TECHNISCHE HOGESCHOOL EINDHOVEN

Onderafdeling der Wiskunde en Informatica

Memorandum 1981-12

Augustus 1981

BESCHRIJVING VAN AUT-68

door

L.S. van Benthem Jutting

Technische Hogeschool

Onderafdeling der Wiskunde en Informatica

Postbus 513, 5600 MB Eindhoven

Nederland

Beschrijving van AUT-68

door

L.S. van Benthem Jutting

Deze beschrijving van de oudste AUTOMATH-taal, AUT-68, is informeel. De eerste paragraaf (§ 0) bevat een inleiding waarin de achtergronden van de taal worden geschetst en een kort historisch overzicht wordt gegeven. Dan wordt (in § 1 t/m § 4) een vereenvoudigde (ongetypeerde) versie van de taal gepresenteerd. In tweede instantie (§ 5 t/m 8) worden typen toegevoegd. In § 9 worden de taaltheoretische resultaten in vogelvlucht vermeld. Voor de gebruiker is § 10 van belang.

0. Inleiding

AUTOMATH is een taal om wiskunde te beschrijven op een zodanig precieze manier dat verificatie van een tekst machinaal (met een computer) kan worden uitgevoerd.

Voordelen van zo'n taal zijn:

i) De taal vergroot de zekerheid dat lange "vervelende" bewijzen van niet intuïtief voor de hand liggende stellingen formeel correct zijn.

ii) De taal legt bloot welke stappen in een bewijs moeilijk zijn. Sommige moeilijkheden zijn wellicht inherent aan de taal (die haar beperkingen heeft), maar soms geeft het gebruik van de taal ons ook meer inzicht in de moeilijkheden van een bewijs.

iii) De taal kan (mede in verband met het bovenstaande) zijn nut hebben bij de didactiek van wiskunde. Zij dwingt tot bezinning op de structuur van bewijzen en maakt het mogelijk na te gaan welke axioma's en afleidingsregels in een bewijs zijn gebruikt.

iv) In de taal zou een soort "databank van de wiskunde" kunnen worden opgebouwd, een bibliotheek van wiskundeboeken die naar elkaar kunnen verwijzen.

AUTOMATH-talen verschillen in hun opzet van de meeste andere formele systemen waarin wiskunde beschreven kan worden. In dergelijke theorieën is meestal predicatenlogica met haar axioma's en afleidingsregels vooronderstelt. Daaraan worden de axioma's toegevoegd over de wiskundige objecten die worden besproken, b.v. verzamelingen, relaties, functies, getallen, vectorruimten etc. In AUTOMATH daarentegen zijn functies, typen en afleidingsregels daarover in de taal vooronderstelt. Hieraan worden toegevoegd de axioma's over andere zaken, b.v. proposities, predicaten, verzamelingen, getallen.

N.G. de Bruijn is de geestelijke vader van AUTOMATH. Hij heeft de taal ontworpen, de ideeën geleverd en verbreid, de initiatieven genomen en de organisatie van het AUTOMATH-project geleid, en veel werk gedaan aan formele definities, experimenten met practisch gebruik en taaltheorie. Verdere medewerkers zijn of zijn geweest:

Nederpelt, De Vrijer, Van Daalen : taaltheorie

Zandleven, Kornaat : computerprogramma voor verificatie

Van Benthem Jutting, Zucker, Wieringa : schrijven van boeken in AUTOMATH.

Werk dat is verricht:

i) Construeren van de taal, taaldefinities, theorie over de taal, bewijzen van consistentie en beslisbaarheid.

ii) Schrijven van boeken in de taal:

Vertaling van E. Landau, Grundlagen der Analysis.

Een boek over grondslagen van de reële analyse: reële getallen, differentiëren, machtreeksen, exponentiële functies.

Een boek over correctheid van computerprogramma's.

iii) Het maken van een interactief verificatie-programma.

iv) Stages en afstudeeropdrachten door studenten, o.m. Marcelis, Penning, Wieringa, Udding, Braun.

1. Boek en context

Alles wat in de taal geschreven wordt, moet worden gelezen in een zekere "context". We gebruiken dit woord hier nog informeel en in een wijde zin, het krijgt hieronder een engere en meer formele betekenis.

In deze "context in wijde zin" onderscheiden we twee delen:

i) Alles wat reeds eerder is gedefinieerd of afgesproken, primitieve begrippen die zijn ingevoerd, axioma's daarover en bewezen stellingen.
Dit gedeelte heet het boek.
Het boek bevat alles wat reeds in de taal is geschreven, en wordt uitgebreid met wat we schrijven.

ii) De variabelen die geldig zijn ("gedeclareerd" zijn) op de plaats waar we schrijven, en de onderstellingen die op die plaats gelden.
Dit noemen we de context (in enge zin).
De context is locaal, hij kan worden uitgebreid en ingekrompen; de context kan ook leeg zijn.

2. Lijnen, naam en inhoud; expressies

Het boek is een (eindige) rij van lijnen ("lijn" staat voor regel = line, ter onderscheiding van regel = rule).

Een lijn bestaat uit:

i) Een context.
Dit is de formalisering van bovenstaande "context in enge zin".

In deze voorlopige (ongetypeerde) versie bestaat de context uit een eindige rij van onderling verschillende variabelen. We noteren deze variabelen tussen vierkante haken,

b.v. [x][y][z].

De context kan leeg zijn. De lege context duiden we aan met ∅.

ii) Een naam.

Deze moet *nieuw* zijn, d.w.z. hij mag niet al in het boek voorkomen. Achter de naam vermelden we de variabelen uit de context,

b.v. [x][y][z] a(x,y,z).

← context → ↑ naam

iii) Een inhoud.

De inhoud geeft betekenis aan de naam, de naam is een aanduiding (of afkorting) voor de inhoud. Naam en inhoud scheiden we door het symbool := .

Voor de inhoud zijn twee mogelijkheden:

a) De inhoud kan zijn het woord 'prim'.

