Annals of Pure and Applied Logic 37 (1988) 111–127
North-Holland

ON THE PROGRAMS-AS-FORMULAS INTERPRETATION OF PARALLEL PROGRAMS IN PEANO ARITHMETIC

E.J. FARKAS*
Department of Mathematics, Concordia University, Montreal, Canada

M.E. SZABO*
Department of Mathematics, Concordia University, Montreal, Canada

Communicated by D. van Dalen
Received 2 September 1985

0. Introduction

The operational semantics for parallel programs in [1], [2], [7], [8] and elsewhere is based on the interpretation of parallel computations as non-deterministic sequential executions of parallel processes. In this context, properties of programs are studied as properties of the resulting finitely branching computation trees. In this paper we show that relative to a suitably defined type structure on such trees, parallel programs have a natural interpretation as formulas of Peano arithmetic (referred to below as PA) and that the computation trees of deadlock-free parallel programs determine intuitionistic fans which code their own spread laws. We also show that the choice sequences of these fans characterize the partial correctness of such programs. Arithmetical formulas associated with sequential programs have been studied in [3] and [4].

1. A type structure for parallel programs

1.1. *The language PL*

The class of programs considered in this paper is similar to the class of programs studied in [7], [8], and [1], restricted to successor and predecessor arithmetic as in [9]. The language is slightly weaker than that in [8] because of restrictions on the occurrences of await statements, but is stronger than that in [1] because of the broader class of await statements allowed. Programs are constructed inductively as certain types of 'program statements' from atomic statements as follows:

* The research of both authors is supported 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.

1.1.1. **null** is an (atomic) program statement.

1.1.2. If x is a variable and v is x, $(x+1)$, or $(x-1)$, then $[x/v]$ is an (atomic) program statement.

1.1.3. If $P1, \ldots, Pn$ are program statements, then **compose**$(P1, \ldots, Pn)$ is a program statement.

1.1.4. If $C(x1, \ldots, xn)$ is a quantifier-free formula of PA, $P1$ and $P2$ are program statements, and $x1, \ldots, xn$ are among the variables of $P1$ and $P2$, then **if**$(C, P1, P2)$ is a program statement.

1.1.5. If $C(x1, \ldots, xn)$ is a quantifier-free formula of PA, $P1$ is a program statement, and $x1, \ldots, xn$ are among the variables of $P1$, then **while**$(C, P1)$ is a program statement.

1.1.6. If $P1, \ldots, Pn$ are program statements, then **parallel**$(P1, \ldots, Pn)$ is a program statement.

1.1.7. If $C(x1, \ldots, xn)$ is a quantifier-free formula of PA, $P1$ is a program statement constructed by means of 1.1.1–1.1.5 and $x1, \ldots, xn$ are among the variables of $P1$, then **await**$(C, P1)$ is a program statement.

The following program statements are programs:

1.1.8. The **null** statement is a program.

1.1.9. All assignment statements $[x/v]$ are programs.

1.1.10. If $P1, \ldots, Pn$ are neither await statements nor compose statements, then **compose**$(P1, \ldots, Pn)$ is a program.

1.1.11. If $P1$ and $P2$ are not await statements, then **if**$(C, P1, P2)$ is a program.

1.1.12. If $P1$ is not **null** and is not an await or while statement, then **while**$(C, P1)$ is a program.

1.1.13. If $P1, \ldots, Pn$ are programs or await statements, then **parallel**$(P1, \ldots, Pn)$ is a program.

The condition in 1.1.10 that Pi is not a compose statement amounts to assuming that composition is associative. We also assume that the program **while**$(C,$ **while**$(C, P1))$ has the same meaning as **while**$(C \wedge C, P1)$ and do not allow nested while statements. Sometimes we write $(P1 // P2)$ as an abbreviation of **parallel**$(P1, P2)$ and treat $((P1 // P2) // P3)$ and $(P1 // (P2 // P3))$ as synonymous with $(P1 // P2 // P3)$. Every program statement that enters into the inductive construction of a program P at some stage is called a component of P.

In order to be able to describe the program components occurring in the computation trees of parallel programs diagrammatically, we define the type of a program statement.

1.2. *Program types*

1.2.1. The natural numbers $1, 2, \ldots$ are atomic types.

1.2.2. Every atomic type is a type.

1.2.3. If t is a type, then $(1, t)$ and $(2, t)$ are types.

1.2.4. If $t1$ and $t2$ are types, then $(3, (t1, t2))$ is a type.

1.2.5. If $t1, \ldots, tn$ are types, then $(4, (t1, \ldots, tn))$ and $(5, (t1, \ldots, tn))$ are types.

1.3. *The type $t(P)$ of a program statement P*

1.3.1. $t(\mathbf{null}) = 1$.

1.3.2. $t([x/v]) = 2, 3, \ldots$.

1.3.3. $t(\mathbf{while}(C, P1)) = (1, t)$, where $t(P1) = t$.

1.3.4. $t(\mathbf{await}(C, P1)) = (2, t)$, where $t(P1) = t$.

1.3.5. $t(\mathbf{if}(C, P1, P2)) = (3, (t1, t2))$, where $t(P1) = t1$ and $t(P2) = t2$.

1.3.6. $t(\mathbf{compose}(P1, \ldots, Pn)) = (4, (t1, \ldots, tn))$, where $t(Pi) = ti$.

1.3.7. $t(\mathbf{parallel}(P1, \ldots, Pn)) = (5, (t1, \ldots, tn))$, where $t(Pi) = ti$.

We call two types disjoint if they 'share' no atomic types except possibly the type 1. The specific types of the different occurrences of assignment statements in the analysis of a program P are intended to be context-dependent and are to be chosen so that all occurrences of assignment statements have disjoint types.

