AN ANALYSIS OF PROGRAMS BY ALGEBRAIC MEANS

A. Blikle

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

CONTENTS

1. Introduction (167).
2. Nets and quasiets (170); 2.1. Abstract quasiets (170); 2.2. Sequences—general notation (172); 2.3. Nets of generalized languages (173); 2.4. Quasiets of bundles of computations (174); 2.5. Quasiets of $\delta$-relations (176).
3. Equations in quasiets (178); 3.1. Existence of solutions (178); 3.2. Elimination of variables (179).
4. Mazurkiewicz algorithms in abstract quasiets (181); 4.1. The definition (181); 4.2. Operational semantics (182); 4.3. Fixedpoint semantics (186); 4.4. Proofs of the fixedpoint theorems (188); 4.5. Plate-control algorithms (191).
5. Algorithms over bundles of computations (195); 5.1. Extension of the calculus of bundles (195); 5.2. Fixedpoint theorems in $QNR(0)$ (196); 5.3. Proving properties of programs by bundles of computations (197).
6. Algorithms over $\delta$-relations (201); 6.1. An extension of the calculus of relations (201); 6.2. $QNR(D)$-oriented theory of algorithms (202); 6.3. Partial and total correctness of programs (206); 6.4. Proving properties of programs by $\delta$-relations (207).

References (211).

INTRODUCTION

The rapid development of software techniques and technologies in the past decade raises the urgent necessity of creating a theoretical background adequate to fulfill 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

[167]
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

\[
S : L \to R,
\]

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 \( \delta \)-relations—for input-output analysis covering description of infinitistic behaviour (lopping),
3. sets of sequences 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 “topology” 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 lattice-like algebras called quasinet. 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 \( a_1, \ldots, a_n \) of Fig. 1.1.1 are partial functions corresponding to elementary instruc-
ations, then \( g \) is the output function of the program, if \( a_1, \ldots, a_n \) are names of instructions (one-character languages) and control is described by a flowchart, then \( g \) is the corresponding regular language containing all possibleEngeler’s signatures of the program (see Engeler [20]), if \( a_1, \ldots, a_n \) are sets of microcomputations, then \( g \) is the set of global computations of the program, etc.

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 \( a_1, \ldots, a_n \)), 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 quasineut. 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
2. NETS AND QUASINETS

2.1. Abstract quasinesTs

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

- \(\bigcup P\)---the l.u.b. of \(P\),
- \(\bigcap P\)---the g.l.b. of \(P\),
- \(a \sqcup b = \bigcup \{a, b\}\)---the join of \(a\) and \(b\),
- \(a \sqcap b = \bigcap \{a, b\}\)---the meet of \(a\) 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 \(\varphi: U \to U\) is said to be additive if it preserves finite joins, i.e., for any elements \(a\) and \(b\) in \(U\)

\[\varphi(a \sqcup b) = \varphi(a) \sqcup \varphi(b),\]

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(\bigcup P) = \bigcup \{\varphi(p)\} \quad p \in P\}

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

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

1. \((U, \leq)\) is a complete lattice,
2. \((U, *, e)\) is a monoid with unit \(e\),
3. \(*\) is additive and continuous,
4. for any \(a\) in \(U\),

\[0 * a = 0,\]

where \(0 = \bigcap U\) is the bottom element of the lattice.

By a net we mean any quasinet \(N = (U, \leq, *, e)\) with the additional property:

5. for any \(a\) in \(U\),

\[a * 0 = 0,\]

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

Lemma 2.1.1. If \(\varphi: U \to U\) is continuous and additive in \((U, \leq)\), then for any finite (but nonempty) or denumerable set \(P \subseteq U\)

\[\varphi(\bigcup P) = \bigcup \{\varphi(p)\} \quad 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 \(U\):

\[a^n = e, \quad a^{n+1} = a*a \quad \text{for} \quad n = 0, 1, ...,\]

\[a^\infty = \bigcup_{n=0}^{\infty} a^n.\]

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 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 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 \(a = 0\), then the compound algorithm generating \(ab\) has no runs at all, and therefore \(ab = 0\). If, however, \(b = 0\), 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 \(ab = 0\). On the other hand, we may be willing to develop the
analysis of algorithms neglecting infinitistic behaviour. In such a case we impose condition (S) and we deal with nets. For example, the usual binary relations constitute nets (looping is assumed to have no effect in this case) and A-relations (Section 2.5) constitute quasinefits 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 $\text{seq}(U)$ denote the set of all finite and infinite sequences of elements of $U$, the empty sequence $e$ included. By a generalized composition in $N$ we mean any operation $C: \text{seq}(U) \rightarrow U$ with the following properties:

(1) $C[e] = e$

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

(3) $C[t_1 \cdot t_2] = C[t_1]C[t_2]$ for any finite $t_1$ and finite or infinite $t_2$, where $t_1 \cdot t_2$ 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 quasinefits 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 $N = (U, \preceq, *, e, C)$.

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

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

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

(2.1.3) $a^m = a^n$

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

2.2. Sequences—general notation

The particular quasinefits, 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 $(a_1, ..., a_n) \preceq \infty$ and $(a_1, a_2, ...)$. We shall also write $(a_1, ..., a_n) \preceq \infty$ to denote a sequence that may be finite or infinite. The empty sequence is denoted by $e$ and is assumed to be of length 0. We denote by:

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

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

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

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

$\text{Fin}(A) = A \cap \text{seq}^*(D)$,

$\text{Inf}(A) = A \cap \text{seq}^\infty(D)$.

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

2.3. Nets of generalized languages

The set $D$ (Section 2.2.2) will now be called alphabet and sequences over $D$ will be called words. In the set $\text{seq}(D)$ we define a total operation $\cdot^\infty$ of concatenation:

(1) $(a_1, ..., a_n) \cdot^\infty (b_1, ..., b_m) = (a_1, ..., a_n, b_1, ..., b_m)$ for $0 \leq n < \infty$ and $0 \leq m \leq \infty$.

(2) $(a_1, a_2, ...) \cdot^\infty (b_1, ..., b_m) = (a_1, a_2, ...) \preceq \infty$ for $m \leq \infty$.

As is easy to see, $(\text{seq}^*(D) \cdot^\infty, e)$ and $(\text{seq}^\infty(D) \cdot^\infty, e)$ are both monoids.

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

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

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

$L_1 \cdot^\infty L_2 = \{x_1 \cdot^\infty x_2 \mid x_1 \in L_1 \text{ and } x_2 \in L_2\}$.

It is easy to show now that $\text{NGL}(D) = \{GLan(D) \subseteq \cdot^\infty, \{e\}\}$ is a set-theoretic net with $\emptyset = \emptyset$—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 subset of $\text{NGL}(D)$.

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

$(\forall n \geq 1)(x_1 \cdot^\infty x_2 \cdot^\infty ... \cdot^\infty x_n \preceq C[x_1, x_2, ...])$.

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

(1) $C[e] = \{e\}$

$C[L_1, ..., L_n] = C[L_1, ..., L_{n-1}] \cdot^\infty L_n$ for $1 \leq n \leq \infty$.

(2) $C[L_1, L_2, ...] = \{C[x_1, x_2, ...] \mid (\forall n \geq 1)(x_n \in L_n)\}$.

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

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

2.4. Quasinet 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) \((a_1, \ldots, a_n) \circ (b_1, \ldots, b_m) = \text{if } a_n = b_1 \text{ then } (a_1, \ldots, a_{n-1}, b_2, \ldots, b_m) \text{ else } e,\)

\(\) for \(0 \leq n < \infty\) and \(0 \leq m < \infty;\)

(2) \((a_1, a_2, \ldots) \circ (b_1, \ldots, b_m) = (a_1, a_2, \ldots) \text{ for } 0 \leq m < \infty.

In particular, we have

\(e \circ x = e \text{ for } x \in \text{seq}(D),\)

\(x \circ e = e \text{ for } x \in \text{seq}(D),\)

\((a, b) \circ (b, c) = (a, b, c),\)

\(\ldots \ldots \ldots \ldots \ldots \text{ etc.}\)

Interpretation. In the majority of cases, elements in D are interpreted as data-vectors 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 \((a, \ldots, a)\) is a finite computation of \(A_1\) and its last (output) element \(a_n\) is equal to the first

![Diagram](image_url)

Fig. 2.4.1

(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)).

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), it is not a monoidal operation.

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

\(\text{Bun}(D) = \{ B \mid B \subseteq \text{seq}(D) \& e \in B \}.\)

The assumption that e is an element of any bundle is of technical character and will be explained below. In the set \(\text{Bun}(D)\) we define an operation of composition:

For any \(B_1, B_2\) in \(\text{Bun}(D)\):

\(B_1 \circ B_2 = \{ x \in B_1 | y \in B_2 \}.\)

Let \(E = \text{seq}(D) \cup \{ e \} \). It is easy to prove that \(\text{QNB}(D) = (\text{Bun}(D), \leq, \circ, E)\) is a set-theoretic quasinet (but not a net), with \(\emptyset = \{ e \}\). The bundle \(\emptyset\) is called the empty bundle and for any \(B\) in \(\text{Bun}(D)\), we have

\[0 \circ B = B,\]

\[B \circ 0 = \text{Inf}(B) \cup \emptyset.\]

Equation (2.4.1) asserts that \(\text{QNB}(D)\) is not a net. This equation is a formal consequence of the assumption that \(e\) 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 finitistic provided all its computations are infinite. Finitistic bundles constitute a set which is a subquasinet of \(\text{QNB}(D)\).

