MATHEMATICAL CENTRE TRACTS 63

J.W. DE BAKKER (ed.)

FOUNDATIONS OF COMPUTER SCIENCE

SECOND PRINTING

MATHEMATISCH CENTRUM AMSTERDAM 1975

AMS(MOS) subject classification scheme (1970): 68A05, 68A20, 68A50

ACM-Computing Reviews-categories: 3.70, 4.35, 5.21, 5.24, 5.25

ISBN 90 6196 111 4

First printing 1975

Second printing 1976

PREFACE

An Advanced Course on the Foundations of Computer Science organized by the Mathematical Centre as part of an international effort under the auspices of the European Communities, was held at the University of Amsterdam, May 20-31, 1974. This Tract collects the lecture notes of five of the courses given. The sixth course, given by Dr. R. Kowalski on Predicate Logic as a Programming Language in Artificial Intelligence, is scheduled to appear elsewhere.

We are very grateful to the Netherlands Organization for the Advancement of Pure Research (Z.W.O.) for generously supplying the money to organize the Course and to the lecturers for their excellent contributions.

J.W. de Bakker

Director of the Course

THE FIXED POINT APPROACH IN SEMANTICS: THEORY AND APPLICATIONS

by J.W. DE BAKKER

MATHEMATICAL CENTRE TRACTS 63, 1975, 3-53

THE FIXED POINT APPROACH IN SEMANTICS: THEORY AND APPLICATIONS

J.W. DE BAKKER

Mathematical Centre, Amsterdam / Free University, Amsterdam, (NL)

0. INTRODUCTION

The present notes are devoted to an exposition of the "fixed point approach" in programming theory. In particular, we are concerned with those programming concepts which are related to the control structure of a program, viz. recursion and iteration. The methods to be developed will be applied to obtain proofs of program properties and, also, to analyze program proving methods stemming from a variety of sources.

Let f: $\mathcal{D} \to \mathcal{D}$ be a function mapping some domain $\mathcal{D}$ to itself. An element $x \in \mathcal{D}$ is called a *fixed point* of f iff f(x) = x holds. This definition includes the case that $\mathcal{D}$ itself is a collection of functions, from E to F, say. Then, for F: $\mathcal{D} \to \mathcal{D}$ -such F which maps functions to functions we shall call a *functional*-, we have again that $f \in \mathcal{D}$ is a fixed point of F iff F(f) = f, i.e., iff for all $x \in E$, F(f)(x) = f(x).

The first three sections of our paper contain the development of the main mathematical properties of recursion (including iteration as a special case). We show that recursive procedures are *least* fixed points -under a suitable ordering- of the functionals to be associated with the body of their declarations (section 1). Next, the important notion of the *continuity* of these functionals is introduced, and a powerful proof rule (Scott's induction rule) is based on it (section 2). In section 3 we propose a method for associating binary relations with various programming constructs -composition, conditionals, while statements, parameterless recursive procedures- and show how to apply the results of sections 1 and 2 to them. This section also brings the introduction of the so-called μ-notation, together with its justification. This requires, among others, an extension of the continuity result of section 2.

The essential ideas of the first three sections were first presented

in [29], though the fixed point approach to recursion goes back to KLEENE (see e.g. [17], p.348). Independently of [29], PARK proposed his so-called "fixpoint induction" in [26], and BEKIC obtained a number of related results in [5]. The least fixed point operator has also been studied extensively by the Polish school of programming theory, for which we refer to [6,7] and the references contained in these papers.

Sections 4 and 5 bring a large number of applications of the results obtained in sections 1 to 3. We have drawn here mainly from the papers DE BAKKER [2], DE BAKKER and DE ROEVER [4], and DE BAKKER and MEERTENS [3], though a few remarks due to other authors are mentioned in section 4.4 and the exercises in section 6. (The advanced parts of these papers are usually omitted, however.)

There is, besides the results to be treated in these notes, a great variety of other applications to be found in the literature. A brief and incomplete survey follows: In a series of papers by MANNA and his colleagues [9,10,20,21,30], the problems dealing with recursive procedures with more than one parameter, and the various ways of "parameter passing" are investigated, and an impressive number of examples of Scott's induction are provided. We also mention their discussion of other induction principles which have been proposed such as "truncation induction" (MORRIS [25]) and "structural induction" (BURSTALL [8]). Decidability problems about recursive program schemes are treated e.g. in ASHCROFT, MANNA and PNUELI [1] and COURCELLE, KAHN and VUILLEMIN [12]. In an intriguing paper [15], HITCHCOCK and PARK investigate the relationship between the μ-formalism and second order predicate logic, and, moreover, the use of wellfounded relations in proofs of program termination. Here the notion of greatest fixed point makes its first appearance, without being explicitly mentioned, however. The least fixed point operator has also found its place in SCOTT's models of the λ-calculus, where furthermore the relationship with CURRY's "paradoxical combinator" Y is settled. (No published description of this seems to exist.) Finally, we mention the formal system embodying (a generalization of) the main ideas of section 1 to 3, viz. the LCF (Logic for Computable Functionals) system of MILNER, who has also implemented this yielding an interactive program proving facility [23,24].

We now give a summary of the applications we do treat in the present paper. Section 4 contains a number of examples concerning program equivalence. In section 4.1, a simple while statement example is dealt with, the proof of which uses some results with independent interest. Section 4.2 deals with a problem which initially had the appearance of an equivalence between two tree traversal algorithms, but which is then shown to be an instance of a much more general equivalence result, not depending on this specific area. In section 4.3 we study the well-known 91-function, and in section 4.4 we make a few remarks concerning a (still open) problem, viz. how to provide a convincing explanation of COOPER's extension of recursion induction as proposed in [11]. Section 5 deals at some length with the variety of attacks on proving (partial) program correctness, all going back to FLOYD's inductive assertion method [14]. After a brief review of this method in section 5.1, section 5.2 is devoted to a proof of its consistency and completeness. This may be seen as a generalization of MANNA's approach to partial correctnes [18]. In section 5.3 we discuss HOARE's axiomatic framework for proving program properties [16]. In particular, we justify his while statement axiom, and investigate in how far it fully characterizes such statements. In section 5.4, we try to find an interpretation for the ideas in a recent paper by DIJKSTRA [13]. We conclude that, if our interpretation is right, his main theorem is incorrect. A slight modification, however, is sufficient to yield a true proposition, which is proven by a two line application of Scott's induction. Finally, section 6 contains a small collection of not-too-simple exercises.

The general aim of our paper is to stimulate the reader's interest in the many new insights which have resulted from the fixed point approach. No results which essentially extend the literature as listed above, are given, though a few small remarks and points of presentation may be new. As said already, considerations which were thought to be of a too advanced nature have been omitted. Moreover, no attention has been paid to the fixed point approach in formal language theory, where a few fruitful applications have also been found.

1. RECURSIVE PROCEDURES AS LEAST FIXED POINTS

We shall present an analysis of the main mathematical properties of recursive procedures, leading to their characterization as least fixed points of certain transformations.

Let us first consider three simple examples of recursive procedures, written in ALGOL 60:

(1.1)
```
integer procedure f(x);
    f := if x = 0 then 1 else x*f(x-1)
```

(1.2)
```
integer procedure g(x);
    g := if x > 100 then x - 10 else g(g(x+11))
```

(1.3)
```
integer procedure h(x);
    h := h(x)
```

We see that (1.1) gives the well-known recursive definition of the factorial function, (1.2) is the remarkable 91-function, the name of which is derived from the fact that for g(x) we have, for all integer x, g(x) = if x > 100 then x - 10 else 91 (this is one of the results to be proven later (section 4.3)). As to the procedure declared by (1.3), this is immediately seen to lead to a non-terminating computation for each argument x, whence we see that the corresponding function is *nowhere defined*. Thus, as soon as we introduce this type of recursive definitions, we are directly led to the consideration of *partial* functions, i.e., functions which may be undefined for some (possibly all) of their arguments. For such functions, the following definition of a partial ordering "$\subseteq$" is rather natural. Also, we give a special name to the *least* function with respect to "$\subseteq$":

DEFINITION 1.1. Let f,g be partial functions from $\mathcal{D}_1$ to $\mathcal{D}_2$.

a. $f \subseteq g$ iff $\forall x \in \mathcal{D}_1, y \in \mathcal{D}_2[f(x) = y \rightarrow g(x) = y]$

b. Ω denotes the nowhere defined function (from $\mathcal{D}_1$ to $\mathcal{D}_2$).

Remarks:

1. One might emphasize in the notation that Ω refers to the sets $\mathcal{D}_1, \mathcal{D}_2$. This will turn out to serve no purpose and is therefore omitted.
2. Indication of the sets through which the variables, such as x,y above, range, will usually be omitted in the sequel.
3. Observe that $f \subseteq g$ iff for all x, whenever f is defined in x then g is defined in x and both functions yield the same value. Also, $f = g$ iff $f \subseteq g$ and $g \subseteq f$. Moreover, we clearly have for all f,g,h: $\Omega \subseteq f$; $f \subseteq f$; if $f \subseteq g$ and $g \subseteq h$ then $f \subseteq h$.

We now introduce a general format for the procedures as exemplified by (1.1) to (1.3). We shall use, instead of ALGOL 60, a shorter notation which for (1.1) reads: $f(x) \Leftarrow$ if $x = 0$ then 1 else $x*f(x-1)$, and similarly for (1.2) and (1.3). " $\Leftarrow$ " may be read as "is recursively defined by". In general we have declarations of the form

$$f(x) \Leftarrow \tau(f)(x), \tag{1.4}$$

where τ determines a *functional*, $\tau(f)$ a function, and $\tau(f)(x)$ yields an element of the domain $\mathcal{D}$ of values we are concerned with in the case at hand (e.g., in (1.1) $\mathcal{D}$ is the set of natural numbers, in (1.2) the set of integers, and in (1.3) any set). $\tau(f)(x)$ is constructed *) by applying certain rules of construction, to be given presently, to an initial system consisting of

---

*) A proper development of the theory would need the introduction of a formal language used for specifying the $\tau(f)$, and a system of interpretation (including suitable rules of computation) to associate functions with these formal constructs. In order to adhere to this distinction, we would require a treatment in a style which we consider to be unsuitable for the present introductory exposition. Thus, the experienced reader will have to forgive a few inconsistencies in the treatment we give her. A more rigorous development is given in [31].

- A given collection of base functions of $m \geq 0$ variables, the elements of which are denoted by $a, a_1, b, \ldots$ They map $\mathcal{D}^m$ to $\mathcal{D}$. (For m=0 they denote an element of $\mathcal{D}$ (a constant).)
- A given collection of predicates, of one variable, the elements of which are $p, q, \ldots$ These map $\mathcal{D}$ to $\{0,1\}$.
- The function f.

Before we can give these construction rules, we need to explain two notational conventions:

1. In the sequel we shall make a -modest- employ of the λ-notation. For any variable x, let $\phi(x)$ be some formula which possibly contains occurrences of x. Then $\lambda x \cdot \phi(x)$ is the function which maps its argument, a say, to the result of substituting a for all occurrences of x in $\phi(x)$. Example: Let $\phi(x)$ be the formula $x + 2*y$. Then $(\lambda y \cdot (\lambda x \cdot x+2*y)(3))(4) = (\lambda y \cdot 3+2*y)(4) = 3 + 2*4 = 11$, whereas $(\lambda x \cdot (\lambda y \cdot x+2*y)(3))(4) = (\lambda x \cdot x+2*3)(4) = 10$.
2. We use the following notation for composition of functions: Let b be any function of $m \geq 1$ variables, and let $a_1, a_2, \ldots, a_m$ be functions of one variable. Then we write $b \circ (a_1, \ldots, a_m)$ for the function defined by $(b \circ (a_1, \ldots, a_m))(x) = b(a_1(x), \ldots, a_m(x))$. For $b \circ (a)$ we usually write $b \circ a$.

DEFINITION 1.2 (Syntax for $\tau(f)$). $\tau(f)$ is either

a. A one-place base function
b. f
c. Constructed from already given $\tau_1(f)$, $\tau_2(f)$ by
   1. Composition: $\tau_1(f) \circ \tau_2(f)$
   2. Selection : $\lambda x \cdot$ if $p(x)$ then $\tau_1(f)(x)$ else $\tau_2(f)(x)$
d. Constructed from already given $\tau_1(f), \ldots, \tau_m(f)$ $(m \geq 1)$ and given m-place base function b yielding $b \circ (\tau_1(f), \ldots, \tau_m(f))$

Example 1. We show how to construct the right-hand side of (1.1) according to the rules of this definition. Choose for p, a_1, a_2, a_3 and b: $\lambda x \cdot x = 0$, $\lambda x \cdot 1$, $\lambda x \cdot x$, $\lambda x \cdot x-1$ and $\lambda x \lambda y.\ x*y$, respectively. Then $\tau(f) \equiv \lambda x \cdot$ if $p(x)$ then $\tau_1(f)(x)$ else $\tau_2(f)(x)$, with $\tau_1(f) \equiv a_1$, $\tau_2(f) \equiv b \circ (\tau_{21}(f), \tau_{22}(f))$, $\tau_{21}(f) \equiv a_2$, $\tau_{22}(f) \equiv \tau_{221}(f) \circ \tau_{222}(f)$, $\tau_{221}(f) \equiv f$, $\tau_{222}(f) \equiv a_3$.

Example 2. Next consider (1.2). Let p: $\lambda x \cdot x > 100$, a_1: $\lambda x \cdot x-10$, a_2: $\lambda x \cdot x+11$. Then $\tau(g) \equiv \lambda x \cdot$ if $p(x)$ then $\tau_1(g)(x)$ else $\tau_2(g)(x)$, with $\tau_1(g) \equiv a_1$, $\tau_2(g) \equiv \tau_{21}(g) \circ \tau_{22}(g)$, $\tau_{21}(g) \equiv g$, $\tau_{22}(g) \equiv \tau_{221}(g) \circ \tau_{222}(g)$, $\tau_{221}(g) \equiv g$, $\tau_{222}(g) \equiv a_2$.

Definition 1.2 gave the syntax of our formulas; the next definition gives their semantics. We show how to evaluate an arbitrary formula $\tau(f)$ in the presence of the declaration $f(x) \Leftarrow \tau_0(f)(x)$:

DEFINITION 1.3 (Semantics of $\tau(f)$). Let $f(x) \Leftarrow \tau_0(f)(x)$ be the declaration of f. The *value* of any term $\tau(f)$ is defined inductively by

a. If $\tau(f) \equiv a$ then $\tau(f)(x) = y$ if $a(x) = y$.

b. if $\tau(f) \equiv f$, then $\tau(f)(x) = y$ if $\tau_0(f)(x) = y$ (i.e., in order to evaluate $f(x)$, we replace f by the body of its declaration, viz. $\tau_0(f)$).

c. If $\tau(f) \equiv \tau_1(f) \circ \tau_2(f)$, then $\tau(f)(x) = y$ if there exists z such that $\tau_2(f)(x) = z$ and $\tau_1(f)(z) = y$.

d. If $\tau(f) \equiv \lambda x \cdot$ if $p(x)$ then $\tau_1(f)(x)$ else $\tau_2(f)(x)$, then $\tau(f)(x) = y$ if either $p(x) = 1$ and $\tau_1(f)(x) = y$, or $p(x) = 0$ and $\tau_2(f)(x) = y$.

e. If $\tau(f) \equiv b \circ (\tau_1(f), \ldots, \tau_m(f))$, then $\tau(f)(x) = y$ if there exist $x_1, \ldots, x_m$ such that $\tau_i(f)(x) = x_i$, $i=1,\ldots,m$ and $b(x_1, \ldots, x_m) = y$.

Example: We evaluate g(100) with respect to (1.2) (some shortcuts are used): $g(100) \overset{(b)}{=}$ if 100 > 100 then 100 - 10 else $g(g(100+11)) \overset{(d)}{=} g(g(111)) \overset{(c)}{=} g($if 111 > 100 then 111 - 10 else $g(g(100+11)) \overset{(d)}{=} g(101) \overset{(b)}{=}$ if 101 > 100 then 101 - 10 else $g(g(112) \overset{(d)}{=} 91$.

Our first result on the functionals τ concerns their monotonicity:

LEMMA 1.1. *Let* $\tau(f)$ *be as given in definition 1.2. Let* $\tau(f_i)$, $i=1,2$, *denote the result of substituting* f_i *for all occurrences of* f *in* $\tau(f)$. *Then we have: If* $f_1 \subseteq f_2$ *then* $\tau(f_1) \subseteq \tau(f_2)$.

PROOF. Induction on the complexity of τ.

a. If $\tau(f) \equiv a$ or $\tau(f) \equiv f$, trivial.

b. Let $\tau(f) \equiv \tau_1(f) \circ \tau_2(f)$. In order to prove $\tau(f_1) \subseteq \tau(f_2)$, we must show (definition 1.1): For all x,y, if $\tau(f_1)(x) = y$, then $\tau(f_2)(x) = y$. Assume $\tau(f_1)(x) = y$. By definition 1.3, there exists z such that

$\tau_2(f_1)(x) = z$ and $\tau_1(f_1)(z) = y$. By the induction hypothesis and the assumption we have that $\tau_2(f_2)(x) = z$ and $\tau_1(f_2)(z) = y$. Hence, $(\tau_1(f_2) \circ \tau_2(f_2))(x) = y$ follows, i.e., $\tau(f_2)(x) = y$ holds.
The remaining cases are also direct from the definitions. □

We now come to an important property of recursive procedures. First we define, for any formula $\tau(f)$, $\tau^i(\Omega)$ for any integer $i \geq 0$, as follows:

DEFINITION 1.4.

a. $\tau^0(\Omega) = \Omega$;

b. $\tau^{i+1}(\Omega)$ is the result of substituting $\tau^i(\Omega)$ for all occurrences of f in $\tau(f)$.

Example: For $\tau(f)$ as in (1.1), $\tau^2(\Omega)$ is $\lambda x \cdot$ if x = 0 then 1 else x*($\lambda x \cdot$ if x=0 then 1 else x*Ω(x-1))(x-1) which, using the notation $\lambda x \cdot \omega$ for Ω, yields after simplification:
$\lambda x \cdot$ if x = 0 then 1 else x* (if x-1=0 then 1 else ω).

We have the following lemma:

LEMMA 1.2. *Let $\tau(f)$ be a formula, with f declared by $f(x) \Leftarrow \tau_0(f)(x)$. If $\tau(f)(x) = y$, then there exists an integer i($\geq$0, and in general depending on x), such that $\tau(\tau_0^i(\Omega))(x) = y$.*

PROOF. With each evaluation $\tau(f)(x) = y$ we associate the pair (N,γ), where N is the number of applications of step b of definition 1.3 in the evaluation of $\tau(f)(x) = y$, and γ is the complexity of the formula $\tau(f)$. We define $(N_1,\gamma_1) < (N_2,\gamma_2)$ iff either $N_1 < N_2$, or $N_1 = N_2$ and $\gamma_1 < \gamma_2$. (A formal definition of γ is left to the reader.) We prove the lemma by induction on (N,γ). Let $\tau(f)(x) = y$ have as associated pair (N,γ), and let the assertion of the lemma be proved for all evaluations with $(N',\gamma') < (N,\gamma)$. We distinguish the following cases:

a. $\tau(f) \equiv a$. Then we may take i = 0.

b. $\tau(f) \equiv f$. Then, according to step b in the definition of the evaluation of $f(x) = y$, we have that $\tau_0(f)(x) = y$, with associated pair

$(N-1,\gamma')$. Hence, by the induction hypothesis, there exists i_0 such that $\tau_0(\tau_0^{i_0}(\Omega))(x) = y$, or $\tau_0^{i_0+1}(\Omega)(x) = y$; thus, taking $i = i_0 + 1$ yields the desired result, since, for $\tau(f) \equiv f$, $\tau(\tau_0^{i_0+1}(\Omega))(x) = \tau_0^{i_0+1}(\Omega)(x) = y$.

c. $\tau(f) = \tau_1(f) \circ \tau_2(f)$. There exists z such that $\tau_2(f)(x) = z$, with associated pair (N_1,γ_1), $\tau_1(f)(z) = y$, with associated pair (N_2,γ_2), where $N_1,N_2 \leq N$, $\gamma_1,\gamma_2 < \gamma$. Thus, by induction, there exist i_1,i_2 such that $\tau_2(\tau_0^{i_2}(\Omega))(x) = z$, $\tau_1(\tau_0^{i_1}(\Omega))(z) = y$. Let $i = \max(i_1,i_2)$. Then, by monotonicity, $\tau_2(\tau_0^{i}(\Omega))(x) = z, \tau_1(\tau_0^{i}(\Omega))(z) = y$, hence $\tau(\tau_0^{i}(\Omega))(x) = y$ follows.

d. The remaining two cases follow similarly by induction and monotonicity. □

Lemma 1.2 may be reformulated using the following notation: Let

(1.5) $$f_0 \subseteq f_1 \subseteq \ldots \subseteq f_i \subseteq \ldots$$

be a *chain* of functions such that $f_i \subseteq f_{i+1}$, i=0,1,... . Each such chain has a least upper bound (l.u.b.) denoted by $U_{i=0}^{\infty} f_i$, and defined as follows

DEFINITION 1.5. Let f_i,i=0,1,... be as in (1.5). The function $U_{i=0}^{\infty} f_i$ is defined by: $(U_{i=0}^{\infty} f_i)(x) = y$ iff for some $i \geq 0$, we have $f_i(x) = y$.

It is easily checked that $U_{i=0}^{\infty} f_i$ is indeed a function which satisfies the l.u.b. requirements:

1) $f_i \subseteq U_{i=0}^{\infty} f_i$, for all i. 2) If $f_i \subseteq g$ for all i, then $U_{i=0}^{\infty} f_i \subseteq g$.

Using this notation, from lemma 1.2 we obtain

LEMMA 1.3. *Let* f *be declared by* $f(x) \Leftarrow \tau_0(f)(x)$. *Then* $f \subseteq U_{i=0}^{\infty} \tau_0^i(\Omega)$.

PROOF. By definition 1.5, $U_{i=0}^{\infty} \tau_0^i(\Omega)$, which is the l.u.b. of the chain $\Omega \subseteq \tau_0(\Omega) \subseteq \ldots \subseteq \tau_0^i(\Omega) \subseteq \ldots$, is defined as $(U_{i=0}^{\infty}\tau_0^i(\Omega))(x) = y$ iff $\tau_0^i(\Omega)(x) = y$ for some i. Now apply lemma 1.2 with $\tau(f) \equiv f$. □

As next step we observe that from definition 1.3, part b, we immediately obtain

LEMMA 1.4. *Let* $f(x) \Leftarrow \tau(f)(x)$. *Then* $f(x) = y$ *iff* $\tau(f)(x) = y$, i.e., $f = \tau(f)$.

In other words, we have that f is a *fixed point* of the functional τ of the body of its declaration.

We need one more lemma for the proof of our first theorem:

LEMMA 1.5. *Let* $f(x) \Leftarrow \tau(f)(x)$, *and let* g *be any function satisfying* $\tau(g) \subseteq g$. *Then* $\bigcup_{i=0}^{\infty} \tau^i(\Omega) \subseteq g$.

PROOF. By the definition of $\bigcup_{i=0}^{\infty} \tau^i(\Omega)$, it is sufficient to show $\tau^i(\Omega) \subseteq g$, for all i. We use induction on i.

a. i = 0. Clearly, $\Omega \subseteq g$.

b. Assume the result for i: $\tau^i(\Omega) \subseteq g$. Then, by monotonicity and the assumption on g, $\tau^{i+1}(\Omega) \subseteq \tau(\tau^i(\Omega)) \subseteq \tau(g) \subseteq g$.

THEOREM 1.1. *Let* $f(x) \Leftarrow \tau(f)(x)$.

a. $f = \bigcup_{i=0}^{\infty} \tau^i(\Omega)$

b. *Let* g *be any function satisfying* $\tau(g) \subseteq g$.
Then $f \subseteq g$.

c. f *is the least fixed point of* τ.

PROOF.

a. $\subseteq$ is lemma 1.3. By lemma 1.4, f is a fixed point of τ, thus, à fortiori, $\tau(f) \subseteq f$. Now apply lemma 1.5.

b. Direct from lemma 1.5 and part a.

c. Direct from lemma 1.4 and part b. □

With this important theorem we conclude our first section.

2. CONTINUITY AND SCOTT's INDUCTION RULE

According to theorem 1.1, for f declared by $f(x) \Leftarrow \tau(f)(x)$, we have $f = \bigcup_{i=0}^{\infty} \tau^i(\Omega)$. This result (which may be viewed as a technique for "successively approximating" a recursive procedure) is applied in the justi-

fication of a powerful rule for proving properties of programs with recursive procedures, viz. Scott's induction rule. Before presenting it, we need another important idea. We introduce the notion of *continuity* of our functionals, as defined in

DEFINITION 2.1. $\tau(f)$ is called continuous in f iff for each chain $g_0 \subseteq g_1 \subseteq \ldots \subseteq g_i \subseteq \ldots$ the following holds:

$$\tau(\bigcup_{i=0}^{\infty} g_i) = \bigcup_{i=0}^{\infty} \tau(g_i)$$

THEOREM 2.1. *Let* $\tau(f)$ *be as in definition 1.2. Then* $\tau(f)$ *is continuous in* f.

PROOF. Induction on the complexity of τ.

a. If $\tau(f) \equiv a$ or $\tau(f) \equiv f$, the proof is trivial.

b. Let $\tau(f) \equiv \tau_1(f) \circ \tau_2(f)$. We have to show that

$$\tau_1(\bigcup_i g_i) \circ \tau_2(\bigcup_j g_j) = \bigcup_k (\tau_1(g_k) \circ \tau_2(g_k))$$

This is established by:

$$\tau_1(\bigcup_i g_i) \circ \tau_2(\bigcup_j g_j) = \text{(ind. hypothesis)}$$

$$\bigcup_i \tau_1(g_i) \circ \bigcup_j \tau_2(g_j) = \bigcup_{i\,j} (\tau_1(g_i) \circ \tau_2(g_j)) = \text{(monotonicity)}$$

$$\bigcup_{i\,j} (\tau_1(g_{\max(i,j)}) \circ \tau_2(g_{\max(i,j)})) = \bigcup_k (\tau_1(g_k) \circ \tau_2(g_k))$$

c. Let $\tau(f) \equiv b \circ (\tau_1(f), \ldots, \tau_m(f))$. Then

$$\tau(\bigcup_i g_i) = b \circ (\tau_1(\bigcup_i g_i), \ldots, \tau_m(\bigcup_i g_i)) = \text{(ind.hyp.)}$$

$$b \circ (\bigcup_i \tau_1(g_i), \ldots, \bigcup_i \tau_m(g_i)) =$$

$$b \circ (\bigcup_{i_1} \ldots \bigcup_{i_m} (\tau_1(g_{i_1}), \ldots, \tau_m(g_{i_m}))) = (k=\max(i_1,\ldots,i_m))$$

$$b \circ (\bigcup_k (\tau_1(g_k), \ldots, \tau_m(g_k))) =$$

$$\bigcup_k (b \circ (\tau_1(g_k), \ldots, \tau_m(g_k))) = \bigcup_k \tau(g_k)$$

d. Similar to part b or c. □

The continuity of the $\tau(f)$ plays an essential role in the justification of Scott's proof rule, as given in the next theorem:

THEOREM 2.2. (Scott's induction rule, simple case). *Let* $\tau_1(f)$, $\tau_2(f)$, $\tau(f)$ *be as in definition 1.2, with* f *declared by* $f(x) \Leftarrow \tau(f)(x)$. *Assume that the following two conditions are satisfied:*

a. $\tau_1(\Omega) \subseteq \tau_2(\Omega)$.

b. *For any function* X, *if* $\tau_1(X) \subseteq \tau_2(X)$ *then* $\tau_1(\tau(X)) \subseteq \tau_2(\tau(X))$.

Then we may conclude that

c. $\tau_1(f) \subseteq \tau_2(f)$.

PROOF. First we show that $\tau_1(\tau^i(\Omega)) \subseteq \tau_2(\tau^i(\Omega))$, for all i, by induction on i.

1. i = 0. Follows from assumption a of the theorem.
2. Assume $\tau_1(\tau^i(\Omega)) \subseteq \tau_2(\tau^i(\Omega))$. Apply assumption b of the theorem with $X = \tau^i(\Omega)$. Then $\tau(X) = \tau^{i+1}(\Omega)$, and we obtain $\tau_1(\tau^{i+1}(\Omega)) \subseteq \tau_2(\tau^{i+1}(\Omega))$, as desired.

From $\tau_1(\tau^i(\Omega)) \subseteq \tau_2(\tau^i(\Omega))$, for all i, we infer that $U_{i=0}^{\infty} \tau_1(\tau^i(\Omega)) \subseteq U_{i=0}^{\infty} \tau_2(\tau^i(\Omega))$. By the continuity theorem, from this we obtain $\tau_1(U_{i=0}^{\infty}\tau^i(\Omega)) \subseteq \tau_2(U_{i=0}^{\infty}\tau^i(\Omega))$. An application of theorem 1.1 then yields $\tau_1(f) \subseteq \tau_2(f)$, as was to be shown. □

COROLLARY 2.2. (Scott's induction rule, general case).

Let $\tau(f)$, $\tau_{1i}(f)$, $\tau_{2i}(f)$, $i \in I$ *(*I *any index set) be as in definition 1.2, and let* $f(x) \Leftarrow \tau(f)(x)$ *be the declaration of* f. *Let* $\Phi(f)$ *be the family of inclusions* $\{\tau_{1i}(f) \subseteq \tau_{2i}(f)\}_{i\in I}$. *Assume that the following two conditions are satisfied:*

a. $\Phi(\Omega)$ *holds.*

b. *For any function* X, *if* $\Phi(X)$ *holds, then* $\Phi(\tau(X))$ *holds.*

Then we may conclude that

c. $\Phi(f)$ *holds.*

PROOF. A direct generalization of theorem 2.2. □

Before exhibiting our first examples of applying the rule, we generalize our results for *systems* of $n \geq 2$ simultaneously declared recursive procedures. We consider the system of declarations

$$(2.1) \qquad \begin{cases} f_1(x) \Leftarrow \tau_1(f_1,\ldots,f_n)(x) \\ \\ f_n(x) \Leftarrow \tau_n(f_1,\ldots,f_n)(x) \end{cases}$$

or, in shorter notation,

$$\{f_i(x) \Leftarrow \tau_i(f_1,\ldots,f_n)(x)\}_{i=1,\ldots,n}.$$

First we have to extend definition 1.2, where in clause b we now allow any of the functions f_i, instead of the single f. The obvious extension of definition 1.3 is left to the reader. Next, we generalize theorem 1.1 as follows: *For any* $\tau = \tau(f_1,\ldots,f_n)$, *we define the approximations* $\tau^{(i)}$, *with respect to (2.1), as follows:*

1. $\tau^{(0)} = \Omega$
2. $\tau^{(i+1)} = \tau\circ(\tau_1^{(i)},\ldots,\tau_n^{(i)})$

It can then be shown that

1. $\tau = \bigcup_{i=0}^{\infty} \tau^{(i)}$
2. *For each system* $g_1,\ldots,g_n$ *such that* $\{\tau_i(g_1,\ldots,g_n) \subseteq g_i\}_{i=1,\ldots,n}$ *we have* $\{f_i \subseteq g_i\}_{i=1,\ldots,n}$.
3. *The system* $\{f_1,\ldots,f_n\}$ *is the least (simultaneous) fixed point of the* $\{\tau_1,\ldots,\tau_n\}$.

Theorem 2.1 also has a straightforward generalization, statement of which is omitted here. Scott's rule (simple case) for systems becomes: *Assume, for* $\tau' = \tau'(f_1,\ldots,f_n)$, $\tau'' = \tau''(f_1,\ldots,f_n)$,

a. $\tau'(\Omega,\ldots,\Omega) \subseteq \tau''(\Omega,\ldots,\Omega)$.

b. *For all functions* $X_1,\ldots,X_n$, *if* $\tau'(X_1,\ldots,X_n) \subseteq \tau''(X_1,\ldots,X_n)$ *then* $\tau'(\tau_1(X_1,\ldots,X_n),\ldots,\tau_n(X_1,\ldots,X_n)) \subseteq \tau''(\tau_1(X_1,\ldots,X_n),\ldots,\tau_n(X_1,\ldots,X_n))$.

Then we may conclude that

c. $\tau'(f_1,\ldots,f_n) \subseteq \tau''(f_1,\ldots,f_n)$.

Example 1. As first example we consider the system

$$
\begin{aligned}
&f_1(x) \Leftarrow \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ x\ \underline{\text{else}}\ f_1(a_1(x))\\
&\\
&f_2(x) \Leftarrow \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ a_2(x)\ \underline{\text{else}}\ f_2(a_1(x)).
\end{aligned}
\tag{2.2}
$$

We show that $a_2\circ f_1 = f_2$, by applying Scott's induction rule twice, with $\tau_1(f_1,f_2)$ and $\tau_2(f_1,f_2)$ as in (2.2), and $\tau'(f_1,f_2) \equiv a_2\circ f_1$, $\tau''(f_1,f_2) \equiv f_2$.

1. $\tau'(f_1,f_2) \subseteq \tau''(f_1,f_2)$. We have to verify
   a. $a_2\circ\Omega \subseteq \Omega$. This is clear.
   b. Assume $a_2\circ X_1 \subseteq X_2$. We show that $a_2\circ\tau_1(X_1,X_2) \subseteq \tau_2(X_1,X_2)$:
      $a_2\circ\tau_1(X_1,X_2) =$ (df.τ_1)
      $a_2\circ(\lambda x.\ \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ x\ \underline{\text{else}}\ (X_1\circ a_1)(x)) =$
      $\lambda x.\ \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ a_2(x)\ \underline{\text{else}}\ (a_2\circ X_1\circ a_1) \subseteq$ (ind.)

$\lambda x\cdot$ if $p(x)$ then $a_2(x)$ else $(X_2 \circ a_1)(x) = (df.\tau_2)$

$\tau_2(X_1, X_2)$.

2. $\tau''(f_1, f_2) \subseteq \tau'(f_1, f_2)$.

   a. $\Omega \subseteq a_2 \circ \Omega$ is clear.

   b. Assume $X_2 \subseteq a_2 \circ X_1$.

   $\tau_2(X_1, X_2) = (df.\tau_2)$

   $\lambda x\cdot$ if $p(x)$ then $a_2(x)$ else $(X_2 \circ a_1)(x) \subseteq$ (ind.)

   $\lambda x\cdot$ if $p(x)$ then $a_2(x)$ else $(a_2 \circ X_1 \circ a_1)(x) =$

   $a_2 \circ (\lambda x\cdot$ if $p(x)$ then x else $(X_1 \circ a_1)(x) = (df.\tau_1)$

   $a_2 \circ \tau_1(X_1, X_2)$.

Example 2. Let $f(x) \Leftarrow \tau(f)(x)$, with $\tau(f) \equiv \lambda x\cdot$ if $p(x)$ then x else $f(f(a(x)))$. We show that $f \circ f = f$. In this case we do not need to prove two inclusions, but we apply a slightly modified version of Scott's rule (the formulation of which is left to the reader) to prove directly the following equivalence: Let g be any function satisfying $g = \lambda x\cdot$ if $p(x)$ then x else $g(g(a(x)))$. Let $\tau_1(f) \equiv g \circ f$, $\tau_2(f) \equiv f$. We show that $\tau_1(f) = \tau_2(f)$ by establishing

a. $\tau_1(\Omega) = \tau_2(\Omega)$, i.e., $g \circ \Omega = \Omega$. This is clear.

b. Assume $\tau_1(X) = \tau_2(X)$, i.e., $g \circ X = X$. We have $\tau_1(\tau(X)) =$

   $g \circ (\lambda x\cdot$ if $p(x)$ then x else $X(X(a(x)))) =$

   $\lambda x\cdot$ if $p(x)$ then $g(x)$ else $(g \circ X \circ X \circ a)(x) =$ (ind.ass.)

   $\lambda x\cdot$ if $p(x)$ then $g(x)$ else $(X \circ X \circ a)(x) =$ (ass. on g)

   $\lambda x\cdot$ if $p(x)$ then (if $p(x)$ then x else $g(g(a(x)))$

   else $(X \circ X \circ a)(x) =$

   $\lambda x\cdot$ if $p(x)$ then x else $(X \circ X \circ a)(x) =$

   $\tau_2(\tau(X))$.

From a and b, $g \circ f = f$ follows. Since this has been proved for arbitrary g satisfying $g = \tau(g)$, we have in particular that $f \circ f = f$.

Remarks.

1. In the sequel, when justification of the "Ω-case" of Scott's rule is trivial -which it usually is- no explicit mention will be made of it.

2. The introduction of g in example 2 was needed in order to avoid dealing with $\tau_2(f) \equiv f \circ f$, which would lead to undesirable complications in proving that from $X = X \circ X$, $\tau(X) = \tau(X) \circ \tau(X)$ may be inferred. By similar techniques it is always possible to avoid, if necessary, simultaneous

induction on *all* occurrences of f. Again, no explicit attention will be paid to this in the sequel. (It is even the case that with a slightly different formal presentation of Scott's rule, we would not have encountered this difficulty at this place at all).

Before dealing with our third example, which is concerned with functions of *two* variables, we have to make some comments on the extension of our theory to functions of more than one variable. The syntactic part of this, which amounts to another extension of definition 1.2, is not difficult. However, some complications arise when we want to extend the evaluation rules. Consider e.g. the declaration f(x,y) ⇐ if x = 0 then 0 else f(x-1,f(x,y)), and suppose we want to evaluate f(1,0). Since the first argument is ≠ 0, we replace f(1,0) by f(0,f(1,0)). We now have a choice between replacing the inner- or the outermost f. If a computation rule is chosen which prescribes that the innermost occurrence is always to be dealt with first, then the computation does not terminate, whereas (consistently) choosing the outermost f yields the value 0. Thus we see that different computation rules may lead to different results. The problems connected with this phenomenon have been investigated extensively in papers by CADIOU, MANNA & VUILLEMIN [9,10,21,30]. They have reached the conclusion that the function determined by a recursive procedure of the form f(x,y) ⇐ τ(f)(x,y), say, is not necessarily the least fixed point of the functional τ. However, this seems to be explained by their failure to realize that different parameter mechanisms lead to, possibly, different functionals which may have different least fixed points. We feel that when such distinction between the functionals *is* made, then the least fixed point result remains true. We do not have a complete proof of this,*) but our claim finds strong support in as yet unpublished work of NIVAT, who also finds no deviation from the least fixed point characterization. These matters will not be dealt with any further in our notes: From now on we assume that all functions defined by a recursive procedure declaration satisfy the least fixed point property. The sceptical reader may, if he so wishes, impose the restriction that a computation rule is chosen -such as the equivalent of call-by-name- which justifies this also in the eyes of MANNA c.s.

---

*) (Added in proof) A complete proof is given in DE BAKKER [31].

Example 3 (MORRIS [25]). Let

$$f(x,y) \Leftarrow \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ y\ \underline{\text{else}}\ h(f(k(x),y))$$
$$g(x,y) \Leftarrow \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ y\ \underline{\text{else}}\ g(k(x),h(y))$$

We show that f = g, by applying Scott's induction (general case) to the two equalities: f = g, and $\lambda x\lambda y\cdot f(x,h(y)) = \lambda x\lambda y\cdot h(f(x,y))$. The "Ω-case" is left to the reader. Next assume as induction hypothesis: $X_1 = X_2$, *and* $\lambda x\lambda y\cdot X_1(x,h(y)) = \lambda x\lambda y\cdot h(X_1(x,y))$. We have

$\lambda x\lambda y\cdot \tau_1(X_1,X_2)(x,y)$ = (df.f)
$\lambda x\lambda y\cdot$ if p(x) then y else $h(X_1(k(x),y))$ = (second ind.hyp.)
$\lambda x\lambda y\cdot$ if p(x) then y else $X_1(k(x),h(y))$ = (first ind.hyp.)
$\lambda x\lambda y\cdot$ if p(x) then y else $X_2(k(x),h(y))$ = (df.g)
$\lambda x\lambda y\cdot \tau_2(X_1,X_2)(x,y)$.

Also

$\lambda x\lambda y\cdot \tau_1(X_1,X_2)(x,h(y))$ = (df.f)
$\lambda x\lambda y\cdot$ if p(x) then h(y) else $h(X_1(k(x),h(y)))$ = (second ind.hyp.)
$\lambda x\lambda y\cdot$ if p(x) then h(y) else $h(h(X_1(k(x),y)))$ =
$\lambda x\lambda y\cdot h$ (if p(x) then y else $h(X_1(k(x),y)))$ = (df.f)
$\lambda x\lambda y\cdot h(\tau_1(X_1,X_2)(x,y))$.

Example 4. Recursion induction and fixed point induction.
The first technique for proving equivalence of recursive procedures was the so-called method of recursion induction, due to McCARTHY [22]. This works as follows: In order to prove the equivalence $f_1 = f_2$, one tries to find some *total* f, recursively defined by $f(x) \Leftarrow \tau(f)(x)$, such that τ satisfies $\tau(f_1) = f_1$ and $\tau(f_2) = f_2$. If such f is found, one may infer that $f_1 = f_2$. The method is easily explained using theorem 1.1, part b: From this we obtain that, under the given assumptions, $f \subseteq f_1$ and $f \subseteq f_2$. The requirement that f be total then immediately yields $(f=)f_1 = f_2$. A variant of recursion induction is PARK's fixed point induction: Let f be declared by $f(x) \Leftarrow \tau(f)(x)$, and assume we want to show $f \subseteq g$ for some g. Then, again according to theorem 1.1, part b, it is sufficient to show $\tau(g) \subseteq g$, and this is the method called fixed point induction by PARK in [26]. It is an interesting, and as yet open, question to determine precisely which class of inclusions can be shown by fixed point induction, and which class needs the additional power of Scott's induction. Observe that fixed point induction uses only the monotonicity of the τ's, whereas Scott's induction in

the form presented here uses in addition their continuity. To what extent this requirement may be weakened has been investigated in [15]. (cf. also exercise 6.3.)

This concludes our first series of examples of applying Scott's rule. Many more will follow in sections 4 and 5.

3. PROGRAMS AND RELATIONS

We now introduce a way of looking at programs which will enable us to apply the techniques of the previous sections to obtain both proofs of a variety of program properties, as well as an insight in the methods proposed for deriving such proofs, such as FLOYD's inductive assertion method, and its reformulations and extensions as proposed by MANNA, HOARE and DIJKSTRA.

The first idea is the conception of a program as a specification of a *mapping* between *states*. When a program P with initial state x prescribes a computation resulting in final state y, we say that $y = P(x)$. It is convenient to allow also non-deterministic programs. Therefore, we consider P as a *binary relation*, and now write xPy to indicate that initial state x is transformed by P to final state y. Note that xPy_1 and xPy_2, with $y_1 \neq y_2$, are possible.

Next, we indicate how certain important programming concepts are modelled in such a relational framework. We start with a class of elementary actions, $A, A_1, A_2, \ldots$, each of which determines in some way we do not care to analyze further a relation between states. (If the reader insists, he may take assignment statements as examples of such elementary actions. The description of their effect will then need the introduction of the state components, corresponding to the variables manipulated by the program. An assignment changes one such component, and leaves the others invariant. However, this level of detail will not enter our considerations.) Starting with elementary actions, more complex programs are constructed by means of the go-on operator (" ; ") which prescribes sequential execution, the conditional-, and the while-statement. Before we deal with these, we introduce some further tools. Let $\mathcal{D}$ be the domain of states, and let $R, R_1, \ldots, S, \ldots$ be binary relations over $\mathcal{D}$, i.e., subsets of $\mathcal{D} \times \mathcal{D}$. As operations between relations we have:

a. Binary operations. *Composition*: $R_1;R_2 =$ $\{(x,y) \mid \exists z[xR_1z \text{ and } zR_2y]\}$. *Union*: $R_1 \cup R_2 =$ $\{(x,y) \mid xR_1y \vee xR_2y\}$. *Intersection*: $R_1 \cap R_2 = \{(x,y) \mid xR_1y \wedge xR_2y\}$.

b. Unary operation. *Conversion*: $\breve{R} = \{(x,y) \mid yRx\}$.

c. Nullary operations. The *empty* relation (i.e. the empty subset of $\mathcal{D} \times \mathcal{D}$) is again denoted by Ω. The *identity* relation I is defined as $I = \{(x,x) \mid x \in \mathcal{D}\}$. The *universal* relation is defined as $U = \mathcal{D} \times \mathcal{D}$.

d. The *star* operation: $R^* = I \cup R \cup (R;R) \cup \ldots = \bigcup_{i=0}^{\infty} R^i$.

We apply these operations in our modelling of programming concepts. Sequencing is easy: If the program $S_1;S_2$ maps initial x to final y, then there must be an intermediate state z with xS_1z and zS_2y. Thus, sequential execution of programs corresponds directly to relational composition. For the treatment of conditionals, we need a new convention. Consider a statement <u>if</u> p <u>then</u> S_1 <u>else</u> S_2. Whereas S_1 and S_2 may be considered as binary relations, this is not the case with the predicate p [1], since it maps states to {0,1}, say. Therefore, we use the following device: With the predicate p we associate *two* relations, p [2] and $\bar{p}$ both of which are subsets of the identity relation I, and defined by

$$p = \{(x,x) \mid p(x) = 1\}$$
$$\bar{p} = \{(x,x) \mid p(x) = 0\}.$$

Observe that these definitions imply that $p \cap \bar{p} = \Omega$. We can now give the relation corresponding to the statement <u>if</u> p <u>then</u> S_1 <u>else</u> S_2:

(3.1) $\qquad (p;S_1) \cup (\bar{p};S_2)$.

We shall adopt the convention that " ; " binds stronger than "$\cup$", and omit parentheses such as in (3.1) from now on.

---

1) Not all terminologies agree on our use of the words relation and predicate. Therefore, we emphasize that in our paper a (binary) relation is a subset of $\mathcal{D} \times \mathcal{D}$ (elsewhere possibly identified with a two-place predicate), and a predicate is a mapping from $\mathcal{D}$ to {0,1} (elsewhere maybe called a one-place predicate).

2) Using p to denote both a predicate and a relation is admittedly ambiguous, but the reader will soon get used to the transition between the two, and, hopefully, will eventually appreciate its advantages.

As an example of applying the notation, we look at one of McCARTHY's axioms for conditionals [22] which states the equivalence of *if* p *then* (*if* p *then* S_1 *else* S_2) *else* S_3, and *if* p *then* S_1 *else* S_3. As corresponding relations we have $p;(p;S_1 \cup \bar{p};S_2) \cup \bar{p};S_3$, and $p;S_1 \cup \bar{p};S_3$, respectively. The first of these may be simplified by applying some obvious properties of the relational system such as

- distributivity of ";" over "∪", i.e.,
  $R_1;(R_2 \cup R_3) = R_1;R_2 \cup R_1;R_3$
- for $p,q \subseteq I$ we have $p;q = p \cap q$
- $\Omega;R = R;\Omega = \Omega$
- $\Omega \cup S = S \cup \Omega = S$

etc. (Simple properties such as these will be applied tacitly in the sequel.) We now obtain $p;(p;S_1 \cup \bar{p};S_2) \cup \bar{p};S_3 = p;p;S_1 \cup p;\bar{p};S_2 \cup p;S_3 =$
$= p;S_1 \cup \Omega;S_2 \cup \bar{p};S_3 = p;S_1 \cup \Omega \cup \bar{p};S_3 = p;S_1 \cup \bar{p};S_3$, as was to be shown.