De *interpretatie* is dat de naam een primitief object aanduidt.

b.v. Een boek over natuurlijke getallen kan beginnen met de lijnen:

∅ 1 := 'prim'

[x] suc(x) := 'prim'

In deze lijnen zijn 1 en de opvolger als primitieve begrippen ingevoerd.

b) De inhoud kan ook een expressie zijn.

We behandelen expressies in de volgende paragraaf.

3. Expressies, abstractie en applicatie

We definiëren in deze paragraaf wat expressies zijn. We gaan daarbij uit

van een boek en een context, die we op de meeste plaatsen niet expliciet noemen.

i) Een variabele uit de context is een expressie.

ii) Als het boek een lijn bevat

[x][y][z] a(x,y,z) := ...

en als A,B,C expressies zijn, dan is a(A,B,C) een expressie.

Analoog voor langere of kortere contexten.

Dit soort expressies heten constante-expressies.

b.v. Het in de vorige paragraaf begonnen boek kunnen we uitbreiden met de lijnen:

∅	2	:= suc(1)
∅	3a	:= suc(2)
[x]	plus2(x)	:= suc(suc(x))
∅	3b	:= plus2(1)

iii) Als in de context [x][y][z], A een expressie is, dan is in de context [x][y], [z]A een expressie.

Dit soort expressies heten abstractie-expressies.

Opm. z kan dus "in A voorkomen" hoewel z niet in de context voorkomt.

Interpretatie. [z]A betekent:

de functie die aan z toevoegt de functiewaarde A.

NB. [z]A wordt in de λ-calculus genoteerd als λz.A . (Dat we van deze gebruikelijke notatie afwijken komt onder meer door beperkingen van de verzameling van karakters op onze input/output-apparatuur).

b.v. T.o.v. het bovenstaande boek zijn op lege context de volgende uitdrukkingen expressies:

[x]suc(x)

[x]x

[y]1.

Ze hebben als interpretatie achtereenvolgens: de opvolgerfunctie, de identieke functie en de constante functie met waarde 1.

We kunnen het boek uitbreiden met de lijnen:

Ø sucf := [x]suc(x)

Ø idf := [x]x

iv) Als A en B expressies zijn, dan is <A>B een expressie.

Dit soort expressies heten applicatie-expressies.

Interpretatie. <A>B betekent:

de functiewaarde van de functie B in het punt A.

NB. <A>B wordt in gewone teksten meestal genoteerd als B(A);

<A>B wordt in de λ-calculus genoteerd als (BA).

Wij schrijven de functie dus achter het argument.

Daar zijn technische redenen voor, het is even wennen.

b.v. T.o.v. het bovenstaande boek zijn op lege context de volgende uitdrukkingen expressies:

<2>sucf

<3a>idf

<1>[x]2

met als interpretatie respectievelijk:

de waarde van de opvolgerfunctie in het punt 2,

de waarde van de identieke functie in het punt 3a,

de waarde van de constante functie 2 in het punt 1.

Het boek kan worden uitgebreid met de lijn:

Ø 3c := <2>sucf.

Verder is op context [f][g] de uitdrukking

[x]<<x>g>f een expressie.

De interpretatie hiervan is: de functie die aan iedere x toevoegt de waarde van f in het punt <x>g, dat is de compositie van f en g, f o g.

We breiden het boek nog uit met de lijnen:

[f][g] comp(f,g) := [x]<<x>g>f

Ø plus2f := comp(sucf,sucf)

Het boek is nu:

Ø	1	:=	'prim'
[x]	suc(x)	:=	'prim'
Ø	2	:=	suc(1)
Ø	3a	:=	suc(2)
[x]	plus2(x)	:=	suc(suc(x))
Ø	3b	:=	plus2(1)
Ø	sucf	:=	[x]suc(x)
Ø	idf	:=	[x]x
Ø	3c	:=	<2>sucf
[f][g]	comp(f,g)	:=	[x]<<x>g>f
Ø	plus2f	:=	comp(sucf,sucf)

4. Substitutie, reductie, definitiegelijkheid

Zij E een expressie op context [x][y][z].

Laat A,B,C expressies zijn op een zekere context c. Dan is

$$S \begin{smallmatrix} x & y & z \\ A & B & C \end{smallmatrix} E$$

de expressie die je krijgt door in E simultaan x door A, y door B en z door C te vervangen.

$S \begin{smallmatrix} x & y & z \\ A & B & C \end{smallmatrix} E$ is dus een expressie op context c.

Opm. 1. Het zou kunnen dat in E variabelen uit de context c *gebonden* voorkomen; dat geeft problemen, b.v.

$$S_y^x\ [y\ ]\mathrm{suc}(x) \neq [y\ ]\mathrm{suc}(y)$$

Deze problemen zijn bekend, en opgelost, we gaan er hier niet op in.

2. De substitutie-operator is een operator *in de metataal*.

In het volgende gaan we ervan uit dat substitutie goed gedefinieerd is. I.v.m de bovenvermelde interpretatie ligt het voor de hand sommige expressies als "hetzelfde betekenend" of "gelijk" te beschouwen. Zo lijken in ons boek 3a, 3b en 3c allemaal hetzelfde te betekenen, nl. suc(suc(1)).

We kunnen gelijkheid vaststellen door expressies "uit te werken". Dit "uitwerken" noemen we reduceren. Er zijn in onze taal twee belangrijke reductieprincipes:

i) δ-reductie:

Als [x][y][z] a(x,y,z) := E een lijn is in het boek en als a(A,B,C) een expressie is op zekere context, dan geldt:

$$a(A,B,C) >_\delta S \begin{smallmatrix} x & y & z \\ A & B & C \end{smallmatrix} E.$$

We zeggen: a(A,B,C) reduceert naar $S \begin{smallmatrix} x & y & z \\ A & B & C \end{smallmatrix} E$.

