de orde typering. Parameters zijn niet-polymorf; hun type wordt niet per toegepast voorkomen geinstantieerd. In de tweede orde typering mag een parameter best het type (a ∈ tp, (a → a), a → a) hebben! Maar zoals gezegd, in de praktijk blijkt dit nogal zelden voor te komen.

Naast de talen genoemd in [Milner 1978], is de polymorfe typering ook aanwezig in de taal B, zie [Meertens 1982] en [Geurts 1982].

***

Met een kleine elegante taaluitbreiding is het nu mogelijk om sommige datatypen zelf te definieren, en wel deze: datatypen zonder parameters of met typen --niet getallen etc-- als parameters. (De uitbreiding lijkt sterkt op die welke in Aanhangsel B behandeld is, maar is nu aangepast voor de hier behandelde typepolymorfie.) We nemen weer een voorbeeld om het een en ander uit te leggen: rijen van elementen. Herinner je dat in Deel 3.2 rijen van integers, intseq, waren gedefinieerd, die definitie wordt nu dus veralgemeend.

Beschouw de volgende programmatekst.

```
def newtp seq(a) = void + <a, seq(a)>
    with kop = (s): case reprseq(s)
                    in (x): error, (g): g.1
         staart = (s): case reprseq(s)
                       in (x): error, (g): g.2
         isLeeg = (s): case reprseq(s)
                       in (x): true, (g): false
         opKopVan = (i,s): absseq (in2(<i, s>))
         legeRij = absseq (in1 (empty))
    end
in ... ni
```

Achter def staan er zes definities, die allen tussen in en ni geldig zijn. Daarenboven bestaan er tussen with en end, en alleen daar, nog twee "standaard"funkties, namelijk

```
absseq: ((void + <a, seq(a)>) → seq(a))
reprseq: (seq(a) → (void + <a, seq(a)>))
```

Zij dienen er alleen voor om de typetoewijzing te sturen, en zijn semantisch louter de identiteit. Van deze funkties ligt het type, een polymorf type waarin a generisch instantieerbaar is, vast. De typen van kop, staart, enz. worden op de al eerder beschreven manier bepaald. Het blijkt dan dat zij de volgende typen hebben.

kop: (seq(a) → a)

staart: (seq(a) → seq(a))

isLeeg: (seq(a) → bool)

opKopVan: (a, seq(a) → seq(a))

legeRij: seq(a)

Hierbij is a steeds een generisch instantieerbare type variabele. Dus type-checking zal nu uitwijzen dat de volgende regels goed(O.K.) respectievelijk niet goed typebeerbaar (K.O. van Knock Out) zijn.

kop (opKopVan(17, legeRij)) O.K.

staart (opKopVan(true, legeRij)) O.K.

opKopVan (17, staart(opKopVan(true, legeRij))) K.O.

opKopVan (17, in_1(empty)) K.O.

Met andere woorden, buiten with ... end kan er ten gevolge van de typeringsregels geen gebruik gemaakt worden van de representatiekeuze:

seq(a) = void + <a, seq(a)>.

Binnen with ... end kan dat wel, door middel van de polymorfe funkties absseq en reprseq.

Aanhangsel A. Een implementatie voor arrays

In dit aanhangsel schetsen we hoe de funktie arrayADT van deel 4.2 gedefinieerd kan worden. We geven twee uitwerkingen. In de eerste kiezen we ervoor om array-waarden als funkties te representeren. In de tweede worden arraywaarden ter lengte n als n-voudige paren, i.e. <<...,... >,...>, gerepresenteerd. Dus dan hangt de eerste komponent van arrayADT(n, real) op niet-triviale wijze van n af!

De eerste methode, arrays als funkties. De subscriptie wordt daarmee triviaal: funktietoepassing (op de index). Ook de rest is niet moeilijk. Hier is de uitwerking.

```
    def arrayADT: (elt ∈ tp, int → < repr ∈ tp, (..), ... >)
            = (elt: tp, lth: int):
                < {repr=} (int → elt)
                , {upd=} (a: (int → elt), i: int, e: elt): a'
                        where a'= (j: int): if i=j then e else a(j)
                , {sub=} (a: (int → elt), i: int): a(i)
                .
                .
                >
```

Nu de tweede methode, arrays als geneste paren. We maken gebruik van een voorgedefinieerde funktie

error: (z ∈ tp → z)

