Vérification des textes mathématiques par un ordinateur

Notitie Nr. 18 (1969-1970) 14 november 1969
Technological University Eindhoven
Department of Mathematics

AUT18

Vérification des textes mathématiques par un ordinateur.

Conférence à Lille, le 21 novembre 1969,

par N.G. de Bruijn (Eindhoven).

Le problème de vérification des textes mathématiques est au fond le problème de définir un langage. Il faut que ce langage satisfait aux propriétés suivantes:

1. Tout ce qu'est correct en mathématiques peut être exprimé avec ce langage, conform à sa grammaire, et tout ce qu'on peut dire au moyen ce langage est correct en mathématiques.

2. On admet que les textes de mathématiques ordinaires s'allongent par là traduction, mais on n'admet pas que la proportion de l'allongement tend vers l'infini à mesure qu'on fait des progrès.

3. On dispose d'un programme avec lequel un ordinateur peut trancher si un livre donné est correct ou non.

La tâche no. 1 est vraiment impossible. En premier lieu, il semble impossible de préciser ce qu'il veut dire "mathématiques". Pourtant, on peut essayer de circonscrire une partie de mathématiques, assez large pour la pratique quotidienne. Il semble que le langage AUTOMATH, développé à Eindhoven depuis 1967, satisfait à cette condition, ainsi qu'aux conditions 2 et 3.

Dans cette conférence-ci, nous considérons principalement la définition du langage. A fin de mieux établir les difficultés principales, nous allons discuter d'abord deux langages plus simples:premièrement SEMIPAL, où on a déjà la structure de bloc et le système d'abréviation, et ensuite PAL, où on a, en outre, l'administration de catégories. On arrive à AUTOMATH en ajoutant un système de quantification en forme d'un calcul lambda généralisé.

Première phase: le langage SEMIPAL.

SEMIPAL est un système pour exprimer des notions en termes d'autres, en utilisant la structure de blocs imbriqués. Les expressions en SEMIPAL ont une structure donnée par

<expression> ::= <identificateur> | <identificateur>(<liste d'expressions>)

<liste d'expressions> ::= <expression> | <liste d'expressions>,<expression>

Un livre en SEMIPAL consiste d'un liste de lignes. Les lignes ont la forme

<ligne> ::= <identificateur> := <explication>

ou

<explication> ::= —— | PN | <expression>

Les identificateurs des lignes diverses sont tous différents; si l'on veut, on peut choisir le numéro d'une ligne comme son identificateur.

La liste des lignes est structurée en blocs qui peuvent s'imbriquer les uns dans les autres, sans se chevaucher.

A la tête d'un bloc on trouve une ligne de la forme

<identificateur> := ——

En ce cas-ci, l'identificateur s'appelle variable.

L'indicateur d'une ligne est le variable à la tête du plus petit bloc contenant cette ligne, sauf pour les têtes de bloc elles-mêmes, où l'indicateur est la tête du plus petit bloc un seul excepté. Si une ligne ne se trouve dans aucun bloc, l'indicateur est le symbole 0.

La liste d'indicateurs d'une ligne est la liste des variables de têtes de blocs contenant cette ligne, dans l'ordre imposé par le livre. Encore, quand la ligne est à la tête d'un bloc, son identificateur n'appartient pas à cette liste.

Quand une expression figure dans une ligne, elle ne peut pas contenir des variables qui n'appartiennent pas à la liste d'identificateurs. D'autre part, les identificateurs non variables peuvent être utilisés partout, pourvu qu'ils sont munis d'une liste de sous-expressions dont la longueur satisfait à une certaine condition. Pour l'expression $b(\Sigma_1, \ldots, \Sigma_\ell)$ utilisée dans la ligne λ, cette condition s'exprime comme suit: Si $x_1, \ldots, x_r$ est la liste

d'indicateurs de b, nous convenons que $\ell \leq r$ et que la ligne λ se trouve dans le bloc de $x_{r-\ell}$. Dans ce cas on peut considérer $b(\Sigma_1, \ldots, \Sigma_\ell)$ et $b(x_1, \ldots, x_{r-\ell}, \Sigma_1, \ldots, \Sigma_\ell)$ comme identiques. On a des modifications triviales quand $r = 0$ ou $r = \ell$.

