BANACH CENTER PUBLICATIONS
VOLUME 2

AN ANALYSIS OF PROGRAMS BY ALGEBRAIC MEANS

A. BLIKLE

Institute of Computer Science, Polish Academy of Sciences, PKiN, P.O.Box 22, 00-901 Warsaw

CONTENTS

1. INTRODUCTION

The rapid development of software techniques and technologies in the past decade raises the urgent necessity of creating a theoretical background adequate to fulfil the following three tasks:

1. Improving our general understanding of software; e.g., answering questions such as: What should a programming language look like? What are data structures and how should they be described? What can program complexity mean and how to measure it? etc.

2. Providing tools for analysis, documentation and designing of software objects such as programs, operating systems, programming languages, data banks, etc.

3. Improving methods of teaching programming by introducing theories with well-defined ideas—e.g., of a program, of recursion and iteration, of structuring—and eliminating teaching by examples as the only method of education in software.

The problem that stimulated the present contribution—and a large number of others—is connected with (2) above and is usually referred to as proving properties of programs. In fact, we try here to construct a mathematical tool for analysis of programs.

Programs are clearly expressions in more or less formal languages, however, proving properties of programs is not proving properties of expressions but of their meaning. What are, therefore, the meanings of programs?

For the majority of authors these meanings are functions, or binary relations, describing the input-output processing. How these relations are associated with programs is not a straightforward matter and therefore must be described formally. The most common method of carrying out such description consists in introducing a formal language L, whose expressions are interpreted as programs (cf. de Bakker [1], de Bakker and Meertens [2], de Bakker and de Roever [3], de Bakker and Scott [4], Cadiou [18], Engelfriet [21], Hitchcock and Park [22], Manna and Vuillemin [28], Nivat [33], Park [34], Rasiowa [35], Salwicki [37], [38], [39]). The way one associates such expressions with programs is considered obvious and therefore is not described in the system. The way one associates relations with expressions is not obvious at all and is usually described by a semantic function

$$S\colon L \to R,$$

where R denotes an appropriate set of relations. Proving properties of programs consists in proving properties of the associated relations. This may be carried out either within the usual mathematical theories such as set theory, algebra of relations, and the like, or in purpose-oriented formalized theories frequently equipped with a special logic. In the latter case, L is assumed to be a set of terms in the theory and appropriate formulas are considered as theorems about programs.

In the present approach we proceed in a different way. Mathematical models of programs are assumed to be abstract algorithms, and their meanings to be objects generated (defined) by these algorithms. The generated objects, however, need not to be solely relations. According to what we want to prove about a program, we associate with it an algorithm generating objects of appropriate type:

1. the usual binary relations—for the usual input-output analysis,

2. the so-called δ-relations—for input-output analysis covering description of infinitistic behaviour (looping),

3. sets of sequcnces of states (called computations)—to extend the input-output description to the description of runs,

4. languages—to prove properties of classes of programs with the same "toplology" but different interpretations of modules,

5. other objects—for other purposes.

To prove properties of programs in this way, we need a concept of a program-oriented algorithm and an algebraic tool to describe and analyze algorithmic definability. The former is fulfilled by abstract Mazurkiewicz algorithms, the latter by attice-like algebras called quasinets. A few comments concerning these concepts are given below.

Let U be a set of objects (e.g. relations, sets of computations or languages) to be generated (defined) by algorithms of a given class. Every algorithm over U is assumed to consist of:

1. A finite number of elementary modules (e.g. boxes in flowcharts, elementary instructions in recursive procedures, etc.), each of them associated with an element of U.

2. A control mechanism to define the succession in which the elementary modules are to be executed (e.g. flowchart structure, recursive calls, coroutine mechanism, etc.).

Each algorithm is assumed to generate a resulting object g in U. For example, if $a_1, \ldots, a_4$ of Fig. 1.1.1 are partial functions corresponding to elementary instructions, then g is the output function of the program, if $a_1, \ldots, a_4$ are names of instructions (one-character languages) and control is described by a flowchart, then g is the corresponding regular language containing all possible Engeler's signatures of the program (see Engeler [20]), if $a_1, \ldots, a_4$ are sets of microcomputations, then g is the set of global computations of the program, etc.

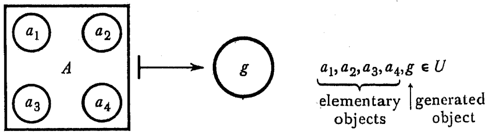

Fig. 1.1.1

Proving properties of programs in our case will consist in proving properties of objects generated by appropriate algorithms. Starting with known local properties, i.e., properties of elements associated with modules (such as $a_1, \ldots, a_4$), we shall prove a given global property, i.e., a property of the element generated by the algorithm (such as g). To provide a mathematical tool for this approach we introduce special algebras over U called quasinets. Operations in these algebras are powerful enough to describe each generated element g by means of the corresponding local elements $a_1, \ldots, a_n$. In other words, we are able to describe the algorithmic definability in algebraic terms. Very briefly each quasinet is a complete lattice over U with an additional monoidal additive and continuous operation. Completeness of the lattice provides the infinitistic operations necessary in dealing with iteration and recursion. The purpose of the finitistic join is to describe branching and that of monoid operation is to describe sequencing in algorithms.

Warsaw in December 1974

Acknowledgement

I wish to thank H. Müller who has pointed out some mistakes in the early draft of my fixedpoint theorems, and suggested how to correct them. I express also my

gratitude to W. D. Maurer, who has taken the trouble to go through my manuscript and make corrections in English.

2. NETS AND QUASINETS

2.1. Abstract quasinets

Let $(U, \leqslant)$ be a complete lattice. For any set $P \subseteq U$ and any elements a and b in U we write:

$$\bigcup P\text{—the l.u.b. of } P,$$
$$\bigcap P\text{—the g.l.b. of } P,$$
$$a \cup b = \bigcup \{a, b\}\text{—the join of } a \text{ and } b,$$
$$a \cap b = \bigcap \{a, b\}\text{—the meet of } a \text{ and } b.$$

A set $P \subseteq U$ is said to be *directed* if any finite subset of P has an upper bound in P.

A function $\varphi\colon U \to U$ is said to be *additive* if it preserves finite joins, i.e., for any elements a and b in U

$$\varphi(a \cup b) = \varphi(a) \cup \varphi(b),$$

and is said to be *continuous* if it preserves the l.u.b. of directed sets, i.e., for any directed set $P \subseteq U$

$$\varphi\left(\bigcup P\right) = \bigcup \{\varphi(p)|\ p \in P\}.$$

A function $\Phi\colon U^n \to U$ with $n \geqslant 1$ is called *additive* (resp. *continuous*) if it is additive (resp. *continuous*) for each argument separately.

By a *quasinet* we shall mean any system $N = (U, \leqslant, \circ, e)$, where U is a set, $\leqslant$ is a partial ordering in U, $\circ$ is a binary operation in U called *composition*, e is an element in U and the following conditions are satisfied:

(1) $(U, \leqslant)$ is a complete lattice,
(2) $(U, \circ, e)$ is a monoid with unit e,
(3) $\circ$ is additive and continuous,
(4) for any a in U,

$$\mathbf{0} \circ a = \mathbf{0},$$

where $\mathbf{0} = \bigcap U$ is the *bottom element* of the lattice.

By a *net* we mean any quasinet $N = (U, \leqslant, \circ, e)$ with the additional property:
(5) for any a in U,

$$a \circ \mathbf{0} = \mathbf{0}.$$

In the sequel we omit the symbol "$\circ$" and write ab instead of $a \circ b$. We assume also that "$\circ$" binds more strongly than "$\cup$" and "$\cap$", i.e., $ab \cup c$ means $(a \circ b) \cup c$ and for "$\cap$" similarly. A quasinet (net) is said to be *set-theoretic* provided U is a family of sets and $\leqslant$ is an ordering by inclusion. In this case join and meet are just union and intersection.

LEMMA 2.1.1. *If $\varphi\colon U \to U$ is continuous and additive in $(U, \leqslant)$, then for any finite (but nonempty) or denumerable set $P \subseteq U$*

$$\varphi\left(\bigcup P\right) = \bigcup \{\varphi(p)|\ p \in P\}.$$

In consequence, our composition is left and right distributive over finite and denumerable joins.

By the *n*th *power* and *-*iteration* we mean respectively the following operations in U:

$$a^0 = e,\ a^{n+1} = a^n a \quad \text{for} \quad n = 0, 1, \ldots,$$
$$a^* = \bigcup_{n=0}^{\infty} a^n.$$

The iteration and the power are assumed to bind more strongly than join and meet.

Interpretation. The set U is supposed to be a set of objects that we associate with algorithms and their modules. The lattice and the monoid over U are interpreted in such a way, that the join corresponds to alternative branching of algorithms and composition to sequencing of algorithms (Fig. 2.1.1). This interpretation motivates in a natural way the additivity of composition (Fig. 2.1.2). The denumerable join $\bigcup_{n=0}^{\infty} a_n$ corresponds to a denumerable alternative branching, hence loop can be described by *-iteration. Let us now explain conditions (4) and (5). The bottom element $\mathbf{0}$ can be interpreted as generated by an "empty algorithm", i.e., an algorithm without runs whatsoever. Suppose now we link sequentially two algorithms as in Fig. 2.1.1(b). If $a = \mathbf{0}$, then the compound algorithm generating ab has no runs at all, and therefore $ab = \mathbf{0}$. If, however, $b = \mathbf{0}$, then the set of runs of the compound algorithm will consist exactly of all infinite runs of the algorithm generating a. Since infinite runs may also generate an element in U, we cannot assume now that $ab = \mathbf{0}$. On the other hand, we may be willing to develop the

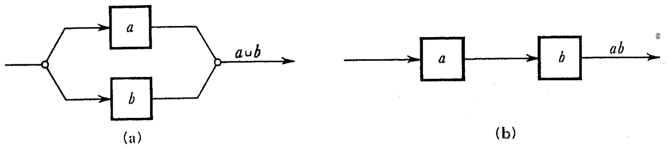

Fig. 2.1.1

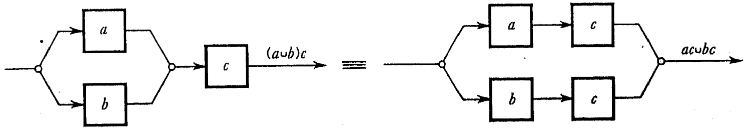

Fig. 2.1.2

analysis of algorithms neglecting infinitistic behaviour. In such a case we impose condition (5) and we deal with nets. For example, the usual binary relations constitute nets (looping is assumed to have no effect in this case) and δ-relations (Section 2.5) constitute quasinets which are not nets, and therefore permit us to describe the infinitistic behaviour of algorithms. ■

We shall introduce now a generalized composition in N that will play an important part in our definition of operational semantics of algorithms. Let $seq(U)$ denote the set of all finite and infinite sequences of elements of U, the empty sequence ε included. By a *generalized composition* in N we mean any operation $C\colon seq(U) \to U$ with the following properties:

(1) $C[\varepsilon] = e$,

(2.1.1) (2) $C[a] = a$ for any a in U,

(3) $C[t_1{}^\frown t_2] = C[t_1]C[t_2]$ for any finite t_1 and finite or infinite t_2, where $t_1{}^\frown t_2$ denotes the usual concatenation.

As can be proved (D. Scott; personal communication), an operation C, as defined above, exists in any quasinet. On the other hand, there exist quasinets with more than one C-operation (e.g., the net of generalized languages). In consequence, dealing with a quasinet, where a particular generalized composition C is defined, we have to set it explicitly and consider the extended system $N = (U, \leqslant, \circ, e, C)$.

Given a C-operation in N, we define an infinitary iteration, called also ∞-*iteration*, in U. For any a in U:

(2.1.2) $$a^\infty = C[a, a, \ldots].$$

As is easy to see (cf. property (3) in (2.1.1)),

(2.1.3) $$aa^\infty = a^\infty.$$

The concept of an abstract net was originally introduced in Blikle [8]. Particular concrete quasinets appeared first in Blikle [13] and as abstract concepts in Blikle [14].

2.2. Sequences—general notation

The particular quasinets, considered in Sections 2.3 and 2.4, are algebras of sets of sequences. To deal with such sets, we introduce the following notation, assumed to be fixed for the sequel.

Let D be an arbitrary (finite or infinite) set. Finite and infinite sequences over D are written respectively as $(a_1, \ldots, a_n)$; $n < \infty$ and $(a_1, a_2, \ldots)$. We shall also write $(a_1, \ldots, a_n)$; $n \leqslant \infty$ to denote a sequence that may be finite or infinite. The empty sequence is denoted by ε and is assumed to be of length 0. We denote by:

$seq^*(D)$—the set of all finite sequences over D,

$seq^\infty(D)$—the set of all infinite sequences over D,

$seq(D) = seq^*(D) \cup seq^\infty(D)$.

For any set $A \subseteq seq(D)$, we have

$$Fin(A) = A \cap seq^*(D),$$
$$Inf(A) = A \cap seq^\infty(D).$$

Given two sequences $x = (a_1, \ldots, a_n)$ and $y = (b_1, \ldots, b_m)$ with $n, m \leqslant \infty$, we say that x is an *initial segment* of y—in symbols $x \sqsubseteq y$—if $n \leqslant m$ and $a_i = b_i$ for $i = 1, \ldots, n$.

2.3. Nets of generalized languages

The set D (Section 2.2) will now be called *alphabet* and sequences over D will be called *words*. In the set $seq(D)$ we define a total operation "$^\frown$" of *concatenation*:

(1) $(a_1, \ldots, a_n)^\frown(b_1, \ldots, b_m) = (a_1, \ldots, a_n, b_1, \ldots, b_m)$ for $0 \leqslant n < \infty$ and $0 \leqslant m \leqslant \infty$,

(2) $(a_1, a_2, \ldots)^\frown(b_1, \ldots, b_m) = (a_1, a_2, \ldots)$ for $0 \leqslant m \leqslant \infty$.

As is easy to see, $(seq^*(D), {}^\frown, \varepsilon)$ and $(seq(D), {}^\frown, \varepsilon)$ are both monoids.

By a *generalized language* (or shortly just *language*) over D we mean any subset of $seq(D)$. We shall write

$$GLan(D) = \{L \mid L \subseteq seq(D)\}.$$

The operation of concatenation for L_1, L_2 in $GLan(D)$ is defined in a usual way

$$L_1{}^\frown L_2 = \{x_1{}^\frown x_2 \mid x_1 \in L_1 \;\&\; x_2 \in L_2\}.$$

It is easy to show now that $NGL(D) = (GLan(D), \subseteq, {}^\frown, \{\varepsilon\})$ is a set-theoretic net with $\mathbf{0} = \emptyset$—the empty set. A generalized language L, with finite words only, will be said to be *finitistic* and, with infinite words only, to be *inherently infinitistic*. As is easy to check, finitistic languages constitute a subnet of $NGL(D)$.

To define C in $NGL(D)$, we need now an auxiliary operation in $seq(D)$. Let $x_1, x_2, \ldots$ be an infinite sequence of words in $seq(D)$. By $C[x_1, x_2, \ldots]$ we shall mean the shortest word in $seq(D)$ with the following property:

$$(\forall n \geqslant 1)(x_1{}^\frown x_2{}^\frown \ldots {}^\frown x_n \sqsubseteq C[x_1, x_2, \ldots]).$$

To define the generalized concatenation in $NGL(D)$, let $L_1, L_2, \ldots$ be an arbitrary infinite sequence of languages and let ε denote now the empty sequence of languages. We set:

(1) $C[\varepsilon] = \{\varepsilon\}$,
$C[L_1, \ldots, L_n] = C[L_1, \ldots, L_{n-1}]^\frown L_n$ for $1 \leqslant n < \infty$,

(2) $C[L_1, L_2, \ldots] = \{C[x_1, x_2, \ldots] \mid (\forall i \geqslant 1)(x_i \in L_i)\}$.

It is easy to see that this operation is a generalized composition, as defined in Section 2.1.

In $NGL(D)$ we shall identify elements in D with sequences of length 1, and shall therefore write—in this particular case—D^* instead of $seq^*(D)$.

Generalized languages and generalized concatenations of these languages were introduced and applied to program analysis in Redziejowski [36]. It must be emphasized that Redziejowski's ideas strongly stimulated the introduction of quasinets with infinitistic composition.

2.4. Quasinets of bundles of computations

Given an algorithm (program, abstract machine, etc.) or a module of it, one can investigate sequences of its states which are generated whenever the algorithm or module is run. These sequences, called computations, may be finite or infinite according to finite or infinite runs of the algorithm (module). The set of all such computations is a good representation of an algorithm (module) and can be successfully used in analysis of algorithms.

Again let D be an arbitrary set which we shall now call the *set of states*. Sequences in $seq(D)$ are called *abstract computations* or simply *computations*. The following operation of composition is defined in $seq(D)$:

(1) $(a_1, \ldots, a_n) \circ (b_1, \ldots, b_m) = \textbf{if } a_n = b_1 \textbf{ then}$
$(a_1, \ldots, a_n, b_2, \ldots, b_m) \textbf{ else } \varepsilon,$
for $0 \leqslant n < \infty$ and $0 \leqslant m \leqslant \infty$;

(2) $(a_1, a_2, \ldots) \circ (b_1, \ldots, b_m) = (a_1, a_2, \ldots)$ for $0 \leqslant m \leqslant \infty$.