The next programming construct we treat is simple *iteration*, in the form of the while statement *while* p *do* S, with the usual semantics: iterate S as long as p remains true (including the case "do nothing" (I) if p is false to begin with). As corresponding relation we have

$$(3.2) \qquad (p;S)^* ; \bar{p}$$

for which we shall also use the notation p*S. On the base of (3.2) we can prove simple properties such as 1. $p*S = p;S;p*S \cup \bar{p}$. 2. $p*(p*S) = p*S$. 3. $(p_1 \vee p_2)*S = p_1*S;p_2*(S;p_1*S)$, etc. We give the proof of the first two, leaving the third to the reader (see section 4.1, however):

1. $p*S = (p;S)^*;\bar{p} = ((p;S);(p;S)^* \cup I);\bar{p} = p;S;(p;S)^*;\bar{p} \cup I;\bar{p} =$
   $= p;S;p*S \cup \bar{p}$.
2. $p*(p*S) = (p;(p*S))^*;\bar{p} = (p;(p;S)^*;\bar{p})^*;\bar{p} =$
   $= (I \cup p;(p;S)^*;\bar{p} \cup p;(p;S)^*;\bar{p};p;(p;S)^*;\bar{p} \cup \ldots);\bar{p} =$
   $= (I \cup p;(p;S)^*;\bar{p} \cup \Omega \cup \ldots);\bar{p} =$
   $= \bar{p} \cup p;(p;S)^*;\bar{p};\bar{p} = \bar{p} \cup (p;S)^*;\bar{p} = (p;S)^*;\bar{p} = p*S.$

Manipulations with relations as just exhibited are rather elementary, and one would hope for a more powerful method of dealing with such simple properties. This is achieved by the extension of our relational approach to recursive procedures, making available the tools of sections 1 and 2. Consider a procedure declaration of the form

(3.3) $\underline{\text{procedure}}\ P;T(P)$

where T(P), the procedure body, is a statement of one of the forms discussed above, and made up by means of composition and selection from the elementary actions, and, possibly, P itself. e.g., we might have as instances of (3.3)

(3.4)
$$\underline{\text{procedure}}\ P_1;\ p;A;P_1 \cup \bar{p}$$
$$\underline{\text{procedure}}\ P_2;\ p;A_1;P_2;A_2 \cup \bar{p};A_3$$

etc. (Note that a call of P_1 has the same effect as performing p*A). A point of possible confusion should be mentioned here. The procedures we just introduced are parameterless, whereas the recursive procedures of sections 1 and 2 were assumed to have $n \geq 1$ arguments. This is explained by considering the procedures as in (3.3) or (3.4) as having the state as only, suppressed, parameter. In the notation of section 1, we would write for (3.4):

$$f_1(x) \Leftarrow \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ f_1(a(x))\ \underline{\text{else}}\ x$$
$$f_2(x) \Leftarrow \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ a_2(f_2(a_1(x)))\ \underline{\text{else}}\ a_3(x).$$

Thus, we see that there is a direct transliteration between the parameterless recursive procedures of this section, and the one-parameter recursive procedures of sections 1 and 2. This is an important insight, since we can now immediately apply the fundamental results of these sections, which only need slight notational reformulation. We have:

THEOREM 3.1. *Let* $P \Leftarrow T(P)$ *be the declaration of a recursive procedure.*

a. (*Monotonicity*) *If* $X_1 \subseteq X_2$ *then* $T(X_1) \subseteq T(X_2)$.

b. (*Fixed point property*) $P = T(P)$.

c. $P = \bigcup_{i=0}^{\infty} T^i(\Omega)$.

d. (*Fixed point induction*) *If* $T(Q) \subseteq Q$ *then* $P \subseteq Q$.

e. (*Least fixed point property*) $P = \bigcap \{X : X = T(X)\}$.

f. (*Continuity*) *Let* $X_0 \subseteq X_1 \subseteq \ldots \subseteq X_i \subseteq \ldots \subseteq \bigcup_{i=0}^{\infty} X_i$. *Then* $T(\bigcup_{i=0}^{\infty} X_i) = \bigcup_{i=0}^{\infty} T(X_i)$.

g. (*Scott's induction*) *From the two assumptions*

1. $T_1(\Omega) \subseteq T_2(\Omega)$
2. *For arbitrary* X, *if* $T_1(X) \subseteq T_2(X)$ *then* $T_1(T(X)) \subseteq T_2(T(X))$,

it may be concluded that

3. $T_1(P) \subseteq T_2(P)$.

h. (*Scott's induction, general case*). Left to the reader.

PROOF. Reformulation of theorems 1.1, 2.1 and 2.2. □

REMARK. It may be of interest to compare result e of theorem 3.1 with the KNASTER-TARSKI result [28] stating that each monotonic function mapping subsets of some set V to subsets of V (or, more generally each monotonic function on a complete lattice) has a least fixed point. This is shown as follows: Let $D = \cap\{X: T(X) \subseteq X\}$. Note that $\{X: T(X) \subseteq X\}$ is non-empty, since V itself is a member of this collection. Let $D_1 = \cap\{X: T(X) = X\}$. We show that $D = D_1$. $D \subseteq D_1$ is clear. In order to show $D_1 \subseteq D$, it is sufficient to prove that $T(D) = D$.

a. $T(D) \subseteq D$. Let X be such that $T(X) \subseteq X$. Then, by definition of D, $D \subseteq X$; hence, by monotonicity of T, $T(D) \subseteq T(X) \subseteq X$. We see that T(D) is included in each X such that $T(X) \subseteq X$, and, since D is the greatest element with this property, we have that $T(D) \subseteq D$.

b. $D \subseteq T(D)$. From part a we have that $T(T(D)) \subseteq T(D)$. Hence $D \subseteq T(D)$, by the definition of D.

Thus, we see that the *existence* of a least fixed point of T is already implied by its monotonicity. In order to obtain the characterization in terms of successive approximations, i.e., as $\bigcup_{i=0}^{\infty} T^i(\Omega)$, we need in addition the continuity of T. In fact, $T(\bigcup_{i=0}^{\infty} T^i(\Omega)) =$ (cont.) $\bigcup_{i=0}^{\infty} T(T^i(\Omega)) = \bigcup_{i=0}^{\infty} T^{i+1}(\Omega) = \bigcup_{i=0}^{\infty} T^i(\Omega)$. Thus, $\bigcup_{i=0}^{\infty} T^i(\Omega)$ is a fixed point which is (lemma 1.5) included in each fixed point, whence it is the least fixed point.

Another important result which was not yet mentioned in section 1 or 2 is the following. Consider the *system* of declarations

$$\begin{aligned} P_1 &\Leftarrow T_1(P_1, P_2) \\ P_2 &\Leftarrow T_2(P_1, P_2). \end{aligned} \tag{3.5}$$

According to a direct generalization of theorem 3.1, part e, we have

$$(3.6) \qquad (P_1,P_2) = \cap\{(X_1,X_2) : X_1 = T_1(X_1,X_2),\ X_2 = T_2(X_1,X_2)\}$$

which, e.g. according to the KNASTER-TARSKI result, may be replaced by

$$(P_1,P_2) = \cap\{(X_1,X_2) : T_1(X_1,X_2) \subseteq X_1,\ T_2(X_1,X_2) \subseteq X_2\}$$

In other words, (P_1,P_2) is obtained as the simultaneous least fixed point of the pair of transformations (T_1,T_2). However, it can also be approached as *iterated* least fixed points in the following sense:

THEOREM 3.2. *Let* P_1,P_2 *be as in (3.6), and let*

$$P_1' = \cap\{X_1 : X_1 = T_1(X_1,\cap\{X_2 : X_2 = T_2(X_1,X_2)\})\}$$

$$P_2' = \cap\{X_2 : X_2 = T_2(P_1',X_2)\}.$$

Then $P_1 = P_1'$, $P_2 = P_2'$.

PROOF. By the definitions of P_1', P_2' we have $P_1' = T_1(P_1',P_2')$, $P_2' = T_2(P_1',P_2')$. (This uses the fact that $\cap\{X_2 : X_2 = T_2(X_1,X_2)\}$ is monotonic in X_1, verification of which is left to the reader (who may either work this out for himself, or consult theorem 3.3)). Hence we infer that $P_1 \subseteq P_1'$, $P_2 \subseteq P_2'$, by (3.6). Now let $P_2'' \overset{df}{=} \cap\{X_2 : X_2 = T_2(P_1,X_2)\}$. Since $T_2(P_1,P_2) = P_2$, we have $P_2'' \subseteq P_2$; hence, $T_1(P_1,P_2'') \subseteq T_1(P_1,P_2) = P_1$. Replacing P_2'' by its definition we obtain $T_1(P_1,\cap\{X_2 : X_2 = T_2(P_1,X_2)\}) \subseteq P_1$; thus by the definition of P_1' and fixed point induction, it follows that $P_1' \subseteq P_1$. From this we conclude that $T_2(P_1',P_2) \subseteq T_2(P_1,P_2) = P_2$, and $P_2' \subseteq P_2$ follows by the definition of P_2' and fixed point induction. □

Remark. The straightforward generalization of this theorem to a system with $n > 2$ declarations is left to the reader.

Next, we introduce a new notation, which provides an alternative for denoting recursive procedures. Consider the declaration $P \Leftarrow T(P)$. Clearly, one would expect a procedure Q, declared by $Q \Leftarrow T(Q)$, to have the same effect as P, assuming that T(Q) is obtained from T(P) by substituting Q for all occurrences of P in T, and, moreover, that T(P) did not contain any

occurrences of Q to begin with. Thus, we see that in procedure declarations one is confronted with another instance of the phenomenon of a *variable-binding* operator (such as ∀ which binds x in ∀x[x > y → x+1 > y], or λ which binds x in λx•x+2*y). This is made explicit in the following notation: For a procedure P declared by P ⇐ T(P), we denote P by

$$\mu X[T(X)].$$

E.g., for P_1 declared by $P_1 \Leftarrow p;A;P_1 \cup \bar{p}$, we have $P_1 = \mu X[p;A;X \cup \bar{p}]$, and for P_2 declared by $P_2 \Leftarrow p;A_1;P_2;A_2 \cup \bar{p};A_3$, we have $P_2 = \mu X[p;A_1;X;A_2 \cup p;A_3]$.

The μ-operator has the usual consequences for the notions of *free* and *bound* occurrences of variables. In particular, all occurrences of X in μX[T(X)] are bound, and an occurrence of Y in some T_1 is free iff it is not a bound occurrence. Moreover, if Y is any variable not occurring free in T, we have that μX[T(X)] = μY[T(Y)], where T(Y) is the result of substituting Y for all free occurrences of X in T(X). (Without the proviso on Y, we would obtain e.g. the undesirable result that μX[p;Y;X ∪ $\bar{p}$] and μY[p;Y;Y ∪ $\bar{p}$] are equivalent. The reader should check that this would imply the absurd result that, for any Y, p*Y = $\bar{p}$.)

The μ-notation can also be applied directly to systems of procedures in a way which is justified by theorem 3.2. Let $P_1 \Leftarrow T_1(P_1,P_2)$, $P_2 \Leftarrow T_2(P_1,P_2)$ be such a system. Then, by theorem 3.2, $P_1 = \mu X[T_1(X,\mu Y[T_2(X,Y)])]$, $P_2 = \mu Y[T_2(P_1,Y)]$. In order to obtain the full profit of this notation, we are interested in the justification of an iterated version of Scott's induction (examples will follow in the next section). This requires the following extension of our (monotonicity and) continuity result from theorem 3.1:

THEOREM 3.3.

a. *Let* T(X,Y) *be monotonic in* X *and* Y. *Then* μX[T(X,Y)] *is monotonic in* Y.

b. *Let* T(X,Y) *be continuous in* X *and* Y. *Then* μX[T(X,Y)] *is continuous in* Y.

PROOF.

a. (Monotonicity). Let $Y_1 \subseteq Y_2$, $P_1 = \mu X[T(X,Y_1)]$, $P_2 = \mu X[T(X,Y_2)]$. We show that $P_1 \subseteq P_2$. By fixed point induction, it is sufficient to show $T(P_2,Y_1) \subseteq P_2$. By the fixed point property, $P_2 = T(P_2,Y_2)$. Now

$T(P_2,Y_1) \subseteq T(P_2,Y_2)$ follows by the monotonicity of $T(X,Y)$ in Y.

b. (Continuity). Let $Y_0 \subseteq Y_1 \subseteq \ldots \subseteq \bigcup_{i=0}^{\infty} Y_i$. We show that $\mu X[T(X,\bigcup_i Y_i)] = \bigcup_i \mu X[T(X,Y_i)]$. By monotonicity, $\mu X[T(X,Y_i)] \subseteq \mu X[T(X,\bigcup_i Y_i)]$; hence, $\bigcup_i \mu X[T(X,Y_i)] \subseteq \mu X[T(X,\bigcup_i Y_i)]$ is established. To prove the reverse inclusion, we introduce the notation $T^i(X,Y)$, defined by $T^0(X,Y) = \Omega$, $T^{i+1}(X,Y) = T(T^i(X,Y),Y)$. By the continuity of T in X, $\mu X[T(X,Y_i)] = \bigcup_j T^j(\Omega,Y_i)$, and $\mu X[T(X,\bigcup_i Y_i)] = \bigcup_j T^j(\Omega,\bigcup_i Y_i)$. Thus, we see that we have to prove $\bigcup_j T^j(\Omega,\bigcup_i Y_i) \subseteq \bigcup_i \bigcup_j T^j(\Omega,Y_i) = \bigcup_j \bigcup_i T^j(\Omega,Y_i)$. Thus, it is sufficient to show $T^j(\Omega,\bigcup_i Y_i) \subseteq \bigcup_i T^j(\Omega,Y_i)$. We use induction on j.

1. $j = 0$. Clear.

2. Assume the result for j. Then

$$
\begin{aligned}
&T^{j+1}(\Omega,\textstyle\bigcup_i Y_i) = \text{(df.)}\\
&T(T^j(\Omega,\textstyle\bigcup_i Y_i),\bigcup_i Y_i) \subseteq \text{(ind. hypothesis)}\\
&T(\textstyle\bigcup_i T^j(\Omega,Y_i),\bigcup_k Y_k) = \text{(cont. of } T(X,Y) \text{ in } X \text{ and } Y)\\
&\textstyle\bigcup_i \bigcup_k T(T^j(\Omega,Y_i),Y_k) = \text{(mon. of } T(X,Y) \text{ in } X \text{ and } Y)\\
&\textstyle\bigcup_i \bigcup_k T(T^j(\Omega,Y_{\max(i,k)}),Y_{\max(i,k)}) =\\
&\textstyle\bigcup_n T(T^j(\Omega,Y_n),Y_n) =\\
&\textstyle\bigcup_n T^{j+1}(\Omega,Y_n).
\end{aligned}
$$

With this theorem we have completed our presentation of the main mathematical properties of programs with recursive procedures in a relational framework. We saw that the important notions developed in the first two sections can be carried over to this setting, and, moreover, that a new notation required some additional justifications. The next sections will bring a variety of applications of these ideas.

4. APPLICATIONS TO PROGRAM EQUIVALENCE

4.1. A while statement example

We derive a series of results, having some independent interest as well, leading up to the proof of $(p_1 \vee p_2)^*S = p_1^*S;p_2^*(S;p_1^*S)$. We shall use "f.p.p." (fixed point property), "f.p.i." (fixed point induction), and

"l.f.p.p." (least fixed point property) to indicate an appeal to theorem 3.1, parts b, d and e respectively.

a. $\mu X[T(X,X)] = \mu X[\mu Y[T(X,Y)]]$.

$\subseteq$: Call the left-hand side P_1 and the right-hand side P_2. By f.p.p, $P_2 = \mu Y[T(P_2,Y)]$; hence, by f.p.p. again, $P_2 = T(P_2,P_2)$. Hence, by l.f.p.p., $P_1 \subseteq P_2$.

$\supseteq$: By Scott's induction it is sufficient to show: if $X \subseteq P_1$ then $\mu Y[T(X,Y)] \subseteq P_1$. In order to establish this, we apply once more Scott's induction, and now we must show: if $X \subseteq P_1$, and $Y \subseteq P_1$, then $T(X,Y) \subseteq P_1$. Since $P_1 = T_1(P_1,P_1)$ by f.p.p., the result follows by monotonicity.

(Observe that we have here a case of iterated Scott's induction, as justified by theorem 3.3.)

b. $\mu X[T_1(T_2(X))] = T_1(\mu X[T_2(T_1(X))])$, or $P_1 = T_1(P_2)$ for short.

$\subseteq$: Assume $X \subseteq T_1(P_2)$. Then $T_1(T_2(X)) \subseteq T_1(T_2(T_1(P_2))) = T_1(P_2)$, by monotonicity and f.p.p. Hence, the result follows by Scott's induction.

$\supseteq$: Assume $T_1(X) \subseteq P_1$. Then $T_1(T_2(T_1(X))) \subseteq P_1 = T_1(T_2(P_1))$ by monotonicity, and the result follows again by Scott's induction.

c. $\mu X[A;T(X)] = A;\mu X[T(A;X)]$

Proof: special case of b.

d. $\mu X[p;A_1;X \cup \bar{p};A_2] = \mu X[p;A_1;X \cup \bar{p}];A_2$

Relational reformulation of example 1 of section 2.

e. $(p_1 \vee p_2)^*S =$

$\mu X[(p_1 \cup p_2);S;X \cup \overline{p_1 \cup p_2}] =$

$\mu X[p_1;S;X \cup p_2;S;X \cup \bar{p}_1;\bar{p}_2] =$

$\mu X[p_1;S;X \cup \bar{p}_1;(p_2;S;X \cup \bar{p}_2)] =$ (part a)

$\mu X[\mu Y[p_1;S;Y \cup \bar{p}_1;(p_2;S;X \cup \bar{p}_2]]$ = (part d)

$\mu X[\mu Y[p_1;S;Y \cup \bar{p}_1](p_2;S;X \cup \bar{p}_2)]$ = (def. *)

$\mu X[p_1^*S;(p_2;S;X \cup \bar{p}_2)]$ = (part c)

$p_1^*S;\mu X[p_2;S;p_1^*S;X \cup \bar{p}_2]$ = (def. *)

$p_1^*S;p_2^*(S;p_1^*S)$.

4.2. Tree traversal

We present an example (taken from [4]) which originated from the desire to prove the equivalence of two ways of tree- (actually, forest-)

traversal. It soon appeared, however, that the desired equivalence is a special case of a more general result. First we state the original problem. We use the "family-oriented" terminology for trees: Let s and b be predicates which when applied to node x, are interpreted as

s(x) is true iff x has a son
b(x) is true iff x has a younger brother.

Let S, B and F be elementary actions which, when applied in node x, have the following effect:

S(x): visit the eldest son of x
B(x): visit the next-younger brother of x
F(x): visit the father of x.

Let A be an arbitrary elementary action, to be performed in all nodes of the tree, without side effect on the traversal mechanism. Let

$$Q_1 = \mu X[A;(s;S;X;F \cup \bar{s});(b;B;X \cup \bar{b})]$$
$$Q_2 = \mu X[A;(s;S;X;b^*(B;X);F \cup \bar{s})].$$

The following relationship was conjectured to hold between Q_1 and Q_2:

$$Q_1 = Q_2;b^*(B;Q_2).$$

However, a need was felt for a formal verification of this equivalence, and this was provided in the following way: First of all, some study of the form of the definitions of Q_1 and Q_2, and their alleged relationship, yielded that the result to be shown is in fact independent of the special domain of trees and is a special case of a rather general equivalence stated as: Let us define

(4.1) $$P_0 = \mu X[T_0(T_1(X),T_2(X))]$$

(4.2) $$P_1(Y) = \mu X[T_1(T_0(X,Y))]$$

(4.3) $$P_2 = \mu X[T_2(T_0(P_1(X),X))].$$

Then

(4.4) $\quad P_0 = T_0(P_1(P_2),P_2)$.

Assuming this to be established, we see that our tree-traversal result may be obtained by taking

$$T_0(X,Y) = Y;X$$
$$T_1(X) = b;B;X \cup \bar{b}$$
$$T_2(X) = A;(s;S;X;F \cup \bar{s})$$

In fact, with this choice for the T's, $P_0 = \mu X[T_2(X);T_1(X)] = Q_1$, $P_1(Y) = \mu X[b;B;Y;X \cup \bar{b}] = b^*(B;Y)$, and $P_2 = \mu X[A;(s;S;X;b^*(B;X);F \cup \bar{s})] = Q_2$, whence $T_0(P_1(P_2),P_2) = P_2;P_1(P_2) = P_2;b^*(B;P_2) = Q_2;b^*(B;Q_2)$, as was to be shown.

We now prove (4.4).

$\subseteq$: We have, by (4.2) and (4.3), that $P_1(P_2) = T_1(T_0(P_1(P_2),P_2)$, and $P_2 = T_2(T_0(P_1(P_2),P_2))$. Hence, $T_0(T_1(T_0(P_1(P_2),P_2)), T_2(T_0(P_1(P_2),P_2)))$ $= T_0(P_1(P_2),P_2)$, and $P_0 \subseteq T_0(P_1(P_2),P_2)$ follows by (4.1) and l.f.p.p.

$\supseteq$: By (4.1), $T_1(P_0) = T_1(T_0(T_1(P_0),T_2(P_0)))$. Hence, by (4.2) and l.f.p.p.

(4.5) $\quad P_1(T_2(P_0)) \subseteq T_1(P_0)$.

Using this, we show that $T_0(P_1(P_2),P_2) \subseteq P_0$ by Scott's induction on P_2. Assume $T_0(P_1(X),X) \subseteq P_0$. We prove that then $T_0(P_1(T_2(T_0(P_1(X),X))), T_2(T_0(P_1(X),X))) \subseteq P_0$. Using the induction assumption, this simplifies to verification of $T_0(P_1(T_2(P_0)),T_2(P_0)) \subseteq P_0 = T_0(T_1(P_0),T_2(P_0))$, which follows by (4.5), using the definition of P_0 and monotonicity. □

The result of this section has been further generalized by DE ROEVER in [27].

4.3. The 91-function

In (1.2) we defined the 91-function g(x) (which was first considered By McCARTHY) as

$$g(x) \Leftarrow \underline{\text{if}}\ x > 100\ \underline{\text{then}}\ x - 10\ \underline{\text{else}}\ g(g(x+11))$$

and promised to show that $g(x) = h(x)$, with $h(x)$ defined as

$$h(x) \Leftarrow \underline{\text{if}}\ x > 100\ \underline{\text{then}}\ x - 10\ \underline{\text{else}}\ 91$$

a. $g \subseteq h$.

By f.p.i. it is sufficient to show

$\lambda x \cdot \underline{\text{if}}\ x > 100\ \underline{\text{then}}\ x - 10\ \underline{\text{else}}\ h(h(x+11)) \subseteq \lambda x \cdot h(x)$.

We have

$\lambda x \cdot \underline{\text{if}}\ x > 100\ \underline{\text{then}}\ x - 10\ \underline{\text{else}}\ h(h(x+11)) =$

$\lambda x \cdot \underline{\text{if}}\ x > 100\ \underline{\text{then}}\ x - 10\ \underline{\text{else}}\ h(\underline{\text{if}}\ x+11 > 100\ \underline{\text{then}}\ x+11-10\ \underline{\text{else}}\ 91) =$

$\lambda x \cdot \underline{\text{if}}\ x > 100\ \underline{\text{then}}\ x - 10\ \underline{\text{else}}\ \underline{\text{if}}\ x + 11 > 100\ \underline{\text{then}}\ h(x+1)\ \underline{\text{else}}\ h(91) =$

$\lambda x \cdot \underline{\text{if}}\ x > 100\ \underline{\text{then}}\ x - 10\ \underline{\text{else}}\ 91 =$

$\lambda x \cdot h(x)$

since, if $x \leq 100$ and $x + 11 > 100$, then either $89 < x < 100$, whence $h(x+1) = 91$, or $x = 100$, whence $h(x+1) = h(101) = 91$.

b. $h \subseteq g$.

Let $k(x) \Leftarrow \underline{\text{if}}\ x > 100\ \underline{\text{then}}\ x - 10\ \underline{\text{else}}\ k(x+1)$. Then $h = k$. In fact, $h \subseteq k$ follows as in part a; that $k \subseteq h$ can be proved from basis properties of the integers (see below), as exhibited in [2]. The proof of this is not repeated here. We now show $k \subseteq g$: It is sufficient to show $\lambda x \cdot g(x) \subseteq \lambda x \cdot g(g(x+10))$. This follows once more by Scott's induction. As hypotheses we take $X \subseteq \lambda x \cdot g(x)$ and $X \subseteq \lambda x \cdot g(g(x+10))$.

We verify that

1. $\lambda x \cdot \underline{\text{if}}\ x > 100\ \underline{\text{then}}\ x - 10\ \underline{\text{else}}\ X(X(x+11)) \subseteq \lambda x \cdot g(x)$, which is clear, and
2. $\lambda x \cdot \underline{\text{if}}\ x > 100\ \underline{\text{then}}\ x - 10\ \underline{\text{else}}\ X(X(x+11)) \subseteq \lambda x \cdot g(g(x+10))$.

   The left-hand side of this inclusion is rewritten as

   $\lambda x \cdot \underline{\text{if}}\ x + 10 > 100\ \underline{\text{then}}\ (\underline{\text{if}}\ x > 100\ \underline{\text{then}}\ x-10\ \underline{\text{else}}\ X(X(x+11)))$
   $\underline{\text{else}}\ X(X(x+11))$

   and the right-hand side as

   $\lambda x \cdot \underline{\text{if}}\ x + 10 > 100\ \underline{\text{then}}\ g(x)\ \underline{\text{else}}\ g(g(g(x+10+11)))$.

   Now we see that the left-hand side is indeed included in the right-hand side since

   $\lambda x \cdot \underline{\text{if}}\ x > 100\ \underline{\text{then}}\ x - 10\ \underline{\text{else}}\ X(X(x+11)) \subseteq \lambda x \cdot g(x)$,

   by the definition of g and the first hypothesis, and

$\lambda x \cdot X(X(x+11)) \subseteq \lambda x \cdot g(g(g(x+10+11)))$, which follows from $X \subseteq g$ and $X \subseteq \lambda x \cdot g(g(x+10))$.

(The basic properties of the natural numbers referred to above are the following: We use the relational version, with S the successor relation, $\breve{S}$: predecessor, and p_0 (=test for zero) = $\overline{\breve{S};S} \cap I$ (- denoting complementation with respect to the universal relation U.) Then we postulate: $S;\breve{S} = I$, $\breve{S};S \subseteq I$, and $U \subseteq \breve{S}^*;p_0;S^*$. From these assumptions one can show that 1) p_0 is an "atom", i.e., $p_0;U \cap U;p_0 \subseteq p_0$. 2) $I \subseteq \mu X[p_0 \cup \breve{S};X;S]$. Property 2, together with a suitable inductive definition of the ">" relation, is used in [2] to show that $k \subseteq h$.)

4.4. Miscellaneous

We present here two remarks which are a side-effect of our attempts at an understanding of the method described in COOPER [11]. We shall not try to summarize the method, since part of our problem is that we do not sufficiently grasp what is proposed in it. We conjecture, however, that remark 1 and / or remark 2 have some (possibly common) generalization explaining COOPERS's ideas.

REMARK 1. Let P_1, P_2, P_3 and P_4 be defined by

$$P_1 = \mu X[T_1(X)] \qquad P_3 = \mu X[T_1(T_2(X))]$$
$$P_2 = \mu X[T_2(X)] \qquad P_4 = \mu X[T_2(T_1(X))]$$

Suppose we know that P_1, P_2, P_3 and P_4 are all total functions. Then $P_1 = P_2$ iff $P_3 = P_4$.

PROOF.

a. Assume $P_3 = P_4$. We have
$P_3 = \mu X[T_1(T_2(X))] =$ (cf. section 4.1, part b)
$T_1(\mu X[T_2(T_1(X))]) = T_1(P_4) =$ (assumption) $T_1(P_3)$.
Hence, $P_1 \subseteq P_3$, by f.p.i. Similarly, $P_2 \subseteq P_4$. Since P_1 and P_2 are total functions, $P_1 = (P_3 = P_4 =) P_2$ follows.

b. Assume $P_1 = P_2$. We have
$P_1 = T_1(P_1) = T_1(P_2) = T_1(T_2(P_2)) = T_1(T_2(P_1))$.
Hence, $P_3 \subseteq P_1$, by f.p.i., and similarly $P_4 \subseteq P_2$. That $P_3 = P_4$ now

follows as in part a. □

REMARK 2. Let T_1, T_2 be continuous, and such that

a. $T_1(\Omega) = T_2(\Omega)$.

b. For some fixed $i \geq 1$, and all X,
$T_1^i(T_2(X)) = T_2(T_1(X))$.

Then $\mu X[T_1(X)] = \mu X[T_2(X)]$.

PROOF. We first show that, for each k=1,2,...

$$T_2^k(\Omega) = T_1^{1+i+\ldots+i^{k-1}}(\Omega) \tag{4.6}$$

by induction on k.

1. Basis step: $T_2(\Omega) = T_1(\Omega)$ is direct from assumption a.
2. Assume the result for k. From assumption b we obtain that for each integer $\ell \geq 0$: $T_2(T_1^{\ell}(X)) = T_1^{i*\ell}(T_2(X))$. Thus, we derive

$$T_2^{k+1}(\Omega) = T_2(T_2^k(\Omega)) = \text{(ind. hyp.)}$$

$$T_2(T_1^{1+i+\ldots+i^{k-1}}(\Omega)) = T_1^{i*(1+i+\ldots+i^{k-1})}(T_2(\Omega)) = \text{(ass. a)}$$

$$T_1^{i*(1+i+\ldots+i^{k-1})+1}(\Omega) = T_1^{1+i+\ldots+i^{k}}(\Omega).$$

This gives the proof of (4.6). Now $U_{j=0}^{\infty} T_1^j(\Omega) = U_{j=0}^{\infty} T_2^j(\Omega)$ is direct by monotonicity of T_1 and T_2. □

For some time it was thought that remark 2 could not be proved using only the pure μ-calculus (i.e., using only Scott's induction, and not the characterization of recursive procedures as infinite unions). Recently however, MILNER showed (private communication) how such a proof could be given: Assume a and b above (i=2,say), and let $P_1 = \mu X[T_1(X)]$, $P_2 = \mu X[T_2(X)]$.

1. $P_2 \subseteq P_1$. This is easily shown by applying Scott's induction to obtain the proof of $T_2(P_1) \subseteq P_1$.
2. $P_1 \subseteq P_2$. We apply the general form of Scott's induction to $\Phi(P_1)$, where $\Phi(X)$ consists of the three inclusions

$$X \subseteq T_1(X) \tag{4.7}$$

(4.8) $T_1(X) \subseteq T_2(X)$

(4.9) $X \subseteq P_2$.

$\Phi(\Omega)$ is immediate from the assumption that $T_1(\Omega) = T_2(\Omega)$. Now assume $\Phi(X)$, and show $\Phi(T_1(X))$, i.e.,

(4.10) $T_1(X) \subseteq T_1(T_1(X))$

(4.11) $T_1(T_1(X)) \subseteq T_2(T_1(X))$

(4.12) $T_1(X) \subseteq P_2$

The proofs of these are obtained as follows:

(4.10): immediate from (4.7)

(4.11): $T_1(T_1(X)) \overset{(4.7)}{\subseteq} T_1(T_1(T_1(X))) \overset{(4.8)}{\subseteq}$

$T_1(T_1(T_2(X))) \subseteq$ (ass. b) $T_2(T_1(X))$

(4.12): $T_1(X) \overset{(4.8)}{\subseteq} T_2(X) \overset{(4.9)}{\subseteq} T_2(P_2) = P_2$

Thus, $\Phi(P_1)$ holds by Scott's induction, implying $P_1 \subseteq P_2$. □

It is of some interest to observe that remark 2 cannot be proved using only monotonicity of T_1 and T_2. This was shown by VUILLEMIN, who provided a counter-example for i = 1, see exercise 6.3.

5. APPLICATIONS TO PROGRAM CORRECTNESS

5.1. The inductive assertion method

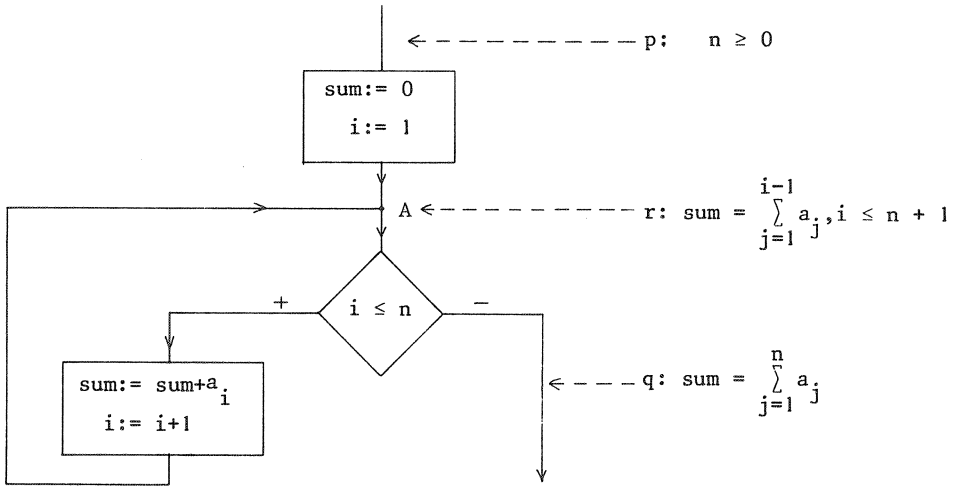
A program P is called *partially correct* with respect to the (initial) and final) predicates p and q iff

(5.1) $\forall x,y[p(x) \wedge xPy \rightarrow q(y)]$

i.e., iff for all states x,y, if state x satisfies p, and P transforms x to y, then state y satisfies q. This is the formulation which leads to the *inductive assertion method*, as proposed by FLOYD [14] and further developed by HOARE [16] and MANNA (e.g. [18,19]).

Our aim here is to study the theoretical properties of the method, and in particular to prove its consistency and completeness in the framework of our relational theory (section 5.2). Furthermore, we shall deal in some detail with HOARE's formulation of the method when applied to while statements, and investigate the power of the axiom he has proposed (section 5.3). Finally, we spend some attention on ideas in a recent paper by DIJKSTRA [13], which we try to interpret in our framework, with as main result a very short proof of a (corrected) version of the main result of that paper (section 5.4).

For the benefit of the reader who has not seen the inductive assertion method before, we first briefly explain it using a simple example from [14]. Consider the following flow diagram for calculating the sum of the n numbers $a_1, a_2, \ldots, a_n$:



For this program we want to show that, if the initial condition p: n ≥ 0 is satisfied, then the final condition q: sum = $\sum_{j=1}^{n} a_j$, holds. This is done via the introduction of a suitable intermediate (so-called inductive) assertion r, which has to satisfy:

1. Basis step: When control arrives at point A in the diagram for the first time, the current values of the variables satisfy r. This follows since sum = 0, i = 1 imply sum = $\sum_{j=1}^{i-1} a_j$ (=0), and n ≥ 0, i = 1 imply i ≤ n + 1.
2. Inductive step: Assume r holds at point A at any intermediate stage in the computation. Then we verify that r holds again when control arrives in A after once going through the loop. I.e., assume sum = $\sum_{j=1}^{i-1} a_j$, i ≤ n + 1. Since the + exit of the test is taken, we know that i ≤ n. Executing sum:= sum + a_i gives sum = $\sum_{j=1}^{i} a_j$. Next, i:= i + 1 yields that sum = $\sum_{j=1}^{i-1} a_j$, and i ≤ n + 1, together establishing r.

From 1 and 2 we conclude that r holds at all stages of the computation. Thus, when eventually the - exit from the test is taken, it is easily checked that ¬(i≤n) and r together imply q.

It should be observed that the method does not deal with *termination*. Separate means are needed in the example to prove that i ≤ n will become false eventually. (This explains the qualification *partial* in our terminology. When termination is also shown, it is customary [18] to speak of *total* correctness.)

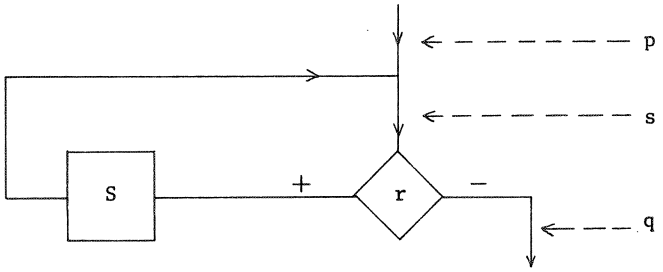
Further details of the inductive assertion method, with many examples, can be found e.g. in [19].

5.2. Consistency and completeness of the method

Relationally, we write for (5.1)

(5.2) $\quad p;P \subseteq P;q.$

By way of introduction to the general problem, we shall first deal with the simple case that P is a while statement, say P = r*S. The formulation of the inductive assertion method for this case can be read off from the following picture:



In order to prove $p;r^*S \subseteq r^*S;q$, we try to find intermediate s such that

$$(5.3) \qquad \begin{cases} p \subseteq s \\ s;r;S \subseteq r;S;s \\ s;\bar{r} \subseteq \bar{r};q \end{cases}$$

The consistency of the inductive assertion method is then expressed by the following formula (from second-order predicate logic):

$$(5.4) \qquad \forall p,q \left[\exists s \left[\begin{array}{c} p \subseteq s \\ s;r;S \subseteq r;S;s \\ s;\bar{r} \subseteq \bar{r};q \end{array} \right] \Rightarrow p;r^*S \subseteq r^*S;q \right]$$

Verification of (5.4) is immediate, by using $r^*S = (r;S)^*;\bar{r}$.

The *completeness* of the method is expressed by the converse of (5.4):

$$(5.5) \qquad \forall p,q \left[p;r^*S \subseteq r^*S;q \Rightarrow \exists s \left[\begin{array}{c} p \subseteq s \\ s;r;S \subseteq r;S;s \\ s;\bar{r} \subseteq \bar{r};q \end{array} \right] \right] .$$

(This is actually a reformulation of ideas by MANNA, which needs to be refined, however, in order to deal with his treatment of total correctness, see [3].)

Before proving (5.5), we first develop some further tools. Let us look once more at (5.1)

$$\forall x,y[p(x) \wedge xPy \rightarrow q(y)].$$

This may be rewritten in two other, equivalent, forms:

$$\forall y[\exists x[p(x) \wedge xPy] \rightarrow q(y)]$$
$$\forall x[p(x) \rightarrow \forall y[xPy \rightarrow q(y)]]$$

leading to the introduction of two new operators, denote by "$\circ$" (not to be confused with the operator denoting composition of functions of section 1) and "$\rightarrow$", respectively.

DEFINITION 5.1.

$$(p \circ P)(x) \leftrightarrow \exists y[p(y) \wedge yPx]$$
$$(P \rightarrow p)(x) \leftrightarrow \forall y[xPy \rightarrow p(y)].$$

From the definition the following lemma is easily obtained:

LEMMA 5.1.

1. $p;P \subseteq P;(p \circ P)$
   $(P \rightarrow q);P \subseteq P;q.$
2. *For all* p,q, *if* $p;P \subseteq P;q$, *then* $p \circ P \subseteq q$ *and* $p \subseteq (P \rightarrow q)$.
3. $p \circ P = \bigcap \{q \mid p;P \subseteq P;q\}$
   $P \rightarrow q = \bigcup \{p \mid p;P \subseteq P;q\}$.

Some further properties of "$\circ$" and "$\rightarrow$" are collected in lemma 5.2.

LEMMA 5.2.

1. $\Omega \circ P = p \circ \Omega = \Omega$, $P \rightarrow I = I$, $I \rightarrow p = p$.
2. $P;I \circ P = P$, $(P \rightarrow \Omega);P = \Omega$.
3. $p \circ q = p \cap q = p;q$, $I \subseteq (p_1 \rightarrow p_2)$ *iff* $p_1 \subseteq p_2$.
4. $p \circ (P_1;P_2) = (p \circ P_1) \circ P_2$, $(P_1;P_2) \rightarrow p = P_1 \rightarrow (P_2 \rightarrow p)$.
5. $p \circ (P_1 \cup P_2) = (p \circ P_1) \cup (p \circ P_2)$, $(P_1 \cup P_2) \rightarrow p = (P_1 \rightarrow p) \cap (P_2 \rightarrow p)$.
6. *If* $P_1 \subseteq P_2$, *then* $p \circ P_1 \subseteq p \circ P_2$, *and* $P_2 \rightarrow p \subseteq P_1 \rightarrow p$.
7. *If* $p \subseteq q$, *then* $p \circ P \subseteq q \circ P$, *and* $P \rightarrow p \subseteq P \rightarrow q$.
8. $(p \cup q) \circ P = (p \circ P) \cup (q \circ P)$, $P \rightarrow (p \cap q) = (P \rightarrow p) \cap (P \rightarrow q)$.
9. *If* $\breve{P}$ *is a function (i.e.,* $P;\breve{P} \subseteq I$*), then* $(p \cap q) \circ P = (p \circ P) \cap (q \circ P)$.
   If P *is a function (i.e.,* $\breve{P};P \subseteq I$*), then* $P \rightarrow (p \cup q) = (P \rightarrow p) \cup (P \rightarrow q)$.

PROOF. Clear from the definitions. □

Let U, as before, denote the universal relation, and let $\bar{}$ this time denote complementation with respect to I. Then

LEMMA 5.3.

1. $p \circ P = U;p;\underline{P \cap I}$.
2. $P \rightarrow p = \overbrace{\bar{p} \circ P}$.

PROOF. Left to the reader. □

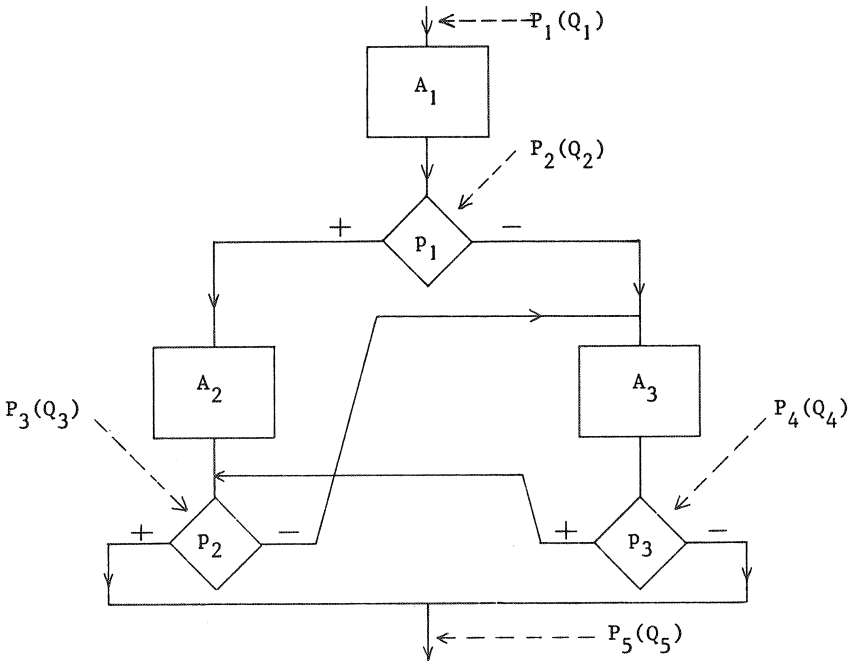
The operators "$\circ$" and "$\rightarrow$" provide us with the tools to prove (5.5) and its generalizations. For (5.5), we may take either $s = p \circ (r;S)^*$, or $s = (r;S)^*;\bar{r} \rightarrow q$. We verify only that the first choice of s satisfies the conditions.

a. $p = p \cap I = p \circ I \subseteq p \circ (r;S)^*$,
using lemma 5.2, parts 3 and 6.

b. $(p \circ (r;S)^*);r;S \subseteq r;S;(p \circ (r;S)^*)$ iff (lemma 2.1.3)
$(p \circ (r;S)^*) \circ (r;S) \subseteq p \circ (r;S)^*$ iff (lemma 2.2.4)
$p \circ ((r;S)^*;r;S) \subseteq p \circ (r;S)^*$ if (lemma 2.2.6)
$(r;S)^*;r;S \subseteq (r;S)^*$
and the last inclusion is clear from the definition of *.

c. $(p \circ (r;S)^*);\bar{r} \subseteq \bar{r};q$ iff (lemma 2.1.3)
$(p \circ (r;S)^*) \circ \bar{r} \subseteq q$ iff (lemma 2.2.4)
$(p \circ ((r;S)^*;\bar{r})) \subseteq q$ iff
$p \circ (r^*S) \subseteq q$ iff (lemma 2.1.3)
$p;r^*S \subseteq r^*S;q$
and the last inclusion holds by assumption. □

The next step is to extend (5.4) and (5.5) to flowcharts which are not just simple while statements. Here we use the well-known idea of associating a system of recursive procedures with a flowchart, such that execution of, say, the first procedure of the system is equivalent to executing the flowchart. We shall not describe this association in any formal detial, but illustrate it by an example. Consider the diagram:



Two solutions are provided, which are dual to each other in a rather natural sense:

1. $P_1 \Leftarrow A_1;P_2$
   $P_2 \Leftarrow P_1;A_2;P_3 \cup \bar{P}_1;A_3;P_4$
   $P_3 \Leftarrow P_2;P_5 \cup \bar{P}_2;A_3;P_4$
   $P_4 \Leftarrow P_3;P_3 \cup \bar{P}_3;P_5$
   $P_5 \Leftarrow I.$
2. $Q_1 \Leftarrow I$
   $Q_2 \Leftarrow Q_1;A$
   $Q_3 \Leftarrow Q_2;P_1;A_2 \cup Q_4;P_3$
   $Q_4 \Leftarrow Q_2;\bar{P}_1;A_3 \cup Q_3;\bar{P}_2;A_3$
   $Q_5 \Leftarrow Q_3;P_2 \cup Q_4;\bar{P}_3.$

We expect the reader to have no difficulties in convincing himself that execution of the flowchart is equivalent to execution of either P_1 or Q_5. Lacking a formal definition of the way a flowchart is executed, we cannot present a formal proof of this. What we do prove is (the generalization of) the equivalence between P_1 and Q_5.

