CANADIAN THESES ON MICROFICHE

THÈSES CANADIENNES SUR MICROFICHE

National Library of Canada
Collections Development Branch

Canadian Theses on
Microfiche Service

Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Direction du développement des collections

Service des thèses canadiennes
sur microfiche

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

NL 339 (r 86/09)

Canada

A TYPE STRUCTURE FOR PARALLEL PROGRAMS

Erika Jennifer Farkas

A Thesis

in

The Faculty

of

Arts and Science

Presented in Partial Fulfilment of the Requirements
for the Degree of Doctor of Philosophy at
Concordia University
Montréal, Québec, Canada

August 1985

© Erika Jennifer Farkas, 1985

ABSTRACT

A Type Structure for Parallel Programs

Erika Jennifer Farkas, Ph.D.
Concordia University, 1985

This thesis deals with the development of a type structure for parallel programs that yields an operational semantics for interacting processes in which the structural and computational aspects of a program can be mathematically separated. We demonstrate that this structure is rich enough to serve as a basis for a programs-as-formulas interpretation of parallel programs in Peano arithmetic. This interpretation depends on our method of diagrams of program types. We prove that the set of values over the natural numbers computed by a program is definable in PA and give a new characterization of partial correctness. Next we show that our semantics admits a nonstandard interpretation in which while and await statements are subject to star-finite bounds. We demonstrate that relative to a first order extension of our method of diagrams, this approach leads to a star-finite semantics for deadlock-free parallel programs. In conclusion we show that the set of program types can be endowed with an algebraic structure that yields a complexity measure for parallel programs.

ACKNOWLEDGEMENTS

I would like to express my profound appreciation and gratitude to my advisor, Dr. M. E. Szabo, for having been my constant source of inspiration, guidance and encouragement over the years. He has taught me everything I know about logic and the writing, organizing and presenting of mathematical research.

I would also like to thank Dr. J. Meloul and Dr. S. Mullett for serving on my advisory committee.

I gratefully acknowledge the support of my research 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.

Finally I would like to thank the Natural Sciences and Engineering Research Council of Canada for continuing to support my research by having awarded me a Postdoctoral Fellowship.

CONTENTS

Introduction 1

Chapter 1. **Peano arithmetic and parallel programs** 5

1.1. A language and an axiom system for PA 6

1.2. Recursive functions and relations 12

1.3. The language PL 17

Chapter 2. **Programs as formulas** 20

2.1. The type structure 22

2.2. The next-node construction 28

2.3. Embedding PL in PA 34

2.4. Characterizing correctness 47

Chapter 3. **Definable trees and complexity types** 51

3.1. Nonstandard cancellation trees 56

3.2. The internal-fan construction 67

3.3. Complexity types 81

Bibliography 90

INTRODUCTION

In 1908 Bertrand Russell introduced a system of mathematical logic based on the theory of types which was to form a foundation for mathematics avoiding the pitfalls of the vicious-circle principle which had led to contradictions in earlier systems of logic. He thought of a type as a range of significance to which universal quantifiers could apply. Since then hierarchies of types have been introduced into other areas of mathematics in order to study metamathematical properties of a variety of classes of objects (cf. FEFERMAN [1977]). Around 1939, for example, Turing showed that every formula of Church's simple type theory has a normal form (cf. GANDY [1980]) and this technique was later applied to a theory of functions known as the lambda calculus to prove that if the functions involved are stratified by a system of types, then every function in the system has a uniquely identifiable normal form. It is well known that an analogous normalization of functions is impossible without the use of types. The reason again is the vicious-circle principle, this time in the form of self-application where the same element is used simultaneously as "function" and "argument". Since normalization procedures have become central to decidable equivalence theories in various branches of mathematics in the guise of Church-Rosser theorems and in cut-elimination arguments in constructive mathematics, many different type structures have been formulated for many different purposes. The aim of this thesis is to present a coherent

hierarchical system of types designed to classify the computation trees of parallel programs in a way that allows us to separate the form and content of a program and to isolate its structural and computational components and to develop an operational semantics for such programs that distinguishes between their metamathematical and arithmetical properties. The separation should be such that both the structure of a program and the actions it defines can be described by arithmetical relationships that are definable in any (necessarily incomplete) first order axiomatization of the natural numbers. We work with a classical class of assignment-based parallel programs as studied (from a syntactic point of view) in HOARE [1972], OWICKI [1975], APT [1981, 1984], et al. By extending the nonstandard analysis of sequential programs in RICHTER and SZABO [1983] to parallel programs, we are able to achieve the desired separation of the structural and operational aspects of the semantics of parallel programs.

In chapter 1 we introduce our programming language PL and summarize the essential facts about Peano arithmetic (PA) required in the thesis. We also describe an effective system of Gödel numbering required for our programs-as-formulas interpretation in chapter 2 and define the recursive functions necessary for the coding aspects of this embedding.

In chapter 2 we introduce program types and define the notion of the execution schemes associated with a program. On the basis of these constructs, we organize the computations of a program as cancellation trees and develop a method of diagrams of program types which is central to our proof of the semantically consistent embeddability of PL in PA. We then show that the next value relation of any program in PL is definable in PA and characterize the partial correctness of deadlock-free parallel programs relative to "finite" choice sequences.

In chapter 3 we introduce the notion of nonstandard cancellation trees and prove that the associated (nonstandard extension of the) "intuitionistic" fan defined by such a tree is an internal star-finite object of the nonstandard universe containing the fixed nonstandard model *N of PA used in this thesis. We define the class CT of complexity types obtained by partitioning the class of program types and prove that CT is a partially ordered distributive monoid with identity. The partial ordering is chosen to reflect the degree of parallelism, i.e., the degree of nondeterminism, contained in a given program.

CHAPTER 1

PEANO ARITHMETIC AND PARALLEL PROGRAMS

1.1. A language and an axiom system for PA

This thesis is based on a first order system PA of Peano arithmetic and a class of parallel programs PL strong enough to compute all partial recursive functions. The fundamental data types used in our work is "the" standard model N of PA, together with a fixed nonstandard extension *N of N. The choice of a particular presentation of PA is irrelevant, but the precise formulations of the first order formulas involved are important in the definitions of programs and their executions. We therefore provide the necessary details. For further information on PA and the representation of recursive functions in PA we refer to MENDELSON [1979]. Formulas of PA will serve both as input and output conditions on computations of programs and as "programs" in their own right under the interpretation presented in chapter 2.

1.1.1. Alphabet:

(a) Constant symbol: 0.

(b) Function symbols: S, pred, +, and •.

(c) Variables: v1, v2, v3, ...

(d) Relation symbols: =, <.

(e) Logical symbols: $\wedge$, $\neg$, $\exists$.

(f) Grammatical symbols: (, ",",).

1.1.2. Terms:

(a) 0 is a term.

(b) v1, v2, v3, ... are terms.

(c) If t is a term, then S(t) and pred(t) are terms.

(d) If t and t' are terms, then (t + t') and (t • t') are terms.

1.1.3. Atomic formulas:

(a) If t and t' are terms, then (t = t') is an atomic formula.

(b) If t and t' are terms, then (t < t') is an atomic formula.

(c) **TRUE** is an atomic formula.

1.1.4. Formulas:

(a) Every atomic formula is a formula.

(b) If A is a formula, then ¬A is a formula.

(c) If A and B are formulas, then (A ∧ B) is a formula.

(d) If A is a formula and x is a variable, then (∃x)A is a formula.

1.1.5. Abbreviations:

(a) $(A \lor B) \equiv \neg(\neg A \land \neg B)$.

(b) $(A \Rightarrow B) \equiv (\neg A \lor B)$.

(c) $(\forall x)A \equiv \neg(\exists x)\neg A$.

(d) $(\forall x<t)A \equiv (\forall x)((x < t) \Rightarrow A)$.

(e) $(\exists x<t)A \equiv (\exists x)((x < t) \land A)$.

(f) $(t \leq t') \equiv ((t < t') \lor (t = t'))$.

(g) **FALSE** $\equiv \neg$**TRUE**

(h) $S^0(0) = 0$.

(i) $S^{n+1}(0) = S(S^n(0))$.

1.1.6. Mathematical axioms:

(a) $\neg(0 = S(x))$.

(b) $((S(x) = S(y)) \Rightarrow (x = y))$.

(c) $(pred(0) = 0)$.

(d) $(pred(S(x)) = x)$.

(e) $\neg(x < 0)$.

(f) $((x < S(y)) \Leftrightarrow ((x < y) \lor (x = y)))$.

(g) $((x < y) \lor (x = y) \lor (y < x))$.

(h) $((x + 0) = x)$.

(i) $((x + S(y)) = S(x + y))$.

(j) $((x \cdot 0) = x)$.

(k) $((x \cdot S(y)) = ((x \cdot y) + x))$.

(g) If A is a formula, then $((A[x/0] \land (\forall x)(A \Rightarrow A[x/S(x)])) \Rightarrow A)$ is an axiom.

1.1.7. Logical axioms:

(a) $(A \Rightarrow (B \Rightarrow A))$.

(b) $((A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C)))$.

(c) $((\neg A \Rightarrow \neg B) \Rightarrow (B \Rightarrow A))$.

(d) $((A \wedge B) \Rightarrow A)$.

(e) $((A \wedge B) \Rightarrow B)$.

(f) $((A \Rightarrow B) \Rightarrow ((A \Rightarrow C) \Rightarrow (A \Rightarrow (B \wedge C))))$.

(g) $(A \Rightarrow (A \vee B))$.

(h) $(B \Rightarrow (A \vee B))$.

(j) $((A \Rightarrow B) \Rightarrow ((C \Rightarrow B) \Rightarrow ((A \vee C) \Rightarrow B)))$.

(k) $((\forall x)A(x) \Rightarrow A[x/t])$ if t is substitutable for x in A.

(l) $(A[x/t] \Rightarrow (\exists x)A(x))$ if t is substitutable for x in A.

(m) **TRUE**.

1.1.8. Equality axioms:

(a) $(t = t)$.

(b) $((t = t') \Rightarrow (t' = t))$.

(c) $(((t = t') \wedge (t' = t'')) \Rightarrow (t = t''))$.

(d) $(((t = s) \wedge (t' = s')) \Rightarrow ((t = t') \Rightarrow (s = s')))$.

(e) $(((t = s) \wedge (t' = s')) \Rightarrow ((t < t') \Rightarrow (s < s')))$.

(f) $((t = t') \Rightarrow (S(t) = S(t')))$.

(g) $((t = t') \Rightarrow (pred(t) = pred(t')))$.

(h) $(((t = t') \wedge (s = s')) \Rightarrow ((t + s) = (t' + s')))$.

(i) $(((t = t') \wedge (s = s')) \Rightarrow ((t \cdot s) = (t' \cdot s')))$.

1.1.9. Rules of inference:

(a) $A, (A \Rightarrow B) \vdash B$.

(b) If x is not free in A, then $(A \Rightarrow B(x)) \vdash (A \Rightarrow (\forall y)A(y))$.

(c) If x is not free in A, then $(A(x) \Rightarrow B) \vdash ((\exists y)A(y) \Rightarrow B)$.

1.1.10. If A is a formula, x a variable occurring free in A and t is a term containing no bound variable of A, then A[x/t] is the formula obtained from A by replacing x by t in A.