Exemple. Nous prenons le texte qui s'ecrit normalement comme

non (x) := notion primitive
et (x,y) := notion primitive
ou (x,y) := non (et(non(x), non(y))
impl (x,y) := ou (non(x), y)
equiv (x,y) := et (impl(x,y), impl (y,x))
outroi (x,y,z):= ou (ou(x,y), z)
tralala (x,y) := outroi (et(x,y), non(impl(x,y)), non(impl(y,x)))

Ce text se lit en SEMIPAL comme suit:

```
0 |       x := ————————
x |     non := PN
x |  |    y := ————————
y |  |   et := PN
y |  |   ou := non (et(non, non(y)))
y |  | impl := ou (non,y)
y |  | equiv := et (impl, impl(y,x))
y |  |    z := ————————
z |  |outroi := ou (ou, z)
y |  |tralala := outroi (et, non(impl), non(impl(y,x)))
```

Dans la première colonne on trouve les indicateurs.

Remarque. Sans changer l'effet, on peut remplacer, par exemple, la définition de "equiv" par

equiv := et (impl(y), impl(y,x)) ,

ou: equiv := et (impl(x,y), impl(y,x)).

Deuxième phase. Le langage PAL.

En PAL chaque expression a une catégorie, sauf les expressions qui sont des catégories elles-mèmes. Dans le cas dernier, on écrit au lieu de la catégorie le symbole type. On a le droit d'introduire des catégories nouvelles, soit comme des variables, soit comme des notions primitives, soit comme des expressions. Il y a quelques restrictions que nous donnons en forme abrégée:

1. Une tête de bloc doit avoir une catégorie qui est valable dans le bloc où cette tête de bloc est insérée.

2. La catégorie d'une notion primitive doit être une catégorie valable au même lieu.

3. Quand on écrit des expressions comme $b(\Sigma_1, \ldots, \Sigma_\ell)$, on demande (quand $x_1, \ldots, x_r$ est la liste d'indicateurs de b) que Σ_1 a la catégorie de $x_{r-\ell+1}$, que Σ_2 a la catégorie de $x_{r-\ell+2}$ avec substitution de Σ_1 pour $x_{r-\ell}$, etc.

Exemple.

```
                réel := PN        type
                nat  := PN        type
0               a    := ———       réel
a               n    := ———       nat
                ..........
                ..........
a               pot  := ———       réel
0               x    := ———       type
x               y    := ———       type
y               produit cartésien := PN    type
0               u    := ———       produit cartésien (real, nat)
u               ..........
```

On peut écrire des assertions en PAL, et par suite, on peut donner certaines démonstrations. Pour ça on convient qu'uneassertion correspond à une catégorie. Alors, faire l'assertion veut dire: exprimer qu'on a une expression avec cette catégorie. Par exemple si l'on veut introduire égalité de nombres réels comme notion primitive, on écrit

```
0 | x    :=  ————————    réel
x |  y   :=  ————————    réel
y |  EGAL := PN          type
```

Par suite, quand on a les reéls p et q , on peut écrire l'hypothèse p = q comme

```
0 | u    :=  ————————    EGAL (p,q)
          ........
```

D'ailleurs, on peut écrire l'axiome p = q comme

```
v    := PN          EGAL (p,q),
```

et le théorème p = q comme

```
w    := ...         EGAL (p,q),
```

où ... est une expression qui peut être considérée comme la démonstration du théorème. Cette expression contient, soit implicitement, soit explicitement, toute information nécessaire pour comprendre la démonstration. Le théorème n'est pas énoncé avant que la démonstration soit terminée.

Si l'on veut, on peut considérer les propositions comme objets mathématiques:

```
bool := PN          type
```

et on peut postuler que chaque proposition b a son catégorie d'affirmation:

```
0 | b    :=  ————————    bool
b | TRUE := PN          type
```

Alors, si c est une proposition quelconque, on peut affirmer c (soit comme hypothèse, soit comme axiome, soit comme théorème) par une ligne

```
...  := ...          TRUE (c).
```

C'est ainsi qu'en PAL on peut décrire la structure de blocs en mathématiques où les têtes de blocs sont des variables aussi bien que des hypothèses.