Let us consider the general form of a system of declarations of which that for P_1 to P_5 is a special case:

$$
\begin{aligned}
&P_i \Leftarrow A_{i,1};P_1 \cup \ldots \cup A_{i,n};P_n \cup A_{i,n+1};P_{n+1}, \quad i=1,\ldots,n \\
&P_{n+1} \Leftarrow I
\end{aligned}
\tag{5.6}
$$

which we compare with the "transposed" system

$$
Q_j \Leftarrow Q_1;A_{1,j} \cup \ldots \cup Q_n;A_{n,j} \cup \Delta_j, \qquad j=1,\ldots,n+1
$$

with Δ_j defined as

$$
\begin{aligned}
&\Delta_1 = I \\
&\Delta_j = \Omega \qquad\qquad j=2,\ldots,n+1.
\end{aligned}
$$

We shall prove by Scott's induction that

$$
Q_j;P_j \subseteq P_1 \cap Q_{n+1}, \qquad j=1,2,\ldots,n+1. \tag{5.7}
$$

Once this has been done, the result $P_1 = Q_{n+1}$ is immediately obtained: Application of (5.7) with j = 1 yields that $P_1 = \Delta_1;P_1 \subseteq Q_1;P_1 \subseteq Q_{n+1}$, and with j = n + 1 we get $Q_{n+1} = Q_{n+1};I = Q_{n+1};P_{n+1} \subseteq P_1$.

1. Proof of $Q_j;P_j \subseteq P_1$, j=1,2,...,n+1. By Scott's induction it is sufficient to verify that from the assumption $X_j;P_j \subseteq P_1$, j=1,...,n+1. we may infer that $\{(\bigcup_{k=1}^{n} X_k;A_{k,j}) \cup \Delta_j\};P_j \subseteq P_1$, j=1,...,n+1. We have, for j=1,...,n+1, that $\bigcup_{k=1}^{n} X_k;A_{k,j};P_j \subseteq$ (df.P_k) $\bigcup_{k=1}^{n} X_k;P_k$ $\subseteq$ (ind. ass.) P_1. Also, $\Delta_j;P_j \subseteq P_1$, j=1,...,n+1, is immediate from the definition of Δ_j.
2. Proof of $Q_j;P_j \subseteq Q_{n+1}$, j=1,...,n+1. Assume $Q_j;X_j \subseteq Q_{n+1}$, j=1,...,n+1. We have, for j=1,...,n, $Q_j;\{(\bigcup_{k=1}^{n} A_{j,k};X_k) \cup A_{j,n+1}\}$ $\subseteq \bigcup_{k=1}^{n}(Q_j;A_{j,k};X_k \cup Q_j;A_{j,n+1}) \subseteq$ (df. Q_k) $\subseteq \bigcup_{k=1}^{n}(Q_k;X_k) \cup Q_{n+1} \subseteq$ (ind. ass.) Q_{n+1}. For j=n+1, $Q_j;I \subseteq Q_{n+1}$ is obvious. □

We now return to the main theme of this subsection: the consistency and completeness of the inductive assertion method. For the flowchart case,

i.e., for systems such as (5.6), this is formulated as follows:
Let (5.6) be given. We are interested in the partial correctness of P_1, say, with respect to given p,q. Now we assert that

$$\forall p,q\left[p;P_1 \subseteq P_1;q \quad \text{iff} \quad \exists p_1,\ldots,p_{n+1}\left[\begin{array}{l} p \subseteq p_1 \\ p_{n+1} \subseteq q \end{array}, \; p_i;A_{i,j} \subseteq A_{i,j};p_j, \; \substack{i=1,\ldots,n,\\ j=1,\ldots,n+1}\right]\right]$$

PROOF.

1. If : Direct by Scott's induction.
2. Only if: Assume $p;P_1 \subseteq P_1;q$. We have two possible solutions for the p_i : $p_i = P_i \to q$, and $p_i = p \circ Q_i$. We verify the first solution.
   From $p;P_1 \subseteq P_1;q$ we have $p \subseteq (P_1 \to q) = p_1$. Also, $p_{n+1} = P_{n+1} \to q =$
   $= I \to q = q$. In order to show that $p_i;A_{i,j} \subseteq A_{i,j};p_j$, we must check whether $(P_i \to q);A_{i,j} \subseteq A_{i,j};(P_j \to q)$, i.e.,
   $\forall x,y[\forall z[xP_i z \to q(z)] \wedge xA_{i,j}y \to \forall t[yP_j t \to q(t)]]$
   Assume $\forall z[xP_i z \to q(z)]$, $xA_{i,j}y$, and $yP_j t$. Then $xA_{i,j};P_j t$, hence $xP_i t$, and q(t) follows. □

The result just proved can be extended to systems of recursive procedures which are not restricted to the "regular" form of (5.6). (Note that there is a natural way of associating a grammar with the system (5.6) which, according to standard terminology, is regular. The systems to be dealt with presently have arbitrary context-free grammars associated with them.) We shall not present the full development of this, which is rather complicated and the main topic of our paper [3]. Rather than doing this, we shall give some hints on the direction this generalization takes.

First we consider a declaration of the form

$$P \Leftarrow A_1;P;A_2 \cup A_3. \tag{5.8}$$

We are interested in the extension of the preceding results to such P. The solution needs an extension of the inductive assertion method in that now an infinity of intermediate assertions is used. In fact, we have that

$$(5.9)\quad \forall p,q\left[\begin{array}{l} p;P \subseteq P;q \quad \text{iff} \\ \\ \exists\{p_i,q_i\}_{i=0,1,\ldots}\left[\begin{array}{l} p \subseteq p_0\,, \\ q_0 \subseteq q\,, \\ \left\{\begin{array}{l} p_i;A_1 \subseteq A_1;p_{i+1} \\ q_{i+1};A_2 \subseteq A_2;q_i \\ p_i;A_3 \subseteq A_3;q_i \end{array}\right\}_{i=0,1,\ldots} \end{array}\right] \end{array}\right]$$

The proof of the if-part is again direct. For the only-if-part we define

$$p_i = p \circ A_1^i \quad , \qquad i=0,1,\ldots$$

$$q_i = p \circ \left(\overset{\infty}{\underset{n=i}{\cup}}\, A_1^n;A_3;A_2^{n-i}\right) , \qquad i=0,1,\ldots$$

or

$$p_i = \left(\overset{\infty}{\underset{n=i}{\cup}}\, A_1^{n-i};A_3;A_2^n\right) \rightarrow q, \qquad i=0,1,\ldots$$

$$q_i = A_2^i \rightarrow q \quad , \qquad i=0,1,\ldots$$

Verification that these two solutions for the p_i,q_i satisfy the conditions is left to the reader. The following example may help the intuition. Consider as special case of (5.8):

$$P \Leftarrow [n>0 \mid n:=n-1];P;[n:=n+1] \cup [n=0]$$

in a notation we hope is self-explanatory. We want to show that for each non-negative integer n, we have nPn. This is proved by taking, for some fixed n_0, $p_i(n)$ as : $n = n_0 - i$, and $q_i(n) \equiv p_i(n)$. It is not difficult to see that the p_i,q_i satisfy the conditions of (5.9) for this particular choice of the A_1,A_2 and A_3. Hence, $p_0;P \subseteq P;q_0$, or $\forall n,m[n=n_0 \wedge nPm \rightarrow m=n_0]$, which is equivalent to $\forall n[nPn]$, is established. □

The situation becomes more complex with our next example:

$$P \Leftarrow A_1;P;A_2;P;A_3 \cup A_4.$$

The simple indexing of the assertions used above (p_i,q_i,i=0,1,...) is now no longer sufficient. Instead of this, we use assertions indexed with finite sequences of 0's and 1's: Let ε be the empty such sequence, and σ an

arbitrary element of $\{0,1\}^*$. Then we have as analogue of (5.9):

$$(5.10)\quad \forall p,q\left[\begin{array}{l} p;P \subseteq P;q \quad \text{iff } \exists\{p_\sigma,p_\sigma\}_{\sigma\in\{0,1\}^*} \text{ such that} \\ \qquad p \subseteq p_\varepsilon\ , \quad q_\varepsilon \subseteq q\ , \quad \left\{\begin{array}{l} p_\sigma;A_1 \subseteq A_1;p_{\sigma 0} \\ q_{\sigma 0};A_2 \subseteq A_2;p_{\sigma 1} \\ q_{\sigma 1};A_3 \subseteq A_3;q_\sigma \\ p_\sigma;A_4 \subseteq A_4;q_\sigma \end{array}\right\}_{\sigma\in\{0,1\}^*} \end{array}\right]$$

The proof of the if-part of (5.10) is again not difficult, but the only-if-part needs additional tools which will not be developed in these notes. (see [3] for the full story). What we do provide is an indication how to view (5.10) in such a way that application to practical proofs becomes feasible. Consider the example

$$P \Leftarrow [n>0] \mid n:=n-1];P;[t:=t+1];P;[n:=n+1] \cup [n=0].$$

(The reader might recognize here part of the control structure of the recursive solution of the towers of Hanoi puzzle. The result proved presently yields the number of necessary disc-movements.) We want to prove that $(n,t)P(n,t+2^n-1)$. A direct proof using (5.10) is possible but awkward. A more convenient method is based upon a stronger version of (5.10):

$$\forall p,q\left[\begin{array}{l} p;P \subseteq P;q \quad \text{iff} \\ \exists\mathcal{D},f\colon \mathcal{D}\to\mathcal{D},g\colon \mathcal{D}\to\mathcal{D},\{p(\sigma),q(\sigma)\}_{\sigma\in\mathcal{D}},\sigma_0\in\mathcal{D} \text{ such that} \\ \qquad p \subseteq p(\sigma_0),\ q(\sigma_0) \subseteq q, \quad \left\{\begin{array}{l} p(\sigma);A_1 \subseteq A_1;p(f(\sigma)) \\ q(f(\sigma));A_2 \subseteq A_2;p(g(\sigma)) \\ q(g(\sigma));A_3 \subseteq A_3;q(\sigma) \\ p(\sigma);A_4 \subseteq A_4;q(\sigma) \end{array}\right\}_{\sigma\in\mathcal{D}} \end{array}\right]$$

i.e., instead of assertions p,q indexed by $\sigma\in\{0,1\}^*$ we use p,q with σ, element of some suitable domain $\mathcal{D}$, as *parameter*. Applying the idea to our example, which manipulates states x consisting of pairs of integers (n,t), we make the following choices for $\mathcal{D}$, p,q, f and g. Let $\mathcal{D}$ also consist of pairs (ν,τ) of integers. We put

$$p(\sigma)(x) = p(\nu,\tau)(n,t)\colon \{n=\nu,\ t=\tau\}$$
$$q(\sigma)(x) = q(\nu,\tau)(n,t)\colon \{n=\nu,\ t=\tau+2^{\nu}-1\}$$

$$f(\sigma) = f(\nu,\tau) = (\nu-1,\tau)$$
$$g(\sigma) = g(\nu,\tau) = (\nu-1,\tau+2^{\nu-1}).$$

We easily see that these choices satisfy our requirements, i.e., that

1. $\{n=\nu,\ t=\tau\}[n>0 \mid n:=n-1] \subseteq [n>0 \mid n:=n-1]\{n=\nu-1,\ t=\tau\}$
2. $\{n=\nu-1,\ t=\tau+2^{\nu-1}-1\}[t:=t+1] \subseteq [t:=t+1]\{n=\nu-1,\ t=\tau+2^{\nu-1}\}$
3. $\{n=\nu-1,\ t=\tau+2^{\nu-1}+2^{\nu-1}-1\}[n:=n+1] \subseteq [n:=n+1]\{n=\nu,\ t=\tau+2^{\nu}-1\}$
4. $\{n=\nu,\ t=\tau\}[n=0] \subseteq [n=0]\{n=\nu,\ t=\tau+2^{\nu}-1\}$.

From this we conclude that $p(\sigma);P \subseteq P;q(\sigma)$, i.e., that $\{n=\nu,\ t=\tau\};P \subseteq P;\{n=\nu,\ t=\tau+2^{\nu}-1\}$, as was to be demonstrated. □

5.3. Hoare's while statement axiom

In [16], HOARE has proposed an axiomatic formulation of the FLOYD method. As basic formal construct he uses $\{p\}P\{q\}$, which is another way of writing $p;P \subseteq P;q$. Various axioms and proof rules are then given, depending upon the form of the P. E.g., for P an assignment statement, x: = e, say, HOARE's axiom (H_a) is: $\{p[x/e]\}$ x: = e$\{p\}$, where p[x/e] denotes the result of substituting the expression e for all occurrences of x in p. (Complications arise in the definition of substitution when x is a subscripted variable. Treatment of this is omitted here (and in most other places the method is presented as well!)). As rule for composition (H_c) we have: if $\{p\}P_1\{q\}$ and $\{q\}P_2\{r\}$, then $\{p\}P_1;P_2\{r\}$. We shall be concerned in particular with HOARE's while statement axiom (H_w) which reads as follows: If $\{p\wedge u\}S\{u\}$, then $\{u\}p^*S\{\neg p\wedge u\}$. In words, if S leaves property u invariant (under the additional assumption that p holds), then p^*S also leaves u invariant. Moreover, p is always false upon exit from the while statement.

(For the uninitiated reader, application of the system to FLOYD's summation example may be helpful. We want to show that $\{n\geq 0\}s:=0;i:=1;$ **while** $i \leq n$ **do** $(s:=s+a_i;i:=i+1)\{s = \sum_{j=1}^{n} a_j\}$. The main step is the application of H_w as follows:

Choose for u: $\{s = \sum_{j=1}^{i-1} a_j,\ i \leq n+1\}$. For p we have: $i \leq n$.

We verify whether $\underbrace{\{s = \sum_{j=1}^{i-1} a_j,\ i \leq n+1,}_{u}\ \underbrace{i \leq n\}}_{p}S\{s = \underbrace{\sum_{j=1}^{i-1} a_j,\ i \leq n+1\}}_{u}$

By H_a, $\{s = \sum_{j=1}^{i} a_j,\ i+1 \leq n+1\}i:=i+1\{s = \sum_{j=1}^{i-1} a_j,\ i \leq n+1\}$ and

$$\{s+a_i = \sum_{j=1}^{i} a_j,\ i+1 \leq n+1\}s:=s+a_i\{s = \sum_{j=1}^{i} a_j,\ i+1 \leq n+1\}.$$

The desired result then follows by H_c. Filling in the further details of the proof is omitted.)

Relationally, H_w is written as

(5.11) $\quad \forall u[u;p;S \subseteq p;S;u \Rightarrow u;p^*S \subseteq p^*S;\bar{p};u]$

The question now arises whether (5.11) is a complete characterization of the while statement, i.e., whether the following holds: Let X be any relation satisfying

(5.12) $\quad \forall u[u;p;S \subseteq p;S;u \Rightarrow u;X \subseteq X;\bar{p};u]$

Is $X = p^*S$? The answer is *no*, as can be seen by taking e.g. $X = \Omega$. However, we do have that $X \subseteq p^*S$. This is proved as follows [1)]

1. First we show that the following holds:
   If $\forall u[u;X \subseteq X;u \Rightarrow u;Y \subseteq Y;u]$ then $Y \subseteq X^*$.
   Proof: Take x_0 fixed, and define $u_0(x) \leftrightarrow x_0X^*x$. It is direct that $u_0;X \subseteq X;u_0$; hence, $u_0;Y \subseteq Y;u_0$, i.e., $\forall x,y[x_0X^*x \wedge xYy \rightarrow x_0X^*y]$. This implies that $\forall y[x_0Yy \rightarrow x_0X^*y]$. Since x_0 was arbitrary, $Y \subseteq X^*$ follows.
2. Assume (5.12). Since $X;\bar{p};u \subseteq X;u$, application of part 1 yields that $X \subseteq (p;S)^*$. Taking $u \equiv I$ in (5.12) yields that $X \subseteq X;\bar{p}$. Hence, $X \subseteq X;\bar{p} \subseteq (p;S)^*;\bar{p} = p^*S$. □

This result settles the question as to the precise status of HOARE's

---

1) This proof is due to SCOTT.

axiom: In itself, it does not give the whole truth about the while statement. In particular, it is not a consequence of the axiom that $p^*S = p;S;p^*S \cup \bar{p}$. However, taken together with the f.p.p., it does characterize the while statement, i.e., for each X which satisfies

a. $X = p;S;X \cup \bar{p}$

b. $\forall u[u;p;S \subseteq p;S;u \Rightarrow u;X \subseteq X;\bar{p};u]$

we have $X = p^*S$.

The next question which arises is whether (5.11) may be strengthened in such way that a complete characterization is obtained. This is easy to answer on the basis of the results of the preceding section. In fact, we have: Let X satisfy

$$\forall p,q\left[p;X \subseteq X;q \iff \exists s\left[\begin{array}{c} p \subseteq s \\ s;r;S \subseteq r;S;s \\ s;\bar{r} \subseteq \bar{r};q \end{array}\right]\right]$$

Then $X = r^*S$

By (5.4) and (5.5) it is sufficient to show: Assume

$$\forall p,q[p;X \subseteq X;q \iff p;r^*S \subseteq r^*S;q]$$

Then $X = r^*S$.

Now this result is nothing but a simple consequence from the fact that $X \subseteq Y \iff \forall p,q[p;Y \subseteq Y;q \Rightarrow p;X \subseteq X;q]$, the proof of which is immediate by taking, for fixed x_0, $p(x) \leftrightarrow x = x_0$, and $q(x) \leftrightarrow x_0 Y x$. □

5.4. A "theorem" due to Dijkstra

In this section we try to provide an interpretation to the ideas developed in a recent paper by DIJKSTRA [13]. It will turn out that a corrected version of the main result of that paper is immediately obtained by an application of Scott's induction.

DIJKSTRA also takes (HOARE's formulation of) partial correctness as a starting point. Consider once more (5.2):

$$p;P \subseteq P;q$$

Now, quoting from [13]: "We consider the semantics of a program P fully determined when we can derive for any postcondition q to be satisfied by the final state, the weakest precondition that for this purpose should be satisfied by the initial state. We regard this weakest precondition as a function of the postcondition q and denote it by $f_P(q)$,".

This suggests to us that what is meant here is that $f_P(q) = P \to q = \cup\{p \mid p;P \subseteq P;q\}$. The use of the function f_P (called a "predicate transformer" by DIJKSTRA) in the paper furthermore seems to imply that satisfaction of $f_P(q)$ guarantees termination, i.e., that $f_P(q)$ should be taken as $f_P(q) = (I \circ \breve{P}) \cap (P \to q)$ (or, $f_P(q)(x) \leftrightarrow \exists y[xPy] \wedge \forall z[xPz \to q(z)]$), or, equivalently, that $f_P(q) = (q \circ \breve{P}) \cap (P \to q)$. The addition of the requirement of termination is in particular motivated by DIJKSTRA's "law of excluded miracle", which is his way of referring to the fact that $f_P(\Omega) = \Omega$. Observe that, for P a not-everywhere defined program, $(P \to \Omega) \neq \Omega$, and we see that the interpretation $f_P(q) = P \to q$ fails. DIJKSTRA also imposes the restriction that P be a function, in which case $f_P(q)$ reduces to $q \circ \breve{P}$, as can easily be checked by the reader. With this interpretation, the axioms in [13] become provable. E.g., the first four of them are the first halves of lemma 5.2, parts 1, 4, 7, 8, 9.

Next, we look at the main result from [13], which DIJKSTRA has baptized as "Fundamental Invariance Theorem for Recursive Procedures". We again quote from [13]: Consider a text, called H", of the form

H": ... H' ... H' ... H' ...

to which corresponds a predicate transformer $f_{H"}$ such that for a specific pair of predicates q and r, the assumption $q \subseteq f_{H'}(r)$ is a sufficient assumption about $f_{H'}$ for proving $q \subseteq f_{H"}(r)$. In that case the recursive procedure H given by

proc H; ... H ... H ... H ...

enjoys the property that

$q \cap f_H(I) \subseteq f_H(r)$."

First we observe that, as stated, this theorem is incorrect. Choose $q = I$, $r = \Omega$. Then the hypothesis reduces to: $I \subseteq f_{H'}(\Omega)$ is a sufficient assumption to prove $I \subseteq f_{H''}(\Omega)$, or, by the "law of excluded miracle", $I \subseteq \Omega$ is a sufficient assumption to prove $I \subseteq \Omega$. This is clearly satisfied, and we infer that $I \cap f_H(I) \subseteq f_H(\Omega)$, i.e., $f_H(I) = \Omega$. This is nothing but the assertion that $\forall x \neg \exists y[xHy]$, i.e., H is nowhere defined. Since H was an arbitrary procedure, we have derived a contradiction.

It is not difficult, however, to remedy the situation. The corrected version is: Assume

a. If $q \cap f_{H'}(I) \subseteq f_{H'}(r)$ then $q \cap f_{H''}(I) \subseteq f_{H''}(r)$.

Then we may conclude that

b. $q \cap f_H(I) \subseteq f_H(r)$.

This may be seen as follows: Let us rewrite $q \cap f_H(I) \subseteq f_H(r)$:

$$\forall x[q(x) \wedge f_H(I)(x) \rightarrow f_H(r)(x)] \quad , \text{ or}$$

$$\forall x[q(x) \wedge \exists y[y\breve{H}x] \rightarrow \exists t[r(t) \wedge t\breve{H}x]] \quad , \text{ or}$$

$$\forall x,y[q(x) \wedge xHy \rightarrow \exists t[xHt \wedge r(t)]] \quad , \text{ or}$$

$$\forall x,y[q(x) \wedge xHy \rightarrow r(y)],$$

where the last step follows since H is a function. Thus, we see that $q \cap f_H(I) \subseteq f_H(r)$ is nothing but a complicated way of writing $q;H \subseteq H;r$ Thus, the theorem obtains the form: Assume

a. If $q;H' \subseteq H';r$ then $q;H'' \subseteq H'';r$.

Then we may conclude that

b. $q;H \subseteq H;r$.

Since H" = text (H') = T(H'), say, and for H we have the declaration proc H;T(H), we finally see that the theorem is a direct consequence of Scott's induction. □

6. EXERCISES

6.1. (The $\circ$-operator with while statements [2,4])

a. Show that $p \circ (q*A) = \bar{q} \cap \mu X[p \cup (q \cap X) \circ A]$.

b. Assume $p \cup \bar{p} = I$, $p \circ A_i \subseteq p_i$, $i=1,2$. Is it true that $q*(p;A_1 \cup \bar{p};A_2) = p;q*A_1 \cup \bar{p};q*A_2$?

c. Let $P \circ p := p \circ \breve{P}$.

Assume $p \cup \bar{p} = I$, $A \circ p \subseteq p$. Show that $(q*A) \circ p \subseteq ((p \wedge q)*A) \circ p$, $(q*A) \circ \bar{p} \subseteq ((p \wedge q)*A) \circ \bar{p}$.

6.2 (Extinction of relations [3])

Let, for any R, $R^{\dagger}$ be defined by

$$R^{\dagger} = (I \circ \breve{R})*R$$

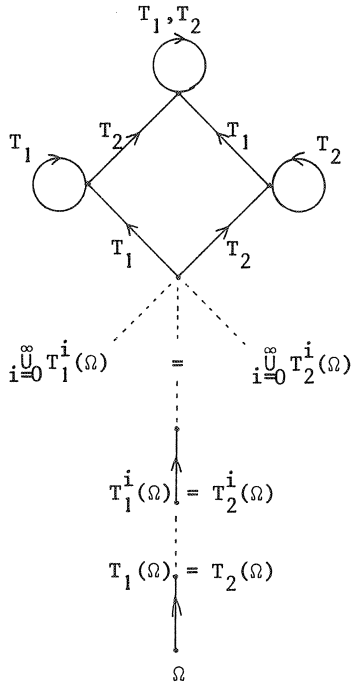
i.e., $R^{\dagger}$ denotes the result of iterating R as long as it is defined. E.g., when R is the relation of "direct descendant" in a tree, $R^{\dagger}$ connects the root of the tree with all its leaves.

a. Show by an example that $R, R^{\dagger}, R^{\dagger\dagger}$ and $R^{\dagger\dagger\dagger}$ may all be different.

b. Show that, for all R, $R^{\dagger\dagger\dagger\dagger} = R^{\dagger\dagger}$.

6.3 (Continuity vs. monotonicity, cf. 4.4 (VUILLEMIN))

Let T_1, T_2 satisfy the properties as suggested by the following picture:



Derive from this picture that for T_1, T_2 satisfying

1. $T_1(\Omega) = T_2(\Omega)$,
2. $T_1(T_2(X)) = T_2(T_1(X))$, for all X,
3. T_1, T_2 monotonic,

it is not necessarily the case that $\mu X[T_1(X)] = \mu X[T_2(X)]$.

6.4 (Greatest fixed points)

a. (PARK). Let T be monotonic. Let $\bar{}$ denote complementation with respect to U. Prove

$$\overline{\mu X[T(\bar{X})]} = \nu X[T(X)]$$

where $\nu X[T(X)]$ is the greatest fixed point of T, i.e., $\nu X[T(X)] = \cup\{X : X = T(X)\}$.

b. ([15] and MAZURKIEWICZ). Let R be any relation, and let p be defined as

$$p = \cup\{q \mid q = q \circ \breve{R}\}.$$

Show that p(x) holds iff there exists an infinite sequence $x = x_0, x_1, x_2, \ldots$, such that $x_0 \, Rx_1 \, Rx_2 \ldots$.

c. Interpret $\mu X[R \mapsto X]$ for any R.

6.5 (Axiomatization of the natural numbers)

Let S (successor) be a relation satisfying

1. $S;\breve{S} = \underline{I}$. 2. $\breve{S};S \subseteq I$. 3. $U \subseteq \breve{S}^{\dagger};S^*$

Put $p_0 = \overline{\breve{S};S} \cap I$ († and $\bar{}$ as in 6.2 and 6.4). Prove

a. $p_0;U \cap U;p_0 \subseteq p_0$.

b. $I = \mu X[p_0 \cup \breve{S};X;S]$.

c. Let F be any function satisfying $p_0;F = p_0;A_1$ and $S;F = F;A_2$. Then $F = \mu X[p_0;A_1 \cup \breve{S};X;A_2]$.

6.6 (Applications in formal language theory)

Let Σ be any alfabet, Σ^* and ϵ as usual, $a, b, a_1, \ldots, a_n, b_1, \ldots, b_m \in \Sigma$. Let, for $A, B \subseteq \Sigma^*$, $AB = \{wx \mid w \in A, x \in B\}$, let $aB = \{a\}B$, etc. Let, for T any monotonic function from 2^{Σ^*} to 2^{Σ^*}, $\mu X[T(X)]$ be the least subset of Σ^* satisfying $X = T(X)$.

a. Let, in standard notation, $G = (\{S\},\Sigma,\{S\to aSb, S\to\varepsilon\},S)$. Show that $L(G) = \mu X[aXb \cup \varepsilon]$.

b. Let $a^* = \mu X[aX \cup \varepsilon]$. Prove that $\{a,b\}^* = a^*(ba^*)^*$. (cf. 4.1).

c. Let

$$G_1 = (\{S\},\Sigma,\left\{\begin{matrix} S\to b_1,\ldots,S\to b_m \\ S\to a_1,\ldots,S\to a_n \end{matrix}\right\},S)$$

$$G_2 = (\{S,T\},\Sigma,\left\{\begin{matrix} S\to b_1,\ldots,S\to b_m \\ S\to b_1T,\ldots,S\to b_mT \\ T\to a_1,\ldots,T\to a_n \\ T\to a_1T,\ldots,T\to a_nT \end{matrix}\right\},S)$$

Show that $L(G_1) = L(G_2)$ using fixed point techniques (cf. 5.2; this result is used e.g. in the usual proof of the Greibach Normal Form theorem).

d. Let a^* be as in part b, and $a^+ = a^*a$. Prove, using the notation of 6.4 referring to a universe consisting of all finite *and infinite* sequences over $\Sigma = \{a,b\}$:

$$\nu X[a^*b^+aX] = \nu X[b^*a^+bX].$$

REFERENCES

[1] ASHCROFT, E.A., Z. MANNA & A. PNUELI, *Decidable properties of monadic functional schemes*, J.ACM, 20 (1973) 488-499.

[2] DE BAKKER, J.W., *Recursive Procedures*, Mathematical Centre Tracts 24, Mathematisch Centrum, Amsterdam, 1971.

[3] DE BAKKER, J.W. & L.G.L.T. MEERTENS, *On the completeness of the inductive assertion method*, to appear in Journal of Comp. Syst. Sc.

[4] DE BAKKER, J.W. & W.P. DE ROEVER, *A calculus for recursive program schemes*, in *Automata, Languages and Programming* (M. Nivat, ed.), p. 167-196, North-Holland, Amsterdam, 1973.

[5] BEKIĆ, H., *Definable operations in general algebra, and the theory of automata and flowcharts*, Report IBM Laboratory Vienna, 1969.

[6] BLIKLE, A., *An algebraic approach to the mathematical theory of programs*, CC PAS Report 119, Warsaw, 1973.

[7] BLIKLE, A. & A. MAZURKIEWICZ, *An algebraic approach to the theory of programs, algorithms, languages and recursiveness*, in Proc. of an International Symposium and Summer School on the Mathematical Foundations of Computer Science, Warsaw-Jablonna, 1972.

[8] BURSTALL, R.M., *Proving properties of programs by structural induction*, Computer J., 12, (1969) 41-48.

[9] CADIOU, J.M., *Recursive definitions of partial functions and their computations*, Memo AIM-163, Stanford University, 1972.

[10] CADIOU, J.M. & Z. MANNA, *Recursive definitions of partial functions and their computations*, in Proc. of an ACM Conference on Proving Assertions about Programs, p. 58-65, ACM, 1972.

[11] COOPER, D.C., *On the equivalence of certain computations*, Computer J., 9 (1966) 45-52.

[12] COURCELLE, B., G. KAHN & J. VUILLEMIN, *Algorithmes d'équivalence et de réduction à des expressions minimales dans une classe d'equations récursives simples*, to appear in Proc. 2nd. Colloquium on Automata, Languages and Programming, Springer Lecture Notes in Computer Science, 1974.

[13] DIJKSTRA, E.W., *A simple axiomatic basis for programming language constructs*, Indagationes Mathematicae, 36 (1974) 1-15.

[14] FLOYD, R.W., *Assigning meanings to programs*, in Proc. of a Symposium in Applied Mathematics Vol. 19-Math. Aspects of Computer Science (J.T. Schwartz, ed.), p. 19-32, 1967.

[15] HITCHCOCK, P. & D.M.R. PARK, *Induction rules and proofs of program termination*, in Automata, Languages and Programming (M. Nivat, ed.), p. 225-251, North-Holland, Amsterdam, 1973.

[16] HOARE, C.A.R., *An axiomatic basis for computer programming*, C. ACM, 12 (1969) 576-580.

[17] KLEENE, S.C., *Introduction to Metamathematics*, North-Holland, Amsterdam, 1952.

[18] MANNA, Z., *Mathematical theory of partial correctness*, in Symposium on Semantics of Algorithmic Languages (E. Engeler, ed.), p. 252-269, Lecture Notes in Mathematics, Vol. 188, Springer, Berlin, 1971.

[19] MANNA, Z., *Introduction to the Mathematical Theory of Computation*, McGraw-Hill, 1974.

[20] MANNA, Z., S. NESS & J. VUILLEMIN, *Inductive methods for proving properties of programs*, C. ACM, 16 (1973) 491-502.

[21] MANNA, Z., & J. Vuillemin, *Fixpoint approach to the theory of computation*, C. ACM, 15 (1972) 528-536.

[22] MCCARTHY, J., *A basis for a mathematical theory of computation*, in *Computer Programming and Formal Systems*, p. 33-70 (P. Braffort & D. Hirschberg, eds.), North-Holland, Amsterdam, 1963.

[23] MILNER, R., *Implementation and applications of Scott's logic for computable functions*, in Proc. of an ACM Conference on Proving Assertions about Programs, p. 1-6, ACM, 1972.

[24] MILNER, R., *Logic for computable functions, description of a machine implementation*, Memo AIM-169, Stanford University, 1972.

[25] MORRIS, J.H., *Another recursion induction principle*, C. ACM, 14, (1971) 351-354.

[26] PARK, D., *Fixpoint induction and proof of program semantics*, in *Machine Intelligence*, Vol. 5, (B. Meltzer & D. Michie, eds.), p. 59-78, Edinburgh University Press, Edingburgh, 1970.

[27] DE ROEVER, W.P., *Operational, mathematical and axiomatized semantics for recursive procedures and data structures*, Report ID 1/74, Mathematisch Centrum, Amsterdam, 1974.

[28] TARSKI, A., *A lattice theoretical fixpoint theorem and its applications*, Pacific J. of Math., 5 (1955) 285-309.

[29] SCOTT, D. & J.W. DE BAKKER, *A theory of programs*, unpublished notes, IBM Seminar, Vienna, 1969.

[30] VUILLEMIN, J., *Proof techniques for recursive programs*, IRIA Report, 1973.

[31] (added in proof) DE BAKKER, J.W., *Least fixed points revisited*, Report IW 22/74, Mathematisch Centrum, Amsterdam, 1974.

ALGORITHMIC LOGIC by E. ENGELER

MATHEMATICAL CENTRE TRACTS 63, 1975, 57-85

ALGORITHMIC LOGIC

E. ENGELER

Eidgenössische Technische Hochschule, Zürich, (CH)

Algorithmic logic, like so many mathematical concepts, is perhaps best defined by a closure operation. Let me indicate the first few steps in the iteration:

- Methods of formal logic applied to study basic concepts and problems about algorithms, (meaning, correctness of programs, proofs of existence of functions viz. construction of algorithms, etc.).
- Algorithmic study of methods in logic, (decision procedures and their complexity, simplification of Boolean expressions, etc.).
- Algorithmic study of methods of logic to study algorithms, (automatic proofs of correctness of programs, automatic program generation, etc.).

etc.

The overriding aspect of algorithmic logic, thus conceived, is the stress put on the formal manipulative component: the concept of an algorithm as a formula, linked with other means of formal expression in a formal logical system. Such a formal system should ideally be powerful enough for the three main modes of employ for a formal system:

- unambiguous expression of the relevant notions (for example: properties of programs such as termination, equivalence, partial correctness),
- availability of a formal proof system
- treatment of metatheoretic questions about the formal system (e.g. limits of realizability of proof procedures on a computer, relations to other systems of logic).

It is clear, that as goals become more definite, choices of the formal systems become more important and more varied. For the present exposition we select only a small sampling of algorithmic problems, chosen in order to make the presentation of the formal system particularly simple, but still representative enough to show the spirit of algorithmic logic.

1. FORMULAS AND THEIR MEANING

We are all familiar with the fact that, as mathematical scientists, we can at once recognize the meaning of a given formula. For example:

Formula	Meaning
$(x^2-z \sin y)/(y-x)$	partial function $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$
$(x>0) \to (3x-x^3y>x^2-2xy)$	Boolean function $\mathbb{R} \times \mathbb{R} \to \mathbb{B}$
`var x,y: integer;`	
`begin y := 0;`	partial function $\mathbb{N} \times \mathbb{N} \to \mathbb{N} \times \mathbb{N}$.
`while sqr(y) ≠ x do y := y+1`	
`end.`	

These examples are taken from formal languages, and our ability to determine their meanings depends on the fact that formulas can be uniquely decomposed in components whose meaning in turn determines uniquely the meaning of the composite formula.

Let us consider a simple example, namely the language first-order predicate logic for the natural numbers

$$N = \langle N,+,\cdot,=,0,1\rangle,$$

to be denoted by

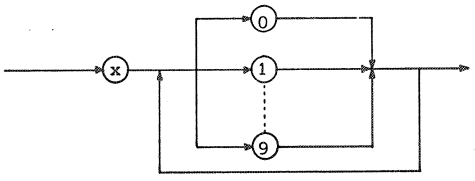
$$L(+,\cdot,=,0,1).$$

The set of formulas is determined by a contextfree grammar on the terminal alphabet

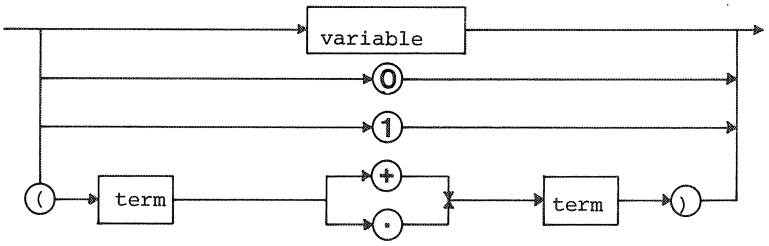
$$(,),\wedge,\vee,\neg,\exists,\forall,0,1,+,\cdot,=,\times,0,1,\ldots,9.$$

This grammar is best presented by its syntax diagram

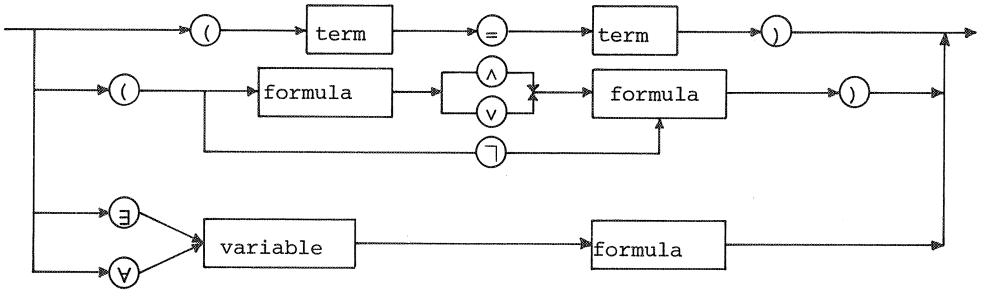
variable:



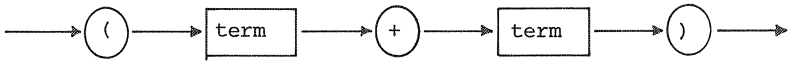
term:



formula:



To each diagram corresponds a set of expressions, called a syntactical category (*variables*, *terms*, *formulas*). We may consider each line of a diagram involving boxes as defining an operation on the syntactical categories. E.g. the line



associates to any pair τ_1, τ_2 of elements of *terms* another element of *terms*

$$O_2(\tau_1, \tau_2) = (\tau_1 + \tau_2),$$

where the right-hand side is the string of symbols obtained by concatenating the symbols "(", ")", "+", and the strings τ_1 and τ_2 in the order indicated.

Not all expressions in L(+,·,=,0,1) are composite by means of one of the eight operations associated to the grammar. Such expressions are called *atoms*; in the present case the terms 0 and 1 and all variables are atoms.

We are now ready to state the two basic properties of the language which will allow us to define semantics:

(i) (Induction axiom).
L(+,·,=,0,1) is the closure of the set of atoms under the syntactical operations $0_1,\ldots,0_8$.

(ii) (Unique readability axiom).
Each composite expression determines uniquely an operation 0_i and component expressions out of which it is composed.
{Abstractly: if $a = 0_i(a_1,\ldots,a_{n_i})$, $b = 0_j(b_1,\ldots,b_{n_j})$ are composite elements and $a = b$ then $i = j$ and $a_1 = b_1$, $a_2 = b_2,\ldots,a_{n_i} = b_{n_i}$.}

These two axioms make L(+,·,=,0,1) a uniquely readable (or "structured") language; in the terminology of contextfree grammars one would say that the grammar given by the diagrams is unambiguous. Their importance lies in the fact that they allow us, quite generally, to define mappings

$$S: L(+,\cdot,=,0,1) \to M$$

into any non-empty set M by recursion. To determine S it is sufficient to prescribe its effect on the atoms and its behaviour with respect to compositions $0_1,\ldots,0_8$: Let $M \neq \emptyset$ and let $p_1,\ldots,p_8$ be operations on M of the same arity as the operations $0_1,\ldots,0_8$.

THEOREM. *For any function* S_0: atoms $\to$ M *there exists a unique extension* $S: L(+,\cdot,=,0,1) \to M$ *such that* $S(0_i(a_1,\ldots,a_{n_i})) = p_i(S(a_1),\ldots,S(a_{n_i}))$ *for all* i *and* $a_1,\ldots,a_{n_i}$.

The proof of this theorem generalizes the well-known proof of the existence of recursively defined number-theoretic functions; we leave it to the reader. Instead, we illustrate the use of this theorem for the introduction of semantics for L(+,·,=,0,1). {For notational convenience we restrict the language to finitely many variables, say $x_1,\ldots,x_n$. Also, we write, as customary in mathematics $x_1,x_2,\ldots$ instead of $\times1,\times2,\ldots,\times52,\ldots$ as prescribed by the grammar.} Since *terms* $\cap$ *formulas* $= \emptyset$, we may treat the definition of S on these two syntactical categories disjointly, and propose

$$S_T: \text{terms} \to (\text{set of functions } N^n \to N),$$

$$S_F: \text{formulas} \to (\text{set of functions } N^n \to \{\underline{\text{true}},\underline{\text{false}}\}).$$

The details of the definitions of S_T and S_F are obvious from the intended meaning:

$S_T(x_i)[a_1,\ldots,a_n] = a_i$;

$S_T(0)[a_1,\ldots,a_n] = 0$;

$S_T(1)[a_1,\ldots,a_n] = 1$;

$S_T((\tau_1+\tau_2))[a_1,\ldots,a_n] = S_T(\tau_1)[a_1,\ldots,a_n] + S_T(\tau_2)[a_1,\ldots,a_n]$
where + on the right-hand side designates addition in N;

$S_T((\tau_1,\tau_2))[a_1,\ldots,a_n] = S_T(\tau_1)[a_1,\ldots,a_n] \cdot S_T(\tau)[a_1,\ldots,a_n]$.

$$S_F((\tau_1=\tau_2)) = \begin{cases} \underline{\text{true}} \text{ if } S_T(\tau_1)[a_1,\ldots,a_n] = S_T(\tau_2)[a_1,\ldots,a_n], \\ \underline{\text{false}} \text{ otherwise;} \end{cases}$$

$S_F((\phi\wedge\psi))[a_1,\ldots,a_n] = S_F(\phi)[a_1,\ldots,a_n] \wedge S_F(\psi)[a_1,\ldots,a_n]$,
where $\wedge$ on the right-hand side designates the Boolean "and";

$S_F((\phi\vee\psi))[a_1,\ldots,a_n] = S_F(\phi)[a_1,\ldots,a_n] \vee S_F(\psi)[a_1,\ldots,a_n]$;

$S_F((\neg\phi))[a_1,\ldots,a_n] = \neg S_F(\phi)[a_1,\ldots,a_n]$;

$$S_F(\exists x_i\phi)[a_1,\ldots,a_n] = \begin{cases} \underline{\text{true}} \text{ if } S_F(\phi)[a_1,\ldots,a_{i-1},m,a_{i+1},\ldots,a_n] \\ \qquad = \underline{\text{true}} \text{ for some } m \in N, \\ \underline{\text{false}} \text{ otherwise;} \end{cases}$$

$$S_F(\forall x_i\phi)[a_1,\ldots,a_n] = \begin{cases} \underline{\text{true}} \text{ if } S_F(\phi)[a_1,\ldots,a_{i-1},m,a_{i+1},\ldots,a_n] \\ \qquad = \underline{\text{true}} \text{ for all } m \in N, \\ \underline{\text{false}} \text{ otherwise.} \end{cases}$$

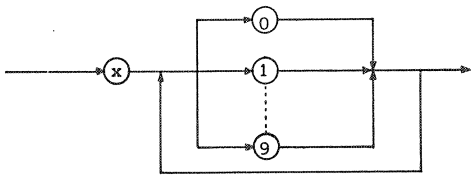
{The semantic function S_F allows us to define the concept of "a formula ϕ without free variables holds in N" by

$$N \models \phi \quad \text{iff} \quad S(\phi)[a_1,\ldots,a_n] = \text{true for all } a_1,\ldots,a_n.$$

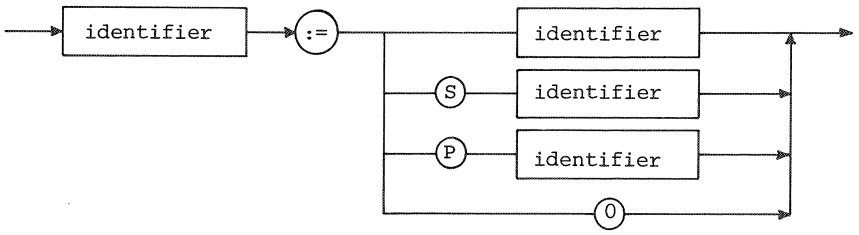
We shall use this abbreviation in sections below.}

As a second example of a structured language we introduce a rudimentary programming language PASIC.

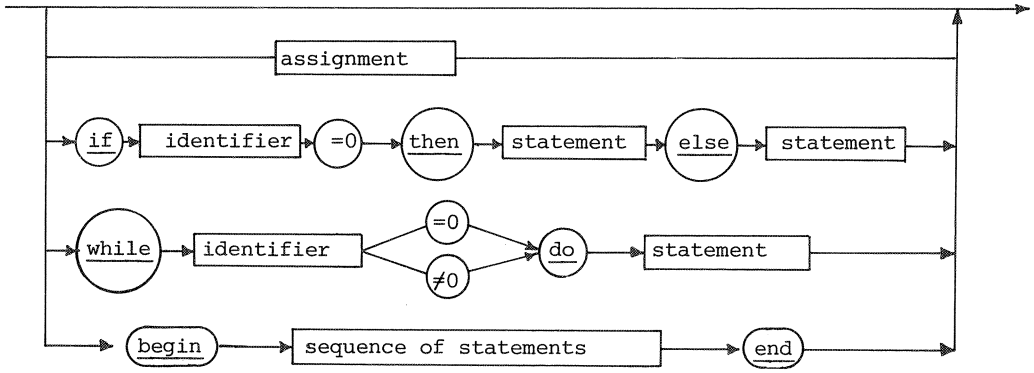
identifier:



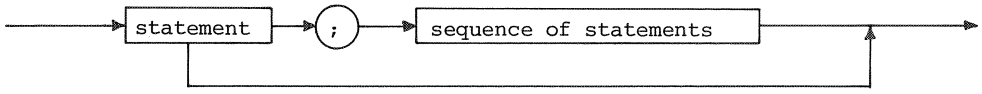
assignment:



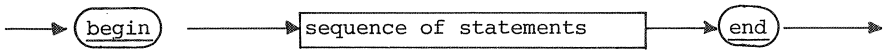
statement:



sequence of statements:



program:



The reader observes that this language is a small fragment of PASCAL. For the immediate expository purposes at hand it is however expressive enough. In particular, it can easily be shown that PASIC programs suffice to compute all partial recursive functions.

Indeed, "the function computed by the program π" is the first type of meaning, semantics, that we associate to PASIC programs. Thus, let π be a PASIC program in which all identifiers that occur are among $x_1,\ldots,x_n$.

$$S\colon \text{PASIC} \to (\text{set of partial functions } N^n \to N^n)$$

is defined recursively as follows according to the three syntactical

categories *assignments, statements, sequence of statements.*

$S_A(x_i := x_j)[a_1,\ldots,a_n] = [a_1,\ldots,a_{i-1},a_j,a_{i+1},\ldots,a_n]$;

$S_A(x_i := Sx_j)[a_1,\ldots,a_n] = [a_1,\ldots,a_{i-1},a_j+1,a_{i+1},\ldots,a_n]$;

$S_A(x_i := Px_j)[a_1,\ldots,a_n] = [a_1,\ldots,a_{i-1},a_j-1,a_{i+1},\ldots,a_n]$;

$S_A(x_i := 0)[a_1,\ldots,a_n] = [a_1,\ldots,a_{i-1},0,a_{i+1},\ldots,a_n]$.