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

\[C[x_1, x_2, \ldots] = \text{the shortest computation in seq(D) with the following properties:}\]

(i) If \(e \in \exists(n)\{x_1, \ldots, x_n = e\}\), then \(C[x_1, x_2, \ldots] = e,\)

(ii) If \(e \in \exists(n)\{x_1, \ldots, x_n \neq e\}\), then

\[e \text{ is provided } b \neq e,\]

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

\[C[(a, b), (b, c), (c, \ldots)] = e,\]

\[C[(a, b), (b, c), (c, \ldots)] = (a, b, c),\]

\[C[(a, b), (b, c), (c, \ldots)] = (a, b, c).\]

Now, to define the generalized composition in \(\text{Bun}(D)\), we set

(1) \(C[e] = E,\)

\(C[B_1, \ldots, B_n] = C[B_1, \ldots, B_n] \circ B_n\) for \(1 \leq n < \infty,\)

(2) \(C[B_1, B_2, \ldots] = C[C[x_1, x_2, \ldots]| (\forall i \geq 1)(x_i \in B_i)],\)

As we have seen, the difference between \(\text{NGL}(D)\) and \(\text{QNB}(D)\) arises from the difference between concatenation and composition. Algebraically this leads to the fact that there are zero divisors in \(\text{QNB}(D)\) — i.e., there exist \(B_1 \neq \emptyset\) and \(B_2 \neq \emptyset\).
with \( B_1 \cdot B_2 = \emptyset \)—while in \( \text{NGL}(D) \) 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 \( \text{NGL}(D) \) 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
\[
A_1 = \{"x < 0"\}, \quad A_2 = \{"x \geq 0"\}, \\
G = \{"x := x + 1"\}, \quad H = \{"x := 4x"\}.
\]

What we can prove now about the algorithm is only that its finitistic behaviour corresponds to the language
\[
L_{\text{fin}} = (A_1; G)^* A_2 H,
\]
and its entire behaviour to the language
\[
L_{\text{ent}} = (A_1; G)^* A_2 H C (A_1; G)^*.
\]

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:
\[
A_1 = \{(d) \mid d < 0\} \cup \{e\}, \quad A_2 = \{(d) \mid d \geq 0\} \cup \{e\}, \\
G = \{(d, d+1) \mid d \in D\} \cup \{e\}, \quad H = \{(d, 4d) \mid d \in D\} \cup \{e\}.
\]

The finitistic and entire behaviour of the algorithm is described now by the following bundles:
\[
B_{\text{fin}} = (A_1; G)^* A_2 H, \quad B_{\text{ent}} = (A_1; G)^* A_2 H C (A_1; G)^*.
\]

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) \( (A_1; G)^* = 0 \), i.e., there are no infinite runs in the algorithm,
2) \( B_{\text{fin}} = B_{\text{ent}} \),
3) \( B_{\text{fin}} = \{(d, d+1, d+2, \ldots , -1, 0, 0) \mid d < 0\} \cup \{(d, 4d) \mid d \geq 0\} \).

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

**2.5 Quasimetrics of \( \delta \)-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 \((a_i, a_j)\) of states with the property that there exists a finite computation \((b_1, \ldots , b_n)\) with \( a_i = b_1 \) and \( a_j = b_n \). 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 (loopping). 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 finitistic behaviour at the same time, we add to our relations so-called input-infinity pairs. What we get in this way are the \( \delta \)-relations described below.

Let \( D \) be an arbitrary (finite or infinite) set called, as in the preceding case, the set of states. Let \( \delta \) be an arbitrary object not in \( D \), and let
\[
D_\delta = D \cup \{\delta\}, \quad \delta \in D_\delta,
\]
then set \( (D \cup \{\delta\}) \subseteq \delta \in D \cup \{\delta\} \).

Elements of \( \text{Rel}(D) \) are called \( \delta \)-relations over \( D \). Given \( (a, b) \in \text{Rel}(D) \) we shall write \( a \delta b \) instead of \( (a, b) \in \text{Rel}(D) \).

The \( \delta \)-relationships are described as follows: If \( (a, b) \in \text{Rel}(D) \) is an \( \delta \)-relationship, then for any \( a \) and \( b \) in \( D \) we read:
\[
a \delta b \text{ as there exists a finite run in } A \text{ that starts with the state-vector } (\text{begin}, a) \text{ and terminates with } (\text{end}, b),
\]
\[
a \delta b \text{ as there exists an infinite run in } A \text{ that starts with } (\text{begin}, a).
\]

The assumption that \( a \delta b \) for any \( \delta \)-relationship \( \delta \) has a technical character and plays the same part here as the assumption that \( e \) is in any bundle.

Some authors (e.g., Cadiou [18]) dealing with binary partial relations, introduce a so-called "undefined element" \( 0 \) and set \( d \delta R \) 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 \( \delta \)-relations which are partial, i.e., for any \( R \) in \( \text{Rel}(D) \) its domain \( \{d \mid (3d \in D) \delta R d\} \) may be a proper subset of \( D \). If \( d \delta d \) is not in the domain of \( d \), then it should be interpreted as "\( \delta \) involves blocking"—any run that starts with \( (\text{begin}, d) \) terminates with some \( (c, d) \) where \( c \) is a control state different from \( \text{end} \).

**Example 2.5.1.** Consider the following program:
```markdown
begin
    while \( x < 0 \) do
        \( x := x - 1; \)
    \( x := x^{-1}; \)
end.
```
The \( \delta \)-relation in the set of all reals which corresponds to this program is the following: \( R = \{(d, d^{-1}) \mid d > 0\} \cup \{(d, 0) \mid d < 0\} \cup \{(0, 0)\} \). 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 \( Rel(D) \) in the usual way: for any \( R_1 \) and \( R_2 \)
\[
aR_1 \circ R_2 b \equiv (\exists c) aR_1 c \land cR_2 b.
\]

Let \( I \equiv \{(d, d) \mid d \in D\} \). It is easy to prove that \( QNR(D) = \{Rel(D), \subseteq, \circ, I\} \) is a set-theoretic quasinet with \( \theta = \{(0, 0)\} \).

A \( \delta \)-relation \( R \) is said to be finitistic provided \( R \cap D = \emptyset \) and is said to be inherently infinitistic provided \( R \subseteq D \). Finitistic \( \delta \)-relations constitute a net, while \( QNR(D) \) is not a net. In fact, for any \( R \) in \( Rel(D) \), we have
\[
\theta \ast R = 0,
\]
\[
R \ast \theta = R \cap D.
\]

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 \( \delta \)-relations. Let \( R_1, R_2, \ldots \) be an arbitrary infinite sequence of \( \delta \)-relations.

1. \( C[1] = I \)
2. \( C[R_1, \ldots, R_n, R_{n+1}] = R_n \circ (\cdots \circ (R_1 \circ R_2) \cdots) \circ R_{n+1} \) for \( 1 \leq n < \infty \),
3. \( C[R_1, R_2, \ldots] = \{(a, b) \mid \exists a_1, a_2, \ldots \in seq^n(D)[a_1 = a \land \forall i \geq 1] (\exists R_i a_1 a_{i+1})\} \).

Quasinets of \( \delta \)-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 \( N = (U, \leq, \ast, e) \) 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:

\[
\begin{align*}
x_1 &= \varphi_1(x_1, \ldots, x_n), \\
&\quad \cdots \\
x_i &= \varphi_i(x_1, \ldots, x_n),
\end{align*}
\]

(3.1.1)

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

\[
x = \Phi(x),
\]

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

Each fixpoint of \( \Phi \) is said to be a solution of (3.1.1). If \( a \) is a fixpoint of \( \Phi \) and for any other fixpoint \( b \) of \( \Phi \) we have \( a \leq b \), resp. \( a' \neq b \), in the product quasinet, then \( a \) is said to be the least solution (least fixpoint), resp. the greatest solution (greatest fixpoint), of (3.1.1) of \( \Phi \).

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

A function \( \Phi : U^n \to U^n \) is said to be monotone if for any \( a \) and \( b \) in \( U^n \)
\[
a \leq b \quad \text{implies} \quad \Phi(a) \leq \Phi(b).
\]

Clearly, if
\[
\Phi(x) = (\varphi_1(x), \ldots, \varphi_n(x)) \quad \text{with} \quad \varphi_i : U^n \to U \quad \text{for} \quad i \leq n,
\]
then \( \Phi \) is monotone (continuous) if all \( \varphi_i \) are monotone (continuous). Moreover, each continuous function is monotone but not conversely.

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

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

\[
\bigcup_{i=1}^{\infty} \Phi^i(\theta),
\]

where \( \Phi^i(x) = (\cdots (\Phi(x) \cdots) \cdots) \) \( i \) -times and \( \theta = (0, \ldots, 0) \in U^n \).

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 \( x_i \) by solving \( x_i = \varphi_i(x_1, \ldots, x_n) \) with respect to \( x_i \) and then substitute this solution for all appearances of \( x_i \) 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 \( \Phi(x, y) \) be an arbitrary function of \( n + m \) arguments, where \( x = (x_1, \ldots, x_n) \) ranges over \( U^n \), \( y = (y_1, \ldots, y_m) \) ranges over \( U^m \) and \( \Phi \) ranges over \( U^n \), i.e., \( \Phi : U^{n+m} \to U^n \). Of course, \( (x, y) \) is used to abbreviate \( (x_1, \ldots, x_n, y_1, \ldots, y_m) \).

If there exists a function \( \Gamma : U^n \to U^n \) such that for any \( a \in U^n \), \( \Gamma(a) \) is the least fixpoint of \( \Phi(x, a) \), then \( \Gamma \) will be called the resolvent of \( \Phi \) with respect to \( x \) and will be written

\[
\Gamma(y) = (\mu x) \Phi(x, y).
\]

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

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

**Theorem 3.2.1.** For every continuous function \( \Phi(x, y) \), where \( x \) and \( \Phi \) range over the same \( U^\ast \), 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 \( l \)-equivalent provided their least solutions exist and are equal.