1.1.11. If A is a formula and x1, ..., xn are variables, then A(x1,...,xn) expresses the fact that the free variables are among x1, ..., xn.

1.1.12. By $PA \vdash A$ we mean that the formula A is formally provable in PA.

1.1.13. We usually omit outermost parentheses from terms and formulas and omit certain inner parentheses by assuming associativity to the left. In order to increase the readability of abbreviated formulas, we also introduce the following ranking of the arithmetical, relational, and logical operators of PA which allows us to omit further parentheses whenever convenient: The symbols S and pred have rank 6, the symbols + and $\cdot$ have rank 4, the symbols $=$, $<$, and $\leq$ have rank 3, the symbols $\neg$, $(\exists x)$, $(\forall x)$, $(\exists x<t)$, and $(\forall x<t)$ have rank 2, and the symbols $\wedge$, $\vee$, and $\Rightarrow$ have rank 1. For bracketing purposes we agree that symbols of higher rank have precedence over symbols of lower rank.

1.1.14. If N is "the" standard model of PA and a: {v1,v2,...} -> N is a valuation of the variables of PA in N, then B[a] is the sentence obtained from the formula B by replacing all free occurrences of the variables vi in B by the term $S^{a(vi)}(0)$. Similarly, t[a] denotes the variable-free term obtained from a term t by the same type of replacement.

1.1.15. The letters A, B, C, and D range over formulas and x, y, z range over variables.

1.2. Recursive functions and relations

We use Kleene's axiomatization and define the class of **recursive functions** to be the smallest class of functions on the standard model N of PA which contains the following three types of initial functions and is closed under three rules of construction:

1.2.1. Initial functions:

(a) The successor function $S : N \rightarrow N$.

(b) The zero function $Z : N \rightarrow N$.

(c) The projection functions $U^n_i : N^n \rightarrow N$.

1.2.2. Rules of construction:

(a) **(Primitive recursion).** If g(x1,...,xn) and h(y,z,x1,...,xn) are recursive functions, then the function f(w,x1,...,xn) defined by

f(0,x1,...,xn) = g(x1,...,xn)

f(m+1,x1,...,xn) = h(f(m,x1,...,xn),m,x1,...,xn)

is recursive.

(b) **(Composition).** If g_0(x1,...,xn), ..., g_k(x1,...,xn) and h(y0,...,yk) are recursive functions, then the function f(x1,...,xn) defined by:

f(x1,...,xn) = h(g_0(x1,...,xn),...,g_k(x1,...,xn))

is recursive.

(c) (**Minimization**). If $g(y,x1,\ldots,xn)$ is a total-recursive function, then the function $f(x1,\ldots,xn)$ defined by:

$f(x1,\ldots,xn)$ = the least y such that $g(y,x1,\ldots,xn) = 0$

if such a y exists is recursive, where it is understood that $f(x1,\ldots xn)$ is undefined if there is no such y.

We need the following additional recursion-theoretic concepts:

1.2.3. A relation $R \subseteq N^n$ is **recursive** if the function $f(R) : N^n \rightarrow N$ defined by

$$\begin{aligned} f(R)(x1,\ldots,xn) &= 0 \text{ if } (x1,\ldots,xn) \in R \\ &= 1 \text{ if } \neg((x1,\ldots,xn) \in R) \end{aligned}$$

is recursive.

1.2.4. An n-ary relation R on N is **definable** in PA if there exists a formula $A(x1,\ldots,xn)$ in PA such that

$N \models A(x1,\ldots,xn)[x1/a1,\ldots,xn/an]$ iff $(a1,\ldots,an) \in R$.

1.2.5. An n-ary relation R on N is **representable** in PA if there exists a formula $A(x1,\ldots,xn)$ in PA such that

$(a1,\ldots,an) \in R$ implies $PA \vdash A(S^{a1}(0),\ldots,S^{an}(0))$, and

$(a1,\ldots,an) \notin R$ implies $PA \vdash \neg A(S^{a1}(0),\ldots,S^{an}(0))$.

1.2.6. A function $f: N^n \to N$ is representable in PA if there exists a formula $A(x1,\ldots,xn,x(n+1))$ in PA such that

(a) if $f(a1,\ldots,an) = a$, then $PA \vdash A(S^{a1}(0),\ldots,S^{an}(0),S^{a}(0))$

(b) $PA \vdash (\exists x(n+1))A(x1,\ldots,xn,x(n+1))$.

We note that since there are only countably many formulas in the language PA there are only countably many of the uncountably many functions and relations on N which are definable or representable in PA. It is clear that representable relations are definable, but the converse is false. The relevant facts about representable functions and relations for this thesis are the following:

1.2.7. (MENDELSON [1979]) Every recursive relation is representable in PA.

1.2.8. (MENDELSON [1979]) Every recursive function is representable in PA.

The concepts of definability and representability are easily extended to non-arithmetical objects by embedding such objects faithfully in N. For this purpose we introduce the well-known Goedel function G which serves to code finite sequences and finite sequences of finite sequences of natural numbers (etc.) as natural numbers.

1.2.9. Let Words(N) be the union of the sets N^n, $n \in N$, and define

G : Words(N) $\rightarrow$ N by the equations

$G(e1,\ldots,en) = 2^{e1+1} \cdot 3^{e2+1} \cdot \ldots \cdot p_n^{e(n)+1}$, and

G : Words(Words(N)) $\rightarrow$ N by the equations

$G(w1,\ldots,wm) = G(G(w1),\ldots,G(wm))$, etc.

It is easily shown that G is injective and effectively calculable. The sets Words(N), Words(Words(N)), etc. are therefore faithfully embedded in N and we can identify them with their images in N whenever convenient.

1.2.10. For an arbitrary subset $S \subseteq$ Words(N x N), we introduce three special functions

(a) $\tau = \tau(S)$: Words(N x N) $\rightarrow$ (N x N)

(b) $\lambda = \lambda(S)$: Words(N x N) $\rightarrow$ N

(c) $\rho = \rho(S)$: Words(N x N) $\rightarrow$ N,

defined on words $e = (e1,\ldots,en)$, with $ei = (\lambda(i),\rho(i))$, as follows:

(a') $\tau(e)$	$= en$	if $e \in S$
	$= (0,0)$	if $e \notin S$.
(b') $\lambda(e)$	$= \lambda(en)$	if $e \in S$
	$= 0$	if $e \notin S$.
(c') $\rho(e)$	$= \rho(en)$	if $e \in S$
	$= 0$	if $e \notin S$.

If S is a decidable subset of Words($N \times N$), then the functions $\tau(S)$, $\lambda(S)$ and $\rho(S)$ are clearly effective and, relative to the effective embedding of Words($N \times N$) in N, yield three effectively calculable functions $\tau : N \rightarrow (N \times N)$, $\lambda : N \rightarrow N$, and $\rho : N \rightarrow N$. Church's thesis asserts that the effective arithmetical functions are precisely the recursive ones. Hence under the assumption of Church's thesis, τ, λ, and ρ are recursive, and by 1.2.8, are therefore representable in PA.

1.3. The language PL

The class of programs considered in this thesis is similar to the class of programs studied in OWICKI [1975], OWICKI and GRIES [1976], and APT [1981], restricted to successor and predecessor arithmetic as in RICHTER and SZABO [1983]. The language is slightly weaker than that in OWICKI and GRIES [1976] because of restrictions on the occurrences of await statements, but is stronger than that in APT [1981] because of the broader class of await statements allowed. Programs are constructed inductively as certain types of "program statements" from atomic statements as follows:

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

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

1.3.3. If P1, ..., Pn are program statements, then **compose**(P1,...,Pn) is a program statement.

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

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

1.3.6. If P1, ..., Pn are program statements, then **parallel**(P1,...,Pn) is a program statement.

1.3.7. If C(x1,...,xn) is a quantifier-free formula of PA, P1 is a program statement constructed by means of 1.3.1-1.3.5 and x1, ..., xn are among the variables of P1, then **await**(C,P1) is a program statement.

The following program statements are programs:

1.3.8. The null statement is a program.

1.3.9. All assignment statements [x/v] are programs.

1.3.10. If P1, ..., Pn are not await statements or compose statements, then **compose**(P1,...,Pn) is a program.

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

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

1.3.13. If P1, ..., Pn are programs or await statements, then **parallel**(P1,...,Pn) is a program.

The condition in 1.3.10 that P1 is not a compose statement amounts to assuming that composition is associative. We also assume that the program **while**(C,**while**(D,P1)) has the same meaning as **while**(C∧D,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.

CHAPTER 2

PROGRAMS AS FORMULAS

In this chapter we introduce a system of abstract objects called types which serves to classify the programs of PL according to the nesting and frequency of the various programming constructs involved in their definition. The numerical codes of these types under the function G defined in 1.2.9 have no particular significance. We combine types and programs to formulate the central concept of an execution scheme. All inductive definitions and constructions in this thesis are based on this concept. With the help of this structure we develop a new operational semantics for parallel programs and organize the computations of a program as finitely branching trees whose properties are analogous to those found in the literature, e.g., in APT [1981]. By interpreting programs as formulas we prove that the input-output relations determined by the parallel programs considered in this thesis are definable in PA. The programs-as-formulas interpretation is related to a similar, but considerably more elementary interpretation of sequential programs as arithmetical formulas in CSIRMAZ [1984]. But the non-deterministic nature of parallel programs represents a major hurdle in the extension of sequential methods. By using program types to separate the structural and arithmetical properties of the computations of a program we are able to prove that cancellation trees are "isomorphic to" intuitionistic fans whose finite choice sequences, i.e., eventually constant maximal paths, characterize the partial correctness of the program involved.

2.1. The type structure

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

2.1.1. Program types

(a) The natural numbers 1,2, ... are atomic types.

(b) Every atomic type is a type.

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

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

(e) If t1, ..., tn are types, then (4,(t1,...,tn)) and (5,(t1,...,tn)) are types.

2.1.2. The type t(P) of a program statement P

(a) t(**null**) = 1.

(b) t([x/v]) = 2, 3, ...

(c) t(**while**(C,P1)) = (1,t), where t(P1) = t.

(d) t(**await**(C,P1)) = (2,t), where t(P1) = t.

(e) t(**if**(C,P1,P2)) = (3,(t1,t2)), where t(P1) = t1 and t(P2) = t2.

(f) t(**compose**(P1,...,Pn)) = (4,(t1,...,tn)), where t(Pi) = ti.

(g) t(**parallel**(P1,...,Pn)) = (5,(t1,...,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.

2.1.3. 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. By 1.2.9 we can think of such types t as natural numbers G(t), whenever convenient. They 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.

2.1.4. Execution schemes

We distinguish between tagged and untagged **execution schemes** determined by the different 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,...,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.

(a) The execution scheme of **null** is **null**.

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

(c) The execution schemes of [x/v] are [x/v] and **null**.

(d) The execution schemes of t:[x/v] are t:[x/v] and the execution schemes of t:**null**.

(e) 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.

(f) 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.

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

(h) 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.

(i) 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**.

(j) 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**.

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

(l) 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.

(m) 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.