$S_{St}(\ )[a_1,\ldots,a_n] = [a_1,\ldots,a_n]$;

$S_{St}(\alpha)[a_1,\ldots,a_n] = S_A(\alpha)[a_1,\ldots,a_n]$, where α is any assignment;

$$S_{St}(\underline{\text{if}}\ x_i = 0\ \underline{\text{then}}\ \pi_1\ \underline{\text{else}}\ \pi_2)[a_1,\ldots,a_n] = \begin{cases} S_{St}(\pi_1)[a_1,\ldots,a_n] \text{ if } a_i = 0, \\ S_{St}(\pi_2)[a_1,\ldots,a_n] \text{ otherwise;} \end{cases}$$

$$S_{St}(\underline{\text{while}}\ x_i = 0\ \underline{\text{do}}\ \pi)[a_1,\ldots,a_n] = \begin{cases} (S_{St}(\pi)\circ S_{St}(\underline{\text{while}}\ x_i = 0\ \underline{\text{do}}\ \pi))[a_1,\ldots,a_n] \text{ if } a_i = 0, \\ [a_1,\ldots,a_n] \text{ otherwise;} \end{cases}$$

$$S_{St}(\underline{\text{while}}\ x_i \neq 0\ \underline{\text{do}}\ \pi)[a_1,\ldots,a_n] = \begin{cases} (S_{St}(\pi)\circ S_{St}(\underline{\text{while}}\ x_i = 0\ \underline{\text{do}}\ \pi))[a_1,\ldots,a_n] \text{ if } a_i \neq 0, \\ [a_1,\ldots,a_n] \text{ otherwise;} \end{cases}$$

$S_{St}(\underline{\text{begin}}\ \Sigma\ \underline{\text{end}})[a_1,\ldots,a_n] = S_{SS}(\Sigma)[a_1,\ldots,a_n]$.

{In the four recursive definitions above we have used the symbol $\circ$ for composition of functions; π, π_1, π_2 are statements, Σ is a sequence of statements.}

$S_{SS}(\pi)[a_1,\ldots,a_n] = S_{St}(\pi)[a_1,\ldots,a_n]$;

$S_{SS}(\pi;\Sigma)[a_1,\ldots,a_n] = (S_{St}(\pi)\circ S_{SS}(\Sigma))[a_1,\ldots,a_n]$.

Finally, we define S on PASIC programs by

$$S(\underline{\text{begin}}\ \Sigma\ \underline{\text{end}})[a_1,\ldots,a_n] = S_{SS}(\Sigma)[a_1,\ldots,a_n].$$

2. EXPRESSING PROPERTIES OF PROGRAMS IN INFINITARY LANGUAGES

As we will show in section 3, the first-order language L(+,·,=,0,1) is quite sufficient to express all that may be desirable to say about PASIC programs run on the natural numbers. Since first-order logic is a good enough tool to work with, all is well. However, the need to extend the

logical frame of first-order arises as soon as PASIC is generalized to formulate programs that operate on other types of data. Indeed, we shall even prove that some extension is necessary.

To appreciate the more general setting for PASIC, assume that we have an arbitrary relational structure

$$\underline{A} = \langle A, R_1, \ldots, R_n, f_1, \ldots, f_m \rangle$$

where $A \neq \emptyset$, R_i are relations on A : $R \subseteq A^{n_i}$, $i=1,\ldots,n$, f_j are operations on A : $f_j : A^{m_j} \to A$, $j=1,\ldots,m$. Assume furthermore that we are in possession of some devices, oracles as it where, which allow us to effect the decisions corresponding to the relations R_i and the operations corresponding to the operation f_j, just as for the natural numbers

$$\underline{N} = \langle N, =0, S, P, 0 \rangle$$

we assumed the executability of the basic tests

$$x_i = 0; \quad x_i \neq 0,$$

and basic assignments

$$x_i := Sx_j; \quad x_i := Px_j; \quad x_i := 0.$$

In other words, we make the obvious replacements in the syntax diagram of PASIC with

$$R_i(x_{K_1}, \ldots, x_{K_{m_i}}); \quad \neg R_i(x_{K_1}, \ldots, x_{K_{m_i}}), \quad i=1,\ldots,n,$$

respectively

$$x_i := f_j(x_{K_1}, \ldots, x_{K_{n_i}}), \quad j=1,\ldots,m.$$

Let us first specify what kinds of properties of programs we envision for our - still to be determined formalism - to express.

Termination

Let π be a PASIC program for $\underline{A}$ with identifiers $x_1,\ldots,x_n$.
$\mathrm{Term}_\pi(x_1,\ldots,x_n)$ is the formula, which should express that π terminates on input $x_1,\ldots,x_n$. Thus

$$\underline{A} \models \forall x_1 \forall x_2 \ \ldots \ \forall x_n \ \mathrm{Term}_\pi(x_1,\ldots,x_n)$$

is true if in $\underline{A}$ the program π terminates for all inputs. {We shall from now on use the abbreviation $\vec{x}$ for $x_1\ldots x_n$.}

Transduction

Let $\mathrm{Trans}_\pi(x_1,\ldots,x_n;\ y_1,\ldots,y_n)$ express that π, if it terminates on input $x_1,\ldots,x_n$ at all, will terminate with values $y_1,\ldots,y_n$ assigned to the identifiers.

Strong equivalence

Using the transduction formula, we may express that π_1 and π_2 are equivalent by

$$\underline{A} \models \forall\vec{x}\forall\vec{y}(\mathrm{Trans}_{\pi_1}(\vec{x},\vec{y}) \Leftrightarrow \mathrm{Trans}_{\pi_2}(\vec{x},\vec{y})).$$

Partial correctness

Given that the values assigned to the identifiers $x_1,\ldots,x_n$ at input time satisfy the formula $\phi(x_1,\ldots,x_n)$ and assuming that π terminates on this input, do the values assigned to these identifiers at output time satisfy the predicate ψ? This question is formulated by

$$\underline{A} \models \forall\vec{x}\forall\vec{y}(\phi(\vec{x}) \wedge \mathrm{Trans}_\pi(x,y) \rightarrow \psi(\vec{y})).$$

Thus, in addition to the power of expression for Term_π and Trans_π, our proposed language should be able to formulate the relevant pre- and post-conditions for program correctness investigations.

Algorithmic solvability

The language should also be strong enough for the formulation of problem predicates that might interest us. By this we mean a formula $\rho(x_1,\ldots,x_n,y_1,\ldots,y_n)$ which poses an algorithmic problem in the following sense: does there exist a program $\pi(x_1,\ldots,x_n)$ whose output, if it exists, is always a solution of ρ?

Formally

$$\underline{A} \models \forall\vec{x}\forall\vec{y}(\mathrm{Trans}_{\pi}(\vec{x},\vec{y}) \rightarrow \rho(\vec{x},\vec{y})).$$

This list could be prolonged; but what we learned from it is the fact, that many important algorithmic problems are of the form

$$\underline{A} \models \forall\vec{z}\phi(\vec{z}),$$

where $\phi(\vec{z})$ is a finite Boolean combination of termination and transduction predicates. Thus, the immediate goal is to find a language which expresses these predicates.

The following definition associates a regular language, lang(π), the *language of the program* π to each PASIC program π. Words in this language are formed from symbols

$$ {}_{j}R^{+}_{i_1\ldots i_{m_j}}, \qquad {}_{j}R^{-}_{i_1\ldots i_{m_j}} $$

associated to the tests $R_j(x_{i_1},\ldots,x_{i_{m_j}})$, and

$$ {}_{j}f_{k}^{\,i_1\ldots i_{n_j}} $$

associated to the assignments $x_j := f_k(x_{i_1},\ldots,x_{i_{n_k}})$. The basic idea is that a word in the language of π describes a possible path through the program π, successive symbols ${}_{j}f_{k}$ showing which assignments were made, ${}_{j}R^{\pm}_{i_1\ldots i_{n_j}}$ which decision branch was taken. The definition of lang(π) is obvious from this idea and the fact that such definitions may be given by recursion. {We again make the distinction between assignments, statements, statement sequences and programs in a step-by-step definition of lang(π).}

$$S'_A(x_i := f_j(x_{k_1},\ldots,x_{k_{n_i}})) = {}_{j}f_{k}^{\,i_1\ldots i_{n_k}};$$

$$S'_{St}(\ ) = \lambda, \text{ (the empty word)};$$

$$S'_{St}(\alpha) = S'_A(\alpha) \text{ if } \alpha \text{ is an assignment};$$

$$S'_{St}(\underline{\text{if}}\ R_i(x_{k_1},\ldots,x_{k_{m_i}})\ \underline{\text{then}}\ \pi_1\ \underline{\text{else}}\ \pi_2) = {}_{i}R^{+}_{k_1\ldots k_{m_i}} \circ \mathrm{lang}(\pi_1) \vee {}_{i}R^{-}_{k_1\ldots k_{m_i}} \circ \mathrm{lang}(\pi_2);$$

$$S'_{St}(\underline{\text{while}}\ R_i(x_{k_i},\ldots,x_{k_{m_i}})\ \underline{\text{do}}\ \pi) = ({}_{i}R^{+}_{k_1\ldots k_{m_i}} \circ \mathrm{lang}(\pi))^{*} \circ {}_{i}R^{-}_{k_1\ldots k_{m_i}};$$

$S'_{St}(\underline{\text{while}}\ \neg R_i(x_{k_1},\ldots,x_{k_{m_i}})\ \underline{\text{do}}\ \pi)$
$= ({}_iR^-_{x_1\ldots k_{m_i}} \circ \text{lang}(\pi))^* \circ {}_iR^+_{k_1\ldots k_{m_i}}$;

$S'_{St}(\underline{\text{begin}}\ \Sigma\ \underline{\text{end}}) = S'_{SS}(\Sigma)$;

$S'_{SS}(\pi) = S'_{St}(\pi)$;

$S'_{SS}(\pi;\Sigma) = S'_{St}(\pi) \circ S'_{SS}(\Sigma)$;

$S'(\underline{\text{begin}}\ \Sigma\ \underline{\text{end}}) = S'_{SS}(\Sigma)$.

{Here π, π_1, π_2 denote statements, Σ a statement sequence.}

Next, we associate to any $w \in \text{lang}(\pi)$ a quantifier-free first-order formula $\phi_w(x_1,\ldots,x_n)$ which expresses that π, upon input $x_1,\ldots,x_n$, takes path w through π. The definition of ϕ_w is by induction on the length of w.

$\phi_\lambda(x_1,\ldots,x_n) = (x_1{=}x_1 \wedge \ldots \wedge x_n{=}x_n)$

$\phi_{{}^{i,\ldots i_{n_j}}_{j f_k} \circ w}(x_1,\ldots,x_n) = \phi_w(x_1,\ldots,x_{j-1},f_k(x_{i_1},\ldots,x_{i_{n_j}}),x_{j+1},\ldots,x_n)$;

$\phi_{{}_iR^+_{k_1\ldots k_{m_i}} \circ w}(x_1,\ldots,x_n) = R_i(x_{k_1},\ldots,x_{k_{m_i}}) \wedge \phi_w(x_1,\ldots,x_n)$;

$\phi_{{}_iR^-_{k_1\ldots k_{m_i}} \circ w}(x_1,\ldots,x_n) = \neg R_i(x_{k_1},\ldots,x_{k_{m_i}}) \wedge \phi_w(x_1,\ldots,x_n)$.

Now, we simply define

$$\text{Term}_\pi(x_1,\ldots,x_n) \stackrel{\text{def}}{=} \bigvee_{w \in \text{lang}(\pi)} \phi_w(x_1,\ldots,x_n).$$

There should be no need for a formal proof of the fact that $\text{Term}_\pi(a_1,\ldots,a_n)$ holds in $\underline{A}$ for elements $a_1,\ldots,a_n$ of A iff π terminates on input $a_1,\ldots,a_n$; we have constructed it that way. But in the course of this construction we were forced to admit infinite disjunctions (of a very constructive kind) and so extend first-order logic. The best-known logical framework already in existence to treat this extension is $L_{\omega_1\omega}$ (see books by C. KARP and by KEISLER). {Since the present author was an early contributor to the field it was natural for him to think in those terms.}

A construction very similar to the one used above gives us a (possibly infinite) formula $\text{Trans}_\pi(x_1,\ldots,x_n, y_1,\ldots,y_n)$ expressing the transduction predicate. We define

$$\psi_w(x_1,\ldots,x_n, y_1,\ldots,y_n)$$

as a first-order quantifier-free formula, expressing that π, on input $x_1,\ldots,x_n$ takes path w through π and obtains final values $y_1,\ldots,y_n$. The only difference to the definition of ϕ_w is for $\lambda = w$ in which case we set

$$\psi_\lambda(x_1,\ldots,x_n, y_1,\ldots,y_n) = (x_1=y_1 \wedge \ldots \wedge x_n=y_n).$$

Altogether:

$$\mathrm{Trans}_\pi(x_1,\ldots,x_n, y_1,\ldots,y_n) = \bigvee_{w \in \mathrm{lang}(\pi)} \psi_w(x_1,\ldots,x_n, y_1,\ldots,y_n).$$

Actually, Trans_π can be expressed as $\mathrm{Term}_{\pi'}$ for some appropriate π'. From the fact alone that Term_π can be expressed as an infinite disjunction of quantifier-free first-order formulas we can draw some pretty conclusions.

DEFINITION. $\underline{A}$ has the unwind property iff for any program π which terminates for all inputs there exists a strongly equivalent program π' which has no loops and for which $\mathrm{lang}(\pi') \subseteq \mathrm{lang}(\pi)$.

It has been noted (e.g. by PATERSON, private communication), that some structures such as the reals do have the unwind property. We give a criterion by which this will follow.

Let $\mathrm{algT}(\underline{A}) = \{\forall\vec{x}\phi(x) : \phi$ is a finite Boolean combination of formules Term_π and $\underline{A} \models \forall\vec{x}\phi(\vec{x})\}$. We shall show in section 4 that algT for the reals has the property that it can be axiomatized by a set of universal first-order formules, say Γ. Now, if $\underline{A} \models \forall\vec{x}\ \mathrm{Term}_\pi(\vec{x})$ then $\Gamma \vdash \forall\vec{x} \bigvee_{w\in\mathrm{lang}(\pi)} \phi_w(\vec{x})$.

By the compactness theorem of first-order logic it follows that there exists a finite subset $W \subseteq \mathrm{lang}(\pi)$ such that $\Gamma \vdash \forall\vec{x} \bigvee_{w\in W} \phi_w(\vec{x})$. Let π' be a (loop-free!) program such that $\mathrm{lang}(\pi') = W$. We have π' strongly equivalent to π as easily seen. More generally:

THEOREM. $\underline{A}$ *has the unwind property iff there exists a set* Γ *of universal first-order formules such that all termination sentences of* algT(A) *follow from* Γ.

PROOF. One direction has been shown above as a consequence of compactness. In the other direction, let Γ be the set of all universal first-order sentences which follow from $\mathrm{AlgT}(\underline{A})$ and assume that $\underline{A}$ has the unwind property. Let π be such that $\underline{A} \models \forall\vec{x}\ \mathrm{Term}_\pi(\vec{x})$ and let π' be as a loop-free program

strongly equivalent to π and with $lang(\pi') \subseteq lang(\pi)$ as per assumption. Then $\forall\vec{x}\ Term_{\pi'}(\vec{x})$, which is a universal first-order sentence, holds in $\underline{A}$ and hence $\forall\vec{x}\ Term_{\pi'}(\vec{x}) \in \Gamma$. Thus $\Gamma \vdash \forall\vec{x} \bigvee_{w\in lang(\pi')} \phi_w(\vec{x})$ and a fortiori $\Gamma \vdash \forall\vec{x} \bigvee_{w\in lang(\pi)} \phi_w(\vec{x})$, since $lang(\pi') \subseteq lang(\pi)$. □

COROLLARY. *If π halts for all inputs in all models of a first-order theory Γ then π can be unwound.* □

3. FORMALIZATION OF THE NOTION OF COMPUTATION IN FIRST-ORDER LOGIC

In special cases, in particular if number theory can be interpreted into the theory of a structure, it is possible to stay within the framework of first-order logic in order to express the basic algorithmic notions. In the present section we take advantage of this fact and use it to give an exposition of the very beautiful result of RABIN & FISCHER on the complexity of decision procedures for additive number theory.

Let us consider the natural number system

$$\mathbb{N} = \langle N,+,\cdot,=,0,1\rangle$$

and its corresponding first-order language

$$L(+,\cdot,=,0,1).$$

As mentioned before, all computations (i.e. partial recursive functions) can be performed by PASIC programs on the capabilities P (predecessor), S (successor), 0 (zero function) and =0 (test for zero) alone. {Remark: the language L(P,S,=0,0) is, however, not sufficient to express termination. This follows from the fact that it is decidable, while the halting problem for PASIC is undecidable.}

Our first goal is to associate to each program π the set of "computation sequences" associated to it and to define that concept in an appropriate formal fashion.

Let us explain the notion by a simple example.

The program

```
            begin
q1              while ×2 ≠ 0 do
                begin
q3                  ×2 := P×2;
q5                  ×1 := S×1;
                end
q7          end
```

has two variables and seven lines, which we number as indicated (these numbers are not labels). A *computation sequence* according to this program is a finite sequence of natural numbers

$$y_{11}\ y_{12}\ p_1\ y_{21}\ y_{22}\ p_2\ y_{31}\ y_{32}\ p_3 \cdots y_{m1}\ y_{m2}\ p_m.$$

The parts $(y_{i1}\ y_{i2})$ indicate the values of ×1 and ×2 at the successive stages of the computation; the numbers p_i indicate the line numbers arrived at. This sequence is a computation sequence according to π iff

$$p_i = q_1 \wedge \forall i(1 \leq i \leq m-1 \rightarrow$$
$$\{[p_i = q_1 \wedge y_{12} \neq 0 \wedge p_{i+1} = q_3 \wedge y_{i+1,1} = y_{i,1} \wedge y_{i+1,2} = y_{1,2}]$$
$$\vee [p_i = q_1 \wedge y_{i2} = 0 \wedge p_{i+1} = q_7 \wedge y_{i+1,1} = y_{i,1} \wedge y_{i+1,2} = y_{1,2}]$$
$$\vee [p_i = q_3 \wedge p_{i+1} = q_5 \wedge y_{i+1,1} = y_{i,1} \wedge y_{i+1,2} = y_{i,2} - 1]$$
$$\vee [p_i = q_5 \wedge p_{i+1} = q_1 \wedge y_{i+1,1} = y_{i,1} + 1 \wedge y_{i+1,2} = y_{i,2}]\})$$
$$\wedge p_m = q_7.$$

We express, for purposes that will become apparent later, the same fact by using a particular encoding of computation sequences by means of the Gödel β-function defined as follows:

$$f(b,c,h) = b \bmod (1 + (h+1) \cdot c).$$

Using the Chinese remainder theorem, it can be shown:

for any sequence $k_0,\ldots,k_l$ there exist b,c such that
$k_h = f(b,c,h), \quad 0 \leq h \leq l.$

Thus, we propose to choose b,c such that

$$\left.\begin{aligned} y_{ij} &= f(b,c,(i-1)(n+1)+j) \\ p_i &= f(b,c,i(n+1)) \end{aligned}\right\} \; 1 \le i \le m, \quad 1 \le j \le n \; (=2).$$

The formula defining computation sequences of length m can now obviously be rewritten as a formula

$$B_\pi(b,c,m)$$

with the aid of the function symbol f. To write $B_\pi(b,c,m)$ as a formula of $L(+,\cdot,=,0,1)$ we need to express the predicate $\le$ in that language, which is easy, and define the relation $f(b,c,h) = r$, which is accomplished by

$$\exists y(b=(1+(h+1)\cdot c)\cdot y+r \wedge \exists x(r+x = 1+(h+1)\cdot c)).$$

The formula $\tilde{B}_\pi(b,c,m)$ which is obtained by the rewritings indicated above is the main tool for expressing algorithmic notions in $L(+,\cdot,=,0,1)$.

For example, it can be used to show the result (of CHURCH), that there is no decision program for $L(+,\cdot,=,0,1)$. We use below a variant of that proof to show that there is no *practical* decision program for the language $L(+,=,0,1)$. {PRESBURGER has shown in 1929 that a decision procedure exists; the best known procedure take something of the order $2_2^{2^n}$ steps on formulas of the length n. The result below shows that there is no hope of drastically improving this.}

Let a Gödel-numbering of formulas $\phi \in L(+,=,0,1)$ be given; assume that it can be easily computed (see below) and that the length $|\bar{\phi}|$ of the Gödel number of ϕ, written to base 2 does grow linearly in the length of the formula ϕ.

Let, for any PASIC program $\pi(x1,\ldots,xn)$, the partial function

$$S(\pi): N^n \to N$$

be the function computed by π. {See earlier sections for the method to define $S(\pi)$}. By $S(\pi)[a_1,\ldots,a_n]$ we denote the value of $S(\pi)$ at the argument $(a_1,\ldots,a_n)$.

THEOREM (RABIN & FISCHER 1973). *Assume that* Σ *is a decision program of* $L(+,=,0,1)$, *i.e. assume that for all closed formulas* $\phi \in L(+,=,0,1)$ *we have*

$$S(\Sigma)[\bar{\phi},0,\ldots,0] = \begin{cases} 1, & \text{if } \phi \text{ holds in } \mathbb{N}, \\ 0, & \text{otherwise.} \end{cases}$$

Then there exists c *such that for infinitely many formulas* ϕ *the program* Σ *takes more than* $2^{2^{c|\bar{\phi}|}}$ *steps.*

PROOF.

1. To each $\pi \in$ PASIC and each $k \in \mathbb{N}$ we propose to construct a formula $\phi_{\pi,k} \in L(+,=,0,1)$ which holds exactly if

$$\begin{cases} S(\pi)[\bar{\pi},k,0,\ldots,0] = 0 \\ \text{and } \pi \text{ stops on this input in at most } 2^{2^k} \text{ steps.} \end{cases}$$

We shall take care that $|\bar{\phi}_{\pi,k}| = 0(k)$ and observe that $\bar{\phi}_{\pi,k}$ can be "easily computed".

Observe: on input $[s,0,\ldots,0]$ a computation of length $m \leq 2^{2^k}$ can find values of the variables at most of size $2^{2^k}+s$ (the biggest increase of a variable in one step is by 1). Thus, let us concentrate on $s \leq k$ and observe the bound of $2^{2^{k+1}}$.

Next, one shows that if m,s are within these bounds that the encoding of computation sequences by means of f can be accomplished by $b,c \leq 2^{2^{2k+4}}$. These observations can be put to immediate use as follows. The following formula expresses (in a somewhat extended language) the property $\phi_{\pi,k}$:

$$\neg\exists b\exists c\exists m(m \leq 2^{2^k} \wedge \tilde{B}_{\pi}(b,c,m) \wedge f(b,c,(m-1)(n+1)+1) = 0).$$

However, multiplication is used in these formulas in various places (even after the symbol f is eliminated as above). The basic observation now is the following

(a) there exists a formula $M_i^*(x,y,z) \in L(+,=,0,1)$ and numbers $r_i \geq 2^{2^{2^i}}$ such that $|M_i^*| = 0(i)$ and $M_i^*(x,y,z) \Leftrightarrow x=y,z \wedge x,y,z \leq r_n$.

(b) there exists a formula $L_k(m) \in L(+,=,0,1)$ such that $|L_k(m)| = 0(k)$ and

$$L_k(m) \Leftrightarrow m \leq 2^{2^k}.$$

Thus, the above proposal for a formula for $\phi_{\pi,k}$ can be realized, using the known bounds on b,c , by $\neg\exists b\exists c\exists m(b,c \le r_{k+4} \wedge m \le 2^{2^k} \wedge \tilde{B}_\pi(b,c,m) \wedge \ldots)$ translated by making the appropriate substitutions: Replacing multiplication in $\tilde{B}_\pi$ by M^*_{k+4} we obtain $\tilde{B}^{k+4}_\pi(b,c,m)$, etc.: $\neg\exists b\exists c\exists m(M^*_{k+4}(0,0,b) \wedge M^*_{k+4}(0,0,c) \wedge L_k(m) \wedge \tilde{B}^{k+4}_\pi(b,c,m) \wedge \ldots)$. Provided that we have indeed these L_k and M^*_k, the formula $\phi_{\pi,k}$ is thus obtained, and grows - clearly - linearly in k in the sense of $O(k)$.

2. Supposing, then, that we have $\phi_{\pi,k} \in L(+,=,0,1)$, let us finish the proof of the theorem. Let Σ be the program as per assumption. Consider the following program ρ:

```
ρ:
begin
      if "×1 = Gödel number of a program π" then "×1 := φ̄π,×2";
      ×2 := 0;
      Σ
end
```

{The routines in quotation marks would still have to be written, our assumptions on Gödel-numberings and ease of computation of M^*_k, L_k are to be used to observe that these routines do not take more than polynomial time.}
Now

$$S(\rho)[\bar{\pi},k,0,\ldots,0] = S(\Sigma)[\bar{\phi}_{\pi,k},0,\ldots,0] = \begin{cases} 1 & \text{if } \phi_{\pi,k}, \\ 0 & \text{else} \end{cases}$$

$$= \begin{cases} 1 & \text{if } S(\pi)[\bar{\pi},k,0,\ldots,0] \text{ in} \le 2^{2^k} \text{ steps,} \\ 0 & \text{otherwise.} \end{cases}$$

A program ρ with the above property, has the property - as is immediately seen - that ρ terminates on $[\bar{\rho},k,0,\ldots,]$ at best in $> 2^{2^k}$ steps. Hence Σ terminates on $[\bar{\phi}_{\rho,k},0,\ldots,0]$ in $O(2^{2^k})$ steps, while $|\bar{\phi}_{\rho,k}| = O(k)$.

3. We need to find M^*_k, L_k.
Let us start with a predicate $M'_k(x,y,z)$ which accomplishes somewhat less than M^*_k.

$$M'_n(x,y,z) \Leftrightarrow x{=}y,z \wedge 0 \le z \le 2^{2^n} - 1.$$

{This would already give us $L_k(m)$, by the observation that $m \le 2^{2^k}$ is expressed by $M'_k(0,0,m)$.}

Clearly

$$M'_0(x,y,z) \underset{\text{def}}{\Leftrightarrow} (z=0 \wedge x=0) \vee (z=1 \wedge x=y)$$

will do. The step from n to n+1 relies on the remark that for $a \geq 2$ we have

$$\{n \mid 0 \leq n \leq a^2-1\} = \{x_1x_2+x_3+x_4 \mid 0 \leq x_1,\ldots,x_4 \leq a-1\},$$

which is immediate from $a^2-1 = (a+1)(a-1)$. Namely, we set

$$\begin{aligned} M'_{n+1}(x,y,z) \underset{\text{def}}{\Leftrightarrow} \exists z_1\ldots z_5\, \exists x_1\ldots x_4 \\ (M'_n(z_5,z_1,z_2) \wedge z=z_5+z_3+z_4 \\ \wedge M'_n(x_1,y,z_1) \wedge M'_n(x_2,x_1,z_2) \wedge M'_n(x_3,y,z_3) \\ \wedge M'_n(x_4,y,z_4) \wedge x=x_2+x_3+x_4). \end{aligned}$$

Note that $0 \leq z_i \leq 2^{2^n}-1$ by assumption, since they occur in the context $M'_n(.,.,z_i)$. It follows, because

$$z = z_1z_2+z_3+z_4,$$

that $0 \leq z \leq (2^{2^n})^2 - 1$ by the above remark, hence $0 \leq z \leq 2^{2^{n+1}}-1$. Moreover, we observe

$$x = x_2+x_3+x_4 = x_1,z_2+y,z_3+y,z_4 = y,z_iz_2+y,z_3+y,z_4 = y \circ z.$$

Unfortunately $|M'_{n+1}| \geq 5 \circ |M'_n|$, which won't do. But observe the form of M'_{n+1}:

$$\exists\vec{w}(M'_n(\vec{w}_1) \wedge \ldots \wedge M'_n(\vec{w}_5) \wedge C(\vec{w}))$$

where $\vec{w}_i$ are subsequences of the sequence $\vec{w}$ of variables. By predicate logic, the above formula is equivalent to

$$\exists\vec{w}\forall\vec{v}((\bigvee_{i=1}^{5} \vec{v}=\vec{w}_i \rightarrow M'_n(\vec{v})) \wedge C(\vec{w})),$$

whose length is now additive in the length of M'_n : $|M'_{n+1}| = c+|M'_n|$, hence $|M'_n| = 0(n)$. The thus changed M'_n will be denoted by M_n.
In order to obtain M^*_n let us first define:

$x=r'_n$: "x = smallest w with $\forall z \exists y (M_n(0,0,z) \rightarrow M_n(w,y,z))$".

In essence, r'_n is the least common multiple of numbers $\leq 2^{2^n} - 1$. It can be shown, using Hadamard's theorem on the distribution of primes, that there exists c_1, c_2 such that

$$2^{2^{2^{c_1 \cdot n}}} \leq r'_n \leq 2^{2^{2^{c_2 \cdot n}}} .$$

Let α be such that $\alpha \cdot c_1 \leq 1$ and define

$$r_n = r'_{\alpha \cdot n} ,$$

which ensures $r_n \geq 2^{2^{2^n}}$, as desired. Furthermore, let β be such that $r_n^2 \leq r_{\beta \cdot n}$.

Now, $M_n^*(x,y,z)$ is defined by

"$x,y,z \leq r_n \wedge x \underline{\text{mod}}\ q \equiv y \underline{\text{mod}}\ q \cdot z \underline{\text{mod}}\ q$

for all $q \leq 2^{2^{\alpha \cdot \beta \cdot n}} - 1$".

This definition will work in L(+,=,0,1) since r'_n, hence r_n is defined in that framework and multiplication now takes place in a range for which formulas M_n can be used. The correctness of the formula follows from the following two remarks.

If $x,y \cdot z \leq$ least common multiple of $\{q: q \leq s\}$ then $x = y \cdot z$ iff $x \underline{\text{mod}}\ q \equiv y \underline{\text{mod}}\ q \cdot z \underline{\text{mod}}\ q$ (mod q) for all $q \leq s$. {Which is easy from modular arithmetic.} Now, if $x,y,z \leq r_n$ then $x,y \cdot z \leq r_n^2 \leq r_{\beta \cdot n} = r'_{\alpha \cdot \beta \cdot n} =$ l.c.m$\{q: q \leq 2^{2^{\alpha \cdot \beta \cdot n}} - 1\}$. Thus, taking $s = 2^{2^{\alpha \cdot \beta \cdot n}}$, we obtain the definition for M_n^*. Again, we see $|M_n^*| = O(n)$, and that it is "easy to compute" in the sense used above. □

4. THE AXIOMATIZATION OF ALGORITHMIC THEORIES

Let $\underline{A} = \langle A, R_1, \ldots, R_n, f_1, \ldots, f_n \rangle$ be a data structure, and let algL be the corresponding algorithmic language. For the present purposes (i.e. for expressing the kind of algorithmic facts that are of interest to us for now)

we let algL consist of all finite Boolean combinations of formulas Term_π, where π ranges over all PASIC programs associated to the structure $\underline{A}$, (see section 2).

algL is clearly a constructive sublanguage of $L_{\omega_1\omega}$, consisting moreover only of universal formulas, i.e. of formulas whose only quantifiers are prefixed to the formula and are universal quantifiers. Clearly, we can expect some rather special properties of such languages, some of which will be made use of in the sequel.

The first task is to devise an appropriate formal deductive system for algL. For this purpose any proof system for $L_{\omega_1\omega}$ will do - as long as it has the subformula property. This means that the proofs must always be obtainable through rules whose antecedents consist only of subformulas of the conclusions. Instead of writing down such a system if full detail (which the reader may do if he likes by consulting LOPEZ-ESCOBAR, Fund. Math. 1965, p. 253 ff). we discuss briefly a system which works directly with the programs.

Let $\pi(x_1,\ldots,x_n)$ be a PASIC program, and let $\underline{A}$ be a data-structure in which π is defined. We write $\underline{A} \models \pi[a_1,\ldots,a_n]$ if π terminates in $\underline{A}$ on input $\langle a_1,\ldots,a_n\rangle$. In analogy to predicate logic we introduce meanings for $\underline{A} \models \pi_1 \vee \pi_2$, $\underline{A} \models \neg\pi$, etc.. In addition, we allow *prefixed* universal quantifiers,

$$\underline{A} \models \forall x_1 \ldots x_n \; \pi(x_1,\ldots,x_n)$$

being true, as an example, if π terminates in $\underline{A}$ on all possible inputs. Let us call AlgL the set of all such expressions. Let M,N be sets of formulas of AlgL; we write

$$M \models N$$

iff for each $\underline{A}$ in which all expressions ϕ hold true at least one expression in N holds true (free variables both in M,N are assigned the same element of $\underline{A}$).

Clearly, if M' and N' are sets of *straight-line* programs and $M' \models N'$ then $M \models N$ for any sets M and N with $M' \subseteq M$, $N' \subseteq N$. We shall take such pairs M,N as *axioms* of our formal system. Indeed, our formal system is devised only to derive pairs of sets of formulas (P,Q) such that $P \models Q$.

Thus it starts by the axioms introduced just now, which we write

$$M \vdash N \qquad \text{(called a sequent)}$$

and proceeds from these by rules of proof as enumerated below. These rules are understood as: the sequent below the bar follow from the sequent or sequents above the bar.

Conventions

(a) σ denotes a straight-line program;

(b) π denotes a statement;

(c) Σ denotes a sequence of statements.

(d) Furthermore, we introduce an additional line in the syntax diagram for PASIC:

undefined

is a statement with no variables; the intended meaning is a program which never terminates on any input.

(e) The expression while^k b do π is an abbreviation for the statement

if $\neg$b then undefined else π;
if $\neg$b then undefined else π;
.
.
. } k lines
if $\neg$b then undefined else π;
if b then undefined

(f) If $\pi(x_1,\ldots,x_n)$ denotes a program with variables $x_1,\ldots,x_n$ and if $t_1,\ldots,t_n$ are terms, then $\pi(t_1,\ldots,t_n)$ denotes the program which starts by (simultaneously) assigning to $x_1,\ldots,x_n$ the values $t_1,\ldots,t_n$ respectively and then proceeds to do π; analoguously for $\phi(t_1,\ldots,t_n)$.

Rules of proof

(0) $\dfrac{M \cup \{\phi\} \vdash N,\ M \cup \{\psi\} \vdash N}{M \cup \{\phi \vee \psi\} \vdash N}$

(1) $\dfrac{M \vdash N \cup \{\phi,\psi\}}{M \vdash N \cup \{\phi \vee \psi\}}$

(2) $\dfrac{M \cup \{\phi\} \vdash N}{M \vdash N \cup \{\neg\phi\}}$

(3) $\dfrac{M \vdash N \cup \{\phi\}}{M \cup \{\neg\phi\} \vdash N}$

(4) $\dfrac{M \cup \{\underline{\text{begin}}\ \sigma;\ \underline{\text{if}}\ b\ \underline{\text{then}}\ \underline{\text{undefined}}\ \underline{\text{else}}\ \pi_2;\ \Sigma\ \underline{\text{end}}\} \vdash N,\quad M \cup \{\underline{\text{begin}}\ \sigma;\ \underline{\text{if}}\ b\ \underline{\text{then}}\ \pi_1\ \underline{\text{else}}\ \underline{\text{undefined}};\ \Sigma\ \underline{\text{end}}\} \vdash N}{M \cup \{\underline{\text{begin}}\ \sigma;\ \underline{\text{if}}\ b\ \underline{\text{then}}\ \pi_1\ \underline{\text{else}}\ \pi_2;\ \Sigma\ \underline{\text{end}}\} \vdash N}$

(5) $\dfrac{M \vdash N \cup \{\underline{\text{begin}}\ \sigma;\ \underline{\text{if}}\ b\ \underline{\text{then}}\ \underline{\text{undefined}}\ \underline{\text{else}}\ \pi_2;\ \Sigma\ \underline{\text{end}},\ \underline{\text{begin}}\ \sigma;\ \underline{\text{if}}\ b\ \underline{\text{then}}\ \pi_1\ \underline{\text{else}}\ \underline{\text{undefined}};\ \Sigma\ \underline{\text{end}}\}}{M \vdash N \cup \{\underline{\text{begin}}\ \sigma;\ \underline{\text{if}}\ b\ \underline{\text{then}}\ \pi_1\ \underline{\text{else}}\ \pi_2;\ \Sigma\ \underline{\text{end}}\}}$

(6) $\dfrac{M \cup \{\underline{\text{begin}}\ \sigma;\ \underline{\text{while}}^k\ b\ \underline{\text{do}}\ \pi;\ \Sigma\ \underline{\text{end}}\} \vdash N,\ k=1,2,\ldots}{M \cup \{\underline{\text{begin}}\ \sigma;\ \underline{\text{while}}\ b\ \underline{\text{do}}\ \pi;\ \Sigma\ \underline{\text{end}}}$

(7) $\dfrac{M \vdash N \cup \{\underline{\text{begin}}\ \sigma;\ \underline{\text{while}}^k\ b\ \underline{\text{do}}\ \pi;\ \Sigma\ \underline{\text{end}} : k=1,2,\ldots\}}{M \vdash N \cup \{\underline{\text{begin}}\ \sigma;\ \underline{\text{while}}\ b\ \underline{\text{do}}\ \pi;\ \Sigma\ \underline{\text{end}}\}}$

(8) $\dfrac{M \cup \{\underline{\text{begin}}\ \sigma;\ \Sigma_1;\ \Sigma_2\ \underline{\text{end}}\} \vdash N}{M \cup \{\underline{\text{begin}}\ \sigma;\ \underline{\text{begin}}\ \Sigma_1\ \underline{\text{end}};\ \Sigma_2\ \underline{\text{end}}\} \vdash N}$

(9) $\dfrac{M \vdash N \cup \{\underline{\text{begin}}\ \sigma;\ \Sigma_1;\ \Sigma_2\ \underline{\text{end}}\}}{M \vdash N \cup \{\underline{\text{begin}}\ \sigma;\ \underline{\text{begin}}\ \Sigma_1\ \underline{\text{end}};\ \Sigma_2\ \underline{\text{end}}\}}$

(10) $\dfrac{M \cup \{\phi(t)\} \vdash N}{M \cup \{\forall x \phi(x)\} \vdash N}$ if t is any term

(11) $\dfrac{M \vdash N \cup \{\phi(y)\}}{M \vdash N \cup \{\forall x \phi(x)\}}$ if y is a variable not occurring in the conclusion.

Observe that the above axiom system for $\vdash$ lacks effectiveness on several counts, which preclude the possibility of using the rules of proof "backwards" in order to decide a given sequent. First, the axioms are not given in an effective way, second, there is a rule, rule (7) which is infinitary, i.e. the conclusion requires infinitely many premisses. This cannot be circumvented by any adequate proof system for algorithmic logic: Either one has an infinitary rule or one has an incomplete proof system. The infinitary rule of our system of algorithmic logic is closely related to Carnap's rule of arithmetic (which makes it complete) and serves as an induction principle. There are parallels, and distinctions, to be made with recursion induction (MC CARTHY), truncation induction (MORRIS), fixedpoint induction (PARK), computational induction (SCOTT), structural induction (BURSTELL). But we cannot here embark on these. Neither shall we prove the Completeness theorem $M \models N$ *iff* $M \vdash N$ *is provable*, which follows

by translation of the well-known methods in infinitary logic. We shall, however, have occasion to use this result.

Let us now return to algL. For a particular structure $\underline{A}$ we are interested in those closed formulas $\forall x_1 \ldots x_n \; \phi(x_1,\ldots,x_n)$, with $\phi \in$ algL, which are true in $\underline{A}$. As we have seen, such formulas allow us to express important enough algorithmic properties of $\underline{A}$ to warrant special attention. Let algT($\underline{A}$) denote this set, we call it the *algorithmic theory* of $\underline{A}$. In the presence of a complete proof system for algL, we may reasonably ask for an axiomatization of various algorithmic theories: Conceivably, an axiom system for algT($\underline{A}$) may have more than just $\underline{A}$ as a model, which would provide us with additional insight into the power of computation (computations failing to distinguish between the various models).

Let, for any Γ the deductive closure $\Gamma^{\vdash}$ be defined as

$$\Gamma^{\vdash} = \{\forall\vec{x}\phi(\vec{x}) \mid \phi \in \text{algL}, \; \Gamma \vdash \forall\vec{x}\phi(\vec{x}) \text{ provable}\}.$$

We seek, for given $\underline{A}$, a set Γ such that

$$\Gamma^{\vdash} = \text{algT}(\underline{A}).$$

(i) Let $\underline{N} = \langle N,S,0,=0\rangle$ be the natural number system. Then algT($\underline{N}$) is axiomatized by finitely many axioms, namely the Peano axioms

$$\forall x(S(x) \neq 0)$$
$$\forall x\forall y(S(x)=S(y) \rightarrow x=y)$$
$$\forall x(\bigvee_{i=0}^{\infty} S^{(i)}(0) = x)$$

which are clearly universal quantifications of formulas of algL($\underline{N}$) expressing termination of appropriate programs. Namely: if $\mathbb{N}' = \langle N',S',0',=0'\rangle$ satisfies the Peano axioms then $\underline{N}' \cong \underline{N}$, hence,

$$\{\text{Peano axioms}\} \vdash \phi \quad \text{iff} \quad \underline{N} \models \phi.$$

(ii) The ordered field of reals

$$\underline{R}_{\leq} = \langle R,\leq,+,\cdot,-,^{-1},0,1\rangle$$

may be axiomatized by the axioms of an archimedian ordered field:

. ordered field axioms (universally quantified first-order formulas of $L(\leq,+,\cdot,-,^{-1},0,1)$),

. archimedian axiom $a>0 \wedge b>0 \rightarrow \bigvee_{n=1}^{\infty} \underbrace{a+a+\ldots+a}_{n} \geq b$.

These axioms can clearly be expressed by equivalent formulas of algL. They may serve as axiomatization. Clearly, if {arch.O.f.} $\vdash \phi$ then $\underline{R}_{\leq} \models \phi$ since $\underline{R}_{\leq}$ is such a field. Conversely, if $\underline{R}_{<} \models \phi$ and $\underline{F}$ is arch.o.f. then, by algebra, $\underline{F} \subseteq \underline{R}$. Since universal formulas are inherited by substructures, $\underline{F} \models \phi$; hence {arch.o.f} $\vdash \phi$ by the completeness theorem.

Thus, from a computational point of view, all arch.ordered fields are the same (as regards that algorithmic properties). The field of reals of course is not computable in the technical sense; this would ask for an interpretation of $\underline{R}_{\leq}$ into $\underline{N}$ by which all tests $\leq$ and operations $\cdot,+,\ldots$ become computable number theoretic functions. However, the axioms have some computable models, e.g. the rationals and the algebraic numbers.
{We could have asked something more of the axioms, namely to characterize those structures $\underline{F}$ for which the following statement holds:

$$\underline{F} \models \forall\vec{x}\phi(\vec{x}) \text{ iff } \underline{R}_{\leq} \models \forall\vec{x}\phi(\vec{x})$$

for all $\phi \in$ algL. Then the axioms for archimedian order would not suffice; e.g. it is simple to devise a program which holds for all inputs on the rationals, but does not so on the reals. The class of structures which have this stronger property could be called the computationally closed fields. Using Tarski's decision procedure for the reals it can be shown (ENGELER 1968), that such fields are obtained from the rationals by admitting all reals which are the limits of nested intervals $(a_n,b_n) \in \mathbb{Q}^2$ computable by a program in successive loops.}

(iii) Slightly harder is the task of axiomatizing the algorithmic theory of the reals without the ordering relation. It can be shown to lead to the axioms for formally real fields, (see ENGELER 1973). An important fact of this axiomatization is that it leads to (universal) first-order formulas; this will be used in section 5. The reader may puzzle a moment about this difference between ordered and non-ordered reals, because, clearly, ordering can be defined algebraically

in formally real fields (use notion of positiveness). But this definition is, as the axiomatization theorem proves, not expressible in algL.

Even if, for a given structure $\underline{A}$, there is no axiomatization immediately available, we may ask for those structures $\underline{B}$ which are *algorithmically equivalent* to $\underline{A}$. One sufficient criterion (LOPEZ-ESCOBAR 1966) can be borrowed from $L_{\omega_1\omega}$: Let IsoSub($\underline{A}$) be the class of isomorphic copies of substructures of $\underline{A}$: Then $\underline{A}$ is algorithmically equivalent to $\underline{B}$ iff IsoSub($\underline{A}$) = IsoSub($\underline{B}$). This criterion can be considerably improved.

5. THE GROUP OF A PROBLEM

In this last lecture, the author may be allowed to ride a personal hobby, namely the structural relations between programs and problems. We have some promising results, but most of what we'll show now is of a programmatic nature.

Let Γ be an algorithmic theory, and let $\rho \in$ algL be an algorithmic problem, say

$$\rho(x_1,\ldots,x_n,y)$$

where for given $x_1,\ldots,x_n$ we seek solutions y_i making ρ hold. Let Y be the set of solutions (for given $x_1,\ldots,x_n$), and assume that we have programs $\pi_i(x_1,\ldots,x_n)$ computing the y_i (using some additional capabilities perhaps). Often, the nature of the problem ρ imposes on Y an mathematical structure, e.g. symmetries may put a group structure on Y.

Let us take the paradigm of classical Galois theory. There $\rho(x_0,\ldots,x_n,y)$ is a polynomial over a field F, $a_0,\ldots,a_n \in F$ and say, $y_1,\ldots,y_n \in E$, the splitting field of ρ over F. If ρ is separable then the Galois group of ρ is G(E/F), the set of automorphisms of E leaving F pointwise fixed. This group can be determined by algebraic manipulations with ρ (determined in principle, the complexity of the algorithm is large polynomial). An investigation of the structure of the group (e.g. for its solvability) leads at once to solution programs π_i. Indeed, as can be observed, the group G(E/F) is also present as a group of substitutions on the π_i.

Without going into the details of this connection in this particular case, let us rather ask for conditions which allow us to develop a Galois theory with respect to arbitrary Γ and ρ. As it turns out, this can be done for an interesting array of theories.

Let Γ be a set of (universally closed) algorithmic formulas and let $\rho(x,y)$ be an algorithmic problem. (Observe that we have, for notational convenience, chosen $n = 1$).

We first formulate an analogue to the notion of "separable polynomial":

DEFINITION. ρ is *well-posed* in Γ if there exists a set Y of new variables, and a diagram $\Delta(x,Y)$ with the following properties:

(a) $\Gamma \cup \Delta(x,Y)$ is consistent,
$\Gamma \cup \Delta(x,Y) \vdash \rho(x,y_i)$ for all $y_i \in Y$;

(b) $y_i \neq y_j \in \Delta(x,Y)$ for all $i \neq j$;

(c) $\Gamma \cup \Delta(x,Y) \cup \rho(x,y) \vdash \{y=y_i : y_i \in Y\}$.