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

\[
\begin{align*}
x &= \Phi(x, y), \\
y &= \Psi(x, y),
\end{align*}
\]

(3.2.1)

where \( \Phi: U^\ast \times U^\ast \to U^\ast \), \( \Psi: U^\ast \times U^\ast \to U^\ast \), \( x \) ranges over \( U^\ast \) and \( y \) over \( U^\ast \). If \( \Phi \) and \( \Psi \) are continuous and \( \Gamma(y) = (\mu x)\Phi(x, y) \), then the above set is \( l \)-equivalent to the following:

\[
\begin{align*}
x &= \Gamma(y), \\
y &= \Psi(\Gamma(y), y).
\end{align*}
\]

**Theorem 3.2.3.** Let \( \Phi \) and \( \Psi \) be defined as above. The set (3.2.1) is equivalent to the following:

\[
\begin{align*}
x &= \Phi(x, \Psi(x, y)), \\
y &= \Psi(x, y).
\end{align*}
\]

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. \((\mu x)(axb) = a^\ast b\ast\)
2. \((\mu x)(xa^\ast b) = ba^\ast\)
3. \((\mu x)(axb^\ast x) = \cup_{i=0}^n d^\ast b^\ast\)

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

\[
\begin{align*}
x &= ax \cup by, \\
y &= x \cup ay \cup c.
\end{align*}
\]

It is \( l \)-equivalent to

\[
\begin{align*}
x &= a^\ast by, \\
y &= (a^\ast b \cup d)y \cup c,
\end{align*}
\]

and the latter to

\[
\begin{align*}
x &= a^\ast b(a^\ast b \cup d)^\ast c, \\
y &= (a^\ast b \cup d)^\ast c,
\end{align*}
\]

hence, this is the least solution of the set. \( \blacksquare \)

### 4. Mazurkiewicz Algorithms in Abstract Quasines

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 quasines.

#### 4.1. The definition

Let \( N = (U, \leq, \ast, \varepsilon, C) \) be an arbitrary quasinet with generalized composition to be fixed for the rest of Section 4.

By a pushdown algorithm (abr. PD-algorithm) over \( N \) we mean any system

\[
A = (V, s_0, \varepsilon, \beta, P),
\]

where \( V = \{ a_1, \ldots, a_k \} \) is an arbitrary finite set of characters called labels; \( s_0 \in V \) and \( a_0 \) is called the initial label; \( \varepsilon \) is the empty string (of labels); \( P \) is a finite set of triples of the form \((s_0, a, z)\), where \( a \in U, s_0 \in V, x \in V^\ast \), such that for any \( a \in V \) there exists \((s_0, a, z)\) in \( P \) with \( s_0 = s_0 \).

Each triple \((s_0, a, z)\) in \( P \) is called an instruction and can be interpreted as a procedure declaration, where \( s_0 \) 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:

```plaintext
procedure FACTORIAL; begin
if n = 0 then z := 1
else begin
    n := n - 1;
    FACTORIAL;
    n := n + 1;
    z := y \times n;
end;
end;
```

The corresponding PD-algorithm is the following:

\[
V = \{ a_1, a_2 \},
\]
\( P \) consists of the three instructions,

\[
\begin{align*}
(\sigma_1, \text{if } n = 0 \text{ then } s := 1),
(\sigma_2, \text{if } n \neq 0 \text{ then } n := n - 1),
(\sigma_3, [r := n + 1; s := s \times n], s).
\end{align*}
\]

The actions of these instructions may now be interpreted, depending on what we want to prove about the procedure, either as character languages (cf. example of a program in Section 2.4), or as \( \delta \)-relations in the set \( \text{Re} \times \text{Re} \) (Re—the set of reals) or as appropriate bundles of computations, e.g., \( [[n \neq 0 \text{ then } n' = n - 1] = \{(n, n), [n-1, n] \} \} \). \( n \neq 0 \& n \in \text{Re} \).

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 \( A = (V, \alpha, s, \rho, P) \) over \( N \). Strings in \( V \) will be called control states of the algorithm. Given an instruction \((a_i, \alpha, s)\), the pair \((a_i, s)\) describes a binary relation in \( V \), called the control relation of the instruction. We denote this relation, which in fact is a function, by \( (a_i, s) \Rightarrow \) and, for any \( y_1, y_2 \in V^* \), we define:

\[
y_1(a_i \rightarrow s) y_2 = (3w \in V^*)y_1 y_2 w = y_2 w.
\]

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

Consider now an arbitrary label \( \alpha \) and a finite or infinite sequence of instructions

\[
(a_1, a_2, a_3, \ldots, a_n, \alpha), \quad m \leq \infty,
\]

such that there exists a sequence of control states

\[
y_1, \ldots, y_m \in V^*
\]

with the following properties:

1. \( y_1 = a_1 = \alpha, \) and \((V \leq m)[y(a_i \rightarrow s)] y_{i+1} \).
2. If \( m < \infty \), then \( y_{m+1} = \varepsilon \).

Each such a sequence of instructions will be called an \( \alpha \)-run of the algorithm.

The corresponding sequence of actions

\[
(a_1, \ldots, a_n)
\]

is called an \( \alpha \)-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

\[
C[a_1, \ldots, a_n]
\]

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

\[
Tr(\alpha).
\]

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

\[
\bigcup \{C(\tau) \mid \tau \in Tr(\alpha)\},
\]

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

For each \( a_i \in V \) we now define:

\[
\begin{align*}
\text{Run}(a_i) &= \bigcup \{C(\tau) \mid \tau \in Tr(a_i)\} \text{—} \alpha_i\text{-run element},
\text{Tail}(a_i) &= \bigcup \{C(\tau) \mid \tau \in \text{FinTr}(a_i)\} \text{—} \alpha_i\text{-tail element},
\text{Perp}(a_i) &= \bigcup \{C(\tau) \mid \tau \in \text{InfTr}(a_i)\} \text{—} \alpha_i\text{-perpetual element},
\end{align*}
\]

where \( \text{Fin} \) and \( \text{Inf} \) operators are defined as in Section 2.2. We have, of course, for any \( a_i \in V \):

\[
\text{Run}(a_i) = \text{Tail}(a_i) \cup \text{Perp}(a_i).
\]

In the sequel, by \( \text{Run}, \text{Tail}, \text{Perp}, \text{Tr}, \text{FinTr} \) and \( \text{InfTr} \), we shall mean respectively the vectors \( \text{Run}(a_1), \ldots, \text{Run}(a_n), (\text{Tail}(a_1), \ldots, \text{Tail}(a_n)) \), etc. In analyzing algorithms we shall be especially interested in the following outcomes:

\[
\text{Run}(a_i)\text{—the entire outcome of the algorithm,}
\text{Tail}(a_i)\text{—the finitistic outcome of the algorithm,}
\text{Perp}(a_i)\text{—the infinitistic outcome of the algorithm.}
\]

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

\[
P = \{(a_1, a, s); (a_1, b, a, a_2); (a_2, c, s)\}.
\]

The following are examples of \( \alpha \)-runs with explicitly written intermediate control states:

\[
a_1(a_1, a, s),
a_1(a_1, b, a, a_2) a_1 a_2 (a_1, a, a) a_3 (a_2, c, s),
a_1(a_1, b, a, a_2) a_1 a_2 (a_1, b, a, a_2) a_1 a_2 a_3 (a_1, b, a, a_2) a_1 a_2 a_3 a_4 \ldots
\]

Therefore, examples of \( \alpha \)-traces are the following:

\[
(a), (b, a, c), (b, b, a, c, c), \ldots \text{—finite traces},
(b, b, b, \ldots) \text{—infinite trace}.
\]
As can now be proved,

\[ \text{Tr}(a) = \bigcup_{n=0} ((b)^n \cup (a)^n \cup (c)^n) \cup (b)^n, \]

where \( (b)^n \) represents the concatenation of \( b \) with itself \( n \) times.

\[ \text{Run}(a) = \bigcup_{n=0} (b)^n \cup (a)^n \cup (b)^n, \]

\[ \text{Tail}(a) = \bigcup_{n=0} (b)^n \cup (a)^n, \]

\[ \text{Perp}(a) = b^n. \]

Notice that the operations appearing in the formula describing \( \text{Tr}(a) \) are operations in \( NGL((a, b, c)) \), hence, traces can be considered as generalized words. Operations in the remaining formulas are of course formulas 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 \( a, b, c \) 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 \( \text{"if } n = 0 \text{ then } s := 1\text{"} \);
- **B** is the name of \( \text{"if } n \neq 0 \text{ then } n := n - 1\text{"} \);
- **C** is the name of \( \text{"n := n + 1; s := s \times n\"} \).

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 \( N = NQR(Re \times Re) \), where \( Re \) is the set of all real numbers. We set \( a, b, \) and \( c \) to be the following \( \delta \)-relations in \( Rel(Re \times Re) \):

- \( a = \{(0, s), (1, 0), s \in Re \cup \{(\delta, \delta)\} \}
- \( b = \{(n, s), (n + 1, s), n \in Re \cup \{(\delta, \delta)\} \}
- \( c = \{(n, s), (n + 1, s \times (n + 1)), n \in 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. \( \text{Tail}(a) = \{(n, s), (n, n) \} n \in \text{Nat} \cup s \in Re \cup \{(\delta, \delta)\} \), where \( \text{Nat} \) 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. \( \text{Perp}(a) = \{(n, s), (n, n) \} n \in \text{Re} \cup \{(\delta, \delta)\} \), i.e., if the initial value of \( n \) is not a natural number, then—indeed by 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 \( N = QNR(Re \times Re) \) and we assume \( a, b, \) and \( c \) to be the following bundles of computations:

- \( a = \{(0, s), (1, 0), s \in Re \cup \{(\delta, \delta)\} \}
- \( b = \{(n, s), (n - 1, s), n \in Re \cup \{(\delta, \delta)\} \}
- \( c = \{(n, s), (n + 1, s \times (n + 1)), n \in Re \cup \{(\delta, \delta)\} \}

It is to be emphasized here that, while the choice of interpretations for \( a, b, \) and \( c \) 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 \times (n + 1)), n \in Re \cup \{(\delta, \delta)\} \}

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. \( \text{Tail}(a) = \{(n, s), (n - 1, 0), s \in \text{Nat} \cup \{(\delta, \delta)\} \}
- \( \{(n, s), (n + 1, s \times (n + 1)), n \in \text{Nat} \cup \{(\delta, \delta)\} \}
- \( \{(n, s), (n - 1, s), n \in Re \cup \{(\delta, \delta)\} \}
- \( \{(n, s), (n + 1, s \times (n + 1)), n \in Re \cup \{(\delta, \delta)\} \}

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. \( \text{Perp}(a) = \{(n, s), (n - 1, 0), s \in \text{Re} \cup \{(\delta, \delta)\} \}
- \( \{(n, s), (n + 1, s \times (n + 1)), n \in \text{Re} \cup \{(\delta, \delta)\} \}

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

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 quasinet 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 programs within quasinet 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(α) were introduced originally over nets of usual binary relations by Mazurkiewicz (29), [30], [31], [32].