In particular, we have

$$\begin{aligned} &\varepsilon \circ x = \varepsilon \quad \text{for} \quad x \in seq(D),\\ &x \circ \varepsilon = \varepsilon \quad \text{for} \quad x \in seq^*(D),\\ &(a, b) \circ (b) = (a, b) = (a) \circ (a, b),\\ &(a, b) \circ (b, c) = (a, b, c),\\ &\ldots\ldots\ldots\ldots\ldots\ldots, \text{ etc.} \end{aligned}$$

Interpretation. In the majority of cases, elements in D are interpreted as data-vector states rather than control states. Now, given two algorithms A_1 and A_2 linked sequentially as in Fig. 2.4.1 into a compound algorithm $A_1;A_2$, if $(a_1, \ldots, a_n)$ is a finite computation of A_1 and its last (output) element a_n is equal to the first (input) element of some (finite or infinite) computation $(b_1, \ldots, b_m)$ of A_2, then $(a_1, \ldots, a_n, b_2, \ldots, b_m)$ is a computation of $A_1;A_2$ (case (1)). Similarly, if $(a_1, a_2, \ldots)$ is an infinite computation of A_1, then $(a_1, a_2, \ldots)$ is also an infinite computation of $A_1;A_2$ (case (2)). ■

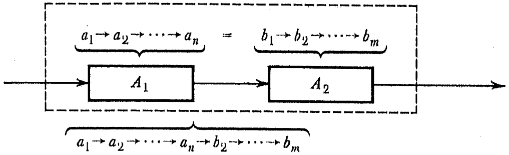
$a_1 \to a_2 \to \cdots \to a_n$
=
$b_1 \to b_2 \to \cdots \to b_m$
A_1
A_2
$a_1 \to a_2 \to \cdots \to a_n \to b_2 \to \cdots \to b_m$

Fig. 2.4.1

It is easy to show that composition of computations is associative in $seq(D)$, but, in contrast to the case of words (Section 2.3), is not a monoidal operation.

By a *bundle* over D we mean any subset B of $seq(D)$, with $\varepsilon \in B$. We denote:

$$Bun(D) = \{B \mid B \subseteq seq(D) \;\&\; \varepsilon \in B\}.$$

The assumption that ε is an element of any bundle is of technical character and will be explained below. In the set $Bun(D)$ we define an operation of *composition*. For any B_1, B_2 in $Bun(D)$:

$$B_1 \circ B_2 = \{x \circ y \mid x \in B_1 \;\&\; y \in B_2\}.$$

Let $E = seq^1(D) \cup \{\varepsilon\}$. It is easy to prove that $QNB(D) = (Bun(D), \leqslant, \circ, E)$ is a set-theoretic quasinet (but not a net), with $\mathbf{0} = \{\varepsilon\}$. The bundle $\mathbf{0}$ is called the *empty bundle* and for any B in $Bun(D)$, we have

$$\begin{aligned} \mathbf{0} \circ B &= \mathbf{0},\\ B \circ \mathbf{0} &= Inf(B) \cup \mathbf{0}. \end{aligned} \tag{2.4.1}$$

Equation (2.4.1) asserts that $QNB(D)$ is not a net. This equation is a formal consequence of the assumption that ε is in any bundle. In fact, this assumption has been made in order that (2.4.1) be satisfied. An interpretation of this property has been explained in Section 2.1.

A bundle B is said to be *finitistic* provided all its computations are finite and is said to be *inherently infinitistic* provided all its computations are infinite. Finitistic bundles constitute a net which is a subquasinet of $QNB(D)$.

Analogously as in Section 2.3, we now introduce the operation of generalized composition in $seq(D)$ and $Bun(D)$. Let $x_1, x_2, \ldots$ be a infinite sequence of computations, let $B_1, B_2, \ldots$ be an infinite sequence of bundles and let ε denote both the empty computation and the empty sequence of bundles.

$C[x_1, x_2, \ldots]$ is the shortest computation in $seq(D)$ with the following properties:

(i) If $(\exists n)(x_1 \circ \ldots \circ x_n = \varepsilon)$, then $C[x_1, x_2, \ldots] = \varepsilon$.

(ii) If $(\forall n)(x_1 \circ \ldots \circ x_n \neq \varepsilon)$, then

$$(\forall n)(x_1 \circ \ldots \circ x_n \sqsubseteq C[x_1, x_2, \ldots]).$$

Notice that $C[x_1, x_2, \ldots]$ may be empty, finite or infinite. For example:

$$\begin{aligned} &C[(a, b), (b, c), (b, c), \ldots] = \varepsilon \quad \text{provided} \quad b \neq c,\\ &C[(a, b), (b, c), (c), (c), \ldots] = (a, b, c),\\ &C[(a, a), (a, a), \ldots] = (a, a, \ldots). \end{aligned}$$

Now, to define the generalized composition in $Bun(D)$, we set

(1) $C[\varepsilon] = E,$
$C[B_1, \ldots, B_n] = C[B_1, \ldots, B_{n-1}] \circ B_n$ for $1 \leqslant n < \infty$,

(2) $C[B_1, B_2, \ldots] = \{C[x_1, x_2, \ldots] \mid (\forall i \geqslant 1)(x_i \in B_i)\}$.

As we have seen, the difference between $NGL(D)$ and $QNB(D)$ arises from the difference between concatenation and composition. Algebraically this leads to the fact that there are zero divisors in $QNB(D)$—i.e., there exist $B_1 \neq \mathbf{0}$ and $B_2 \neq \mathbf{0}$

with $B_1 \circ B_2 = \mathbf{0}$—while in $NGL(D)$ this is of course not true. To explain this difference for proving properties of programs, let us consider the following example.

Let A be an algorithm given by the flowchart of Fig. 2.4.1. Proving this algorithm in $NGL(D)$ requires associating a language with each operation box and a pair of languages (one for "no" and one for "yes") with each conditional box. A standard solution (frequently called the regular expression method, e.g., in Kaplan [24]) is to use one-word languages containing names of appropriate operations or conditions. Let thus

$$A_1 = \{\text{"}x < 0\text{"}\}, \qquad A_2 = \{\text{"}x \geqslant 0\text{"}\},$$
$$G = \{\text{"}x := x+1\text{"}\}, \qquad H = \{\text{"}x := 4x\text{"}\}.$$

What we can prove now about the algorithm is only that its finitistic behaviour corresponds to the language

$$L_{\mathrm{fin}} = (A_1 G)^* A_2 H,$$

and its entire behaviour to the language

$$L_{\mathrm{ent}} = (A_1 G)^* A_2 H \cup (A_1 G)^\infty.$$

We can prove nothing here that is a consequence of the interpretation (meaning) of the names of boxes, since we cannot make use of these interpretations within the algebra of languages. For bundles, on the contrary, we can and make use of it. We set D equal to the set of all integers and for modules:

$$A_1 = \{(d) \mid d < 0\} \cup \{\varepsilon\},$$
$$A_2 = \{(d) \mid d \geqslant 0\} \cup \{\varepsilon\},$$
$$G = \{(d, d+1) \mid d \in D\} \cup \{\varepsilon\},$$
$$H = \{(d, 4d) \mid d \in D\} \cup \{\varepsilon\}.$$

The finitistic and entire behaviour of the algorithm is described now by the following bundles:

$$B_{\mathrm{fin}} = (A_1 G)^* A_2 H,$$
$$B_{\mathrm{ent}} = (A_1 G)^* A_2 H \cup (A_1 G)^\infty.$$

These formulas, however, much like the former ones, permit us to prove many relevant properties of the algorithm. For example, we can prove the following equalities:

1) $(A_1 G)^\infty = \mathbf{0}$, i.e., there are no infinite runs in the algorithm,

2) $B_{\mathrm{fin}} = B_{\mathrm{ent}}$,

3) $B_{\mathrm{fin}} = \{(d, d+1, d+2, \ldots, \ldots, -1, 0, 0) \mid d < 0\} \cup \{(d, 4d) \mid d \geqslant 0\}$.

Quasinets of bundles have been originally introduced and applied to proving properties of programs in Blikle [13], [15].

2.5. Quasinets of δ-relations

In the majority of approaches to mathematical theories of programs, mathematical objects representing behaviour of programs are sets of so-called input-output pairs. The latter are pairs (a_1, a_2) of states with the property that there exists a finite computation $(b_1, \ldots, b_n)$ with $a_1 = b_1$ and $a_2 = b_n$. A set of pairs is nothing but a binary relation, hence binary relations are the usual concepts in carrying out the input-output analysis.

Dealing with input-output pairs one must neglect, however, the infinitistic behaviour of algorithms. In other words, one is not able to distinguish between inputs that are not in the domain of the algorithm because all computations they generate terminate before reaching the end of the algorithm (blocking), and inputs that are not in the domain because all computations they generate are infinite (looping). In many cases this distinction is fairly important, e.g., in the case we are dealing with the termination problem. We may describe it clearly by means of bundles but this introduces additional information (computations instead of pairs) that may not actually be needed. In fact, calculations on bundles are usually more complicated than these on relations. To deal with relations and be able to describe infinitistic behaviour at the same time, we add to our relations so-called input-infinity pairs. What we get in this way are the δ-relations described below.

Let D be an arbitrary (finite or infinite) set called, as in the preceding case, the *set of states*. Let δ be an arbitrary object not in D, and let

$$D_\delta = D \cup \{\delta\},$$
$$\Delta = \{(a, \delta) \mid a \in D_\delta\},$$
$$Rel_\delta(D) = \{R \cup S \mid R \subseteq seq^2(D) \;\&\; S \subseteq \Delta \;\&\; (\delta, \delta) \in S\}.$$

Elements of $Rel_\delta(D)$ are called *δ-relations* over D. Given R in $Rel_\delta(D)$ we shall write aRb instead of $(a, b) \in R$.

Interpretation. If R in $Rel_\delta(D)$ is assumed to describe the behaviour of an algorithm A, then for any a, b in D we read:

aRb as *there exists a finite run in A that starts with the state-vector* (**begin**, a) *and terminates with* (**end**, b),

$aR\delta$ as *there exists an infinite run in A that starts with* (**begin**, a).

The assumption that $\delta R \delta$ for any δ-relation R has a technical character and plays the same part here as the assumption that ε is in any bundle.

Some authors (e.g., Cadiou [18]) dealing with binary partial relations, introduce a so-called "undefined element" ω and set $dR\omega$, whenever d is not in the domain (left domain) of R. In this way they get total in place of partial relations. It should be strongly emphasized that this is not the case for δ-relations which are partial, i.e., for any R in $Rel_\delta(D)$ its domain $\{d \mid (\exists d_1 \in D_\delta) dRd_1\}$ may be a proper subset of D_δ. If d is not in the domain of R, then it should be interpreted as "*d involves blocking*"—any run that starts with (**begin**, d) terminates with some (c, d), where c is a control state different from **end**. ■

EXAMPLE 2.5.1. Consider the following program:

```
begin
    while $x < 0$ do $x := x-1$;
    $x := x^{-1}$;
end.
```

The δ-relation in the set of all reals which corresponds to this program is the following: $R = \{(d, d^{-1}) |\ d > 0\} \cup \{(d, \delta) |\ d < 0\} \cup \{(\delta, \delta)\}$. The real number "0" is not in the domain of R because for $x = 0$ our program is blocked before reaching the end.

The operation of composition is defined in $Rel_\delta(D)$ in the usual way: for any R_1 and R_2

$$aR_1 \circ R_2 b \equiv (\exists c)\, aR_1 c \,\&\, cR_2 b.$$

Let $I_\delta = \{(d, d) |\ d \in D_\delta\}$. It is easy to prove that $QNR(D) = \{Rel_\delta(D), \subseteq, \circ, I_\delta\}$ is a set-theoretic quasinet with $\mathbf{0} = \{(\delta, \delta)\}$.

A δ-relation R is said to be *finitistic* provided $R \cap \Delta = \mathbf{0}$ and is said to be *inherently infinitistic* provided $R \subseteq \Delta$. Finitistic δ-relations constitute a net, while $QNR(D)$ is not a net. In fact, for any R in $Rel_\delta(D)$, we have

$$\begin{aligned} \mathbf{0} \circ R &= \mathbf{0}, \\ R \circ \mathbf{0} &= R \cap \Delta. \end{aligned} \tag{2.5.1}$$

Equation (2.5.1) has an interpretation analogous to that of (2.4.1).

Similarly as for languages and bundles, we can define here a generalized composition of δ-relations. Let $R_1, R_2, \ldots$ be an arbitrary infinite sequence of δ-relations.

(1) $C[\varepsilon] = I_\delta$,
$C[R_1, \ldots, R_n] = C[R_1, \ldots, R_{n-1}] \circ R_n$ for $1 \leqslant n < \infty$,

(2) $C[R_1, R_2, \ldots] = \{(a, \delta) |\ (\exists (a_1, a_2, \ldots) \in seq^\infty(D_\delta))[a_1 = a \,\& \,(\forall i \geqslant 1)(a_i R_i a_{i+1})]\}$.

Quasinets of δ-relations have been originally introduced in Blikle [15], [16].

3. EQUATIONS IN QUASINETS

Equations in quasinets are the main algebraic tools in our approach. They are of course the well-known "continuous" equations in complete lattices as used e.g. in [4], [3], [5], [9], [17], [22], [41]. There is therefore nothing new in this section; just a short review to make the paper self-contained and to introduce a uniform notation.

3.1. Existence of solutions

Let $N = (U, \leqslant, \circ, e)$ be an arbitrary quasinet to be fixed for the rest of Section 3. The most general form of a set of equations in N, with which we shall deal, is the following:

$$\begin{aligned} x_1 &= \varphi_1(x_1, \ldots, x_n), \\ &\ldots\ldots\ldots\ldots \\ x_n &= \varphi_n(x_1, \ldots, x_n), \end{aligned} \tag{3.1.1}$$

where φ_i: $U^n \to U$ for $i = 1, \ldots, n$. Clearly, this set of equations can be written as one "vectorial" equation

$$\boldsymbol{x} = \Phi(\boldsymbol{x}),$$

where Φ: $U^n \to U^n$, $\boldsymbol{x} = (x_1, \ldots, x_n)$ and $\Phi(\boldsymbol{x}) = (\varphi_1(\boldsymbol{x}), \ldots, \varphi_n(\boldsymbol{x}))$. Notice that U^n constitute a product quasinet with componentwise ordering and componentwise composition.

Each fixpoint of Φ is said to be a *solution* of (3.1.1). If $\boldsymbol{a}$ is a fixpoint of Φ and for any other fixpoint $\boldsymbol{b}$ of Φ we have $\boldsymbol{a} \leqslant \boldsymbol{b}$, resp. $\boldsymbol{a} \geqslant \boldsymbol{b}$, in the product quasinet, then $\boldsymbol{a}$ is said to be the *least solution* (*least fixpoint*), resp. the *greatest solution* (*greatest fixpoint*), of (3.1.1) (of Φ).

Fixpoints (in particular the least and the greatest) need not always exist. Below we quote some well-known theorems concerning this problem.

A function Φ: $U^n \to U^m$ is said to be *monotone* if for any $\boldsymbol{a}$ and $\boldsymbol{b}$ in U^n

$$\boldsymbol{a} \leqslant \boldsymbol{b} \quad \text{implies} \quad \Phi(\boldsymbol{a}) \leqslant \Phi(\boldsymbol{b}).$$

Clearly, if

$$\Phi(\boldsymbol{x}) = (\varphi_1(\boldsymbol{x}), \ldots, \varphi_m(\boldsymbol{x})) \quad \text{with} \quad \varphi_i\colon U^n \to U \quad \text{for } i \leqslant m,$$

then Φ is monotone (continuous) iff all φ_i are monotone (continuous). Moreover, each continuous function is monotone but not conversely.

THEOREM 3.1.1 (Tarski [42]). *If Φ: $U^n \to U^n$ is monotone, then the least fixpoint and the greatest fixpoint of Φ exist.*

THEOREM 3.1.2 (Kleene [25]). *If Φ: $U^n \to U^n$ is continuous, then the least fixpoint of Φ exists and is equal to*

$$\bigcup_{i=1}^{\infty} \Phi^i(\boldsymbol{0}),$$

where $\Phi^i(\boldsymbol{x}) = \Phi(\ldots \Phi(\boldsymbol{x}) \ldots)$ i-times and $\boldsymbol{0} = (0, \ldots, 0) \in U^n$.

3.2. Elimination of variables

Consider the set of equations (3.1.1) and suppose we want to solve it. Can we eliminate the unknown x_1 by solving $x_1 = \varphi_1(x, \ldots, x_n)$ with respect to x_1 and then substituting this solution for all appearances of x_1 in the remaining equations? Can we repeat this operation for all x_i and solve the set in this way? Formal answers to these questions are given below.

Let $\Phi(\boldsymbol{x}, \boldsymbol{y})$ be an arbitrary function of $n+m$ arguments, where $\boldsymbol{x} = (x_1, \ldots, x_n)$ ranges over U^n, $\boldsymbol{y} = (y_1, \ldots, y_m)$ ranges over U^m and Φ ranges over U^n, i.e., Φ: $U^{n+m} \to U^n$. Of course, $(\boldsymbol{x}, \boldsymbol{y})$ is used to abbreviate $(x_1, \ldots, x_n, y_1, \ldots, y_m)$.