(n) The execution schemes of t1,...,t(n-1),tn:P are all schemes of the form t1,...,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 proposed programs-as-formulas interpretation:

2.1.5. Lemma. Each program P $\in$ PL determines only finitely many distinct execution schemes S1, ..., 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 (b) 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 (i)), S2 = (1,2):[x/x+1] (by (i)), S3 = (1,2):**null** (by (d)), and S4 = **null** (by (i)), and the execution schemes of P2 = **compose**([x/x+1],[y/y+1]) are S1 = P2 (by (e)), S2 = (4,(2,1)):[y/y+1] (by (e)), S3 = (4,(2,1)):**null** (by (d)), S4 = [x/x+1] (by (b)), and S5 = **null** (by (c)).

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:

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

For a correct coding of tagged and untagged execution schemes in section 2.4, 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 numerically both types of execution schemes.

2.2. 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 Tr(P,**a**) as a **cancellation tree.** The root of Tr(P,**a**) is the pair (P,**a**) and every other node is computed from an earlier node by the next node function σ.

2.2.1. Next nodes

The nodes of the tree Tr(P,**a**) are defined inductively from the root (P,**a**) as follows:

(a) σ(**null**,**b**) is undefined

σ(t:**null**,**b**) = (S,**b**), where u = t(S) and t = (1,u) or (4,(u,1)).

(b) σ([x/x],**b**) = (**null**,**b**)

σ(t:[x/x],**b**) = (t:**null**,**b**),

(c) σ([xi/xi + 1],(..,xi,..)) = (**null**,(..,xi + 1,..))

σ(t:[xi/xi + 1],(..,xi,..)) = (t:**null**,(..,xi + 1,..)).

(d) σ([xi/xi - 1],(..,xi,..)) = (**null**,(..,xi - 1,..))

σ(t:[xi/xi - 1],(..,xi,..)) = (t:**null**,(..,xi - 1,..)).

(e) σ(**compose**(P1,P2),**b**) = (t:P2,**b**),

where t = t(**compose**(P1,**null**))

σ(**compose**(P1,**null**),**b**) = (P1,**b**)

σ(u:**compose**(P1,P2),**b**) = (ut:P2,**b**),

where t = t(**compose**(P1,**null**))

σ(u:**compose**(P1,**null**),**b**) = (u:P1,**b**).

(f) σ(**while**(C,P1),**b**) = (t:P1,**b**) if C[**b**] is true

= (**null**,**b**) if C[**b**] is false

where t = t(**while**(C,P1))

σ(u:**while**(C,P1),**b**) = (ut:P1,**b**) if C[**b**] is true

= (u:**null**,**b**) if C[**b**] is false

where t = t(**while**(C,P1)).

(g) σ(**if**(C,P1,P2),**b**) = (P1,**b**) if C[**b**] is true

= (P2,**b**) if C[**b**] is false

σ(u:**if**(C,P1,P2),**b**) = (u:P1,**b**) if C[**b**] is true

= (u:P2,**b**) if C[**b**] is false.

(h) σ(P1//P2,**b**) = (σ1(P1//P2,**b**),σ2(P1//P2,**b**))

σ(P1//(P2//P3),**b**) = σ(P1//P2//P3,**b**)

σ((P1//P2)//P3,**b**) = σ(P1//P2//P3,**b**)

σ(t:(P1//P2),**b**) = (σ1(t:(P1//P2),**b**),(σ2(t:(P1/P2),**b**)))

σ(t:(P1//(P2//P3)),**b**) = σ(t:(P1//P2//P3),**b**)

σ(t:((P1//P2)//P3),**b**) = σ(t:(P1//P2//P3),**b**)

σ(t:(**null**//**null**),**b**) = (S,**b**),

where u = t(S) and t = (1,u) or (4,(u,1)).

(i) σl(**null**//P,**b**) = undefined

If P = **await**(C,P1), u = t(S), and t = (1,u) or (4,(u,1)), then

σl(t:**null**//P,**b**) = (S//P,**b**) if C[**b**] is false

= (S//P1,**b**) if C[**b**] is true;

if P ≠ **await**(C,P1), u = t(S), and t = (4,(u,1)), then

σl(t:**null**//P,**b**) = (S//P,**b**);

if P = **null** or t':**null**, u = t(S), and t = (1,u), then

σl(t:**null**//P,**b**) = (S//P,**b**);

σl(t:**null**//P,**b**) is undefined otherwise.

(j) σl([x/v]//P,**b**) = (**null**//P,**b**[x/v])

σl((t:[x/v])//P,**b**) = ((t:**null**)//P,**b**[x/v]).

(k) σl(**compose**(P1,P2)//P,**b**) = ((t:P2)//P,**b**),

where t = t(**compose**(P1,**null**))

σl(**compose**(P1,**null**)//P,**b**) = (P1//P,**b**),

σl((u:**compose**(P1,P2))//P,**b**) = ((ut:P2)//P,**b**),

where t = t(**compose**(P1,**null**)).

σl((u:**compose**(P1,**null**))//P,**b**) = ((u:P1)//P,**b**).

(l) σl(**if**(C,P1,P2)//P,**b**) = (P1//P,**b**) if C[**b**] is true

= (P2//P,**b**) if C[**b**] is false

σl((t:**if**(C,P1,P2))//P,**b**) = ((t:P1)//P,**b**) if C[**b**] is true

= ((t:P2)//P,**b**) if C[**b**] is false.

(m) $\sigma1$(**while**(C,P1)//P,**b**) = ((t:P1)//P,**b**) if C[**b**] is true

= (**null**//P,**b**) if C[**b**] is false,

where t = t(**while**(C,P1))

$\sigma1$((u:**while**(C,P1))//P,**b**) = ((ut:P1)//P,**b**) if C[**b**] is true

= ((u:**null**)//P,**b**) if C[**b**] is false

where t = t(**while**(C,P1)),

(n) $\sigma1$(**await**(C,P1)//P,**b**) = (P1//P,**b**) if C[**b**] is true

= undefined if C[**b**] is false.

The definition of $\sigma2$ 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 σ(P1//P2,**b**) may give rise to only a single node if $\sigma1$(P1//P2,**b**) or $\sigma2$(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 $\sigma1$(P1//P2,**b**) to the left of $\sigma2$(P1//P2,**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 require therefore 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[**b**] as indivisible operations. This assumption agrees with the operational interpretation of parallel programs in OWICKI [1975]. A similar motivation also underlies the description of the semantics for parallel programs in APT [1984] based on the work of Hennessy and Plotkin.

2.2.2. Example. Let P = **parallel**(P1,P2), where

P1 = **while**(**TRUE**,[x/x+1]) and

P2 = **while**(**TRUE**,**compose**([x/x+1],[y/y+1])).

In order to list the execution schemes of P we must, according to 2.1.4, compute the execution schemes of P1 and P2. The execution schemes of P1 and their types are the following:

S1 = **while**(**TRUE**,[x/x+1])   t1 = (1,2)

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

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

S4 = **null**   t4 = 1

and the execution schemes of P2 are their types are

```
T1 = while(TRUE,compose([x/x+1],[y/y+1]))   t4 = (1,(4,(3,4)))
T2 = t4:compose([x/x+1],[y/y+1])            t5 = ((6,t4),(4,(3,4)))
T3 = t4,t6:[y/y+1]                          t7 = ((6,t4,(4,(3,1))),4)
T4 = t4,t6:null                             t8 = ((6,t4,t6),1)
T5 = t4:[x/x+1]                             t9 = ((6,t4),3)
T6 = t4:null                                t10 = ((6,t4),1)
T7 = null                                   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 2.1.3 are met.

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