Tail(α) was defined in an equivalent but different way. The first extension of these ideas to other quasirelations 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 explicitly, 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(α), we have to specify FinTr(α), 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 referring to Tr, FinTr 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, α, ρ, F) be an arbitrary PD-algorithm with V = {α1, ..., αn}. Without loss of generality (i.e., without changing Run, Tail and Perp vectors), we can assume that for any ᾱ in V and z in V* there is at most one ᾱ with (ᾱ, a, z) in P. Indeed, if there are two instructions (ᾱ1, a, z) and (ᾱ2, b, z) with the same control relation, then we can replace them by (ᾱ1, a, b, z). For every a and z appearing in an instruction, the only corresponding action will be denoted by aα. For every i ≤ n we shall set

\[ V_i = \{ z | (3a)(ᾱ, a, z) ∈ P \}. \]

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

1. If \( z = e \), then \( φ_z(x_1, ..., x_n) = e \).
2. If \( z = α_i ... α_i, \) then \( φ_z(x_1, ..., x_n) = x_i ... x_n \).

For instance, if \( z = α_1 α_2 α_3 \), then \( φ_z(x_1, ..., x_n) = x_1 x_2 x_3 \).

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

\[
\begin{align*}
\forall z & \in V^*: x_1 = \bigcup_{α \in V} a_α φ_z(α_i, ..., α_n), \\
x_n & = \bigcup_{α \in V} a_α φ_z(α_i, ..., α_n).
\end{align*}
\]

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

\[
\begin{align*}
x_1 & = b x_1 x_2 \cup a, \\
x_2 & = c.
\end{align*}
\]

A PD-algorithm \( A = (V, α, ρ, F) \) is said to be admissible if either \( a_n 0 = 0 \) for all \( a_n \in V \) and \( i \leq n \), or \( \text{FinTr}(α) \not= \emptyset \) for \( i \leq n \).

**Theorem 4.3.1.** For every admissible PD-algorithm the vector (Tail(α), ... , Tail(α)) 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(α) by solving the appropriate CSE, e.g., considering the CSE (4.3.2), we can solve it in the following way:

\[
\begin{align*}
x_1 & = b x_1 x_2 \cup a, \\
x_2 & = c,
\end{align*}
\]

therefore,

\[
\begin{align*}
x_1 & = b x_1 x_2 \cup a, \\
x_2 & = c,
\end{align*}
\]

hence (by Lemma 3.2.1),

\[
\begin{align*}
x_1 & = \bigcup_{α \in V} b α c^* a, \\
x_2 & = c.
\end{align*}
\]

We get, therefore, Tail(α) = \( \bigcup b α c^* a \) and Tail(α) = c (cf. Example 3.2.1).

To get a fixed point characterization of the Run vector, we now introduce an auxiliary notion. A sequence \( t \in \text{seq}(U) \) 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 \( a \in U \)

\[ C(t) \cdot a = C(t). \]

The following facts about proper sequences can be easily observed:

1. Each finite sequence is proper.
2. Each sequence \( t \) with \( C(t) = \emptyset \) is proper.
3. Each sequence in \( \text{NTR}(D) \) is proper (since for any infinite \( t \), \( C(t) \subseteq D \)).
4. Each infinite sequence \( t \) = \( (L_1, L_2, ...) \) of languages with an infinite number of \( \epsilon \)-free \( L_i \)'s (\( \epsilon \not= L_i \)) is proper (since in this case

\[ C(t) \subseteq \text{seq}^\epsilon(D) \cup \emptyset. \]

(5) Each infinite sequence \( t \) = \( (B_1, B_2, ...) \) 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

\[ C(t) \subseteq \text{seq}^\epsilon(D) \cup \emptyset. \]
We introduce now another operation

\[ C : \text{GLan}(U) \to U, \]

defined for any \( S \subseteq \text{seq}(U) \), as follows:

\[ C(S) = \bigcup \{ C(t) \mid t \in S \}. \]

This operation associates sets of sequences (traces) in \( U \) with elements of \( U \).

In fact, for any algorithm \( A \) and any label \( a_1 \) in this algorithm, we have

\[ \text{Run}(a_1) = C(\text{Tr}(a_1)), \]

\[ \text{Tail}(a_1) = C(\text{InfTr}(a_1)). \]

**Lemma 4.4.1.** For any family \( \{ S_p \mid p \in P \} \) with \( S_p \subseteq \text{seq}(U) \) we have

\[ C_{\bigcup_P}(S_p) = \bigcup_{p \in P} C(S_p). \]

Easy proof is left to the reader. A set \( S \subseteq \text{seq}(U) \) 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 \( S_1, S_2 \subseteq \text{seq}(U) \) let \( S_1 \sim S_2 \) denotes their concatenation within \( \text{GLan}(U) \), as defined in Section 2.3.

**Lemma 4.4.2.** For any finite or denumerable sets \( S_1, S_2 \subseteq \text{seq}(U) \), if \( S_1 \) is proper, then

\[ C(S_1 \sim S_2) = C(S_1) C(S_2), \]

unless \( S_1 \neq \emptyset = S_2 \) and \( C(S_1) \emptyset \neq 0 \).

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

Now let \( A = (F, a_1, e, P) \) be an admissible algorithm over \( N \) and let \( V = \{ a_1, \ldots, a_n \} \). By analogy to \( a_n \)-traces, we can define \( x \)-traces for arbitrary \( x \in F^* \), \( \text{Tr}(x) \) will denote the set of all \( x \)-traces. We shall assume \( \text{Tr}(x) = \{ x \} \).

**Lemma 4.4.3.** For any \( x \in F^* \), if \( x = x_1 \ldots x_n \), then

\begin{enumerate}
  \item \( \text{Tr}(x) = \text{Tr}(x_n) \ldots \text{Tr}(x_2) \),
  \item \( \text{FinTr}(x) = \text{FinTr}(x_n) \ldots \text{FinTr}(x_2) \),
  \item \( \text{InfTr}(x) = \text{InfTr}(x_n) \ldots \text{InfTr}(x_2) \).
\end{enumerate}

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

\[ \text{Tr}(x) = \text{Tr}(x) \sim \text{Tr}(x) \]

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

Let \( t \in \text{Tr}(x) \sim \text{Tr}(x') \) with \( t = t_1 \sim t_2, t_1 \in \text{Tr}(x), t_2 \in \text{Tr}(x') \). If \( t_1 \) is infinite, then of course \( t = t_1 \in \text{Tr}(x) \). If \( t_1 \) is finite, then let \( y_1, \ldots, y_k \) and \( z_1, \ldots, z_n \) be...
sequences of control states corresponding to $t_1$ and $t_2$, respectively. It is easy to show that $t_1^{-1} t_2$ is an $x$-trace with the sequence of control states $z_1, z_2, \ldots, z_n$.

- **Lemma 4.4.4.** For any $i \leq n$
  (1) $\text{Tr}(a_i) = \bigcup_{x \in (a_i)_m}^{x \neq n} x^{-1}\text{Tr}(x)$.
  (2) $\text{InfTr}(a_i) = \bigcup_{x \neq n}^{x \neq n} (a_i)_m^{-1}\text{InfTr}(x)$.
  (3) $\text{FinTr}(a_i) = \bigcup_{x \neq n}^{x \neq n} (a_i)_m^{-1}\text{FinTr}(x)$.

Proof: Obvious by the definition of $\text{Tr}(a_i)$ and $\text{Tr}(x)$.

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 $m \geq 1$ let $\text{Tr}^m(x)$ denote the set of all $x$-traces of length not greater than $m$. Clearly,

$$\text{FinTr}(x) = \bigcup_{n=1}^{m} \text{Tr}^m(x)$$

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

$$\text{Tail}(a) = \bigcup_{m=0}^{t^m} t^m$$

for $i = 1, \ldots, n$. We shall show that for $i \leq m$ and $m \geq 1$, $t^m \equiv d_i$. (The proof is by induction on $m$).

Initial step: $t^0 \equiv a_\infty \equiv \bigcup_{x \neq n}^{x \neq n} (a_\infty)_m^{-1}\text{Tr}(x)$.

