MATHEMATICAL PROBLEMS IN COMPUTATION THEORY
BANACH CENTER PUBLICATIONS, VOLUME 21
PWN – POLISH SCIENTIFIC PUBLISHERS
WARSAW 1988

ON SOME ARITHMETICALLY EXPRESSIBLE PROPERTIES OF PROGRAMS

M. E. SZABO*

Department of Mathematics, Concordia University, Montreal, Canada

0. Introduction

The purpose of this paper is to show that the method of diagrams of types in [4] and [5] can be used to express familiar properties of parallel programs in Peano arithmetic (PA). By an expression of a property we mean a formula of PA which holds in the standard model N of PA precisely when a given program has the specified property. Although non-temporal, the underlying semantics of programs is similar to the branching-time interpretation of parallel programs of [3] in which the meaning of a program is identified with the totality of computation trees over a fixed data base generated by the given program. In the present context, this data base is the standard set of natural numbers. We show that the properties of programs discussed in [2] and [3] can be described effectively as first order properties of computation trees. By Church's thesis and by virtue of the representability of recursive functions and relations in PA, all extraneous function and relation symbols used can be eliminated from the resulting formulas in favour of genuine formulas of PA expressing the same properties. One of the key elements of such descriptions is the inclusion, at each node of a computation tree, of a numeric analogue of the current instruction of a given program.

1. Programs

Programs are constructed inductively from the atomic assignment statements $[x/v]$ ($v = xi$, $xi+1$, or $xi-1$), the **null** statement, and from quantifier-free

* This paper was written while the author was a guest of the International Mathematical Banach Center and was stimulated by discussions with Professor Helena Rasiowa. The present research is supported in part by the Natural Sciences and Engineering Research Council of Canada and by the Fonds F.C.A.C. pour l'aide et le soutien à la recherche du Québec.

formulas $C(v1, \ldots, vn)$ of PA using the programming constructs **compose**$(P1, \ldots, Pn)$, **if** $(C, P1, P2)$, **while**$(C, P1)$, **parallel**$(P1, \ldots, Pn)$, and **await**$(C, P1)$ in the usual way. The set of programs is based on the fixed infinite list $x1, x2, \ldots$ of program variables.

The construction of programs is subject to the following syntactic restrictions: The variables $v1, \ldots, vn$ in C are assumed to be among the program variables of the programs in which the condition C occurs, the statements $P1, \ldots, Pn$ of a **compose** statement must not be **await** or **compose** statements, the statements $P1$ and $P2$ in an **if** statement must not be **await** statements, the statement $P1$ in a **while** statement must not be a **null**, **await**, or **while** statement, and the statements $P1, \ldots, Pn$ in a **parallel** statement must not themselves be **parallel** statements.

2. Execution schemes

Each program determines a finite number of labelled "subprograms" which correspond to the current instructions in the course of a computation of a program. We refer to these "subprograms" as **execution schemes.** Their definition requires the introduction of a numerical type structure on the occurrences of the subprograms of a given program. For this purpose we think of a program as represented by its construction tree, with the occurrences of its atomic statements numbered consecutively from left to right, and all occurrences of **null** labelled with type 10, and assume as given a fixed recursive Gödel numbering G: $N^* \to N$ of al finite sequences of natural numbers. Based on these data, we introduce the following system of *program types*:

2.1. $t(\textbf{null}) = 10;$
2.2. $t(\mathrm{occ}(i, [x/x])) = 10G(i, x)+1;$
2.3. $t(\mathrm{occ}(i, [x/x-1])) = 10G(i, x)+2;$
2.4. $t(\mathrm{occ}(i, [x/x+1])) = 10G(i, x)+3;$
2.5. $t(\textbf{while}(C, P1)) = 10t(P1)+4;$
2.6. $t(\textbf{await}(C, P1)) = 10t(P1)+5;$
2.7. $t(\textbf{if}(C, P1, P2)) = 10G(G(t(P1), t(P2)))+6;$
2.8. $t(\textbf{compose}(P1, \ldots, Pn)) = 10G(t(P1), \ldots, t(Pn))+7;$
2.9. $t(\textbf{parallel}(P1, \ldots, Pn)) = 10G(t(P1), \ldots, t(Pn))+8;$
2.10. $t(t1, \ldots, tn: E) = 10G(G(t1, \ldots, tn), t(E))+9,$

where the variable $x = xi$ in 2.2–2.4 is coded by its index i, $\mathrm{occ}(i, [x/x])$ stands for the ith occurrence of $[x/x]$ in a given program, and $t1, \ldots, tn: E$ is a pair consisting of a finite sequence $t1, \ldots, tn$ of types of **compose** and **while** statements, together with a program statement E.