If there exists a function Γ: $U^m \to U^n$ such that for any $\boldsymbol{a}$ in U^m, $\Gamma(\boldsymbol{a})$ is the least fixedpoint of $\Phi(\boldsymbol{x}, \boldsymbol{a})$, then Γ will be called the *resolvent* of Φ with respect to $\boldsymbol{x}$ and will be written

$$\Gamma(\boldsymbol{y}) = (\mu \boldsymbol{x})\Phi(\boldsymbol{x}, \boldsymbol{y}).$$

Clearly, $\Gamma(\boldsymbol{y})$—if it exists—can be said to be the *least solution* of the following parametric equation:

$$\boldsymbol{x} = \Phi(\boldsymbol{x}, \boldsymbol{y}).$$

Moreover, if $m = 0$ and $(\mu x)\Phi(x)$ exists, then it is the least fixedpoint of Φ. The following theorem is also well-known:

THEOREM 3.2.1. *For every continuous function $\Phi(x, y)$, where x and Φ range over the same U^n, the resolvent $(\mu x)\Phi(x, y)$ exists and is again a continuous function.*

Two sets of equations are said to be *equivalent* provided their sets of solutions are equal and are said to be *ls-equivalent* provided their least solutions exist and are equal.

THEOREM 3.2.2 (Bekić [5], Leszczyłowski [26]). *Consider a set of equations of the form*

$$x = \Phi(x, y),\quad y = \Psi(x, y), \tag{3.2.1}$$

where $\Phi\colon U^{n+m} \to U^n$, $\Psi\colon U^{n+m} \to U^m$, x ranges over U^n and y over U^m. If Φ and Ψ are continuous and $\Gamma(y) = (\mu x)\Phi(x, y)$, then the above set is ls-equivalent to the following:

$$x = \Gamma(y),$$
$$y = \Psi(\Gamma(y), y).$$

THEOREM 3.2.3. *Let Φ and Ψ be defined as above. The set* (3.2.1) *is equivalent to the following*:

$$x = \Phi(x, \Psi(x, y)),$$
$$y = \Psi(x, y).$$

The above theorems permit us to eliminate variables in sets of equations (in particular to solve equations in this way), provided we know how to compute the resolvents of the appropriate functions. The following lemma gives resolvents for functions that appear frequently in the sequel.

LEMMA 3.2.1. *For any a and b in U*

(1) $(\mu x)(ax \cup b) = a^*b$,

(2) $(\mu x)(xa \cup b) = ba^*$,

(3) $(\mu x)(axb \cup c) = \bigcup_{i=0}^{\infty} a^n cb^n$.

Clearly, a, b and c can be replaced by arbitrary functions that are constant with respect to x. The proof consists in a direct application of the Kleene approximation theorem (Theorem 3.1.2).

EXAMPLE 3.2.1. We shall solve a set of equations by elimination of variables. Consider the set

$$x = ax \cup by,$$
$$y = x \cup ay \cup c.$$

It is *ls*-equivalent to

$$x = a^*by,$$
$$y = (a^*b \cup d)y \cup c,$$

and the latter to

$$x = a^*b(a^*b \cup a)^*c,$$
$$y = (a^*b \cup a)^*c,$$

hence, this is the least solution of the set. ■

4. MAZURKIEWICZ ALGORITHMS IN ABSTRACT QUASINETS

Mazurkiewicz PD-algorithms are adequate mathematical models for a large class of programs including iterative programs and recursive procedures without parameters. Given a program to be analyzed, we fix an appropriate abstract algorithm, according to the control structure of the program, and a quasinet to be dealt with, depending on what we want to prove about the program. Below we define Mazurkiewicz algorithms in the general case of abstract quasinets.

4.1. The definition

Let $N = (U, \leqslant, \circ, e, C)$ be an arbitrary quasinet with generalized composition to be fixed for the rest of Section 4.

By a *pushdown algorithm* (abbr. PD-*algorithm*) over N we mean any system

$$A = (V, \alpha_1, \varepsilon, P),$$

where $V = \{\alpha_1, \ldots, \alpha_n\}$ is an arbitrary finite set of characters called *labels*; $\alpha_1 \in V$ and α_1 is called the *initial label*; ε is the empty string (of labels); P is a finite set of triples of the form (α_i, a, z), where $a \in U$, $\alpha_i \in V$, $z \in V^*$, such that for any $\alpha_i \in V$ there exists (α_j, a, z) in P with $\alpha_j = \alpha_i$.

Each triple (α_i, a, z) in P is called an *instruction* and can be interpreted as a procedure declaration, where α_i is the procedure name, a is a "segment" of procedure body, free of other procedure calls—we call it the *action* of the instruction—and z is a string of procedure names to be called (activated) in succession as they appear in z. If $z = \varepsilon$, then z is interpreted as the **end** symbol.

EXAMPLE 4.1.1. Consider the following recursive procedure:

```
procedure FACTORIAL;
    begin
    if $n = 0$ then $s := 1$
            else begin
            $n := n-1$;
            FACTORIAL;
            $n := n+1$;
            $s := s \times n$;
            end;
    end
```

The corresponding PD-algorithm is the following:

$$V = \{\alpha_1, \alpha_2\},$$

$\boldsymbol{P}$ consists of the three instructions,

$(\alpha_1, [\textbf{if } n = 0 \textbf{ then } s := 1], \varepsilon)$,
$(\alpha_1, [\textbf{if } n \neq 0 \textbf{ then } n := n-1], \alpha_1\alpha_2)$,
$(\alpha_2, [n := n+1;\ s := s \times n], \varepsilon)$.

The actions of these instructions may now be interpreted, depending on what we want to prove about the procedure, either as one character languages (cf. example of a program in Section 2.4), or as δ-relations in the set Re × Re (Re—the set of reals) or as appropriate bundles of computations, e.g., $[\textbf{if } n \neq 0 \textbf{ then } n := n-1] = \{([n, s], [n-1, s]) |\ n \neq 0 \;\&\; n, s \in \mathrm{Re}\}$.

It is to be explained that, in contrast to other authors (e.g., de Bakker [1]), we do not follow the ALGOL style, where a distinction is made between a set of procedure declarations (references) and a program containing their calls. In this approach, procedures are nothing but programs or their modules. It is clear that this assumption has only a technical character and does not restrict the generality of our investigations.

4.2. Operational semantics

As mentioned in the Introduction, each algorithm defines a resulting element in U (e.g., a relation, a bundle or a language), to be called outcome in the sequel. How outcomes are defined, clearly depends on the way we suppose our algorithm to be executed. This will be described below.

Consider an arbitrary PD-algorithm $A = (V, \alpha_1, \varepsilon, \boldsymbol{P})$ over $\boldsymbol{N}$. Strings in V will be called *control states* of the algorithm. Given an instruction (α_i, a, z), the pair (α_i, z) describes a binary relation in V, called the *control relation* of the instruction. We denote this relation, which in fact is a function, by $(\alpha_i \to z)$ and, for any y_1, y_2 in V^*, we define:

$$y_1(\alpha_i \to z)y_2 \equiv (\exists w \in V^*)[y_1 = \alpha_i w \;\&\; y_2 = zw].$$

In this way each instruction can modify the actual control state, provided it is applicable to that state.

Consider now an arbitrary label α_i and a finite or infinite sequence of instructions

$$(\alpha_{i_1}, a_1, z_1), \ldots, (\alpha_{i_m}, a_m, z_m); \quad m \leqslant \infty,$$

such that there exists a sequence of control states

$$y_1, \ldots, y_{m+1}$$

with the following properties:

(1) $y_1 = \alpha_i = \alpha_{i_1}$ and $(\forall j \leqslant m)[y_j(\alpha_{i_j} \to z_j)y_{j+1}]$.
(2) If $m < \infty$, then $y_{m+1} = \varepsilon$.

Each such a sequence of instructions will be called an *α_i-run* of the algorithm. The corresponding sequence of actions

$$(a_1, \ldots, a_m)$$

is called an *α_i-trace* of the algorithm. Elements of this sequence are actions that can be considered as consecutive in the algorithm. Therefore, the outcome generated by this trace is

$$C[a_1, \ldots, a_m]$$

(cf. the interpretation of composition given in Section 2.1). Given a label α_i, we now have more than one α_i-trace in A. The set of all α_i-traces (finite and infinite) will be denoted by

$$Tr(\alpha_i).$$

Each of these traces produces its own outcome, therefore, the outcome of the whole set will be the join

$$\bigcup \{C(t) |\ t \in Tr(\alpha_i)\},$$

since any two traces can be considered as independent executions of the algorithm.

For each $\alpha_i \in V$ we now define:

$Run(\alpha_i) = \bigcup \{C(t) \mid t \in Tr(\alpha_i)\}$—*$\alpha_i$-run element*,
$Tail(\alpha_i) = \bigcup \{C(t) \mid t \in FinTr(\alpha_i)\}$—*$\alpha_i$-tail element*,
$Perp(\alpha_i) = \bigcup \{C(t) \mid t \in InfTr(\alpha_i)\}$—*$\alpha_i$-perpetual element*,

where *Fin* and *Inf* operators are defined as in Section 2.2. We have, of course, for any α_i in V:

$$Run(\alpha_i) = Tail(\alpha_i) \cup Perp(\alpha_i).$$

In the sequel, by *Run*, *Tail*, *Perp*, *Tr*, *FinTr* and *InfTr*, we shall mean respectively the vectors $(Run(\alpha_1), \ldots, Run(\alpha_n))$, $(Tail(\alpha_1), \ldots, Tail(\alpha_n))$, etc. In analyzing algorithms we shall be especially interested in the following outcomes:

$Run(\alpha_1)$—the entire outcome of the algorithm,
$Tail(\alpha_1)$—the finitistic outcome of the algorithm,
$Perp(\alpha_1)$—the infinitistic outcome of the algorithm.

EXAMPLE 4.2.1. Consider an abstract PD-algorithm with the same control structure as the algorithm in Example 4.1.1:

$$P = \{(\alpha_1, a, \varepsilon); (\alpha_1, b, \alpha_1\alpha_2); (\alpha_2, c, \varepsilon)\}.$$

The following are examples of α_1-runs with explicitly written intermediate control states:

$\alpha_1(\alpha_1, a, \varepsilon)^\varepsilon$,
$\alpha_1(\alpha_1, b, \alpha_1\alpha_2)\alpha_1\alpha_2(\alpha_1, a, \varepsilon)\alpha_2(\alpha_2, c, \varepsilon)^\varepsilon$,
$\alpha_1(\alpha_1, b, \alpha_1\alpha_2)\alpha_1\alpha_2(\alpha_1, b, \alpha_1\alpha_2)\alpha_1\alpha_2\alpha_2(\alpha_1, b, \alpha_1\alpha_2)\alpha_1\alpha_2\alpha_2\alpha_2\ldots$

Therefore, examples of α_1-traces are the following:

(a), (b, a, c), (b, b, a, c, c), …—finite traces,
$(b, b, b, \ldots)$ —infinite trace.

As can now be proved,

$$Tr(\alpha_1) = \bigcup_{n=0}^{\infty} \{(b)\}^n \frown \{(a)\} \frown \{(c)\}^n \cup \{(b)\}^{\infty},$$

$$(4.2.1) \qquad Run(\alpha_1) = \bigcup_{n=0}^{\infty} b^n a c^n \cup b^{\infty},$$

$$Tail(\alpha_1) = \bigcup_{n=0}^{\infty} b^n a c^n,$$

$$Perp(\alpha_1) = b^{\infty}.$$

Notice that the operations appearing in the formula describing $Tr(\alpha_1)$ are operations in $NGL(\{a, b, c\})$, hence, traces can be considered as generalized words. Operations in the remaining formulas are of course operations in the quasinet N, over which the algorithm is defined.

Suppose now that we want our algorithm to be a model of the recursive procedure described in Example 4.1.1. To this effect, we first have to specify the quasinet N and then we have to specify a, b and c in this quasinet.

First, let $N = NGL(\{A, B, C\})$, where A, B and C are one-character names of the following modules:

A is the name of "**if** $n = 0$ **then** $s := 1$",

B is the name of "**if** $n \neq 0$ **then** $n := n-1$",

C is the name of "$n := n+1$; $s := s \times n$",

whatever the meanings of these modules can be. Now we set $a = \{A\}$, $b = \{B\}$ and $c = \{C\}$ and, from (4.2.1), we can conclude the following:

(1) Each finite run of the program (if any) consists in performing the module B a finite number of times, then the module A, and then the module C the same number of times as for B.

(2) There is at most one infinite run of the algorithm, which consists in repeating the module B an infinite number of times.

Let now $N = QNR(\mathrm{Re} \times \mathrm{Re})$, where Re is the set of all real numbers. We set a, b and c to be the following δ-relations in $Rel_\delta(\mathrm{Re} \times \mathrm{Re})$:

$$a = \{([0, s], [0, 1]) |\ s \in \mathrm{Re}\} \cup \{(\delta, \delta)\},$$
$$b = \{([n, s], [n-1, s]) |\ n, s \in \mathrm{Re} \;\&\; n \neq 0\} \cup \{(\delta, \delta)\},$$
$$c = \{([n, s], [n+1, s \times (n+1)]) |\ n, s \in \mathrm{Re}\} \cup \{(\delta, \delta)\}.$$

In this case we can prove much more about the algorithm on the basis of (4.2.1) (detailed calculations are given in Example 6.4.1):

(3) $Tail(\alpha_1) = \{([n, s], [n, n!]) |\ n \in \mathrm{Nat} \;\&\; s \in \mathrm{Re}\} \cup \{(\delta, \delta)\}$, where Nat is the set of all natural numbers. In other words, given natural n and arbitrary s our algorithm runs a finite amount of time and preserves the value of n, while the value of s becomes $n!$.

(4) $Perp(\alpha_1) = \{([n, s], \delta) |\ n \in \mathrm{Re} - \mathrm{Nat} \;\&\; s \in \mathrm{Re}\} \cup \{(\delta, \delta)\}$, i.e., if the initial value of n is not a natural number, then—independently of the value of s—our algorithm does not terminate its run.

Suppose now we are not satisfied with the input-output description of the algorithm, but we want to know what is happening during each run. In such a case we set $N = QNB(\mathrm{Re} \times \mathrm{Re})$ and we assume a, b and c to be the following bundles of computations:

$$a = \{([0, s], [0, 1]) |\ s \in \mathrm{Re}\} \cup \{\varepsilon\},$$
$$b = \{([n, s], [n-1, s]) |\ n, s \in \mathrm{Re} \;\&\; n \neq 0\} \cup \{\varepsilon\},$$
$$c = \{([n, s], [n+1, s \times (n+1)]) |\ n, s \in \mathrm{Re}\} \cup \{\varepsilon\}.$$

It is to be emphasized here that, while the choice of interpretations for a, b, c was rather obvious in the case of relations, it is not at all obvious in the case of bundles. We have in fact assumed that all the bundles consist entirely of computations of length 2. This is nothing but an arbitrary choice depending on the way we want our procedure to be analyzed. In fact, we could assume c to be following:

$$c = \{([n, s], [n+1, s], [n+1, s \times (n+1)]) |\ n, s \in \mathrm{Re}\} \cup \{\varepsilon\},$$

or even longer, e.g., if we wanted to take into account that multiplication is performed by successive addition.

Returning now to our former interpretations of a, b and c, (4.2.1) permits us to show the following:

(5) $Tail(\alpha_1) = \{([n, s], [n-1, s], \ldots, [1, s], [0, s], [0, 1], [1, 1], [2, 2], [3, 6], \ldots, [n, n!]) |\ n \in \mathrm{Nat} \;\&\; s \in \mathrm{Re}\} \cup \{\varepsilon\}$.

From this formula, besides the input-output characterization, we can draw a number of detailed conclusions. For example:

(i) the value of n never exceeds its initial value,

(ii) the value of s never exceeds the maximum of its initial value and $n!$,

(iii) the number of steps to compute $n!$ is $2n+1$, etc.

(6) $Perp(\alpha_1) = \{([n, s], [n-1, s], [n-2, s], \ldots) |\ n \in \mathrm{Re} - \mathrm{Nat} \;\&\; s \in \mathrm{Re}\} \cup \{\varepsilon\}$.

Analyzing this formula, we can see that in each infinite run the absolute value of n increases indefinitely, and hence an overflow will be signalized.

Concluding our example, we can formulate some general observations of methodological character:

(1) Every assertion we prove about a program (algorithm) within $NGL(D)$ can be also proved within quasinets of relations or bundles, but not conversely. Any theorem formulated in terms of languages is a kind of tautology—it is true for any program with the same control structure as the one under consideration.

(2) Every assertion we prove about a program within $QNR(D)$ can be derived from theorems proved about the program within quasinets of bundles of computations. The converse is not true.

(3) The above observations remain true for proving relationships between programs such as equivalence, inclusion, etc.

Mazurkiewicz algorithms and the concept of $Tail(\alpha_i)$ were introduced originally over nets of usual binary relations by Mazurkiewicz ([29], [30], [31], [32]). $Tail(\alpha_i)$ was defined in an equivalent but different way. The first extension of these ideas to other quasinets was given in Blikle [13] for the case of bundles, the first generalization to the abstract case in Blikle [14].