1.3.8. Lemma. *For any program P there exists a choice of atomic types for the distinct occurrences of the atomic statements in P which produces disjoint types for the distinct components of P.*

Proof. By an induction on programs. The result holds because a program has only finitely many possible components and because the supply of atomic types is infinite. □

From now on we work with programs whose distinct occurrences of atomic components have been assigned disjoint types. We assume as given an effective system G of Gödel numbers for finite sequences and finite sequences of finite sequences, etc., of natural numbers and can therefore think of a type t as a natural number $G(t)$, whenever convenient. Types will be used to describe the currently active component in the course of a run of a program P. We usually write t in place of $G(t)$.

We base the definition of the currently active component of a program on the concept of an execution scheme. These schemes describe the possible program components of the nodes of computation trees of parallel programs.

1.4. *Execution schemes*

In order to describe the interactions of the different components of a program P in the course of a run of P, we introduce the concept of the tagged and untagged execution schemes of the components of P. Certain subschemes of an execution scheme are 'tagged' with the type of a scheme whose execution must be delayed until the subscheme in question has been completely executed. The required tags correspond to while and compose statements and will always be of the form $(1, u)$ or $(4, (t1, \ldots, u, 1))$, where $u = t(S)$. If $t = t(S)$ and E is another execution scheme, $t:E$ denotes the fact that E has been tagged with t. Unless E is **null**, $t(S):E$ means that E has priority over S in the execution of P. We allow strings of tags to tag nested tagged schemes, i.e., we allow $t2, t1:E$, etc. These schemes correspond to the nesting of while and compose statements.

1.4.1. The execution scheme of **null** is **null**.

1.4.2. The execution schemes of t:**null** are t:**null** and, if $t = (4, (t1, \ldots, u, 1))$ and $u = t(S)$, the execution schemes of S.

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

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

1.4.5. 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 $P1$.

1.4.6. 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 $u:P1$.

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

1.4.8. 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 $t:P2$.

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

1.4.10. 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**.

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

1.4.12. 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 $P2$.

1.4.13. 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 $P2$.

1.4.14. 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. The following obvious lemma is essential for our programs-as-formulas interpretation:

1.5. Lemma. *Each program $P \in$ PL determines only finitely many distinct execution schemes $S1, \ldots, Sn$, and each Si occurs only finitely often in any calculation of the execution schemes determined by P.*

Proof. By an induction on programs. The only non-trivial case is that of a while scheme and the restriction in 1.4.2 ensures that every calculation of execution schemes terminates after finitely many steps. □

Thus if $t([x/x+1]) = 2$, the execution schemes of the program $P1 =$ **while**(C, $[x/x+1]$), for example, are $S1 = P$ (by 1.4.9), $S2 = (1, 2):[x/x+1]$ (by 1.4.9), $S3 = (1, 2):$**null** (by 1.4.4), and $S4 =$ **null** (by 1.4.9), and the execution schemes of $P2 =$ **compose**($[x/x+1]$, $[y/y+1]$) are $S1 = P2$ (by 1.4.5), $S2 = (4, (2, 1)):[y/y+1]$ (by 1.4.5), $S3 = (4, (2, 1)):$**null** (by 1.4.4), $S4 = [x/x+1]$ (by 1.4.5), and $S5 =$ **null** (by 1.4.3).

In certain calculations below, the information contained in the tags of the execution schemes occurring inside parallel statements is explicitly required. We therefore extend the notion of a type to execution schemes:

1.6. The type of a tagged execution scheme of the form $t1, \ldots, tn:Q$ is the pair $((6, t1, \ldots, tn), t(Q))$.

For a correct coding of tagged and untagged execution schemes in Section 4 we agree that whenever needed, an untagged execution scheme Q is represented by the term $u = ((6, 0), t(Q))$, so that the number $G(u)$ describes both types of execution schemes.

2. The next-node construction

We structure the possible orders of execution of a parallel program P with initial input **a** in the form of a rooted tree Tr(P, **a**) whose nodes contain both a

program and a data component describing the possible currently active components of P and the possible values computed up to the given point in a computation. Since the program component of a node is obtained by a kind of cancellation procedure from the execution schemes determined by P, we refer to $\mathrm{Tr}(P, \mathbf{a})$ as a cancellation tree. The root of $\mathrm{Tr}(P, \mathbf{a})$ is the pair $(P, \mathbf{a})$ and every other node is computed from an earlier node by the next node function σ.

2.1. *Next nodes*

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

(a) $\sigma(\mathbf{null}, \mathbf{b})$ is undefined.
$\sigma(t{:}\mathbf{null}, \mathbf{b}) = (S, \mathbf{b})$,
where $u = t(S)$ and $t = (1, u)$ or $(4, (u, 1))$.

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

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

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

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

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

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

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

(h) $\sigma(P1/\!/P2, \mathbf{b}) = (\sigma 1(P1/\!/P2, \mathbf{b}), \sigma 2(P1/\!/P2, \mathbf{b}))$.
$\sigma(P1/\!/(P2/\!/P3), \mathbf{b}) = \sigma(P1/\!/P2/\!/P3, \mathbf{b})$.
$\sigma((P1/\!/P2)/\!/P3, \mathbf{b}) = \sigma(P1/\!/P2/\!/P3, \mathbf{b})$.
$\sigma(t{:}(P1/\!/P2), \mathbf{b}) = (\sigma 1(t{:}(P1/\!/P2), \mathbf{b}), \sigma 2(t{:}(P1/P2), \mathbf{b}))$.