```
(S1//T1,(0,0)) -> (S2//T1,(0,0)) ->
(S3//T1,(1,0)) -> (S3//T2,(1,0)) ->
(S3//T3,(1,0)) -> (S3//T4,(1,1)) ->
(S3//T5,(1,1)) -> (S3//T6,(2,1)) ->
(S3//T1,(2,1)) -> (S1//T1,(2,1)) -> ... ♦
```

2.3. Embedding PL in PA

The purpose of this section is to show that the computational behaviour of a program P ∈ PL can be characterized by formulas Φ(P) of PA in which we can express the idea of a sequence **a**1 ∈ N^n, ..., **a**n ∈ N^n of values consisting of successive data components of a cancellation tree of P. For this purpose we represent the nodes (E,**b**) of all trees Tr(P,**a**) by (G(u),G(**b**)), where E is either a tagged or untagged execution scheme of P, and u is the type of E as defined in 2.1.2 and 2.1.6. We then let the set S in 1.2.10 be the set of all finite sequences e = (e1,...,en) of such pairs with the property that e1 = (G(t(P)),G(**a**)), for some **a** ∈ N^n, and such that if e(i+1) ∈ e, then e(i+1) corresponds to a next node of e(i) in Tr(P,**a**). It is clear from the algorithmic nature of the next node function that S is a decidable set. It therefore follows from the remarks in 1.2.10 that the functions τ(S), λ(S), and ρ(S) are representable in PA. We call the elements of S the **finite paths** determined by P, so that τ(e) represents the last node en of the path e, and λ(e) = λ(n) codes the program component and ρ(e) = ρ(n) the data component of en. We let π be a new variable ranging over the (Gödel numbers of the) elements of S. 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 codes of the types of the obvious "next execution schemes", according to 2.1.4, determined by a given scheme. We omit the definition of next execution schemes since it

is implied by the clauses of 2.1.4. 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 a. The variables z, z', ri and si are assumed to be distinct. We put r = (r1,...,rn) and s = (s1,...,sn) and we simplify the notation further by writing (s=r) in place of (s1=r1 ∧ ... ∧ sn = rn) and, ambiguously, in place of (s1=r1 ∧ ... ∧ s(i-1)=r(i-1) ∧ s(i+1)=r(i+1) ∧ ... ∧ sn=rn) if the intended conjunction is clear from the context.

The formula Φ(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, ..., rn and output variables s1, ..., sn. The construction of Φ(P) involves four steps:

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

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

(c) The specification of the "diagram" Diag(P) of P determined by t1, ..., tn.

(d) The definition of Φ(P) from the data accumulated in (a)-(c).

We illustrate $\Phi(P)$ by several examples. Among the cases considered are those required for the induction basis of 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.

2.3.1. 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 $\Phi(P) = \phi(t1) = (z1{=}t1 \wedge z1'{=}t1 \wedge r{=}\rho(1){=}a \wedge s{=}r)$. The same formula is used for any other trivial program whose only atomic component is **null**. ♦

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

S1 = P	t1 = t(S1) = 2
S2 = **null**	t2 = t(S2) = 1, Diag(P) = t1 -> 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 2.2.1, of the node with program component Si.

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

2.3.3. 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	t1 = (4,(2,3))
S2 = t0:[y/y+1]	t2 = ((6,t0),3)
S3 = t0:**null**	t3 = ((6,t0),1)
S4 = [x/x+1]	t4 = 2
S5 = **null**	t5 = 1,

$$Diag(P) = t1 \rightarrow t2 \rightarrow t3 \rightarrow t4 \rightarrow t5.$$

For each arrow (ti -> tj) in 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 $\phi(i,j)$:

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

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

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

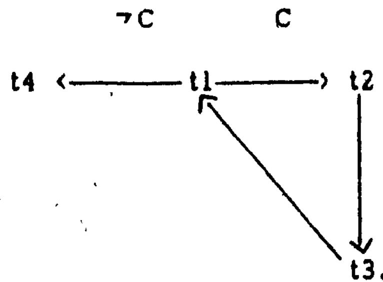
$$\phi(4,5) = (z=t4 \wedge z'=t5 \wedge \lambda(4)=t4 \wedge r=\rho(4) \wedge s1=r1+1 \wedge s2=r2).$$

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

2.3.4. 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**	$t4 = 1,$

Diag(P) =



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

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

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

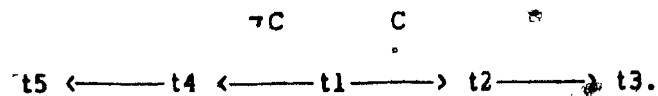
$\phi(3,1) = (z=t3 \wedge z'=t1 \wedge \tau(\pi)=(\lambda(\pi),\rho(\pi)) \wedge \lambda(\pi)=t3 \wedge \rho(\pi)=r \wedge s=r).$

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

2.3.5. 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	t1 = t(S1) = (3,(2,3))
S2 = [x/x+1]	t2 = t(S2) = 2
S3 = null	t3 = t(S3) = 1
S4 = [y/y+1]	t4 = t(S4) = 3
S5 = null	t5 = t(S5) = 1,

Diag(P) =



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

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

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

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

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

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

$S1 = P$ $\qquad t1 = t(S1) = (5,((2,3),1))$

$S2 = \textbf{parallel}([x/x+1],\textbf{null})$ $\qquad t2 = t(S2) = (5,(3,1))$

$S3 = \textbf{parallel}(\textbf{null},\textbf{null})$ $\qquad t3 = t(S3) = (5,(1,1))$,

Diag(P) =

$$t1 \xrightarrow{C} t2 \longrightarrow t3.$$

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

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

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

2.3.7. Example. If P = **parallel(await(C,[x/x+1]),[y/y+1])**, then the execution schemes of P and their types are

S1 = P	t1 = t(S1) = (5,((2,3),4))
S2 = [x/x+1]//[y/y+1]	t2 = t(S2) = (5,(3,4))
S3 = **null**//[y/y+1]	t3 = t(S3) = (5,(1,4))
S4 = **null**//**null**	t4 = t(S4) = (5,(1,1))
S5 = **await**(C,[x/x+1])//**null**	t5 = t(S5) = (5,((2,3),1))
S6 = [x/x+1]//**null**	t6 = t(S6) = (5,(3,1))
S7 = **null**//**null**	t7 = t(S7) = (5,(1,1)),

Diag(P)=

$$t7 \longleftarrow t6 \overset{C}{\longleftarrow} t5 \longleftarrow t1 \overset{C}{\longrightarrow} t2 \longrightarrow t3 \longrightarrow t4$$

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

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

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

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

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

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

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

2.3.8. Example. If P = **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 = **null**//[y/y+1]	t2 = t(S2) = (5,(1,4))
S3 = **null**//**null**	t3 = t(S3) = (5,(1,1))
S4 = [x/x+1]//**null**	t4 = t(S4) = (5,(3,1))
S5 = **null**//**null**	t5 = t(S5) = (5,(1,1)),

Diag(P)= t5 <- t4 <- t1 -> t2 -> t3.

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

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

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

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

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

The general construction is obtained by applying the induction hypothesis to component programs and modifying the above examples in the appropriate way. We describe the case of a program of the form P = **if**(C,P1,P2) in relation to our example 2.3.5. The modifications required in the remaining cases are similar.

Suppose that the execution schemes of P1 are S2, ..., Sp, with respective types t2, ..., tp, that the execution schemes of P2 are S(p+1), ..., S(p+q), the respective types t(p+1), ..., t(p+q), and that Diag(P1) and Diag(P2) are the diagrams of P1 and P2 and that Φ(P1) and Φ(P2) are given. Then we let S1 = P, t1 = t(P), and let Diag(P) be the diagram

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

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

$$\Phi(P) = \phi(1,2) \vee \phi(1,p+1) \vee \Phi(P1) \vee \Phi(P2),$$

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

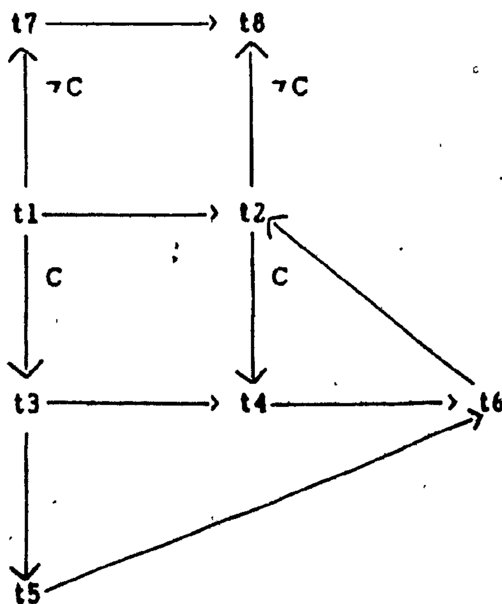
$$\phi(1,p+1) = (z{=}t1 \wedge z'{=}t2 \wedge \lambda(1){=}t1 \wedge \rho(1){=}r \wedge \neg C[x/r] \wedge s{=}r).$$

This completes the definition of Φ(P). The following example illustrates the utility of a non-trivial diagram in the construction of Φ(P):

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

	t0 = t(**while**(C,[x/x+1]))
S1 = P	t1 = t(S1)
S2 = **while**(C,[x/x+1])//**null**	t2 = t(S2)
S3 = t0:[x/x+1]//[y/y+1]	t3 = t(S3)
S4 = t0:[x/x+1]//**null**	t4 = t(S4)
S5 = t0:**null**//[y/y+1]	t5 = t(S5)
S6 = t0:**null**//**null**	t6 = t(S6)
S7 = **null**//[y/y+1]	t7 = t(S7)
S8 = **null**//**null**	t8 = t(S8),

Diag(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 $\Phi(P)$:

2.3.10. 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 $N \models \phi(i,j)[a]$. ♦

2.3.11. 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,...,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,π/e,r/bi,s/bj]. ♦

Since every 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], ..., [ap] = [cp,dp] be these assignments. Then we can construct the formulas $\Phi(P)$[z/c1,z'/d1], ..., $\Phi(P)$[z/cp,z'/dp] in which the variables z and z' are replaced by the terms $S^{ci}(0)$ and $S^{di}(0)$ (in the notation of 1.1.5) 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 w 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 (**b**,**b'**) $\in N^n \times N^n$ with the property that (Q,**b**), (Q',**b'**) belong to Tr(P,**a**) and (Q',**b'**) is σ(Q,**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(Φ(P)) by equivalent formulas not involving these symbols and obtain a formula D'(Φ(P)) $\in$ PA with the property that

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

The desired result therefore follows from 2.3.11.

2.3.12. Definability theorem. For any program P $\in$ PL, the next value relation NVR of P is definable in PA. ♦

2.4. 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 $P \in PL$ is a parallel program with an associated formula $\Phi(P) \in PA$ and $Tr(P,\mathbf{a})$ is the cancellation tree generated by P and the initial input $\mathbf{a} \in N^n$, we say that P **terminates at a** if $Tr(P,\mathbf{a})$ is finite and all leaves of $Tr(P,\mathbf{a})$ are real. We further say that P is **partially correct in N** with respect to an input condition $A \in PA$ and output condition $B \in PA$ if A is quantifier-free whose variables are among the input variables r of $\Phi(P)$ and B is quantifier-free whose variables are among the output variables s of $\Phi(P)$, and if $N \models A[r/\mathbf{a}]$ implies that $N \models B[s/\mathbf{b}]$ for all $\mathbf{a} \in N^n$ for which P terminates and for all data components $\mathbf{b}$ of the real leaves of $Tr(P,\mathbf{a})$. We use the usual notation and write $N \models A\{P\}B$ to express the fact that P is partially correct with respect to A and B. It is an exercise to verify that $N \models A\{P\}B$ if and only if it is partially correct in the traditional sense, as defined in APT [1981, 1984], 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 Tr(P,a), because of the nature of P or because of the specific arithmetical conditions induced by the initial input a, then the cancellation trees of P determine intuitionistic **fans** whose choice sequences characterize the partial correctness of P. We recall from DUMMETT [1977] 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.

2.4.1. The construction of Fan(P,a)

(a) The root of Fan(P,a) is (0,w1,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.

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

(c) We replace every node of the form (t1,...,tn:Q,b) by (w2,w1,b), where w1 = t(Q) and where w2 is the Gödel number of the sequence of tags (t1,...,tn) of Q.

(d) To every node (w2,w1,b) of Fan(P,a) corresponding to a real leaf of Tr(P,a) we append an infinite number of copies of (w2,w1,b).

If P is a sequential program, then Fan(P,a) is a run of P in the sense of CSIRMAZ [1981, 1984] in which codes of sequences of tags correspond to numerical labels of programming instructions. It is clear from Lemma 2.3.10 and Corollary 2.3.11 and from the construction of Fan(P,a) that c' = (w2',w1',b') is a next node of a node c = (w2,w1,b) of Fan(P,a) if and only if there exists a path e = (e1,...,en,e(n+1)) $\in$ Tr(P,a) with the property that en = $(\lambda(n),\rho(n))$ = (w2,w1,b) and

$e(n+1) = (\lambda(n+1),\rho(n+1)) \approx (w2',w1',\mathbf{b}')$ such that

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

where $ti = ((6,w2),w1)$ and $tj = ((6,w2'),w1')$, as defined in 2.1.6. 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 Φ(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 ∈ 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:

2.4.2. 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 (**b1**,...,**bn**) determined by P. ♦

CHAPTER 3

DEFINABLE TREES AND COMPLEXITY TYPES