Induction step: Let $t^k \equiv d_i$ for $k \leq m$ and $t^{k+1} \equiv d_i$. Of course, for any $i \leq m$

$$t^{m+1} = C(\text{Tr}^{m+1}(a)) = C\left(\bigcup_{x \neq n}^{x \neq n} (a_\infty)_m^{-1}\text{Tr}(x)\right) = \bigcup_{x \neq n}^{x \neq n} C(\text{Tr}^m(x)).$$

It is also clear that for any $x = (x_0, x_1, \ldots, x_{l-1})$ in $V^*$, where $l(x)$ is the length of $x$,

$$\text{Tr}^m(x) \subseteq \text{Tr}^m(x_0) \subseteq \cdots \subseteq \text{Tr}^m(x_{l-1}).$$

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

$$t^{m+1} = \bigcup_{x \neq n}^{x \neq n} a_\infty C(\text{Tr}^m(x))$$

$$\equiv \bigcup_{x \neq n}^{x \neq n} C(\text{Tr}^m(x_0)) \subseteq \cdots \subseteq \text{Tr}^m(x_{l(x)}).$$

For the CSE, we can write the following:

$$x_i = a_\infty x_i \cup \cdots \cup a_{x_i} x_i \cup a_{x_i+1} x_i \cup a_{x_i+1}$$

(4.5.1)

It is also clear that for any $z = (x_0, x_1, \ldots, x_{l-1})$ in $V^*$, where $l(x)$ is the length of $x$,

$$\text{Tr}^m(x) \subseteq \text{Tr}^m(x_0) \subseteq \cdots \subseteq \text{Tr}^m(x_{l-1}).$$

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

$$t^{m+1} = \bigcup_{x \neq n}^{x \neq n} a_\infty C(\text{Tr}^m(x))$$

$$\subseteq \bigcup_{x \neq n}^{x \neq n} C(\text{Tr}^m(x)) \subseteq \cdots \subseteq \text{Tr}^m(x_{l(x)}).$$

For the CSE, we can write the following:

$$x_i = a_\infty x_i \cup \cdots \cup a_{x_i} x_i \cup a_{x_i+1} x_i \cup a_{x_i+1}$$

(4.5.2)

**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 $A = (V, x_1, x_2, \ldots, x_n)$ be a PD-algorithm with $V = (x_1, \ldots, x_n)$. Its algorithm is said to be a *finite-control algorithm* (abbrev. FC-algorithm), provided each instruction is either of the form $(a_1, a_2, a_3)$ or of the form $(a_1, a_2, a_3)$ with $x_1, x_2 \in V$.

Instructions of this form have a very natural ALGOL-like interpretation:

$$(a_1, a_2, a_3) \text{ as } "a_1: \text{ do } a_2; \text{ go to } a_3"$$

$$(a_1, a_2, a_3) \text{ as } "a_1: \text{ do } a_2; \text{ go to } a_3"$$

Denote $a_{n+1} = \infty$. We shall make the assumption, as we did in Section 4.3, that for each $x_1, x_2$ with $i \leq n, j \leq n+1$ there is at most one $a$ with $(x_1, a, x_2) \in P$.

We shall denote this $a$ by $a$ and we shall set $a_0 = 0$ whenever there is no $a$ with $(x_1, a, x_2) \in P$.

The CSE can be written now as follows:

$$x_i = a_{x_1} x_i \cup \cdots \cup a_{x_i} x_i \cup a_{x_i+1} x_i \cup a_{x_i+1}$$

(4.5.1)

$$x_i = a_{x_1} x_i \cup \cdots \cup a_{x_i} x_i \cup a_{x_i+1} x_i \cup a_{x_i+1}$$

(4.5.2)

Whenver $(\text{Tail}(a_1), \ldots, \text{Tail}(a_n))$ is the least solution of this set it can be computed effectively and expressed by means of $a_0$ and the operations $\cup$, $\bullet$, and $*$ (Section 3.3). The vector $(\text{Run}(a_1), \ldots, \text{Run}(a_n))$ can now be proved to be a solution of CSE without assuming that the algorithm is proper.

Moreover, we can get an algorithm (in a wide sense of the term) that, starting with the CSE, produces formulas expressing the Run vector by means of $a_0$ and the operations $\cup$, $\bullet$, and $*$. The main idea consists in a modified variable elimination method. While in computing the least solution we eliminate $x_i$ from the equation

$$x_i = a_{x_1} x_i \cup \cdots \cup a_{x_i} x_i \cup a_{x_i+1} x_i \cup a_{x_i+1}$$

by setting (Lemma 3.2.1)

$$x_i = a_{x_1} x_i \cup \cdots \cup a_{x_i} x_i \cup a_{x_i+1} x_i \cup a_{x_i+1}$$

in order to get the Run-vector, we set

$$x_i = a_{x_1} x_i \cup \cdots \cup a_{x_i} x_i \cup a_{x_i+1} x_i \cup a_{x_i+1}$$

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

The operation $C$ (fixed at the beginning of Section 4.1) is said to be *adequate* provided for any proper set $S \subseteq S^n(U)$ with $\infty \in S$ we have

$$\bigcup C(S) \subseteq (\bigcup C(S) \subseteq (\bigcup C(S) \subseteq \infty$$

(4.5.2)

$$C(S^n) = \infty$$

i.e.,
Observe that $S^m$ means the $m$-iteration of $S$ within $NGL(U)$, while $(C(S))^m$ is effectuated within the abstract quasinet $N$.

It can be proved (A. Lingas, personal communication), that there exist quasinets with nonadequate $C$-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 $QN0(D)$, are adequate.

The proof is in Blikle [14].

Let $a_i, a_j$ be arbitrary labels with $i \leq n, j \leq n+1$. By an $a_i, a_j$-trace in $A$ we mean any finite sequence $(a_i, ..., a_j)$ of elements of $U$ such that there exist labels $b_1, ..., b_{n+1}$ with the following properties:

1. $b_1 = a_i$ and $b_{n+1} = a_j$,
2. $(b_p, a_p, b_{p+1}) \in P$ for $p = 1, ..., m$.

We denote by $Tr(a_i, a_j)$ the set of all $a_i, a_j$-traces in $A$.

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

\[ x_1 = d_1 x_1 x_2 \ldots x_m x_{n+1} \]

\[ x_2 = d_2 x_1 x_2 \ldots x_m x_{n+1} \]

\[ x_3 = d_3 x_1 x_2 \ldots x_m x_{n+1} \]

\[ x_4 = d_4 x_1 x_2 \ldots x_m x_{n+1} \]

Suppose that the $C$ operation in $N$ is adequate and (4.5.3) satisfies the following condition:

- there exist sets $D_{ij} \subseteq \text{seq}(U)$, $i \leq n, j \leq n+1$, such that:
  1. $e \notin D_{ij}$ and $d_0 = C(D_0)$ for $i \leq n, j \leq n+1$,
  2. $D_{ij} \subseteq Tr(a_i, a_j)$ for $i, j \leq n$,
  3. $Tr(a_i) = D_{ij} \cup Tr(a_i) \cup D_{i+1} \cup D_{i+n+1}$ for $i \leq n$.

Then $(\text{Run}(a_i), ..., \text{Run}(a_n))$ is a solution of (4.5.4), and (4.5.4) satisfies (\ast) as well.

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

\[ Tr(a_i) = D_{ij} \cup Tr(a_i) \cup D_{i+1} \cup D_{i+n+1} \]

where for simplification the symbols $\leftarrow \rightarrow$ of concatenation are omitted. The inclusion "\rightarrow" is obvious by (2) and (3) (the latter implies $D_{i+1} \subseteq Tr(a_i)$ for $i \leq n$).

To show the converse inclusion, consider an arbitrary $t \in Tr(a_i)$. By (3):

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

\[
\begin{align*}
(x_1, a) & \rightarrow (a_1, a_2) \rightarrow (a_3, a_4), \\
(x_2, b_1, a_3) & \rightarrow (a_4, d_1, a_3) \rightarrow (a_5, g_2, a_6), \\
(x_3, b_2, a_4) & \rightarrow (a_6, h, a_7),
\end{align*}
\]

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

\[
\begin{align*}
x_1 &= ax_2, \\
x_2 &= b_1x_3 + b_2x_4, \\
x_3 &= cx_5, \\
x_4 &= dx_6, \\
x_5 &= g_2x_6 + g_1, \\
x_6 &= hx_7.
\end{align*}
\]

(4.5.6)

Suppose now that all we want to get is \( \text{Run}(a_1) \). 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

\[
\begin{align*}
x_1 &= ax_2, \\
x_2 &= b_1cx_3 + b_4dx_4, \\
x_3 &= g_2hdx_5 + g_7, \\
x_4 &= hx_6.
\end{align*}
\]

We now apply our transformation to \( x_2 \) and \( x_3 \):

\[
\begin{align*}
x_1 &= ax_2, \\
x_2 &= (b_1c)x_3 + b_2dx_4, \\
x_3 &= (g_2hd)x_5 + (g_7h)d^m, \\
x_4 &= hx_6.
\end{align*}
\]

Now we substitute again and get

\[
\text{Run}(a_1) = a(b_1c)x_3 + b_2dx_4((g_2hd)x_5 + (g_7h)d^m) + a(b_1c)m.
\]

Notice that, if we wish to find \( \text{Tail}(a_1) \), we apply to (4.5.6) the usual solving algorithm described in Section 3.2, and we get:

\[
\text{Tail}(a_1) = a(b_1c)x_3 + b_2dx_4((g_2hd)x_5 + (g_7h)d^m).
\]

In the case of FC-algorithms we also get a fixpoint characterization of the \( \text{Perp} \) vector.

Theorem 4.5.1. For any FC-algorithm the vector \( (\text{Perp}(a_1), \ldots, \text{Perp}(a_2)) \) is a solution of the following set of equations:

\[
\begin{align*}
x_1 &= a_1x_2 + \ldots + a_{14}x_{13}, \\
& \quad \ldots \\
x_8 &= a_{61}x_9 + \ldots + a_{63}x_{69}.
\end{align*}
\]

(4.5.7)

In the general case \( \text{Perp} \) vector need 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 \( \text{Perp} \) vector from (4.5.7). We need, however, one more assumption about the algorithm to the effect that \( a_{ij} = 0 \) for \( i \leq n, j \leq n+1 \) (details in 14).