4.3. Fixedpoint semantics

Proving properties of algorithms (programs) consists usually in proving properties of three corresponding vectors: *Run*, *Tail*, and *Perp*. In consequence, any analysis of an algorithm must start with finding these vectors explicitely, i.e., with obtaining formulas describing them. This may turn out to be quite difficult, however, if we shall take the operational semantics as our starting point. For example, in order to find $Tail(\alpha_1)$, we have to specify $Fin\,Tr(\alpha_1)$, which requires the examination of all finite executions of the algorithm. There is no need to explain that an examination of executions is actually what we want to avoid in a mathematical analysis of programs.

In this, and the next two sections, we show how *Run*, *Tail* and *Perp* can be described in a nonoperational way, i.e., without refering to *Tr*, *Fin Tr* and *InfTr*. We prove namely, that the vectors *Run* and *Tail* are solutions of the so-called *canonical set of equations*, which can be effectively specified for each given PD-algorithm. Also for *Perp* we obtain a nonoperational description, however, not always in fixedpoint terms.

Let $A = (V, \alpha_1, \varepsilon, P)$ be an arbitrary PD-algorithm with $V = \{\alpha_1, \ldots, \alpha_n\}$. Without loss of generality (i.e., without changing *Run*, *Tail* and *Perp* vectors), we can assume that for any α_i in V and z in V^* there is at most one a with (α_i, a, z) in P. Indeed, if there are two instructions (α_i, a, z) and (α_i, b, z) with the same control relation, then we can replace them by $(\alpha_i, a\cup b, z)$. For every α_i and z appearing in an instruction, the only corresponding action will be denoted by a_{iz}. For every $i \leqslant n$ we shall set

$$V_i = \{z|\ (\exists a)(\alpha_i, a, z) \in P\}.$$

With any z in V^* we now associate a total function $\varphi_z\colon U^n \to U$ (n is the number of elements in V) defined as follows:

(1) If $z = \varepsilon$, then $\varphi_z(x_1, \ldots, x_n) = e$.

(2) If $z = \alpha_{i_1} \ldots \alpha_{i_m}$, then $\varphi_z(x_1, \ldots, x_n) = x_{i_1} \ldots x_{i_m}$.

For instance, if $z = \alpha_1\alpha_2\alpha_1$, then $\varphi_z(x_1, \ldots, x_n) = x_1x_2x_1$.

By the *canonical set of equations* (abbr. *CSE*) of the algorithm A, we shall mean the set

$$\begin{aligned} x_1 &= \bigcup_{z\in V_1} a_{1z}\varphi_z(x_1, \ldots, x_n), \\ &\ldots\ldots\ldots\ldots\ldots\ldots \\ x_n &= \bigcup_{z\in V_n} a_{nz}\varphi_z(x_1, \ldots, x_n). \end{aligned} \tag{4.3.1}$$

For example, in the case of the algorithm in Example 4.2.1 the *CSE* is the following:

$$\begin{aligned} x_1 &= bx_1x_2\cup a, \\ x_2 &= c. \end{aligned} \tag{4.3.2}$$

A PD-algorithm $A = (V, \alpha_1, \varepsilon, P)$ is said to be *admissible* if either $a_{iz}0 = 0$ for all $a_{iz} \in V_i$ and $i \leqslant n$, or $Fin\,Tr(\alpha_i) \neq \emptyset$ for $i \leqslant n$.

THEOREM 4.3.1. *For every admissible* PD-*algorithm the vector* $(Tail(\alpha_1), \ldots$ $\ldots, Tail(\alpha_n))$ *is the least solution of CSE.*

The proof is given in Section 4.4. It should be emphasized that the assumption about admissibility of the algorithm cannot be omitted in this theorem. A counterexample has been given by H. Müller (personal communication).

On the strength of this theorem, we can get $Tail(\alpha_1)$ by solving the appropriate *CSE*, e.g., considering the *CSE* (4.3.2), we can solve it in the following way:

$$\begin{aligned} x_1 &= bx_1x_2\cup a, \\ x_2 &= c, \end{aligned}$$

therefore,

$$\begin{aligned} x_1 &= bx_1c\cup a, \\ x_2 &= c, \end{aligned}$$

hence (by Lemma 3.2.1),

$$\begin{aligned} x_1 &= \bigcup_{n=0}^{\infty} b^n ac^n, \\ x_2 &= c. \end{aligned}$$

We get, therefore, $Tail(\alpha_1) = \bigcup_{n=0}^{\infty} b^n ac^n$ and $Tail(\alpha_2) = c$ (cf. Example 4.2.1).

To get a fixed point characterization of the *Run* vector, we now introduce an auxiliary notion. A sequence $t \in seq(U)$ is said to be *proper* (with respect to the given C operation) if it is either finite or, whenever it is infinite, it has the following property: for any a in U

$$C(t)\circ a = C(t).$$

The following facts about proper sequences can be easily observed:

(1) Each finite sequence is proper.

(2) Each sequence t with $C(t) = \mathbf{0}$ is proper.

(3) Each sequence in $QNR(D)$ is proper (since for any infinite t, $C(t) \subseteq \Delta$).

(4) Each infinite sequence $t = (L_1, L_2, \ldots)$ of languages with an infinite number of ε-free L_i's ($\varepsilon \notin L_i$) is proper (since in this case

$$C(t) \subseteq seq^\infty(D)).$$

(5) Each infinite sequence $t = (B_1, B_2, \ldots)$ of bundles with an infinite number of B_i's, where each nonempty computation is of length at least 2, is proper (since in this case

$$C(t) \subseteq seq^\infty(D)\cup\mathbf{0}).$$

(6) For any two sequences t_1 and t_2 in $seq(U)$, if t_1 is proper, then

$$C(t_1{}^\frown t_2) = C(t_1) \circ C(t_2).$$

A PD-algorithm is called *proper* whenever all its α_i-traces for $i \leqslant n$ are proper sequences. This notion seems to be fairly natural. In fact, any algorithm over $QNR(D)$ is proper. Moreover, if we consider an algorithm over $QNB(D)$ and we assume it to be adequate mathematical model of a real program, then the only actions (bundles) with computations of length 1 can be actions corresponding to such tests which have negligible time consumption. Since in practice we usually have no loops consisting of tests only, each trace of such an algorithm satisfies the property required in (5) and, therefore, the algorithm is proper. In a similar way, we can motivate this notion in $NGL(D)$.

THEOREM 4.3.2. *For any proper* PD-*algorithm the vector* $\big(Run(\alpha_1), \ldots, Run(\alpha_n)\big)$ *is a solution of CSE.*

The proof is in Section 4.4. It should be stressed that the assumption about properness cannot be omitted in the theorem. In fact, if we consider the algorithm from Example 4.2.1, where $a = \{A\}$, $b = \{B, \varepsilon\}$, $c = \{C\}_\infty$ are languages over $D \cdot = \{A, B, C\}$ with $B \neq C$, then the vector $\big(\bigcup_{n=0}^{\infty} b^n ac^n \cup b^\infty, c\big)$ is not a solution of the corresponding CSE.

In the general case, $\big(Run(\alpha_1), \ldots, Run(\alpha_n)\big)$ is neither the least nor the greatest solution of CSE. In consequence, there is no unambiguous fixedpoint description of the *Run* vector in the abtsract case. As we shall see however, there are such descriptions in each of the cases $NGL(D)$, $QNR(D)$ and $QNB(D)$.

The idea of a fixedpoint approach to programs is not new and has been developed by a number of authors. Some main references are: de Bakker and Scott [4], Bekić [5], Park [34], Scott [40], Scott and Strachay [41], de Bakker [1], Mazurkiewicz [29], de Bakker and Meertens [2], de Bakker and de Roever [3], Blikle and Mazurkiewicz [17], Cadiou [18], Manna and Vuillemin [28], Nivat [33], Hitchcock and Park [22]. All these papers deal with finitistic behaviour (least fixedpoints). The present approach extends the usual input-output analysis of programs to an input-output and input-looping analysis (δ-relations) and to an analysis of program executions (bundles). It also was the author's intent, not to develop here any formalized theory, but to suggest an algebraic tool applicable within usual mathematics in analyzing programs. An earlier exposition of the author's fixedpoint approach to programs was given in Blikle [10]–[16] and Blikle and Mazurkiewicz [17].

4.4. Proofs of the fixedpoint theorems

This section contains proofs of Theorems 4.3.1 and 4.3.2, together with some auxiliary notions and results.

Let $N = (U, \leqslant, \circ, e, C)$ be an arbitrary quasinet with a specified operation

$$C\colon seq(U) \to U.$$

We introduce now another operation

$$\mathsf{C}\colon GLan(U) \to U,$$

defined for any $S \subseteq seq(U)$, as follows:

$$\mathsf{C}(S) = \bigcup \{C(t) |\ t \in S\}.$$

This operation associates sets of sequences (traces) in U with elements of U. In fact, for any algorithm A and any label α_i in this algorithm, we have

$$\begin{aligned} Run(\alpha_i) &= \mathsf{C}\big(Tr(\alpha_i)\big), \\ Tail(\alpha_i) &= \mathsf{C}\big(Fin\,Tr(\alpha_i)\big), \qquad (4.4.1) \\ Perp(\alpha_i) &= \mathsf{C}\big(Inf\,Tr(\alpha_i)\big). \end{aligned}$$

LEMMA 4.4.1. *For any family* $\{S_p |\ p \in P\}$ *with* $S_p \subseteq seq(U)$ *we have*

$$\mathsf{C}\bigcup_{p \in P}(S_p) = \bigcup_{p \in P} \mathsf{C}(S_p).$$

Easy proof is left to the reader. A set $S \subseteq seq(U)$ is said to be *proper* if all its elements are proper sequences (see Section 4.3). The empty set is therefore a proper set. For two sets $S_1, S_2 \subseteq seq(U)$ let $S_1{}^\frown S_2$ denotes their concatenation within $NGL(U)$, as defined in Section 2.3.

LEMMA 4.4.2. *For any finite or denumerable sets* $S_1, S_2 \subseteq seq(U)$, *if* S_1 *is proper, then*

$$\mathsf{C}(S_1{}^\frown S_2) = \mathsf{C}(S_1)\mathsf{C}(S_2),$$

unless $S_1 \neq \emptyset = S_2$ *and* $\mathsf{C}(S_1)\mathbf{0} \neq \mathbf{0}$.

Proof is obvious and consists in application of Lemma 2.1.1 and property (6) of proper sequences (Section 4.3).

Now let $A = (V, \alpha_1, \varepsilon, P)$ be an admissible algorithm over N and let $V = \{\alpha_1, \ldots, \alpha_n\}$. By analogy to α_i-traces, we can define x-traces for arbitrary $x \in V^*$. $Tr(x)$ will denote the set of all x-traces. We shall assume $Tr(\varepsilon) = \{\varepsilon\}$.

LEMMA 4.4.3. *For any* x *in* $V^* - \{\varepsilon\}$, *if* $x = \alpha_{i_1} \ldots \alpha_{i_k}$, *then*

(1) $Tr(x) = Tr(\alpha_{i_1})^\frown \ldots {}^\frown Tr(\alpha_{i_k})$,

(2) $Fin\,Tr(x) = Fin\,Tr(\alpha_{i_1})^\frown \ldots {}^\frown Fin\,Tr(\alpha_{i_k})$,

(3) $Inf\,Tr(x) = Tr(\alpha_{i_1})^\frown \ldots {}^\frown Tr(\alpha_{i_{k-1}})^\frown Inf\,Tr(\alpha_{i_k})$.

Proof. The proofs of (1), (2) and (3) are, of course, analogous. To prove (1), we show that

$$Tr(x) = Tr(\alpha)^\frown Tr(x')$$

for any $x = \alpha^\frown x'$ with $\alpha \in V$ and $x' \in V^*$. Consider arbitrary t in $Tr(x)$ and let $y_1, \ldots, y_m$ with $m \leqslant \infty$ be a corresponding sequence of control states (Section 4.2). If there exists $k \leqslant m$ with $y_k = x'$, then some initial segment of t is in $Tr(\alpha)$ and the remainder must be in $Tr(x')$. If there is no k with this property, then t is an infinite trace in $Tr(\alpha)$, and therefore, $t \in Tr(\alpha)^\frown Tr(x')$. In consequence, $Tr(x) \subseteq Tr(\alpha)^\frown Tr(x')$.

Let $t \in Tr(\alpha)^\frown Tr(x')$ with $t = t_1{}^\frown t_2$, $t_1 \in Tr(\alpha)$, $t_2 \in Tr(x')$. If t_1 is infinite, then of course $t = t_1 \in Tr(x)$. If t_1 is finite, then let $y_1, \ldots, y_k$ and $z_1, \ldots, z_m$ be sequences of control states corresponding to t_1 and t_2, respectively. It is easy to show that $t_1{}^\frown t_2$ is an x-trace with the sequence of control states $y_1{}^\frown z_1, \ldots, y_n{}^\frown z_1, z_2, \ldots, z_m$. ∎

LEMMA 4.4.4. *For any $i \leqslant n$*

(1) $$Tr(\alpha_i) = \bigcup_{x\in V_i} \{(a_{ix})\}^\frown Tr(x),$$

(2) $$Fin\,Tr(\alpha_i) = \bigcup_{x\in V_i} \{(a_{ix})\}^\frown Fin\,Tr(x),$$

(3) $$Inf\,Tr(\alpha_i) = \bigcup_{x\in V_i-\{\varepsilon\}} \{(a_{ix})\}^\frown Inf\,Tr(x).$$

Proof is obvious by the definition of $Tr(\alpha_i)$ and $Tr(x)$.

Proofs of Theorems 4.3.1 *and* 4.3.2. Applying equations (4.4.1) and the above lemmas, we easily show that the *Tail* and the *Run* vectors are solutions of *CSE*. We shall show now that the former is the least solution of *CSE*.

Let $(d_1, \ldots, d_n)$ be an arbitrary solution of *CSE* and for any x in V^* and any $m \geqslant 1$ let $Tr^m(x)$ denote the set of all x-traces of length not greater than m. Clearly,

$$Fin\,Tr(x) = \bigcup_{m=1}^{\infty} Tr^m(x)$$

for any x in V^*. Let $t_i^m = \mathsf{C}(Tr^m(\alpha_i))$. By Lemma 4.4.1, we get therefore

$$Tail(\alpha_i) = \bigcup_{m=1}^{\infty} t_i^m$$

for $i = 1, \ldots, n$. We shall show that for $i \leqslant n$ and $m \geqslant 1$, $t_i^m \subseteq d_i$. The proof is by induction on m.

Initial step: $t_i^1 = a_{i\varepsilon} \subseteq \bigcup_{z\in V_i-\{\varepsilon\}} a_{iz}\varphi_z(d_1, \ldots, d_n) \cup a_{i\varepsilon} = d_i$.

Induction step: Let $t_i^k \subseteq d_i$ for $k \leqslant m$ and $i = 1, \ldots, n$. Of course, for any $i \leqslant n$

$$t_i^{m+1} = \mathsf{C}\big(Tr^{m+1}(\alpha_i)\big) = \mathsf{C}\big(\bigcup_{z\in V_i} \{(a_{iz})\}^\frown Tr^m(z)\big) = \bigcup_{z\in V_i} a_{iz}\mathsf{C}\big(Tr^m(z)\big).$$

It is also clear that for any $z = \alpha_{i(1,z)} \ldots \alpha_{i(l(z),z)}$ in V^*, where $l(z)$ is the length of z,

$$Tr^m(z) \subseteq Tr^m(\alpha_{i(1,z)})^\frown \ldots {}^\frown Tr^m(\alpha_{i(l(z),z)}).$$

Therefore, by monotonicity of composition and by the inductive assumption, we get

$$\begin{aligned} t_i^{m+1} &= \bigcup_{z\in V_i} a_{iz}\mathsf{C}\big(Tr^m(z)\big) \\ &\subseteq \bigcup_{z\in V_i} a_{iz}\mathsf{C}\big(Tr^m(\alpha_{i(1,z)})^\frown \ldots {}^\frown Tr^m(\alpha_{i(l(z),z)})\big) \\ &= \bigcup_{z\in V_i} a_{iz}\varphi_z(t_1^m, \ldots, t_n^m) \\ &\subseteq \bigcup_{z\in V_i} a_{iz}\varphi_z(d_1, \ldots, d_n) = d_i. \ ∎ \end{aligned}$$

4.5. Finite-control algorithms

Finite-control algorithms constitute a particular case of PD-algorithms and are models of programs without recursion (flowchart programs). Their theory is simpler and better developed and therefore it is worth to be described with more details.

Let $\boldsymbol{A} = (V, \alpha_1, \varepsilon, \boldsymbol{P})$ be a PD-algorithm with $V = \{\alpha_1, \ldots, \alpha_n\}$. This algorithm is said to be a *finite-control algorithm* (abbr. FC-*algorithm*), provided each instruction is either of the form (α_i, a, α_j) or of the form $(\alpha_i, a, \varepsilon)$ with $\alpha_i, \alpha_j \in V$. Instructions of this form have a very natural ALGOL-like interpretation:

(α_i, a, α_j) as "α_i: **do** a; **go to** α_j",
$(\alpha_i, a, \varepsilon)$ as "α_i: **do** a **end**".

Denote $\alpha_{n+1} = \varepsilon$. We shall make the assumption, as we did in Section 4.3, that for each α_i, α_j with $i \leqslant n, j \leqslant n+1$ there is at most one a with $(\alpha_i, a, \alpha_j) \in \boldsymbol{P}$. We shall denote this a by a_{ij} and we shall set $a_{ij} = \mathbf{0}$ whenever there is no a with $(\alpha_i, a, \alpha_j) \in \boldsymbol{P}$.