δ-reductie komt dus neer op "definitie toepassen".

b.v. $3b >_\delta \mathrm{plus2}(1) >_\delta \mathrm{suc}(\mathrm{suc}(1))$.

ii) β-reductie:

Als <A>[x]B een expressie is dan geldt:

$$\langle A\rangle[x\ ]B >_\beta S_A^x B.$$

We zeggen <A>[x]B reduceert naar $S_A^x B$.

β-reductie komt dus neer op "functie toepassen".

b.v. <1>[x]<<x>[y]suc(y)>[z]suc(z) $>_\beta$

<<1>[y]suc(y)>[z]suc(z) $>_\beta$

suc(<1>[y]suc(y)).

> is de monotone afsluiting van $>_\beta$ en $>_\delta$. De relatie > heet reductie.

b.v. 1. <1>[x]<<x>[y]suc(y)>[z]suc(z) $>_\beta$ ... $>_\beta$

suc(<1>[y]suc(y)) >

suc(suc(1)) (vergelijk voorbeeld bij β-reductie)

2. 3c > <2>sucf > <2>[x]suc(x) > suc(2) > suc(suc(1))

3. plus2f > comp(sucf,sucf) > [x]<<x>sucf>sucf >

[x]<<x>[y]suc(y)>sucf > [x]<suc(x)>sucf >

[x]<suc(x)>[y]suc(y) > [x]suc(suc(x))

$\equiv_D$ is de equivalentierelatie voortgebracht door >. De relatie $\equiv_D$ heet definitiegelijkheid.

Interpretatie. twee definitiegelijke expressies zijn namen voor hetzelfde object.

b.v. 1. 3a $\equiv_D$ 3c want 3a > suc(2) en 3c > suc(2)

2. 3a $\equiv_D$ 3b want 3a > suc(suc(1)) en 3b > suc(suc(1))

3. 3b $\equiv_D$ 3c want 3a $\equiv_D$ 3c en 3a $\equiv_D$ 3b.

Opgaven. 1. bewijs 3a $\equiv_D$ <1>plus2f

2. bewijs plus2f $\equiv_D$ [x]plus2(x)

5. Typering

Tot zover hebben we een "ongetypeerde versie" gegeven van AUT-68. Bezwaren hiertegen zijn b.v.:

1) In ons boek kun je t.o.v. een lege context opschrijven

comp(1,2)

wat reduceert naar

[x]<<x>suc(1)>1.

Dit is niet goed te interpreteren.

2) Er doen zich rare verschijnselen voor, er zijn b.v. expressies die je kan *blijven* reduceren:

<[x]<x>x>[x]<x>x >$_\beta$ naar zichzelf!

Deze bezwaren kunnen we opheffen door van ieder object de "funktionele structuur" vast te leggen. We willen voor ieder object kunnen vaststellen of het een functie is of niet, en zo ja, wat zijn "domein" en zijn "bereik" zijn. We eisen dan ook nog dat alléén *functies* mogen worden "toegepast" en dan nog alléén op objecten uit hun domein. Dit wordt bereikt door typering. Het toevoegen van typen geeft de taal AUT-68.

In AUT-68 zijn de objecten (aangeduid door expressies) ingedeeld in disjuncte klassen, de zgn. typen. Ook deze typen worden aangeduid door expressies.

We geven eerst een aantal definities en conventies in de metataal:

Expressies die objecten aanduiden noemen we 3-expressies.

" " typen " " " 2-expressies.

Het woord 'type' " " 1-expressie.

We duiden 3-expressies aan met hoofdletters: A,B,C,...

" " 2-expressies " " griekse letters: $\alpha,\beta,\gamma,...$

2-expressies hebben dezelfde syntactische structuur als 3-expressies; het kunnen zijn:

variabelen

constante-expressies

abstractie-expressies

applicatie-expressies.

Het zal van elke expressie vaststaan of het een 3-expressie of een 2-expressie is, en als het een 3-expressie is, wat zijn type is.

In de taal hebben we formules:

A:α met interpretatie: A heeft type α

α:'type' " " : α is een type.

6. Contexten en lijnen

Op een aantal plaatsen geeft de typering aanleiding tot wijziging van de eerder gegeven taalregels.

i) contexten, variabelen voor objecten, variabelen voor typen

Bij de variabelen in de context wordt aangegeven wat hun type is als de variabele een 3-expressie is: [x:α] en er wordt aangegeven dat hij een type is als de variabele een 2-expressie is: [y:'type']

b.v. Een mogelijke context is dus

[x:'type'][y:x].

ii) de inhoud van een lijn

Zoals in §2 vermeld, geeft de inhoud van de lijn de betekenis van een naam.

a) De naam kan een primitief object aanduiden. Dan moet echter zijn type vaststaan, de inhoud is dan

'prim':α

De naam kan ook een primitief type aanduiden. De inhoud is dan

'prim':'type'.

b.v. Een AUT-68 boek over natuurlijke getallen kan beginnen met de

lijnen:

```
Ø        nat     := 'prim' : 'type'
Ø        1       := 'prim' : nat
[x:nat]  suc(x)  := 'prim' : nat
```

In deze lijnen is nat ingevoerd als een primitief type, het type van de natuurlijke getallen. Daarna zijn 1 en de opvolger van een natuurlijk getal als primitieve begrippen ingevoerd.

We kunnen nu ook, voor willekeurige n, $\mathbb{R}^n$ als primitief type invoeren:

```
[n:nat]  R(n) := 'prim' : 'type'.
```

En daarna in iedere $\mathbb{R}^n$ de nulvector als primitief object:

```
[n:nat]  0(n) := 'prim' : R(n).
```

b) De naam hoeft niet een primitief object of een primitief type aan te duiden, hij kan ook een gedefinieerd object aanduiden of een gedefinieerd type.