Exemple. Nous introduirons la notion de conjunction de deux propositions comme notion primitive et nous exprimerons quelques axiomes et un théorème; après ça nous indiquerons comment on peut appliquer ce résultat dans une situation donnée.

0	bool	:=	PN	type
0	|b	:=	—	bool
b	|TRUE	:=	PN	type
0	|x	:=	—	bool
x	||y	:=	—	bool
y	||and	:=	PN	bool
y	|||asp 1	:=	—	TRUE(x)
asp 1	||||asp 2	:=	—	TRUE(y)
asp 2	||||ax 1	:=	PN	TRUE(and)
y	|||asp 3	:=	—	TRUE(and)
asp 3	|||ax 2	:=	PN	TRUE(x)
asp 3	|||ax 3	:=	PN	TRUE(y)
asp 3	|||theorem	:=	ax 1(y,x,ax 3, ax 2)	TRUE(and(y,x))

Maintenant, supposons que les lignes suivantes sont disponibles:

0	u	:=	....	bool
0	v	:=	....	bool
0	w	:=	....	TRUE(and(u,v))

alors nous pouvons écrire

0	z	:=	theorem(u,v,w)	TRUE(and(v,u))

3. Troisième phase: le langage AUTOMATH.

AUTOMATH est plus général que PAL; chaque texte en PAL est aussi valable en AUTOMATH. La différence entre les deux est qu'en AUTOMATH on se sert du calcul lambda de Church, ou plutôt d'une extension de ce calcul.

Si ξ est une catégorie quelconque, une application qui donne, à tout x du type ξ, une valeur f(x) du type $\eta(x)$, est écrite comme

$$[x,\xi]\ f(x),$$

et on dit que son type est

$$[x,\xi]\ \eta(x).$$

Une telle application peut être introduite comme une variable:

p := _____ $[x,\xi]\ \eta(x)$

ou comme notion primitive

q := PN $[x,\xi]\ \eta(x)$.

En outre, si l'expression Σ a le type $[x, \xi]\ \eta(x)$, et si l'expression Λ a le type ξ, on a le droit de former

$\{\Lambda\}\ \Sigma$

Du reste, si Σ a la forme $[x,\xi]\ \Gamma$ (où Γ est une expression qui contient x), il est entendu que $\{\Lambda\}\ \Sigma$ peut être remplacé par l'expression obtenue par substitution de Λ pour x dans Γ.

Exemple. f := [x,ξ] plus (x,x)		[x,ξ] réel
a := ...		réel
b := {a}f		réel

Ainsi b et plus (a,a) sont interchangeables (égal par définition).

Les règles d'operation pour [ ] et { } sont intuitivement claires. Pourtant, cette matière n'est pas simple. En particulier, la substitution peut être assez compliquée à cause des variables cachées.

Dans les pages suivantes on trouve un exemple d'un texte un peu plus long, écrit en AUTOMATH. On verra la possibilité de donner des définitions assez compliquées et partagées en étappes.

Sans doute, le lecteur remarquera que ce texte est difficile à écrire et difficile à lire. C'est vrai, mais - dans le sens de langages de programmation - on peut le comparer avec un langage de machine, et nous espérons qu'on reusisse à construire des langages artificiels plus simples, lesquels peuvent être écrits et lus plus facilement, et lesquels peuvent être traduits par un ordinateur en notre "langage de machine".

1. Quelques fondaments de mathématiques qui suffisent pour le théorème sur limites.

1.1	0	bool	:=	PN	type
1.2	0	|b1	:=	———	bool
1.3	b1	|TRUE	:=	PN	type
1.4	0	|ksi	:=	———	type
1.5	ksi	|nonempty	:=	PN	bool
1.6	ksi	||iks	:=	———	ksi
1.7	iks	||then 2	:=	PN	TRUE(nonempty)
1.8	ksi	||P	:=	———	[u,ksi] bool
1.9	P	||exists	:=	PN	bool
1.10	P	|||v	:=	———	ksi
1.11	v	||||ass 1	:=	———	TRUE({v} P)
1.12	ass 1	||||then 13*	:=	PN	TRUE(exists)
1.13	P	||ALL	:=	[u,ksi] TRUE({u} P)	type
1.14	P	||all	:=	nonempty(ALL)	bool
1.15	0	|b2	:=	———	bool
1.16	b2	||b3	:=	———	bool
1.17	b3	||IMPL	:=	[u,TRUE(b2)] TRUE(b3)	type
1.18	b3	||impl	:=	nonempty(IMPL)	bool