We laten de indices van een array ter lengte lth lopen van 1 tot en met lth. Foutieve indicering leidt tot "error".

```
def arrayADT: (elt ∈ tp, int → <repr ∈ tp, .....>)
 = (elt: tp, lth: int):
    if lth = 0
      then <{repr:=} tp {dit is "zo maar een keuze"}
           ,{upd:=} (a: tp, i: int, e: elt): error(tp)
           ,{sub:=} (a: tp, i: int): error(elt)
           .
           .
           >
      else def innerADT: ... == arrayADT(elt, lth-1)
           , repr: tp == <innerADT.1, elt>
           in <{repr:=} repr
              ,{upd:=} (a: repr, i: int, e: elt):
                if i=lth then <a.1, e>
                else <innerADT.2 (a.1, i, e), a.2>
              ,{sub:=} (a: repr, i: int):
                if i=lth then a.2
                else innerADT.3 (a.1, i)
              .
              .
              >
           ni
```

Aanhangsel B. Type-equivalentie in de conventionele typering

We beschouwen de taal van hoofdstuk 3, met name deel 3.1 uitgebreid met expliciete type-vermeldingen bij introducties (=definities en parameters). We breiden die taal nu ook nog uit met de mogelijkheid om de definitievorm

z = t

toe te laten, voor type-expressie t en type-identifier z. Aldus kunnen nieuwe typen worden gedefinieerd, vergelijk de mode definities in Algol 68 en de type-definities in Pascal. De vraag rijst nu, hoe we "gelijkheid van typen" moeten definieren die geeist wordt bij aanroep (tussen argumenttype en parametertype), bij selectie, toekenning, if-then-else, enzovoorts. Er zijn twee uitersten: structurele gelijkheid, en naam-gelijkheid.

Bij structurele gelijkheid is de betekenis van de type-definitie z = t volkomen bepaald door in het geldigheidsgebied van die definitie overal z door t te vervangen. Dus zo'n definitie komt overeen met wat wij in de SVPtypering met z == t noteerden. Op een (kleine?) uitzondering na worden mode-definities in Algol 68 zo opgevat. Bij structurele gelijkheid kunnen geen representatiekeuzen syntactisch afgeschermd worden. Dus een verandering van de definitie z = t tot z = t' kan veranderingen tot gevolg hebben verspreid over de gehele programmatekst, en niet beperkt tot dat deel waar de "typische operaties voor type z" worden gedefinieerd.

Bij naam-gelijkheid geldt per definitie dat expressies "van gelijk type zijn" als

(i) "hun introducerende type-voorkomens" samenvallen, of

(ii) zij eenzelfde identifier als type hebben.

Het "introducerend type-voorkomen" van een identifier x is het type-voorkomen dat bij de introductie van x vermeld is. Als voor e het introducerend type-voorkomen v bekend is, dan is dat voor e.l enzovoorts, en voor $e(e_1,\ldots,e_n)$ op een voor de hand liggende manier binnen v te vinden. (Maar voor de funktietekst $(x_1,\ldots,x_n)$:e en de recordexpressie $\langle e_1,\ldots,e_n \rangle$ lijkt een definitie voor het introducerend type-voorkomen in het algemeen niet of niet makkelijk te geven).

Een voorbeeld moge dit verduidelijken. Beschouw de definitie

```
def compl = <real, real>
  , x, y : compl = ...,...
  , z    : compl = ...
  , u    : <real,real> = ....
  , v, w : <real, real> = ...,...
in ...
```

Dan geldt dat x, y, z van gelijk type zijn, dat v, w van gelijk type zijn, en dat overige stellen identifiers (bv. x, u of u, v) van verschillend type zijn. Maar let op; x.1, x.2, y.1, y.2, ...u.1, u.2, ... w.1, w.2 zijn alle wel van gelijk type! (Overigens zijn in ADA v en w niet van gelijk type.)
De afscherming van representatiekeuzen die wordt geboden door x en u van verschillend type te verklaren, wordt weer teniet gedaan door x.1 en u.1 van gelijk type te laten zijn.

Een alternatieve beschrijving van naam-gelijkheid is de volgende. Vervang ieder samengestelde type-expressie t die niet in een definitie z = t voorkomt, door een nieuwe identifier z′ en voeg op geschikte plaats een definitie z′= t toe. Twee expressies in de oorspronkelijke tekst heten dan van gelijk type als ze in de nieuwe tekst eenzelfde identifier als type hebben. Deze beschrijving verklaart tevens de term "naam-gelijkheid". Bovenstaand voorbeeld ziet er dan als volgt uit:

```
def compl = <real, real>
  , x, y  : compl = ...,...
  ,  z    : compl = ...
  , new1 = <real, real>
  ,  u    : new1 = ...
  , new2 = <real, real>
  , v, w  : new2 = ...,...
in ...
```

Deze vorm van type-gelijkheid wordt in (veel varianten van) Pascal geeist, ook bij de internationale Pascal standaard.