De inhoud van de lijn is in deze gevallen een formule:

A:α resp. α:'type'.

b.v. Het boek over natuurlijke getallen kan doorgaan met:

```
Ø        2          := suc(1)          : nat
Ø        3a         := suc(2)          : nat
[x:nat]  plus2(x)   := suc(suc(x))     : nat
Ø        3b         := plus2(1)        : nat
```

7. De typeringsoperator τ

Het zal duidelijk zijn dat we hierboven als laatste lijn niet willen accepteren

```
Ø   3b := plus2(1) : R(2).
```

Dat dit niet acceptabel is kan "mechanisch" worden vastgesteld. We kunnen nl. bij ieder object (gegeven door een 3-expressie) zijn type berekenen. Er is een operator τ: 3-expressies $\rightarrow$ 2-expressies zó dat voor iedere 3-expressie A geldt $\quad A : \tau(A)$.

We hebben vier gevallen:

i) $A \equiv x$, een variabele.

Dan is x gegeven met zijn type: $[x:\alpha]$ en er geldt $\tau(x) \equiv \alpha$.

ii) $A \equiv a(B,C)$, een constante-expressie.

Dan staat in het boek een lijn:

$[x:\ldots][y:\ldots] \quad a(x,y) := E : \alpha$

of $[x:\ldots][y: \quad] \quad a(x,y) :=$ 'prim' $: \alpha$

en er geldt: $\tau(a(B,C)) \equiv S^{x\ y}_{B\ C}\,\alpha$.

b.v. $\tau(\text{plus2}(1)) = S^{x}_{1}\text{nat} \equiv \text{nat}$

$\tau(0(2)) \equiv S^{n}_{2}\mathbb{R}(n) \equiv \mathbb{R}(2)$.

iii) $A \equiv [x:\alpha]B$, een abstractie-expressie.

A is dan een functie, de variabele x heeft ook in abstractie-expressies zijn type bij zich. De functie A heeft als domein α en als bereik het type van B. We definiëren: $\tau([x:\alpha]B) \equiv [x:\alpha]\tau(B)$.

NB. $[x:\alpha]\beta$ moet dus worden geïnterpreteerd als "het type van de functies van α naar β". $[x:\alpha]\beta$ stelt *geen functie* voor!

Complicatie. Zoals we hebben gezien kunnen ook typen afhangen van een parameter: we construeerden al het type $\mathbb{R}(n)$ voor ieder natuurlijk getal n. Dit geeft ons (in de interpretatie) een "rij van typen" $\mathbb{R}(1)$, $\mathbb{R}(2)$,..., maar voor deze rij als "object" is geen expressie in de taal. Wel geldt:

$\tau([x:\text{nat}]0(x)) \equiv [x:\text{nat}]\tau(0(x)) \equiv [x:\text{nat}]\mathbb{R}(x)$.

[x:nat]**R**(x) moet dus worden geïnterpreteerd als "het type van de functies die aan elke n ∈ **N** een vector in **R**(n) toevoegen"

ofwel

"het cartesisch produkt **R**(1) × **R**(2) × **R**(3) × ...".

Voor de rij **R**(1), **R**(2), **R**(3),... is dus geen expressie in de taal, maar wel voor zijn cartesisch produkt.

Samenvattend. [x:α]B wordt geïnterpreteerd als een functie.
Als x niet in B voorkomt is dit een constante functie.

[x:α]β wordt geïnterpreteerd als een cartesisch produkt.
Als x niet in β voorkomt is dit isomorf met β^{α}.

iv) A = <B>C.

We definiëren τ(<B>C) = <B>τ(C).

We eisen nu voor de inhoud van een lijn in het boek:

A : α is een correcte inhoud alléén als $\tau(A) \underset{D}{=} \alpha$.

b.v. We kunnen ons boek uitbreiden met de lijnen:

Ø	sucf	:= [x:nat]suc(x)	: [x:nat]nat
Ø	idf	:= [x:nat]x	: [x:nat]nat
Ø	3c	:= <2>sucf	: nat
[f:[x:nat]nat][g:[x:nat]nat]			
	comp(f,g)	:= [x:nat]<<x>g>f	: [x:nat]nat
Ø	plus2f	:= comp(sucf,sucf)	: [x:nat]nat
Ø	0f	:= [x:nat]0(x)	: [x:nat]**R**(x)

Merk op dat [x:nat]nat staat voor "het type van de functies nat → nat" en [x:nat]**R**(x) voor "het cartesisch produkt van de typen **R**(x)".
Merk ook op dat voor de lijn 3c geldt:

$\tau(<2>sucf) = <2>\tau(sucf) = <2>[x:nat]nat >_{\beta}$ nat.

[illegible] worden geïnterpreteerd als het cartesisch produkt

van aftelbaar veel copieën van **N**.

<2>[x:nat]nat moet worden geïnterpreteerd als de tweede copie van **N**, en dus niet als een zekere functie toegepast op 2.

β-reductie voor 2-expressies is dus het selecteren van een "coördinaat-as" van een cartesisch produkt.

8. Correctheid

We willen nu beperkingen aan expressies opleggen zodat er geen "expressies zonder interpretatie" ontstaan. Zo hebben we b.v. geen interpretatie voor <2>1 "omdat 1 geen functie is", en ook niet voor comp(1,2) "omdat 1 en 2 geen functies zijn". We zeggen in dit laatste geval: de rij (1,2) *past niet* bij de context [f:[x:nat]nat][g:[x:nat]nat] (dat is de context van de lijn waar comp is gedefinieerd).

We definiëren nu wat passen is.

In de definitie stellen de letters A,B en C 2- of 3-expressies voor (dit in afwijking van wat elders is afgesproken!).

Definitie. De rij (A,B,C) past in de context [x:...][y:...][z:...] als geldt:

$\tau(A) \underset{D}{=} \tau(x)$ of A en x zijn beide 2-expressies,

$\tau(B) \underset{D}{=} S^{x}_{A}\tau(y)$ of B en y zijn beide 2-expressies,

$\tau(C) \underset{D}{=} S\,{}^{x\ y}_{A\ B}\,\tau(z)$ of C en z zijn beide 2-expressies.

Analoog voor langere of kortere rijen en contexten.

Doordat substitutie een rol speelt, is de definitie gecompliceerd. Dat dit nodig is komt door contexten zoals

[x:'type'][y:x].

Een rij die hierin past is (nat,2).