$\sigma(t:(P1//(P2//P3)), \mathbf{b}) = \sigma(t:(P1//P2//P3), \mathbf{b})$.
$\sigma(t:((P1//P2)//P3), \mathbf{b}) = \sigma(t:(P1//P2//P3), \mathbf{b})$.
$\sigma(t:(\mathbf{null}//\mathbf{null}), \mathbf{b}) = (S, \mathbf{b})$,
where $u = t(S)$ and $t = (1, u)$ or $(4, (u, 1))$.

(i) $\sigma 1(\mathbf{null}//P, \mathbf{b}) =$ undefined.
If $P = \mathbf{await}(C, P1)$, $u = t(S)$, and $t = (1, u)$ or $(4, (u, 1))$, then
$\sigma 1(t:\mathbf{null}//P, \mathbf{b}) = (S//P, \mathbf{b})$ if $C[\mathbf{b}]$ is false,
$= (S//P1, \mathbf{b})$ if $C[\mathbf{b}]$ is true;
if $P \neq \mathbf{await}(C, P1)$, $u = t(S)$, and $t = (4, (u, 1))$, then
$\sigma 1(t:\mathbf{null}//P, \mathbf{b}) = (S//P, \mathbf{b})$;
if $P = \mathbf{null}$ or $t':\mathbf{null}$, $u = t(S)$, and $t = (1, u)$, then
$\sigma 1(t:\mathbf{null}//P, \mathbf{b}) = (S//P, \mathbf{b})$;
$\sigma 1(t:\mathbf{null}//P, \mathbf{b})$ is undefined otherwise.

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

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

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

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

(n) $\sigma 1(\mathbf{await}(C, P1)//P, \mathbf{b}) = (P1//P, \mathbf{b})$ if $C[\mathbf{b}]$ is true,
$=$ undefined if $C[\mathbf{b}]$ 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 $\sigma(P1//P2, \mathbf{b})$ may give rise to only a single node if $\sigma 1(P1//P2, \mathbf{P})$ or $\sigma 2(P1//P2, \mathbf{b})$ is undefined. The ordered pair notation is intended to convey the idea that the next nodes of $(P1//P2, \mathbf{b})$ are considered to be ordered from left to right, with $\sigma 1(P1//P2, \mathbf{b})$ to the left of $\sigma 2(P1//P2, \mathbf{b})$. Similarly for more than two next nodes. The motivation behind the various steps

in the definition of σ is as follows: Assignment statements are indivisible operations, are always executable, and are carried out in a single step. Compose statements are executed from right to left, reflecting our interpretation of composition as functional application. The execution of a while statement requires that we separate notationally the component to be executed from the statement as a whole and only revert back to the while statement once the component has been completely executed. Moreover, parallel statements are executed as non-deterministic sequential statements and we therefore require all pairs of execution schemes of $P1$ and $P2$ to model parallel execution. In all cases the definition reflects our intention to treat only assignments and the evaluation of the truth of $C[\mathbf{b}]$ as indivisible operations. This assumption agrees with the operational interpretation of parallel programs in [7]. A similar motivation also underlies the description of the semantics for parallel programs in [2] based on the work of Hennessy and Plotkin.

2.2. Example. Let $P = \mathbf{parallel}(P1, P2)$, where

$$P1 = \mathbf{while}(\mathbf{TRUE}, [x/x+1]) \quad \text{and}$$

$$P2 = \mathbf{while}(\mathbf{TRUE}, \mathbf{compose}([x/x+1], [y/y+1])).$$

In order to list the execution schemes of P we must, according to 1.4.12, compute the execution schemes of $P1$ and $P2$. The execution schemes of $P1$ and their types are the following:

$$S1 = \mathbf{while}(\mathbf{TRUE}, [x/x+1]), \quad t1 = (1, 2),$$
$$S2 = t1 : [x/x+1], \quad t2 = ((6, t1), 2),$$
$$S3 = t1 : \mathbf{null}, \quad t3 = ((6, t1), 1),$$
$$S4 = \mathbf{null}, \quad t4 = 1,$$

and the execution schemes of $P2$ and their types are

$$T1 = \mathbf{while}(\mathbf{TRUE}, \mathbf{compose}([x/x+1], [y/y+1])), \quad t4 = (1, (4, (3, 4))),$$
$$T2 = t4 : \mathbf{compose}([x/x+1], [y/y+1]), \quad t5 = ((6, t4), (4, (3, 4))),$$
$$T3 = t4, t6 : [y/y+1], \quad t7 = ((6, t4, (4, (3, 1))), 4),$$
$$T4 = t4, t6 : \mathbf{null}, \quad t8 = ((6, t4, t6), 1),$$
$$T5 = t4 : [x/x+1], \quad t9 = ((6, t4), 3),$$
$$T6 = t4 : \mathbf{null}, \quad t10 = ((6, t4), 1),$$
$$T7 = \mathbf{null}, \quad t11 = 1.$$

The two distinct occurrences of $[x/x+1]$ have been assigned the distinct types 2 and 3 and the statement $[y/y+1]$ has been assigned the type 4. Hence the conditions of Lemma 1.3.8 are met.