5. ALGORITHMS OVER BUNDLES OF COMPUTATIONS

This section is devoted to algorithms over \( \mathcal{QNB}(D) \). 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 emptying 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 \( \mathcal{QNB}(D) \), let \( f : D \rightarrow D \) be an arbitrary partial function and let \( p : D \rightarrow \{\text{true}, \text{false}\} \) be an arbitrary partial predicate in \( D \). We denote

\[
\begin{align*}
[p(x)]_x &= f(x), \\
[p(x)]_d &= \{d \in D. f(d) = \text{true} \}, \\
[p(x)]_{\{d\}} &= \{d \in D. f(d) = \text{true} \}, \\
[p(x)]_{\{d\}} &= \{d \in D. f(d) = \text{true} \}.
\end{align*}
\]

We shall also write \( [x \leftarrow f(x)] \) instead of \( [\text{true}] \leftarrow f(x) \) and \( [d] \) instead of \( [x = d] \).

Let \( p \) and \( q \) be arbitrary predicates and let \( c = (d_1, \ldots, d_m) \) with \( 1 \leq m \leq \infty \) be an arbitrary (finite or infinite) computation. We say that \( c \) satisfies \( p \) at the entrance if \( d_1 = \text{true} \) and we say that \( c \) satisfies \( p \) at the exit if either \( m = \infty \) or \( m < \infty \) and \( p(d_m) = \text{true} \). 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 \( [p(x)]_B \) are subsets of the unit \( B \) in the monoid of bundles. In consequence, for any bundle \( B \) and any predicate \( p \), we get the following subsets of \( B \):

\[
[p(x)]_B = \{x \in B. [p(x)]_x \}
\]

where \( [p(x)]_x \) is the bundle of all computations in \( B \) that satisfy \( p \) at the entrance, \( B[p(x)] \) is 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. \( B = [p(x)]_B \) for each \( p(x) \) —the bundle of all computations in \( B \) that satisfy \( p \) at the entrance,
2. \( B[p(x)] = [p(x)]_B \) for each \( p(x) \) —the bundle of all computations in \( B \) that satisfy \( p \) at the exit,
3. \( B[p(x)] = [p(x)]_B \) for each \( p(x) \) —the bundle of all computations in \( B \) that satisfy \( p \) at the exit.

Observe that (4) is equivalent to

\[
[p(x)]_B = [p(x)]_B[p(x)]
\]
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

\[ \text{Eut}(B) = \{d \mid [d]B \neq 0 \}, \]

\[ \text{Ext}(B) = \{d \mid [d]B \cap \text{inf}(D) \neq 0 \}, \]

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 \((\text{Run}(a_1), \ldots, \text{Run}(a_n))\) 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 \(Q_i = \text{Tail}(a_i) \cup S_i\) with \(\text{Tail}(a_i) \cap S_i = 0\) for \(1 \leq i \leq n\), then

\[ S_i \subseteq \text{Perp}(a_i) \]

for all \(1 \leq i \leq n\).

The proof is immediate since, by Theorem 5.2.1, \(\text{Tail}(a_i) \cup S_i \subseteq \text{Tail}(a_i) \cup \text{Perp}(a_i)\), but \(\text{Tail}(a_i) \cap S_i = 0\), hence \(S_i \subseteq \text{Perp}(a_i)\).

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 \(\text{Run}(a_i)\) 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 \((\text{Tail}(a_1), \ldots, \text{Tail}(a_n))\) is the only finitistic solution of CSE.

This is an immediate consequence of Corollary 5.2.1 since for proper algorithms \(\text{Perp}(a_i)\) is always inherently infinitistic.

**Theorem 5.2.2.** For any proper and finitistic FC-algorithm with the set of instructions \(P = \{(a_i, B_i, \beta_i) \mid 1 \leq i \leq n; j \leq n+1\}\) the vector \((\text{Perp}(a_1), \ldots, \text{Perp}(a_n))\) is the greatest solution of the following set of equations:

\[ X_1 = B_1 X_1 \cup \ldots \cup B_n X_n, \]

\[ \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \]

\[ X_n = B_1 X_1 \cup \ldots \cup B_n X_n. \] (5.2.1)

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—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 NINETEEN;

```
procedure NINETEEN;
begin
  if \(x > 100\) then \(x := x - 10\)
  else begin
    \(x := x + 11\);
    NINETEEN;
  end;
end;
```

We assume \(D = \text{Int}\) and we set:

- \(Q = [x > 100] x := x - 10\), \(a_1, \delta, \lambda\),
- \(F = [x < 100] x := x + 11\), \(a_1, F, a_1\).

The CSE is

\[ X = FXX \cup Q. \]

Now let

\[ C_0 = [100]QQ, \]

\[ C_i = [100 - i]F \cup C_{i-1} \quad \text{for} \quad 1 \leq i \leq 9, \]

\[ C = C_0 \cup C_1 \cup \ldots \cup C_9. \]
We shall prove the following:

\[ \text{Tail}(a_i) = \bigcup_{i=1}^{\infty} FQCC_i^{a_i} \cup \bigcup_{i=1}^{\infty} FQ^2C_i^{a_i} \cup Q. \]

This equation shows explicitly what all the finite computations of our procedure look like. To prove (3.3.1), we set

\[ B_1 = \bigcup_{i=1}^{\infty} FQCC_i^{a_i}, \quad B_2 = \bigcup_{i=1}^{\infty} FQ^2C_i^{a_i}, \quad R = B_1 \cup B_2 \cup Q. \]

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. \( C = [91 \leq x \leq 100]C[91], \)
2. \( C_0 = [91]C[91], \)
3. \( B_1 = B_2[91], \)
4. \( B_2 = B_2[91], \)
5. \( [91]B_1 = 0, \)
7. \( B_1Q = [91]Q = 0, \)
8. \( B_2Q = [91]Q = 0, \)
9. \( 100[B_2] = 0, \)
10. \( FQ = FQ[91] \leq x \leq 100, \)
11. \( [90 \leq x < 99]B_1 = 0. \)

Now we can show \( R \) to be a solution of CSE. Compute:

\[ \text{FRR} \cup Q = \bigcup_{i=2}^{\infty} FQCC_i^{a_i} \cup \bigcup_{i=2}^{\infty} FQ^2C_i^{a_i} \cup FQC \cup Q = R. \]

In effect, we get

\[ [d] \text{Tail}(a_i) = [d]Q = \{(d, d-10), \epsilon\}, \]

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

\[ [100-11 \times n] \text{Tail}(a_i) = [100-11 \times n]B_2 = [100-11 \times n]FQ[91]C[91], \]

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

\[ [100-11 \times n-10] \text{Tail}(a_i) = [100-11 \times n-10]B_1 = [100-11 \times n-10]FQ[91]C[91]. \]

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 \( D = \text{Int}, \) since \( \text{Entr}(\text{Tail}(a_i)) = \text{Int} \) and the algorithm is deterministic in the sense that any two different computations are disjoint. In consequence, \( \text{Perp}(a_i) = 0. \)

---

**Example 5.3.2.** Consider a sequential consumer-producer program, as given by the flowchart of Fig. 5.3.1. We shall assume \( z_1, z_2, q_1, q_2 \) 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 \((F_1, F_3)^\omega\), 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

- \(P_1 = [x \geq s_1] (x, y) \leftarrow (x, y)\)
- \(\overline{P_1} = [x < s_1] (x, y) \leftarrow (x, y)\)
- \(Q_1 = [(x, y) \leftarrow (x-s_1, y+q_1)]\)
- \(P_2 = [y \geq s_2] (y, x) \leftarrow (y, x)\)
- \(\overline{P_2} = [y < s_2] (y, x) \leftarrow (y, x)\)
- \(Q_2 = [(x, y) \leftarrow (x+q_2, y-s_2)]\)
- \(D = \text{Nat} \times \text{Nat}\)

The CSE is

\[
X_1 = P_1 Q_1 X_1 \cup \overline{P_1} X_1,
\]

\[
X_2 = P_2 Q_2 X_2 \cup \overline{P_2} X_2.
\]

To get \(\text{Run}(a_i)\) from this set, we apply the algorithm described in Section 4.5:

\[
X_1 = (P_1 Q_1)^* \overline{P_1} X_1 \cup (P_1 Q_1)^\omega\]

\[
X_2 = (P_2 Q_2)^* \overline{P_2} X_2 \cup (P_2 Q_2)^\omega.
\]

It is easy to prove now that \((P_1 Q_1)^\omega = (P_2 Q_2)^\omega = 0\), hence

\[
X_1 = (P_1 Q_1)^* \overline{P_1} (P_2 Q_2)^* \overline{P_2} X_1,
\]

therefore

\[
X_1 = ((P_1 Q_1)^* \overline{P_1} (P_2 Q_2)^* \overline{P_2})^\omega \cup (P_1 Q_1)^\omega \overline{P_1} (P_2 Q_2)^\omega \overline{P_2}^\omega,
\]

and finally,

\[
\text{Run}(a_i) = ((P_1 Q_1)^* \overline{P_1} (P_2 Q_2)^* \overline{P_2})^\omega.
\]

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

\[
R = (P_1 Q_1)^* \overline{P_1} (P_2 Q_2)^* \overline{P_2}.
\]

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

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

(i) For every \(k \geq 0\), \([(n, m)]^R (F_1, F_3)^\omega = 0\), i.e., deadlock never occurs,

(ii) Overload does occur for both \(x\) and \(y\).

Case 2. \((q_1 -s_1) + (q_2 -s_2) < 0\): underload.

In this case, for any \(n, m \geq 0\), there exists \(k \geq 0\) such that

\[
[(n, m)]^R (F_1, F_3)^\omega = 0.
\]

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. \(q_1 - s_1 = 0, q_2 - s_2 = 0, n \geq s_1, m \geq s_2\). In this subcase we can prove the following:

(i) For any \(k \geq 0\), \([(n, m)]^R (F_1, F_3)^\omega = 0\), i.e., deadlock never occurs,

(ii) For any \(k \geq 0\), if \([(n, m)]^R (F_1, F_3)^k = (n, m)]^R (F_1, F_3)^k\), then \(n+m = n_1 + m_1\), which means that \(x+y\) is an invariant of the program. In consequence, the overload will not occur.