Een rij die niet past is (nat,sucf).

We definiëren nu correctheid van 2- en 3-expressies, t.o.v. een gegeven

boek en een gegeven context.

i) Als [x:'type'] in de context, dan is x een correcte 2-expressie.

Als [x:α] in de context, dan is x een correcte 3-expressie.

ii) Als A,B,C correct zijn, A,B en C zijn 2- of 3-expressies, als (A,B,C) past in [x:...][y:...][z:...], en

a) als nu [x:...][y:...][z:...] a(x,y,z) := 'prim' : 'type'

of [x:...][y:...][z:...] a(x,y,z) := α : 'type'

een lijn is in het boek, dan is a(A,B,C) een correcte 2-expressie.

b) als [x:...][y:...][z:...] a(x,y,z) := 'prim' : α

of [x:...][y:...][z:...] a(x,y,z) = D : α

een lijn is in het boek, dan is a(A,B,C) een correcte 3-expressie.

iii) Als in context [x:...][y:...] α een correcte 2-expressie is, en

a) als nu in context [x:...][y:...][z:α] β een correcte 2-expressie is,

dan is in context [x:...][y:...] [z:α]β een correcte 2-expressie.

b) Als in context [x:...][y:...][z:α] A een correcte 3-expressie is,

dan is in context [x:...][y:...] [z:α]A een correcte 3-expressie.

iv) a) Als A een correcte 3-expressie is en α een correcte 2-expressie, en

als $\alpha \underset{D}{=} [x:\tau(A)]\beta$ voor zekere β,

dan is <A>α een correcte 2-expressie

Opm. α is een cartesisch produkt van typen, <A>α is een "coördinaat-as".

b) Als A en B correcte 3-expressies zijn en als

$\tau(B) \underset{D}{=} [x:\tau(A)]\beta$ voor zekere β,

dan is <A>B een correcte 3-expressie.

Opm. Omdat $\tau(B) \underset{D}{=} [x:\tau(A)]\beta$, is B een functie met domein τ(A).

We stellen als eis: alle expressies in een boek moeten correct zijn.

b.v. Dit geldt voor ons boek:

```
Ø          nat          := 'prim'              : 'type'
Ø          1            := 'prim'              : nat
[x:nat]    suc(x)       := 'prim'              : nat

[n:nat]    R(n)         := 'prim'              : 'type'
[n:nat]    0(n)         := 'prim'              : R(n)

Ø          2            := suc(1)              : nat
Ø          3a           := suc(2)              : nat
[x:nat]    plus2(x)     := suc(suc(x))         : nat
Ø          3b           := plus2(1)            : nat
Ø          sucf         := [x:nat]suc(x)       : [x:nat]nat
Ø          idf          := [x:nat]x            : [x:nat]nat
[f:[x:nat]nat][g:[x:nat]nat]
           comp(f,g)    := [x:nat]<<x>g>f      : [x:nat]nat
Ø          plus2f       := comp(sucf,sucf)     : [x:nat]nat

Ø          0f           := [x:nat]0(x)         : [x:nat]R(x)
```

9. Taaltheorie in een notedop

Het belangrijkste probleem bij verificatie is het vaststellen van definitiegelijkheid van expressies. De theorie van AUTOMATH, hoofdzakelijk ontwikkeld door Nederpelt en Van Daalen, houdt zich dan ook vooral bezig met definitiegelijkheid en reductie.

Zij >> de reflexieve en transitieve afsluiting van >. Een expressie heet normaal als zij niet reduceert. Een expressie A normaliseert als er een normale expressie B bestaat met A>>B.

Een expressie A normaliseert sterk als er geen oneindige rij $A_1, A_2, \ldots$

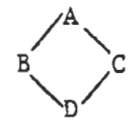
bestaat met $A > A_1 > A_2 > \ldots$.

Church-Rosser geldt voor A als geldt:

Als A >> B en A >> C

dan bestaat D met

B >> D en C >> D.



Als Church-Rosser geldt voor A dan is er hoogstens één normale B waarnaar A reduceert. Als A bovendien sterk normaliseert dan wordt B in eindig veel stappen bereikt, welke weg je ook kiest.

Voor veel AUTOMATH-talen (daaronder AUT-68) zijn Church-Rosser en sterke normalisatie bewezen voor alle correcte expressies. Dit geeft "theoretische" beslisbaarheid van de relatie $\underset{D}{=}$. Voor "practische" beslisbaarheid is een goede strategie van het programma nodig. Het is niet doenlijk om definitie-gelijkheid alleen via normale expressies vast te stellen.

10. Enkele technische opmerkingen

i) Contextbeheer

In tegenstelling tot wat tot hiertoe is gesteld, worden contexten uitgebreid in aparte lijnen, zgn. contextlijnen. In gewone lijnen wordt eventueel naar zo'n contextlijn verwezen door een context-indicatie. De context-indicatie wordt van de naam gescheiden door het symbool @. Lijnen zonder @ hebben de context van de vorige lijn. We illustreren het gebruik aan een voorbeeld:

b.v.

@[x:α][y:β][z:γ]	context-uitbreiding (op Ø)
a(x,y,z) := ...	context [x][y][z]
b(x,y,z) := ...	context [x][y][z]
@ c := ...	context Ø

```
y @ d(x,y)        :=  ...      context [x ][y ]
x @ e(x)          :=  ...      context [x ]
[z:δ]                          context-uitbreiding (op [x ])
f(x,z)            :=  ...      context [x ][z:δ]
y @ g(x,y)        :=  ...      context [x ][y ]
z @ h(x,z)        :=  ...      context [x ][z:δ]
y @ [t:ε][u:ζ]                 context-uitbreiding (op [x ][y ])
t @ j(x,y,t)      :=  ...      context [x ][y ][t ]
```

ii) Weglaten van variabelen

In tegenstelling tot wat tot hiertoe is gesteld, worden bij de namen niet de variabelen uit de context vermeld. In het stuk hierboven wordt dan ook niet geschreven

a(x,y,z) := ..., b(x,y,z) := ..., d(x,y) := ... etc,

maar a := ..., b := ..., d := ... etc.