The *CSE* can be written now as follows:

$$\begin{aligned} x_1 &= a_{11}x_1 \cup \ldots \cup a_{1n}x_n \cup a_{1n+1}, \\ &\ldots \\ x_n &= a_{n1}x_1 \cup \ldots \cup a_{nn}x_n \cup a_{nn+1}. \end{aligned} \tag{4.5.1}$$

Whenever $\big(Tail(\alpha_1), \ldots, Tail(\alpha_n)\big)$ is the least solution of this set it can be computed effectively and expressed by means of a_{ij} and the operations $\cup$, $\circ$ and $*$ (Section 3.3). The vector $\big(Run(\alpha_1), \ldots, Run(\alpha_n)\big)$ can now be proved to be a solution of *CSE* without assuming that the algorithm is proper. Moreover, we get an algorithm (in a wide sense of the word) that, starting with the *CSE*, produces formulas expressing the *Run* vector by means of a_{ij} and the operations $\cup$, $\circ$, $*$ and ∞.

The main idea consists in a modified variable elimination method. While in computing the least solution we eliminate x_1 from the equation

$$x_1 = a_{11}x_1 \cup \ldots \cup a_{1n}x_n \cup a_{1n+1},$$

by setting (Lemma 3.2.1)

$$x_1 = a_{11}^*(a_{12}x_2 \cup \ldots \cup a_{1n}x_n \cup a_{1n+1}),$$

in order to get the *Run*-vector, we set

$$x_1 = a_{11}^*(a_{12}x_2 \cup \ldots \cup a_{1n}x_n \cup a_{1n+1}) \cup a_{11}^{\infty}.$$

To formulate and prove an appropriate lemma justifying the algorithm two auxiliary notions are needed.

The operation $\boldsymbol{C}$ (fixed at the beginning of Section 4.1) is said to be *adequate* provided for any proper set $S \subseteq seq(U)$ with $\varepsilon \notin S$ we have

$$\bigcup\{C(t) \mid t \in S^\infty\} = \big(\bigcup\{C(t) \mid t \in S\big)^\infty,$$

i.e.,

$$\mathsf{C}(S^\infty) = \big(\mathsf{C}(S)\big)^\infty. \tag{4.5.2}$$

Observe that S^∞ means the ∞-iteration of S within $NGL(U)$, while $(\mathsf{C}(S))^\infty$ is effectuated within the abstract quasinet N.

It can be proved (A. Lingas, personal communication), that there exist quasinets with nonadequate $\boldsymbol{C}$-operations. On the other hand, (4.5.2) is satisfied in all quasinets of our interest.

LEMMA 4.5.1. *The C-operations, as defined in the quasinets $NGL(D)$, $QNR(D)$ and $QNB(D)$, are adequate.*

The proof is in Blikle [14].

Let α_i, α_j be arbitrary labels with $i \leqslant n, j \leqslant n+1$. By an α_i, α_j-trace in A we mean any finite sequence $(a_1, \ldots, a_m)$ of elements of U such that there exist labels $\beta_1, \ldots, \beta_{m+1}$ with the following properties:

(1) $\beta_1 = \alpha_i$ and $\beta_{m+1} = \alpha_j$,

(2) $(\beta_p, a_p, \beta_{p+1}) \in \boldsymbol{P}$ for $p = 1, \ldots, m$.

We denote by $Tr(\alpha_i, \alpha_j)$ the set of all α_i, α_j-traces in A.

Note that $Tr(\alpha_i, \varepsilon) = Fin\,Tr(\alpha_i)$ for $i = 1, \ldots, n$.

LEMMA 4.5.2. *Consider two right-linear sets of equations:*

$$\begin{array}{l} x_1 = d_{11}x_1 \cup \ldots \cup d_{1n}x_n \cup d_{1n+1}, \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ x_n = d_{n1}x_1 \cup \ldots \cup d_{nn}x_n \cup d_{nn+1}, \end{array} \tag{4.5.3}$$

and

$$\begin{array}{l} x_1 = d_{11}^*(d_{12}x_2 \cup \ldots \cup d_{1n}x_n \cup d_{1n+1}) \cup d_{11}^\infty, \\ x_2 = d_{21}x_1 \cup \ldots \cup d_{2n}x_n \cup d_{2n+1}, \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ x_n = d_{n1}x_1 \cup \ldots \cup d_{nn}x_n \cup d_{nn+1}. \end{array} \tag{4.5.4}$$

Suppose that the $\boldsymbol{C}$ *operation in* N *is adequate and* (4.5.3) *satisfies the following condition:*

(*) *there exist sets $D_{ij} \subseteq seq(U)$, $i \leqslant n$, $j \leqslant n+1$, such that:*

(1) $\varepsilon \notin D_{ij}$ *and* $d_{ij} = \boldsymbol{C}(D_{ij})$ *for* $i \leqslant n$, $j \leqslant n+1$,

(2) $D_{ij} \subseteq Tr(\alpha_i, \alpha_j)$ *for* $i, j \leqslant n$,

(3) $Tr(\alpha_i) = D_{i1}{}^\frown Tr(\alpha_1) \cup \ldots \cup D_{in}{}^\frown Tr(\alpha_n) \cup D_{in+1}$ *for* $i \leqslant n$.

Then $(Run(\alpha_1), \ldots, Run(\alpha_n))$ *is a solution of* (4.5.4), *and* (4.5.4) *satisfies* (*) *as well.*

Proof. Note first that each $Tr(\alpha_i, \alpha_j)$ consists of finite traces and therefore each D_{ij} with $i, j \leqslant n$ is proper (Section 4.4). We shall now prove the equality

$$Tr(\alpha_1) = D_{11}^*(D_{12}Tr(\alpha_2) \cup \ldots \cup D_{1n}Tr(\alpha_n) \cup D_{1n+1}) \cup D_{11}^\infty, \tag{4.5.5}$$

where for simplification the symbols "$^\frown$" of concatenation are omitted. The inclusion "$\supseteq$" is obvious by (2) and (3) (the latter implies $D_{in+1} \subseteq Tr(\alpha_i)$ for $i \leqslant n$). To show the converse inclusion, consider an arbitrary t in $Tr(\alpha_1)$. By (3):

—either $t \in D_{1n+1}$ and we are done,

—or $t = t_1{}^\frown t_2$ with $t_1 \in D_{1j}$, $t_2 \in Tr(\alpha_j)$, $j \leqslant n$.

Two other cases are to be considered now:

—either $j > 1$, i.e., $t \in D_{ij}Tr(\alpha_i)$, and again we are done,

—or $j = 1$, i.e., $t_2 \in Tr(\alpha_1)$, and the reasoning must be repeated.

It is easy to see now that either $t \in (D_{11})^m D_{1j}Tr(\alpha_j)$ for some $m \geqslant 1$, $j \geqslant 1$ or $t \in (D_{11})^\infty$. In both cases (4.5.5) holds. Now applying the operation C to both sides of (4.5.5), we get (by Lemmas 4.4.1, 4.4.2 and by equation 4.5.2):

$$Run(\alpha_1) = d_{11}(d_{12}Run(\alpha_2) \cup \ldots \cup d_{1n}Run(\alpha_n) \cup d_{1n+1}) \cup d_{11}^\infty$$

and, by (3), we get, in a similar way,

$$Run(\alpha_i) = d_{i1}Run(\alpha_1) \cup \ldots \cup d_{in}Run(\alpha_n) \cup d_{in+1}$$

for $i \geqslant 2$. In effect, $(Run(\alpha_1), \ldots, Run(\alpha_n))$ is a solution of (4.5.4).

To prove that (4.5.4) satisfies (*), all we have to show is that

$$\varepsilon \notin D_{11}D_{1j} \quad \text{and} \quad D_{11}^*D_{1j} \subseteq Tr(\alpha_1, \alpha_j)$$

for $j \leqslant n$. This, however, is an immediate consequence of (*) for (4.5.3). ■

To get the algorithm producing *Run* vector from *CSE* the following facts must now be observed:

(1) If (4.5.3) is a *CSE* of an algorithm, then it satisfies (*), since we can set $D_{ij} = \{(d_{ij})\}$.

(2) Due to symmetry of variables the transformation described in the lemma permits us to eliminate each variable x_i with $d_{ii} \neq 0$ and preserves the property that $(Run(\alpha_1), \ldots, Run(\alpha_n))$ is a solution.

(3) The transformation can be repeated as long as there exists x_i in the set with $d_{ii} \neq 0$. If this is no longer the case, then the transformation is still applicable but has no effect. In this case we make a substitution, as described in Theorem

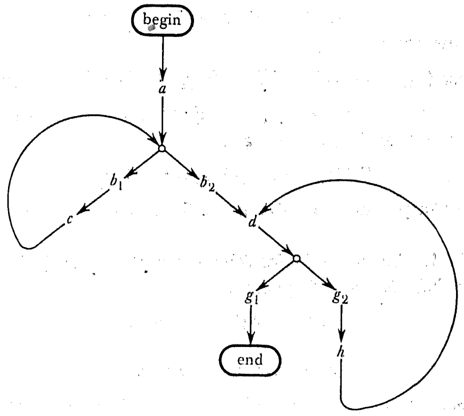

Fig. 4.5.1

3.2.3. This substitution preserves all solutions of the set and also preserves the property (*).

EXAMPLE 4.5.1. Consider the abstract FC-algorithm

$$\begin{array}{lll} (\alpha_1, a, \alpha_2) & (\alpha_3, c, \alpha_2) & (\alpha_5, g_1, \varepsilon) \\ (\alpha_2, b_1, \alpha_3) & (\alpha_4, d, \alpha_5) & (\alpha_5, g_2, \alpha_6) \\ (\alpha_2, b_2, \alpha_4) & & (\alpha_6, h, \alpha_4) \end{array}$$

with the flowchart of Fig. 4.5.1. Clearly the pairs of actions b_1, b_2 and g_1, g_2 represent tests in the program. The *CSE* of this algorithm is

$$\begin{aligned} x_1 &= ax_2, \\ x_2 &= b_1x_3 \cup b_2x_4, \\ x_3 &= cx_2, \\ x_4 &= dx_5, \\ x_5 &= g_2x_6 \cup g_1, \\ x_6 &= hx_4. \end{aligned} \tag{4.5.6}$$

Suppose now that all we want to get is $Run(\alpha_1)$. Therefore, we solve the *CSE* with respect to x_1, using the algorithm described in Lemma 4.5.2. After some obvious substitutions we get

$$\begin{aligned} x_1 &= ax_2, \\ x_2 &= b_1cx_2 \cup b_2dx_5, \\ x_5 &= g_2hdx_5 \cup g_1. \end{aligned}$$

We now apply our transformation to x_2 and x_5:

$$\begin{aligned} x_1 &= ax_2, \\ x_2 &= (b_1c)^*b_2dx_5 \cup (b_1c)^\infty, \\ x_5 &= (g_2hd)^*g_1 \cup (g_2hd)^\infty. \end{aligned}$$

Now we substitute again and get

$$Run(\alpha_1) = a(b_1c)^*b_2d\big((g_2hd)^*g_1 \cup (g_2hd)^\infty\big) \cup a(b_1c)^\infty.$$

Notice that, if we wish to find $Tail(\alpha_1)$, we apply to (4.5.6) the usual solving algorithm described in Section 3.2, and we get,

$$Tail(\alpha_1) = a(b_1c)^*b_2d(g_2hd)^*g_1.$$

In the case of FC-algorithms we also get a fixpoint characterization of the *Perp* vector.

THEOREM 4.5.1. *For any* FC-*algorithm the vector* $\big(Perp(\alpha_1), \ldots, Perp(\alpha_n)\big)$ *is a solution of the following set of equations*:

$$\begin{aligned} x_1 &= a_{11}x_1 \cup \ldots \cup a_{1n}x_n, \\ &\ldots \\ x_n &= a_{n1}x_1 \cup \ldots \cup a_{nn}x_n. \end{aligned} \tag{4.5.7}$$

In the general case Perp vector needs not to be neither the least nor the greatest solution of this set.

The proof is analogous to that of Theorem 4.3.2 and consists in successive application of Lemmas 4.4.4 eq. (3), 4.4.1 and 4.4.2. Also in this case we can use the algorithm described above to get the *Perp* vector from (4.5.7). We need, however, one more assumption about the algorithm to the effect that $a_{ij} \circ \mathbf{0} = \mathbf{0}$ for $i \leqslant n$, $j \leqslant n+1$ (details in [14]).

5. ALGORITHMS OVER BUNDLES OF COMPUTATIONS

This section is devoted to algorithms over $QNB(D)$. It is shown how these algorithms can be used in proving properties of programs and what sort of program properties can be investigated by these means.

5.1. Extension of the calculus of bundles

The calculus of bundles is extended here with some specific notation to be used in describing properties of programs. Since in the sequel we deal only with non-empty computations, they will be referred to, for simplicity, as computations.

Consider an arbitrary quasinet $QNB(D)$, let $f\colon D \to D$ be an arbitrary partial function and let $p\colon D \to \{true, false\}$ be an arbitrary partial predicate in D. We denote

$$\begin{aligned} [p(x)|\ x \leftarrow f(x)] &= \big\{(d, f(d))|\ p(d) = true \,\&\, d \in fD\big\} \cup \{\varepsilon\}, \\ [p(x)] &= \{(d)|\ p(d) = true\} \cup \{\varepsilon\}. \end{aligned}$$

We shall also write $[x \leftarrow f(x)]$ instead of $[true|\ x \leftarrow f(x)]$ and $[d]$ instead of $[x = d]$.

Let p and q be arbitrary predicates and let $c = (d_1, \ldots, d_m)$ with $1 \leqslant m \leqslant \infty$ be an arbitrary (finite or infinite) computation. We say that c *satisfies* p *at the entrance* if $p(d_1) = true$ and we say that c *satisfies* p *at the exit* if either $m = \infty$ or $m < \infty \,\&\, p(d_m) = true$. A bundle B is said to *satisfy* p *at the entrance* (*exit*) provided all computations in B satisfy p at the entrance (exit). Notice that inherently infinitistic bundles satisfy any predicate at the exit.

Bundles of the form $[p(x)]$ are subsets of the unit E in the monoid of bundles. In consequence, for any bundle B and any predicate p, we get the following subsets of B:

$[p(x)]B$—the bundle of all computations in B that satisfy p at the entrance,

$B[p(x)]$—the bundle of all computations in B that satisfy p at the exit.

The following facts about bundles, which appear frequently in analysis of programs, can now be easily formulated:

(1) $B = [p(x)]B[q(x)]$—B satisfies p at the entrance and q at the exit,

(2) $[p(x)]B = \mathbf{0}$—no computation in B satisfies p at the entrance,

(3) $B[p(x)] = \mathbf{0}$—no computation in B satisfies p at the exit,

(4) $[p(x)]B \subseteq B[q(x)]$—each computation in B that satisfies p at the entrance satisfies q at the exit (partial correctness in the case $B = Tail(\alpha_1)$).

Observe that (4) is equivalent to

$$[p(x)]B = [p(x)]B[q(x)]$$

and therefore (1) implies (4). In fact, (1) asserts, besides partial correctness, that each computation in B satisfies p at the entrance.

For any bundle B we set

$$Ent(B) = \{d|\ [d]B \neq \mathbf{0}\},$$
$$Ext(B) = \{d|\ B[d] \cap seq^*(D) \neq \mathbf{0}\},$$

and we call these sets the set of entries and the set of exits of B. A vector of bundles $(B_1, \ldots, B_n)$ is said to be *finitistic* (*inherently infinitistic*) if all B_i are finitistic (inherently infinitistic); cf. Section 2.4.

5.2. Fixedpoint theorems in $QNB(D)$

THEOREM 5.2.1. *For any proper* PD-*algorithm the vector* $\big(Run(\alpha_1), \ldots, Run(\alpha_n)\big)$ *is the greatest solution of CSE.*

Proof is in Blikle [15]. Also in this case the assumption about properness of the algorithm cannot be omitted.

COROLLARY 5.2.1. *Let* $(Q_1, \ldots, Q_n)$ *be an arbitrary solution of CSE of a proper* PD-*algorithm. If* $Q_i = Tail(\alpha_i) \cup S_i$ *with* $Tail(\alpha_i) \cap S_i = 0$ *for* $i \leqslant n$, *then*

$$S_i \subseteq Perp(\alpha_i)$$

for all $i \leqslant n$.

The proof is immediate since, by Theorem 5.2.1, $Tail(\alpha_i) \cup S_i \subseteq Tail(\alpha_i) \cup Perp(\alpha_i)$, but $Tail(\alpha_i) \cap S_i = 0$, hence $S_i \subseteq Perp(\alpha_i)$. ■

A PD-algorithm over $QNB(D)$ is said to be *finitistic* provided all its actions are finitistic. Of course every finitistic algorithm is admissible. Finitistic algorithms correspond to the usual programs we deal with, but the nonfinitistic case also has some reasonable motivation. In fact, to prove properties of a large program, we have to structure it into a number of modules. Each module is then considered independently and the corresponding $Run(\alpha_1)$ bundle is established. These bundles are assumed then to be actions of instructions of a global algorithm associated with the program. This global algorithm, of course, need not be finitistic.