It is clear that Tr(P, (0, 0)), for example, has only infinite branches such as

$$(S1//T1, (0, 0))\rightarrow(S2//T1, (0, 0))\rightarrow$$
$$(S3//T1, (1, 0))\rightarrow(S3//T2, (1, 0))\rightarrow$$
$$(S3//T3, (1, 0))\rightarrow(S3//T4, (1, 1))\rightarrow$$
$$(S3//T5, (1, 1))\rightarrow(S3//T6, (2, 1))\rightarrow$$
$$(S3//T1, (2, 1))\rightarrow(S1//T1, (2, 1))\rightarrow\cdots \quad \square$$

3. Programs as formulas

The purpose of this section is to show that the computational behaviour of a program $P \in \mathrm{PL}$ can be characterized by formulas $\Phi(P)$ of PA in which we can express the idea of a sequence $\mathbf{a}1 \in N^n, \ldots, \mathbf{a}n \in N^n$ of values consisting of successive data components of a cancellation tree of P. For this purpose we represent the nodes $(E, \mathbf{b})$ of all trees Tr(P, $\mathbf{a}$) by $(G(u), G(\mathbf{b}))$, where E is either a tagged or untagged execution scheme of P, and u is the type of E as defined in 1.3 and 1.6. We let Paths(P) be the set of all finite sequences $e = (e1, \ldots, en)$ of such pairs with the property that $e1 = (G(t(P)), G(\mathbf{a}))$, for some $\mathbf{a} \in N^n$, and such that if $e(i+1) \in e$, then $e(i+1)$ corresponds to a next node of $e(i)$ in Tr(P, $\mathbf{a}$). It is clear from the algorithmic nature of the next-node function that Paths(P) is a decidable set. We introduce three special functions $\tau = \tau(\mathrm{Paths}(P))$, $\lambda = \lambda(\mathrm{Paths}(P))$, and $\rho = \rho(\mathrm{Paths}(P))$, defined on finite sequences $e = (e1, \ldots, en)$, by

$$\begin{aligned}
&\text{(a)}\ \tau(e) = en && \text{if } e \in \mathrm{Paths}(P),\\
&\qquad = (0, 0) && \text{if } e \notin \mathrm{Paths}(P),\\
&\text{(b)}\ \lambda(e) = \lambda(en) && \text{if } e \in \mathrm{Paths}(P),\\
&\qquad = 0 && \text{if } e \notin \mathrm{Paths}(P),\\
&\text{(c)}\ \rho(e) = \rho(en) && \text{if } e \in \mathrm{Paths}(P),\\
&\qquad = 0 && \text{if } e \notin \mathrm{Paths}(P),
\end{aligned}$$

and let $(\lambda(i), \rho(i)) = ei$. The functions $\tau(\mathrm{Paths}(P))$, $\lambda(\mathrm{Paths}(P))$ and $\rho(\mathrm{Paths}(P))$ are clearly effective and, relative to the coding of the finite sequences e as natural numbers $G(e)$, yield three calculable functions $\tau: N \rightarrow (N \times N)$, $\lambda: N \rightarrow N$, and $\rho: N \rightarrow N$. By Church's thesis these functions are recursive and are therefore representable in PA. We call the elements of Paths(P) the finite paths determined by P, so that $\tau(e)$ represents the last node en of the path e, and $\lambda(e) = \lambda(n)$ codes the program component and $\rho(e) = \rho(n)$ the data component of en. We let π be a new variable ranging over all finite sequences of the kind described. In addition, we introduce two new (control) variables z and z', with z ranging over the numerical codes of the types of the execution schemes of P and z' ranging over

the numerical codes of the types of the obvious 'next execution schemes' determined by a given scheme. We omit the definition of next execution schemes since it is clear from 1.4. For each variable vi in P we introduce a new (input) variable ri and a new (output) variable si ranging over the current and next values of the variables of P in the course of a run of P relative to an initial state $\mathfrak{a}$. The variables z, z', ri and si are assumed to be distinct. We put $r=(r1,\ldots,rn)$ and $s=(s1,\ldots,sn)$ and simplify the notation by writing $(s=r)$ in place of $(s1=r1\wedge\cdots\wedge sn=rn)$ and $(s1=r1\wedge\cdots\wedge s(i-1)=r(i-1)\wedge s(i+1)=r(i+1)\wedge\cdots\wedge sn=rn)$ if the intended conjunction is clear from the context.

The formula $\Phi(P)$ characterizes the different possible kinds of changes of the control variables z and z' determined by P and describes, at the same time, the accompanying changes in the values of the input variables $r1,\ldots,rn$ and output variables $s1,\ldots,sn$. The construction of $\Phi(P)$ involves four steps:

(a) The listing of the occurrences $S1,\ldots,Sn$ of the execution schemes of the given program.

(b) The determination of the types $t1,\ldots,tn$ of $S1,\ldots,Sn$.

(c) The specification of the 'diagram' Diag(P) of P determined by $t1,\ldots,tn$.

(d) The definition of $\Phi(P)$ from the data accumulated in (a)–(c).

We illustrate $\Phi(P)$ by several examples which, at the same time, form the induction basis for the construction. We usually write finite paths as $e=((\lambda(1),\rho(1)),\ldots,(\lambda(n),\rho(n)))$, with $\lambda(i)=\rho(i)=0$ if the finite sequence e is not a finite path of P.