{By a diagram $\Delta(x,Y)$ we mean a set of basic, i.e. negated and unnegated atomic, formulas of the laguage of Γ augmented by Y, such that for any such formula α either $\alpha \in \Delta(x,Y)$ or $\neg\alpha \in \Delta(x,Y)$}. Let $L_0(x)$ denote the set of basic formulas in x. Next, we need a rather strange-looking property of Γ, which will, however, be seen to follow from a far more familiar concept.

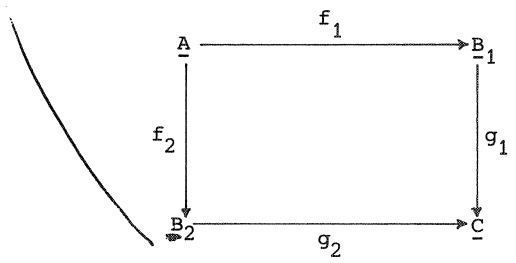
DEFINITION. Γ has the *permutation property* with respect to ρ if, whenever $\Delta(x,Y)$ and $\Delta'(x,Y)$ satisfy (a), (b), (c) above and if $\Delta(x,Y) \cap L_0(x) = \Delta'(x,Y) \cap L_0(x) = \Delta(x)$ then there exists a permutation $s \in S(Y)$, the set of all permutations of Y with

$$\Delta'(x,Y) = \Delta^s(x,Y).$$

{$\phi^s(y_{i_1},\ldots,y_{i_k})$ is defined as $\phi(s(y_{i_1}),\ldots,s(y_{i_k}))$, and $\Delta^s(x,Y) = \{\alpha^s : \alpha \in \Delta(x,Y)\}$}.

The more familiar property mentioned above is the *amalgamation property*.

DEFINITION. Γ has the amalgamation property, if for any models $\underline{A}$, $\underline{B}_1$, $\underline{B}_2$ of Γ and injections f_1 and f_2, there exists a model $\underline{C}$ of Γ and injections g_1 and g_2 such that the following diagram commutes:



The following is a list of theories, expressable in algL, which have the amalgamation property:

groups, lattices,
various fields (e.g. arch.o.fields, formally real fields),
various geometries,
Boolean and cylindrical algebras,
differential fields.

By the results below, Galois theory becomes available for these theories in a uniform manner. {Generalizations of G-th have been known in some of the cases, these turn out to be special cases of our approach.}

THEOREM. *If* Γ *is a set of universally quantified algorithmic formulas and if* Γ *has the amalgamation property, then* Γ *has the permutation property with respect to all* ρ.

The proof is obtained by building Herbrand-models of $\Gamma \cup \Delta(x,Y)$ and $\Gamma \cup \Delta'(x,Y)$ and using universality of Γ. The details need not interest us here, (see ENGELER 1974). But now, in presence of the permutation property, we may at once define the group of a well-posed problem.

DEFINITION. Let $\rho(x,y)$ be well-posed in Γ, let $\Delta(x,Y)$ have properties (a), (b), (c) and let $\Delta(x) = \Delta(x,Y) \cap L_0(x)$. We set

$$G^{L'}_{\Delta(x)} = \{t \in S(Y) : \Gamma \cup \Delta(x,Y) \vdash \phi \equiv \phi^t, \text{ all } \phi \in L'(x,Y)\},$$

where L' is a sublanguage of algL, $L_0 \subseteq L'$.

THEOREM. *If* Γ *has the permutation property with respect to* $\rho(x,y)$, *then*

(i) $G^{L'}_{\Delta(x)}$ *is a group (under composition of permutations) and depends only on* $\Delta(x)$;

(ii) *if* L' *is contained in the closure of* L_0 *under infinite disjunctions or if* $\Gamma \cup \Delta(x,Y)$ *is* L'*-complete (i.e. if* Γ *is* L'*-model-complete), then* $G^{L'}_{\Delta(x)} = G^{L_0}_{\Delta(x)}$;

(iii) $G^{L_0}_{\Delta(x)} = \{s \in S(Y) : \Delta(x,Y) = \Delta^s(x,Y)\}$.

PROOF. We prove only some of the statements above; a full account is in ENGELER 1974.

The fact that $G^{L'}_{\Delta(x)}$ is a group is shown by observing closure under composition as follows: Assume that $\Gamma \cup \Delta(x,Y) \vdash \phi_1 \equiv \phi_1^{t_1}$ $\Gamma \cup \Delta(x,Y) \vdash \phi_2 \equiv \phi_2^{t_2}$ all ϕ_1, ϕ_2. Take $\phi_2 = \phi_1^{t_1}$ and notice that $\Gamma \cup \Delta(x,y) \vdash \phi_1^{t_1} \equiv \phi_1^{t_1 t_2}$, by logic then $\Gamma \cup \Delta(x,Y) \vdash \phi_1 \equiv \phi_1^{t_1 t_2}$. If $\Gamma \cup \Delta(x,Y)$ is L'-complete, i.e. if for all $\phi \in L'$ either $\Gamma \cup \Delta(x,Y) \vdash \phi$ or $\Gamma \cup \Delta(x,Y) \vdash \neg\phi$, then $G^{L'}_{\Delta(x)} = \{s \in S(Y) : \Delta(x,Y) = \Delta^s(x,Y)\}$. Namely: Suppose that $s \in S(Y)$ transforms $\Delta(x,Y)$ into itself, but that for some $\phi \in L'$ we have $\Gamma \cup \Delta(x,Y) \not\vdash \phi \equiv \phi^s$. Then by completeness, either ϕ and $\neg\phi^s$ are provable from $\Gamma \cup \Delta(x,Y)$, or symmetrically. In the first case we would have $\Gamma \cup \Delta^s(x,Y) \vdash \phi^s$ and hence $\Gamma \cup \Delta(x,Y) \vdash \phi^s$; a contradiction. Conversely, if $s \in G^{L'}_{\Delta(x)}$ then $\Gamma \cup \Delta(x,Y) \vdash \phi \to \phi^s$ for all $\phi \in \Delta(x,Y)$. But $\Gamma \cup \Delta(x,Y) \vdash \phi$ for all $\phi \in \Delta(x,Y)$, hence $\Gamma \cup \Delta(x,Y) \vdash \phi^s$, all $\phi \in \Delta(x,Y)$. By completeness of $\Delta(x,Y)$ we have therefore, $\Delta(x,Y) = \Delta^s(x,Y)$. □

For now we have done no more than define a group of permutations. What does it have in common with the group of automorphisms in the classical case? Our result above allows to make the connection: Let $\underline{A}(x)$, $\underline{A}(x,Y)$ be the minimal models of $\Gamma \cup \Delta(x)$, $\Gamma \cup \Delta(x,Y)$ respectively. (For such universal theories, these are simply the models on the Herbrand universe of x, $Y \cup \{x\}$.) Let $G(\underline{A}(x,Y)/\underline{A}(x))$ be the group of all automorphisms of $\underline{A}(x,Y)$ which leave $\underline{A}(x)$ pointwise fixed. Then $G^{L_0}_{\Delta(x)} \cong G(\underline{A}(x,Y)/\underline{A}(x))$, as we will presently show. Thus, we have recouped the old type of definition; $\underline{A}(x,Y)$ "splits ρ over $\underline{A}(x)$".

Clearly, any automorphism $s \in G(\underline{A}(x,Y)/\underline{A}(x))$ induces a permutation $s' \in S(Y)$ such that $\Delta(x,Y) = \Delta^{s'}(x,Y)$; this is just the meaning of automorphism. Conversely, if $s' \in S(Y)$ guarantees $\Delta(x,Y) = \Delta^{s'}(x,Y)$ then the map of the Herbrand terms in $\{x\} \cup Y$ can be obtained from s' and is an automorphism. □

Of course, we have not said anything yet about how to get the group from a problem statement. We expect this to be a major problem for any non-trivial Γ. Even the case of free monoids is completely open: what is the

group of

$$w_1 = w_2$$

if w_1, w_2 are words in the generators of the free monoid and some unknowns $y_1, \ldots, y_n$?

The case of classical Galois theory suggests that we seek a *resolvent*, i.e. a formula (polynomial) in the (unknown) solutions such that the group of ρ is just the group of permutations in the resolvent, which leave it fixed.

In the general case we know (non-constructively) at least that such a resolvent exists if the theory and the problem satisfy some additional requirements.

DEFINITION. $\rho(x,y)$ is of degree n over Γ if $\Gamma \cup \{\rho(x,y_i)\}_{i=0}^{n} \vdash \bigvee_{0 \le i < j \le n} y_i = y_j$.

THEOREM. *If* Γ *has the permutation property with respect to* ρ, *and if* ρ *is of degree* n *then there exists a finite conjunction of basic formulas* $\Theta(x,y_1,\ldots,y_n)$ *such that* $G^{L_0}_{\Delta(x)} = \{s \in S(y_1,\ldots,y_n) : \Gamma \cup \Delta(x,y_1,\ldots,y_n) \vdash \Theta \equiv \Theta^s\}$. *If*, *moreover*, Γ *is first-order, then there even exists a finite conjunction* ρ_0 *of basic formulas such that* $\Gamma \cup \Delta(x) \vdash \rho(x,y) \equiv \rho_0(x,y)$.

The proof of the first part is quite easy, all we need is to observe how to eliminate finitely many permutations from $S(y_1,\ldots,y_n)$ which fail, by not transforming some basic formula into an equivalent one, to belong to the group. The second part involves a use of the compactness theorem of first-order logic (hence the additional condition on Γ). □

What we learn from this theorem is that for $\underline{R}_{=}$, classical Galois theory can deal with all algorithmic problems of finite degree. This is open for $\underline{R}_{\le}$, whose algorithmic theory is not first-order.

OPERATING SYSTEM STRUCTURES by A.N. HABERMANN

MATHEMATICAL CENTRE TRACTS 63, 1975, 89-118

OPERATING SYSTEM STRUCTURES

A.N. HABERMANN

Carnegie-Mellon University, Pittsburgh, Pa. (USA)

1. INTRODUCTION

1.1. One may wonder why *Operating Systems* is considered a valid topic in its own right. In the past the primary reason has been that it was difficult to construct a system which operated reliably enough to satisfy its users. At present the subject is of interest to computer scientists because of the special nature of the problems it has spawn. There is in the first place the aspect of concurrency: a variable set of almost independent computations share the facilities and resources of a machine. The term *concurrency* has a connotation slightly different from *parallel computation*. The latter is used when a given computation is organized such that some parts are executed in overlapping time intervals. The former brings into the picture the question of how to design and maintain the proper environment in which a variable set of computations can be executed simultaneously.

Another reason why *Operating Systems* is (still) considered a valid topic is because of the size and variety of the programs involved. These aspects confronted us with the necessity of organizing the design of programs and developing a methodology for documentation, maintenance, modification and error detection, correction and recovery.

Finally, the design of operating systems generated a variety of interesting problems. To name a few, processor allocation stimulated queuing theory; storage management pushed the design of data structures and a whole spectrum of placement and replacement algorithms; the general problem of sharing devices led into the study of scheduling, system deadlocks and protection against unauthorized access.

It may be that *Operating Systems* will not anymore be seen as a coherent topic in the near future. On the one hand the drastic changes in machine architecture and the need for highly specialized systems make it difficult to see

what future operating systems will have in common that is not present in other large programs. On the other hand, the categories above, which are presently considered part of *Operating Systems* will be integrated into the general field of program construction. So, the title of the topic may loose its significance, the objects of research will remain, be it as part of another classification.

1.2. The function of an operating system essentially is to transform a given machine into a preprogrammed machine which is (meant to be) more convenient to its user community. An operating system is a set of programs which together realize such a transformation. One gets some feeling for the nature of such programs considering an oversimplified machine model consisting of a central processor (CP), a mainstore (MS), a card reader (CR) and a line printer (LP). Among the instructions which CP can execute are an input command and an output command. When CP executes an input command, CR places the information found on a card in a designated area of MS. Similarly, when CP executes an output command, a line image is taken from a designated area in MS and is printed on the line printer paper. The actual reading of a card (or printing of a line) is performed character by character, entirely controlled by a control unit which is part of the input or output device. Such a control unit must be activated by a command transmitted to the device controller by the central processor. Let us assume that the store is large enough to hold a user program, its data and the necessary control programs. A straight forward strategy for running user programs on this machine is given by this sequence of steps:

```
repeat read  card deck
       compile user program
       load user program + support programs
       execute loaded program
       print out results
until  machine halt
```

However, there is no built-in hardware which executes these steps, nor is there a mechanism which activates these steps in the desired order. There must apparently be a program already stored in MS which controls the sequence of steps. This program is the frame work of an operating system for the given machine and mode of running user programs. Traditionally, the compile, load and execute phase are not considered as part of an operating

system. But the programming of the transition from one phae to the next and also the input and output phases are part of the design task of an operating system.

1.3. Two very common types of operating system are a "batch system" and a "timesharing" system. The basic idea of a batch system is to assemble a series of user programs called a "batch", run those programs successively while in the meantime printing the output of a preceding batch and assembling a new batch. It is desirable to continue reading as fast as possible because an input device is very slow compared to a central processor. On the other hand, the size of an MS does not allow us to read arbitrarily far ahead. Also, output is at times generated much faster than it can be printed. This conflict was solved by adding a relatively slow, but very large, back-up store (BS) to the machine. When a new batch is assembled, the jobs are stored on this secondary storage device. The jobs are successively transferred to MS at the time that their compilation starts. Generated output is also stored on BS from where it is retrieved when the time has come to print it out.

A timesharing system gives its user direct access to programs and data stored in the machine via a teletype or scope terminal. Every user has his private working space and a set of data objects called "files". Files can be shared by several users. The system controls the terminal input and out put, it provides a set of basic operations on files and it protects a user's files against misuse by others.

Below follows a schematic description of a batch-system known as "Spooling System", because the first systems of this type used magnetic tapes as back-up storage.

Input path: CR → inbuf (in MS) → input batch (on BS) → READ buf (in MS)
Output path: WRITE buf (in MS) → output batch (on BS) → outbuf (in MS) → LP

Controlprograms: CRcontrol, LPcontrol, BScontrol, READ,WRITE

CRcontrol:

```
repeat  wait until CR is done with copying a card image into MS
        increment card count
        if current inbuf full then
        wait until other inbuf empty
        send request for dumping current inbuf to BScontrol
```

```
            switch current to other inbuf
            clear card count
        fi
            prepare next card command
            send command to CR device
    until   CR control halt
function READ=
            designate an MS area as next READ buf
            send load request to BScontrol
            wait until BScontrol signal, completion
BScontrol:
  repeat    wait until some transfer request
            i ← select one of the pending requests
            prepare transfer command on behalf of request_i
            send command to BS device
            wait until BS device is done
            signal completion of transfer to requestor_i
  until     BScontrol halt
```

It would lead us too far into questions of programming and design if we described a timesharing system model in a similar way. The description above supplies enough of a frame work to discuss matters of coordination, cooperation and communication among almost independent action sequences.

1.4. The term *multiprogramming* is used to describe a situation in which several partially executed computations are partly leaded in MS. Multyprogramming is possible independent of the number of central processors (we need at least one). A central processor can switch from one process to another through a hardware interrupt.

The term *process* has been very useful for the design and description of operating systems. It denotes the activity which is invoked when input is submitted to an *abstract machine* in its initial state. An abstract machine defines a mapping of a set of input data onto a set of output data. It consists of a pair {P,S} where P is a processor which is controlled by a given program, and S is a set of three states: the initial state, the busy state, and the final state. A process is the activity of an abstract machine which transforms a given input data set into a resulting output data set. The process terminates if the abstract machine reaches its final state.

The term *process* is often used to denote both process as described above and abstract machine. An abstract machine or process is called *parallel* if it can execute several instructions of its control program simultaneously; otherwise it is called *sequential*.
A process is called *I/O deterministic* in the case of a functional relationship between input and output. That is to say, the term applies to a process (or abstract machine) for which the output is uniquely determined by given input.

Examples of abstract machines are: a card reader, a central processor, the device controllers for card reader, line printer, back-up storage as described previously.

2. COORDINATION OF CONCURRENT PROCESSES.

2.1. Concurrency may cause race conditions if several processes operate on shared data in overlapping time intervals. For example, let processes $P_1, P_2, \ldots, P_n$ (n>1) operate on a common stack which is implemented as an array STACK [1:M] and a variable *top*. Initially all elements of the stack array have a meaningful value and top = M. The processes either take an element from the stack or return one. The number of returns by one process never exceeds the number of times it takes one. (The stack elements could for instance represent free storage frames.)
The operations on the stack are
take element (x) = T_1: x ← STACK[top]; T_2: top ← top-1
return element (x) = R_1: top ← top+1; R_2: STACK[top] ← x

The storage hardware protects against access conflicts. If two processes attempt to access a storage cell at the same time, the storage device accepts only one and allows the other to proceed after the first is done. So, operations on the stack elements in the programs above can not get confused. But the sequence of the operations still may cause trouble. The timing could for instance be

$$T_1;\ R_1;\ R_2;\ T_2$$

when one process calls take element (a) and another return element (b). The result is that the returned element gets lost while the element taken from the stack has not been properly removed. Other timings would also be disas-

trous; e.g. R_1; T_1; T_2; R_2 or T_1; T_1; T_2; T_2. Coherent pieces of program which must not be executed in overlapping time intervals are called *critical sections*. The set of all critical sections in a system can be partitioned in a set of classes of mutually critical sections. We mark the critical sections of one class by a unique bracket pair for that class. If we choose [ ] for the stack operation, we get:

take element (x) = [T_1: x ← STACK[top]; T_2: top ← top-1]

return element [R_1: top ← top + 1; R_2: STACK[top] ← x]

2.2. The rule that a process does not return more elements than it takes prevents overflow of the stack. However, a process should not try to take an element while the stack is empty (i.e. we require top ≥0). The operation take-element (x) is therefore replaced by the function
remove-element(x) = wait until stack not empty; take-element(x)
this arrangement makes that the processes cooperate in maintaining a correct state of the stack.

Question: *why can "wait until stack not empty" not be programmed as part of take-element*(x)?

Implementation of "wait until stack not empty" is not trivial.
Wrong is

<u>while</u> top = 0 <u>do</u> < nothing > <u>od</u>

because, when an element is returned, two processes may find the condition in the while clause false and proceed.

Two solutions:
a) embed remove-element in another critical section
remove-element(x) = {<u>while</u> top=0 <u>do</u> < nothing > <u>od</u>; take-element(x)}
so that only one element can be removed at a time.

b) use Dijkstra's P,V operations and apply to a semaphore "free" initialized at M.
remove-element(x) = P(free); take-element(x)
return-element(x) = [ ]; V(free)

```
Definition of P and V operation:
P(sem=semaphore) =
  if  (sem←sem-1) < 0 then
      stop calling process; put it on waitinglist(sem)
      j ← select process from readylist; start process j
  fi

V(sem=semaphore) =
  if  (sem←sem+1) ≤ 0 then
      j ← select process from waitinglist(sem)
      transfer process j from waitinglist (sem) to readylist
  fi
```

P,V on semaphore sem are themselves critical sections
Embed in bracket pair implemented by LOCK and UNLOCK:

```
type  lockbit =
      constant locked = 0, unlocked = 1
      operation LOCK(ref lb=lockbit) =
      begin local x = locked; while x = locked do exchange (x,lb) od  end
      operation UNLOCK(ref lb=lockbit) = lb ← unlocked
end
```

A semaphore is a triple: (semcount,semwaitinglist,semlockbit).
Critical sections can be programmed with LOCK and UNLOCK or with P,V operations, except for P,V themselves. The advantage of P,V operations is that waiting processes are not busy, whereas LOCKed processes are.

2.3. P,V operations as used in the preceding programs do not take care of deadlock situations nor of scheduling rules. These can be taken care of if a P operation is replaced by a critical section + a P operation on a so-called private semaphore. A V operation must then be replaced by a corresponding critical section. A private semaphore is a semaphore whose owner performs a P operation on it, but no other process does.
Let RV be a request vector and E an array of private semaphores. A version of remove-element(x) and return-element(x) in which deadlocks and scheduling can be taken care of is

remove-element(x) = [RV_i←1;considerallocationto(i)]; P(E_i)

```
returnelement(x) = [top←top+1; STACK(top)←x; free←free+1
                    if RV ≠ 0 then
                    j ← select from RV; considerallocationto j) fi]

where considerallocationto(i) =
   if  free > 0 and deadlockfree(i) then
       free ← free -1; x_i ← STACK[top]; top ← top - 1
       RV_i ← 0; V(E_i)
   fi
```

An alternative solution to deadlock and scheduling problems is a system in which a "supervisor" or "monitor" owns the critical data structure. In such a system all requests for modifications are addressed to the supervisor. Let RM be a vector indicating remove requests and RT a vector indicating return requests.

remove-element(x) = $RM_i \leftarrow 1$; V(monitorsem); $P(E_i)$

return-element(x) = $RT_i \leftarrow 1$; V(monitorsem); $P(E_i)$

Monitor:

```
            P(monitorsem)
   if     RT ≠ 0 then j ← select from RT
                      top ← top +1; STACK[top] ← x_j; free ← free +1
                      RT_j ← 0; V(E_j)
   else   j ←  select from RM; RM_j ← "spotted" fi

   while free > 0 and ∃_k RM_k = "spotted" do
         if allocationto(k) is deadlockfree then
         free ← free - 1; x_k ← STACK[top]; top ← top -1
         RM_k ← 0; V(E_k)
   od
```

2.4. On the other hand, cooperation implies in many case some form of scheduling and it can be used on purpose to implement desirable scheduling rules. However, if one writes the programs with P,V operations, it is often very difficult to verify that the desired scheduling is present and remains undisturbed.

There are three ways of approaching the problem of program correctness. The first approach tries to give an answer to the question: *"Given a program, prove that it is correct"*. This approach was first attempted by

R. FLOYD with his method of inductive assertions and later refined by C.A.R. HOARE with his axiom system representing language semantics. An other approach is the one which attempts to answer the question. *"Of which classes of program can I prove the correctness"*. This is the approach by SCOTT/DE BAKKER et al with their fixed point computations and also of PATERSON with the program schemata. A third approach tries to answer the question: *"How can I construct a correct program"*. This line of thought is followed by DIJKSTRA in his structured programming and also by DENNIS/HOLT et al. by means of PETRI nets.

We follow the third approach if we specify in a set of strict rules how program may be constructed for the problem we want to solve. We will establish such a set of rules for synchronizing concurrent processes. The synchronization of a process will be represented as a string of brackets of various types. The state of the system is given by an unordered set of open and close brackets. An initial state will be given. Execution of a process means that it attempts to place brackets into the state in succession according to its program. A process is allowed to move whenever it can, but these three rules restrict the possibilities:

R1: the state can always be modified by adding an open bracket
R2: a close bracket can be added if and only if there is a matching open bracket in the state(if it cannot move, it must wait)
R3: a matching bracket pair in the state cancels out
this rule allows us to forget part of the history)

```
Example 1:  P = ] f {
                                initial state: [
            Q = } g [
```

Q cannot move in the initial state, but P can. When P is done, Q can move, but P cannot. The model represents the interaction between device and control process. Note that there may be many Ps and Qs.

```
Example 2:  P = ] } f { [
                                initial state = [{
            G = } g {
```

Q has priority over P, because if a Q is waiting while a P is executing f, this P enables a Q to proceed before it enables another P.

Example 3: (Cigarette Smokers Problem)

```
Sources: S1 = ] ( (   S2 = ] ( (   S3 = ] ( (
                r w          r b          w b

Agency:  R = ) selector+4{    W = )selector+2{    B = )selector+1{
             r          set      w          set      b          set

                                                  selector:

Sinks: RW = } selector ← 0 [          initial state: [
         set=6

       RB = } selector ← 0 [          provable is: the regular expr
         set=5
                                      (S_xy;A_x,A_y;Sink_sel)*
       WB = } selector ← 0 [
         set=3                        represents the system's behavior.
```

3. COMMUNICATION AND SYSTEM DEADLOCKS

3.1. A communication path between two processes can either be fixed for all times, or it can be established for the exchange of one piece of information, or it can be created for a sequence of information exchanges. The communication of a device control process and its peripheral device is an example of a fixed communication path. The second mode of communication is known as *message switching* or a *mailingsystem*. A sender process places a message in a queue. Receiver processes remove messages from the queues. An extended, multiprogramming, version of the batch system discussed earlier could be designed with a mailing system for transfers between the two storage levels. The sender processes would be the processes which request the transfers, the receiver would be the BScontrol process. The third class of communications could be named the class of *dialogues*. Here a sender process monopolizes a receiver process for an indefinite length of time. Processes communicate with peripheral devices such as magnetic tape units in this manner.

In a mailing system senders and receivers operate on the *Communication buffer* by means of the operation *deposit* and *receive*. The buffer provides space for several messages so that the senders may get somewhat ahead of the receivers. If the buffer is implemented as an array B[1:N] where N is the upperlimit of the number of messages which are permitted to be in the buffer we may use two pointers *front* and *rear*, both initialized at zero, pointing respectively to the frame from which the last message was taken and the frame in which the latest message was placed.

Let ⊕ denote addition modulo N; this operation is used for advancing the pointers front and rear.

```
deposit(m=message) =                receive(m=message) =
    P(numframe)                         P(nummes)
    {rear ← rear ⊕ 1                    ∑front ← front ⊕ 1
d3:  B[rear] ← m }                  r3:  m ← B[front]
    V(nummes)                           V(numframe)
```

Semaphore numframe is initialized at N and semaphore nummes at zero. Deposit may proceed only if there is room in the buffer for another message. Receive may commence only if there are messages in the buffer.
If there are many senders, it is necessary to cast operation deposit in a critical section because of the use of the rear pointer likewise for using the frontpointer if there are many receivers.

It is not necessary to put the operations on the buffer in deposit and receive in critical sections using the same semaphore. This means that simultaneous operations by senders and receivers can be allowed. One can prove that the pointers rear and frame never point to the same frame when the statements d3 and r3 are executed simultaneously. This can easily de derived from the bracket representation:

```
S = ] }  d3:...{(          initial state: [[...[
                                          {∑
K = ) ∑  r3:...∑[
                                          N
```

SKETCH OF A PROOF.
Let rear have the value r and front the value f; let "#..." mean the number of ... added to the state.
Let r^1 and f^1 be the number of times that rear and front were incremented.
The equality r = f is equivalent to $r^1 = f^1 \bmod(N)$.

At the time that a sender executes statement d3 and a receiver statement r3 we have

$$\#\,] \geq r^1 = 1 + \#(\quad \text{and} \quad \#) \geq f^1 = 1 + \#[,$$

because all but the last sender added an open parenthesis to the state and all but the last receiver added an open square bracket. The bracket rules enforce $\#(\geq\#)$ and $\#[\geq\#] - N$, so

$$r^1 = 1 + \#(\geq 1+\#) \geq 1 + f^1 \text{ and}$$

$$f^1 = 1 + \#[\geq 1+\#] - N \geq 1 - N + r^1 \rightarrow r^1 \leq f^1 + N-1$$

Thus, $1 + f^1 \leq r^1 \leq f^1 + N - 1$, so $r^1 = f^1 \bmod(N)$ is not true.

3.2. In a dialogue communication system deadlocks may occur.
Let a machine have six magnetic tape units $MT_1,\ldots,MT_6$ and assume there are users of three different types:
a type P user takes one MT at a time;
a type Q user starts using one MT, asks later for another one and releases the units in either order;
a type R user needs at some time three MTs simultaneously. A deadlock occurs if users R_1, R_2 and R_3 are holding two MTs each. In that case everyone of them is going to ask for another MT and no one is willing to give up an MT yet.

The problem can be solved if we program the classes P, Q and R as follows:

P =) MT (
Q =]) MT_1) MT_2 (([
R = }]) MT_1]) MT_2) MT_3 (([([{

with initial state:

initial state: ((((((
[[[[[
{ { { {

Three users of type R each holding two MTs would not be possible, because they would have placed six close square brackets in the state whereas there are only five open square brackets.

The problem presented is one in the category of *one-type of resource deadlock*. A general statement of the problem is: a set if n (n>1) competing processes $C_1,\ldots,C_n$ use resources $R_1,\ldots,R_t$ (t>1). Given for all $i \in \{1,\ldots,n\}$ that C_i ultimately needs $claim_i$ resources and is using at the moment $alloc_i$ resources, test whether the given allocation state may degenerate into a deadlock state.
We assume that $0 \leq alloc_i \leq claim_i \leq t$ for all $i \in \{1,\ldots,n\}$. A state for which this relation holds is called *realizable*. Define $rank_i = claim_i - alloc_i$. A process C_i is entirely satisfied if $rank_i = 0$. We assume that it will release the resources allocated to it at some time after that.
The value of $rank_i$ is also in the range $[0,\ldots,t]$.

Let x_k be the number of processes for which rank < k ≤ claim. We call this number the number of *promotions* in rank = k. Vector $\underline{x} = (x_1, x_2, \ldots x_t)$ describes the promotions in all ranks.
Define

$$p_k = \sum_{j=k}^{t} x_j \text{ and vector } \underline{p} = (p_1, p_2, \ldots, p_t).$$

Define

$$\underline{t} = (t, t-1, t-2, \ldots, 1); \text{ vector } \underline{t} \text{ is constant}$$

THEOREM. *the relation* $\underline{p} \leq \underline{t}$ *is a necessary and sufficient condition for a safe state. (safe* in the sense that the state cannot degenerate into a deadlock state).

SKETCH OF A PROOF.
Promotions counted in p_k for given $k \in \{1, \ldots, t\}$ helped competitors with a claim ≥ k to reach at most rank = k - 1. Processes which reached rank = k - 1 are able to get all the resources needed if and only if at least k - 1 resources are available. Every promotion counts for one allocation, so the number remaining after the promotions counted in p_k equals $t - p_k$. So, the processes at rank = k - 1 are able to finish if and only if

$$t - p_k \geq k - 1, \text{ or } p_k \leq t + 1 - k, \text{ or } p_k \leq t_k$$

Thus, all processes are able to finish if and only if $\underline{p} \leq \underline{t}$

Let y_k represent the number of competitors starting with claim = k.
Define

$$n_k = \sum_{j=k}^{t} y_k \text{ and } \underline{n} = (n_1, n_2, \ldots n_t)$$

Of course, $\underline{n} \leq \underline{p}$. Thus, $\underline{n} \leq \underline{t}$ is a necessary condition for safe states. The interesting point is that $\underline{n}$ does not depend on the allocation, but solely on the set of competitors. Vector $\underline{n}$ changes only if another process is admitted to this set or if one departs. The test $\underline{n} \leq \underline{t}$ could nicely be applied as an *admission test*. We may expect to make higher scores of success in the safety test $\underline{p} \leq \underline{t}$ if the admission test rules out sets of competitors which generate only unsafe states.

It turns out that the total number of possible values for $\underline{n}$ is $T = \binom{2t}{t}$,

whereas the number of acceptable values is only $A = \binom{2t}{t} - \binom{2t}{t+2}$ for $\left(t \geq 2\right)$ The usefulness of applying an admission test is shown by the fact that $\frac{A}{T} = C\left(\frac{1}{t}\right)$. This says that the acceptable states constitute a decreasing fraction of the total number of possible states when t increases.

One can show that not all states are reachable if the admission test is applied. The number of reachable states is $R = \binom{2t}{t} - \binom{2t}{t+3}$. It nevertheless seems useful to apply and admission test, because $\frac{A}{R} = \frac{4}{9} + C\left(\frac{1}{t}\right)$

3.3. Define

$$N_k = \sum_{j=k}^{t} n_j \text{ and } \underline{N} = (N_1, N_2, \ldots, N_t)$$

One can easily show that $\underline{p} \leq \underline{N}$, so $\underline{N} \leq \underline{t}$ is a *sufficient* condition for safe states.

Unfortunately, $N_1 = \sum_{1}^{t} n_j = \sum_{j=1}^{t} \sum_{i=j}^{t} y_i = \sum_{j=1}^{t} j * y_j$ = total claim

So $N_1 \leq t_1$ implies that total claim $\leq t$.

This says that requiring $\underline{N} \leq \underline{t}$ is equivalent to requiring that the processes together claim not more than the available number of resources. However, if the processes do not use their claim number of resources all the time, such a requirement would be very restrictive and leave a certain number of resources idle. Moreover, the processes would be restricted with respect to concurrency and this causes degeneration in throughput.

A weaker condition is more helpful.

THEOREM, *if there is a* $k \in \{1,\ldots,t\}$ *for which* $N_k \leq t_k$ *then* $N_l \leq t_l$ *for all* $l \in \{k,\ldots,t\}$

SKETCH OF A PROOF (by induction)

$$N_k = N_{k+1} + n_k \leq t_k = t + 1 - k \rightarrow N_{k+1} \leq t + 1 - k - n_k$$

if $n_k = 0$ then $N_k = 0$ and $N_{k+1} = 0$, so $N_{k+1} \leq 1 - k$ is true

if $n_k > 0$ then $n_k \geq 1$, so $N_{k+1} \leq t + 1 - k - 1 \leq t - k$.

In applying the admission test and safety test it is advantageous to remember the smallest index k for which $N_k \leq t_k$. The test $\underline{n} \leq \underline{t}$ or $\underline{p} \leq \underline{t}$ can be broken off after index k - 1, because $n_l \leq N_l \leq t_l$ and $p_l \leq N_l \leq t_l$ for $l \in \{k,\ldots,t\}$.

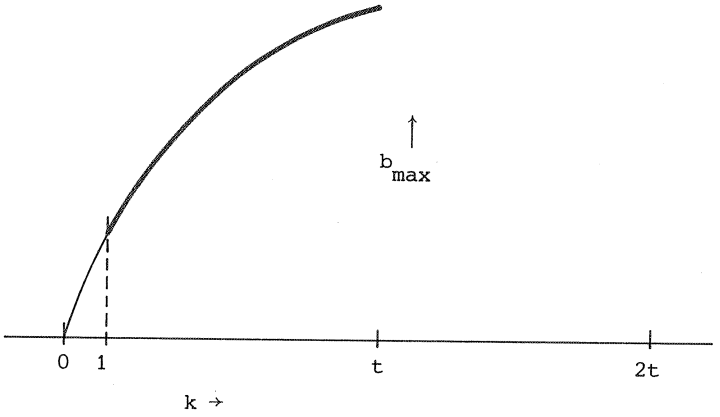
Define the *booking factor*

$$b = \frac{\text{total claim}}{t} = \frac{N_1}{t} = \frac{\sum_{i=1}^{k-1} n_i + N_k}{t}$$

where k is the smallest index for which $N_k \leq t_k$.
The upperbound b_{max} for b depends on the index k:

$$b \leq \frac{1}{t}\left[\sum_{i=1}^{k-1} n_i + N_k\right] \leq \frac{1}{t}\left[\sum_{i=1}^{k-1} (t+1-k) + (t+1-k)\right] \leq \frac{1}{t}\sum_{i=1}^{k} (t+1-i)$$

$$\leq \frac{1}{t}\left[(t+1) * k - \frac{1}{2} k * (k-1)\right] \leq k\left(1 - \frac{k-1}{2t}\right) = b_{max}\ (k)$$



If the smallest index for which $N_k \leq t_k$ equals one, $b_{max} = 1$. This says that if the total claim is not permitted to be larger than the total available number of resources, then the booking factor should not be permitted to be larger than one. This confirms what we expect. The value b_{max} can be used to force a reasonable cut off point in the tests, because it can be used to set a value for the index k.

4. STORAGE MANAGEMENT

4.1. Basic to all automatic storage management systems is the notion of *virtual address*. A virtual address is the position of a program instruction or data item relative to a chosen base address. Code generated by a compiler or written in an assembly language by a programmer is entirely expressed in terms of virtual addresses. The purpose of using virtual addresses is to

make the program and its data completely independent of the specific locations where it resides when being executed. This gives the operating system the flexibility of moving a program and its data from one area to another without affecting the logic of the program. The value of a base address is not specified by a compiler or assembly language programmer, it is entirely under control of the operating system.
The virtual addresses used in a program are interpreted at execution time and are mapped into physical locations. How the mapping is performed depends on the chosen address translation mechanism of the computer in use. Address mapping is in most machines partly done in hardware, so that the *"normal case"* is handled efficiently. The most common storage management strategies are: swapping, segmentation and paging.

Swapping: a virtual address is a positive integer va; the address map is defined by `loc ← Base + va,` where Base is set by the operating system before execution commences. In addition to the Base an upper limit is set for a program and its data and the address map checks for va overflow.
Segmentation: a virtual address is a pair (s,w), where s is an index pointing into a "segment table", ST, and w an offset within a segment. The segment table maps the segment indices onto physical base locations.
The address map is

```
loc ← ST[s] + w
```

The mapping hardware tests for segment table overflow (by s) and segment overflow (by w).
Paging: a virtual address is also a pair (p,w), where p is an index pointing into a "page table", PT, and w an offset within a page. The page table maps the page indices onto a fixed set of physical base locations which are a multiple of the chosen "page size".
The address map is

```
loc ← PT[p] + w
```

Overflow tests can entirely be avoided if the lay-out of a virtual address is fixed and page tables have a fixed length. This makes the mapping much faster than in case of segmentation.

4.2. The address translation of a swapping system is much faster than that of a segmentation system. However, a segmentation system has the advantage that

a) not all segments of a program and its data have to be loaded in MS. This saves transfer time in both directions
b) segments not written into don't have to be copied on BS
c) it is easier to fit smaller pieces together in MS than large ones.

Swapping and segmentation have in common the phenomenon of *"external fragmentation"*.

When segments of various sizes have been loaded and removed, there are some unused pieces of storage space scattered through store. One gets some feeling for the amount of space wasted through external fragmentation using Runth 's 50% rule. This rule says that the number of holes (= unused pieces of storage) is on the average half the number of blocks in use. If the average hole size equals k * average used block size, then an estimate of the fraction wasted is

$$\frac{\frac{1}{2} * N * k * a}{\frac{1}{2} * N * k * a + N * a} = \frac{1}{1 + \frac{2}{k}}$$

For k = ½ we find that as much as 20% of the store may be wasted!

4.3. When a segment is to be loaded in MS, a hole must be found which is large enough to accomodate the segment. Two most obvious algorithms for doing this are the *first fit* algorithm and the *best fit* algorithm. The former starts at some hole and steps through the set of holes until it finds one large enough. The latter inspects all the holes and selects the smallest one which is large enough for the given segment. The first fit algorithm performs especially in a satisfactory manner if it remembers where it left off the last time and starts searching from there the next time. If it would start always at the beginning, the small pieces tend to accumulate at the front end of the list.

Experimental data has shown that a well-organized first fit algorithm outperforms a best fit algorithm.

When a block of used space is released, it must be added to the list of holes. If the list is not ordered, the hole can be appended at the end of the list. However, this has the disadvantage that it is very hard to merge the new hole with a neighboring hole (if any). If the list is ordered by ascending store address, such neighboring holes can be detected and

merging can take place right away. In that case the release function has this structure:

```
release(address,size) =
            go down the list of holes until address < ADDRESS (next hole)
                                        or  nexthole = NIL
            insert given new hole in the list
            if adjacent to left neighbor hole, merge with leftneighbor
            if adjacent to right neighbor hole, merge with rightneighbor.
```

One could adopt yet another strategy and not bother about merging in function release so as to speed up the execution of this function. By the time that the placement algorithm cannot find a hole large enough, this algorithm goes through the list and merges adjacent holes as far as necessary.

Yet another approach is never to merge holes, but instead *compact* when need be. If the placement algorithm cannot find a hole large enough, it moves all the blocks in use to one end of MS (assuming the code is relocatable!). This procedure automatically leaves one maximal hole at the other end of MS. Experience with this strategy is generally not encouraging.

4.4. All management systems have in common the fact that space must be created in MS if not enough is available when a segment or page must be placed into MS. In case of paging, the size is fixed, so it is always sufficient to throw one page out for every page that must be loaded into MS. Algorithms that select a page or segment to be thrown out of MS are known as *replacement algorithms*.

The best algorithm one can imagine is BELADY's algorithm. It selects the page with the largest future reference interval; that is to say, it picks out the page which will be referenced later than any other page presently in MS. Unfortunately, this algorithm cannot be realized in practice, because the future reference string is (probably) unknown.

Other algorithms which have been investigated are

a) random; this algorithm is based on the belief that the future reference string is unpredictable. This, however, turns out not to be true.
b) Round Robbin (RR); this algorithm is based on the idea that page frames are used for almost equal length in time. The algorithm employs a pointer which cycles through the page frames and it is believed that by the time

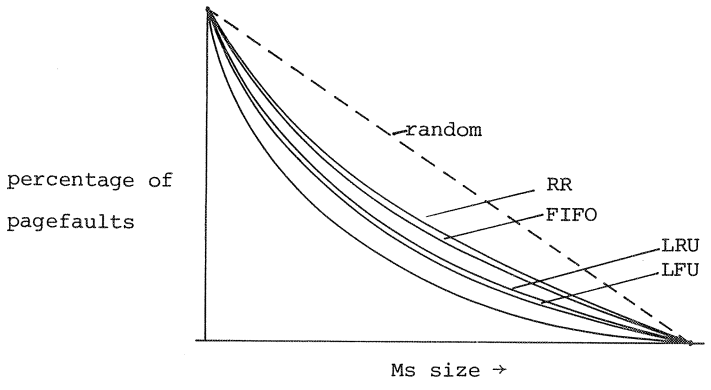
the pointer returns to a frame, the page in that frame is likely to be of little interest.

c) First In First Out (FIFO); this algorithm selects the page which has been in MS longer than any other page. It allows for pages to be discarded and frames to be released (which RR does not do), but it assumes that the interest in a page is a uniform and descending function of its lifetime.

d) Least frequently used (LFU); here the idea is that a page which has not frequently been referenced in the past will also not be referenced frequently in the future. Therefore, the page which will probably be referenced later than any other page is in all likelihood the least frequently referenced page.

e) Least recently used (LRU); this algorithm is based on the assumption that the least recently referenced page is probably the least frequently used page or a page which is of no interest anymore. It assumes that the probability of referencing a given page in MS is proportional to the length of time this page has not been referenced till the present time.

Many experiments have been carried out to measure the performance of these algorithms. The objective is to minimize the number of page turns. The results of these experiments are unanimous in the sense that LFU and LRU show a significantly better performance than RR or FIFO. The graphs one usually sees are like the one plotted below.



However, there is a question of how hard it is to implement the algorithms. The implementation of FIFO or RR is clearly trivial. The implementation of LFU is rather inefficient. Every time a page is referenced, a count must be incremented and when the replacement algorithm is activated

all pages must be scanned after computing their frequency of use. This overhead has shown to be too elaborate and the algorithm is therefore not applied anymore in present day systems.

A straight forward implementation of LRU is not much better. Every time a page is referenced, it must get a time stamp and when the replacement algorithm is invoked, it must find the oldest page. Contemporary machine architecture, however, enables an acceptable approximation of LRU which can be implemented quite efficiently. The address mapping is considerably improved if the machine uses a small associative memory AM. The AM contains a few copies of page table entries. We call a page which has a descriptor in AM "active" and the others "passive". An entry in AM consists of a key and a value. AM responds to the question: give me the value of the entry whose key is x. Address translation takes place as follows (for given virtual address(p,w))

if (Temp ← AM [p]) ≠ 0 then loc ← Temp + w
else addressexception fi

so that the most likely case does take a minimal amount of time. Addressexception occurs if the page descriptor is not in AM. At that time the page descriptor must be copied in AM at the expense of removing one which is presently in AM.

The LRU can be approximated if we consider the moment that a page is removed from AM as its last reference. Since we are interested in the relative age of a page, the page which is removed from AM is appended to a list of "used pages". The pages longest in the list is selected in the replacement algorithm. If a page in the "used page" list is referenced again, it is removed from the list and its descriptor is copied into AM. The list operations can be designed very efficiently if page indices are used.

5. SCHEDULING POLICIES

5.1. A scheduling policy is applied if several processes wish to use a set of resources while not all resource requests can be satisfied simultaneously. Scheduling algorithms can be classified in two major groups: one consisting of algorithms which take into account the history of a process with respect to using the resource set and an other one in which such history is forgotten. The latter group consists primarily of two very simple disciplines, RR and FIFO. The FIFO algorithm is particularly suitable in those cases in

which no other scheduling criterion is relevant. It can fairly be stated that FIFO is considered as the *default* scheduling rule.

Scheduling is particularly relevant in case of *preemptive resources*. These are resources such as a central processor or main storage of which the system can decide to take them away from the user without disturbing the logic of the user's computation. Whether FIFO or RR is the suitable discipline for scheduling a preemptive resource type depends on the service times needed by the users (this term is here used in the sense of *process*). The total time which elapses between a resource request and release of that resource is known as the *response time*. This time depends on the service time the user needs and on the time a user has to wait until the resource becomes available.

One can easily show that, if a FIFO rule is applied, the response time depends on the number of waiting users and the average service time. Such a discipline works very much against the *small* users which need only a short service time. The RR discipline does not have this disadvantage, it guarantees a user a response time proportional to the service time needed. On the other hand, the overhead caused by applying an RR discipline is much higher than for a FIFO discipline, because the resource is preempted more frequently and it takes time to deallocate and reallocate a resource. So, if users vary widely in needed service time, an RR discipline is to be preferred, but, if not, FIFO performs more satisfactorily.

5.2. We discuss some history-minding scheduling disciplines for using a central processor. First we discuss a linear priority scheme, then a weighted CP utilization discipline and finally deadline scheduling.

A linear priority scheduling discipline attaches to every user a priority which is a linear function of time. While the user runs on a CP, its priority changes by

$$\Delta p = a\ \Delta t$$

and while the user is waiting, its priority changes by

$$\Delta p = b\ \Delta t, \text{ where } b \neq 0.$$

Let $C_i(t)$ be the characteristic function of $user_i$ describing whether $user_i$ is running or not. If $user_i$ enters the system at $t = t_0$ with an initial priority = 0, the priority at $t > t_0$ is

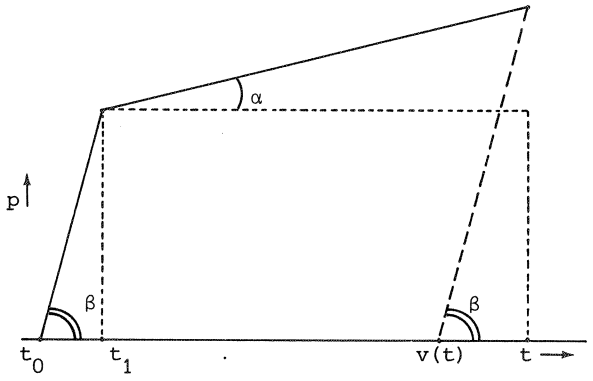
$$p(t) = a \int_{t_0}^{t} C_i(u)\ du + b \int_{t_0}^{t} [1-C_i(u)]\ du$$

or

$$p(t) = b(t-t_0) + (a-b) \int_{t_0}^{t} C_i(u)\ du$$

This priority function would not be acceptable for an implementation because of the fact that it has to be updated for all users every time that the function is used, no matter whether the user is running or waiting. We require that an implementation satisfies the rule that the priority must be updated only for a running user by the time it stops running, but not for a waiting user.