In this chapter we presuppose familiarity with the basic facts and constructions of nonstandard analysis as found in DAVIS [1977], for example. We assume as given a standard universe U containing the elements of N as individuals and a nonstandard extension W equipped with a faithful embedding $* : U \rightarrow W$. We also assume as given a first order language L with bounded quantifiers which extends PA (with $(\forall x)$ interpreted as $(\forall x \in N)$ and $(\exists x)$ interpreted as $(\exists x \in N)$) and in which we can describe the properties of the standard objects in U, together with a corresponding language *L in which we can express facts about the nonstandard objects in W. The difference between L and *L lies in the fact that the set of constants of L is taken to be U, whereas the set of constants of *L is W. This induces the obvious differences on the sets of terms and formulas of the two languages. Every formula $A \in L$ determines a translation $*A \in *L$ obtained by replacing every standard constant c in A by its image *c. One of the fundamental deductive techniques of nonstandard analysis is based on the fact that a formula $A \in L$ is satisfiable in U if and only if its translation $*A \in *L$ is satisfiable in W. This result is known as the **Transfer Principle**. A further nonstandard property required in this chapter is the fact that although the universe W is not closed under arbitrary subsets, it is closed under **definable subsets**, i.e., if $S \in W$ and $S' \subseteq S$ is definable by a formula $B \in *L$, then $S' \in W$. We call this result the **Definability**

Principle. It is customary to refer to sets $S \in W$ as **internal** sets. Thus the Definability Principle asserts that definable subsets of internal sets are internal. For example, for any $n \in {}^*N$, the interval $[0,n]$ is definable by the formula $(0 \;{}^*\!\leq x) \wedge (x \;{}^*\!\leq n) \in {}^*L$ and is therefore internal. By the Transfer Principle, this interval therefore enjoys all the properties of a standard finite set such as the fact that every subset of a finite set is finite. Because of this property, any internal set which is in internal bijective correspondence with a set of the form $[0,n]$ is called **star-finite.**

In order to be able to extend our coding techniques from finite paths of standard trees to star-finite paths of nonstandard trees it is important that the embedding $* : U \rightarrow W$ preserves finite cartesian products, countable unions, the composition of functions, injectivity, surjectivity, and the usual Boolean operations. Thus the coding functions $G : Words(N) \rightarrow N$ and $G : Words(Words(N)) \rightarrow N$, etc., of 1.2.9 yield injections ${}^*G : {}^*Words(N) \rightarrow {}^*N$ and ${}^*G : {}^*Words(Words(N)) \rightarrow {}^*N$, etc., whose properties will be used for a programs-as-formulas representation in ${}^*L$. In particular, we require the functions

(a) ${}^*\tau_\omega = {}^*\tau(S) : {}^*\,Words(N \times N) \rightarrow {}^*(N \times N)$

(b) ${}^*\lambda = {}^*\lambda(S) : {}^*\,Words(N \times N) \rightarrow {}^*N$

(c) ${}^*\rho = {}^*\rho(S) : {}^*\,Words(N \times N) \rightarrow {}^*N$

to describe the last nodes of star-finite paths in nonstandard trees by equations between terms of ${}^*L$.

In the nonstandard analysis of sequential programs in RICHTER and SZABO [1983], nonstandard bounds on while programs are used to give a star-finite description of properties of programs. The purpose of this chapter is to extend this work to the parallel case. However, the non-deterministic nature of parallel processes requires a considerably more subtle and structured approach to such a star-finite analysis. As in the standard case, the central concept for our work will be that of a program type defined for all programs in a suitable bounded language BPL. Based on this notion, we construct nonstandard cancellation trees $Tr(\mu,P,a)$ in which the number of iterations of while programs and the waiting times in await statements of all programs $P \in BPL$ are uniformly bounded by some $\mu \in {}^*N$. If P contains no while and await statements, then $Tr(\mu,P,a) = Tr(P,a)$. We code the trees $Tr(\mu,P,a)$ as numerical trees, denoted by $Fan(\mu,P,a)$ in view of the analogy with the standard case, and show that for all $a \in N^n$, the trees $Fan(\mu,P,a)$ are definable in $^*L$ and are therefore internal. We then prove that for all $\mu \in {}^*N$, $Fan(\mu,P,a)$ is star-finite. This fact is tantamount to saying that every program $P \in$ BPL terminates, albeit in possibly nonstandard time. We transfer this property to cancellation trees by saying that $Tr(\mu,P,a)$ is star-finite if the associated numerical tree $Fan(\mu,P,a)$ is star-finite. We then

introduce the notion of an operational equivalence between programs in PL ∪ BPL and prove a termination property for deadlock-free programs to the effect that every deadlock-free program in $P \in PL$ is operationally equivalent to a program $\beta(P) \in BPL$. In conclusion, we show that the types of the programs in PL ∪ BPL can be endowed with an algebraic structure that yields a reasonable measure of the complexity of interacting processes.

3.1. Nonstandard cancellation trees

In this section, we develop the specific concepts required for the nonstandard analysis of parallel programs. We convert the programming language PL of 1.3 into a language BPL, whose elements will be called **star-bounded programs**, in which all await and while statements of PL are replaced by **bounded** statements. We then adapt the type structure for PL and construct nonstandard cancellation trees for star-bounded programs. The language BPL is obtained from PL by modifying the **if**, **while** and **await** constructs in 1.3.4, 1.3.5 and 1.3.7 as follows:

3.1.1. If C(x1,...,xn) is a quantifier-free formula D of PA or its translation *D ∈ *L, and P1 and P2 are program statements and x1, ..., xn are the among the variables of P1 and P2, then **if**(C,P1,P2) is a program statement.

3.1.2. If C(x1,...,xn) is a quantifier-free formula D of PA or its translation *D ∈ *L, and μ ∈ *N, and if P1 is a program statement and x1, ..., xn are among the variables of P1, then **while**(μ,C,P1) is a program statement.

3.1.3. If C(x1,...,xn) is a quantifier-free formula D of PA or its translation *D ∈ *L, and μ ∈ *N, and if P1 is a program statement constructed by means of 1.3.1-1.3.3, 3.1.1, and 3.1.2, and if x1, ..., xn are among the variables of P1, then **await**(μ,C,P1) is a program statement.

The language BPL is defined as a subclass of the class of star-bounded program statements by analogy with the definition of PL in 1.3.8-1.3.13. We stipulate that all bounds in the different occurrences of await and while statements in the same program are identical. This restriction is required for the programs-as-formulas interpretation. Since the bounds required in our applications below are always infinite, this restriction is clearly inconsequential.

The intended meaning of **while**(μ,C,P1) is that of a loop in which the program P1 can be iterated precisely μ times if C[**b**] is true in each state **b** produced by an execution of P1.

Programs of the form P = **parallel**(**await**(μ,C,P1),P2), are interpreted similarly to **parallel**(**await**(C,P1),P2)) in 2.2.1 (n) if C[**b**] becomes true at an intermediate value **b** before μ becomes 0, and similarly to **parallel**(P1,P2) if μ = 0 or C[**a**] is true at the initial input **a** $\in N^n$.

3.1.4. Star-finite types

In order to include star-bounded programs in our computational procedures, we adapt the definition of program types to the case of star-bounded while and await statements by substituting the word **type** by **star-finite type** in conditions (a)-(b) and (d)-(e) of 2.1.1, and replacing condition (c) by the following clause:

(c') If t is a star-finite type and $\mu \in {}^*N$, then $(\mu,1,t)$ and $(\mu,2,t)$ are star-finite types.

3.1.5. The type t(P) of a star-bounded program statement P

The definition of the type t(P) of a program P in 2.1.2 is modified as follows to apply to star-bounded while and await statements:

(c') t(**while**(μ,C,P1)) = (μ,1,t)

(d') t(**await**(μ,C,P1)) = (μ,2,t).

3.1.6. Star-bounded execution schemes

In order to apply to star-bounded programs, we modify the definition of execution schemes in 2.1.4 as follows: The syntactic variables P1 and P2 now range over BPL and the type variables t and u range over the types of star-bounded programs as defined in 3.1.5. The wording of clauses (a)-(h) and (l)-(n) remains unchanged, with the formula C in (g) and (h) now referring to a formula $D \in PA$ or to its translation $^*D \in {}^*L$. For each $\mu \in {}^*N$, we redefine clauses (i)-(k) as follows:

(i') The execution schemes of **while**(μ,C,P1) are **while**(μ,C,P1), t:P1, where t = t(**while**(μ-1,C,P1)), the execution schemes of t:P1, and **null**.

(j') The execution schemes of u:**while**(μ,C,P1) are u:**while**(μ,C,P1), ut:P1, where t = t(**while**(μ-1,C,P1)), the execution schemes of ut:P1, and u:**null**.

(k') The execution schemes of **await**(μ,C,P1) are **await**(μ,C,P1) and the execution schemes of P1.

It should be noted that the definition of the execution schemes of a star-bounded program has been chosen so that as in the case of PL, a program determines only finitely many such schemes. This property is essential for the extension of the programs-as-formulas interpretation to BPL..

3.1.7. Example. Let P = **parallel**(P1,P2), where

P1 = **await**(p,**FALSE**,[x/x+1]), p ∈ *N, and

P2 = **while**(q,**TRUE**,[y/y+1]), q ∈ *N.

The execution schemes of P1 and their associated types are the following:

S1 = **await**(p,**FALSE**,[x/x+1])	t1 = t(S1) = (p,2,2)
S2 = [x/x+1]	t2 = t([x/x+1]) = 2
S3 = **null**	t3 = t(**null**) = 1

and the execution schemes of P2 are

T1 = **while**(q,**TRUE**,[y/y+1])	t4 = t(T1) = (q,1,3)
T2 = t5:[y/y+1]	t5 = ((6,(q-1,1,t3)),3)
T3 = t5:**null**	t6 = ((6,(q-1,1,t3)),1)
T4 = **null**	t7 = 1.

The execution schemes of P are therefore of the form **parallel**(Si,Tj), where Si = S1, S2, or S3, and Tj = T1, T2, T3, or T4. ♦

3.1.8. Next nodes

We modify the definition of the next node function σ in 2.2.1 so that it applies to star-bounded programs. As in the case of execution schemes in 3.1.6, the syntactic variables P1, P2, S, and P now range over BPL and the type variables t and u range over the types of star-bounded programs as defined in 3.1.5. The wording of clauses (a)-(e), and (i)-(k) remains unchanged and clauses (f)-(h) and (l)-(n) are modified and supplemented as follows, with the change in (n) incorporated in (h'):

(f') $\sigma(\mathbf{while}(\mu,C,P1),\mathbf{b}) = (t{:}P1,\mathbf{b})$ if $\mu > 0$, $*N \models C[\mathbf{b}]$

$= (\mathbf{null},\mathbf{b})$ if $\mu = 0$ or $*N \not\models C[\mathbf{b}]$

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

$\sigma(u{:}\mathbf{while}(\mu,C,P1),\mathbf{b}) = (ut{:}P1,\mathbf{b})$ if $\mu > 0$, $*N \models C[\mathbf{b}]$

$= (u{:}\mathbf{null},\mathbf{b})$ if $\mu = 0$ or $*N \not\models C[\mathbf{b}]$

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

(g') $\sigma(\mathbf{if}(C,P1,P2),\mathbf{b}) = (P1,\mathbf{b})$ if $*N \models C[\mathbf{b}]$

$= (P2,\mathbf{b})$ if $*N \not\models C[\mathbf{b}]$

$\sigma(t{:}\mathbf{if}(C,P1,P2),\mathbf{b}) = (t{:}P1,\mathbf{b})$ if $*N \models C[\mathbf{b}]$

$= (t{:}P2,\mathbf{b})$ if $*N \not\models C[\mathbf{b}]$.

(h') If P1 is not an await statement, then

$\sigma(P1//P2,\mathbf{b}) = (\sigma 1(P1//P2,\mathbf{b}),\sigma 2(P1//P2,\mathbf{b}))$;

if P1 = **await**(μ,C,P) and $\mu>0$ and $*N \not\models C[\mathbf{b}]$, then

$\sigma(P1//P2,\mathbf{b}) = (\mathbf{await}(\mu-1,C,P)//Q,\mathbf{b}')$, where $\sigma(P2,\mathbf{b}) = (Q,\mathbf{b}')$;

if P1 = **await**(μ,C,P) and $\mu=0$ or $*N \models C[\mathbf{b}]$, then

$\sigma(P1//P2,\mathbf{b}) = ((P//P2,\mathbf{b}),(P1//Q,\mathbf{b}'))$, where $\sigma(P2,\mathbf{b}) = (Q,\mathbf{b}')$;

if P1 = **await**(μ,C,P) and $\mu>0$ and $*N \not\models C[\mathbf{b}]$ and

P2 = **await**(ν,D,Q) and $\nu>0$ and $*N \not\models D[\mathbf{b}]$, then

$\sigma(P1//P2,\mathbf{b}) = (\mathbf{await}(\mu-1,C,P)//\mathbf{await}(\nu-1,D,Q),\mathbf{b})$;

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

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

(i') $\sigma 1(\mathbf{if}(C,P1,P2)//P,\mathbf{b}) = (P1//P,\mathbf{b})$ if $*N \models C[\mathbf{b}]$

$= (P2//P,\mathbf{b})$ if $*N \not\models C[\mathbf{b}]$

$\sigma 1((t{:}\mathbf{if}(C,P1,P2))//P,\mathbf{b}) = ((t{:}P1)//P,\mathbf{b})$ if $*N \models C[\mathbf{b}]$

$= ((t{:}P2)//P,\mathbf{b})$ if $*N \not\models C[\mathbf{b}]$.

(m') $\sigma 1(\textbf{while}(\mu,C,P1)//P,b) = ((t:P1)//P,b)$ if $\mu > 0$, $^*N \models C[b]$

$= (\textbf{null}//P,b)$ if $\mu = 0$ or $^*N \not\models C[b]$

where $t = t(\textbf{while}(\mu-1,C,P1))$.

$\sigma 1((u:\textbf{while}(\mu,C,P1))//P,b) = ((ut:P1)//P,b)$ if $\mu > 0$, $^*N \models C[b]$

$= (u:\textbf{null}//P,b)$ if $\mu = 0$ or $^*N \not\models C[b]$

where $t = t(\textbf{while}(\mu-1,C,P1))$.

This completes the required modification of σ. An induction extends this definition to parallel statements with more than two variables.

3.1.9. Example. Let P = **while**(2,**TRUE**,[x/x+1]) and **a** = 0. Then the cancellation tree Tr(2,P,**a**) is the linearly ordered tree