We define the **execution schemes** of a program as follows:

2.11. The execution scheme of **null** is **null**;

2.12. The execution schemes of t: **null** are t: **null** and, if $t = 10G(t1, \ldots, u, 10)+7$ and $u = t(S)$, the execution schemes of S;

2.13. The execution schemes of $[x/v]$ are $[x/v]$ and **null**;

2.14. The execution schemes of t: $[x/v]$ are t: $[x/v]$ and the execution schemes of t: **null**;

2.15. The execution schemes of **compose**($P1$, $P2$) are **compose**($P1$, $P2$), t: $P2$, where $t = t$(**compose**($P1$, **null**)), and the execution schemes of t: $P2$ and of $P1$;

2.16. The execution schemes of u: **compose**($P1$, $P2$) are u: **compose** ($P1$, $P2$), ut: $P2$, where $t = t$(**compose**($P1$, **null**)), and the execution schemes of ut: $P2$ and of u: $P1$;

2.17. The execution schemes of **if**(C, $P1$, $P2$) are **if**(C, $P1$, $P2$) and the execution schemes of $P1$ and of $P2$;

2.18. The execution schemes of t: **if**(C, $P1$, $P2$) are t: **if**(C, $P1$, $P2$), t: $P1$ and t: $P2$ and the execution schemes of t: $P1$ and of t: $P2$;

2.19. The execution schemes of **while**(C, $P1$) are **while**(C, $P1$), t: $P1$, where $t = t$(**while**(C, $P1$)), the execution schemes of t: $P1$, and **null**;

2.20. The execution schemes of u: **while**(C, $P1$) are u: **while**(C, $P1$), ut: $P1$, where $t = t$(**while**(C, $P1$)), the execution schemes of ut: $P1$, and u: **null**;

2.21. The execution schemes of **await**(C, $P1$) are **await**(C, $P1$) and the execution schemes of $P1$;

2.22. The execution schemes of **parallel**($P1$, $P2$) are all schemes of the form **parallel**(Q, R), where Q is an execution scheme of $P1$ and R is an execution scheme of $P2$;

2.23. The execution schemes of t: **parallel**($P1$, $P2$) are all schemes of the form t: **parallel**(Q, R), where Q is an execution scheme of $P1$ and R is an execution scheme of $P2$;

2.24. The execution schemes of $t1, \ldots, t(n-1)$, tn: P are all schemes of the form $t1, \ldots, t(n-1)$: Q, where Q is an execution scheme of tn: P.

This definition extends in the obvious way to **compose** and **parallel** statements with more than two arguments. An induction on programs shows that every program determines only finite many execution schemes.

3. Cancellation trees

We structure the possible orders of execution of a parallel program P with initial input a in the form of a rooted tree $\mathrm{Tr}(P, a)$ whose nodes contain both a program and a data component describing the currently active

components of P and the values computed up to the given point in a computation. Since the program component of a node is obtained by a "cancellation" procedure from the execution schemes determined by P, we refer to $\mathrm{Tr}(P, a)$ as a *cancellation tree*. The root of $\mathrm{Tr}(P, a)$ is the pair (P, a) and every other node is computed from an earlier node by the *next-node "function"* σ. A related operational semantics for parallel programs may be found in [1].