6. ALGORITHMS OVER 4-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 \(\Omega\) be an arbitrary set. If \(R \in \text{Rel}(\Omega)\) and \(C \subseteq \Omega\), then \(CR\) and \(RC\) denote subsets of \(\Omega\) defined as follows:

\[
a \in CR = \{c \in C \mid c Ra\},
\]

\[
a \in RC = \{c \in C \mid a Rc\}.
\]

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

\[
R_1(R_2 C) = (R_1 R_2) C,
\]

\[
C(R_1 R_2) = (C R_2) R_1,
\]

\[
(\bigcup_{i \in I} R_i) C = \bigcup_{i \in I} R_i C,
\]

\[
C (\bigcup_{i \in I} R_i) = \bigcup_{i \in I} C R_i,
\]

\[
(\bigcap_{j \in J} C_j) R_1 = \bigcap_{j \in J} C_j R_1,
\]

\[
R_1 (\bigcap_{j \in J} C_j) = \bigcap_{j \in J} R_1 C_j.
\]
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 $(\Omega_1, \ldots, \Omega_n)$ and $(\Omega_1, \ldots, \Omega_n)$. 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., $Q = \{(n, s), \{n, s-1\}, s \in \mathbb{N}\}$. In the sequel we shall use a notation introduced by Mazurkiewicz for the case of functions in $\text{Rel}(D)$ and extended here to functions in $\text{Rel}(D)$. Let $f : D \rightarrow D$ be an arbitrary function and let $\mu : D \rightarrow \{\text{true}, \text{false}\}$ be an arbitrary predicate in $D$. We denote:

$$[p(x)] x := f(x) = \{(d, f(d)) \mid p(d) = \text{true} \land d \in D\} \cup \{(\delta, \delta)\}.$$  

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

$$[x := f(x)] \text{ instead of } [\text{true} \mid x := f(x)],$$

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

The following equalities are useful in calculations:

1. $[p(x)] x := f(x) \equiv [p(x) \land g(x)] x := g(x)$.
2. $[p(x)] x := f(x) \lor g(x) \equiv [p(x) \lor g(x)] x := f(x)$.
3. $[p(x)][g(x)] x := f(x) \equiv [g(x)][p(x)] x := f(x)$.
4. $p(x) \equiv q(x)$, then $[p(x)][g(x)] x := f(x) \equiv [q(x)][g(x)] x := f(x)$.

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 $\delta$-relations.

In the remainder of Section 6, $\delta$-relations in $\text{Rel}(D)$ 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 $\text{Rel}(D)$ by capitals $X, X_1, X_2, \ldots$. For any $B \subseteq D$ we shall set $B = B \cup \{\delta\}$, hence, in particular, $D_0 = D \cup \{\delta\}$. We shall also say simply "relation" instead of "$\delta$-relation".

Let $\delta = (V, a_1, a_2, P)$ with $V = \{a_1, \ldots, a_n\}$ be an arbitrary PD-algorithm over $QNR(D)$ to be fixed for the sequel. The CSE for this algorithm is the following:

$$X_1 = \bigcup_{i \in I} P_i \{X_1, \ldots, X_i\},$$

$$X_2 = \bigcup_{i \in I} \{X_i \mid P_i\{X_1, \ldots, X_i\}\}.  $$

where $(a_1, a_2, \ldots, a_n)$ are instructions in $P$ (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 $R_i$ are finitistic (see Section 2.9) and is said to be nonfinitistic otherwise. For the majority of simple programs we deal, of course, with finitistic algorithms but, similarly as for $QNR(D)$, 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 $(a_i, R, z)$ can be considered as a binary relation in the set $V \times D$ of state vectors:

$$(y_1, d_i) \in (\exists w) \{y_2 = a_i w \land y_3 = z w \land d_i R d_i \}$$

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

$$(y_1, d_i) \Rightarrow_A (y_2, d_i) \Leftrightarrow (\exists (a_i, R, z) \in P)(y_1, d_i)(a_i, R, z)(y_2, d_i).$$

It is interesting to note, that for each $a_i \in V$ and any $d_i, d_j \in D$

$$d_i \text{ Tail}(a_i) d_j \Rightarrow_A (a_i, d_i)(\Rightarrow_A)^* (a_i, d_j).$$

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

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 $a_i \in V$, we set:

$$\text{Stop}(a_i) = \text{Tail}(a_i) D_i,$$

$$\text{Loop}(a_i) = \text{Perp}(a_i) \{\delta\} \setminus \{\delta\},$$

$$\text{Block}(a_i) = D \setminus (\text{Stop}(a_i) \cup \text{Loop}(a_i)).$$

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

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

$$\text{Stop}(a_i)$$ —the set of all inputs, where the algorithm terminates its run reaching the end,

$$\text{Loop}(a_i)$$ —the set of all inputs, where the algorithm runs indefinitely,

$$\text{Block}(a_i)$$ —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 $\text{Run}(a_i), \text{Tail}(a_i)$ and Perp$(a_i)$ are all functions for $i \in n$.

Lemma 6.2.2. If $A$ is finitistic, then for each $i \in n$, $\text{Tail}(a_i) \cap \text{Perp}(a_i) = \emptyset$.

Lemma 6.2.3. If $A$ is finitistic, then for each $i \in n$, if $\text{Run}(a_i) \cap T_i F_i$, where $T_i$ is finitistic and $F_i$ is inherently infinitistic, then $T_i \in \text{Tail}(a_i)$ and $F_i \in \text{Perp}(a_i)$.
Lemma 6.2.4. If $A$ is deterministic, then for each $i \leq n$, $\text{Tail}(a_i) \cap \text{Perp}(a_i) = \emptyset$ and $\text{Stop}(a_i) \cap \text{Loop}(a_i) = \emptyset$.

The proofs are obvious.

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

$$(a_1, \{x \geq 0 | x := 2x\}, s),$$

$$(a_1, \{x < 0 | x := -1\}, s).$$

The $\text{CSE}$ is now as follows:

$$X = \{x < 0 | x := -1\} X \cup \{x \geq 0 | x := 2x\}.$$

From this, by Theorem 4.3.1 and the algorithm, to get $\text{Run}$ vector which was described in Section 4.5, we get

$$\text{Tail}(a_1) = \{x \geq 0 | x := 2x\},$$

$$\text{Run}(a_1) = \{x \geq 0 | x := 2x\} \cup \{x < 0 | x := -1\}.$$

On the other hand, each function

$$\phi_x = \{x \geq 0 | x := 2x\} \cup \{x < 0 | x := -1\}$$

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

$$\phi_x = \{x \geq 0 | x := 2x\} \cup \bigcup_{d \in D} \{x < 0 | x := -1\}$$

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

$$\{x < 0 | x := -1\} = \text{Loop}(a_1).$$

Therefore, any of the above solutions consists of $\text{Tail}(a_1)$ and of a component disjoint with $\text{Tail}(a_1)$, whose domain is $\text{Loop}(a_1)$.

This fact can be generalized into the following theorem:

Theorem 6.2.1. Let $(Q_1, ..., Q_n)$ be an arbitrary solution of $\text{CSE}$. If $Q_i = \text{Tail}(a_i) \cup S_i$, where $\text{Tail}(a_i) \cap S_i = \emptyset$ for $i \leq n$, then

$$S_i D_i = \{d \} \subseteq \text{Loop}(a_i)$$

for $i \leq n$.

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

$$(6.2.1) \quad Q_i = \bigcup_{d \in D_i} R_d \phi(Q_i, ..., Q_i); \quad i = 1, ..., n.$$

We prove the following auxiliary assertion:

$(\star)$ For any control state $w \in F^*$, $w \neq e$ and any $d, g \in D_1$ with $d \phi(Q_1, ..., Q_n)$ there exist:

—an instruction $(a_1, R_{a_1}, e),$
Both above corollaries can be used in proofs of nontermination of programs.

**Corollary 6.3.6.** If \( A \) is deterministic, then \((\text{Run}(s_1), \ldots, \text{Run}(s_n))\) is a maximal element in the set of all functional solutions of \( \text{CSE} \).

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 \( \text{CSE} \) (cf. the example preceding Theorem 6.2.5). The corollary says that each extension of the \( \text{Run} \) vector that is a solution of \( \text{CSE} \) 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 \((Q_0, \ldots, Q_n)\) has been shown to be a solution of the corresponding \( \text{CSE} \). Since the \( \text{Tail} \) vector is of course the least solution of the \( \text{CSE} \), we obtain therefore

\[
\text{Tail}(s_i) \subseteq Q_i.
\]

Suppose now \( Q_i \) to be a function defined "explicitly" by the equation

\[
Q_i = [p(x); x := f(x)].
\]

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

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

Observe that \( p(d) = \text{true} \) 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 \((Q_0, \ldots, Q_n)\) is a solution of \( \text{CSE} \). A difficult problem that arises in such proofs is to get the appropriate \( Q_i \)'s. We usually know what the \( \text{Run}(s_i) \)—or at least \( \text{Tail}(s_i) \)—should look like, since this is the intended meaning of the program (algorithm), but at the same time, we know very little about \( \text{Run}(s_i) \) or \( \text{Tail}(s_i) \) for \( i \geq 2 \).

Suppose now that we have proved \((Q_0, \ldots, Q_n)\) to be the least solution of \( \text{CSE} \). This means

\[
\text{Tail}(s_i) = Q_i = [p(x); x := f(x)],
\]

and now we can claim the following:

(\**\*) For any \( d \in D \), if \( p(d) = \text{true} \), then if \( d \) is the initial value, the algorithm stops, and the terminal value is \( f(d) \).

In this case \( p(d) = \text{true} \) 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:

(\*\*) For any \( d \in D \), \( p(d) = \text{true} \) 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., \( d \in \text{Loop}(A) \);
2. the program stops before the control reaches the terminal state \( e \), i.e., \( d \in \text{Block}(A) \).