De variabelen kunnen worden afgelezen uit de context.

Verder is er een mogelijkheid om op andere plaatsen variabelen weg te laten achter een naam, vooraan te beginnen. Als bijvoorbeeld de naam a gedefinieerd was op context [x:α][y:β] dan mag later a(x,A) worden afgekort tot a(A), en a(x,y) tot a.

Opm. a(A,y) kan niet worden afgekort!

iii) De vorm van namen; (in)fix-schrijfwijze

Als naam is iedere rij van letters, cijfers en _ (onderstreepte spatie) toegestaan.

b.v. de_bruijn

1_9_81

aut_68

Dergelijke namen kunnen ook als variabelen worden gebruikt.Het is handig om voor variabelen zoveel mogelijk enkele letters te kiezen en voor namen

langere woorden. Ook is het handig om "sprekende" namen te bedenken voor belangrijke begrippen.

Om de leesbaarheid voor mensen te vergroten is ook (in)fix-notatie toegestaan. Er zijn twee soorten (in)fix-namen:

a) Een aantal speciale symbolen, w.o. =, +, -, *, |, &

b) Vrijwel alles wat geschreven wordt tussen quotes

b.v. '<', '+-', 'of', '=j'

Fix-notatie wordt opgevoerd met (1 of 2) variabelen

b.v. @[x:nat]

```
    x'!' := ...              (voor het post-fix symbool '!').

@[x:nat][y:nat]
    x + y := ...             (voor het in-fix symbool +).
```

Verderop in het boek kan nu geschreven worden 2'!' en 2 + 1. Voor het ontleden van expressies zijn nu haakjes nodig: 1 + 2 + 3 mag niet, en 1 + 2 = 3 ook niet.

iv) Als teken voor "einde van de lijn" wordt gebruikt $.

We geven als illustratie nog een keer ons boek; we hebben sommige namen door fix-namen vervangen.

```
@     nat     := 'prim' : 'type' $
      1       := 'prim' : nat    $
[x:nat]                          $
      x'suc'  := 'prim' : nat    $

@ [n:nat]                        $
      ℝ       := 'prim' : 'type' $
      0       := 'prim' : ℝ(n)   $
```

```
@      2       := 1'suc'            : nat          $
       3a      := 2'suc'            : nat          $
x @    plus2   := (x'suc')'suc'     : nat          $
@      3b      := plus2(1)          : nat          $
       sucf    := [x:nat](x'suc')   : [x:nat]nat   $
       idf     := [x:nat]x          : [x:nat]nat   $

[f:[x:nat]nat][g:[x:nat]nat]                       $
       f'o'g   := [x:nat]<<x>g>f    : [x:nat]nat   $
@      plus2f  := sucf'o'sucf       : [x:nat]nat   $

       0f      := [x:nat]0(x)       : [x:nat]R(x)  $
```

v) Tenslotte willen we opmerken dat in oudere stukken over AUTOMATH andere notaties in zwang zijn:

b.v. PN voor 'prim';

x := EB : α of x := - : α voor contextuitbreiding [x:α]

(er wordt dan dus één variabele per lijn ingevoerd);

{A}B voor <A>B , [x,α]A voor [x:α]A;

A E α voor A : α;

* voor @.

Literatuur

1. N.G. de Bruijn: AUTOMATH, a language for mathematics; Een serie lezingen in Montréal, 1971. Lecture notes by B. Fawcett. Les presses de l'Université Montréal, 1973.

2. D.T. van Daalen: A description of AUTOMATH and some aspects of its language theory; Proceedings of the symposium APLASM Vol 1. Ed. P. Braffort, Orsay France, 1973.

3. J. Zucker: Formalization of classical mathematics in AUTOMATH; Actes of the International Logic Colloquium, Clermont Ferrand 1975.

4. R.P. Nederpelt: Presentation of natural deduction; Recueil des travaux de l'Institut Mathématique, Nouvelle série, tome 2(10), Beograd, 1977.

5. N.G. de Bruijn: A survey of the project AUTOMATH; To H.B. Curry. Essays in combinatory logic, lambda calculus and formalism. Academic Press 1980.