```
   (while(2,TRUE,[x/x+1]),0)
-> ((1,1,3):[x/x+1],0)
-> ((1,1,3):null,1)
-> (while(1,TRUE,[x/x+1]),1)
-> ((0,1,3):[x/x+1],1)
-> ((0,1,3):null,2)
-> (while(0,TRUE,[x/x+1]),2)
-> (null,2). ♦
```

3.1.10. Example. Let P = **parallel**(P1,P2), where

P1 = **await**(2,FALSE,[x/x+1]) and

P2 = [y/y+1].

The execution schemes of P1 and their associated types are as in 3.1.7, i.e.,

S1 = **await**(2,FALSE,[x/x+1]) t1 = (2,2,3)

S2 = [x/x+1] t2 = t([x/x+1]) = 3

S3 = **null** t3 = t(**null**) = 1

and the execution schemes of P2 and their associated types are the following:

T1 = [y/y+1] t4 = t([y/y+1]) = 4

T2 = **null** t5 = t(**null**) = 1.

The execution schemes of P are therefore of the form **parallel**(Si,Tj), where Si = S1, S2, or S3, and Tj = T1 or T2. If a = (0,0), then Tr(2,P,(0,0)) is the linearly ordered tree

(S1//T1,0,0) -> (S11//T2,0,1) -> (S10//T2,0,1) -> (S2//T2,0,1) -> (S3//T2,1,1),

where S11 = **await**(1,FALSE,[x/x+1]) and S10 = **await**(0,FALSE,[x/x+1]). ♦

As in section 2.2, we define the cancellation tree $Tr(\mu,P,a)$ of a (uniformly) star-bounded program $P \in BPL$ and $a \in {}^*(N^n)$ as a rooted tree determined by the root (P,a) and the next-node function σ described in 3.1.8 above. In order to use our nonstandard tools to analyze the properties of $Tr(\mu,P,a)$, we must first code $Tr(\mu,P,a)$ as an appropriate internal object of our nonstandard universe W. This is achieved by adapting the fan construction of 2.4.1. and the programs-as-formulas construction of 2.3 to the present context.

3.2. The internal-fan construction

As in 2.4.1, we use the values of the control and input and output variables involved in the programs-as-formulas construction to code the program components of the nodes of each cancellation tree Tr(μ,P,a). In this way we obtain a nonstandard numerical tree Fan(μ,P,a) whose set of nodes is isomorphic, via the coding function *G, to a subset of *N. In order to show that Fan(μ,P,a) enjoys the same formal properties as a finite tree, i.e., is star-finite, we prove Fan(μ,P,a) corresponds to a definable subset of *N and is therefore internal.

3.2.1. The formula $\Gamma(P)$

We convert every star-bounded program $P \in BPL$ into an arithmetical formula $\Gamma(P)$ by analogy with the standard case, with the obvious modifications required by the presence of bounds in await and while statements and the fact that the execution schemes of P may no longer form all possible program components of the nodes of the cancellation trees of P. If $(Q,\mathbf{b}) = (\mathbf{await}(\mu,C,P1)//\mathbf{null},\mathbf{b})$ is a node of $Tr(\mu,P,\mathbf{a})$, for example, and $\mu>0$ and $C[x/\mathbf{b}]$ if false, then the next node of $(Q,\mathbf{b})$ is $(\mathbf{await}(\mu-1,C,P1)//\mathbf{null},\mathbf{b})$. In order to keep the number of execution schemes of P finite, the program statements $\mathbf{await}(\mu-1,C,P1)$, etc. are not counted among the execution schemes of P. Similarly, $\mathbf{while}(\mu-1,C,P1)$, $\mathbf{while}(\mu-2,C,P2)$, etc. are not counted as execution schemes of P even though $\mathbf{while}(\mu,C,P1)$ may be such a scheme. The program information lost by this necessary restriction on execution schemes is compensated for by the introduction of bounded quantifiers of the form $(\exists m<\mu)$ in $\Gamma(P)$. The syntactic variables P1, P2, and P of 2.3 now range over BPL, the formulas C range over formulas of PA and their nonstandard translations, and the type variables t and u now range over star-finite types. We illustrate the modified construction by two examples that indicate the new features of $\Gamma(P)$.

3.2.2. Example. Let P = **while**$(\mu, C, [x/x+1])$ Then similarly to 2.3.4, the execution schemes of P and their types relative to $t([x/x+1]) = 3$ are defined as follows:

	$t(0,\mu) = (\mu-1,1,3)$
$S1$ = **while**$(\mu,C,[x/x+1])$	$t(1,\mu) = t(S1) = (\mu,1,3)$
$S2 = t(0,\mu):[x/x+1]$	$t(2,\mu) = t(S2) = ((6,t(0,\mu)),3)$
$S3 = t(0,\mu)$:**null**	$t(3,\mu) = t(S3) = ((6,t(0,\mu)),1)$
$S4$ = **null**	$t(4) = t(S4) = 1.$

Next we write down the associated variable scheme, for $0 \leq m \leq \mu$:

	$t(0,m) = (m-1,1,3)$
$S1$ = **while**$(\mu,C,[x/x+1])$	$t(1,m) = t(S1) = (m,1,3)$
$S2 = t(0,m):[x/x+1]$	$t(2,m) = t(S2) = ((6,t(0,m)),3)$
$S3 = t(0,m)$:**null**	$t(3,m) = t(S3) = ((6,t(0,m)),1)$
$S4$ = **null**	$t(4) = t(S4) = 1,$

and use this scheme to determine the diagram Diag(P):

$$t4 \xleftarrow{\neg C \vee (m=0)} t(1,m) \xrightarrow{C \wedge (m>0)} t(2,m) \longrightarrow t(3,m) \longrightarrow t(1,m-1).$$

$\phi(1,m,2,m) =$

$$(x=l(1,m) \land x'=l(2,m) \land \tau(v)=(\lambda(v),\rho(v)) \land \lambda(v)=l(1,m) \land \rho(v)=r \land C[x/r] \land m>0 \land s=r)$$

$\phi(2,m,3,m) =$

$$(x=l(2,m) \land x'=l(3,m) \land \tau(v)=(\lambda(v),\rho(v)) \land \lambda(v)=l(2,m) \land \rho(v)=r \land s=r+1)$$

$\phi(3,m,1,m-1) =$

$$(x=l(3,m) \land x'=l(1,m-1) \land \tau(v)=(\lambda(v),\rho(v)) \land \lambda(v)=l(3,m) \land \rho(v)=r \land s=r)$$

$\phi(1,m,4) =$

$$(x=l(1,m) \land x'=l(4) \land \tau(v)=(\lambda(v),\rho(v)) \land \lambda(v)=l(1,m) \land \rho(v)=r \land (\neg C[x/r] \lor m=0) \land s=r).$$

$$\Gamma(P) = (\exists m \leq \mu)(\phi(1,m,2,m) \lor \phi(2,m,3,m) \lor \phi(3,m,1,m-1) \lor \phi(1,m,4)). \blacklozenge$$

3.2.3. Example. Let P = **parallel(await**$(\mu, C, [x/x+1])$**, null)**. Then the execution schemes of P and their respective types relative to $t([x/x+1]) = 3$ are

$S1 = P$ $\quad t(1,\mu) = t(S1) = (5,((\mu,2,3),1))$

$S2 = [x/x+1]//$**null** $\quad t(2,\mu) = t(S2) = (5,(3,1))$

$S3 =$ **null//null** $\quad t(3,\mu) = t(S3) = (5,(1,1))$.

Next we write down the associated variable scheme, for $0 \leq m \leq \mu$:

$S1 = P$ $\quad t(1,m) = t(S1) = (5,((m,2,3),1))$

$S2 = [x/x+1]//$**null** $\quad t(2,m) = t(S2) = (5,(3,1))$

$S3 =$ **null//null** $\quad t(3,m) = t(S3) = (5,(1,1))$.

The diagram Diag(P) is now

$$t(1,m-1) \xleftarrow{m>0 \wedge \neg C} t(1,m) \xrightarrow{m=0 \vee C} t(2,m) \longrightarrow t(3,m).$$

$\phi(1,m,2,m) =$

$$(z=t(1,m) \wedge z'=t(2,m) \wedge \tau(v)=(\lambda(v),\rho(v)) \wedge \lambda(v)=t(1,m) \wedge (m=0 \vee C[x/r]) \wedge \rho(v)=r \wedge s=r)$$

$\phi(2,m,3,m) =$

$$(z=t(2,m) \wedge z'=t(3,m) \wedge \tau(v)=(\lambda(v),\rho(v)) \wedge \lambda(v)=t(2,m) \wedge \rho(v)=r \wedge s=r+1)$$

$\phi(1,m,1,m-1) =$

$$(z=t(1,m) \wedge z'=t(1,m-1) \wedge \tau(v)=(\lambda(v),\rho(v)) \wedge \lambda(v)=t(1,m) \wedge m>0 \wedge \neg C[x/r] \wedge \rho(v)=r \wedge s=r).$$

$$\Gamma(P) = (\exists m \leq \mu)(\phi(1,m,2,m) \vee \phi(2,m,3,m) \vee \phi(1,m,1,m-1)). \blacklozenge$$

The general construction is by induction on programs and follows the steps described in the two previous examples. The formulas $\Gamma(P) \in {}^*L$ are all of form $(\exists m<\mu)\Phi(P)$, where $\Phi(P)$ is constructed from the execution schemes and the associated variable schemes as in 2.3 and the previous examples.

3.2.4. The definable trees Fan(μ,P,a)

We now construct nonstandard analogues Fan(μ,P,a) of the fans defined in 2.4.1 (a), (b), and (c), by letting the variables w1 and w2 now range over the (*G-values of) star-finite types and the data component **b** range over $*(N^n)$. Here the *G-values of the star-finite types in Tr(μ,P,a) are the nonstandard counterparts of the Gödel numbers of finite types. By an argument similar to that in 2.4.1, we see at once that

$$*N \models (\exists m\langle\mu\rangle\Phi(P)[z/ti,z'/tj,\pi/e,r/b,s/b']$$

if and only if there exists a star-finite path $e = (e1,\ldots,en,e(n+1)) \in$ Tr(μ,P,a) with the property that $en = (\lambda(n),\rho(n)) \approx (w2,w1,b)$ and $e(n+1) = (\lambda(n+1),\rho(n+1)) \approx (w2',w1',b')$, where $ti = ((6,w2),w1)$ and $tj = ((6,w2'),w1')$ as defined in 2.1.6. Hence Fan(μ,P,a) is a definable tree and, by the Definability Principle, is therefore internal. By abuse of language, we call the corresponding cancellation tree Tr(μ,P,a) an internal tree. Although no longer an intuitionistically acceptable object, we continue to refer to the trees Fan(μ,P,a) as "fans".

We illustrate the construction by two examples:

3.2.5. Example. Let P = **while**(2,**TRUE**,[x/x+1]) as in 3.1.9 and let t([x/x+1]) = 3. Then Fan(2,P,0) is the linearly ordered tree

(w2,w1,0) = (0;(2,1,3);0)
-> ((1,1,3);3;0)
-> ((1,1,3);1;1)
-> (0;(1,1,3);1)
-> ((0,1,3);3;1)
-> ((0,1,3);1;2)
-> (0;(0,1,3);2)
-> (0;1;2)

As always, we have written the types in place of their Gödel numbers. Thus (2,1,3) is really the G(G(2,1,3), etc. ♦