A function which satisfies the implementation rule is the *virtual arrival time*. This is the moment in time that a user would have started so as to reach its current priority if it had been waiting all the time. In the picture below the virtual arrival time is found as the parallel projection of the current priority on the t-axis.



a = tan α
b = tan β

t_0 = entry time
t_1 = starting to run
t = present

$b * (t-v_t) = p$ and $b * (t_1-t_0) + a(t-t_1) = p$, or in more general terms,

$$p = b\ (t-t_0) + (a-b) \int_{t_0}^{t} C_i(u)\ du$$

Hence

$$v_t = t_0 + \left(1-\frac{a}{b}\right) \int_{t_0}^{t} C_i(u)\ du$$

The virtual arrival time satisfies the implementation rule, because

$\Delta v = \left(1-\frac{a}{b}\right) \Delta t$ for a running process and

$\Delta v = \left(1-\frac{a}{b}\right) * o$ for a waiting process.

The scheduling algorithm selects the process with the left most virtual arrival time on the t-axis. Which one this is depends on the value of the constant $\left(1-\frac{a}{b}\right)$. The ratio $\frac{a}{b}$ determines the speed at which the virtual arrival time moves along the t-axis.

Let $0 < a < b$. In this case $0 < \frac{a}{b} < 1$, so $0 < \left(1-\frac{a}{b}\right) < 1$. Consider a period of time in which no new processes arrive while several processes are either waiting or running. The virtual arrival time of waiting processes does not change, but that of a running process moves to the right (and must have been to the left of the others when it started running). After a while the virtual arrival time of a running process will overtake the virtual arrival time of a waiting process. At that moment time has come to stop the running process and allocate a resource to the waiting process. The speed of overtaking is determined by the ratio $\frac{a}{b}$. It is low if a is close to b and high if b is much greater than a. The scheduler behaves as an RR discipline. However, if a new process arrives, it is placed on the t-axis to the right of all others. So it starts waiting and it may take a while before the current group of users overtakes the newly arrived process. This is why KLEINROCK called this scheduling strategy the "selfish RR" discipline; it is asif the current users try to keep the resource for themselves and admit a newcomer reluctantly.

Let $0 < b \leq a$. In this case $\left(1-\frac{a}{b}\right) \leq 0$. The virtual arrival time of a running process travels this time to the left. Since the virtual arrival time of a waiting process does not change, the running process remains the one with the left most virtual arrival time. So, in this case the policy is a pure FIFO discipline.

Other policies emerge if the domain of a and b is chosen differently. An additional convenience of applying one of the possible linear scheduling

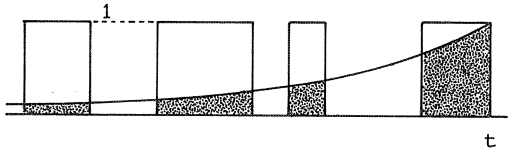
disciplines is the fact that the ratio $\frac{a}{b}$ can be used as a *tuning parameter* of the system. The value can be varied until the desired optimal performance in a given system is achieved.

5.3. The weighted CP utilization discipline takes into account how long ago a process was using the resource. Let the resource be a single central processor and let $C_i(t)$ be the characteristic function describing whether or not process$_i$ is running. In a single processor system, $\sum_{i=1}^{n} C_i(t) = 1$.

Define

$$f_i(t) = a \int_{-\infty}^{t} C_i(u)\, e^{a(u-t)}\, du \quad \text{as the weighted CP}$$

utilization measure for process$_i$. The value of $f_i(t)$ is the sum of the colored areas in the picture. $\sum_{i=1}^{n} f_i = 1$



Unfortunately, $f_i(t)$ does not satisfy the implementation rule, because for a waiting process:

$$\Delta f_i = a\, e^{-a(t+\Delta t)} \int_{-\infty}^{t+\Delta t} C_i(u)\, e^{au}\, du - a\, e^{-at} \int_{-\infty}^{tm} C_i(u)\, e^{au}\, du \neq 0$$

Define

$$F_i(t) = a \int_{-\infty}^{t} C_i(u)\, e^{au}\, du \quad \text{and} \quad S(t) = a \int_{-\infty}^{t} e^{au} = e^{at}$$

$F_i(t)$ satisfies the implementation rule, because

$$\Delta F_i = a \int_{t}^{t+\Delta t} C_i(u)\, e^{au}\, du = 0 \quad (\text{since } C_i(u) = 0 \text{ for a waiting process})$$

The change of $F_i(t)$ for a running process is given by

$$\Delta F_i = a \int_{t}^{t+\Delta t} e^{au}\, du = \Delta S = S * (e^{a\Delta t} - 1) \approx S * a * \Delta t$$

Thus, when a process stops running, we compute ΔS and add this value to $S(t)$ and to $F_i(t)$, where P_i is the stopped process.

The rate at which waiting process "ages" (i.e. the rate at which its CP utilization measure decreases in time) is determined by

$$\frac{f_i(t+\Delta t)}{f_i(t)} = \frac{F_i(t+\Delta t)}{F_i(t)} \cdot \frac{S_i(t)}{S_i(t+\Delta t)} = 1 * \frac{e^{at}}{e^{a(t+\Delta t)}} = e^{-a\Delta t} \approx (1-a\Delta t)$$

The parameter a can be used to tune the behavior of the scheduler. In one time unit f_i is reduced to (1-a) times the original value.

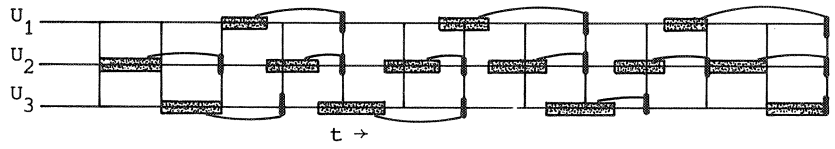
5.4. The objective of a deadline scheduler is to guarantee every process a negotiated fraction of CP time. E.g. it may be decided that the system consists of a batch process which receives 20% of the CP time and a time-sharing subsystem which receives 80% of the CP time. However, it is not sufficient to specify just these fractions, we must also specify within which time limit such a fraction must be allocated. Without this specification we could allocate 80% of every hour to the time-sharing system and 20% of every hour to the batch process. This would have the awkward effect that the time-sharing users might wait for a full 12 minutes if the batch process uses its fraction in one contiguous time interval!

Define for every user U_i a "cycle time" C_i within which the promised fraction of CP time must be allocated. The value of C_i may be in the order of one hour for the batch process and in the order of one second for the time-sharing system.

A user appears to the deadline scheduler as a triple (d_i, c_i, f_i) where d_i is the next deadline for user U_i. The quantity d_i is an absolute moment in time (a future moment) at the end of the first interval of length c_i for which the fraction f_i has not yet been entirely allocated.

The scheduler follows this policy: the resource is allocated to the user whose deadline is the closest. This user receives an amount of $f_i * c_i$ of CP time so that this user is satisfied until his next deadline. When the amount has been used, the deadline of this user is incremented by c_i and the CP is allocated to the next user whose deadline is the closest. If there is a tie, the order of allocation is immaterial.

Example: $U_1 = \left(d_1, 4, \frac{1}{6}\right)$, $U_2 = \left(d_2, 2, \frac{1}{2}\right)$, $U_3 = \left(d_3, 3, \frac{1}{3}\right)$



Moves:	U_2	U_3	U_1	U_2	U_3	U_2	U_1	U_2	U_3	U_2	U_1	U_2	U_3
next deadline:	4	6	8	6	9	8	12	10	12	12			

$\sum_{i=1}^{n} f_i \leq 1$ must of course be true, otherwise CP time would be over committed.

6. SYSTEM DESIGN ISSUES.

6.1. Every process must be protected against errors which may occur in an other process. That is to say, an error occurring in one process should not destroy valid information in another process. This means in the first place that one process should not have arbitrary access rights to information which is part of another process (a process should, for instance, not write outside its allocated storage area). It means furthermore that the integrity of data shared by concurrent processes must be preserved. Shared data must either not be accessible at a given moment, because it is being modified, or it must be in a well-defined state, recognizable by any process which has a right to access it. In this last lecture we focus on the latter point of preserving the integrity of shared data structures.

The integrity can be preserved if processes can not perform arbitrary operations on a shared data structure. Instead, the set of possible operations must be precisely specified and the processes must be forced to use these operations and no others. Programming languages provide since long a very powerful tool for specifying such operations, viz. procedure declarations. Such a declaration allows the designer to separate the specification and the implementation. At the call site the implementation is irrelevant, all that matters is the specified effect of the procedure call. At the place of the definition, the implementation may be modified provided that the specifications do not change. This arrangement greatly enhances debugging, code improvement and other modifications.

We need a tool for specifying datastructures analogous to the procedure declaration used for specifying operations. Such a tool is a type definition (or a class definition in SIMULA 67). A type definition describes the structure of a class of objects and it defines the operations which can be performed on objects of such type. The structure of the typed objects is internal to the type definition, i.e. outside the type definition one has no access to the internal structure of a typed object. (this kind of protection is unfortunately missing in the class concept of SIMULA 67, but this is the only thing missing. The SIMULA classes are entirely adequate in all other respects.)

A typed object behaves outside the type definition as "atomic". Only the names of the operations are exported outside the type definition. The type definition protects a datastructure against errors or malicious use and it allows us to implement a characteristic behavior of a data structure as part of its definition instead of as part of the calling sequences.

For example, a process in a concurrent system can be on the waiting-list of a semaphore, or it can be in the readylist or it may be running. We wish to implement the rule that a process may be on one list at a time only. In a conventional approach to this problem one would reserve a link field in every process control block and use this linkfield to link processes together in various lists. One would have to check the programs to make sure that it can never happen that two different linkfields point to the same process.

Instead we define

```
type process =
 local prior = integer (0); ref succ = self
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
operation exchlink (ref p,q = process) =
 begin ref x = p.succ
   p.succ ← q.succ; q.succ ← x
 end
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
end
```

The only way to modify a successor field outside the type definition is through a call of procedure exchlink. One can easily see that the type definition preserves the property that *succ* defines a permutation of the

processes, so the rule is satisfied.

Another example: the receiver queue. Senders transmit messages to receivers. The senders are allowed to get arbitrarily far ahead of the receivers (the message queue has no upperbound). The receivers must wait as long as the message queue is empty.

```
type  mlist =
    local  head = nullmessage, length = integer (0),
        listempty = lockbit (locked), listlock = lockbit (unlocked)
    comment a definition of lockbit was given on page 10
    proc append (ref m = message, ref ml = mlist) =
    begin
         LOCK (ml. listlock)
         ≪ put message (m) at the end of list (ml) ≫
         if (ml.length ← ml.length + 1) = 1 then UNLOCK (ml.listempty) fi
         UNLOCK (ml.listlock)
    end

    proc remove (ref ml = mlist) = ref message
    begin
         local cond = Boolean (false)
         repeat LOCK (ml.listlock)
             if (cond ← ml.length > 0) then
                  ml.length ← ml.length - 1; ≪ remove ← first of list(ml) ≫
             fi
                UNLOCK (ml.listlock)
         until cond do
                LOCK (ml.listempty)
         od
    end remove
end type mlist
```

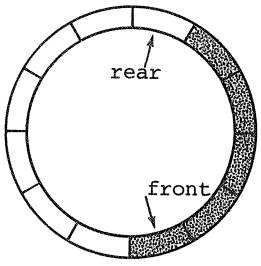
The desired properties are introduced as part of the type definition and are preserved independent of where such a message list is used.

6.2. In addition to specifying exactly the set of operations, it may also be necessary to apply such operations in some order. For example. one cannot start reading a file unless an open file command has first been given.

A tool for specifying the order of executing operations on a shared datasructure is a so-called *path expression*. In its most simple form, the syntax of a path expression is a list of steps separated by semicolons and surrounded by the keywords path and end. A step is a list of operation names separated by commas. More elaborate constructs are still under investigation.

Example: ringbuffer

We define first a type oneslot buffer
and then the type ring buffer




```
type  oneslot buffer =
    local slot = message; path write; read end

    Operation write (- -) = - - - -

    Operation read (- - ) = - - - -
end
```

The given path enforces alternating executions of write and read.

```
type ringbuffer (N=integer) =
     local ring = [N] oneslot buffer
     local front, rear = integer (0)
path advancefr; copyfr end; path advancerr; copyrr end


operation deposit (m = message, ref rb = ringbuffer) =
     begin advancerr (rb); write (m,rb.ring[copyrr(rb)]) end
operation receive (- - ) = - - - -
end type ringbuffer
```

The paths assure that the front pointer cannot be advanced twice in a row, nor can it be copied twice in a row. So, senders calling deposit will access successive ringbuffer slots. It may happen that more than N senders call deposit, so two senders may point to the same slot after all. But the write, read path in the type definition of one slot buffer assures that only one of these senders writes into that slot before it is read out.

6.3. If we add the possibility of concurrent execution to the path expressions, simple path expressions are already a powerful design tool.

Concurrent execution is indicated by a curly bracket pair { }.

Example: path {read}, write end
This path expression allows many concurrent reads or one write.

This path slightly favors reading, because another process can start reading as long as others are reading whereas writing cannot commence until all reading has stopped. Reading and writing can be given an equal chance by the combination of these two paths:

path readreq, WRITE end; path {read}, write end

where READ = b readreq; read e and WRITE = write.
Before actual reading can start, permission must be obtained by passing readreq. (One can program the Readers/Writers problems and variations with a combination of three paths).

6.3. Finally some remarks about implementation.

The path expressions in their simple form can easily be translated into bracket representations. The translation rules are

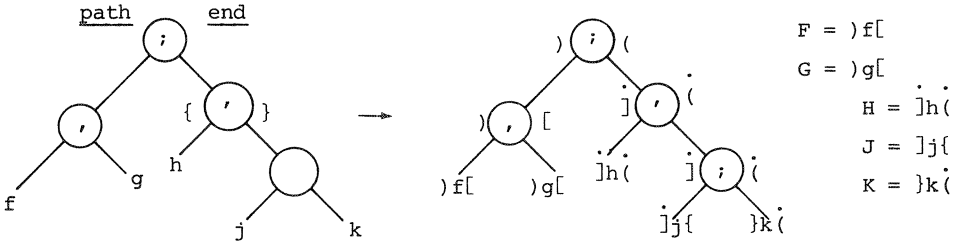
1 path - end → invent a new bracket pair] [

2 f;g → replace semicolon by a close and open bracket of a new chosen bracket type

3 f,g → distribute the bracket pair surrounding f,g over f and g separately

4 {f} → dot the brackets around f indicating that only the first and the last execution must use the brackets.

Example: path (f,g); {h,(j;k)} end



The path assures that the number of executions of K is less than or equal to the number of executions of].

FORMAL PROPERTIES OF DATA BASES by E.J. NEUHOLD

MATHEMATICAL CENTRE TRACTS 63, 1974, 121-177

FORMAL PROPERTIES OF DATA BASES

E.J. NEUHOLD

University of Stuttgart, Stuttgart, (D)

1. INTRODUCTION

When investigating the formal properties of data bases one soon realizes that the formal theories in this area are not yet very advanced. No conclusive theory exists and as a consequence many different techniques have been developed. They are basically very similar, but it is still hard to relate them to each other and to compare their descriptive power.

Instead of investigating all formal work on data bases we shall concentrate on a few essential activities. Especially, we shall disregard the most theoretical approaches, as they are still only applicable to very simple date base models (e.g. C. PAIR [13]). The implementation and hardware oriented investigations (e.g. R. BAYER [14], R.W. TAYLOR [15]) also leave the frame of our discussions. They are usually of high complexity and contain many machine dependent parameters and considerations.

The work underlying these lecture notes is mainly derived from the papers by E.F. CODD [1-6], J.R. ABRIAL [7], B. SUNDGREN [8-9] and E.J. NEUHOLD [10-12]. The general aim of the paper is to stimulate readers interest in the application of precise formal notions to the data base area, where a large range of usually only vaguely defined terms and concepts contributes to the confusion and misunderstandings among the representatives of the field. No results which essentially extend the range of the referenced literature are given though a number of remarks and positions taken may be new.

2. THE INFOLOGICAL APPROACH TO DATA BASES

2.1. The Infological Model of the Human Mind

The organization of a data base and of the information system where the data base is just one part depends very heavily on the characteristics

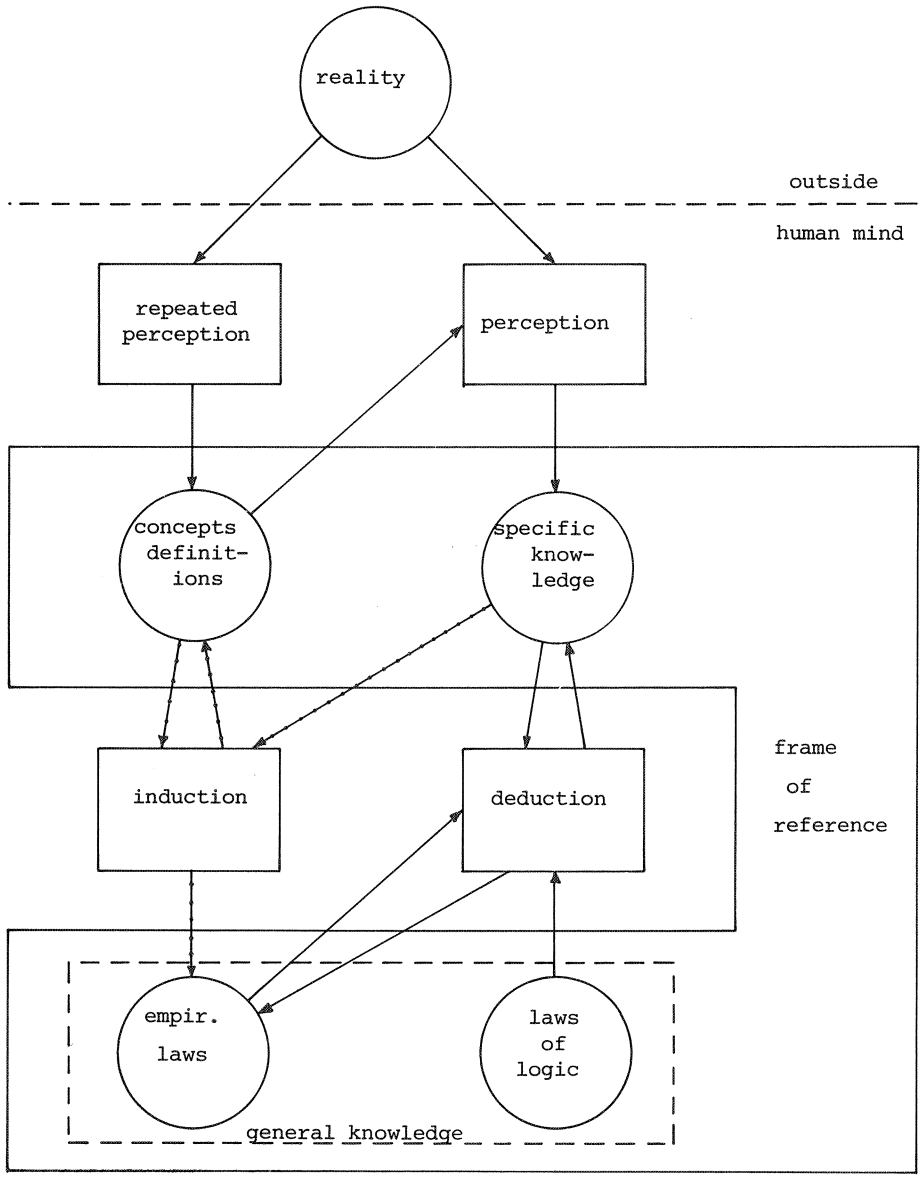


Figure 1: Infological model of the human mind.

of the ultimate users of the system, usually human beings.

To develop a proper framework for the design of information systems we first have to study the working of the human mind as the medium where the data extracted from a data base will be processed to contribute to the problem solution attempted by the person. Figure 1 contains an *infological model* of the human mind (see also [8]) where the following important aspects may be observed:

- The existence of an *outside reality* is assumed. In this reality it is always possible, at least in principle, to determine whether a specific statement about the reality is true or false. This assumption is necessary as soon as we want to conjure that statements, which users cannot agree upon, have no place in data bases. Such an agreement is necessary for the exchange of information between users, one of the essential properties of the information systems we want to investigate.
- To gain any knowledge at all about the reality, *concepts* must be formed (or given to the human being). This formation process is achieved by the repeated perception of pieces of the reality and it continually goes on during a persons life. New concepts are formed, obsolete concepts are deleted and old concepts may be forgotten.
- The concepts together with perception of the reality allow the formulation of *specific knowledge*.
- Concepts and definitions together with specific knowledge allow the induction of new concepts and definitions and of *empirical laws*.
- The specific knowledge, the empirical laws and the apriori given *laws of logic* allow the deduction of other specific knowledge and of new empirical laws.
- Inductions and deductions take place with respect to the *frame of reference*. The frame of reference $R_P(t)$ of a person P at time t encompasses all the concepts, definitions, specific knowledge and general knowledge perceived, deduced or deducible, a person has at time t.
- A person may be more or less conscious of different parts of his frame of reference. For such a part k of his frame of reference a *consciousness function* c(k,P,t) can be given for the person P at time t which may take on values between zero and one. The value of c(k,P,t) is zero if and only if $k \notin R_P(t)$. The value is one if the person P is immediately aware of k at time t.
- Changes in the consciousness value may arise through the ongoing perception or through the usage of k in induction or deduction processes.

- Knowledge may be *explicit* or *implicit*. Explicit knowledge exists already as specific knowledge, laws, concepts or definitions whereas implicit knowledge is deducible knowledge which has not yet been deduced or perceived.

2.2. The Data Concept

The addition of new knowledge may increase and decrease the consciousness of already existing knowledge in $R_p(t)$ or it may even distort the frame of reference and produce inconsistent knowledge, such endangering the inductive and deductive processes of the mind. For these reasons *artificial extensions* of the human mind have been used for a very long time. The artificial extensions allow to *store knowledge* and to *communicate knowledge* to other persons through the use of *data*.

DEFINITION. If a person arranges *intentionally* one piece of reality to represent another, then the former arrangement is called *data*. The arranged piece of reality is the *medium* used for *storing* data.

Examples of such arrangements are: digital and analog representations, spoken and written languages etc. Observe that the *intention* of making such an arrangement is important. Coincidents where one piece of reality by accident represents some other piece do not count as stored data. However it is allowed, that a person only makes the arrangement for storing data, whereas the actual storing is done automatically, for example with the help of automatic sensors.

2.3. The Information Concept

Information is defined as synonymous with *new knowledge*. With this definition a large number of observations can be made using the already investigated properties of knowledge.

- Information is knowledge and can only exist in the mind of a human being where it is part of his frame of reference, or more precisely $I \notin R_p(t^-)$ $I \in R_p(t)$ and $I \in R_p(t^+)$.
- Information therefore is always related to a *reference person*. If information is to be stored a reference person (at least some "average" user) is always assumed for whom the data are intended. However it is usually the case that no such "average" user exists and a lot of complicated data base problems result through erroneous extraction of information from the stored data.

- Information is always related to a set of old knowledge, the *reference knowledge*, i.e. $R_p(t^-)$. For example when a message is sent to some person a typical frame of reference is assumed for the receiver of the message.
- For every information an external source must exist. Otherwise it would not be new knowledge but would already exist in some explicit or implicit form in the reference knowledge.

Sources of Information

Only two sources of information are possible:

a) perception of a piece of reality
b) perception of data representing a piece of reality. Such data are called a *data message* or a *data record*.

The Forming of Information

The creation of new knowledge from a piece of reality or a data message is illustrated in Figure 2. To allow the interpretation of a data message a *compatible* frame of reference $R_p(t)$ must exist. Otherwise the proper concepts, definitions and interpretation rules would not be available and the message would remain meaningless or would be interpreted the wrong way. The conceptual message contains the *semantic contents* of the interpreted message. Using this meaning of the message together with the old knowledge (reference knowledge) the updated knowledge can be derived. Notice however that a message may be meaningful but will still not convey any information if the meaning it represents is already part of the explicit or implicit knowledge of the person.

Even if a message does not carry any information, the action of sending the message may convey information. For example A knows but is also told by B that "the flight from New York to Amsterdam leaves at 9 p.m.". A receives among others the informations:

- B knows that "the flight from New York to Amsterdam leaves at 9 p.m.".
- B does not know that A knows "the flight from New York to Amsterdam leaves at 9 p.m.".

The processes for forming of information do not ensure that the receiver P of a message interprets it as it was intended by the sender. Even if the frames of reference for two persons are compatible it is not sure that they will get the same information. If their knowledge differs at all then one of them may already know part of the meaning contained in the message.

Among further properties of information are:

- A person may be more or less conscious of information as it need not be placed with a high consciousness value into his frame of reference.
- Information may be explicit or implicit. If not all possible deductions are performed at the derivation time implicit information results which at a later point in time may become explicit knowledge through the deduction processes.
- A data message may also have value as a *reminder*, even if no information is conveyed. It may, for example, increase the level of consciousness of some knowledge or it may make implicit knowledge explicit.

2.4. The Information System

Using the infological theory as presented so far we can now relate the developed concepts to the information systems and their data bases. A number of valuable properties of these systems can be derived immediately:

- A data base represents the medium for storing data and since all data are representations of pieces of reality it provides a *model* of the reality. Human beings have used models for a long time. Usually manipulations on the model are much easier and faster than on the reality itself. Different operations and their effects can be composed before they are actually carried out. The model also provides for an extension of the human abilities, where access to data and information is much faster than through observation of the reality.
- The data base provides a source of information for the user and therefore leaves the range of being a simple extension of his mind. Other users may leave messages in the data base which eventually will convey new knowledge to the user of the data base.
- The information system as an extension of the human mind can help as a reminder of knowledge with a low level of consciousness. But where does it become important? For example when calculating

  5 + 12 I definitely will not use the information system.

  18.562 + 4.2436 I may use the information system if it is easy to use, next to my desk and ready.

  sin(4.253) I will use the information system.

  We can deduce that it is important for an information system and its data base to be readily available and in its external representation of messages close to a form which can easily be interpreted by the user.
- The user support system should help the user in his different actions

when solving some problem. According to Figure 1 this involves perception, conceptualizations, inductions and deductions. Perception is supported via automatic data gathering equipment. Deduction support ranges from automatic theorem proving to simple arithmetic calculations. For conceptualization and induction processes current and probably planned systems provide very little direct help.

- Data base builders and users must have compatible frames of references to allow for the proper interpretation of messages. For example, in the statement

  salary ← 15000,

  it is necessary to know whether "salary" is specified by day, month or year, whether the currency is Dollars, Gulden or Lira and also whether the number representation is decimal octal etc.

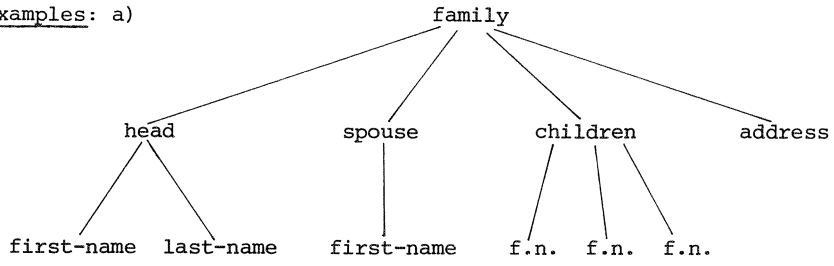
  However it is not necessary that all interpretation rules are a priori knowledge of the users. They can be provided as messages to such users as long as some basic compatible frame of reference exists, to interpret the messages carrying new, additional interpretation rules.

Components of the Information System

To construct information systems which have the properties listed previously a few major components can be identified:

- The data base must provide for the storing of the allowed *basic data values*. These values include numbers, names, descriptions, e.g. 15000, J. SMITH, "*Salary* is in $ and per year", "*Employer* may be *person* or *corporation*". The italic names provide the identification of the different *data categories* existing in the infological base. As a general guideline, data are usually grouped into a category, if their interpretation by a human being leads to a highly overlapped semantic content in this resulting conceptual message.
- *Relations* and *dependencies* between the data in the data base must be expressed by organizational concepts (e.g. data hierarchies), algorithms, laws, and interpretation rules.

Examples: a)



b) $\sin(x) \underset{Df}{=} program$

c) $\neg\, true = false$

d) yearly-salary = 13 * monthly-salary

e) salary: fixed, decimal, 5 digits.

- To allow for the adjustment of the time dependent properties of data bases it is necessary to provide manipulation functions, as, for example for the reorganization of data dependencies or for the deletion/addition of data relations, algorithms, data classes etc.
- Many different users with a wide variety of knowledge will use an information system. To provide each one with a support facility closely related to his frame of reference and subject area it becomes necessary to organize the data base system in such a way, that the data representations, dependencies and relations can easily be adjusted. This leads to a facility for defining *subject oriented data base* submodels.
- To communicate with the information system a user must be able to access, update, insert data, to use algorithms and to receive support for his deduction problems. Here again facilities closely related to the frame of reference of the information system user are required.

Organization of an Information System

Based on the infological approach to data base systems we can now discuss the design and the operation of such a system. Figure 3a describes the various steps involved in the design of the data base system. Figure 3b illustrates the transformations which occur when a user action is to be performed.

Starting with the piece of reality which is relevant for a user or class of users a *subject matter model* of the reality is constructed, which closely corresponds to their frame of reference and allows simple interpretations and deduction processes. With the help of the infological theory the different subject matter models are combined into a common infological

model. The datalogical model is derived from the infological model through the application of data base design theory and the specific design goals. The final mapping of the datalogical model to the physical devices provides us with the physical data base which will contain all the information stored in the system.

If a user of the information system requests some action to be performed, he will state this request in a form closely related to his frame of reference, and therefore to the piece of reality which he has knowledge about. In the context of the information system this means that the actions will be specified with respect to his subject matter model. To perform the actions on the stored data a translation process of the user request must take place. With the help of the infological model, the datalogical and the physical data base description the user actions are transformed into operations on the physical data base. Selected or produced results then are transferred back to a form which the user can interpret.

The approach to information systems described in Figure 3 is still very far from the current abilities of data base systems. In practically all systems the subject matter models, the infological model and the datalogical model are combined into a single model. As a consequence the designers and users have to be aware of all the infological, datalogical and physical aspects of their information system at the same time.

The designer must know about the subject matter models of all users. He has to specify the representation of data, their grouping and their dependencies. He must know about design decisions to be made to realize space and/or time efficiency. The user, on the other side, has to perform in his mind the necessary translation processes, to transform the required action from the form suggested by his frame of reference to the physical operations of the data base.

In the past the data base systems have all been very simple in the infological sense. In their design it was assumed, at least implicitely, that

- only a very small slice of the reality had to be covered,
- a large (nearly total) overlap of the frames of references for all users did exist,
- a relatively small volume of data had to be stored or only simple operations (like "read-next record") were allowed, thus reducing the problems in achieving efficiency.

For large shared data base systems this simplistic approach will not

be sufficient. It will be necessary to separate the infological and datalogical aspects of the system. Arising problems can then be dealt with separately, and no person has to have knowledge about all the different requirements resulting from the infological and datalogical properties of information systems.

The next chapters will discuss data organizations and data manipulation facilities which can be viewed as being oriented to the infological model of the data base. In the last part of the paper we will introduce a representation and description technique for which it is hoped that it will allow the uniform description of all the different levels of an information system.

3. THE RELATIONAL VIEW OF DATA

3.1. General Definitions

In the previous chapter a number of levels for different data models have been introduced. We shall now concentrate our investigations toward the infological data base model as an intermediory between the individual subject matter models and the already computersystem and device dependent datalogical model.

The principal components of the infological model which are to be investigated in this chapter are the different basic data categories and the principles involved in describing the dependencies and relations between the data values. From the many different approaches which exist for the description of data base models only few are sufficiently machine and device independent to be adoptable as infological data models. The relational view of data bases as introduced by E.F. CODD [1-6] has all the required characteristics and in addition has been and still is extensively investigated by many people doing data base research and development. Unfortunately the investigations have been carried out without much explicit considerations for the infological aspects of the model. Of course, many infological concepts have been used implicitely, but I believe a considerable amount of misunderstanding between the people involved in data base research could have been avoided if infological influences would have been presented and illustrated explicitely.

In this chapter we shall use the term *relation* in its accepted mathe-

matical sense, whereas the term *relationship* will be used informally to express any kind of interrelations between data. In addition we shall consider the term *data base* to be synonymous with infological model of a data base.

Given sets $D_1,D_2,\ldots,D_n$ (not necessarily distinct), R is a *relation* on these n sets, if it is a set of elements of the form $(d_1,d_2,\ldots,d_n)$, where $d_j \in D_j$ for $1 \leq j \leq n$. In other words R is a subset of the cartesian product $D_1 \times D_2 \times \ldots \times D_n$. The relation R is said to be of degree n. D_j is the j-th *domain* of R. The elements of a relation of degree n are called *n-tuples*.

A *data base* B is a finite collection of time varying relations defined on a finite collection of domains $D_1,D_2,\ldots,D_p$. As time progresses each relation may be subject to insertion of new elements, deletion of existing ones and alteration of components of existing elements each time resulting in a new *instance* (i.e. set of n-tuples) of the relation.

Example: The data base B is a collection of two relations, describing employees and children,
employee (man #, name, birth year, children)
children (childname, birth year)
defined on the five domains man #, name, birth year, children, childname. For each quadruple contained in the employee relation there exists an instance of the children relation describing the children of the employee. □

From the above definitions a number of observations can be made and additional concepts on properties of the relational model can be developed:

- The domains $D_1,D_2,\ldots,D_p$ may be *simple* or *nonsimple*. A simple domain is defined as a set of basic data values, a nonsimple domain has instances of relations as its elements. In the above example all domains except "children" are simple. The nonsimple domain "children" has instances of the relation children, i.e. sets, as elements.
- A domain D_j contained in a relation R describes the range of values the j-th component of the n-tuples in the relation may have.
- A *compound domain* is the cartesian product of k (k>1) simple domains.
- The simple domains of the relational view of data bases closely correspond to the categories of basic data values found in the infological model.
- The only infological relationships and dependencies between data which can be expressed using the relational view of the data base must be expressible within the framework of relations with simple and nonsimple domains.

- The set of tuples representing a relation R at time t is called an *instance* of R.
- Not all the domains of the n-ary relation R need be distinct. If the same domain appears more than once in a relation R, the different occurrences could be identified through ordering of the domains. Instead of retaining such an ordering we shall introduce distinct names, called *attribute names*, which will uniquely identify the different occurences of domains in the relation R. The domain occurrences in a domain are now order independent and will be called *attributes* of the relation R.

Example: For the description of parts and their contained subparts a "component" relation will be defined on two occurrences of the domain "part". With the help of attribute names we specify the relation as follows: component (sub-part, super-part, quantity) where both attributes "sub-part" and "super-part" correspond to the domain part existing in the relational model. □

- The two values v_1 and v_2, contained in the attributes A_1 and A_2 or R respectively, are *associated* with each other if in the current instance of R there exists at least one n-tuple which contains v_1 and v_2 as the respective values for the attributes A_1 and A_2.
- A relation R is called *normalized* or in *first normal form* if it has the property that none if its attributes are nonsimple. An *unnormalized* relation is one which is not in first normal form.

To restrict a data base to normalized relations has a large number of advantages. The most important probably is, that the infological complexity of unnormalized relations is very high. Data interpretation in the relationally expressed hierarchies is complicated and many problems of implicit data dependencies arise. As we shall see later in this paper even the first normal form of relations is of high infological complexity and additional restrictions have been developed to reach simple but still sufficient data base models.

Normalized relations can be stored as two dimensional matrices with homogeneous values in the columns. This representation does not require pointers or hash addressing schemes and seems to be the form best fitted for bulk data transfer between systems of different structure.

Most data base models containing unnormalized relations may be normalized automatically. Before we can discuss the normalization procedure a few

additional concepts have to be introduced. It should be noted however, that because of the very high infological complexity of unnormalized relations, the automatic normalization process has to make strict assumptions about the reference knowledge to be used for the interpretation of the restructured relations.

- We introduce the notation R.A to denote the attribute A of R, and r.A to select the value of the attribute A in the tuple r ($r \in R$). This notation can be expanded to a list of attributes $A = (A_1, A_2, \ldots, A_j)$ of R. The expression R.A then denotes a *collection* of attributes in R. The notation r.A is expanded to

  $$r.A = (r.A_1, r.A_2, \ldots, r.A_j)$$

  with the additional definition

  $$r.A = r$$

  in case the list A is empty. We shall use the notation $\bar{A}$ to identify the attributes of R which are not contained in A. Similarily $R.\bar{A}$ and $r.\bar{A}$ are defined.

Example: The current instance of the relation

employee(man #, name, birth year, children)

may contain the quadrupel

r = (1345, J. SMITH, 1936, {(JILL, 1964)})

Given the list A = (name, birth year),

$r.A$ = (J. SMITH, 1936)
$r.\bar{A}$ = (1345,{(JILL, 1964)}) □

- For the manipulation of relations we introduce a *concatenation* of two tuples

$$r_1 = (e_1, e_2, \ldots, e_n) \in R_1$$
$$r_2 = (f_1, f_2, \ldots, f_m) \in R_2$$

Its result is defined by the (n+m)-tuple

$$r_1 r_2 = (e_1, e_2, \ldots, e_n, f_1, f_2, \ldots, f_m)$$

- Two simple domains are *union-compatible* if the infological interpretation of their data values allows us to combine the values into a single data category. Observe, that the property of union-compatibility is heavily dependent on the interpretation of the respective data values and therefore on the reference knowldege of the data base user.

  Two compound domains D and E are union-compatbile if they are of the same degree (say n) and for every j ($1 \leq j \leq n$) the j-th simple domain in D is union-compatible with the j-th simple domain in E. Two relations R,S are union-compatible if the compound domains of which R and S are subsets are union-compatible.

3.2. Functional Dependence

When setting up a relational data base the designer must decide on the degree and the properties of the relations to be incorporated in the model. The relational data base is the datalogical model in the general information system and must combine the different aspects of the subject matter models which, as abstractions of the reality, closely correspond to the reference knowledge of the various users.

One such important aspect is the functional dependence between attributes in a relation R. It arises whenever in the n-tuples found in R some compound values are not independent from each other.

We define more formally:
The attribute A_2 of relation R is *functionally dependent* on attribute A_1 of R if, *at every instant of time*, each value of A_1 has *no more than one* value in A_2 associated with it under R.

The infological aspects of this definition can be seen immediately in the request that the condition must hold "at every instant of time". This property cannot be checked without knowledge of the modelled reality and

the abstraction and specification processes involved. We write $R.A_1 \rightarrow R.A_2$ if A_2 is functionally dependent on A_1 in R and $R.A_1 \not\rightarrow R.A_2$ if A_2 is not functionally dependent on A_1 in R. If both $R.A_1 \rightarrow R.A_2$ and $R.A_2 \rightarrow R.A_1$ hold, we write $R.A_1 \leftrightarrow R.A_2$ and $R.A_1$ and $R.A_2$ are at all times in a one-to-one correspondence. The functional dependency satisfies the transitivity condition; if $R.A_1 \rightarrow R.A_2$ and $R.A_2 \rightarrow R.A_3$ then $R_1A_1 \rightarrow R.A_3$.

The definition of functional dependence can be extended to collection of attributes. If D and E are distinct collections of attributes of R, then E is functionally dependent on D if, at every instance of time, each D value has no more than one E value associated. In case the attribute collection E is a subset of D we speak of *trivial functional dependence* since the requirements for functional dependence are trivially satisfied.

Example: To illustrate functional dependence we use a relation

S(Empl #, Dept #, Div #)

where either the reference knowledge or the abstraction from reality determines that
Empl # is the employee serial number
Dept # is the number of the department to which the employee belongs
Div # is the serial number of the division to which the employee belongs.
In addition the infological framework may supply the time independent properties
- an employee never belongs to more than one department
- a department never belongs to more than one division
- an employee belongs to the division to which his department belongs.
Actually these properties are important during the whole lifetime of the data base. Consequently they should themselves be kept in the database system. We shall discuss in the next chapters how this can be achieved at least for a simple subconcept of our relational model.

Using the infological properties and relationships we immediately derive, e.g.

R. Empl # → R. Dept #
R. Dept # → R. Div #
R(Empl #, Dept #) → R. Div #

If we know in addition that
- many employees work in one department
- many departments are associated with a given Division
then we can define

$$\text{R. Dept}^{\#} \not\rightarrow \text{R. Empl}^{\#}$$
$$\text{R. Div}^{\#} \not\rightarrow \text{R. Dept}^{\#}$$

An example of a trivial dependence is given through

$$\text{R(Empl}^{\#}, \text{Dept}^{\#}) \rightarrow \text{R. Empl}^{\#} \quad \square$$

3.3. Candidate Keys and Primary Keys

We define a *candidate key* K of a relation $R(A_1,A_2,\ldots,A_n)$ as a collection of attributes (possibly one) with the following properties:

P1: *Unique identification*
In each tuple r of the relation R the value of the attributes contained in K uniquely identifies the tuple, i.e.

$$R.K \rightarrow R.(A_1,A_2,\ldots,A_n)$$

P2: *Non-redundancy*
No attribute in K can be deleted without destroying the property P1.

With the definitions given earlier we are able to deduce two additional properties of candidate keys

P3: Each attribute of R is functionally dependent on each cnadidate key of R. (An immediate consequence of trivial dependency, transitivity of functional dependence, and property P1.)

P4: The collection of attributes of R found in a candidate key K is a *maximal functionally independent* set. That is,
a) every proper subset S_1 of K is functionally independent of every other proper subset S_2 of K when

$$S_1 \not\subseteq S_2 \quad \text{and} \quad S_2 \not\subseteq S_1,$$

b) no attributes of R can be added to K without destroying the functional independence of K.

(Part a) is a consequence of the transitivity of functional dependence, of trivial dependence and of property P2. Part b) is an immediate consequence of property P3.)

One of the candidate keys is selected as the *primary key* of the relation R. Its selection usually depends on infological reasons, e.g. ease of its interpretation with the help of the reference knowledge of the human users. The primary key serves for the unique identification of the individual tuples contained in the relation. It may appear for this identification purpose as a *foreign key* in some other relation where an infological dependency to R exists. In order to be always able to use the primary key for tuple-identification purposes it is not possible that undefined values for any of its component attributes are allowed. In all other respects the primary key can be handled like any other candidate key.

Example: In the two relations

warehouse (*ident #*, *part #*, quantity-on-hand)
part (*part #*, name, price)

the primary keys have been written in italics. If it is known that the part names will be unique at all times, the attribute "name" represents a candidate key of the relation "part". In the "warehouse" relation the attribute part # is a foreign key and in this example also part of the primary key. □

3.4. The First Normal Form of Relations

We have observed that the unnormalized relations add considerably to the task of message interpretation and require in addition complex mechanisms for their realization in the physical data base.

Using the definitions of functional dependence and primary keys a normalization process for unnormalized relations can be given:

Step 1: Starting with a relation R which does not appear in any nonsimple attributes (a *top* relation)

- the collection of attributes representing its primary key is selected.
- the immediately subordinate relations are expanded with this attribute collection,

- a new primary key in the expanded subordinate relation is formed, which consists of the old primary key and the inserted collection of attributes,
- all value adjustment in the subordinate tuples are made by copying the primary key value from the containing tuple of R,
- all nonsimple attributes are deleted from the top relation.

Step 2: Repeat the process for all nonsimple, expanded attributes.

Example 1: The already discussed data base B with the relations

employee (*man #*, name, birth year, children)
children (*childname*, birth year)

and the primary keys identified by italic letters will be transformed during the normalization process to the normalized relations

employee (*man #*, name, birth year)
children (*childname, man #*, birth year) □

The normalization process can only be applied to relational data base models which satisfy the following two conditions:

1. The graph of interrelationships of nonsimple attributes is a collection of trees.
2. No primary key has a component attribute which is nonsimple.

In the remainder of our discussions on the relational view of data bases the term relation will always mean relations in first normal form except when explicitely noted differently.

3.5. Operations on Relations

In order to work with a relational model of a data base operations have to be defined which allow the manipulation of operations, especially the restructuring of relations and the formation of new relations out of existing ones. Further operations on relational data bases will be discussed in chapter 4.

The operations to be discussed in this section belong to two principal classes; a) traditional set operations and b) operations meaningful because of the infological interpretation given to the relational data base model.

Traditional Set Operations

The set operations *union* (∪), *intersection* (∩) and *difference* (-) can be used with their accepted mathematical meaning with the only restriction that they can only be applied to union-compatible relations.

The *cartesian product* R × S can be applied to any (even unnormalized) relations. The result of the operation is a relation of degree two with the nonsimple attributes R and S. As a consequence, the result of the cartesian product is always an unnormalized relation even if the relations R and S are normalized. It has the additional property that each instance of the relations R and S appearing in the tuples of R × S contains only one element.

Relational Cartesian Product

Much more important than the normal set theoretic cartesian product is the relational cartesian product. Given two relations R,S of degree n,m respectively we define the *relational cartesian product* by

$$R \otimes S = \{(rs) : r \in R \wedge s \in S\}$$

with the resulting degree n + m.

Projection

To allow access to parts of relations the projection operation is defined. Given the relation R and r one of its tuples we define the projection of R on a list of attributes A in R as follows:

$$R[A] = \{r.A : r \in R\}.$$

The resulting relation has the degree card (A).

Examples of projections:

Given the relation R containing three tuples

$R(A_1$	A_2	$A_3)$
a	3	g
b	1	g
c	2	f

we apply different projection operations and get as results

$R[(A_1)]$	$R[(A_1,A_3)]$	$R[(A_3)]$
a	a g	g
b	b g	f
c	c f	

□

Join

To combine two relations without using the full expansion capabilities of the relational cartesian product the join operation has been introduced. We assume two relations R and S and two *comparable* lists of attributes A and B of the relations R and S respectively. The *comparability* of two attribute lists with respect to a comparison operator p is defined such that

- the length of both lists is equal, say k,
- each pair of attributes $A_i \in A$ and $B_i \in B$, $1 \leq i \leq k$, is comparable with respect to the operator p (one of the operators $=, \neq, <, \leq, \geq, >$), that is for every element of R.A and every element of S.B the operation yields either *true* or *false* but not undefined.

The comparison r.Aps.B is *true* iff for all

$A_j \in A$, $B_j \in B$ the comparison $r.A_j pd.B_j$ yields *true*, i.e.

$$\forall A \; \forall_{j=1}^{k} \; (r.A_j ps.B_j)$$

The *p-join* of R and S on the attribute lists A and B is now defined by

$$R(A\,p\,B)S = \{(rs) : r \in R \wedge s \in S \wedge (r.A\,ps\,.B)\}.$$

Examples of joins:

Given the relations

R(A, B, C)			S(D,E)	
a	1	1	2	u
a	2	1	3	v
b	1	2		
c	2	5		

join operations yield

R(B=D)S(A,B,C,D,E)					R(C>D)S(A,B,C,D,E)				
a	2	1	2	u	c	2	5	2	u
c	2	5	2	u	c	2	5	3	v

Natural Join

When joining two relations with respect to the equality operation the resulting tuples always contain equal values in the respective attribute positions of the lists A and B. This redundancy can be eliminated (if desirable from an infological point of view) with the help of the natural-join operation.