COROLLARY 5.2.2. *For any proper and finitistic* PD-*algorithm the vector* $\big(Tail(\alpha_1), \ldots, Tail(\alpha_n)\big)$ *is the only finitistic solution of CSE.*

This is an immediate consequence of Corollary 5.2.1 since for proper algorithms $Perp(\alpha_i)$ is always inherently infinitistic.

THEOREM 5.2.2. *For any proper and finitistic* FC-*algorithm with the set of instructions* $P = \{(\alpha_i, B_{ij}, \beta_j)|\ i \leqslant n,\ j \leqslant n+1\}$ *the vector* $\big(Perp(\alpha_1), \ldots, Perp(\alpha_n)\big)$ *is the greatest solution of the following set of equations*:

$$\begin{array}{ll} & X_1 = B_{11}X_1 \cup \ldots \cup B_{1n}X_n, \\ (5.2.1) & \ldots\ldots\ldots\ldots\ldots \\ & X_n = B_{n1}X_1 \cup \ldots \cup B_{nn}X_n. \end{array}$$

Proof. Let $A = (V, \alpha_1, \varepsilon, \boldsymbol{P})$ be a proper and finitistic FC-algorithm with $\boldsymbol{P}$ as specified in the theorem and let $A_1 = (V, \alpha_1, \varepsilon, \boldsymbol{P}_1)$, where $\boldsymbol{P}_1$ is the result of setting $B_{in+1} = \mathbf{0}$ for $\bar{i} = 1, \ldots, n$ in $\boldsymbol{P}$. Clearly, (5.2.1) is the *CSE* for A_1. On the other hand, for all $i \leqslant n$

$$Tail_{A_1}(\alpha_i) = \complement\big(Fin\,Tr_{A_1}(\alpha_i)\big) = 0$$

and

$$Inf\,Tr_{A_1}(\alpha_i) = Inf\,Tr_A(\alpha_i),$$

hence

$$Run_{A_1}(\alpha_i) = Perp_A(\alpha_i).\ ■$$

5.3. Proving properties of programs by bundles of computations

Two examples are investigated in this section. The first concerns a common procedure with output. In this case bundles permit us to extend the well-known input-output analysis—as effectuated by relations—to an analysis of computations. Properties proved in this case are strong enough to imply partial and total correctness as particular corollaries. In the second example we investigate a system-like program that has no output at all (i.e., it always runs infinitely) and therefore cannot be analyzed by relations. Using bundles, we prove properties concerning deadlock and overload in the program.

EXAMPLE 5.3.1. Consider the well-known procedure NINETYONE

```
procedure NINETYONE;
  begin
      if $x > 100$ then $x := x-10$
      else begin
             $x := x+11$;
             NINETYONE;
             NINETYONE;
         end
  end
```

We assume D = Int and we set:

Notation	The algorithm
$Q = [x > 100\|\ x \leftarrow x-10]$,	$(\alpha_1, Q, \varepsilon)$,
$F = [x \leqslant 100\|\ x \leftarrow x+11]$,	$(\alpha_1, F, \alpha_1\alpha_1)$.

The *CSE* is

$$X = FXX \cup Q.$$

Now let

$$C_0 = [100]FQQ,$$
$$C_i = [100-i]FGC_{i-1} \quad \text{for} \quad 1 \leqslant i \leqslant 9,$$
$$C = C_0 \cup C_1 \cup \ldots \cup C_9.$$

We shall prove the following:

$$Tail(\alpha_1) = \bigcup_{i=1}^{\infty} F^i QCC_9^{i-1} \cup \bigcup_{i=1}^{\infty} F^i Q^2 C_9^{i-1} \cup Q. \tag{5.3.1}$$

This equation shows explicitly what all the finite computations of our procedure look like. To prove (5.3.1), we set

$$B_1 = \bigcup_{i=1}^{\infty} F^i QCC_9^{i-1}, \qquad B_2 = \bigcup_{i=1}^{\infty} F^i Q^2 C_9^{i-1}, \qquad R = B_1 \cup B_2 \cup Q.$$

Now, on the strength of Corollary 5.2.2, all we have to show is that R satisfies *CSE*. Observe first the following equations that are easily proved:

(1) $C = [91 \leqslant x \leqslant 100]C[91]$,
(2) $C_9 = [91]C_9[91]$,
(3) $B_1 = B_1[91]$,
(4) $B_2 = B_2[91]$,
(5) $[91]B_2 = \mathbf{0}$,
(6) $[91]B_1 = [91]FQC = [91]FQ[92]C = [91]FQC_8 = C_9$,
(7) $B_1 Q = B_1[91]Q = \mathbf{0}$,
(8) $B_2 Q = B_2[91]Q = \mathbf{0}$,
(9) $100[B_1] = \mathbf{0}$,
(10) $FQ = FQ[91 \leqslant x \leqslant 100]$,
(11) for all $1 \leqslant s \leqslant 9$,
$[90+s]B_1 = [90+s]FQ[91+s]C = [90+s]FQC_{10-(s+1)} = C_{10-s}$,
(12) $[91 \leqslant x \leqslant 99]B_2 = 0$.

Now we can show R to be a solution of *CSE*. Compute:

$$FRR \cup Q = F(B_1 B_1 \cup B_1 B_2 \cup B_1 Q \cup B_2 B_1 \cup B_2 B_2 \cup B_2 Q \cup QB_1 \cup QB_2 \cup Q^2) \cup Q.$$

Using equations (1)–(12), we now show the following:

$$FB_1 B_1 \underset{(3)}{=} FB_1[91]B_1 \underset{(6)}{=} FB_1 C_9 = \bigcup_{i=2}^{\infty} F^i QCC_9^{i-1},$$

$$FB_1 B_2 \underset{(3)}{=} FB_1[91]B_2 \underset{(5)}{=} \mathbf{0},$$

$$FB_1 Q \underset{(7)}{=} \mathbf{0},$$

$$FB_2 B_1 \underset{(4)}{=} FB_2[91]B_1 \underset{(6)}{=} FB_2 C_9 = \bigcup_{i=2}^{\infty} F^i Q^2 C_9^{i-1},$$

$$FB_2 B_2 \underset{(4)}{=} FB_2[91]B_2 \underset{(5)}{=} 0,$$

$$FB_2 Q \underset{(8)}{=} \mathbf{0},$$

$$FQB_1 \underset{(10)}{=} FQ[91 \leqslant x \leqslant 100]B_1 \underset{(9)}{=} FQ[91 \leqslant x \leqslant 99]B_1 \underset{(11)}{=} FQC,$$

$$FQB_2 \underset{(11)}{=} FQ[91 \leqslant x \leqslant 100]B_2 \underset{(12)}{=} FQ[100]B_2 = FQ[100]FQ^2 = FQC_0 \subseteq FQC.$$

In effect, we get

$$FRR \cup Q = \bigcup_{i=2}^{\infty} F^i QCC_9^{i-1} \cup FQC \cup \bigcup_{i=2}^{\infty} F^i Q^2 C_9^{i-1} \cup FQ^2 \cup Q = R.$$

Therefore, $R = Tail(\alpha_1)$.

Equation (5.3.1) shows any finite computation of the program to be an element of either B_1 or B_2 or Q. We can specify now the conditions under which each of these cases appear:

(1) for any $d > 100$,

$$[d]\,Tail(\alpha_1) = [d]Q = \{(d, d-10), \varepsilon\},$$

(2) for any $n = 0, 1, \ldots,$

$$\begin{aligned}[100-11\times n]\,Tail(\alpha_1) &= [100-11\times n]B_2\\ &= [100-11\times n]F^{n+1}[111]Q^2[91]C_9^n[91],\end{aligned}$$

(3) for any $n = 0, 1, \ldots$ and $t = 1, \ldots, 10$,

$$\begin{aligned}[100-11\times n-t]\,Tail(\alpha_1) &= [100-11\times n-t]B_1\\ &= [100-11\times n-t]F^{n+1}[111-t]Q[101-t]C[91]C_9^n[91].\end{aligned}$$

The three assertions above describe explicitly and totally the bundle of all finite computations of our program. In particular, we get here a proof of termination in the domain D = Int, since $Ent\big(Tail(\alpha_1)\big)$ = Int and the algorithm is deterministic in the sense that any two different computations are disjoint. In consequence, $Perp(\alpha_1) = \mathbf{0}$.

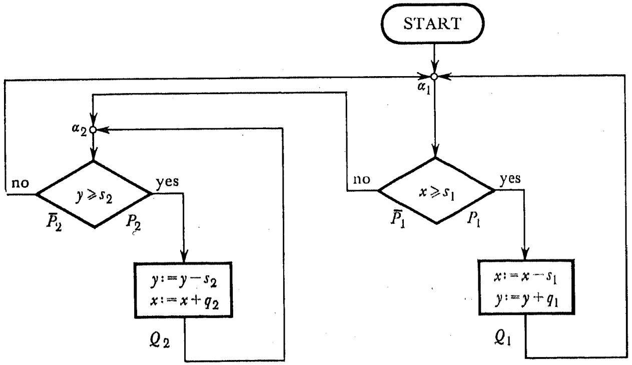

Fig. 5.3.1

EXAMPLE 5.3.2. Consider a sequential consumer-producer program, as given by the flowchart of Fig. 5.3.1. We shall assume s_1, s_2, q_1, q_2 to be arbitrary positive integers and x, y to be integer variables.

The program contains two operational modules Q_1 and Q_2. The former can be regarded as a routine consuming x and producing y, the latter—as a routine consuming y and producing x. The entire program is clearly supposed to run permanently, hence no output is expected. The program properties we shall analyze here concern occurrences of deadlock and overload. Deadlock means here that the program executes permanently the loop $(\overline{P}_1\overline{P}_2)^\infty$, and overload that for one or both variables x and y the sequence of corresponding successive values contains an infinite increasing sequence of numbers.

Notation	The algorithm
$P_1 = [x \geqslant s_1 \mid (x, y) \leftarrow (x, y)]$,	$(\alpha_1, P_1Q_1, \alpha_1)$
$\overline{P}_1 = [x < s_1 \mid (x, y) \leftarrow (x, y)]$,	$(\alpha_1, P_1, \alpha_2)$
$Q_1 = [(x, y) \leftarrow (x-s_1, y+q_1)]$,	$(\alpha_2, P_2Q_2, \alpha_2)$
$P_2 = [y \geqslant s_2 \mid (x, y) \leftarrow (x, y)]$,	$(\alpha_2, P_2, \alpha_1)$
$\overline{P}_2 = [y < s_2 \mid (x, y) \leftarrow (x, y)]$,	
$Q_2 = [(x, y) \leftarrow (x+q_2, y-s_2)]$,	
$D = \mathrm{Nat}\times\mathrm{Nat}$	

The *CSE* is

$$X_1 = P_1Q_1X_1 \cup \overline{P}_1X_2,$$
$$X_2 = P_2Q_2X_2 \cup \overline{P}_2X_1.$$

To get $Run(\alpha_1)$ from this set, we apply the algorithm described in Section 4.5:

$$X_1 = (P_1Q_1)^*\overline{P}_1X_2 \cup (P_1Q_1)^\infty,$$
$$X_2 = (P_2Q_2)^*\overline{P}_2X_1 \cup (P_2Q_2)^\infty.$$

It is easy to prove now that $(P_1Q_1)^\infty = (P_2Q_2)^\infty = \mathbf{0}$, hence

$$X_1 = (P_1Q_1)^*\overline{P}_1(P_2Q_2)^*\overline{P}_2X_1,$$

therefore

$$X_1 = ((P_1Q_1)^*\overline{P}_1(P_2Q_2)^*\overline{P_2})^*\mathbf{0} \cup ((P_1Q_1)^*\overline{P}_1(P_2Q_2)^*\overline{P}_2)^\infty,$$

and finally,

$$Run(\alpha_1) = ((P_1Q_1)^*\overline{P_1}(P_2Q_2)^*\overline{P_2})^\infty. \tag{5.3.2}$$

The deadlock-overload properties of the program depend clearly on the parameters s_1, s_2, q_1, q_2. Full analysis of all the cases would be too long for the purpose of this paper. We shall describe only some of them. Calculations can be found in Blikle [14].

Let n and m represent current values of x and y respectively. Denote

$$R = (P_1Q_1)^*\overline{P}_1(P_2Q_2)^*\overline{P}_2.$$

Case 1. $(q_1-s_1)+(q_2-s_2) > 0$: *overproduction*

Subcase 1.1. $q_1-s_1 > 0$, $q_2-s_2 > 0$, $n \geqslant s_1$, $m \geqslant s_2$. In this subcase we can prove the following:

(i) for every $k \geqslant 0$, $[(n, m)]R^k(\overline{P}_1\overline{P}_2)^\infty = \mathbf{0}$, i.e., deadlock never occurs,

(ii) overload does occure for both x and y.

Case 2. $(q_1-s_1)+(q_2-s_2) < 0$: *overdemand*

In this case, for any $n, m \geqslant 0$, there exists $k \geqslant 0$ such that

$$[(n, m)]\,Run(\alpha_1) = [(n, m)]\,R^k(\overline{P}_1\overline{P}_2)^\infty.$$

In other words, deadlock after some finite number of steps and, of course, no overload in consequence.

Case 3. $(q_1-s_1)+(q_2-s_2) = 0$: *balance*

Subcase 3.1. $q_1-s_1 = 0$, $q_2-s_2 = 0$, $n \geqslant s_1$, $m \geqslant s_2$. In this subcase we can prove the following:

(i) for any $k \geqslant 0$, $[(n, m)]R^k(\overline{P}_1\overline{P}_2)^\infty = \mathbf{0}$, i.e., deadlock never occurs,

(ii) for any $k \geqslant 0$, if $[(n, m)]T^k = [(n, m)]T^k[(n_1, m_1)]$, then $n+m = n_1+m_1$, which means that $x+y$ is an invariant of the program. In consequence, the overload will not occure.

6. ALGORITHMS OVER δ-RELATIONS

This section is devoted to algorithms over $QNR(D)$ considered as a tool in developing input-output analysis of programs. In fact, this analysis is extended now with an analysis of termination and looping.

6.1. An extension of the calculus of relations

The operation of composition of relations can be extended to the case in which one of the arguments (relations) is replaced by a set. We define this extension for the general case of arbitrary binary relations.

Let Ω be an arbitrary set. If $R \in Rel(\Omega)$ and $C \subseteq \Omega$, then CR and RC denote subsets of Ω defined as follows:

$$a \in CR \equiv (\exists c \in C)\, c\,R\,a,$$
$$a \in RC \equiv (\exists c \in C)\, a\,R\,c.$$

Notice that CR is the image of C with respect to R and RC is the appropriate coimage. In particular ΩR is the range of R and $R\Omega$ its domain. It is easy to prove that, for any $R_1, R_2 \in Rel(\Omega)$, $C \subseteq \Omega$ and families $\{R_i \mid i \in I\}$; $R_i \in Rel(\Omega)$, $\{C_j \mid j \in J\}$; $C_j \subseteq \Omega$, we have

$$R_1(R_2C) = (R_1R_2)C \qquad C(R_1R_2) = (CR_1)R_2$$
$$(\bigcup_{i\in I} R_i)C = \bigcup_{i\in I} R_iC \qquad C(\bigcup_{i\in I} R_i) = \bigcup_{i\in I} CR_i$$
$$(\bigcup_{j\in J} C_j)R_1 = \bigcup_{j\in J} C_jR_1 \qquad R_1(\bigcup_{j\in J} C_j) = \bigcup_{j\in J} R_1C_j.$$

Moreover, the functions $\Phi(R, C) = RC$ and $\Psi(R, C) = CR$ are monotone with respect to each of the variables.

In the sequel we shall frequently have to do with vectors of relations $(R_1, \ldots, R_n)$. By the domain and range of $(R_1, \ldots, R_n)$ we shall mean respectively the vectors of sets $(R_1\Omega, \ldots, R_n\Omega)$ and $(\Omega R_1, \ldots, \Omega R_n)$. The vector $(R_1, \ldots, R_n)$ is said to be *functional* if all of the R_i are functions.

To deal with concrete programs and prove their properties in a systematic way, we shall need an explicit calculus of relations. In preceding sections we described relations as sets of pairs, e.g., $Q = \{([n, s], [n(s-1), s-1])|\ n, s \in \mathrm{Re}\} \cup \cup \{(\delta, \delta)\}$. In this section we shall use a notation introduced by Mazurkiewicz for the case of functions in $Rel(D)$ and extended here to functions in $Rel_\delta(D)$. Let $f\colon D \to D$ be an arbitrary function and let $p\colon D \to \{true, false\}$ be an arbitrary predicate in D. We denote:

$$[p(x)|\ x{:} = f(x)] = \{(d, f(d))|\ p(d) = true \,\&\, d \in fD_\delta\} \cup \{(\delta, \delta)\}.$$

Notice that x in the above formula is a binded variable and, therefore, $[p(x)|\ x{:}=f(x)]$ and $[p(y)|\ y{:} = f(y)]$ denote the same function. We shall also write

$$[x{:} = f(x)] \quad \text{instead of} \quad [true|\ x{:} = f(x)],$$
$$[p(x)] \quad \text{instead of} \quad [p(x)|\ x{:} = x].$$

The following equalities are useful in calculations:

(1) $[p(x)|\ x{:} = f(x)][q(x)|\ x{:} = g(x)] = [p(x) \,\&\, q(f(x))|\ x{:} = g(f(x))]$,
(2) $[p(x)|\ x{:} = f(x)] \cup [q(x)|\ x{:} = f(x)] = [p(x) \vee q(x)|\ x{:} = f(x)]$,
(3) $[p(x)][q(x)] = [p(x) \,\&\, q(x)] = [q(x)][p(x)]$,
(4) If $p(x) \Rightarrow q(x)$, then $[p(x)][q(x)] = [p(x)]$.