The nodes of the tree $\mathrm{Tr}(P, a)$ are defined inductively from the root (P, a) as follows:

3.1. $\sigma(\mathbf{null}, b)$ is undefined,
$\sigma(t\colon \mathbf{null}, b) = (S, b)$,
where $u = t(S)$ and $t = 10u+4$ or $10G(u, 10)+7$;

3.2. $\sigma([x/x], b) = (\mathbf{null}, b)$,
$\sigma(t\colon [x/x], b) = (t\colon \mathbf{null}, b)$;

3.3. $\sigma([xi/xi+1], (\ldots, xi, \ldots)) = \mathbf{null}, (\ldots, xi+1, \ldots))$,
$\sigma(t\colon [xi/xi+1], (\ldots, xi, \ldots)) = (t\colon \mathbf{null}, (\ldots, xi+1, \ldots))$;

3.4. $\sigma([xi/xi-1], (\ldots, xi, \ldots)) = (\mathbf{null}, (\ldots, xi-1, \ldots))$,
$\sigma(t\colon [xi/xi-1], (\ldots, xi, \ldots)) = (t\colon \mathbf{null}, (\ldots, xi-1, \ldots))$;

3.5. $\sigma(\mathbf{compose}(P1, P2), b) = (t\colon P2, b)$, where $t = t(\mathbf{compose}(P1, \mathbf{null}))$,
$\sigma(\mathbf{compose}(P1, \mathbf{null}), b) = (P1, b)$,
$\sigma(u\colon \mathbf{compose}(P1, P2), b) = (ut\colon P2, b)$,
where $t = t(\mathbf{compose}(P1, \mathbf{null}))$,
$\sigma(u\colon \mathbf{compose}(P1, \mathbf{null}), b) = (u\colon P1, b)$;

3.6. $\sigma(\mathbf{while}(C, P1), b) = (t\colon P1, b)$ if $C[b]$ is true,
$= (\mathbf{null}, b)$ if $C[b]$ is false,
where $t = t(\mathbf{while}(C, P1))$,
$\sigma(u\colon \mathbf{while}(C, P1), b) = (ut\colon P1, b)$ if $C[b]$ is true,
$= (u\colon \mathbf{null}, b)$ if $C[b]$ is false,
where $t = t(\mathbf{while}(C, P1))$;

3.7. $\sigma(\mathbf{if}(C, P1, P2), b) = (P1, b)$ if $C[b]$ is true,
$= (P2, b)$ if $C[b]$ is false,
$\sigma(u\colon \mathbf{if}(C, P1, P2), b) = (u\colon P1, b)$ if $C[b]$ is true,
$= (u\colon P2, b)$ if $C[b]$ is false;

3.8. $\sigma(P1//P2, b) = (\sigma 1(P1//P2, b), \sigma 2(P1//P2, b))$,
$\sigma(P1//(P2//P3), b) = \sigma(P1//P2//P3, b)$,
$\sigma((P1//P2)//P3, b) = \sigma(P1//P2//P3, b)$,
$\sigma(t\colon (P1//P2), b) = (\sigma 1(t\colon (P1//P2), b), (\sigma 2(t\colon (P1//P2), b)))$,
$\sigma(t\colon (P1//(P2//P3)), b) = \sigma(t\colon (P1//P2//P3), b)$,
$\sigma(t\colon ((P1//P2)//P3), b) = \sigma(t\colon (P1//P2//P3), b)$,
$\sigma(t\colon (\mathbf{null}//\mathbf{null}), b) = (S, b)$,

where $u = t(S)$ and $t = 10u+4$ or $10G(u, 10)+7$;

3.9. $\sigma 1(\mathbf{null}//P, b) =$ undefined.

If $P = \mathbf{await}(C, P1)$, $u = t(S)$, and $t = 10u+4$ or $10G(u, 10)+7$, then

$\sigma 1(t\colon \mathbf{null}//P, b) = (S//P, b)$ if $C[b]$ is false,
$= (S//P1, b)$ if $C[b]$ is true;

if $P \neq \mathbf{await}(C, P1)$, $u = t(S)$, and $t = 10G(u, 10)+7$, then
$\sigma 1(t\colon \mathbf{null}//P, b) = (S//P, b)$;

if $P = \mathbf{null}$ or $t'\colon \mathbf{null}$, $u = t(S)$, and $t = 10u+4$, then
$\sigma 1(t\colon \mathbf{null}//P, b) = (S//P, b)$;

$\sigma 1(t\colon \mathbf{null}//P, b)$ is undefined otherwise;

3.10. $\sigma 1([x/v]//P, b) = (\mathbf{null}//P, b[x/v])$,
$\sigma 1((t\colon [x/v])//P, b) = ((t\colon \mathbf{null})//P, b[x/v])$;