Given two relations R and S and two lists A and B of attributes in R and S respectively, where A and B are comparable for equality, we define the *natural join* of R and S with respect to lists A and B by

$$R(A*B)S = \{(rs.\bar{B}: r \in R \wedge s \in S \wedge (r.A = s.B)\}.$$

It is easy to see that the natural join can also be defined in terms of projections and an equality-join

$$R(A*B)S = (R(A=B)S)[\bar{B}]$$

Example for natural join:

Using the relations given in the examples for the join operation we get

R(B*D)S(A,B,C,E)

	a	2	1	u
	c	2	5	u

□

To increase the convenience in expressing relational operations two additional operations can be introduced which can both be defined in terms of already known operations.

Division

Given two relations R and S and two attribute lists A and B in R and S respectively, where the compound domain defined by A and by B are union-compatible, the *division* operation is defined by

$$R[A \div B]S = R[\bar{A}] - ((R[\bar{A}] \otimes S[B]) - R)[\bar{A}]$$

Restriction

Given a single relation R and two lists of attributes A and B in R

such that they are comparable with respect to the operator p a restriction can be expressed by

$$R[A\,p\,B] = \{r\colon r \in R \wedge (r.A\,p\,r.B)\}.$$

The restriction operation, as shown by E.F. CODD in [3] may also be defined in terms of joins and projections.

The operations introduced in this section can be considered to form a *relational algebra* for the formulation of manipulation requests on the relations of a relational data base. Humans using an information system will express data manipulation requests in terms of their specific subject matter models. These requests, as discussed earlier, will have to be translated into data manipulation requests for the infological model and consequently could be expressed in the relational algebra.

Example of a user request formulated in relational algebra:

Using the normalized relations "employee" and "children" of example 1 we represent the request

"Find the names of employees, each of whom has children with the birth year 1965."

in relational algebra by

(employee (man # = man #)
(children(birth year = birth year){(1965)}[(man #)]))[(name)]

where it is assumed that the single attribute of the relation

{(1965)} has the name "birth year". □

When using the relational algebra for the formulation of data manipulation requests a number of disadvantages are apparent:

- Every operation has to be formulated in the framework of relations. That is, instead of using basic data values directly, e.g. 1965, they have to be expressed in relational form.
- To include additional functions e.g. MAX, SQRT, etc., in the relational algebra requires, that they are expressed as mappings on relations, a property which so far has not been developed for these commonly used functions.

- Bearing in mind, that a user wants to formulate his data manipulation requests in ways closely corresponding to his subject matter model, the relational algebra seems to force unnecessary problems into the required translation processes (see also the next section).

3.6. The Relational Calculus

We have seen in the preceeding section that with the relational algebra a communication between user and information system (at least the infological model) can be established. However the majority of the users will be oriented to their own subject matter models and to languages strongly influenced by these models and by natural languages.

In such an environment, like in everday life, it will be much more likely that the manipulation requests are expressed in terms of properties of objects, e.g. the birth year 1965 of children, and of combinations of such properties. Consequently it seems, that we are able to elminate the disadvantages mentioned in the preceeding section for the relational algebra, by developing a *calculus* oriented language for the formulation of data manipulation requests on the infological model (see E.F. CODD [3]). Such a language will have the special advantage of being closely related to the subject matter model oriented methods used by the humans in formulating data manipulation requests.

For formulating expression in the *relational calculus* the following notions will be used:

basic data values	$a_1, a_2, \ldots$
tuple variables	$r_1, r_2, \ldots$
attribute names	$d_1, d_2, \ldots$
predicates	P_1, P_2
comparison symbols	$=, >, <, \leq, \geq, \neq$
logical symbols	$\exists, \forall, \vee, \wedge, \neg$
delimiters	[,], (,)

In addition we assume a one-to-one correspondence between *predicates* $P_1, P_2, \ldots, P_N$ and the relations $R_1, R_2, \ldots, R_N$ of the relational model, such that P_j indicates membership of tuples in R_j.

Using these basic notions the construction rules for *terms* can be formulated:

1. A *range term* has the form $P_j r$ where P_j is a predicate and r a tuple variable. It establishes that the range of r is the relation R_j.
2. We first define a *tuple component* r.d for a tuple variable r and an attribute name d to identify the d component of the tuple.
   Assuming tuple components e and f, a comparison symbol p and a basic data value a then

   epf epa

   are *join terms*.
3. A *term* is either a *range term* or a *join term*.

Well-formed Formulae

The *well-formed formulae* (WFF) of the relational calculus are defined as follows:

1) Any term is a WFF;
2) If e is a WFF, so is ¬e;
3) If e,f are WFF, then (e∨f) and (e∧f) are WFFs;
4) If e is a WFF in which r occurs as a free variable then ∃r(e) and ∀r(e) are WFFs;
5) No other formulae are WFFs.

Range Separability

The use of tuple variables in an infological framework for the selection of data and for the testing of data properties requires that the range of such variables is clearly defined. Otherwise infological confusion and wrongly interpreted data messages would be the inevitable result. We therefore have to restrict the general WFFs to enforce range definitions for all occurring tuple variables.

The following definitions are made for the purpose of such restrictions:

- A *range WFF* is a quantifier free WFF all of whose terms are range terms.
- A *range WFF over r* is a range WFF whose only free variable is r.
- A *proper range WFF over r* must in addition satisfy
  a) ¬ does not occur at all or it immediately follows ∧. This restriction excludes range WFFs of the form ¬Pr specifying as range of r everything (!) except the relation R associated with P. A situation where most likely no infological interpretation for the range of r could be given.
  b) whenever r occurs in two or more range terms, the relations associated

with the predicates in those terms must be union-compatible. Again this restriction is made to avoid range definitions which cannot be interpreted in the infological frame.

- If a well-formed formula contains quantifiers, the tuple variables bound by these quantifiers must also have clearly defined ranges. Assume that e is a WFF with r as a free variable but without a range term in r. Let f be a proper range WFF over r. When r becomes bound, either by $\exists r(e)$ or $\forall r(e)$ we introduce a range specification into the construct by replacing $\exists r$ or $\forall r$ by the *range coupled quantifiers* $\exists f$ and $\forall f$ respectively. The re-resulting WFFs are defined by the equations

$$\exists fr(e) = \exists r(f \wedge e)$$
$$\forall fr(e) = \forall r(\neg f \vee e)$$

Notice however, that for an infological interpretation the two constructs $\exists fr(e)$ and $\forall r(\neg f \vee e)$ are not at all equivalent. For the infological interpretation of e in the first construct the values of r are restricted to the range f. In the second construct the values of r ranges over the whole universe of discourse (i.e. all possible tuples) and it does not seem likely that a meaningful infological interpretation for e can be found for every one of these values.

- Finally, a WFF is *range-separable* if it has the form

$$W_1 \wedge W_2 \wedge \ldots \wedge W_n \wedge V$$

where

a) $n \geq 1$

b) $W_1, W_2, \ldots, W_n$ are proper range WFFs over n distinct tuple variables

c) V is either nonexistent or it is a WFF in which every quantifier is range coupled, every free variable belongs to the set whose ranges are specified by $W_1, W_2, \ldots, W_n$ and V does not contain any range terms.

Examples of range-separable WFFs

$$P_8 r_3 \wedge (r_3.d_2 = a_1)$$
$$P_7 r_2 \wedge \exists P_2 r_1 (r_1.d_3 = r_2.d_1)$$

Examples of WFFs not range-separable

$$P_7 r_2 \wedge \exists r_1 ((r_1.d_2 = r_2.d_5) \vee P_8 r_1)$$
$$\neg P_2 r_1 \wedge (r_1.d_4 = r_2.d_1)$$

Alpha expressions

Range-separable WFFs allow the specification of logical conditions on the data values to be selected from a data base. Using only conventional set definition capabilities we would only be able to construct a (mostly incoherent) set of tuples. Accordingly we introduce a capability to select from the identified tuples components for constructing the desired target relation. A *simple alpha expression* has the form

$$\{(t_1,t_2,\ldots,t_k): w\}$$

where w is a range-separable WFF and $t_1,t_2,\ldots,t_k$ are either tuple variables or tuple components where the set of tuple variables appearing in $t_1,t_2,\ldots,t_k$ is precisely the set of free variables in w. The list $(t_1,t_2,\ldots,t_k)$ is called the *target list* and w the *qualification expression*. The definition of simple alpha expressions allows the multiple use of a tuple variable in the target list, providing for situations where a selection of components of one tuple are required in the target relation.

The result of evaluating a simple alpha expression is a relation which is a projection, determined by the target list, of that subset of $R_1 \otimes R_2 \otimes \ldots \otimes R_n$ which satisfies the qualification expression, and where $R_1,R_2,\ldots,R_n$ are the ranges of the free tuple variables of w.

Using simple alpha expression we can now define general *alpha expressions* by

1) Every simple alpha expression is an alpha expression
2) If $\{t:w_1\}$ and $\{t:w_2\}$ are alpha expressions, so are

$$\{t:(w_1 \vee w_2)\}$$
$$\{t:(w_1 \wedge \neg w_2)\}$$
$$\{t:(w_1 \wedge w_2)\}$$

3) No other expressions are alpha expressions.

Example of a user request formulated in relational calculus:

Using the normalized relations "employee" and "children" of example 1 and the request

"Find the names of employees, each of whom has children with birth year 1965."

we find the corresponding alpha expression

$$\{(r_1.\text{name}):\ \text{is-employee}\ r_1\ \wedge$$
$$\exists\text{children}\ r_2(r_2.\text{birth year} = 1965\ \&\ r_2.\text{man \#} = r_1.\text{man \#})\}$$

where "is-employee" represents the predicate defining membership in the relation "employee".

Comparing this solution with the same example when expressed in relational algebra (see section 3.5) we immediately see that the relational calculus expression much closer reflects the original formulation of the user request. □

3.7. Relational Completeness

The qualification expressions in the relational calculus closely reflect the constructs allowed in first order predicate calculus. The restrictions placed on the qualification expressions are due to infological considerations but do not restrict the expressive power of the relational calculus. The alpha expressions introduced in section 2.6 can therefore be considered a measure for the expressive power of other relational algebras and calculi.

E.F. CODD [3] defines a relational algebra or a relational calculus to be *relationally complete* if, given a finite collection of relations $R_1, R_2, \ldots, R_p$ in first normal form, the expressions of the algebra or calculus permit the definition of any relations definable from $R_1, R_2, \ldots, R_p$ by using alpha expressions.

E.F. CODD [3] has proven formally that *the algebra defined by the operations of section 3.5 is relationally complete.* The proof is given in a constructive manner and exceeds the scope of our current discussions.
But despite the relational completeness of the relational algebra specified in section 3.5, our observations still remain valid that it is preferable to use a relational calculus when expressing data manipulation request in the infological model of an information system.

4. BINARY RELATIONS

4.1. Some Deficiencies of n-ary Relations

In the preceding chapter the infological model of the data base was defined by a finite collection of relations with assorted degree. Investi-

gation pointed out however, that the full generality of the relations was not needed and a first normal form was introduced. The infological interpretation of the relations become much simpler, but as will be illustrated in the following example, a number of unpleasant characteristics still remain.

Example: A relation "supply" in first normal form is defined by

supply ($S^{\#}$, $P^{\#}$, SC)

Its infological interpretation states that the attribute $S^{\#}$ identifies *suppliers* which supply the *parts* defined by $P^{\#}$. The attribute SC specifies the *city* where the supplier is located. In addition, a given part may be supplied by many suppliers and a supplier may supply many parts.

Using the definitions of functional dependence given in section 3.2 we get the following principal properties:

$$\text{supply}.S^{\#} \not\rightarrow \text{supply}.P^{\#}$$
$$\text{supply}.P^{\#} \not\rightarrow \text{supply}.S^{\#}$$
$$\text{supply}.S^{\#} \rightarrow \text{supply}.SC$$

In Figure 4 an instance of the "supply" relation is illustrated. We may use this relation instance to derive some of the still undesirable properties of relations in first normal form.

supply ($S^{\#}$,	$P^{\#}$,	SC)
a	1	NYC
a	2	NYC
a	3	NYC
b	1	AMS
b	3	AMS

Figure 4

If a supplier relocates his place of business all the tuples with his identification must be changed. This is a variable number depending on the number of different parts the supplier supplies. If a supplier temporarily ceases to supply any parts it becomes impossible to keep his address for

future reference in the relation since no undefined values for the attributes in the primary key are allowed. □

The problem we have encountered arises from the situation that the attribute SC is functionally dependent only on a part of the primary key. To repair this and similar situations E.F. CODD [2] introduced a *second* and a *third normal form* by prescribing a number of specific restrictions on the allowable functional dependencies in a relation. However, functional dependency is a property of infological interpretations and heavily dependent on the intended meaning of a relation. Consequently, the rules for restricting the functional dependencies, as required when formulating relations in second and third normal form, are quite complicate, not easy to apply and to illustrate and still they are in the end not satisfying.

Example: The 3-ary relation

workplace($E^{\#}$, $D^{\#}$, desk $^{\#}$)

has the following infological interpretation:
The employees $E^{\#}$ may work in many departments identified by the attribute $D^{\#}$. A department may have many employees. In each department he works in, an employee may have at most one desk (desk $^{\#}$). In addition a desk has its place in precisely one department.

This infological interpretation leads to the functional dependencies

workplace.($E^{\#}$, $D^{\#}$) → workplace.desk $^{\#}$
workplace.desk $^{\#}$ → workplace.$D^{\#}$

According to the definitions given in [2] the "workplace" relation is in third normal form, but the problem mentioned in the previous example still exists. If a desk, temporarily, is not used by any employee its location in a department cannot be shared in the workplace relation. □

For these reasons we will not investigate any further n-ary relations and their different normal forms. In practical applications they may be of great value, but for our remaining conceptual investigations it is sufficent to consider *binary relations* only.

After giving the principal definitions we shall concentrate much more than with n-ary relations on the functions required in a relational data

base model to introduce new relations, to define new data categories, to insert or update tuples and to control the integrity of the data base (at least to some degree). We base our discussions on the work of J.R. ABRIAL [7], who introduced extensive concepts for handling relational data bases and showed the power of his definitions by describing the data base model itself using the developed operations and functions.

Restricting the infological data base models to binary relations eliminates the problems which arise out of the complex functional dependencies possible between the attributes of n-ary relations. To illustrate the different notions of a binary relational model we shall use the infological contents of the ternary relation "workplace" as it was explained in the preceeding example.

4.2. The Generation of new Data Categories

A new category of data values is created by definitions of the form

identifier = cat

defining a new category with the name expressed by the identifier, e.g.

$E^{\#}$ = cat

To produce a new data value (object) for one of the data categories, e.g. a new unique name for an employee, we specify

generate $E^{\#}$

If a name is to be given to the created object we formulate

JOHN = generate $E^{\#}$

The name will be permanently attached to the created data value, and the system ensures that no other objects can get the same name. An assignment of a new object to a variable may be specified by

x ← generate $E^{\#}$

Objects which are not needed any more in the data base system may be eliminated by

kill JOHN or kill x

4.3. Binary Relations

Binary relations in the binary relational model (BRM) are specified by identifying two categories and two access functions using the formal

$$r_1 = \underline{rel}(\text{domain 1, domain 2, accfct 1} = \underline{afn}(\text{min,max}), \text{accfct 2} = \underline{afn}(\text{min,max}))$$

where

- domain 1 and domain 2 are two categories on which the tuples of the relation are defined.
- using the notations introduced in the preceding chapter we can express the values produced by the application of the functions accfct 1(x) and accfct 2(y) by

  $\{r.\text{domain 2}: r \in r1 \wedge r.\text{domain 1} = x\}$

  and

  $\{r.\text{domain 1}: r \in r1 \wedge r.\text{domain 2} = y\}$

  respectively. The function accfct 1 is a mapping of domain 1 into the powerset of domain 2 and accfct 2 is a mapping of domain 2 into the powerset of domain 1.
- the access functions accfct 1 and accfet 2 are termed *inverse* to each other. We define an inv operator such that inv(accfct 1) = accfct 2 and inv(accfct 2) = accfct 1. Note that this definition of an inversion operator is not equivalent to the conventional mathematical definition of the inverse operation
- the two terms min and max define the minimum and maximum cardinality of the sets defined by the corresponding access functions.

Example: The four binary relations

r_1 = (WL, E#, personofwloc = afn(1,1), afn(0,∞))

r_2 = (WL, D#, deptofwloc = afn(1,1), afn(0,∞))

r_3 = (WL, desk #, deskofwloc = afn(1,1), afn(0,∞))
r_4 = (desk #, D#, deptofdesk = afn(0,1), afn(0,∞))
describe the same infological contents as the ternary relation"work location" of section 4.1, except that a desk may now exist but not be associated with any employee or department. These are situations which are not expressible at all in the ternary relation "work location".

The four relations may be represented as a graph (see Figure 5) illustrating the involved data categories and the defined access functions.

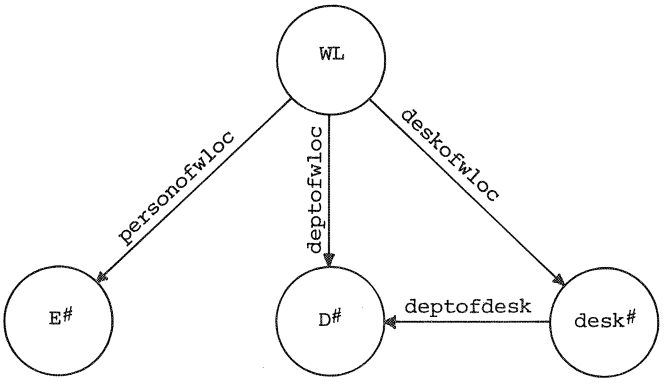


Figure 5

To describe the ternary relation we had to introduce a new category WL (work location) which has as its elements data values indicating the existence of the infological entity

employee(#) in dept(#) on desk(#)

i.e. of the triple in the relation "work location". The implicit functional dependencies of department on desks in "work location" is now expressed explicitly by the relation r4 with the access function deptofdesk.

The above description of the infological meaning attached to the four binary relations leads to a number of consequences:

- The specified cardinalities of the access functions prescribe that, whenever a data value wl ∈ WL exists, the relations r1, r2 and r3 must contain tuples connecting the object wl to an employee, a department and a desk. The precise definition of how this can be assured will be given later.
- A tuple may exist in r4 without the corresponding tuples in the other relations. With this property we have eliminated the undesirable dependency problems occuring in the relation "work location".

4.4. Insertion, Deletion and Tests of Data Values

A number of operations in this sections have a very close resemblance to the operations of the n-ary relational algebra and calculus (see section 3). For consistency reasons they are included here again, sometimes with some slightly modified semantic meaning.

Insertion of a new tuple

We write

deptofdesk(desk):∋d

where desk is either a desk number or a variable with

a desk number as value,
d is either a department number or a variable with a department number as value.

The result of the operation is the inclusion of a new tuple in relation r4. Note that the operation has the implied side effect of

inv(deptofdesk)(d):∋ desk

When executing the :∋ operation the data base system tests whether desk and d belong to the proper categories and whether all cardinality constraints would still be met after insertion of the new tuple.

Deletion of a tuple

With the operation

deptofdesk(desk):∌d

the tuple (desk,d) will be deleted from the relation r4. The access function deptofdesk will reach cardinality zero for the domain value "desk". The system tests during the execution of :∌ whether desk and d belong to the proper categories.

When specifying

deskofwloc(wl):∌ deskofloc(wl)

the corresponding tuple cannot be deleted from the relation r2. The specified cardinality constraints enforce that tuple (w1,unknown) still remains in the relation. The special data value unknown may be included into a tuple, whenever for one of the domains a specific data value is not known in the tuple, but the tuple is to be kept in the relation.

Modification of a tuple

The operation

$$\text{deptofdesk(desk)} \leftarrow d_1$$

replaces the old value of the access function deptofdesk(desk) by d_1. That is, it replaces an already existing tuple, say (desk,d') with the new tuple (desk,d_1).

Testing for Membership

To test for membership of d in category $D^{\#}$ we write

$$d \text{ is } D^{\#}$$

The operation yields true when the value represented by d is a department number.

Membership in a relation may be tested using constructs of the form

$$d \in \text{deptofdesk(desk)}.$$

Comparison Operators

The comparison operators $<$, $>$, $\leq$, $\geq$, $=$, $\neq$ may be used to compare objects of the relational data base.

Quantifiers

The expression

$$\exists z \leftarrow \text{inv(personofwloc)}\ (n1)(\exists y \leftarrow \text{deskofwloc}\ (z)(\text{deptofwloc}(y)=d_1)$$

determines whether an employee n1 has a desk in department d_1. The quantifier expression has the side effect to assign values to the variables z and y. Since there is no control provided for assigning a specific one of the possible values, the usefulness of this side effect seems doubtful. The

formula

$$\forall z \leftarrow \underline{\text{inv}}(\text{personofwloc})\ (n1)\ (\text{deskofwloc}(z) \neq \underline{\text{unknown}})$$

determines whether employee n1 has in each of his departments a desk associated.

Set operations

For the union (∪) and the intersection (∩) the conventional meaning can be retained.

4.5. Semantic Extentions

Besides of operations similar to the ones for n-ary relations we have introduced simple update, insert and delete facilities, but again without too much consideration for infological requirements. Some correctness tests have been included in the operations but they are all standard system functions and therefore cannot show too much flexibility.

We now introduce as a semantic extension capability the possibility to replace one or more of the built-in operators of the binary relational model by user defined actions. For this purpose we establish the operator-function name correspondence shown in Figure 6.

operator	function name
generate	generator
kill	killer
is	recognizer
:∋	updater
:∌	eraser
name of access fct	accessor
∈	tester

Figure 6

To prescribe the new actions for one of these operators a programming language is required. We only introduce a few concepts, others may be found in the paper of J.R. ABRIAL [7].

conditions:

if .... then .... else .... end or if .... then .... end

loops:

for x ← f(y) .... up 1 .... end

do .... end

In the for-loop the up 1 mechanism and the assignment x ← f(y) ensure that the elements defined by the access function f(y) are sequentially processed. In the do-loop no built-in loop control mechanism exists; it must be programmed explicitely.

value return:

return (x)

resume (x)

The return-statement works in the conventional fashion. However the programs to be defined very often will return sets. The resume-statement specifies the return of a value, but it also allows the continuation of the program for additional result elements.

Example: The generate operation of WL objects should ensure that the required cardinality constraints are not violated. Therefore we define

```
generator (WL) ← prog(e,d,desk)
   if ¬(e is E# ∧ d is D# ∧ desk is desk #)then failure end
   x ← std
   personofwloc (x) ← e
   deptofwloc (x) ← d
   deskofwloc (x) ← desk
```

where the standard (built-in) action of generating objects is denoted by the operator std. Whenever we now specify

generate WL

the program defined above is executed instead of the built-in action for generator. Notice however, that for categories c ≠ WL the standard action will still be chosen.

For other operations the same definitional technique can be used. Suppose we want to introduce the relation

r5 = rel(E#, desk #, deskofperson = afn(0,∞), afn(0,∞))

into our relational model. In addition we define

```
accessor(deskofperson) ← prog(e)
    if ¬(e is E#) then failure end
    for x ← inv(personofwloc) (e)
        resume(deskofwloc(x))
        up 1
    end
```

and also

```
updater(deskofperson) ← prog(e,desk)
    failure
```

With these definitions we have introduced a new relation into our model, where the access function is fully described in terms of already existing access functions. Using the updater definition we did also eliminate, for infological reasons, the possibility that independent tuples may be introduced into the relation r5. That is, a person may not have a desk when he and the desk do not belong to the same department.

The semantic extension capabilities may also be used to provide different infological models for the various classes of data base users, e.g. some kind of schema-subschema correspondence.

J.R. ABRIAL defines a number of additional facilities for the relational data base, but we shall restrict ourself to the techniques and operators introduced so far. The main areas left out are

context (environment) considerations

process creation and control (to provide for the execution of more than one program at the same time).

After discussing the basic definitional facilities for binary relational models we shall now attempt to describe the BRM formally using the model defining capabilities of the BRM itself.

4.6. The Formal Description of the Binary Relational Model

Using the concepts of categories, relations, access functions, operators and programming facilities of the BRM we are now able to describe the semantic meaning of the model itself with these mechanisms. The total formal description becomes quite complicate and cannot be presented here. The interested reader may find additional parts in J.R. ABRIAL [7].

All data needed for the semantic description of the actions possible in a BRM has to be kept in the form of binary relations with associated access functions. This organization closely corresponds to the state and

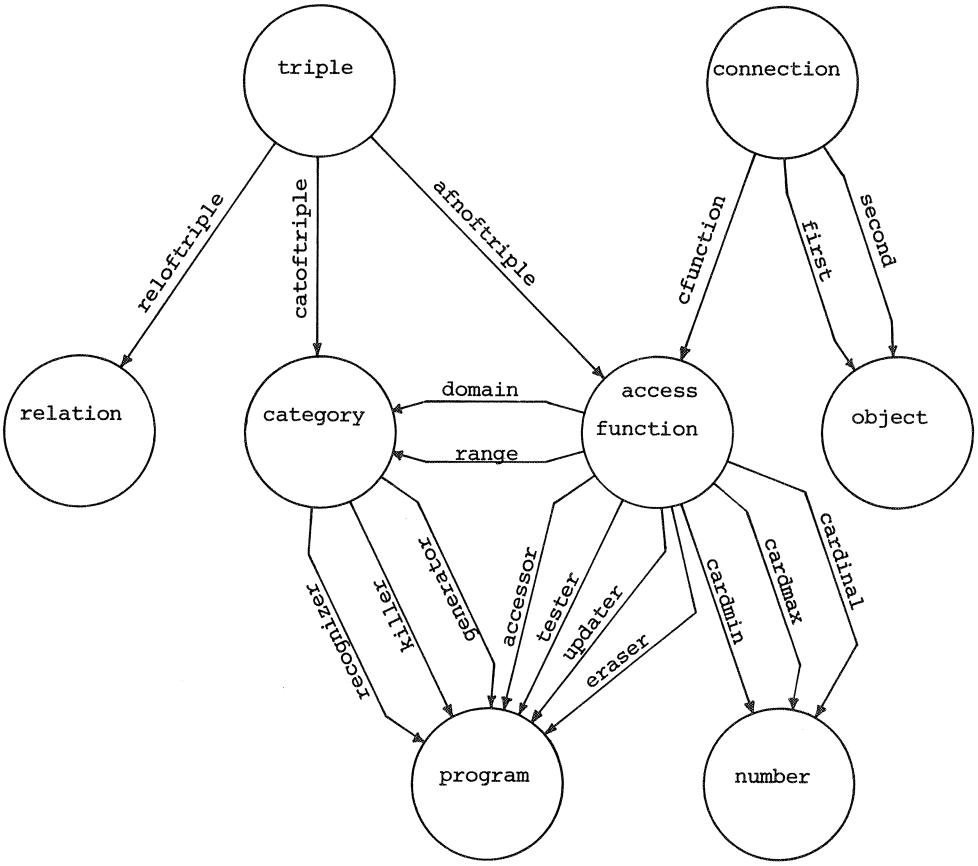

triple
connection
reloftriple
catoftriple
afnoftriple
cfunction
first
second
relation
category
domain
range
access
function
object
recognizer
killer
generator
accessor
tester
updater
eraser
cardmin
cardmax
cardinal
program
number

figure 7

the basic state modification operations as they are introduced with other formal description techniques (see chapter 5). In Figure 7 the graph representation of the required data categories and access functions is given. The corresponding relations are immediately derivable from the graph and are shown in Figure 8. The individual ranges for the cardinality of the access functions will be explained later.

We now discuss the individual categories and access-function together with some of the operations to be defined with their help. Notice, that the graph is not complete. Only the parts needed for our further discussions have been shown.

Categories

The four basic data categories of objects used in the formal description of the model are

relation
category
access function
program

identifying the actual relations, categories, access functions, and programs used in the binary relational model. The other categories are introduced for description purposes and will be explained as we proceed.

Semantic Extension

The last seven relations in Figure 8 provide for the formal description of the semantic extension facility of the BRM. They connect the programs describing special actions for the operations shown in Figure 6 with the categories or access functions for which they have been defined.

Generator-Actions for the Basic Categories

The creation of new BRM relations, categories and access functions requires nonstandard actions.

The generate process for access functions is defined by the program

generator(access function) ← afn = prog(n_1,n_2)
  if ¬(n_1 is number ← n_2 is number)
    then failure end
  f ← std
  cardmin(f) ← n_1
  cardmax(f) ← n_2
  return(f)

```
rel(triple,relation,reloftriple = afn(1,1),afn(2,2))
rel(triple,category,catoftriple = afn(1,1),afn(0,∞))
rel(triple,access function,afnoftriple = afn(1,1),afn(1,2))

rel(access function,number,cardmin = afn(1,1),afn(0,∞))
rel(access function,number,cardmax = afn(1,1),afn(0,∞))
rel(access function,number,cardinal = afn(1,1),afn(0,∞))
rel(access function,category,domain = afn(1,1),afn(0,∞))
rel(access function,category,range = afn(1,1),afn(0,∞))

rel(connection,access function,cfunction = afn(1,1),afn(0,∞))
rel(connection,object,first = afn(1,1),afn(0,∞))
rel(connection,object,second = afn(1,1),afn(0,∞))

rel(category,program,generator = afn(0,1),afn(0,∞))
rel(category,program,killer = afn(0,1),afn(0,∞))
rel(category,program,recognizer = afn(0,1),afn(0,∞))
rel(access function,program,accessor = afn(0,1),afn(0,∞))
rel(access function,program,tester = afn(0,1),afn(0,∞))
rel(access function,program,updater = afn(0,1),afn(0,∞))
rel(access function,program,eraser = afn(0,1),afn(0,∞))
```

Figure 8

We can see that the cardinality restraints of access functions are established at the generate time. The construct afn = prog( ) gives the program the special permanent name afn used when a new access function is defined, e.g. in the definition of relations.

The creation of new relation objects is a little more complicate:

```
generator(relation) ← rel = prog(c1,c2,f12,f21)
   if ¬(c1 is category ∧ c2 is category ∧ f12 is access function ∧
        f21 is access function)then failure end
   r ← std
   generate triple(r,c1,f12)
   domain(f12) ← c1
   range(f12) ← c2
   generate triple(r,c1,f21)
   domain(f21) ← c2
   range(f21) ← c1
   return(r)
```

where the new elements of the category triple are established by

```
generator(triple) ← prog(r,c,f)
   if ¬(r is relation ∧ c is category ∧ f is access function)
            then failure end
   t ← std
   reloftriple(t) ← r
   catoftriple(t) ← c
   afnoftriple(t) ← f
   return(t)
```

The cardinality restraints shown for the access functions of the relations given in Figure 8 are an immediate consequence of this definition of generator(relation):

- The access function and its inverse access function are established as separate entries in the ternary relation symbolized by the category "triple".

The generation of new categories is defined by

```
generator(category) ← cat = prog( )
c ← std
return(c)
```

where the only difference from standard generator action is the introduction of the special name cat for the generation capability of categories.

The Generate-Operator

When a new data object is to be created for a BRM category the generate operator is to be used. Its semantic meaning is established through the program

```
generate = prog(c)
   if ¬(c is category) then failure end
   if ¬(generator(c) = nothing)
      then return(generator(c))
      else return(standard generator(c))
```

where a test is made whether a nonstandard generator action has been supplied for the category c by testing the presence of such a program with generator(c) = nothing i.e. does not exist.

The standard generator used above and in the generator( ) definitions via std is defined by

```
standard generator(c) ← prog( )
   x ← unique name
   return(x)
```

where unique name is a not further defined function delivering upon request unique names.

The Creation of new Relational Tuples

When the operation :∋ is to be performed, e.g. f(x):∋y, that is a new tuple (x,y) is to be inserted in the relation identified by the access function f, the following programs will be processed:

```
:∋ = prog(f,x,y)
   if ¬(f is access function ∧ x is domain(f) and y is range(f))
      then failure end
   if y ∈ f(x) then return end
   if (cardinal(f) = cardmax(f) ∨
       cardinal(inv(f)) = cardmax(inv(f)) then failure end
   if ¬(updater(f) = nothing)
      then updater(f)(x,y)
      else if ¬(updater(inv(f)) = nothing)
              then updater(inv(f))(y,x)
              else standard updater(f)(x,y)
                   standard updater(inv(f))(y,x)
           end
      end
      return
```

where the standard updater operation is defined by the program

```
standard updater(f) ← prog(x,y)
c ← generate connection
c function(c) ← f
first(c) ← x
second(c) ← y
return
```

A new element of the ternary relation represented by the category "connection" is created to indicate the presence of the tuple (x,y) in the relation identified through its access function f.

The other semantic extensions and operator definitions required for the full formal description of the binary relational model can be formulated using the same techniques. The interested reader may attempt their definition. The description of some of them is given by J.R. ABRIAL [7].

5. THE FORMAL REPRESENTATION OF DATA BASES

5.1. General Considerations

The two relational models described in the preceding chapter are both based on the same basic data organization concepts, i.e. relations. Still, when attempting complete journal comparisons of the models the task soon becomes very cumbersome. Of course there exist many other data base models, usually more implementation oriented than the relational models, such making their comparison even less rewarding. Some of the reasons for this problem are very often imprecise terminology, wrong specification documents (e.g. a "user's guide" against an "implementation guide") and not at least the amount of work involved in such a feat.

How can we avoid some of the problems:

1) Use the same data base concepts in all models, the same languages:
   This is not a realistic approach as we ourselves have pointed out that even a single data base should contain at least three models, i.e. a subject matter model, the infological model and the datalogical model. Across information systems the problem gets even harder.
2) Describe the models precise:
   Formal description methods have been applied a number of times even to

complex subject areas, e.g. J.R. ABRIAL's selfdescribing binary relational model, the Vienna IBM Laboratory using the Vienna Definition Language to describe PL/I[16,17]. N. WIRTH and C.A.R. HOARE using an axiomatic technique for the description of PASCAL [18]. But to compare different data base philosophies when they are reflected in different programming languages and formally described using different definition techniques is a complicated task at least.

The most promising way which seems to be open is to try to combine the two solution attempts. In Figure 9a,b a *common* abstract language is introduced as a *representation language* for all data base models and the same language is then used to develop a formal description of the concepts of model types. Applied to, say a binary relational data base, this would mean we would have to express all its different relations, the manipulation functions and semantic extensions using the abstract representation language. In addition we would develop (using the same abstract language) a formal description of all the concepts used in binary relational data bases e.g. like J.R. ABRIAL has used the real binary relational language to express the concepts of his model.

When basing our investigations on such a concept we are now able to procede in a much more orderly and less troublesome way:

1) Using the abstract description of different models (they are all written using the same definition mechanisms!) a comparative study of the models can be made.
2) The abstract description of one model concept can be used for a conceptual study of models which can be developed using the concept. Theorems about their expressive power could be developed, consistency studies could be undertaken.
3) Specific data base models could be investigated. The translation routines for mapping it into another model e.g. an infological model into a datalogical model could be investigated and proven correct. The infological equivalence of differently organized specific data base models could formally be established.

Using such a concept some work (see E.J. NEUHOLD [10-12] and H. BILLER [12]) has already been done. It is based on an expanded version of the Vienna Definition Language which has been found quite convenient to express the many different concepts which must be covered by such an approach. Of course,

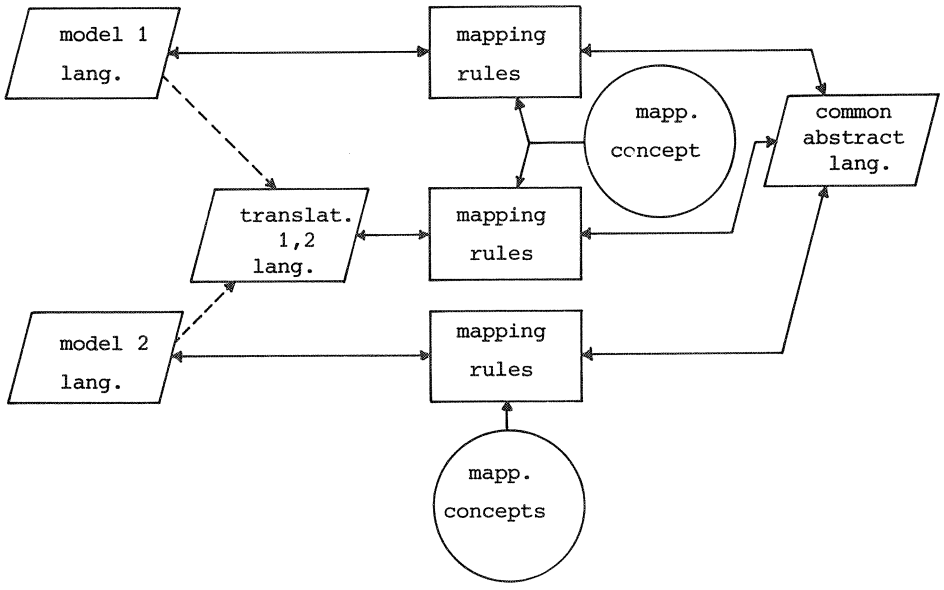


Figure 9a: Development of representation language mapping rules

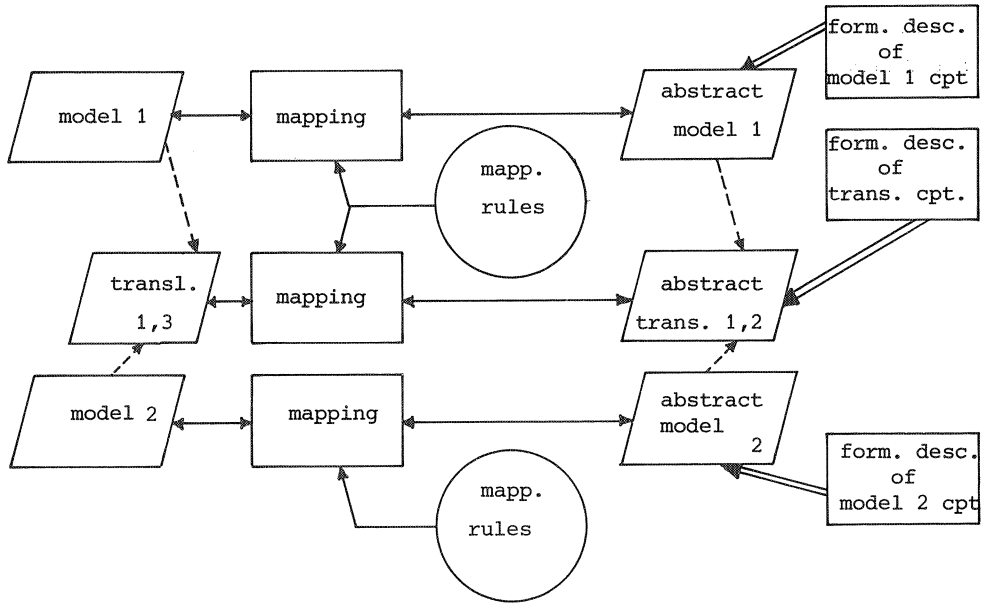


Figure 9b: Development of specific abstract models

considering the large area to be investigated these contributions cover very little ground, but we see no principal problem that could arise with our approach.

In the following a brief introduction into those parts of VDL are given which we shall meet later. The other concepts can be found in the literature [16,17].

5.2. The Extended Vienna Definition Language

The Vienna Definition Language is based on the idea that an abstract interpreter working on an abstract data object (its state) can represent the semantic meaning of the interpreted language, algorithm, concept, etc.

Some of the concepts incorporated in VDL are listed below:

- Conditional expressions

They are used in LISP like form

$$(prop_1 \rightarrow expr_1,\ prop_2 \rightarrow expr_2,\ \ldots\ prop_n \rightarrow expr_n)$$

where $prop_i$ are thruth valued expression and $expr_i$ are expressions yielding genal objects as values.

- Functional composition

$(f \circ g)(x_1,\ldots,x_n) \underset{Df}{=} f(g(x_1,\ldots,x_n))$

- Operators and basic values true(T) and false(F)

$T,\ F,\ \neg,\ \wedge,\ \vee,\ \equiv,\ \supset,\ \exists,\ \forall,\ \exists S,\ \forall S,\ \cap,\ \cup,\ -,\ \in,\ \subset,\ \subseteq,\ <,\ \leq,\ \geq,\ >,\ =,\ \neq,\ Et_{i=0}^{n}$

The operators are used in their conventional meaning but in $\exists S$ and $\forall S$ the S specifies the range of all variables bound by the quantifier. The operator $Et_{i=0}^{n}$ defines an n-ary conjunction.

- Abstract objects and selectors

It is assumed that there exists a set of *elementary objects* E0 and a countable set of *simple selectors* S.

We define S^* to be the set of all $s_1 \circ s_2 \circ \ldots \circ s_n$ where $s_i \in S$, $1 \leq i \leq n$. The identity element with respect to the operation $\circ$ is denoted by I. The sequences $s_1 \circ s_2 \circ \ldots \circ s_n$ are termed *(composite) selectors.*

An *(abstract) object* is defined by a finite set of pairs $\langle\kappa:eo\rangle$ called the *characteristic set* C where $\kappa \in S^*$ and $eo \in EO$.

Example: The characteristic set of the object x, denoted by $\bar{x}$ is given by

$$\bar{x} = \{\langle s_1:eo_1\rangle, \langle s_3 \circ s_2:eo_2\rangle, \langle s_4 \circ s_2:eo_3\rangle\}$$

The object may be represented as a tree with named branches

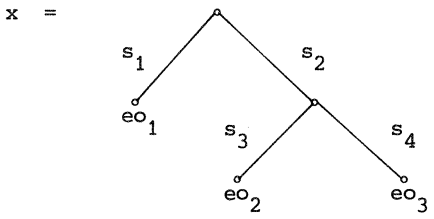


Figure 10

To restrict objects to trees of the above nature the characteristic set C of a well formed object must satisfy the condition

$$(\langle\kappa_1:eo_1\rangle, \langle\kappa_2:eo_2\rangle \in C \wedge \langle\kappa_1:eo_1\rangle \neq \langle\kappa_2:eo_2\rangle) \supset \neg dep(\kappa_1,\kappa_2)$$

where

$$dep(\kappa_1,\kappa_2) = (\exists\kappa)(\kappa_1 = \kappa \circ \kappa_2 \vee \kappa_2 = \kappa \circ \kappa_1)$$

Intuitively:

a) Branches identifying the immediate descendents of a node must be uniquely named, or

b) elementary objects may only be attached at leaves of the tree.

The characteristic set of an elementary object eo is $\{\langle I:eo\rangle\}$. The empty characteristic set defines the *null object* Ω.

- Functional application of selectors

The application of a composite selector κ to an object x, written $\kappa(x)$, is defined by the characteristic set

$$\overline{\kappa(x)} = \{<\kappa_1:eo>:<k_1 ok:eo> \in \bar{x}\}$$

Example: Using the object x of Figure 10 we apply s_2 and get

$$\overline{s_2(x)} = \{<s_3:eo_2>,<s_4:eo_3>\}$$

- The μ-operator (i.e. "make"-operator)

Given an object x and a pair $<\kappa:y>$ with $\kappa \in S^*$ and y an object we introduce an operation $\mu(x;<\kappa:y>)$ defined by

$$\overline{\mu(x,<\kappa:y>)} = \{<\tau:eo>:<\tau:eo> \in \bar{x} \wedge \neg dep(\kappa,\tau)\} \cup \{<\tau\circ\kappa:eo>:<\tau:eo> \in \bar{y}\}$$

The first part represents the characteristic set $\bar{x}$ whose κ component has been deleted; the second part is the characteristic set of an object with y as its κ component only.

Example: Using the object x of Figure 10 and y defined by

$$\bar{y} = \{<s_5\circ s_4:eo_4>,<s_1\circ s_4:eo_1>\}$$

we get

$$\overline{\mu(x,<s_2:y>)} = \{<s_1:eo_1>,<s_5\circ s_4\circ s_2:eo_4>,<s_1\circ s_4\circ s_2:eo_1>\}$$

- Extensions of the μ-operator

a) $\mu(x;<\kappa_1:y_1>,<\kappa_2:y_2>,\ldots,<\kappa_n:y_n>) \underset{Df}{=} \mu(\mu(x;<\kappa_1:y_1>);<\kappa_2:y_2>,\ldots,<\kappa_n:y_n>)$

b) $\mu(x;\{<\kappa:y>:prop\})$

The second operand must be a finite set of pairs $<\kappa:y>$. Arranging these elements in any linear order and applying the operation defined in a) yields the result, provided that the order of the pairs chosen is not significant. If the order is significant, the result is undefined. In addition $\mu(x;\{\}) = x$

c) $\mu_D(x;M)$

The second operand must be a finite set of objects. If $|M| = n$, then n distinct selectors are taken out of the countable set of selectors D and pairs <s:e> are formed with the n elements of M giving a set of n pairs to which the μ-operation like in b) is applied.

Note: This operation is needed when no specific selectors are specified but where a set of objects still has to be combined into a single larger object, e.g. records in a file.

- Predicates

Special classes (categories) of objects are defined by predicates applicable to the objects. The members of a category of objects defined by a predicate are precisely those objects which satisfy the predicate.

Predicates may be a priori given, may be constructed using first order predicate logic or are defined by the special forms:

a) $$(<s_1:p_1>,<s_2:p_2>,\ldots,<s_n:pe_n>)(x) \underset{Df}{=} (\exists x_1 x_2 \ldots x_n)\left(\underset{i=0}{\overset{n}{Et}}\ p_i(x_i) \wedge x = \mu(\Omega;<s_1:x_1>,<s_2:x_2>,\ldots,<s_n:x_n>)\right)$$

Example: The predicate

$$p \underset{Df}{=} (<s_1:p_1>,<s_3 \circ s_2:p_2>,<s_4 \circ s_2:p_3>)$$

defines the category of objects x such that

$$\bar{x} = \{<s_1:eo_1>,<s_3 \circ s_2:eo_2>,<s_4 \circ s_2:eo_3>\}$$

where $p_i(eo_i)$, $1 \leq i \leq 3$, holds.

b) $$p\text{-}coll_D(x) \underset{Df}{=} (\exists x_1 x_2 \ldots x_n s_1 s_2 \ldots s_n)\left(\underset{i=1}{\overset{n}{Et}}\ (p(x_i) \wedge s_i \in D) \wedge x = \mu(\Omega;<s_1:x_1>,<s_2:x_2>,\ldots,<s_n:x_n>)\right)$$

The predicate defines a category of objects, called a *collection* of objects, where the immediate subordinate selectors do not have to be specified but are members of a given set D.

- The abstract interpreter

The abstract interpreter to be used in interpreting abstract objects (states) and defining by this interpretation the semantic meaning of the described entity is specified by the quadruple

$$(\Sigma, \lambda, \xi_0, \Sigma_0)$$

where Σ is a set of states, where Σ is a subset of the wellformed objects. λ is a state transition function

$$\lambda: \Sigma \rightarrow \Sigma^{+}$$

ξ_0 is the initial state, $\xi_0 \in \Sigma$
Σ_e is a set of endstates $\Sigma_e \subseteq \Sigma$

A computation is defined as a sequence

$$\xi_0 \; \xi_1 \; \cdots \; \xi_n \; \cdots$$

where $\xi_{i+1} \in \lambda(\xi_i)$. The computation terminates if there exists an n such that $\xi_n \in \Sigma_e$.