3.1. Example. Let $P=$ **null**. Then the only execution scheme of P is P itself, i.e., $S1=P$ and $t1=t(S1)=1$. Hence Diag(P) consists of the single node $t1$. We let $\Phi(P)=\phi(t1)=(z1=t1\wedge z1'=t1\wedge r=\rho(1)=\mathfrak{a}\wedge s=r)$. The same formula is used for any other trivial program whose only atomic component is **null**. □

3.2. Example. Let $P=[x/x+1]$ and $t(P)=2$. Then

$$S1=P,\qquad t1=t(S1)=2,$$
$$S2=\textbf{null},\qquad t2=t(S2)=1,\qquad \mathrm{Diag}(P)=t1\to t2.$$

The presence of an arrow from ti to tj indicates that Sj is a possible program component of a next node, as determined in 2.1, of the node with program component Si.

$$\Phi(P)=\phi(t1,t2)=\phi(1,2)=(z=t1\wedge z'=t2\wedge r=\rho(1)=\mathfrak{a}\wedge s=r+1).$$ □

3.3. Example. Let $P=$ **compose**$([x/x+1],[y/y+1])$. Then the execution schemes determined by P and their respective types are the following, with $t0=(4,(2,1))$:

$$S1=P,\qquad t1=(4,(2,3)),$$
$$S2=t0:[y/y+1],\qquad t2=((6,t0),3),$$

$S3 = t0$:**null**, $t3 = ((6, t0), 1)$,

$S4 = [x/x+1]$, $t4 = 2$,

$S5 =$ **null**, $t5 = 1$,

$\mathrm{Diag}(P) = t1 \to t2 \to t3 \to t4 \to t5$.

For each arrow $(ti \to tj)$ in $\mathrm{Diag}(P)$, we specify a formula $\phi(i, j)$ which describes the current and next values of z, z', r, and s, and let $\Phi(P)$ be the disjunction of the $\phi(i, j)$:

$$\phi(1,2) = (z = t1 \wedge z' = t2 \wedge r = \rho(1) = \mathbf{a} \wedge s = r),$$

$$\phi(2,3) = (z = t2 \wedge z' = t3 \wedge \lambda(2) = t2 \wedge r = \rho(2) \wedge s1 = r1 \wedge s2 = r2 + 1),$$

$$\phi(3,4) = (z = t3 \wedge z' = t4 \wedge \lambda(3) = t3 \wedge r = \rho(3) \wedge s = r),$$

$$\phi(4,5) = (z = t4 \wedge z' = t5 \wedge \lambda(4) = t4 \wedge r = \rho(4) \wedge s1 = r1 + 1 \wedge s2 = r2).$$

$$\Phi(P) = \phi(1,2) \vee \phi(2,3) \vee \phi(3,4) \vee \phi(4,5). \quad \square$$

3.4. Example. Let $P =$ **while**$(C, [x/x+1])$. Then the execution schemes of P and their types are the following:

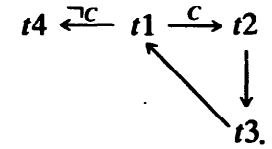
$S1 = P$, $t1 = (1, 2)$,

$S2 = t1$:$[x/x+1]$, $t2 = ((6, t1), 2)$,

$S3 = t1$:**null**, $t3 = ((6, t1), 1)$,

$S4 =$ **null**, $t4 = 1$,

$\mathrm{Diag}(P) =$



$$\phi(1,2) = (z = t1 \wedge z' = t2 \wedge \tau(\pi) = (\lambda(\pi), \rho(\pi)) \wedge \lambda(\pi) = t1 \wedge \rho(\pi) = r \wedge C[x/r] \wedge s = r),$$

$$\phi(1,4) = (z = t1 \wedge z' = t4 \wedge \tau(\pi) = (\lambda(\pi), \rho(\pi)) \wedge \lambda(\pi) = t1 \wedge \rho(\pi) = r \wedge \neg C[x/r] \wedge s = r),$$

$$\phi(2,3) = (z = t2 \wedge z' = t3 \wedge \tau(\pi) = (\lambda(\pi), \rho(\pi)) \wedge \lambda(\pi) = t2 \wedge \rho(\pi) = r \wedge s = r + 1),$$

$$\phi(3,1) = (z = t3 \wedge z' = t1 \wedge \tau(\pi) = (\lambda(\pi), \rho(\pi)) \wedge \lambda(\pi) = t3 \wedge \rho(\pi) = r \wedge s = r).$$

$$\Phi(P) = \phi(1,2) \vee \phi(1,4) \vee \phi(2,3) \vee \phi(3,1). \quad \square$$

3.5. Example. Let $P =$ **if**$(C, [x/x+1], [y/y+1])$. Then the execution schemes of

P and their types relative to $t([x/x+1])=2$ and $t([y/y+1])=3$ are

$$S1=P, \qquad t1=t(S1)=(3,(2,3)),$$
$$S2=[x/x+1], \qquad t2=t(S2)=2,$$
$$S3=\textbf{null}, \qquad t3=t(S3)=1,$$
$$S4=[y/y+1], \qquad t4=t(S4)=3,$$
$$S5=\textbf{null}, \qquad t5=t(S5)=1,$$

$$\text{Diag}(P)= \quad t5 \longleftarrow t4 \xleftarrow{\neg C} t1 \xrightarrow{C} t2 \longrightarrow t3.$$

$$\phi(1,2)=(z=t1 \wedge z'=t2 \wedge \lambda(1)=t1 \wedge \rho(1)=r \wedge C[x/r] \wedge s=r),$$
$$\phi(2,3)=(z=t2 \wedge z'=t3 \wedge \lambda(2)=t2 \wedge \rho(2)=r \wedge s1=r1+1 \wedge s2=r2),$$
$$\phi(1,4)=(z=t1 \wedge z'=t4 \wedge \lambda(1)=t1 \wedge \rho(1)=r \wedge \neg C[x/r] \wedge s=r),$$
$$\phi(4,5)=(z=t4 \wedge z'=t5 \wedge \lambda(2)=t4 \wedge \rho(2)=r \wedge s1=r1 \wedge s2=r2+1).$$

$$\Phi(P)=\phi(1,2) \vee \phi(2,3) \vee \phi(1,4) \vee \phi(4,5). \qquad \square$$