3.11. $\sigma 1(\mathbf{compose}(P1, P2)//P, b) = ((t\colon P2)//P, b)$,
where $t = t(\mathbf{compose}(P1, \mathbf{null}))$,
$\sigma 1(\mathbf{compose}(P1, \mathbf{null})//P, b) = (P1//P, b)$,
$\sigma 1((u\colon \mathbf{compose}(P1, P2))//P, b) = ((ut\colon P2)//P, b)$,
where $t = t(\mathbf{compose}(P1, \mathbf{null}))$,
$\sigma 1((u\colon \mathbf{compose}(P1, \mathbf{null}))//P, b) = ((u\colon P1)//P, b)$;

3.12. $\sigma 1(\mathbf{if}(C, P1, P2)//P, b) = (P1//P, b)$ if $C[b]$ is true,
$= (P2//P, b)$ if $C[b]$ is false,
$\sigma 1((t\colon \mathbf{if}(C, P1, P2))//P, b) = ((t\colon P1)//P, b)$ if $C[b]$ is true,
$= ((t\colon P2)//P, b)$ if $C[b]$ is false;

3.13. $\sigma 1(\mathbf{while}(C, P1)//P, b) = ((t\colon P1)//P, b)$ if $C[b]$ is true,
$= (\mathbf{null}//P, b)$ if $C[b]$ is false,
where $t = t(\mathbf{while}(C, P1))$,
$\sigma 1((u\colon \mathbf{while}(C, P1))//P, b) = ((ut\colon P1)//P, b)$ if $C[b]$ is true,
$= ((u\colon \mathbf{null})//P, b)$ if $C[b]$ is false,
where $t = t(\mathbf{while}(C, P1))$;

3.14. $\sigma 1(\mathbf{await}(C, P1)//P, b) = (P1//P, b)$ if $C[b]$ is true,
$=$ undefined if $C[b]$ is false.

The definition of $\sigma 2$ is analogous and the cases of **compose** and **parallel** statements with more than two variables are obtained by an obvious induction. In clause (*h*) it is understood that $\sigma(P1//P2, b)$ may give rise to only a single node if $\sigma 1(P1//P2, b)$ or $\sigma 2(P1//P2, b)$ is undefined. The ordered pair notation is intended to convey the idea that the next nodes of $(P1//P2, b)$ are considered to be ordered from left to right, with $\sigma 1(P1//P2, b)$ to the left of $\sigma 2(P1//P2, b)$. Similarly for more than two next nodes.

4. Computation paths

In order to be able to describe the computational behaviour of a program P arithmetically, we Gödel-number the nodes (Q, b) of the cancellation trees

$\mathrm{Tr}(P, \boldsymbol{a})$ of P by defining $G(Q, \boldsymbol{b})$ to be $G(t(Q), G(\boldsymbol{b}))$. We then define a *computation path* $G(e) \in N$ to be the Gödel number of a finite sequence $e = (G(e1), \ldots, G(en))$ of Gödel numbers of nodes $e1, \ldots, en$ of a cancellation tree of P with the property that $e1 = (P, \boldsymbol{a})$ for some $\boldsymbol{a}$, and that if $e(i+1) \in e$, then $e(i) = (t(Q), G(\boldsymbol{b}))$, $e(i+1) = (t(Q'), G(\boldsymbol{b}'))$, and $(Q', \boldsymbol{b}')$ is a next node of $(Q, \boldsymbol{b})$ in $\mathrm{Tr}(P, \boldsymbol{a})$. To simplify the notation, we often write ei in place of $G(ei)$ and refer to $e = (e1, \ldots, en)$ as a path. We denote the set of all computation paths of P by $\mathrm{Paths}(P)$. It is clear from the algorithmic nature of the next-node function σ that $\mathrm{Paths}(P)$ is a decidable set of natural numbers and we introduce new "path" variables π and π' to range over $\mathrm{Paths}(P)$. It is immediate from the recursiveness of the divisibility relation that the property of being a subpath of a path is also a decidable property. We shall write $\pi' \subseteq \pi$ to express the fact that π' is a subpath of π. We define three special types of "path" functions $\tau = \tau(P)$, $\lambda = \lambda(P)$, and $\varrho = \varrho(P)$.