Appendix: voorbeeldtekst met commentaar

```
  @ prop              := 'prim'                    : 'type'                    1

[P:prop][q:prop][r:prop]                                                      2

p @ '⊢'p              := 'prim'                    : 'type'                    3

  @ [α:'type']

    pred              := [x:α]prop                 : 'type'                    4

[P:pred(α)]

    ∀                 := 'prim'                    : prop                      5

[u:[x:α]'⊢'<x>P]

    ∀_in              := 'prim'                    : '⊢'∀(α,P)                 6

P @ [u:'⊢'∀(α,P)][a:α]

    ∀_el              := 'prim'                    : '⊢'<a>P                   7

    spec              := ∀_el(α,P,u,a)             : '⊢'<a>P

q @ p'➡'q             := ∀('⊢'p,[x:'⊢'p]q)         : prop                      8

[u:[x:'⊢'p]'⊢'q]

    ➡_in              := ∀_in('⊢'p,[x:'⊢'p]q,u)    : '⊢'(p'➡'q)                9

q @ [u:'⊢'(p'➡'q)][v:'⊢'p]

    ➡_el              := ∀_el('⊢'p,[x:'⊢'p]q,u,v)  : '⊢'q

    mod_pon           := ➡_el(p,q,u,v)             : '⊢'q

p @ refl➡             := ➡_in(p,p,[x:'⊢'p]x)       : '⊢'(p'➡'p)               10
```

```
r @ [u:'⊢'(p'➡'q)][v:'⊢'(q'➡'r)]
    trans➡              := ➡_in(p,r,[x:'⊢'p]
                             ➡_el(q,r,v,➡_el(p,q,u,x)))      : '⊢'(p'➡'r)

  @ ⊥                   := ∀(prop,[x:prop]   x)             : prop          11

p @ [u:'⊢'⊥]
    ⊥_el                := ∀_el(prop,[x:prop]'⊢'x,u,p)      : '⊢'p

    '¬'p                := p'➡'⊥                            : prop          12

[u:[x:'⊢'p]'⊢'⊥]
    ¬_in                := ➡_in(p,⊥,u)                     : '⊢''¬'p

q @ [u:'⊢''¬'p][v:'⊢'p]
    ¬_el                := ⊥_el(q,➡_el(p,⊥,u,v))           : '⊢'q

q @ [u:'⊢''¬'p]
    n_ant_so_imp        := ➡_in(p,q,[x:'⊢'p]¬_el(p,q,u,x)) : '⊢'(p'➡'q)     13

q @ [u:'⊢'q]
    cons_so_imp         := ➡_in(p,q,[x:'⊢'p]u)              : '⊢'(p'➡'q)

p @ [u:'⊢''¬''¬'p]
    dbl_neg_law         := 'prim'                           : '⊢'p          14

q @ [u:'⊢'(p'➡'q)][v:'⊢''¬'q]
    mod_tol             := trans➡(p,q,⊥,u,v)                : '⊢''¬'p       15

u @ contrapos_1         := ➡_in('¬'q,'¬'p,[x:'⊢''¬'q]mod_tol(p,q,u,x))
                                                            : '⊢'('¬'q'➡''¬'p)
```

```
q @ [u:'⊢'('¬'p'⇒''¬'q)]
    contrapos_2                 := ⇒_in(q,p,[x:'⊢'q]
                                   dbl_neg_law(p,¬_in('¬'p,[y:'⊢''¬'p]
                                   ⇒_el(q,⊥,⇒_el('¬'p,'¬'q,u,y),x))))    : '⊢'(q'⇒'p)

q @ [u:'⊢'p][v:'⊢''¬'q]
    ant_n_cons_so_n_imp         := ¬_in(p'⇒'q,[x:'⊢'(p'⇒'q)]
                                   ⇒_el(p,⊥,mod_tol(p,q,x,v),u)          : '⊢''¬'(p'⇒'

q @ [u:'⊢''¬'(p'⇒'q)]
    n_imp_so_ant                := dbl_neg_law(p,¬_in('¬'p,[x:'⊢''¬'p]
                                   ⇒_el(p'⇒'q,⊥,u,n_ant_so_imp(p,q,x)))) : '⊢'p

    n_imp_so_cons               := ¬_in(q,[x:'⊢'q]
                                    ⇒_el(p'⇒'q,⊥,u,cons_so_imp(p,q,x)))  : '⊢''¬'q

q @ p'∨'q                       := '¬'p'⇒'q                               : prop

[u:'⊢'p]
    ∨_in_1                      := ⇒_in('¬'p,q,[x:'⊢''¬'p]¬_el(p,q,x,u)) : '⊢'(p'∨'q)

[u:'⊢'q]
    ∨_in_2                      := cons_so_imp('¬'p,q,u)                  : '⊢'(p'∨'q)

r @ [u:'⊢'(p'∨'q)][v:[x:'⊢'p]'⊢'r][w:[x:'⊢'q]'⊢'r]
    ∨_el                        := dbl_neg_law(r,¬_in('¬'r,[x:'⊢''¬'r]
                                   ⇒_el(r,⊥,x ,
                                   <⇒_el('¬'p,q,u,¬_in(p,
                                   [y:'⊢'p]⇒_el(r,⊥,x,<y>v))>w)))        : '⊢'r

    proof_by_cases              := ∨_el(p,q,r,u,v,w)                      : '⊢'r
```

```
q @ [u:'⊢'(p'∨'q)]

    com_∨                := ∨_el(p,q,q'∨'p,u,[x:'⊢'p]∨_in_2(q,p,x),
                            [x:'⊢'q]∨_in_1(q,p,x))          : '⊢'(q'∨'p)

q @ [u:[x:'⊢''¬'p]'⊢'q]

    ∨_in_3               := ➡_in('¬'p,q,u)                 : '⊢'(p'∨'q)

q @ [u:[x:'⊢''¬'q]'⊢'p]

    ∨_in_4               := com_∨(q,p,∨_in_3(q,p,u))       : '⊢'(p'∨'q)

q @ [u:'⊢'(p'∨'q)][v:'⊢''¬'p]

    not_case_1           := ➡_el('¬'p,q,u,v)               : '⊢'q

u @ [v:'⊢''¬'q]

    not_case_2           := not_case_1(q,p,com_∨(p,q,u),v) : '⊢'p
```

Commentaar

1. "prop" is het type van de proposities.

2. We voeren voor toekomstig gebruik drie proposities p,q en r in.

3. Als p een propositie is kunnen we zeggen: "p".

We bedoelen dan: "ik beweer p",

of misschien: "ik heb een bewijs voor p".

Deze taaldaad heet de assertie van p.

In sommige formele systemen wordt hij weergegeven door "⊢p".

In AUTOMATH wordt de assertie weergegeven door bewijzen van proposities als objecten in te voeren:

"proofs as objects".

(Het zou wel anders kunnen maar dit is de "natuurlijke" manier.)

We hebben voor iedere propositie p het type van de bewijzen van p:

"propositions as types".

We geven dit type weer door '⊢'p.

"u:'⊢'p" wordt geïnterpreteerd als { "u is een bewijs voor p" / "u bewijst p".

We hebben nu drie mogelijke manieren om bewijzen te introduceren:

i) als variabele in een context:

...[u:'⊢'p] interpretatie: { "laat u een bewijs zijn voor p" / "onderstel p",

ii) als primitief object:

γ @ ax := 'prim' : '⊢'p interpretatie: { "ax is een primitief bewijs voor p" / "p is een axioma",

iii) als afkortingsconstante:

γ @ st := A : '⊢'p (A is een expressie) interpretatie: {"st is het bewijs van p zoals gegeven in de expressie A" "p is een stelling (met bewijs A)".

4. Zij α een type.

Een predicaat over α is een functie die aan elke x van type α een propositie toevoegt, dus een functie met type [x:α]prop.