3.2.6. Example. Let P = **while**(1,**TRUE**,[x/x+1]//[y/y+1]), let t([x/x+1]) = 2 and t([y/y+1]) = 3, and let **a** = (7,8). Then Fan(1,P,(7,8)) is the tree determined by the following two maximal branches:

(a) (1) -> (2) -> (3) -> (5) -> (7) -> (8)

(b) (1) -> (2) -> (4) -> (6) -> (7) -> (8),

where

(1) = (w2;w1;7,8)

= (0;(1,1,(5,(2,3)));7,8)

(2) = ((0,1,(5,(2,3)));(5,(2,3));7,8)

(3) = ((0,1,(5,(2,3)));(5,(1,3));8,8)

(4) = ((0,1,(5,(2,3)));(5,(2,1));7,9)

(5) = ((0,1,(5,(2,3)));(5,(1,1));8,9)

(6) = ((0,1,(5,(2,3)));(5,(1,1));8,9)

(7) = (0;(0,1,(5,(2,3)));8,9)

(8) = (0;1;8,9). ♦

One of the distinguishing features of the trees Fan(μ,P,**a**) that justifies their designation as fans is the fact that if the bound μ in the await and while statements of P is standard, then Fan(μ,P,**a**) is a finite tree and corresponds to the cancellation tree of a terminating standard program.

3.2.7. Theorem. If P $\in$ BPL is uniformly bounded by $\mu \in$ N, then Tr(μ,P,**a**) is finite for all **a** $\in$ N^n and all its leaves are real.

Proof. The proof is by an induction on P. If P is **null** or [x/v], the result holds trivially. Suppose that the theorem holds for P1 and P2, and consider Tr(μ,**compose**(P1,P2),**a**). The root of this tree is (**compose**(P1,P2),**a**) with next node (t0:P2,**a**), where t0 is the type of t(**compose**(P1,**null**)). The next nodes of (t0:P2,**a**) and its "immediate" successors are obtain by replacing every node (Q,**b**) of Tr(μ,P2,**a**) by (t0:Q,**b**) and grafting the resulting tree to the node (t0:P2,**a**). By the induction hypothesis, all leaves of the resulting tree are of the form (t0:**null**,**b**), for some **b**, and by clause (a) of 2.2.1, the next node of (t0:**null**,**b**) is (P1,**b**) and Tr(μ,**compose**(P1,P2),**a**) is continued at (t0:**null**,**b**) by grafting Tr(μ,P1,**b**) to (t0:**null**,**b**). Since there are only finitely many nodes of the form (t0:**null**,**b**), and since Tr(μ,P1,**b**) is finite for each **b**, Tr(μ,**compose**(P1,P2,**a**)) is finite. The construction is obviously independent of the nature of the input **a** $\in$ N^n. Similar grafting arguments establish the theorem for programs of the form

if(C,P1,P2) and **parallel**(P1,P2) and it remains to consider while and await statements. We describe only the while case in detail. The await case follows from a similar, but simpler, grafting argument. If P is of the form **while**(μ,C,P1) and C[x/**a**] is false, then P terminates in one step with leaf (**null**,**a**). Hence suppose that C[x/**a**] is true. Then the next node of (**while**(μ,C,P1),**a**) is the node (t(μ-1):P1,**a**), where t(μ) = (μ,1,t(P1)) and t(μ-p) = (μ-p,1,t(P1)). Thus the continuation of Tr(μ,P,**a**) at (t(μ-1):P1,**a**) is obtained by replacing each node (Q,**b**) of Tr(μ,P1,**a**) by (t(μ-1):Q,**b**) and grafting the resulting tree to the root of Tr(μ,P,**a**). By the induction hypothesis and our construction, the nodes of Tr(μ,P,**a**) corresponding to the leaves of Tr(μ,P1,**a**) are all of the form (t(μ-1):**null**,**b**). If C[x/**b**] is false for some **b**, the next two nodes of (t(μ-1):**null**,**b**) are (**while**(μ-2,C,P1),**b**) and (**null**,**b**). Otherwise the described grafting is repeated, with t(μ-2) in place of t(μ-1). Since μ is finite, this process terminates and the resulting leaves are all of the form (**null**,**c**). ♦

3.2.8. Corollary. If P ∈ BPL is uniformly bounded by $\mu \in$ **N**, then Fan(μ,P,**a**) is finite for all **a** $\in$ $\mathbf{N}^n$. ♦

By mapping all $(w2,w1,b) \in Fan(\mu,P,a)$ to $G(G(w2),G(w1),G(b)) \in N$, Corollary 3.2.8 can be rephrased to the effect that for all $\mu \in N$ and all $a \in N^n$ there exists an $\nu \in N$ such that the set $Fan(\mu,P,a) \subseteq [0,\nu]$. By applying the Transfer Principle to this sentence of PA and using the fact that $Fan(\mu,P,a)$ is internal for all $\mu \in {}^*N$ and $a \in {}^*N^n$, we immediately obtain the finiteness property of the fans associated with nonstandard cancellation trees:

3.2.9. Theorem. For all $P \in BPL$, the fan $Fan(\mu,P,a)$ is star-finite for all $\mu \in {}^*N$ and all $a \in {}^*N^n$. ♦

Following the idea of the equivalence of programs outlined in RICHTER and SZABO [1983], we now present a concept of **operational equivalence** for parallel programs that has the desired property that every program $P \in PL$ is equivalent to a program $\beta(P) \in BPL$. Since all programs in BPL terminate in the sense of 2.4, with the requirement that $Tr(P,a)$ is finite replaced by the condition that $Tr(\mu,P,a)$ is star-finite, this amounts to saying that all programs in PL are equivalent to terminating programs in BPL. It is important to observe that this result is only of metamathematical value since termination refers to termination in $^*N$ and may occur only in nonstandard time. We require the following notation: For any tree $Tr(P,a)$ and any tree $Tr(\mu,P,a)$ we define $Data(Tr(P,a))$ and $Data(Tr(\mu,P,a))$ to be the trees obtained from $Tr(P,a)$ and $Tr(\mu,P,a)$ by deleting the program components of the nodes of $Tr(P,a)$ and $Tr(\mu,P,a)$ and call them the underlying **data trees** of $Tr(P,a))$ and $Tr(\mu,P,a))$. By $Data(Tr(\mu,P,a))$ restricted to N we mean the "standard part" of $Data(Tr(\mu,P,a))$, i.e., the tree obtained by restricting all paths to their initial segments over N. We consider two data trees to be identical if they can be transformed into each other by the permutation of maximal paths.

3.2.10. Let P, Q $\in$ PL $\cup$ BPL and a $\in N^n$. Then we define P to be **operationally equivalent** to Q at a, written as $P \equiv_{(a)} Q$, if the standard parts of their data trees coincide.

3.2.11. We define P and Q to be **operationally equivalent** if they are operationally equivalent at all standard inputs. We write $P \equiv Q$ if P and Q are operationally equivalent.

3.2.12. Termination theorem. For every deadlock-free P $\in$ PL there exists a uniformly bounded $\beta(P) \in$ BPL such that $P \equiv \beta(P)$.

Proof. Let $\mu \in {}^*N - N$, and let $\beta(P)$ be the program obtained from P by replacing all statements of the form **await**(C,P1) and **while**(C,P1) in P by **await**(μ,C,P1) and **while**(μ,C,P1). Since the uniform bound μ is infinite, the resulting program $\beta(P) \in$ BPL clearly has the desired property. ♦

3.3. Complexity types

In this section we introduce an equivalence relation ≡ on program types for programs in PL ∪ BPL which is intended to reflect our idea of two programs being of the same structural complexity. We call the resulting equivalence classes **complexity types** and say that two programs P and Q are of the same complexity if t(P) and t(Q) belong to the same complexity type. We then establish some algebraic facts about the resulting hierarchy of programs which point to the plausibility of the defined notion of equivalence. The advantage of defining this equivalence in terms of types instead of defining it directly in terms of programs derives from the fact that the types separate the **form** of a program from its **content** and therefore contain precisely the information relevant to the question of complexity.

3.3.1. Equivalent types

For the purpose of classifying programs via program types we shall refer to the equivalence class of the atomic type 1 as **zero** and to the equivalence class of types made up entirely of even atomic types, possibly together with the atomic type 1, as **one**. When relating complexity types to programs, we agree to use even types for assignments of the form [x/x] and odd types for assignments of the form [x/x+1] and [x/x-1].

The equivalence relation $\equiv$ is the smallest equivalence relation satisfying the following conditions:

(a) If t and t' contain only 1 as an atomic type, then $t \equiv t'$.

(b) If t and t' contain only even atomic types, possibly together with the atomic type 1, then $t \equiv t'$.

(c) If $\mu,\nu \in N$, $\mu>0$ and $\nu>0$, and μ and ν are odd, then $\mu \equiv \nu$.

(d) $(4,(1,t)) \equiv t \equiv (4,(t,1)) \equiv (5,(1,t)) \equiv (5,(t,1))$.

(e) $(4,(t1,\ldots,ti,1,ti+2,\ldots,tn)) \equiv (4,(t1,\ldots,ti,ti+2,\ldots,tn))$ and $(5,(t1,\ldots,ti,1,ti+2,\ldots,tn)) \equiv (5,(t1,\ldots,ti,ti+2,\ldots,tn))$.

(f) $(5,(t1,\ldots,ti,ti+1,\ldots,tn)) \equiv (5,(t1,\ldots,ti+1,ti,\ldots,tn))$.

(g) If $t1 \equiv t1'$, ..., $tn \equiv tn'$, then $(4,(t1,\ldots,tn)) \equiv (4,(t1',\ldots tn'))$ and $(5,(t1,\ldots,tn)) \equiv (5,(t1',\ldots tn'))$.

(h) If $t \equiv t'$ and $u \equiv u'$, then $(3,(t,u)) \equiv (3,(t',u'))$.

(i) If $t \equiv t'$, then $(1,t) \equiv (1,t')$ and $(2,t) \equiv (2,t')$.