(a) $\tau(e) = en$ if $e \in \mathrm{Paths}(P)$,

$= (0, 0)$ if $e \notin \mathrm{Paths}(P)$,

(b) $\lambda(e) = \lambda(en)$ if $e \in \mathrm{Paths}(P)$,

$= 0$ if $e \notin \mathrm{Paths}(P)$,

(c) $\varrho(e) = \varrho(en)$ if $e \in \mathrm{Paths}(P)$,

$= 0$ if $e \notin \mathrm{Paths}(P)$,

where $(\lambda(i), \varrho(i)) = ei$. Strictly speaking, $\tau(e) = \tau(G(e))$ and $en = G(t(Q), G(\boldsymbol{b}))$. The functions $\tau(P)$, $\lambda(P)$ and $\varrho(P)$ are clearly effective.

In addition, we introduce a new type of relation symbol $\Pi = \Pi(P)$ with the property that $N \models \Pi(\pi)[e/\pi]$ if and only if $e \in \mathrm{Paths}(P)$. By Church's thesis and the representability of recursive functions and relations in PA we have the following result:

4.1. THEOREM. *Any formula constructed from the language of* PA *together with the relation symbol* $\Pi(P)$ *and the function symbols* $\tau(P)$, $\lambda(P)$, *and* $\varrho(P)$ *is equivalent to a formula of* PA. ∎

5. Diagrams of types

Next we use the types of the execution schemes of a program P to construct a finite diagram which depicts the possible changes of the "program states" during the execution of P. For this purpose we order the execution schemes determined by P as follows:

(a) $Si > Sj$ if Sj is an execution scheme of Si and $Si \neq Sj$.

(b) $Si \gg Sj$ if $Si > Sj$ and there is no execution scheme Sk such that $Si > Sk > Sj$.

5.1. Using the ordering $\gg$, we define the *typical tree* $T(P)$ as follows:

(a) The root of $T(P)$ is the type $t(P)$ of P.

(b) If $t(Si)$ is a node of $T(P)$ and $Si \gg Sj$, then $t(Sj)$ is a next node of $t(Si)$.

(c) $T(P)$ has no other nodes.

5.2. The *diagram* Diag(P) of P has a vertices the nodes of $T(P)$, and every pair $(t(Si), t(Sj))$ with the property that $t(Sj)$ is a next node of $t(Si)$ in $T(P)$ (by 3.1 to 3.14) determines an arrow $t(Si) \to t(Sj)$ in Diag(P). The remaining arrows of Diag(P) are determined by the execution schemes corresponding to **while** statements as described in 3.1, 3.8, and 3.9. Specifically, if $Sj =$ **while**(C, P) and $Si = t(Sj)$: **null**, we introduce an arrow from $t(Si)$ to $t(Sj)$. Similar arrows are introduced for **while** statements occurring inside **parallel** statements. Since the arithmetical formulas C in **if**, **while**, and **await** statements are lost in the determination of program types, we label the arrows of Diag(P) depending on the truth of C or $\neg C$ with those formulas. In particular, if $Si =$ **while**$(C, P1)$ and $Sj = t(Si)$: $P1$, we label the arrow from $t(Si)$ to $t(Sj)$ with C, if $Si =$ **if**$(C, P1, P2)$ and $Sj = P1$ and $Sk = P2$, we label the arrow from $t(Si)$ to $t(Sj)$ with C and the arrow from $t(Si)$ to $t(Sk)$ with $\neg C$, and if $Si =$ **parallel**(**await**$(C, P1)//P1$) and $Sj =$ **parallel**$(P1, P2)$, we label the arrow from $t(Si)$ to $t(Sj)$ with C, etc. We write $(i, j) \in$ **Diag**(P) if there is an arrow from $t(Si)$ to $t(Sj)$ in Diag(P).