De type-gelijkheid in Algol 68 verschilt van de structurele gelijkheid zoals wij die behandelen in het volgende opzicht. Recordtypen bevatten identifiers voor de komponenten; die kunnen bij selectie gebruikt worden in plaats van de weinig suggestieve selctoren 1, 2, ... Twee recordtypen heten nu gelijk als ze niet alleen gelijke komponenttypen hebben, maar ook gelijke komponentidentifiers. Dus na

```
    def compl1 = <re: real, im: real>
     , compl2  = <rho: real, phi: real>
```

zijn compl1 en compl2 verschillende typen, maar met x: compl1 en y: compl2 zijn de typen van x.re, x.im, y.rho, y.phi weer gelijk.

Met een kleine taaluitbreiding is het wel mogelijk nieuwe typen te definieren zo dat slechts over een beperkt tekstgedeelte bekend is voor welk type de nieuwe type-identifier staat. Aldus worden "representatiekeuzen" afgeschermd, en kan men --in beperkte mate-- zogenaamde abstracte datatypen programmeren. We schetsen de taaluitbreiding met behulp van een voorbeeld: de programmering van het datatype der complexe getallen. Beschouw de volgende tekst.

```
def newtp compl = <real, real>
    with zero: compl = <0.0, 0.0>
         addc: (compl, compl  →  compl)
             = (a:<real,real>, b:<real, real>): <a.1 + b.1, a.2 + b.2>
         mkCompl: (real, real  →  compl)
             = (x: real, y: real): <x,y>
         realPart: (compl  →  real)
             = (a: <real, real>): a.1
         imPart: (compl  →  real)
             = (a: <real, real>): a.2
    end
in ...
ni
```

Hier worden een nieuwe typeidentifier, en nog vijf gewone identifiers gedefinieerd, die allen tussen in en ni gebruikt mogen worden. In dat gebied, tussen in en ni, is compl een typenidentifier die qua typeringseisen ongerelateerd is aan <real, real> ! Dus de gebruiker van dit stel definities kan geen gebruik maken van een eventuele kennis van de representatiekeuze voor complexe getallen --in dit geval: een complex getal wordt door zijn cartesische coordinaten, en niet zijn polaire, voorgesteld-- . De gelijkheid compl = <real, real> geldt wel binnen het gebied tussen with en end. Daarom zijn zero: compl = <0.0, 0.0>, enzovoorts, goedgetypeerd.

Wijzigingen in de representatiekeuze voor complexe getallen blijven nu beperkt tot het gedeelte tussen with en end; de rest van het programma behoeft niet gewijzigd te worden. Dat is het grote voordeel van deze taaluitbreiding. Maar helaas kunnen veel datatypen niet met bovenstaand mechanisme geprogrammeerd worden. Met name zijn dat geparameteriseerde datatypen zoals: rijen van elementen (met het elementtype als parameter), en rijen van getallen met een gegeven lengte (bv. array (1..n) of int, of stack (n) of int)). In hoofdstuk 4 en 5 behandelen we mogelijkheden om geparameteriseerde datatypen te definiëren. De mogelijkheid van hoofdstuk 5 lijkt erg veel op bovengeschetste manier.

Aanhangsel C Een unificatie algoritme

Bij de type-polymorfie van Hoofdstuk 5 moet een stel typevergelijkingen worden opgelost. Wij geven een algoritme dat berekent of er een oplossing is en, indien dat zo is, ook de minst beperkende ofwel meest algemene oplossing berekent. Het oplossen van zo'n stel vergelijkingen staat bekend als unificeren.

Allereerst definieren we het begrip oplossing. Zij gegeven een verzameling V van vergelijkingen tussen typeexpressies waarin type-variabelen z als onbekenden voorkomen. Een oplossing van V is een verzameling vergelijkingen $z_i=t_i$ $(i=1,\ldots,n)$ zo dat alle z_i onderling verschillen en niet voorkomen in de t_j, en substitutie van t_i voor z_i $(i=1,\ldots,n)$ het stelsel V omzet in vergelijkingen tussen identieke expressies. De algoritme luidt nu als volgt.

Begin met de initieel lege oplossing.
Kies herhaaldelijk een van de vergelijkingen, zeg vg, en behandel die als volgt:

- als vg van de vorm z=z is, laat hen dam weg uit het stel vergelijkingen;
- als vg van de vorm z=t of t=z is, waarbij z niet in t voorkomt, dan
  - laat vg weg uit het stel vergelijkingen,
  - vervang in de tot nu gevormde oplossing en in het restant der vergelijkingen overal z door t, en
  - voeg z=t toe aan de oplossing;
- als vg van de vorm <t1,...,tn> = <t1',...,tn'> is, of (t1,..,tn-1 → tn) = (t1',...,tn-1' → tn'), vervang vg dan door het stel t1=t1',...,tn=tn';
- als vg niet van een der bovengenoemde vormen is (met name: z=...z..., <-,-> = <-,...,-> of <....> = (....→..)), dan is er geen oplossing mogelijk.