(j) If t is even, then

$(4,(t1,\ldots,ti,t,ti+1,\ldots,tn)) \equiv (4,(t1,\ldots,ti,ti+1,\ldots,tn))$

and $(4,(t1,t)) \equiv t$.

(k) $(4,(t1,t2,\ldots,tn)) \equiv (4,(t1,(4,(t2,\ldots,tn))))$

$\equiv (4,(4,((t1,\ldots,tn-1),tn)))$.

(l) $(5,(t1,t2,\ldots,tn)) \equiv (5,(t1,(5,(t2,\ldots,tn))))$

$\equiv (5,(5,((t1,\ldots,tn-1),tn)))$.

(m) $(4,(t1,(5,(t2,\ldots,tn)))) \equiv (5,((4,(t1,t2)),\ldots,(4,(t1,tn))))$

$(4,((5,(t2,\ldots,tn)),t1)) \equiv (5,((4,(t2,t1)),\ldots,(4,(tn,t1))))$.

(n) If $\mu,\nu \in {}^*N-N$, then $(\mu,1,t) \equiv (\nu,1,t) \equiv (1,t)$ and $(\mu,2,t) \equiv (\nu,2,t) \equiv (2,t)$.

Clause (a) says that all zero types are equivalent, i.e., we do not not intend to distinguish the complexity of any programs constructed entirely from the **null** statement. Clause (b) says that all types constructed from even types are equivalent, i.e., that we do not distinguish the structural complexity of any programs constructed entirely from identity assignments and **null**. This seems reasonable since we can predict ahead of time that no data component of any cancellation tree associated with such a program changes value. Clause (d) and (e) assert that the introduction of **null** components and their equivalents in compose and parallel statements do not increase the complexity of such programs. Clause (f) states that the complexity of a parallel statement is independent of the order in which parallel processes are listed. Clauses (g)-(i) guarantee that ≡ is stable under **compose**, **parallel**, **if**, **while** and **await**. Clause (j) is intended to achieve that even programs act as identities with respect to composition. Clauses (k) and (l) achieve that **compose** and **parallel** interpret as associative binary operations. Clause (m) asserts that **compose** distributes over **parallel**, i.e., that a program of the form **compose**(P1,**parallel**(P2,P3)) is of the same structural complexity as **parallel**(**compose**(P1,P2),**compose**(P1,P3)). From a computational point of view such an assumption is entirely plausible. Clause (n) establishes the connection between PL and BPL and guarantees that unbounded while and await statements have the same structural complexity as while and await statements with infinite

bounds. Clause (c), finally, achieves the desired separation of the structural and operational aspects of a program by eliminating all arithmetical considerations from the question of equivalence.

We denote the equivalence class of a type t by [t] and now examine some of the algebraic properties of the defined complexity types. As usual, we refer to both t and its equivalence class [t] as a type.

First we define a partial ordering $\propto$ on types which measures the degree of parallelism involved:

3.3.2. The relation $\propto$ is the coarsest partial ordering with the property that $[u] \propto [v]$ if and only if $(u \equiv 1)$ or there exists types t and t' such that $(u \equiv t)$ and $(v \equiv (5,(t,t')))$.

In order to guarantee that $\propto$ is well-defined we must show that if $(u \equiv u')$ and $(v \equiv v')$ then $[u] \propto [v]$ if and only if $[u'] \propto [v']$. If $u \equiv u' \equiv 1$ the result is immediate and the general case follows at once from the transitivity and reflexivity of $\equiv$: If $[u] \propto [v]$, then $u' \equiv u \equiv t$ and $v' \equiv v \equiv (5,(t,t'))$ and hence $u' \equiv t$ and $v' \equiv (5,(t,t'))$, i.e., $[u'] \propto [v']$.

In accordance with our intention to model **parallel** algebraically as a sum and **compose** as a product, we now introduce an arithmetical structure on types which has the desired properties:

3.3.3. For any types [t] and [u] we put

(a) [t] + [u] = [(5,(t,u))]

(b) [t] * [u] = [(4,(t,u))].

As expected, complexity classes provide the expected link between the structural aspects of the different programming constructs of PL and BPL:

3.3.4. Theorem. The class CT of complexity types is a partially ordered distributive commutative monoid with identity.

Proof. The following calculations show that CT is a commutative monoid with respect to addition:

(a) ([t] + [u]) + [v] = [(5,((5,(t,u)),v))]

= [(5,(t,(5,(u,v))))] by 3.3.1 (l)

= [t] + [(5,(u,v))]

= [t] + ([u] + [v])

(b) [t] + [u] = [(5,(t,u))] = [(5,(u,t))] = [u] + [t] by 3.3.1 (f)

(c) [t] + **zero** = [t] + [t'] = [(5,(t,t'))]

= [(5,(t,1))] by 3.3.1. (a) and (g)

= [t] by 3.3.1 (d).

The next equations show that * is associative and that **one** is an identity for *:

(d) ([t]*[u])*[v] = [(4,(t,u))]*[v]

= [(4,((4,(t,u)),v))]

= [(4,(t,(4,(u,v))))] by 3.3.1 (k)

= [t]*[(4,(u,v))]

= [t]*([u]*[v])

(e) [t]*one = [t]*[t'], where t' = 2

= [(4,(t,t'))]

= [(4,(t,2))] by 3.3.1 (b) and (g)

= [t] by 3.3.1 (j).

The following calculations show that * distributes over +:

(f) [t]*([u] + [v]) = [t]*[(5,(u,v))]

= [(4,(t,(5,(u,v))))]

= [(5,((4,(t,u)),(4,(t,v))))] by 3.3.1 (m)

= [(4,(t,u))] + [(4,(t,v))]

= ([t]*[u]) + ([t]*[v]),

and similarly

([t] + [u])*[v] = ([t]*[v]) + ([t]*[u]).

Finally, we show that α is compatible with +:

(g) If ([t] α [u]) and ([v] α [w]), then
(([t] + [v]) α ([u] + [w])):

Let (t ≡ t1) and (v ≡ v1). Then, by 3.3.2,
u = (5,(t1,t2)) and w = (5,(v1,v2)).

Hence ([t] + [v]) = [(5,(t1,v1))] and

([u] + [w]) = [(5,((5,(t1,t2)),(5,(v1,v2))))]
= [(5,((5,(t1,t2)),v1,v2))] by 3.3.1 (i)
= [(5,(v1,v2,(5,(t1,t2))))] by 3.3.1 (f)
= [(5,(v1,v2,t1,t2))] by 3.3.1 (i)
= [(5,(t1,v1,t2,v2))] by 3.3.1 (f)
= [(5,((5,(t1,v1)),(5,(t2,v2))))] by 3.3.1 (i).

Thus there exists an element (5,(t1,v1) ∈ ([t] + [v]) with the property that [(5,((5,(t1,v1)),(5,(t2,v2))))] ∈ ([u] + [w]).

This proves the theorem. ♥

BIBLIOGRAPHY.

APT, K. R.

[1981] **Recursive assertions and parallel programs**, Acta Informatica, 15, pp. 219-232.

APT, K. R.

[1984] **Ten years of Hoare's logic: A survey - Part II: Nondeterminism**, Theoretical Computer Science, **28**, pp. 83-109.

CHANG, C. C. and KEISLER, H. J.

[1973] Model theory, North-Holland, Amsterdam.

CSIRMAZ, L.

[1981] **Programs and program verification in a general setting**, Theoretical Computer Science, **16**, pp. 199-210.

CSIRMAZ, L.

[1984] **Nonstandard semantics in program verification**, Preprint, Hungarian Academy of Sciences, pp. 1-104.

DAVIS, M.

[1977] Applied nonstandard analysis, Wiley, New York.

DUMMETT, M. A. E.

[1977] Elements of intuitionism, Oxford University Press, Oxford.

FARKAS, E. J.

[1981] Nonstandard aspects of analysis, M.Sc. thesis, Concordia University, Montreal.

FARKAS, E. J. and SZABO, M. E.

[1983] **A star-finite relational semantics for parallel programs**, in: Computation and Proof theory (M. M. Richter et al., editors), Lecture Notes in Mathematics, **1104**, Springer-Verlag, pp. 129-142.

FARKAS, E.J. and SZABO, M.E.

[1984] **On the plausibility of nonstandard proofs in analysis**, Dialectica, **38**, pp. 297-310.

FEFERMAN, S.

[1977] **Theories of finite type related to mathematical practice**, in: Handbook of Mathematical Logic (J. Barwise, editor), North-Holland, Amsterdam, pp. 913-971.

GANDY, R. O.

[1980] **An early proof of normalization by A. M. Turing**, in: Essays on Combinatory Logic, Lambda Calculus and Formalism, (J. P. Seldin and J. R. Hindley, editors), Academic Press, New York, pp.453-455.

HOARE, C. A. R.

[1972] **Towards a theory of parallel programming**, A.P.I.C. Studies in Data Processing, (C. A. R. Hoare and R. H. Perrott, editors), Academic Press, New York, pp. 61-71.

MAC LANE, S. and BIRKHOFF, G.

[1967] Algebra, Macmillan, New York.

MENDELSON, E.

[1979] Introduction to mathematical logic, D. Van Nostrand, New York.

OWICKI, S.

[1975] **Axiomatic proof techniques for parallel programs**, Ph.D. thesis, Cornell University, Ithaca, New York.

OWICKI, S. and GRIES, D.

[1976] **An axiomatic proof technique for parallel programs I**, Acta Informatica, 6, pp. 319-340.

RICHTER, M. M. and SZABO, M. E.

[1983] **Towards a nonstandard analysis of programs**, in: Non-standard analysis - recent developments, (A. E. Hurd, editor), Lecture Notes in Computer Science, 983, pp. 186-203.

RICHTER, M. M. and SZABO, M. E.

[1985] **Nonstandard computation theory**, in: Algebra, Combinatorics, and Logic in Computer Science, Colloquia Mathematica Soc. J. Bolyai, 42, (J. Demetrovics, G. Katona, and A. Salomaa, editors). North-Holland, Amsterdam. To appear.

RUSSELL, B.

[1908] **Mathematical logic as based on the theory of types**, in: From Frege to Gödel, (J. van Heijenoort, editor), Harvard University Press, Cambridge, Massachusetts, 1967.