6.2. *QNR(D)*-oriented theory of algorithms

This section contains the case oriented notions and theorems needed in proving properties of algorithms (programs) by δ-relations.

In the remainder of Section 6, δ-relations in $Rel_\delta(D)$ will be denoted by capitals $P, Q, R, S, \ldots$, possibly with indices, subsets of $D \cup \{\delta\}$ by capitals $A, B, C, \ldots$, also possibly indexed, and variables ranging over $Rel_\delta(D)$ by capitals $X, X_1, X_2, \ldots$ For any $B \subseteq D$ we shall set $B_\delta = B \cup \{\delta\}$, hence, in particular, $D_\delta = D \cup \{\delta\}$. We shall also say simply "relation" instead of "δ-relation".

Let $A = (V, \alpha_1, \varepsilon, P)$ with $V = \{\alpha_1, \ldots, \alpha_n\}$ be an arbitrary PD-algorithm over $QNR(D)$ to be fixed for the sequel. The *CSE* for this algorithm is the following:

$$X_1 = \bigcup_{z \in V_1} R_{1z}\varphi_z(X_1, \ldots, X_n),$$
$$\cdots\cdots\cdots\cdots\cdots\cdots$$
$$X_n = \bigcup_{z \in V_n} R_{nz}\varphi_z(X_1, \ldots, X_n),$$

where (α_i, R_{iz}, z) are instructions in $\boldsymbol{P}$ (cf. the assumption made in Section 4.3). It must be recalled that each algorithm in $QNR(D)$ is proper and, therefore, we shall not refer to this notion here.

The algorithm A is said to be finitistic provided all R_{iz} are finitistic (see Section 2.5) and is said to be nonfinitistic otherwise. For the majority of simple programs we deal, of course, with finitistic algorithms but, similarly as for $QNB(D)$, nonfinitistic algorithms may appear in analysis of large programs (see explanation in Section 5.2). Also in this case all finitistic algorithms are admissible.

Notice now that each instruction (α_i, R, z) can be considered as a binary relation in the set $V^* \times D$ of state vectors:

$$(y_1, d_1)(\alpha_i, R, z)(y_2, d_2) \equiv (\exists w)[y_1 = \alpha_i w \,\&\, y_2 = zw] \,\&\, d_1 R d_2$$

for $(y_1, d_1), (y_2, d_2) \in V^* \times D$. With the algorithm A we can associate now the next-state relation $\Rightarrow_A$ in $V^* \times D$ defined as follows:

$$(y_1, d_1) \Rightarrow_A (y_2, d_2) \equiv (\exists (\alpha_i, R, z) \in \boldsymbol{P})(y_1, d_1)(\alpha_i, R, z)(y_2, d_2).$$

It is interesting to note, that for each α_i in V and any d_1, d_2 in D

$$d_1\, Tail(\alpha_i) d_2 \equiv (\alpha_i, d_1)[\Rightarrow_A]^*(\varepsilon, d_2).$$

This is in fact the original Mazurkiewicz definition of $Tail(\alpha_i)$.

The algorithm A is said to be *deterministic* provided $\Rightarrow_A$ is a function and is said to be *nondeterministic* otherwise. A necessary (but not sufficient) condition for A to be deterministic is of course that all actions be functions.

To deal with the problem of termination of algorithms, we now introduce three auxiliary notions. Namely, for each label α_i in V, we set:

$$Stop(\alpha_i) = Tail(\alpha_i)D,$$
$$Loop(\alpha_i) = Perp(\alpha_i)\{\delta\} - \{\delta\},$$
$$Block(\alpha_i) = D - (Stop(\alpha_i) \cup Loop(\alpha_i)).$$

Remark: in the definition of $Stop(\alpha_i)$ we set on purpose D, rather than D_δ. Also, since $\{\delta\}$ is always the range of $Perp(\alpha_i)$, we have $Perp(\alpha_i)\{\delta\} = Perp(\alpha_i)D_\delta$.

In analyzing algorithms we shall be especially interested in the following particular sets:

$Stop(\alpha_1)$—the set of all inputs, where the algorithm terminates its run reaching the end,

$Loop(\alpha_1)$—the set of all inputs, where the algorithm runs indefinitely,

$Block(\alpha_1)$—the set of all inputs, where the algorithm terminates its run without reaching the end (is blocked).

LEMMA 6.2.1. *If A is deterministic, then $Run(\alpha_i)$, $Tail(\alpha_i)$ and $Perp(\alpha_i)$ are all functions for $i \leqslant n$.*

LEMMA 6.2.2. *If A is finitistic, then for each $i \leqslant n$, $Tail(\alpha_i) \cap Perp(\alpha_i) = \mathbf{0}$.*

LEMMA 6.2.3. *If A is finitistic, then for each $i \leqslant n$, if $Run(\alpha_i) = T_i \cup P_i$, where T_i is finitistic and P_i is inherently infinitistic, than $T_i = Tail(\alpha_i)$ and $P_i = Perp(\alpha_i)$.*

LEMMA 6.2.4. *If A is deterministic, then for each $i \leqslant n$, $Tail(\alpha_i) \cap Perp(\alpha_i) = \mathbf{0}$ and $Stop(\alpha_i) \cap Loop(\alpha_i) = \emptyset$.*

The proofs are obvious.

It has been proved in Section 5, that the *Run* vector of a proper algorithm is the greatest solution of *CSE* in *QNB*(*D*). It turns out, however, that this is not the case for *QNR*(*D*). Consider the following example of an algorithm, where D is the set of integers:

$$(\alpha_1, [x \geqslant 0|\ x := 2x], \varepsilon),$$
$$(\alpha_1, [x < 0|\ x := x-1], \alpha_1).$$

The *CSE* is now as follows:

$$X = [x < 0|\ x := x-1]X \cup [x \geqslant 0|\ x := 2x].$$

From this, by Theorem 4.3.1 and the algorithm, to get *Run* vector which was described in Section 4.5, we get

$$Tail(\alpha_1) = [x \geqslant 0|\ x := 2x],$$
$$Run(\alpha_1) = [x \geqslant 0|\ x := 2x] \cup [x < 0|\ x := \delta].$$

On the other hand, each function

$$Q_d = [x \geqslant 0|\ x := 2x] \cup [x < 0|\ x := d],$$

with $d \in D$, is also a solution of *CSE*. As can be proved,

$$Q = [x \geqslant 0|\ x := 2x] \cup \bigcup_{d \in D} [x < 0|\ x := d]$$

is the greatest one. Observe now that for each d,

$$[x < 0|\ x := d]D_\delta - \{\delta\} = Loop(\alpha_1).$$

Therefore, any of the above solutions consists of $Tail(\alpha_1)$ and of a component disjoint with $Tail(\alpha_1)$, whose domain is $Loop(\alpha_1)$.

This fact can be generalized into the following theorem:

THEOREM 6.2.1. *Let $(Q_1, \ldots, Q_n)$ be an arbitrary solution of CSE. If $Q_i = Tail(\alpha_i) \cup S_i$, where $Tail(\alpha_i) \cap S_i = \mathbf{0}$ for $i \leqslant n$, then*

$$S_i D_\delta - \{\delta\} \subseteq Loop(\alpha_i)$$

for $i \leqslant n$.

Proof. The proof is very similar to that of Theorem 5.2.1. Suppose that $(Q_1, \ldots, Q_n)$ is a solution of *CSE*, i.e.,

$$Q_i = \bigcup_{z \in V_i} R_{iz} \varphi_z(Q_1, \ldots, Q_n); \quad i = 1, \ldots, n. \tag{6.2.1}$$

We prove the following auxiliary assertion:

(*) For any control state $w \in V^*$, $w \neq \varepsilon$ and any $d, g \in D_\delta$ with $d\varphi_w(Q_1, \ldots, Q_n)g$ there exist:

—an instruction (α_j, R_{jz}, z),

—an element $d_1 \in D_\delta$ with $dR_{jz}d_1$ and $d_1 \varphi_w (Q_1, \ldots, Q_n)g$, where $w(\alpha_j \to z)w_1$ (see Section 4.2).

Now let $dS_i g$ for some $i \leqslant n$, $d, g \in D_\delta$. Therefore, $dQ_i g$, i.e., $d\varphi_{\alpha_i}(Q_1, \ldots, Q_n)g$, and, by (*), we get a sequence of instructions

$$(\alpha_{j_1}, R_{j_1 z_1}, z_1), \ldots, (\alpha_{j_s}, R_{j_s z_s}, z_s),$$

a sequence of corresponding control states

$$w_0, \ldots, w_s$$

and a sequence of elements in D

$$d_1, \ldots, d_s$$

with the following properties:

(i) $w_0 = \alpha_{j_1} = \alpha_i$,

(ii) $w_k(\alpha_{j_k} \to z_k)w_{k+1}$ for $k \leqslant s$,

(iii) $dR_{j_1 z_1}d_1$ and $d_{k-1}R_{j_k z_k}d_k$ for $1 < k \leqslant s$,

(iv) $d_s \varphi_{w_s}(Q_1, \ldots, Q_n)g$ whenever $s < \infty$.

Due to (*), these sequences can always be extended, provided $w_s \neq \varepsilon$. We can assume therefore our sequence to be either finite with $w_s = \varepsilon$ or infinite with $w_k \neq \varepsilon$ for all k.

In the former case, we get $\varphi_{w_s}(Q_1, \ldots, Q_n) = I_\delta$, $(R_{j_1 z_1}, \ldots, R_{j_s z_s}) \in FinTr(\alpha_i)$ and $dC(R_{j_1 z_1}, \ldots, R_{j_s z_s})g$. In consequence, $dTail(\alpha_i)g$, that implies $d = g = \delta$, since $dS_i g$ and $Tail(\alpha_i) \cap S_i = \mathbf{0}$. If we therefore assume $d \neq \delta$, we must get an infinite trace $(R_{j_1 z_1}, R_{j_2 z_2}, \ldots) \in InfTr(\alpha_i)$ with $dR_{j_1 z_1}d_1$ and $d_{k-1}R_{j_k z_k}d_k$ for $1 < k < \infty$. Therefore, $dC(R_{j_1 z_1}, R_{j_2 z_2}, \ldots)\delta$, and, hence, $dPerp(\alpha_i)\delta$, i.e., $d \in Loop(\alpha_i)$. ■

The above theorem has a number of important corollaries. Easy proofs are omitted here.

COROLLARY 6.2.1. *The domain of $(Run(\alpha_1), \ldots, Run(\alpha_n))$ is the domain of the greatest solution of CSE.*

COROLLARY 6.2.2. *The CSE has exactly one solution iff $\bigcup_{i=1}^{n} Loop(\alpha_i) = \emptyset$, i.e., iff the algorithm does not loop starting with any label.*

COROLLARY 6.2.3. *If $(Q_1, \ldots, Q_n)$ is an arbitrary solution of CSE, then $Q_1 = Tail(\alpha_1)$ iff $Q_1 D_\delta \cap Loop(\alpha_1) = \emptyset$.*

This corollary describes just the known fact that proving total correctness requires proving partial correctness and termination (see Section 6.3).

COROLLARY 6.2.4. *If A is a finitistic algorithm and $(Q_1, \ldots, Q_n)$ is an arbitrary solution of the CSE, then $Q_i\{\delta\} - \{\delta\} \subseteq Loop(\alpha_i)$ for $i \leqslant n$.*

COROLLARY 6.2.5. *Let A finitistic and deterministic and let $(Tail(\alpha_1) \cup S_1, \ldots, Tail(\alpha_n) \cup S_n)$ be an arbitrary solution of CSE. If $S_1 = [x \notin Stop(\alpha_1)|\ x := \delta]$, then $S_1 = Perp(\alpha_1)$.*

Both above corollaries can be used in proofs of nontermination of programs.

COROLLARY 6.2.6. *If A is deterministic, then $(Run(\alpha_1), \ldots, Run(\alpha_n))$ is a maximal element in the set of all functional solutions of CSE.*

The term "maximal" should not be confused with "the greatest". In fact, there is in general more than one maximal element in the set of functional solutions of *CSE* (cf. the example preceding Theorem 6.2.5). The corollary says that each extension of the *Run* vector that is a solution of *CSE* is not functional.

6.3. Partial and total correctness of programs

Suppose we are given an admissible PD-algorithm A to be analyzed and suppose $(Q_1, \ldots, Q_n)$ has been shown to be a solution of the corresponding *CSE*. Since the *Tail* vector is of course the least solution of the *CSE*, we obtain therefore

$$Tail(\alpha_1) \subseteq Q_1. \qquad (6.3.1)$$

Suppose now Q_1 to be a function defined "explicitly" by the equation

$$Q_1 = [p(x)|\ x := f(x)].$$

Together with (6.3.1), this implies the following assertion:

$(*)$ for any $d \in D$, if $p(d) = true$ and if, when d is the initial value, the algorithm stops, then the terminal value is $f(d)$.

Observe that $p(d) = true$ does not imply in this case that A stops for d. Proving properties like $(*)$ is usually referred to as proving partial correctness of programs (cf. Hoare [23]). In consequence, proofs of partial correctness in our approach consists merely in showing (checking) that the appropriate $(Q_1, \ldots, Q_n)$ is a solution of *CSE*. A difficult problem that arises in such proofs is to get the appropriate Q_i's. We usually know what the $Run(\alpha_1)$—or at least $Tail(\alpha_1)$—should look like, since this is the intended meaning of the program (algorithm), but at the same time, we know very little about $Run(\alpha_i)$ or $Tail(\alpha_i)$ for $i \geqslant 2$.

Suppose now that we have proved $(Q_1, \ldots, Q_n)$ to be the least solution of *CSE*. This means

$$Tail(\alpha_1) = Q_1 = [p(x)|\ x := f(x)], \qquad (6.3.2)$$

and now we can claim the following:

$(**)$ for any d in D, if $p(d) = true$, then if d is the initial value, the algorithm stops, and the terminal value is $f(d)$.

In this case $p(d) = true$ does imply that A stops for d. Proving assertions like $(**)$ is referred to as proving total correctness of programs (cf. Manna and Pnueli [27]). Observe that (6.3.2) gives even more than total correctness, since it implies the following:

$(\overset{**}{*})$ for any d in D, $p(d) = true$ iff A stops for d, and, if A stops for d, then the terminal value is $f(d)$.

The usual proofs of total correctness, as carried out by Manna and Pnueli [27] or many other authors, do not deal with the behaviour of the program outside of the domain of termination. In fact, if for some d in D our program does not stop, we have two possible cases to be considered:

(1) the program runs infinitely, i.e., $d \in Loop(\alpha_1)$,

(2) the program stops before the control reaches the terminal state ε, i.e., $d \in Block(\alpha_1)$.

Analyzing programs by means of *Run* vectors in $QNR(D)$, we can systematically distinguish between cases (1) and (2).

The case (2) can be split again into subcases, corresponding to control states, where the program can be blocked. An approach that provides such analysis for FC-algorithms is described in Blikle [12].

6.4. Proving properties of programs by δ-relations

We give here three examples of correctness proofs and one example of an equivalence proof. We shall denote in the sequel:

$$\begin{aligned} \text{Re}&\text{—the set of all real numbers,}\\ \text{Nat} &= \{0, 1, \ldots\},\\ \text{Nat}^+ &= \{1, 2, \ldots\},\\ \text{Int} &= \{\ldots, -1, 0, 1, \ldots\}. \end{aligned}$$

EXAMPLE 6.4.1. Consider again the procedure FACTORIAL from Examples 4.1.1 and 4.2.1:

```
procedure FACTORIAL;
  begin
  if $n = 0$ then $s := 1$
     else begin
            $n := n-1$;
            FACTORIAL;
            $n := n+1$;
            $s := s \times n$;
          end;
  end
```

We assume $D = \text{Re} \times \text{Re}$, and we set

Notation	The algorithm
$Q = [n = 0|\ (n, s) := (n, 1)]$,	$(\alpha_1, Q, \varepsilon)$,
$F = [n \neq 0|\ (n, s) := (n-1, s)]$,	$(\alpha_1, F, \alpha_1\alpha_2)$,
$S = [(n, s) := (n+1, s(n+1))]$,	$(\alpha_2, S, \varepsilon)$.