2. Définition de limite d'une suite.

2.1	0	real	:=	PN	type
2.2	0	|r1	:=	———	real
2.3	r1	||r2	:=	———	real
2.4	r2	||distance	:=	PN	real
2.5	r2	||less	:=	PN	bool
2.6	0	null	:=	PN	real
2.7	0	nat	:=	PN	type
2.8	0	sequence	:=	[x,nat]real	type
2.9	0	1	:=	PN	nat
2.10	0	|k1	:=	———	nat
2.11	k1	||k2	:=	———	nat
2.12	k2	||lessnat	:=	PN	bool
2.13	0	|a	:=	———	sequence
2.14	a	||l	:=	———	real
2.15	l	|||eps	:=	———	real
2.16	eps	||||m_o	:=	———	nat
2.17	m_o	||||m	:=	———	nat
2.18	m	||||w	:=	impl(lessnat(m_o,m), less(distance({m}a,l),eps))	bool
2.19	m_o	||||z	:=	all(nat,[s,nat] w(s))	bool
2.20	eps	|||y	:=	exists(nat,[t,nat] z(t))	bool
2.21	eps	|||q	:=	impl(less(null,eps),y)	bool
2.22	l	|||lim	:=	all(real,[t,real]q(t))	bool
2.23	a	|conv	:=	exists(real,[t,real] lim(t))	bool

3. Un lemme introduit comme axiome.

3.1	0	x1	:=	_____	real
3.2	x1	y1	:=	_____	real
3.3	y1	if	:=	_____	TRUE(less(null,y1))
3.4	if	lemma	:=	PN	TRUE(less(distance(x1,x1),y1))

4. Chaque suite constante est convergente.

4.1	0	c	:=	_____	real
4.2	c	p	:=	[x,nat] c	sequence
4.3	c	delta	:=	_____	real
4.4	delta	abbrev 1	:=	TRUE(less(null,delta))	type
4.5	delta	ass	:=	_____	abbrev 1
4.6	ass	n_o	:=	_____	nat
4.7	n_o	n	:=	_____	nat
4.8	n	abbrev 2	:=	TRUE(less(distance({n}p,c),delta))	type
4.9	n	then	:=	lemma({n}p,delta,ass)	abbrev 2
4.10	n	a	:=	lessnat(n_o,n)	bool
4.11	n	aa	:=	w(p,c,delta,n_o,n)	bool
4.12	n	b	:=	then 2([u,TRUE(a)] abbrev 2,[u,TRUE(a)]then)	TRUE(aa)
4.13	n_o	h	:=	then 2([s,nat] TRUE(aa(s)), [s,nat] b(s))	TRUE(z(p,c,delta,n_o))
4.14	ass	d	:=	then 13*(nat,[x,nat]z(p,c,delta,1),1,h(1))	TRUE(y(p,c,delta))
4.15	delta	e	:=	then 2([x,abbrev 1]TRUE(y(p,c,delta)), [x,abbrev 1]d(x))	TRUE(q(p,c,delta))
4.16	c	f	:=	then 2([s,real]TRUE(q(p,c,s)),[s,real]e(s))	TRUE(lim(p,c))
4.17	c	g	:=	then 13*(real,[s,real]lim(p,s),c,f)	TRUE(conv(p))

Références:

1. N.G. de Bruijn : AUTOMATH, a language for mathematics. Report 68-WSK-05(1968) Technological University Eindhoven, The Netherlands.

2. N.G. de Bruijn : The mathematical language AUTOMATH, its usage, and some of its extensions. To be published in the Proceedings of the Symposium on Automatic Demonstration (IRIA, Versailles, December 1968). Springer Lecture Notes Series.