6. Programs as arithmetical formulas

From the diagram of a program P we construct a formula $\Phi(P)$ equivalent to a formula of PA which describes both the program states and the relationships between the corresponding intermediate values in any computation of P. The basic construction is taken from [4] and [5]. We require three types of new variables for the definition of $\Phi(P)$. For each program variable xi of P, we introduce a new "input" variable ri and a new "output" variable si. We also introduce two new "control" variables z and z' ranging over the vertices of Diag(P). For each arrow $t(Si) \to t(Sj)$ of Diag(P) we define a formula $\varphi(i, j)$ describing the computational effect of Si and let

$$\Phi(P) = \bigvee_{(i,j) \in \mathrm{Diag}(P)} \varphi(i, j),$$

where $\varphi(i, j) = (E \wedge F \wedge G \wedge H)$, with

$$E = (z = t(Si) \wedge z' = t(Sj)),$$

$$F = \tau(\pi) = (\lambda(\pi), \varrho(\pi)) \wedge \lambda(\pi) = t(Si) \wedge \varrho(\pi) = r,$$

$G = \mathbf{TRUE}$ if the arrow $t(Si) \to t(Sj)$ is unlabelled,
$= C[x/r]$ if the arrow $t(Si) \to t(Sj)$ has the label C,
$= \neg C[x/r]$ if the arrow $t(Si) \to t(Si)$ has the label $\neg C$,

$H = (s = r)$ if $t(Si)$ is not the type of an execution scheme 3.3 or 3.4,
$= (s = r+1)$ if $t(Si)$ is the type of an execution scheme in 3.3,
$= (s = r-1)$ if $t(Si)$ is the type of an execution scheme in 3.4.

Here $(s = r)$ stands for $(s1 = r1 \wedge \ldots sn = rn)$, $(s = r+1)$ stand for $(s1 = r1 \wedge \ldots \wedge si = ri+1 \wedge \ldots \wedge sn = rn)$, where $[xi/xi+1]$ is the assignment typed by $t(Si)$. The conjunction $(s = r-1)$ is defined analogously.

We take care of the trivial programs not containing any assignment statements by defining $\Phi(\mathbf{null}) = \Phi(\mathbf{null}//\mathbf{null}) = \varphi(0, 0) = \mathbf{TRUE}$, and treating any other trivial program as a program in the single variable $x1$, so that it is consistent to write $(r1 = s1)$ etc.

The following result is immediate from the definition of $\varphi(i, j)$:

6.1. THEOREM. *$N \models \varphi(i, j)[a]$ if and only if there exists a cancellation tree $\mathrm{Tr}(P, \boldsymbol{a})$ and a path $e = (e1, \ldots, en, e(n+1))$ in $\mathrm{Tr}(P, \boldsymbol{a})$ such that $en = (Si, \boldsymbol{bi})$ and $e(n+1) = (Sj, \boldsymbol{bj})$, and such that the valuation $[\boldsymbol{a}]$ of the variables z, z', π, r, and s in the formula $\varphi(i, j)$ is $[z/ti, z'/tj, \pi/e, r/\boldsymbol{bi}, s/\boldsymbol{bj}]$.* ∎

6.2. COROLLARY. *$N \models \Phi(P)[a]$ if and only if $N \models \varphi(i, j)$ for some i, j.* ∎

It is clear that the formula $\Phi(P)$ described precisely the computational behaviour of the program P and can therefore be taken as the arithmetical meaning of P. By Theorem 4.1, the special function and relation symbols occurring in $\Phi(P)$ can be eliminated and the arithmetical meaning of P can be fully described in PA.

7. Properties of programs as arithmetical formulas

We now use the program formulas $\Phi(P)$ to express several properties of parallel programs in PA. The verification of the appropriateness of these definitions is routine.

7.1. TERMINATION. A parallel program terminates at $\boldsymbol{a} \in N^n$ if every maximal path of $\mathrm{Tr}(P, \boldsymbol{a})$ ends in a leaf of the form $(\mathbf{null}, \boldsymbol{b})$. We can express this fact by means of the formula "$\mathrm{term}(P, \boldsymbol{a})$" defined as

$$(\forall \pi \in \Pi)\big(G(P, \boldsymbol{a}) \subseteq \pi \Rightarrow (\exists \pi' \in \Pi)(\pi \subseteq \pi' \wedge \lambda(\pi') = t(\mathbf{null}))\big).$$

In [2] and [3], the property expressed by $\mathrm{term}(P, \boldsymbol{a})$ is referred to as "universal termination" in contrast to the property expressed by the formula

$$(\exists \pi \in \Pi)(G(P, \boldsymbol{a}) \subseteq \pi \wedge \lambda(\pi) = t(\mathbf{null})),$$

which merely asserts the existence of some terminating path and which therefore represents "existential termination".