3.6. Example. If $P=\textbf{parallel}(\textbf{await}(C,[x/x+1]),\textbf{null})$, then the execution schemes of P and their respective types are

$$S1=P, \qquad t1=t(S1)=(5,((2,3),1)),$$
$$S2=\textbf{parallel}([x/x+1],\textbf{null}), \qquad t2=t(S2)=(5,(3,1)),$$
$$S3=\textbf{parallel}(\textbf{null},\textbf{null}), \qquad t3=t(S3)=(5,(1,1)),$$

$$\text{Diag}(P)= \quad t1 \xrightarrow{C} t2 \longrightarrow t3.$$

$$\phi(1,2)=(z=t1 \wedge z'=t2 \wedge \lambda(1)=t1 \wedge \rho(1)=r \wedge C[x/r] \wedge s=r),$$
$$\phi(2,3)=(z=t2 \wedge z'=t3 \wedge \lambda(2)=t2 \wedge \rho(2)=r \wedge s=r+1).$$

$$\Phi(P)=\phi(1,2) \vee \phi(2,3). \qquad \square$$

3.7. Example. If $P=\textbf{parallel}(\textbf{await}(C,[x/x+1]),[y/y+1])$, then the execution schemes of P and their types are

$$S1=P, \qquad t1=t(S1)=(5,((2,3),4)),$$
$$S2=[x/x+1]//[y/y+1], \qquad t2=t(S2)=(5,(3,4)),$$
$$S3=\textbf{null}//[y/y+1], \qquad t3=t(S3)=(5,(1,4)),$$
$$S4=\textbf{null}//\textbf{null}, \qquad t4=t(S4)=(5,(1,1)),$$
$$S5=\textbf{await}(C,[x/x+1])//\textbf{null}, \qquad t5=t(S5)=(5,((2,3),1)),$$

$S6 = [x/x+1]//\mathbf{null}$, $t6 = t(S6) = (5, (3, 1))$,

$S7 = \mathbf{null}//\mathbf{null}$, $t7 = t(S7) = (5, (1, 1))$,

$$\mathrm{Diag}(P) = \quad t7 \longleftarrow t6 \overset{C}{\longleftarrow} t5 \longleftarrow t1 \overset{C}{\longrightarrow} t2 \longrightarrow t3 \longrightarrow t4.$$

$\phi(1,2) = (z = t1 \wedge z' = t2 \wedge \lambda(1) = t1 \wedge \rho(1) = r \wedge C[x/r] \wedge s = r)$,

$\phi(2, 3) = (z = t2 \wedge z' = t3 \wedge \lambda(2) = t2 \wedge \rho(2) = r \wedge s1 = r1 + 1 \wedge s2 = r2)$,

$\phi(3, 4) = (z = t3 \wedge z' = t4 \wedge \lambda(3) = t3 \wedge \rho(3) = r \wedge s1 = r1 \wedge s2 = r2 + 1)$,

$\phi(1, 5) = (z = t1 \wedge z' = t5 \wedge \lambda(1) = t1 \wedge \rho(1) = r \wedge s1 = r1 \wedge s2 = r2 + 1)$,

$\phi(5, 6) = (z = t5 \wedge z' = t6 \wedge \lambda(2) = t5 \wedge \rho(2) = r \wedge C[x/r] \wedge s = r)$,

$\phi(6, 7) = (z = t6 \wedge z' = t7 \wedge \lambda(3) = t6 \wedge \rho(3) = r \wedge s1 = r1 + 1 \wedge s2 = r2)$.

$\Phi(P) = \phi(1, 2) \vee \phi(2, 3) \vee \phi(3, 4) \vee \phi(1, 5) \vee \phi(5, 6) \vee \phi(6, 7)$. □

3.8. Example. If $P = \mathbf{parallel}([x/x+1], [y/y+1])$, then the execution schemes of P and their types are

$S1 = P$, $t1 = t(S1) = (5, (3, 4))$,

$S2 = \mathbf{null}//[y/y+1]$, $t2 = t(S2) = (5, (1, 4))$,

$S3 = \mathbf{null}//\mathbf{null}$, $t3 = t(S3) = (5, (1, 1))$,

$S4 = [x/x+1]//\mathbf{null}$, $t4 = t(S4) = (5, (3, 1))$,

$S5 = \mathbf{null}//\mathbf{null}$, $t5 = t(S5) = (5, (1, 1))$,

$$\mathrm{Diag}(P) = \quad t5 \longleftarrow t4 \longleftarrow t1 \longrightarrow t2 \longrightarrow t3.$$

$\phi(1, 2) = (z = t1 \wedge z' = t2 \wedge \lambda(1) = t1 \wedge \rho(1) = r \wedge s1 = r1 + 1 \wedge s2 = r2)$,

$\phi(2, 3) = (z = t2 \wedge z' = t3 \wedge \lambda(2) = t2 \wedge \rho(2) = r \wedge s1 = r1 \wedge s2 = r2 + 1)$,

$\phi(1, 4) = (z = t1 \wedge z' = t4 \wedge \lambda(1) = t1 \wedge \rho(1) = r \wedge s1 = r1 \wedge s2 = r2 + 1)$,

$\phi(4, 5) = (z = t4 \wedge z' = t5 \wedge \lambda(2) = t4 \wedge \rho(2) = r \wedge s1 = r1 + 1 \wedge s2 = r2)$,

$\Phi(P) = \phi(1, 2) \vee \phi(2, 3) \vee \phi(1, 4) \vee \phi(4, 5)$. □

The general construction is obtained by applying the induction hypothesis to component programs and modifying the above examples in the obvious way. We describe the case of a program of the form $P = \mathbf{if}(C, P1, P2)$ in relation to our Example 3.5. The modifications required in the remaining cases are similar.