Analyzing programs by means of \( \text{Run} \) 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 \( \delta \)-relations

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

- \( \text{Re} \) —the set of all real numbers,
- \( \text{Nat} = \{0, 1, \ldots\} \),
- \( \text{Nat}^* = \{1, 2, \ldots\} \),
- \( \text{Int} = \{-\ldots, -1, 0, 1, \ldots\} \).

**Example 6.4.1.** Consider again the procedure FACTORIAL from Examples 4.1.1 and 4.2.1.

```plaintext
procedure FACTORIAL;
begin
if \( n = 0 \) then \( s := 1 \)
else begin
\( n := n - 1; \)
FACTORIAL;
\( s := s \times n; \)
end;
end;
```

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

**Notation**

- The algorithm

\[
Q = [n = 0; (n, s) := (n, 1)],
\]

\[
F = [n \neq 0; (n, s) := (n - 1, s)],
\]

\[
S = [(n, s) := (n + 1, s \times (n + 1))];
\]

The corresponding \( \text{CSE} \) is the following:

\[
X_1 = QX_1 \cup Q,
\]

\[
X_2 = S,
\]
and this is equivalent, by Theorem 3.2.3, to the set

\begin{align}
X_1 &= FX_0 S \cup Q, \\
X_2 &= S.
\end{align}

We state now the following hypothesis

\[
\text{Tail}(s_1) = [n \in \mathbb{N}] (n, s) := (n, n!)],
\]

\[
\text{Perp}(s_1) = [n \neq \emptyset \land (n, s) := \emptyset],
\]

and we shall try to prove it. Denote \( R = [n \in \mathbb{N}] (n, s) := (n, n!)] \) and \( P = [n \neq \emptyset \land (n, s) := \emptyset] \). We show first \((R, S)\) to be a solution of (6.4.1):

\[
FRS \cup Q = FR[n \in \mathbb{N}] (n, s) := (n, n!)] S \cup Q \\
= [n \neq \emptyset \land (n-1) \in \mathbb{N}] (n, s) := (n-1, (n-1)!)] S \cup Q \\
= [n \neq \emptyset \land (n-1) \in \mathbb{N}] (n, s) := (n, n!)] S \cup Q \\
= [n \in \mathbb{N}] (n, s) := (n, n!)] R.
\]

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 \( n \in \mathbb{N} \) and \( s \in R_n \), if for \((n, s)\) the algorithm stops, then the output is \((n, n!)]\). We shall now show \((R, S, P)\) to be a solution of CSE:

\[
F(R \cup P) S \cup Q = R \cup FPS = R \cup FR[n \in \mathbb{N}] (n, s) := \emptyset) S \\
= R \cup [n \neq \emptyset \land (n-1) \in \mathbb{N}] (n, s) := \emptyset) S \\
= R \cup [n \neq \emptyset \land (n-1) \in \mathbb{N}] (n, s) := \emptyset) S \\
= R \cup [n \neq \emptyset \land (n-1) \in \mathbb{N}] (n, s) := \emptyset) S \\
= [n \in \mathbb{N}] (n, s) := \emptyset) R.
\]

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

\[
R_1 = [n, s, B] (n, s) := (n, n!)]
\]

Consequently, we get

\[
I(n, s) \in B \quad (n, s) := (n, n!)] \\
= F([n, s, B] (n, s) := (n, n!)] S \cup Q \\
= [n = 0 \lor (n \neq 0 \land (n-1, s) \in B) \quad (n, s) := (n, n!)]
\]

and hence,

\[
B = \{0\} \times \mathbb{N} \cup \{(n, s) \neq 0 \land (n-1, s) \in B\}
\]

This implies

\[
(\forall s \in \mathbb{N}) \{0, s \in B\}, \\
(\forall (n, s) \in B \} (n+1, s) \in B),
\]

hence, \( B = \mathbb{N} \times \mathbb{N} \), contradicting the assumption. Finally, \((R, S, P)\) is the least solution of (6.4.1); thus, by Theorem 4.3.1, \( \text{Tail}(s_1) = R \), and by Corollary 6.2.5, \( \text{Perp}(s_1) = P \).

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

**Example 6.4.2. Procedure NINETEEN.**

**Begin**

\[
\begin{align}
\text{if } x > 100 & \text{ then } x := x - 10 \\
\text{else begin } \\
& x := x + 11; \\
& \text{NINETEEN;} \\
& \text{end;}
\end{align}
\]

**End**

We assume \( D = \mathbb{N} \) and we set:

- **Notation**
  - The algorithm
  - \( Q = [x > 100] \quad x := x - 10, \)
  - \( (s_1, Q, 0), \)
  - \( F = [x \leq 100] \quad x := x + 11, \)
  - \( (s_1, F, s_1, s_1), \)

  **The CSE is:**

\[
X = FXX \cup Q.
\]

**Our hypothesis is:**

\[
\text{Tail}(s_1) = [x \leq 100] \quad x := 91) \cup [x > 100] \quad x := x - 10, \\
\text{Perp}(s_1) = 0.
\]

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

\[
FRR \cup Q = ([x \leq 89] \quad x := 91) \cup [x > 100] \quad x := x - 10. \]

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 \( A \subseteq (-\infty, 100) \), \( A \neq (-\infty, 100) \), \( B \in (100, +\infty) \), \( B \neq (100, +\infty) \) and set

\[
R_1 = \{x \in A \quad x := 91) \cup \{x \in B \quad x := x - 10. \}
\]

be a solution of (6.4.2). We get therefore (we replace here \( R \) by \( R \))

\[
FR_1 R_1 \cup Q = \{[x \leq 100, x := 11 \in A \quad x := 91) \cup \{x \leq 100, x := x + 11 \in B \quad x := x + 11) \cup [x \leq 100, x := x + 11 \in A \quad x := 91) \cup \{x \leq 100, x := x + 11 \in B \quad x := x - 10. \}
\]

This implies

\[
(\forall x \in A) \quad x := 91) \cup \{x \in B \quad x := x - 10.
\]

14 Bemerkung 1. II
The last equality, of course, follows from the assumption that $R_1$ is a solution of CSE. We get from it

\[(6.4.3) \quad B = (100, +\infty),\]

and therefore,

\[\{x \leq 100, x + 11 \in B, x + 1 \in B\} : x := x - 9 = \{x = 100\} : x := 91\].

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

\[(6.4.4) \quad A = \{x \leq 100, x + 11 \in A, 91 \in A\} \cup\]

\[\cup \{x : 89 < x \leq 100, x + 1 \in A\} \cup\]

\[\cup \{100\}\]

Therefore

1) \(100 \in A\),

2) if \(90 < y \leq 100 \& y \in A\), then \(y - 1 \in A\).

In effect,

\[(6.4.5) \quad (89, 100) \subseteq A,\]

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

if \(y \leq 111\) and \(y \in A\) then \(y - 111 \in A\).

This, combined with (6.4.5), implies

\[(-\infty, 100) \subseteq A\]

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, \(R = \text{Tail}(a_1)\).

In consequence, \(\text{Stop}(a_1) = \text{Int}\), hence, by Lemma 6.2.4, \(\text{Loop}(a_1) = \emptyset\), i.e., \(\text{Perp}(a_1) = 0\). In this way our hypothesis is proved.

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

\begin{itemize}
  \item procedure GREATEST COMMON DIVISOR
  \begin{itemize}
    \item while \(n \neq m\) do if \(n > m\) then \(n := n - m\) else \(m := m - n\)
  \end{itemize}
\end{itemize}

Suppose we want to analyse this algorithm only in the case \(n\) is \(m\) range over positive integers, i.e., \(D = \mathbb{N}^* \times \mathbb{N}^*\). We set:

- **Notation**
  - The algorithm
    \[(a_1, [n = m], \ell),\]
    \[(a_2, [n \neq m], a_3),\]
  - \(F = [n > m] : n := n - m\),
  - \(G = [n < m] : m := m - n\).

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

\[(6.4.6) \quad X_1 = \{n \neq m\} X_1 \cup [n = m],\]
\[(6.4.6) \quad X_2 = (F \cup G) X_1,\]

Our hypothesis:

\[\text{Tail}(a_1) = \{(n, m) := (g(n, m), g(n, m))\}; \quad \text{Perp}(a_1) = 0,\]
\[\text{Tail}(a_2) = (F \cup G) \text{Tail}(a_3); \quad \text{Perp}(a_2) = 0,\]

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 \(\langle R, (F \cup G) R \rangle\) 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 \(g(n, m) = g(n - m, m)\),
- if \(n < m\) then \(g(n, m) = g(n, m - n)\).

Computes:

\[\{n \neq m\} (F \cup G) R \cup [n = m] = [n > m] \cup [n < m] \cup [n = m],\]
\[\{n \neq m\} (F \cup G) R \cup [n = m] = [n > m] \cup [n < m] \cup [n = m],\]
\[\{n \neq m\} (F \cup G) R \cup [n = m] = [n > m] \cup [n < m] \cup [n = m],\]
\[\{n \neq m\} (F \cup G) R \cup [n = m] = [n > m] \cup [n < m] \cup [n = m].\]

We shall now \(\text{Perp}(a_1) = 0\). Since this implies \(\text{Loop}(a_1) = \emptyset\), we shall have our hypothesis by Corollary 6.2.3. Solving the CSE (6.4.6) in a modified way to get the Run vector, we get:

\[\text{Run}(a_1) = \{(n \neq m) (F \cup G) R \cup [n = m] \cup (n \neq m) (F \cup G) R\} \cdot 0.\]

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

\[\text{Perp}(a_1) = \{(n \neq m) (F \cup G) R \cup [n = m] \cup (n \neq m) (F \cup G) R\} \cdot 0 = 0.\]

This is of course true by the assumption that \(D = \mathbb{N}^* \times \mathbb{N}^*\).

**References**