We definiëren dus pred := [x:α]prop.

Als nu P : pred(α), dan is P een predicaat over α.

Merk op dat, voor a : α, de propositie <a>P zegt dat P geldt in a.

5. Hoe komen we aan proposities?

Het blijkt dat hiervoor fundamenteel is de

universele quantor.

Deze wordt als primitieve propositie ingevoerd.

6. Hoe leggen we de betekenis van deze propositie vast?

We doen dit door twee gebruiksregels:

i) een regel die zegt hoe we kunnen *concluderen* dat de propositie geldt:

introductieregel,

ii) een regel die zegt hoe we kunnen *gebruiken* dat de propositie geldt:

eliminatieregel.

ad i) Hoe vind ik een bewijs voor ∀(α,P)?

" " " " " in '⊢'∀(α,P)?

Intuïtie: als ik voor iedere x : α een bewijs heb voor <x>P;

" " " " " een object " in '⊢'<x>P;

als ik een functie heb in $[x:\alpha]'\vdash'\langle x\rangle P$.
cartesisch produkt!

Deze intuïtie leggen we vast in een (primitieve) introductieregel.

7. ad ii) Hoe gebruik ik dat ik weet dat $\forall(\alpha,P)$ geldt?

Intuïtie: Als ik een object $a : \alpha$ heb en ik weet $\forall(\alpha,P)$, dan zal gelden $\langle a\rangle P$.

Deze intuïtie leggen we vast in een (primitieve) eliminatieregel.

De regel is in de logica bekend als specialisatie.

Opmerkingen. 1) De regels zijn in zekere zijn elkaars inverse.
Ze zeggen dat functies $u:[x:\alpha]'\vdash'\langle x\rangle P$
corresponderen met objecten $v:'\vdash'\forall(\alpha,P)$.

2) Het logische systeem dat op een dergelijke manier aan de logische operatoren koppelt hun introductie- en eliminatieregels, heet NATUURLIJKE DEDUCTIE.

8. Implicatie

Laat p en q proposities zijn.

We beschouwen nu een predicaat over het type $'\vdash'p$, dus het type van de bewijzen van p, en wel het predicaat

$$[x:'\vdash'p]q,$$

dat aan ieder bewijs van p de propositie q toevoegt.

Dit is dus, in zekere zin, een "constant predicaat", immers in q komt geen x voor.

Wat betekent nu $\forall('\vdash'p,[x:'\vdash'p]q)$?

Dat betekent dat voor ieder bewijs x van p de propositie q geldt.

Ofwel: als p geldt, dan geldt q ook: $p \Rightarrow q$.

Dit wordt onze definitie van $p'\Rightarrow'q$.

9. Met deze definitie kunnen we nu introductie- en eliminatieregels voor p'⇒'q afleiden.

introductie: We kunnen p'⇒'q afleiden,

als we een functie hebben van bewijzen van p naar bewijzen van q;

" " " " " van type [x:'⊢'p]'⊢'q.

Er geldt:
$$\begin{cases} \text{'⊢'p : 'type'} \\ \text{[x:'⊢'p]q : [x:'⊢'p]prop} \equiv_D \text{pred('⊢'p)} \\ \text{u : [x:'⊢'p]'⊢'q} \equiv_D \text{[x:'⊢'p]'⊢'<x>[y:'⊢'p]q.} \end{cases}$$

Hieruit volgt: ⇒_in(p,q,u) : '⊢'∀('⊢'p,[x:'⊢'p]q) $\equiv_D$ '⊢'(p'⇒'q).

eliminatie: Dit is de bekende regel modus ponens.

Ga na dat de afleiding klopt.

10. We kunnen nu ook bewijzen dat geldt p ⇒ p. (reflexiviteit van ⇒)

We onderstellen dat p ⇒ q en q ⇒ r

en bewijzen dan p ⇒ r. (transitiviteit van ⇒)

11. Contradictie

We definiëren contradictie ⊥ als "alle proposities gelden".

Dit geeft ons de regel:

"uit de contradictie volgt p"

"ex falso sequitur quodlibet",

want dit was immers onze definitie van falsum!

12. Negatie

We definiëren: '¬'p := p'⇒'⊥.

Dit geeft ons weer een introductieregel:

als we een functie hebben van type [x:'⊢'p]'⊢'⊥,

dan kunnen we besluiten tot '¬'p,

en een eliminatieregel:

als geldt p en ook '¬'p, dan geldt een willekeurige propositie q.

13. We leiden nu af:

onderstel ¬p , dan geldt p ➡ q;

" q , dan geldt p ➡ q.

14. We voegen een nieuw axioma toe:

onderstel ¬¬p , dan geldt (per axioma) p.

Opmerking. Zonder deze double negation law hebben we een systeem voor intuïtionistische logica.

15. enige stellingen over ➡ en ¬:

modus tollens: onderstel p ➡ q en ¬q, dan geldt ¬p;

contrapositie: { " p ➡ q , dan geldt ¬q ➡ ¬p
{ " ¬p ➡ ¬q , dan geldt q ➡ p

(deze laatste gebruikt de double negation law!).

Verder : onderstel p en ¬q, dan geldt ¬(p ➡ q).

Omgekeerd : onderstel ¬(p ➡ q), dan geldt p en ook ¬q.

16. We definiëren nu

disjunctie:

pvq := ¬p ➡ q.

We leiden af:

twee introductieregels: 1) onderstel p, dan geldt pvq

2) " q, dan geldt pvq,

en een <u>eliminatieregel</u>, het bewijs door gevalonderscheiding:

onderstel $p \vee q$
uit p kan ik r afleiden
" q " " r "
} dan geldt r.

In ons formularium: u : '⊢'p'∨'q

v : [x:'⊢'p]'⊢'r (v is een functie van bewijzen van p naar bewijzen van r)

w : [x:'⊢'q]'⊢'r

<u>dan</u> ∨_el(p,q,r,u,v,w) : '⊢'r.