Suppose that the execution schemes of $P1$ are $S2, \ldots, Sp$, with respective types $t2, \ldots, tp$, that the execution schemes of $P2$ are $S(p+1), \ldots, S(p+q)$, with respective types $t(p+1), \ldots, t(p+q)$, and that $\mathrm{Diag}(P1)$ and $\mathrm{Diag}(P2)$ are the diagrams of $P1$ and $P2$ and that $\Phi(P1)$ and $\Phi(P2)$ are given. Then we let

$S1 = P$, $t1 = t(P)$, and let Diag(P) be the diagram

$$\text{Diag}(P2) \cdots \longleftarrow t(p+1) \xleftarrow{\neg C} t1 \xrightarrow{C} t2 \longrightarrow \cdots \text{Diag}(P1)$$

obtained by grafting the rest of Diag($P2$) to $t(p+1)$ and the rest of Diag($P1$) to $t2$.

$$\Phi(P) = \phi(1,2) \vee \phi(1,p+1) \vee \Phi(P1) \vee \Phi(P2),$$
$$\phi(1,2) = (z = t1 \wedge z' = t2 \wedge \lambda(1) = t1 \wedge \rho(1) = r \wedge C[x/r] \wedge s = r),$$
$$\phi(1,p+1) = (z = t1 \wedge z' = t2 \wedge \lambda(1) = t1 \wedge \rho(1) = r \wedge \neg C[x/r] \wedge s = r).$$

This completes the definition of $\Phi(P)$. The desired properties of the formulas $\Phi(P)$ are summarized in the following lemma which guarantees the soundness of the definition of $\Phi(P)$ for non-trivial programs and is clear from the construction of $\Phi(P)$:

3.9. Lemma. *If P contains at least one assignment statement, then* $N \models \Phi(P)[a]$ *if and only if P has at least two execution schemes Si and Sj with types ti and tj such that* $N \models \phi(i,j)[a]$.

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

Since a program has only finitely many execution schemes, there are only finitely many possible assignments of values to the control variables z and z' that satisfy $\Phi(P)$ in N. Let $[a1] = [c1, d1], \ldots, [ap] = [cp, dp]$ be these assignments. Then we can construct the formulas $\Phi(P)[z/c1, z'/d1], \ldots, \Phi(P)[z/cp, z'/dp]$ in which the variables z and z' are replaced by the terms $S^{ci}(0)$ and $S^{di}(0)$ of PA corresponding to the numbers ci and di. These formulas contain only the free variables π, r, and s. By the previous lemma, the disjunction of these formulas, which we denote by $D(\Phi(P))$, is satisfied only by an assignment e to the path variable π and an assignment **b** to the input variable r and **b'** to the output variable s for which **b'** is the next value of **b** determined by the finite path e in some cancellation tree Tr(P, **a**) of P. Let NVR(**a**) be the set of all $(\mathbf{b}, \mathbf{b}') \in N^n \times N^n$ with the property that $(Q, \mathbf{b})$, $(Q', \mathbf{b}')$ belong to Tr(P, **a**) and $(Q', \mathbf{b}')$ is $\sigma(Q, \mathbf{b})$, and define NVR (which we call the 'next-value relation' of P) as the union of the NVR(**a**), taken over N^n. We wish to show that the next-value relation of P is definable in PA. Since the functions τ, λ, and ρ are representable and hence definable in PA, we can replace the equations involving τ, λ, and ρ in

the formula $D(\Phi(P))$ by equivalent formulas not involving these symbols and obtain a formula $D'(\Phi(P)) \in \mathrm{PA}$ with the property that

$$N \vDash D(\Phi(P))[a] \quad \text{if and only if} \quad N \vDash D'(\Phi(P))[a].$$

The desired result therefore follows from 3.10:

3.11. Definability Theorem. *For any program* $P \in \mathrm{PL}$, *the next value relation* NVR *of* P *is definable in* PA.

4. Characterizing correctness

We now use the cancellation trees and formulas associated with parallel programs to define their partial correctness relative to given input and output conditions. For this purpose we call a leaf of a cancellation tree **real** if its program component is **null**. We use the program **null** as the generic example of all programs of the form **null//null**, **null//null//null**, etc. If $P \in \mathrm{PL}$ is a parallel program with an associated formula $\Phi(P) \in \mathrm{PA}$ and $\mathrm{Tr}(P, \mathbf{a})$ is the cancellation tree generated by P and the initial input $\mathbf{a} \in N^n$, we say that P terminates at $\mathbf{a}$ if $\mathrm{Tr}(P, \mathbf{a})$ is finite and all leaves of $\mathrm{Tr}(P, \mathbf{a})$ are real. We further say that P is partially correct in N with respect to an input condition $A \in \mathrm{PA}$ and output condition $B \in \mathrm{PA}$ if A is quantifier-free and its variables are among the input variables r of $\Phi(P)$ and B is quantifier-free and its variables are among the output varibles s of $\Phi(P)$, and if $N \vDash A[r/\mathbf{a}]$ implies that $N \vDash B[s/\mathbf{b}]$ for all $\mathbf{a} \in N^n$ for which P terminates and for all data components $\mathbf{b}$ of the real leaves of $\mathrm{Tr}(P, \mathbf{a})$. We use the usual notation and write $N \vDash A\{P\}B$ to express the fact that P is partially correct with respect to A and B. It is an exercise to verify that $N \vDash A\{P\}B$ if and only if it is partially correct in the traditional sense, as defined in [1] and [2], for example. The proof is by an induction on programs. It requires a translation of tagged execution schemes into sequences of programs.