Ga hiermee door totdat 'geen oplossing mogelijk' is vastgesteld, of het stel vergelijkingen leeg is geworden.

De algoritme termineert, omdat per slag van de herhaling het aantal onbekenden in het stel op te lossen vergelijkingen afneemt, hetzij --bij gelijkblijvend aantal onbekenden-- het aantal symbolen (of het aantal subexpressies) van het op te lossen stel afneemt.

De rest van de korrektheidsargumentatie gaat als volgt. Definieer dat twee stelsels vergelijkingen gelijkwaardig zijn als zij dezelfde oplossingen hebben, en dat een oplossing de meest algemene is als hij gelijkwaardig is met het op te lossen stelsel. Het is dan niet moeilijk te bewijzen dat gedurende de herhaling het nog op te lossen stelsel verenigd met de alreeds gevormde oplossing gelijkwaardig is met het oorspronkelijke stelsel. Dus is de opgeleverde oplossing de meest algemene (en onafhankelijk van de keuzen die steeds voor vg gemaakt mogen worden).

Door steeds geschikte keuzen voor vg te maken, en de vergelijkingen geschikt te representeren kan bereikt worden dat de tijdsduur en het ruimtegebruik lineair zijn in het aantal symbolen van het op te lossen stelsel, zie [Martelli, Montanari 1982].

Literatuurverwijzingen

de Bruijn, N.G.: A survey of the project AUTOMATH. In: Combinatory Logic, lambda calculus and formal systems. (Eds Hindley and Seldin) Academic Press (1980).

Constable, R.L. & Zlatin, D.R. : The type theory of PL/CV3. In Logic of programs, Proc. (Ed D. Kozen), LNCS 131 (1983) pp 72-93

Dijkstra, E.W.: A discipline of programming. Prentice Hall, (1976).

Fokkinga, M.M.: On the notion of strong typing. In: Algorithmc Languages (Eds J.W. de Bakker, J.C. van Vliet). North-Holland, Amsterdam (1981a), pp 305-320.

Fokkinga, M.M.: ongepubliceerd manuscript (1981 b)

Fortune, S. & Leivant, D. & O'Donell, M.: The expressiveness of simple and second order type structures. IBM RC 8542 (# 37221) (1980)
Ook: J ACM 30 (1983)1, pp 151-185.

Geurts, L.J.M.: An overview of the B programming language, or B without tears. Report IW 208/82, Mathematical Centre, Amsterdam (1982). Ook: SIGPLAN Notices 17 (1982) 12, pp 49-58

Gries, D.: Programming methodology - a collection of articles by members of IFIP WG 2.3, Springer-Verlag, New York (1978)

Grune, D. & Bosch, R. & Meertens, L.G.L.T.: Aleph Manual. IW 17/75, Mathematical Centre, Amsterdam (1975)

Ichbiah, J.D. et al: Rationale for the design of the Ada programming languages. In Sigplan Notices 14 (1979) 6.

Koster, C.H.A.: CDL - A compiler Description Language. TU Berlin, September 1975

Martelli, A. & Montanari, U.: An efficient unification algorithm.
ACM Toplas 4 (1982)2, pp 258:282.

Martin-Lof, P.: An intuitionistic theory of types - predicative part.
In Logic colloquium 73 (Eds Rose, Shepherdson), North-Holland, Amsterdam (1975) pp 73-118

Meertens, L.G.L.T.: Incremental polynomorphic type-checking in B.
IW 214/82, Mathematical Centre, Amsterdam, (1982).

Milner, R.: A theory of type polymorphism in programming.
JCSS 17 (1978) pp 348-375

Nordstrom, B.: Programming in Constructive Set Theory - some examples. Proc. Conf. on Functional Programming Languages and Compiler Archictecture, ACM, 1981

Reynolds, J.C.: Towards a theory of type structure. In: Symp. on programming. LNCS 19 (1974) pp 408-425.

Reynolds, J.C.: The essence of Algol. In: Algoritmic Languages (Eds J.W. de Bakker, J.C. van Vliet). North-Holland, Amsterdam (1981) pp. 345-372.

Scott, D.: Data types as lattices. Siam J on Computing 5 (1976) pp 522-587.

"Steelman": The U.S.A. Department of Defense Requirements for high order computer programming languages (1978)

Swierstra, S.D.: Lawine - an experiment in language and machine design. Dissertatie, T.H. Twente, (1981).

Tennent, R.D.: Principles of Programming Languages. Prentice-Hall, Englewood Cliffs N.J. (1980)

Van Wijngaarden, A. et al: Revised Report on the algoritmic language Algol 68. Springer-Verlag, Berlin, (1976).