7.2. PARTIAL CORRECTNESS. If $A(r)$ is an input condition in the input variables r and $B(s)$ is an output condition in the output variables s of $\Phi(P)$, then P is partially correct at $a \in N^n$ with respect to $A(r)$ and $B(s)$ if

$$N \models A(r) \wedge \Phi(P) \wedge \mathrm{term}(P, a) \Rightarrow B(s).$$

7.3. TOTAL CORRECTNESS. If $A(r)$ is an input condition in the input variables r and $B(s)$ is an output condition in the output variables s of $\Phi(P)$, then P is totally correct at $a \in N^n$ with respect to $A(r)$ and $B(s)$ if

$$N \models A(r) \wedge \Phi(P) \Rightarrow \mathrm{term}(P, a) \wedge B(s).$$

7.4. FREEDOM FROM DEADLOCK. The fact that every node of a cancellation tree $\mathrm{Tr}(P, a)$ which is not a leaf of the form (**null**, b) has a next node can be expressed by the formula

$$(\forall \pi \in \Pi)\big(G(P, a) \subseteq \pi \wedge \lambda(\pi) \neq t(\textbf{null}) \Rightarrow (\exists \pi' \in \Pi)(\pi \subseteq \pi' \wedge \pi \neq \pi')\big).$$

7.5. EQUIVALENCE. We can express the computational equivalence of two programs P and Q at a given input $a \in N^n$ by means of a conjugation $(\Psi(P, Q, a) \wedge \Psi(Q, P, a))$, where

$$\Psi(P, Q, a) = \big(\forall \pi \in \Pi_P\big)\big(\lambda(\pi) = t(\textbf{null}) \wedge G(P, a) \subseteq \pi \Rightarrow \big(\exists \pi' \in \Pi_Q\big)\big(\lambda(\pi') = t(\textbf{null}) \wedge G(Q, a) \subseteq \pi' \wedge \varrho(\pi) = \varrho(\pi')\big)\big),$$

and

$$\Psi(Q, P, a) = \big(\forall \pi' \in \Pi_Q\big)\big(\lambda(\pi') = t(\textbf{null}) \wedge G(Q, a) \subseteq \pi' \Rightarrow \big(\exists \pi \in \Pi_P\big)\big(\lambda(\pi) = t(\textbf{null}) \wedge G(P, a) \subseteq \pi \wedge \varrho(\pi') = \varrho(\pi)\big)\big).$$

The desired computational equivalence of programs is obtained by specifying that

$$P \equiv Q \quad \text{if and only if} \quad N \models (\forall a \in N^n)(\Psi(P, Q, a) \wedge \Psi(Q, P, a)).$$

7.6. FUNCTIONALITY. The following formula asserts that any two terminating computation paths in $\mathrm{Tr}(P, a)$ produce the same output:

$$(\forall \pi, \pi' \in \Pi)\big(G(P, a) \subseteq \pi \wedge G(P, a) \subseteq \pi' \wedge \lambda(\pi) = \lambda(\pi') = t(\textbf{null}) \Rightarrow \varrho(\pi) = \varrho(\pi')\big).$$

In [2], this property is referred to as the "partial determinateness" of P.

References

[1] K. Apt, *Recursive assertions and parallel programs*, Acta Informatica 15 (1981), 219–232.

[2] E. Ashcroft and Z. Manna, *Formalization of properties of parallel programs*, Machine Intelligence 6 (1971), 17–41.

[3] M. Ben-Ari, A. Pnueli and Z. Manna, *The temporal logic of branching time*, Acta Informatica 20 (1983), 207–226.

[4] E. J. Farkas, *A type structure for parallel programs*, Ph. D. thesis, Concordia University, Montreal 1985.

[5] – and M. E. Szabo, *On the programs-as-formulas interpretation of parallel programs in Peano arithmetic*, Ann. Pure and Appl. Logic, to appear.

Presented to the semester
Mathematical Problems in Computation Theory
September 16–December 14, 1985