The main result of this section is the fact that if P is a parallel program in which no component is permanently prevented from executing in $\mathrm{Tr}(P, \mathbf{a})$, because of the nature of P or because of the specific arithmetical conditions induced by the initial input $\mathbf{a}$, then the cancellation trees of P determine intuitionistic fans whose choice sequences characterize the partial correctness of P. We recall from [5] that a fan can be thought of as a decidable rooted tree of sequences of natural numbers having only infinite branches and having the further property that every node has only finitely many next nodes. Relative to our stipulated recursive coding G of n-tuples of natural numbers, the nodes of a fan are n-tuples of natural numbers. The fans constructed below will be of this form. We call a fan terminating if all of its branches are eventually constant. By abuse of language we think of a terminating fan as finite. In this sense Brouwer's fan theorem asserts that every terminating fan is finite. We denote the fan associated with a

cancellation tree Tr(P, **a**) by Fan(P, **a**). The nodes of Fan(P, **a**) describe the values of the control and input and output variables at the various stages in the computation of P.

4.1. *The construction of* Fan(P, **a**)

4.1.1. The root of Fan(P, **a**) is $(0, w1, \mathbf{a})$, where $w1 = t(P)$, i.e., the Gödel number of the type of P. The value $w1$ is the initial value of the control variable $z1$ and 0 the value of the control variable z2.

4.1.2. We replace every node $(Q, \mathbf{b}) \neq (P, \mathbf{a})$ of Tr(P, **a**) by $(0, w1, \mathbf{b})$, where $w1 = t(Q)$. Here $w1$ represents the current value of the control variable $z1$, with $z2 = 0$.

4.1.3. We replace every node of the form $(t1, \ldots, tn{:}Q, \mathbf{b})$ by $(w2, w1, \mathbf{b})$, where $w1 = t(Q)$ and where $w2$ is the Gödel number of the sequence of tags $(t1, \ldots, tn)$ of Q.

4.1.4. To every node $(w2, w1, \mathbf{b})$ of Fan(P, **a**) corresponding to a real leaf of Tr(P, **a**) we append an infinite number of copies of $(w2, w1, \mathbf{b})$.

If P is a sequential program, then Fan(P, **a**) is a run of P in the sense of [3] and [4], in which codes of sequences of tags correspond to numerical labels of programming instructions. It is clear from Lemma 3.10 and Corollary 3.11 and from the construction of Fan(P, **a**) that $c' = (w2', w1', \mathbf{b}')$ is a next node of a node $c = (w2, w1, \mathbf{b})$ of Fan(P, **a**) if and only if there exists a path $e = (e1, \ldots, en, e(n+1)) \in \mathrm{Tr}(P, \mathbf{a})$ with the property that $en = (\lambda(n), \rho(n)) \approx (w2, w1, \mathbf{b})$ and $e(n+1) = (\lambda(n+1), \rho(n+1)) \approx (w2', w1', \mathbf{b}')$ such that

$$N \models \Phi(P)[z/ti, z'/tj, \pi/e, r/\mathbf{b}, s/\mathbf{b}'],$$

where $ti = ((6, w2), w1)$ and $tj = ((6, w2'), w1')$, as defined in 1.6. Since P is a program and therefore determines an algorithm for deciding the next nodes of a node in Tr(P, **a**), this result shows that $\Phi(P)$ defines the spread law of Fan(P, **a**) in PA.

By construction, Fan(P, **a**) is a fan in the real sense only if every finite branch of Tr(P, **a**) ends in a node of the form (**null**, **b**), i.e., is a real leaf. A program $P \in \mathrm{PL}$ is called deadlocked at an initial input **a** if Tr(P, **a**) has a non-real leaf. P is deadlock-free at **a** if every path of Tr(P, **a**) is either infinite or has a real leaf. Thus Fan(P, **a**) is a fan if and only if P is deadlock-free at **a**. By suppressing the values of the control variables in the choice sequences, i.e., in the maximal paths of Fan(P, **a**), we can paraphrase our description of partially correct parallel programs as follows:

4.2. Correctness Theorem. *A deadlock-free parallel program P is partially correct in N with respect to input condition A and output condition B if and only if*

$N \models A[r/\mathbf{b1}]$ implies that $N \models B[s/\mathbf{bn}]$ for all finite choice sequences $(\mathbf{b1}, \ldots, \mathbf{bn})$ *determined by P.*

References

[1] K.R. Apt, Recursive assertions and parallel programs, Acta Informatica 15 (1981) 219–232.
[2] K.R. Apt, Ten years of Hoare's logic: A survey — Part II: Nondeterminism, Theoret. Comput. Sci. 28 (1984) 83–109.
[3] L. Csirmaz, Programs and program verification in a general setting, Theoret. Comput. Sci. 16 (1981) 199–210.
[4] L. Csirmaz, Nonstandard semantics in program verification, Preprint, Hungarian Academy of Sciences (1984).
[5] M.A.E. Dummett, Elements of Intuitionism (Oxford University Press, Oxford, 1977).
[6] E. J. Farkas, A type structure for parallel programs, Ph.D. Thesis, Concordia University, Montreal, 1985.
[7] S. Owicki, Axiomatic proof techniques for parallel programs, Ph.D. Thesis, Cornell University, Ithaca, NY, 1975.
[8] S. Owicki and D. Gries, An axiomatic proof technique for parallel programs I, Acta Informatica 6 (1976) 319–340.
[9] M.M. Richter and M.E. Szabo, Towards a nonstandard analysis of programs, in: A. E. Hurd, ed., Lecture Notes in Computer Science 983 (Springer, New York, 1983) 186–203.