The corresponding *CSE* is the following:

$$\begin{aligned} X_1 &= FX_1X_2 \cup Q,\\ X_2 &= S, \end{aligned}$$

and this is equivalent, by Theorem 3.2.3, to the set

$$X_1 = FX_1 S \cup Q, \qquad X_2 = S. \tag{6.4.1}$$

We state now the following hypothesis

$$Tail(\alpha_1) = [n \in \mathrm{Nat}|\ (n, s) := (n, n!)],$$
$$Perp(\alpha_1) = [n \notin \mathrm{Nat}|\ (n, s) := \delta],$$

and we shall try to prove it. Denote $R = [n \in \mathrm{Nat}|\ (n, s) := (n, n!)]$ and $P = [n \notin \mathrm{Nat}|\ (n, s) := \delta]$. We show first (R, S) to be a solution of (6.4.1):

$$\begin{aligned} FRS \cup Q &= F[n \in \mathrm{Nat}|\ (n, s) := (n, n!)]S \cup Q \\ &= [n \neq 0 \,\&\, n-1 \in \mathrm{Nat}|\ (n, s) := \big(n-1, (n-1)!\big)]S \cup Q \\ &= [n \neq 0 \,\&\, n-1 \in \mathrm{Nat}|\ (n, s) := (n, n!)] \cup Q \\ &= [n \in \mathrm{Nat}|\ (n, s) := (n, n!)] = R. \end{aligned}$$

Hence, (R, S) is a solution of (6.4.1) and, therefore, the partial correctness of the procedure is proved. In fact, we have proved the following: *for any n in* Nat *and s in* Re, *if for (n, s) the algorithm stops, then the output is $(n, n!)$*. We shall show now $(R \cup P, S)$ to be a solution of *CSE*:

$$\begin{aligned} F(R \cup P)S \cup Q = R \cup FPS &= R \cup F[n \notin \mathrm{Nat}|\ (n, s) := \delta]S \\ &= R \cup [n \neq 0 \,\&\, n-1 \in \mathrm{Nat}|\ (n, s) := \delta]S \\ &= R \cup [n \neq 0 \,\&\, n-1 \in \mathrm{Nat}|\ (n, s) := \delta] \\ &= R \cup [n \notin \mathrm{Nat}|\ (n, s) := \delta] = R \cup P. \end{aligned}$$

Corollary 6.2.4 guarantees now that, for any (n, s) with $n \in \mathrm{Nat}$, the algorithm runs infinitely. To show that it loops nowhere else, we shall prove (R, S) to be the least solution of (6.4.1). Suppose this is not the case. Then there exists a set $B \subseteq \mathrm{Nat} \times \mathrm{Re}$ with $B \neq \mathrm{Nat} \times \mathrm{Re}$ such that (R_1, S) is a solution of (6.4.1), where

$$R_1 = [(n, s) \in B|\ (n, s) := (n, n!)].$$

Consequently, we get

$$\begin{aligned} [(n, s) \in B|\ (n, s) := (n, n!)] &= F[(n, s) \in B|\ (n, s) := (n, n!)]S \cup Q \\ &= [n = 0 \vee \big(n \neq 0 \,\&\, (n-1, s) \in B\big)|\ (n, s) := (n, n!)], \end{aligned}$$

and hence,

$$B = \{0\} \times \mathrm{Re} \cup \{(n, s)|\ n \neq 0 \,\&\, (n-1, s) \in B\}.$$

This implies

$$(\forall s \in \mathrm{Re})[(0, s) \in B],$$
$$(\forall (n, s) \in B)[(n+1, s) \in B],$$

hence, $B = \mathrm{Nat} \times \mathrm{Re}$, contradicting the assumption. Finally, (R, S) is the least solution of (6.4.1); thus, by Theorem 4.3.1, $Tail(\alpha_1) = R$, and by Corollary 6.25, $Perp(\alpha_1) = P$.

In this way the full proof of correctness and termination of the procedure FACTORIAL is accomplished.

EXAMPLE 6.4.2. Procedure NINETYONE.

```
procedure NINETYONE;
  begin
  if $x > 100$ then $x := x-10$
     else begin
              $x := x+11$;
              NINETYONE;
              NINETYONE;
           end;
  end
```

We assume $D = \mathrm{Int}$ and we set:

Notation	The algorithm
$Q = [x > 100\|\ x := x-10]$,	$(\alpha_1, Q, \varepsilon)$,
$F = [x \leqslant 100\|\ x := x+11]$,	$(\alpha_1, F, \alpha_1\alpha_1)$,

The *CSE* is:

$$X = FXX \cup Q. \tag{6.4.2}$$

Our hypothesis is:

$$Tail(\alpha_1) = [x \leqslant 100|\ x := 91] \cup [x > 100|\ x := x-10],$$
$$Perp(\alpha_1) = \mathbf{0}.$$

Denote $R = [x \leqslant 100|\ x := 91] \cup [x > 100|\ x := x-10]$. We shall show R to be a solution of (6.4.2). Compute:

$$\begin{aligned} FRR \cup Q &= ([x \leqslant 89|\ x := 91] \cup [89 < x \leqslant 100|\ x := x+1])R \cup Q \\ &= [x \leqslant 89|\ x := 91] \cup \mathbf{0} \cup [89 < x \leqslant 99|\ x := 91] \cup \\ &\qquad \cup [99 < x \leqslant 100|\ x := x-9] \cup [x > 100|\ x := x-10] \\ &= [x \leqslant 100|\ x := 91] \cup [x > 100|\ x := x-10] = R. \end{aligned}$$

This terminates the proof of partial correctness. We have to show now that R is the least solution of (6.4.2). Suppose this is not the case. Let $A \subseteq (-\infty, 100)$, $A \neq (-\infty, 100)$, $B \subseteq (100, +\infty)$, $B \neq (100, +\infty)$ and let

$$R_1 = [x \in A|\ x := 91] \cup [x \in B|\ x := x-10]$$

be a solution of (6.4.2). We get therefore (we replace here & by ,)

$$\begin{aligned} FR_1R_1 \cup Q &= ([x \leqslant 100, x+11 \in A|\ x := 91] \cup \\ &\qquad \cup [x \leqslant 100, x+11 \in B|\ x := x+1])R_1 \cup Q \\ &= [x \leqslant 100, x+11 \in A, 91 \in A|\ x := 91] \cup \\ &\qquad \cup \mathbf{0} \cup \\ &\qquad \cup [x \leqslant 100, x+11 \in B, x+1 \in A|\ x := 91] \cup \\ &\qquad \cup [x \leqslant 100, x+11 \in B, x+1 \in B|\ x := x-9] \cup \\ &\qquad \cup [x > 100|\ x := x-10] \\ &= [x \in A|\ x := 91] \cup [x \in B|\ x := x-10]. \end{aligned}$$

The last equality, of course, follows from the assumption that R_1 is a solution of *CSE*. We get from it

$$B = (100, +\infty), \tag{6.4.3}$$

and therefore,

$$[x \leqslant 100, x+11 \in B, x+1 \in B|\ x := x-9] = [x = 100|\ x := 91].$$

Now we can state the following equation that must be satisfied by the set A:

$$\begin{aligned} A = \{x|\ x \leqslant 100, x+11 \in A, 91 \in A\} &\cup \\ \cup \{x|\ 89 < x \leqslant 100, x+1 \in A\} &\cup \\ \cup \{100\} & \end{aligned} \tag{6.4.4}$$

Therefore

1) $100 \in A$,

2) if $90 < y \leqslant 101\ \&\ y \in A$, then $y-1 \in A$.

In effect,

$$(89, 100\rangle \subseteq A, \tag{6.4.5}$$

hence $91 \in A$. From (6.4.4), we now get the following:

$$\text{if } y \leqslant 111 \text{ and } y \in A \text{ then } y-11 \in A.$$

This, combined with (6.4.5), implies

$$(-\infty, 100\rangle \subseteq A$$

which, together with (6.4.3), contradicts our assumption. We have shown in this way that R is the least solution of *CSE* (6.4.2), hence, by Theorem 4.3.1, $R = \mathit{Tail}(\alpha_1)$. In consequence, $\mathit{Stop}(\alpha_1) = \mathrm{Int}$, hence, by Lemma 6.2.4, $\mathit{Loop}(\alpha_1) = \emptyset$, i.e., $\mathit{Perp}(\alpha_1) = \mathbf{0}$. In this way our hypothesis is proved. ■

EXAMPLE 6.4.3. Consider the following nonrecursive procedure (Dijkstra [19]).

```
procedure GREATEST COMMON DIVISOR
  begin
     while $n \neq m$ do if $n > m$ then $n := n-m$ else $m := m-n$
  end
```

Suppose we want to analyse this algorithm only in the case n and m range over positive integers, i.e., $D = \mathrm{Nat}^+ \times \mathrm{Nat}^+$. We set:

Notation	The algorithm
	$(\alpha_1, [n = m], \varepsilon)$,
	$(\alpha_1, [n \neq m], \alpha_2)$,
$F = [n > m\|\ n := n-m]$,	(α_2, F, α_2),
$G = [n \leqslant m\|\ m := m-n]$,	(α_2, G, α_1).

We write $n := n-m$ to abbreviate $(n, m) := (n-m, m)$ and for $m := m-n$, analogously. The *CSE*:

$$\begin{aligned} X_1 &= [n \neq m]X_2 \cup [n = m], \\ X_2 &= (F \cup G)X_1. \end{aligned} \tag{6.4.6}$$

Our hypothesis:

$$\begin{aligned} \mathit{Tail}(\alpha_1) &= [(n, m) := (g(n, m), g(n, m))]; & \mathit{Perp}(\alpha_1) &= \mathbf{0}, \\ \mathit{Tail}(\alpha_2) &= (F \cup G)\,\mathit{Tail}(\alpha_1); & \mathit{Perp}(\alpha_2) &= \mathbf{0}, \end{aligned}$$

where $g(n, m)$ is the greatest common divisor of n and m.

Denote $R = [(n, m) := (g(n, m), g(n, m))]$. We shall prove $(R, (F \cup G)R)$ to be a solution of *CSE* (6.4.6). The following properties of $g(n, m)$ will be used:

if $n = m$ then $g(n, m) = n$,
if $n > m$ then $g(n-m, m) = g(n, m)$,
if $n < m$ then $g(n, m-n) = g(n, m)$.

Compute:

$$\begin{aligned} [n \neq m](F \cup G)R \cup [n = m] &= [n > m|\ (n, m) := (g(n-m, m), g(n-m, m))] \cup \\ &\quad \cup [n < m|\ (n, m) := (g(n, m-n), g(n, m-n))] \cup \\ &\quad \cup [n = m|\ (n, m) := (n, m)] \\ &= [n > m|\ (n, m) := (g(n, m), g(n, m))] \cup \\ &\quad \cup [n < m|\ (n, m) := (g(n, m), g(n, m))] \cup \\ &\quad \cup [n = m|\ (n, m) := (g(n, m), g(n, m))] \\ &= R. \end{aligned}$$

We shall show now $\mathit{Perp}(\alpha_1) = \mathbf{0}$. Since this implies $\mathit{Loop}(\alpha_1) = \emptyset$, we shall have our hypothesis by Corollary 6.2.3. Solving *CSE* (6.4.6) in a modified way to get the *Run* vector, we get

$$\mathit{Run}(\alpha_1) = ([n \neq m](F \cup G))^*[n = m] \cup ([n \neq m](F \cup G))^\infty.$$

Since our algorithm is finitistic, we get, by Lemma 6.2.3,

$$\begin{aligned} \mathit{Perp}(\alpha_1) &= ([n \neq m](F \cup G))^\infty \\ &= ([n > m|\ n := n-m] \cup [n < m|\ m := m-n])^\infty = \mathbf{0}. \end{aligned}$$

This is of course true by the assumption that $D = \mathrm{Nat}^+ \times \mathrm{Nat}^+$.

References

[1] de Bakker, J. W., *Recursive procedures*, Mathematical Center Tracts 24, Mathematisch Centrum Amsterdam 1971.

[2] —, and L. G. L. Th. Meertens, *Simple recursive program schemes and induction assertion*, ibid. 142 (1972).

[3] —, and W. P. de Roever, *A calculus for recursive program schemes*, Automata Languages and Programming (Proc. Symp. IRIA Rocquencourt, 1972), North-Holland, Amsterdam 1973, pp. 167–196.

[4] —, and D. Scott, *A theory of programs in outline of join work by J. W. de Bakker and Dana Scott* (manuscript), Vienna 1969.

[5] Bekić, H., *Definable operations in general algebras and the theory of automata and flowcharts* (manuscript), IBM Laboratory, Vienna 1969.

[6] Blikle, A., *Algorithmically definable functions*; *A contribution towards the semantics of programming languages*, Diss. Math. 85 (1971).

[7] —, *Languages definable by equations*, Bull. Acad. Polon. Sci., Sér. Sci. Math. Astronom. Phys. 19 (1971), pp. 859–863.

[8] —, *Nets; complete lattices with a composition*, ibid. 19 (1971), pp. 1123–1127.

[9] Blikle, A., *Equational languages*, Inform. Contr. 21 (1972), pp. 134–147.

[10] —, *Iterative systems; an algebraic approach*, Bull. Acad. Polon. Sci., Sér. Sci. Math. Astronom. Phys. 20 (1972), pp. 51–55.

[11] —, *Complex iterative systems*, ibid. 20 (1972), pp. 57–61.

[12] —, *An algebraic approach to mathematical theory of programs*, CC PAS Reports 119 (1973).

[13] —, *An algebraic approach to programs and their computations*, Mathematical Foundations of Computer Science, II (Proc. Symp. High Tatras 1973), High Tatras 1973, pp. 3–8.

[14] —, *An extended approach to mathematical analysis of programs*, CC PAS Reports 169 (1974).

[15] —, *Proving programs by sets of computations*, Mathematical Foundations of Computer Science, III (Proc. Symp. Warsaw–Jadwisin 1974), Lecture Notes in Computer Science, Vol. 28, Springer-Verlag, Heidelberg 1975, pp. 313–358.

[16] —, *Proving programs by δ-relations*, Formalization of Semantics of Programming Languages and Writing of Compilers, (Proc. Symp. Frankfurt am Oder 1974), Elektronische Informationesverarbeitung und Kybernetik, 11 (1975), pp. 267–274.

[17] —, and A. Mazurkiewicz, *An algebraic approach to the theory of programs, algorithms, languages and recursiveness*, Mathematical Foundations of Computer Science, I (Proc. Symp. Warsaw–Jablonna 1972), Warsaw 1972.

[18] Cadiou, J. M., *Recursive definitions of partial functions and their computations*, Report STAN–266–72, Stanford University 1972.

[19] Dijkstra, W., *A short introduction to the art of programming*, EWD 316, 1971.

[20] Engeler, E., *Formal languages: Automata and structures*, Lectures in Advanced Mathematics, Markham Publishing Company, Chicago 1968.

[21] Engelfriet, J., *Translation of simple program schemes*, Automata, Languages and Programming (Proc. Symp. IRIA Rocquencourt 1972), North-Holland, Amsterdam 1973, pp. 215–224.

[22] Hitchcock, P., and D. Park, *Induction rules and termination proofs*, ibid. pp. 225–251.

[23] Hoare, C. A. R., *An axiomatic basis of computer programming*, Communication of ACM 12 (1969), pp. 576–580, 583.

[24] Kaplan, D. M., *Regular expressions and the equivalence of programs*, Journ. Comp. Syst. Sci. 3 (1969), pp. 361–386.

[25] Kleene, S. C., *Introduction to metamathematics*, New York 1952.

[26] Leszczyłowski, J., *A theorem on resolving equations in the space of languages*, Bull. Acad. Polon. Sci., Sér. Sci. Math. Astronom. Phys. 19 (1971), pp. 967–970.

[27] Manna, Z., and A. Pnueli, *Axiomatic approach to total correctness of programs*, Memo AIM–210, STAN–CS–73–382, Stanford Artificial Intelligence Laboratory 1973.

[28] —, and J. Vuillemin, *Fixpoint approach to the theory of computation*, Communication of ACM 15 (1972), pp. 528–536.

[29] Mazurkiewicz, A., *Proving algorithms by tail functions*, Inform. Contr. 18 (1971), pp. 220–226.

[30] —, *Iteratively computable relations*, Bull. Acad. Polon. Sci., Sér. Sci. Math. Astronom. Phys. 20 (1972), pp. 793–798.

[31] —, *Recursive algorithms and formal languages*, ibid. pp. 799–803.

[32] —, *Proving properties of processes*, CC PAS Reports 134 (1973), also in Mathematical Foundations of Computer Science, II (Supplement to Proc. Symp. Hight Tatras 1973), to appear.

[33] Nivat, M., *Langages algébriques sur le magma libre et sémantique des schémas de programmes*, Automata, Languages and programming (Proc. Symp. IRIA Rocquencourt 1972), North-Holland, Amsterdam 1973, pp. 293–308.

[34] Park, D., *Fixedpoint induction and proof of program semantics*, Machine Intelligence 5 (1969), pp. 59–78.

[35] Rasiowa, H., *On ω^+-valued algorithmic logic and related problems*, Mathematical Foundations of Computer Science II, (Supplement to Proc. Symp. Hight Tatras 1973), to appear.

[36] Redziejowski, R. R., *The theory of general events and its application to parallel programming*, TP 18.220, IBM Nordic Laboratory Sweden 1972.

[37] Salwicki, A., *Formalized algorithmic languages*, Bull. Acad. Polon. Sci., Sér. Sci. Math. Astronom. Phys. 18 (1970), pp. 227–232.

[38] —, *On the equivalence of FS-expressions and programs*, ibid. 18 (1970), pp. 275–278.

[39] —, *On the predicate calculi with iteration quantifier*, ibid. 18 (1970), pp. 179–286.

[40] Scott, D., *The lattice of flow diagrams*, PRG–3, Technical Monograph, also IFIP WG 2.2 Bulletin 5, 1970.

[41] —, and Ch. Strachey, *Toward a mathematical semantics for computer languages*, PRG–4, Technical Monograph, Oxford University 1970.

[42] Tarski, A., *A lattice-theoretic fixpoint theorem and its applications*, Pacific Journal of Mathematics 5 (1955), pp. 285–309.