The state transition function defines all the actions of the interpreter. To specify these actions for a specific formal description requires quite an extensive "abstract language" type mechanism. We shall not need it for the remaining discussions. therefore the interested reader is referred to the literature [16,17].

Extensions to the abstract interpreters have also been developed (see E.J. NEUHOLD [19]) but again they leave the frame of our present investigations.

5.3. Relational and Hierarchical Data Organizations

To illustrate the formal description technique and its applicability we specify a few proporties of hierarchical and normalized n-ary relational data organizations and use formal means for a brief comparison. Additional applications may be found in [10-12].

Notice, the following discussions are concerned with classes of relational and hierarchical models and not with a specific data base represented in hierarchical or relational form.

In accordance to the approach outlined in section 5.1 we do not investigate the relational view as introduced by E.F. CODD directly but rather an abstract version of it, where the mapping from the concrete to the abstract version (see Figures 9a,b) is assumed to have happened. This conversion could either have been defined formally, similar to the translator

technique used for transforming concrete PL/I into its abstract form (see [17] and literature referenced there) or it could have been developed (as it actually has happened) by informal reasoning.

We define the various components of the relational view of data first, to be followed by the formal description of the hierarchical organization.

The Relational Model

Elementary objects: ER
numbers: defined by the predicate is-number
character strings: defined by the predicate is-char-string

Simple selectors: S
relation selectors: the set RS is defined by the predicate is-rs
domain selectors: the set DS is defined by the predicate is-ds
tuple selectors: the set TS is defined by the predicate is-ts
The sets RS, DS and TS are distinct.

$$RS \cap DS = RS \cap TS = DS \cap TS = \{\}$$

Wellformed relational models in first normal form

$$\text{is-relat-model-ln}(x) \underset{Df}{=} \text{is-relation-group}$$

$$\text{is-relation-group}(x) \underset{Df}{=} \forall Ss(s(x) \neq \Omega \supset (\text{is-rs}(s) \wedge \text{is-relation}(s(x))))$$

$$\text{is-relation}(x) \underset{Df}{=} \forall Ss(s(x) \neq \Omega \supset (\text{is-ts}(s) \wedge \text{is-tuple}(s(x)) \wedge \forall St((t(x) \neq \Omega \wedge t \neq s) \supset (s(x) \neq t(x) \wedge \underline{\text{STRUCT-EQV}}(s(x),t(x))))))$$

$$\text{is-tuple}(x) \underset{Df}{=} \forall Ss(s(x) \neq \Omega \supset (\text{is-ds}(s) \wedge \text{is-elem-item}(s(x))))$$

$$\text{is-elem-item}(x) \underset{Df}{=} \text{is-number}(x) \vee \text{is-char-string}(x)$$

where

$$\underline{\text{STRUCT-EQV}}(x,y) \underset{Df}{=} (\text{is-number}(x) \rightarrow \text{is-number}(y), \text{is-char-string}(x) \rightarrow \text{is-char-string}(y) \text{is-tuple}(x) \rightarrow \forall Ss((s(x) \neq \Omega \equiv s(y) \neq \Omega) \wedge \underline{\text{STRUCT-EQV}}(s(x),s(y)))$$

The restrictions placed into the definitions of the predicate is-relation ensure that all tuples are different but have the same structural de-

finition as specified by the function STRUCT-EQV.

The Hierarchical Model.

Elementary objects: EH
numbers: defined by the predicate is-number
character strings: defined by the predicate is-char strings
selectors: the set S^* defined by the predicate is-sel

Simple selectors: S
group selectors: the predicate is-gs defines the set GS
collection selectors: the predicate is-cs defines the set CS
The sets GS and CS are distinct

$$GS \cap CS = \{\}$$

Wellformed hierarchical models

$$\text{is-hierarch-db}(x) \underset{Df}{=} \text{is-group}(x)$$

$$\text{is-group}(x) \underset{Df}{=} \forall Ss(s(x) \neq \Omega \supset (\text{is-gs}(s) \wedge \text{is-data-constr}(s(x))))$$

$$\text{is-data-constr}(x) \underset{Df}{=} \text{is-group}(x) \vee \text{is-collection}(x) \vee \text{is-elem-data}(x)$$

$$\text{is-collection}(x) \underset{Df}{=} \forall Ss(s(x) \neq \Omega \supset (\text{is-cs}(s) \wedge \text{is-data-constr}(s(x))))$$

$$\text{is-elem-data}(x) = \text{is-number}(x) \vee \text{is-char-string}(x) \vee \text{is-selector}$$

This definition of hierarchical models allows very general data structures. For example

-files, i.e. collections of groups, where each element, i.e. record, has a different structure,
- networks, where the network properties are expressed by the use of selectors as elementary objects,

are part of wellformed hierarchical models

We shall now illustrate possible investigations by describing the interrelationships of relational and hierarchical data organizations.

The Relational Model as a Restricted Hierarchical Model

Elementary objects

ER = EH - S^* (i.e. no selectors may appear as elementary objects in a relational mode)

Selectors.

We require the relations

$$RS \subset GS$$
$$DS \subset GS$$
$$TS \subseteq CS$$

to hold.

We now provide a function RELATIONAL-MOD(x) which, when applied to a hierarchical data base x, establishes whether the hierarchical data base is of a form allowed for relational models.

$$\text{RELATIONAL-MOD}(x) \underset{Df}{=} (\exists Ss(s(x) \neq \Omega \wedge (\neg\ \text{is-rs}(s) \vee \text{is-group}(s(x)) \vee \text{is-elem-data}(s(x)))) \longrightarrow F,\ T \longrightarrow \forall Ss(s(x) \neq \Omega \subset \text{RELATION}(s(x))))$$

$$\text{RELATION}(x) \underset{Df}{=} (\exists Ss(s(x) \neq \Omega \wedge (\text{is-collection}(s(x)) \vee \text{is-elem-data}(s(x)) \vee \neg\ \text{is-ts}(s))) \vee \exists Ss_1 \exists Ss_2 (s_1 \neq s_2 \wedge s_1(x) \neq \Omega \wedge s_2(x) \neq \Omega \wedge (s_1(x) = s_2(x) \vee \neg\ \underline{\text{STRUCT-EQV}}(s_1(x), s_2(x)))) \longrightarrow F,\ T \longrightarrow \forall Ss(s(x) \neq \Omega \supset \text{TUPLE}(s(x))))$$

$$\text{TUPLE}(x) \underset{Df}{=} (\exists Ss(s(x) \neq \Omega \wedge (\text{is-group}(s(x)) \vee \text{is-collection}(s(x)) \vee \neg\ \text{is-ds}(s))) \longrightarrow F,\ T \longrightarrow \forall Ss(s(x) \neq \Omega \supset \text{ELEMENT}(s(x))))$$

$$\text{ELEMENT}(x) \underset{Df}{=} \text{is-sel}(x) \longrightarrow F,\ T \longrightarrow T$$

The definition of the truthvalued function RELATIONAL-MOD has been made in such a way, that the restrictions which must be placed on a hierarchical data base in order to make it a wellformed relational data base all appear in the first proposition of the various functions.

Similar formal descriptions and investigations of other models have been made. For example in H. BILLER & E. NEUHOLD [12] a description for the SCHEMATA and SUBSCHEMATA of the DBTG Report [20] has been given. The same paper also contains formal criterea, which ensure a usage-equivalence of different data bases. This equivalence definition is given both for retrieve and change operation on data bases and it takes (at least in part) infological interpretations into account.

5.4. A Sample Data Base

Before we close our discussions let us investigate how a specific data base may look when it is specified using the formal description apparatus. We select a very simple model of an airline reservation data base and present it in a form compatible with the relational view of data, that is, in the form of abstract, normalized, n-ary relations.

For these relations we give the predicate definitions:

is-relational-model $\underset{Df}{=}$
(<s-flight: is-flight-coll$_{TS}$>,
<s-reservation: is-reservation-coll$_{TS}$>)

is-flight $\underset{Df}{=}$ (<s-flight # : is-integer>,
<s- # seats: is-integer>,
<s-departure: is-char string>,
<s-from: is-char-string>,
<s-to: is-char string>)

is-reservation $\underset{Df}{=}$ (<s-flight # : is-integer>,
<s-date: is-char-string>,
<s-ticket # : is-integer>,
<s-seat # : is-char-string>)

Where the selectors s-flight and s-reservation are elements of the relation selectors RS. All other explicit selectors are elements of the domain selectors DS.

Using VDL expanded we could now specify a different model of the airline reservation data base, e.g. a hierarchical model, and again using VDL-expanded describe the mapping from one to the other. The formal system could then be used for an automatic translation of user queries oriented toward one of the models into user queries oriented toward the other. The formal system also allows systematic considerations of the time efficiency of such translations and of possible optimization strategies. In addition, the formally specified translation mechanism provides a precise framework for the investigation of the infological equivalence of the two models. Some of this work may be found in the literature [10-12] other is left to the interested reader.

6. SUMMARY

Starting with an infological analysis of the user of data base systems, i.e. a human being, we have defined a few required charateristics of these information systems. Proceding first to n-ary relational models, which are probably the most formally defined models where large commercial implementations are at least under way if not finished, we observed a number of difficulties for further formal investigations. To simplify the problem for the moment we concentrated on binary relational models and their formal description.

However we then wanted to expand our view again, but on a very formal bases. For this reason the extended VDL concepts were introduced to allow both, the abstract description of specific data bases, but also the formal description and investigation of the different approaches to the design of information systems.

We can now conclude that some progress has been made in the development of formal properties of data bases. Much work steill remains to be done, especially in the area of unified description of information systems and in the field of infological interpretations to be given to the data stored in the data bases.

REFERENCES

[1] E.F. CODD, *A Relational Model of Data for Large Shared Data Banks*, C. of ACM, Vol. 13, No. 6, June 1970.

[2] E.F. CODD, *Further Normalization of the Data Base Relational Model*, Proc. of Courant Symp. on Data Base Systems, ed. R. Rustin, Prentice Hall, 1972.

[3] E.F. CODD, *Relational Completeness of Data Base Sublanguages*, Proc. of Courant Symp. on Data Base Systems, ed. R. Rustin, Prentice Hall, 1972.

[4] E.F. CODD, *A Data Base Sublanguage founded on Relational Calculus*, Proc. of SIGFIDET 1971, ACM, Nov. 1971.

[5] E.F. CODD, *Interface Beta*, Private Communication, 1971.

[6] E.F. CODD, *Seven Steps to Rendezvous with the Casual User*, Proc. of IFIP-TC-2 Conf. "Data Base Management Systems", Corsica, 1974.

[7] J.R. ABRIAL, *Data Semantics*, Proc. of IFIP-TC-2 Conf. "Data Base Management Systems", Corsica, 1974.

[8] B. SUNDGREN, *An Infological Approach to Data Bases*, Urval No. 7, National Central Bureau of Statistics, Stockholm, 1973.

[9] B. SUNDGREN, *Conceptual Foundation of the Infological Approach to Data Bases*, Proc. of IFIP-TC-2 Conf. "Data Base Management Systems", Corsica, 1974.

[10] E.J. NEUHOLD, *Data Mapping: A Formal Hierarchical and Relational View*, Report No. 10, Inst. for Angewandte Informatik, Univ. Karlruhe, 1973. (also Courant Symp. on Data Base Systems, 1971)

[11] E.J. NEUHOLD, *The Use of Formal Description Techniques in Large Data Banks*, Proc. of Int. Comp. Sym. of ACM, Venice, 1972.

[12] H. BILLER, E.J. NEUHOLD, *Formal View on Schema-Subschema Correspondence*, Proc. IFIP Cong. 1974.

[13] C. PAIR, *Formalization of the Notions of Data, Information and Information Structure*, Proc. of IFIP-TC-2 Conf. "Data Base Management Systems" Corsica, 1974.

[14] R. BAYER, *Storage Characteristics and Methods for Searching and Addressing*, Proc. IFIP Cong. 1974.

[15] R.W. TAYLOR, *Generalized Data Base Management System Data Structure and their mapping to Physical Storage*, Ph.D. thesis, Univ. of Michigan, 1971.

OTHER REFERENCES

[16] E.J. NEUHOLD, *The Formal Description of Programming Languages*, IBM System Journal, Vol. 10, No. 2, 1971.

[17] P. LUCAS, K. WALK, *On the Formal Description of PL/I*, Ann. Rev. in Autom. Programming, Vol. 6, Part 3, 1969.

[18] C.A.R. HOORE & N. WIRTH, *An Axiomatic Definition of thePrograming Language Pascal*, Acta Informatica, Vol. 2, No. 4, 1973.

[19] E.J. NEUHOLD, *The Formal Semantic of Operating Systems*, ACM Int. Computing Symposium, Davos, 1973.

[20] CODASYL, *Data Base Task Group*, Report to the Programming Language Committee, April 1971.

COMPLEXITY OF MATRIX ALGORITHMS by M.S. PATERSON

MATHEMATICAL CENTRE TRACTS 63, 1975, 181-215

COMPLEXITY OF MATRIX ALGORITHMS

M.S. PATERSON

Warwick University, Coventry, (GB)

1. INTRODUCTION

In studying the complexity of algorithms we develop techniques for evaluating the amount of 'resource', usually time or storage space, used by new or existing programs; we attempt to prove lower bounds for the resources required by *any* program which performs a given task; we look for interesting relationships among different algorithms for the same problem or explore possible connections between seemingly unconnected problem areas; and in all we aim for a deeper understanding of the essential difficulties of, and possible solutions to, a variety of computational problems.

In this series of lectures I shall only be considering a restricted class of algorithms, all concerned with matrices. There are several reasons for my choice. Firstly, matrix methods have important applications in many scientific fields, and frequently account for large amounts of computer time. The practical benefit from improvements to algorithms is therefore potentially very great. Secondly the basic algorithms, such as matrix multiplication are simple enough to invite total comprehension, yet rich enough in structure to offer challenging mathematical problems and some elegant solutions. Finally, the subject matter is well enough known for us to start immediately without an extensive introduction.

Definitions of matrix arithmetic.

If A is a $p \times q$ matrix and B a $q \times r$ matrix then their *product* $C = A.B$ is a $p \times r$ matrix with entries given by

$$c_{ij} = \sum_{k=1}^{q} a_{ik}.b_{kj} \qquad \text{for } i = 1,\ldots,p \text{ and } j = 1,\ldots,r.$$

Sometimes it is also useful to think of A as composed of its p row vectors $\underline{A}_1,\ldots,\underline{A}_p$, and B as composed of its r column vectors $\underline{B}_1,\ldots,\underline{B}_r$. Then c_{ij}

is the *inner product* of vectors $\underline{A}_i$ and $\underline{B}_j$.

The *sum* of two matrices A, B with the same dimensions is the matrix C = A + B given by

$$c_{ij} = a_{ij} + b_{ij} \qquad \text{for all } i, j.$$

Arithmetic complexity.

A computer program for an arithmetic algorithm will usually execute many instructions other than the explicit arithmetic operations of the algorithm. There will, for example, be fetching, storing, loading and copying operations. The proportion of the total execution time which is spent on such 'overheads' will be very dependent on the computer and programming language used. For simplicity and independence we shall usually take account only of the arithmetic operations involved. This measure will be referred to as the *arithmetic complexity*. The consequences of this simplification in particular practical applications must of course be carefully considered.

It is easy to see that in the product of a p × x matrix by a q × r matrix (a p × q × r *product*) each of the pr entries of the product can be computed using q multiplications and q - 1 additions. We can write this arithmetic complexity as $q \cdot \underline{m} + (q-1) \cdot \underline{a}$ and then get a total for the p × q × r product of

$$pqr \cdot \underline{m} + p(q-1)r \cdot \underline{a}.$$

The sum of two p × q matrices uses only $pq.\underline{a}$. We shall never distinguish between the complexity of a basic addition and a subtraction and such an operation will be referred to as an *addition/subtraction* (a/s). Similarly we shall sometimes write "*multiplication/division*' (m/d).

The kinds of question to which we shall seek answers are:

"*Can product be computed by another algorithm using fewer operations?*"

"*What is the minimum number of arithmetic operations required?*"

The first question is answered affirmatively; the second has as yet only very incomplete answers.

2. WINOGRAD'S ALGORITHM FOR MATRIX PRODUCT. [Win 70]

To compute $a_1.b_1 + a_2.b_2$ certainly requires 2 multiplications/divisions (and 1 addition/subtraction), and more generally we shall show in Section 6 that $a_1.b_1 +\ldots+ a_n.b_n$ requires n multiplication/divisions. An alternative way to compute $a_1.b_1 + a_2.b_2$ is the following.

$$\mu_1 = a_1.a_2$$

$$\mu_2 = b_1.b_2$$

$$\mu_3 = (a_1+b_2).(a_2+b_1)$$

$$\text{result} = \mu_3 - \mu_1 - \mu_2.$$

It needs considerable insight to see the significance for matrix product of this identity which, at first glance, appears merely to take more multiplications and more additions than the obvious algorithm. The important feature is that μ_1 and μ_2 are multiplications which involve only a's and only b's respectively. Why is this so important?

We have already remarked that matrix product can be regarded as finding the inner product of each row of one matrix with each column of the other matrix. If in the sub-algorithm used for inner product there is a computation involving the elements from only one of the vectors then it can be performed just once for that row (column) instead of every time that vector is used. This idea of 'pre-processing' is very important and leads in this instance to *Winograd's algorithm*. The algorithm is described first for the simple case of n × n matrices with n even.

For $\underline{x} = (x_1,\ldots,x_n)$ define

$$W(\underline{x}) = x_1.x_2 + x_3.x_4 +\ldots+ x_{n-1}.x_n.$$

(i) For each row $\underline{A}_i$ of A compute $W(\underline{A}_i)$, and for each column $\underline{B}_j$ of B compute $W(\underline{B}_j)$.

(ii) For each pair (i,j), if $\underline{a} = \underline{A}_i$ and $\underline{b} = \underline{B}_j$, compute

$$\underline{a}.\underline{b} = (a_1+b_2).(a_2+b_1) + (a_3+b_4).(a_4+b_3) + \ldots$$

$$\ldots + (a_{n-1}+b_n).(a_n+b_{n-1}) - W(\underline{a}) - W(\underline{b})$$

The arithmetic complexity for (i) is

$$2n(n/2.\underline{m}+(n/2-1).\underline{a})$$

and for (ii) is

$$n^2.(n/2.\underline{m}+(3n/2+1).\underline{a})$$

which gives a total of

$$(n^3/2 + n^2).\underline{m} + (3n^3/2 + 2n(n-1)).\underline{a} .$$

Neglecting the lower order terms, we have exchanged roughly $n^3/2$ multiplications for an extra $n^3/2$ additions/subtractions. The algorithm is easily extended to the general $p \times q \times r$ product. If q is even the algorithm is essentially the same. If q is odd then one elementary multiplication in each inner product is done in the conventional manner and added in separately, which does not significantly affect the arithmetic complexity. The extra storage requirements of Winograd's algorithm are minimal; just one extra location for each row and column is needed to store the value of W.

This algorithm is of obvious value whenever $\underline{m} > \underline{a}$. Typical applications are when the matrix elements are complex numbers or multiple-precision numbers. A significant restriction of the algorithm is that its correctness depends on the commutativity of multiplication. This is seen in the original identity for $a_1b_1 + a_2b_2$ above.

Let us consider the case of complex matrices in further detail. Assuming that the complex numbers are represented by pairs of reals giving their real and imaginary parts, the obvious algorithm to compute

$$(x+iy).(u+iv) = (xu-yv) + i(xv+yu)$$

takes $4\underline{m} + 2\underline{a}$, whereas complex addition costs $2\underline{a}$. This seems a good application for Winograd's algorithm. If we are on the look-out for unusual methods, we may find the following alternative for complex product.

$$\lambda_1 = x.u$$

$$\lambda_2 = y.v$$

$$\lambda_3 = (x+y).(u+v)$$

Then $(x+iy).(u+iv) = (\lambda_1-\lambda_2) + i(\lambda_3-\lambda_1-\lambda_2)$.

Although this identity is reminiscent of the identity underlying Winograd's algorithm, note that commutativity of multiplication need not be assumed here. Since this method uses $3\underline{m} + 5\underline{a}$, instead of $4\underline{m} + 2\underline{a}$, it requires a situation where $\underline{m}$ is much larger than $\underline{a}$ to be useful. If the elements involved are themselves large matrices this condition holds. This observation yields a new class of algorithms for complex matrix product. Note the relevance of the remark above about commutativity.

Given complex matrices, A and B, split them into their real and imaginary parts so that we may write

$$A = X + iY \quad , \quad B = U + iV$$

where X, Y, U, V are real matrices. Then the identity above is used to compute A.B using only 3 real matrix products and 5 real matrix sums.

We now have a plethora of algorithms to consider, of which we identify eight. Given two complex matrices they may be multiplied directly using either the classical method (C) or Winograd's algorithm (W), and then the complex entries can be multiplied in the straight-forward way (S) or the unusual, underhand (?), way (U) given by the above identity. We can denote these methods by

CS, CU, WS, WU.

Alternatively the original matrices may be split up and multiplied by real and imaginary parts separately using methods S or U. The real matrix products required are done by C or W, yielding four more methods

SC, UC, SW, UW.

We shall analyse the arithmetic complexity of these methods for $n \times n \times n$ product as the ratio of $\underline{m}$ to $\underline{a}$ varies. This is only a theoretical exercise since in practice the 'overheads' may be the crucial criterion in a comparison of similar algorithms. We set out in the table below the leading coefficients of the $\underline{m}$ and $\underline{a}$ components of the arithmetic complexity.

Method	*Coefft. of* $n^3.\underline{m}$	*Coefft. of* $n^3.\underline{a}$
CS	4	4
CU	3	7
WS	2	4
WU	$1\frac{1}{2}$	$5\frac{1}{2}$
SC	4	4
SW	2	6
UC	3	3
UW	$1\frac{1}{2}$	$4\frac{1}{2}$

As one would expect from the above discussion, if one is going to split up the matrices initially it should be done with U rather than S, and if the matrices are to be multiplied directly, Winograd's is better than C. Looking at the remaining complexities we find that

(i) if $\underline{m} > \underline{a}$ UW has the lowest

(ii) if $\underline{m} < \underline{a}$ UC has the lowest

(iii) if $\underline{m} = \underline{a}$ WS, UW, UC are the joint leaders

but if lower order terms are taken into account UC has the lowest complexity $(3n^3.\underline{m}+(3n^3+2n^2).\underline{a})$ in case (iii).

In [Bre 70] BRENT compares the running times of some ALGOLW programs for various matrix product algorithms. He concludes that the methods using an initial 'U' splitting cannot be helpful, since he found in practice that no program for complex matrices took as much as three times the time for real matrices. This was because a large part of the total execution time was concerned with initialization, and calculating the indices and addresses of the arguments for operations. A promising approach which I have not tried out in practice but which may overcome some of the inefficiencies in methods such as UC and UW is the following. We take advantage of the circumstance that there are three real matrix products, all of the same dimensions, to be computed and that they may be performed in parallel. If the corresponding operations of these products are interleaved then some of the 'overheads' can be shared.

3. A RECURSIVE METHOD & RECURRENCE RELATIONS

For a different style of algorithm for matrix product we can use partitioned matrices and 'block multiplication'. To simplify matters suppose A, B are n × n matrices with n > 1. If we regard A, B as composed of sub-matrices in the following way

$$A = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right), \quad B = \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right)$$

where A_{11}, B_{11} are r × r matrices, 0 < r < n, then the product is given by

$$A.B = \left(\begin{array}{c|c} A_{11}.B_{11} + A_{12}.B_{21} & A_{11}.B_{12} + A_{12}.B_{22} \\ \hline A_{21}.B_{11} + A_{22}.B_{21} & A_{21}.B_{12} + A_{22}.B_{22} \end{array}\right)$$

That the result is correct is easily proved, and uses only the associativity property of addition. The product A.B is thus computed by performing 8 products of the sub-matrices, followed by 4 sums of the resulting sub-matrices. The sub-matrix products may be done in a similar manner by further partitioning into smaller matrices, and so on until the resulting matrices are small, maybe 1 × 1. Thus we have a recursive procedure for matrix product. If we take $r = \lceil n/2 \rceil$ ($\lceil x \rceil$ = least integer ≥ x) so the partitioning is as nearly as possible into equal parts, and if we write P(n), S(n) for the arithmetic complexity of n × n × n product and n × n sum respectively, we derive the following recurrence relation.

$$P(n) \leq 8\, P(\lceil n/2 \rceil) + 4\, S(\lceil n/2 \rceil)$$

But $S(n) = n^2.\underline{a} = 0(n^2)$ operations, so we have

$$P(n) \leq 8\, P(\lceil n/2 \rceil) + 0(n^2)$$

Recurrence relations of the above form will occur frequently so we shall give below a general solution to such forms. For the above relation this will imply that

$$P(n) = 0(n^{\log_2 8}) = 0(n^3)$$

This comes as no surprise to the observant reader who has seen that precisely the same multiplications are performed as in the 'classical' algorithm and the additions have just been rearranged using associativity.

THEOREM. *If* F *is a non-negative function on the positive integers such that for some* $a \geq 1$, $b > 1$ *and* $\beta \geq 0$,

$$F(n) \leq a.F(\lceil n/b \rceil) + O(n^{\beta})$$

then if

$$\alpha = \log_b a$$

$$F(n) = O(n^{\alpha}) \text{ if } \alpha > \beta$$

$$= O(n^{\beta}) \text{ if } \alpha < \beta$$

$$= O(n^{\alpha}.\log n) \text{ if } \alpha = \beta$$

PROOF. Left to the reader! □

4. STRASSEN'S ALGORITHM [Str 69]

In the light of Winograd's algorithm it would be tempting to conjecture that, while some trade-off between multiplications and additions is possible, the total number of arithmetic operations required is of order n^3 for $n \times n \times n$ product. This is not so! Strassen's simple and astonishing observation is that for multiplying 2×2 matrices only 7 (not 8) multiplications are needed, even if *multiplication of elements is non-commutative*. Using this fact, the block multiplication algorithm described in the last section may be up-graded to one satisfying:

$$P(n) \leq 7.\ P(\lceil n/2 \rceil) + O(n^2),$$

which, by the theorem given above, yields

$$P(n) = O(n^{\log_2 7}) = O(n^{2.81})$$

$$[\log_2 7 \simeq 2.80735492].$$

Recall that P(n) is the *total* number of arithmetic operations (multiplications, additions/subtractions). It should be apparent that with a straightforward implementation of this algorithm on a machine with reasonable properties, the total execution time is also of the stated order.

Strassen's identities.

We assume for simplicity that A, B are $n \times n$ matrices, and that n is even so the matrices can be partitioned into 4 equal quarter-matrices. For

$$\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) . \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right) = \left(\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right),$$

compute:

$$m_1 = (A_{11}+A_{21}).(B_{11}+B_{12})$$
$$m_2 = (A_{12}+A_{22}).(B_{21}+B_{22})$$
$$m_3 = (A_{11}-A_{22}).(B_{11}+B_{22})$$
$$m_4 = A_{11}.(B_{12}-B_{22})$$
$$m_5 = (A_{21}+A_{22}).B_{11}$$
$$m_6 = (A_{11}+A_{12}).B_{22}$$
$$m_7 = A_{22}.(B_{21} - B_{11})$$

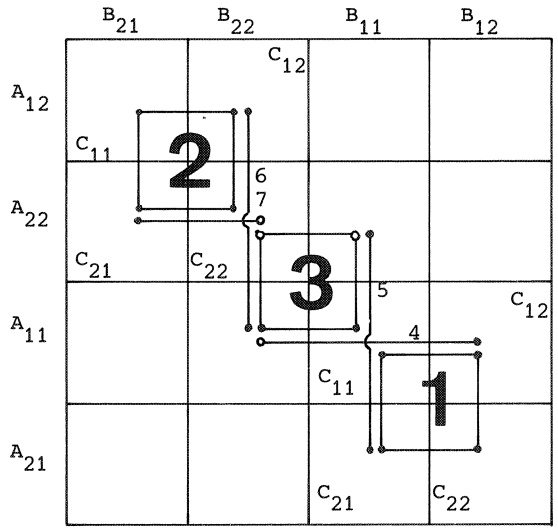
Then

$$C_{11} = m_2 + m_3 - m_6 - m_7$$
$$C_{12} = m_4 + m_6$$
$$C_{21} = m_5 + m_7$$
$$C_{22} = m_1 - m_3 - m_4 - m_5$$

Thus, $P(n) = 7.P(n/2) + 18.S(n/2)$

and so $P(n) = O(n^{\log_2 7})$.

These identities may be conveniently expressed in the form of a diagram, where ●(0) in cell (A_{ij},B_{kl}) represents the term $+(-)A_{ij}.B_{kl}$. The connected groups of circles represent the terms occurring in the respective products. It is now easy to verify the correctness of the identities.



A small improvement may be obtained by applying linear transformations to the above identities and reducing the number of matrix sums required from 18 to 15. Of course this has no effect on the exponent, $\log_2 7$, but merely reduces the arithmetic complexity by a constant factor. The resulting identities and diagram are given below. An amusing feature is that the first two of the seven products are $A_{11}.B_{11}$ and $A_{12}.B_{21}$, which would also be done by the obvious block multiplication.

$$m_1 = A_{11}B_{11}$$

$$m_2 = A_{12}B_{21}$$

$$m_3 = (-A_{11}+A_{21}+A_{22})(B_{11}-B_{12}+B_{22})$$

$$m_4 = (A_{11}-A_{21})(-B_{12}+B_{22})$$

$$m_5 = (A_{21}+A_{22})(-B_{11}+B_{12})$$

$$m_6 = (A_{11}+A_{12}-A_{21}-A_{22})B_{22}$$

$$m_7 = A_{22}(-B_{11}+B_{12}+B_{21}-B_{22})$$

Then

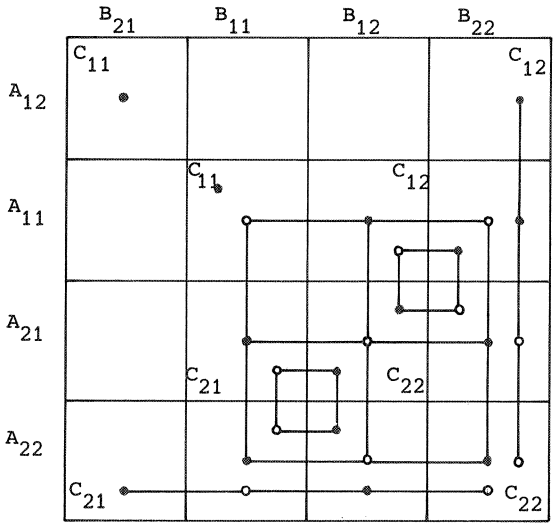
$$C_{11} = m_1 + m_2$$

$$C_{12} = m_1 + m_3 + m_5 + m_6$$

$$C_{21} = m_1 + m_3 + m_4 + m_7$$

$$C_{22} = m_1 + m_3 + m_4 + m_5$$

Note the claimed 15 additions is only achieved by a careful sharing of common terms. PROBERT [Pro 75] has shown that 15 is optimal.



Some related results.

Can the 2 × 2 × 2 product be computed using fewer than 7 multiplications? WINOGRAD [Win 71] shows that, even if multiplication is commutative, 7 is the optimal number. HOPCROFT & MUSINSKI [Hop 73] show that, for any non-commutative ring obtained by adjoining indeterminates to a commutative ring, any algorithm with 7 multiplications for 2 × 2 × 2 product can be got by applying linear transformations to Strassen's algorithm. An example is provided by the two sets of identities given above. They also use a notion of duality of linear forms and of algorithms to show that the minimum number of multiplications required is the same for p × q × r, p × r × q, q × r × p, q × p × r, r × p × q and r × q × p products, and thus depends only on the triple {p, q, r}. This symmetry is implicit in the tensor formulation of the problem used in [Str 72] and [Fid 72]. Using results from [Hop 71] with this result we have that the minimal number for the triple {p, q, 2} is

$$\lceil \tfrac{1}{2}(3pq + \max(p,q)) \rceil ,$$

e.g. 7 for p = q = 2 , and 15 for p = q = 3.

It is clear that any improvement on Strassen's bound using the same kind of recursion has to be based on a larger basic product than 2 × 2 × 2.

If 3 × 3 matrices could be multiplied using only 21 multiplications (non-commutative) then a faster algorithm would be obtained since $\log_3 21 < \log_2 7$. Nothing better than 24 has yet been achieved, but neither has any close lower bound been proved. For 4 × 4 matrices, obviously 48 would need to be achieved. A recursion could be based also on non-square decompositions. The results of HOPCROFT & MUSINSKI show that a result of k multiplications for p × q × r product, yields k^3 for pqr × pqr × pqr product and hence an exponent for n of $3.\log_{pqr} k$.

In an algorithm for the product of matrices of arbitrary shapes and sizes it is very inefficient merely to fill out the matrices with 0's to the next power of two. Halving each dimension and adding one row or column of 0's is more efficient, but the best strategy involves partitioning into varying sizes, using some of the non-square matrix recurrences, and transferring to Winograd's or the classical method for small matrices. It is certainly inefficient to use Strassen's recursion right down to 1 × 1 matrices. Brent [Bre 70] has written and compared programs for Strassen's algorithm and for the other two algorithms both for real and complex numbers.

The idea mentioned in section 2 for sharing some of the non-arithmetic overheads by performing several matrix products in parallel would seem to be useful in an implementation of Strassen's algorithm also. Care must be exercised however to avoid an unacceptable increase in the storage required.

5. REDUCTIONS AND EQUIVALENCES TO MATRIX PRODUCT

In STRASSEN's original paper [Str 69], he also shows how any fast matrix product algorithm yields a correspondingly fast algorithm for matrix inversion and computing determinants. These reductions are based on the following 'block LDU factorization' formula which is easily verified.

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} I & 0 \\ A_{21}A_{11}^{-1} & I \end{pmatrix} \begin{pmatrix} A_{11} & 0 \\ 0 & \Delta \end{pmatrix} \begin{pmatrix} I & A_{11}^{-1}A_{12} \\ 0 & I \end{pmatrix}$$

if A_{11} is non-singular, I is the unit matrix, 0 the zero matrix, and $\Delta = A_{22} - A_{21}A_{11}^{-1}A_{12}$.
So,

$$A^{-1} = \begin{pmatrix} I & -A_{11}^{-1}A_{12} \\ 0 & I \end{pmatrix} \begin{pmatrix} A_{11}^{-1} & 0 \\ 0 & \Delta^{-1} \end{pmatrix} \begin{pmatrix} I & 0 \\ -A_{21}A_{11}^{-1} & I \end{pmatrix}$$

$$= \begin{pmatrix} A_{11}^{-1} + A_{11}^{-1}A_{12}\Delta^{-1}A_{21}A_{11}^{-1} & -A_{11}^{-1}A_{12}\Delta^{-1} \\ -\Delta^{-1}A_{21}A_{11}^{-1} & \Delta^{-1} \end{pmatrix}$$

provided Δ is also non-singular. *Assuming* these non-singularities we have immediately the recurrence relation for I(n), the arithmetic complexity of inverting an n × n matrix, given by

$$I(n) \leq 2I(\lceil n/2 \rceil) + O(P(\lceil n/2 \rceil) + O(n^2)$$

If we assume an algorithm for product giving $P(n) = O(n^\alpha)$, for some $\alpha \geq 2$, the general solution given in section 3 yields

$$I(n) = O(n^\alpha)$$

Similarly, from the LDU factorization, we have

$$\text{Det} \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \text{Det}(A_{11}).\text{Det}(\Delta)$$

If D(n) is the arithmetic complexity for determinants we have the recurrence

$$D(n) \leq 2D(\lceil n/2 \rceil) + I(\lceil n/2 \rceil) + O(P(\lceil n/2 \rceil))$$

and so with the same hypothesis

$$D(n) = O(n^\alpha).$$

The algorithm for inversion uses block LDU factorization recursively and so will fail, even when A is non-singular, whenever "A_{11}" or "Δ" at *any* level of the recursion happens to be singular. In general, a pivotal method, interchanging rows or columns is necessary to obtain non-singular factorizations. Such a method, still achieving the same $O(n^\alpha)$ bound, is given by BUNCH & HOPCROFT [Bun 72].

Is it possible that I(n) is of *lower* order than P(n)? We show directly that this is not so.

$$\begin{pmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{pmatrix}^{-1} = \begin{pmatrix} I & -A & A.B \\ 0 & I & -B \\ 0 & 0 & I \end{pmatrix}$$

as is easily verified. Thus to find the product of two n × n matrices A, B, it is sufficient to invert an appropriately constructed non-singular 3n × 3n matrix. We therefore have

$$P(n) \leq I(3n)$$

Combining this with a previous result we obtain

THEOREM. *For all* $\alpha \geq 2$,

$$P(n) = O(n^{\alpha}) \Leftrightarrow I(n) = O(n^{\alpha}).$$

A similar result for squaring matrices follows from

$$\begin{pmatrix} 0 & A \\ B & 0 \end{pmatrix}^2 = \begin{pmatrix} A.B & 0 \\ 0 & B.A \end{pmatrix} .$$

6. LOWER BOUNDS FOR MATRIX ALGORITHMS OVER FIELDS.

No lower bound, for the arithmetic complexity of n × n × n product, greater than $O(n^2)$ has yet been proved. It is open to conjecture whether this is because this bound is achievable or because we do not have good techniques yet for proving lower bounds. For the simpler problem of 'matrix times vector' product, WINOGRAD [Win 70] proved a powerful theorem which we shall describe in this section.

Let F be an infinite field and $x_1,\ldots,x_n$, indeterminates. We consider straight-line programs (i.e. involving no test instructions) for computing sets of linear forms of the x's. The basic operations used will be +, -, ×, ÷, and the initial input values will be taken from $F \cup \{x_1,\ldots,x_n\}$, so that successive values computed by the programs are elements of $F(x_1,\ldots,x_n)$,

the field of rational functions over F of the indeterminates. We shall establish lower bounds on the number of multiplications/divisions required. Let G be any infinite subfield of F. For the purposes of the theorem to be proved, we think of multiplications and divisions by elements of G as 'free'. We use the phrase 'm/d which is *counted*" for either a multiplication neither of whose arguments is in G or a division whose divisor is not in G.

The main idea of the proof is that the first m/d which is counted in such an algorithm can be eliminated (or made not to be counted) by replacing one of the indeterminates by a linear combination of the other indeterminates with coefficients in G. The resulting algorithm is still a computation of some linear forms of the remaining indeterminates. We find a lower bound on the number of such reductions needed to eliminate all m/d's which are counted in terms of an algebraic property of the linear forms computed.

DEFINITION. A set of vectors $\underline{v}_1,\ldots,\underline{v}_q$ in F^k is G-*independent* if, for all $c_1,\ldots,c_q \in G$, $\sum_{i=1}^{q} c_i.\underline{v}_i \in G^k \Rightarrow c_i = 0$ for all i.

The usefulness of the theorem we now state is much enhanced by the generality allowed in the choice of F and G. The slight extra complication of the proof is thereby justified.

THEOREM. (WINOGRAD) [Win 70]
Let $\psi,\phi_1,\ldots,\phi_n$ *be vectors in* F^k, *let* Φ *be the* k × n *matrix with columns* $\phi_1,\ldots,\phi_n$, *and let* $\underset{\sim}{x}$ *be the column vector* $(x_1,\ldots,x_n)$. *If there is a subset of* q *vectors in* $\{\phi_1,\ldots,\phi_n\}$ *which is* G-*independent then any algorithm over* F *computing* $\Phi.\underset{\sim}{x} + \psi$ *has at least* q m/d's *which are counted.*

Note that the subfield G can be chosen freely, but the larger G is the fewer sets of vectors are G-independent and the fewer m/d's are counted.

PROOF OF THEOREM. Suppose the conditions of the theorem are satisfied but that there is an algorithm α with only q - 1 m/d's which are counted. If q = 1 the contradiction is immediate since with *no* m/d's counted only linear combinations of elements of $F \cup \{x_1,\ldots,x_n\}$ with coefficients in G can be computed and so all the elements of Φ are in G and it has not even a set of *one* G-independent column.

If q > 1, then consider the first m/d which is counted. If both argu-

ments are in F then the result is also in F and there is no need for that operation since the result could have been taken as an input. Otherwise at least one argument, the divisor if the operation is a division, is in $F(x_1,\ldots,x_n) \neg F$, and since it has been computed without any m/d which counts, it must be of the form

$$f + \sum_{i=1}^{n} c_i.x_i \quad \text{where } f \in F \ , \ c_i \in G \ \text{ for all } i$$

and not all the c_i are zero. Without loss of generality we can assume that $c_n \neq 0$ and since multiplication by elements of G is free we may as well assume that $c_n = -1$. If we precede α by a computation of $\delta = f + g + \sum_{i=1}^{n-1} c_i.x_i$ for some $g \in G$, and then replace any occurrence of x_n as an input by δ, the m/d we have been considering has g as its argument and is no longer counted. Since the computation of δ requires no m/d which is counted, we have a new algorithm α' with at most $q - 2$ m/d's which are counted. The element g is chosen with the sole requirement that no division by zero will occur in α'. This is possible since G is infinite and there are only a finite number of "bad" values to be avoided. α' has indeterminates $x_1,\ldots,x_{n-1}$ and computes $\Phi'.\underset{\sim}{x}' + \psi'$ where $\underset{\sim}{x}' = (x_1,\ldots,x_{n-1})$

$$\phi_j' = \phi_j + c_j \phi_n \quad \text{for } j = 1,\ldots,n-1,$$

and $$\psi' = \psi + \phi_n(f+g).$$

We claim that Φ' has a set of $q - 1$ G-independent columns. Suppose not and define $\phi_n' = \underset{\sim}{0}$. Let $Q = \{\phi_{i_1},\ldots,\phi_{i_q}\}$ be a set of q independent columns of Φ. By the supposition $\exists d_1,\ldots,d_{q-1} \in G$ so that $G^k \ni \sum_{j=1}^{q-1} d_j.\phi'_{i_j} = \sum_{j=1}^{q-1} d_j.\phi_{i_j} + k_1\phi_n$ for some $k_1 \in G$, by the definition of the ϕ's. Without loss of generality we may suppose $d_1 \neq 0$. Similarly $\exists e_2,\ldots,e_q \in G$ such that that

$$G^k \ni \sum_{j=2}^{q} e_j.\phi'_{i_j} = \sum_{j=2}^{q} e_j.\phi_{i_j} + k_2\phi_n$$

for some $k_2 \in G$. Eliminating the explicit ϕ_n terms between these two linear combinations produces a contradiction to the G-independence of Q, since the coefficients k_1, k_2 must both be non-zero and so the coefficient of ϕ_{i_1} in the final linear combination is non-zero. An inductive argument now

proves the theorem. □

To illustrate the power of the theorem, the first corollary shows that Horner's rule for evaluating polynomials uses the minimal number of m/d's.

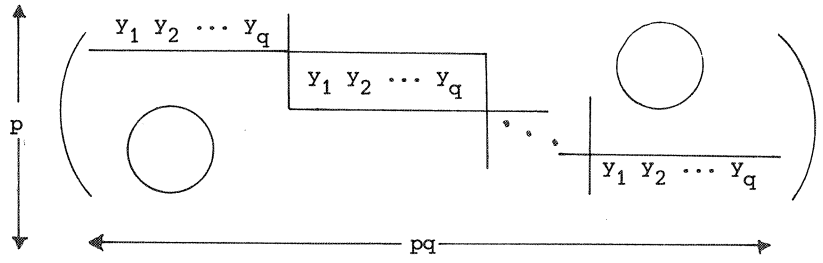
COROLLARY 1. *For any infinite field* G, *any algorithm computing* $\sum_{i=0}^{n} x_i.y^i$ *from* $G(y) \cup \{x_0,\ldots,x_n\}$ *requires at least* n m/d's.

PROOF. Let F = G(y) and take Φ to be the 1 × (n+1) matrix $(1\ y\ y^2 \ldots y^n)$. Since the last n 'columns' are G-independent, the theorem can be applied with q = n. □

The main result we require for 'matrix × vector' products is proved by a pleasing interchange of rôles.

COROLLARY 2. *Let* X *be a* p × q *matrix and* $\underset{\sim}{y} = (y_1,\ldots,y_q)$ *a column vector. Any algorithm for computing* $X.\underset{\sim}{y}$ *requires at least* pq m/d's *even when operations on* $\underset{\sim}{y}$ *alone are not counted.*

PROOF. Let $F = G(y_1,\ldots,y_q)$ so that 'operations on $\underset{\sim}{y}$ alone' produce values in F and need not be counted. Define Φ to be the p × pq matrix



and $\hat{\underset{\sim}{x}}$ to be the column vector $(x_{11},\ldots,x_{1q},x_{21},\ldots x_{2q},\ldots x_{pq})$. Then $\Phi.\hat{\underset{\sim}{x}} = x.\underset{\sim}{y}$ and the set of all pq columns of Φ is G-independent. □

Note. In the proof of both corollaries we assumed that multiplication was commutatuve even over indeterminates. This was not necessary. We only need to prove a symmetric form of the theorem for computations of $\underset{\sim}{x}.\Phi + \psi$.

Of course it cannot be deduced from corollary 2 that the product of a p × q matrix X with a q × r matrix Y requires pq.r m/d's since the r columns of Y need not be multiplied independently, and indeed Strassen's algorithm beats this bound.

FIDUCCIA [Fid 71,72] and WINOGRAD [Win 70] have proved several interest-

ing extensions of the above theorem. In particular a proof of the following appears in [Fid 72].

THEOREM (FIDUCCIA).
Under the conditions of Winograd's theorem (above), if Φ *has an* r × c *submatrix* Θ *and there are no non-trivial vectors* $\alpha \in G^r$, $\beta \in G^c$ *such that* $\alpha.\Theta.\beta \in G$, *then at least* r + c - 1 m/d's *are required to compute* $\Phi.\underset{\sim}{x}$.

An immediate corollary of this theorem is that at least three real multiplications are needed to multiply two complex numbers represented by their real and imaginary parts. We shall not give a proof of the theorem here, but shall give an idea of the technique by proving the corollary directly. This time we remove the *last* m/d which is counted, by eliminating its occurrences from the linear forms. This particular result is proved also in [Win 71].

THEOREM. *To compute* xu - yv *and* xv + yu *from* $\mathbb{R} \cup \{x,y,u,v\}$ *requires at least* 3 m/d's, *even if* m/d's *by elements of* $\mathbb{R}$ *are not counted.*

PROOF. Suppose there is an algorithm computing these forms using only 2 m/d's. Let μ_1, μ_2 be the results of the first and second m/d respectively. Then since μ_2 can occur at most linearly in xu - yv and xv + yu, we can eliminate μ_2 to get a non-trivial linear combination

$$\lambda_1(xu-yv) + \lambda_2(xv+yu) \quad , \quad \lambda_1,\lambda_2 \in \mathbb{R},$$

which can be computed with only one m/d which is counted. This can be written as a matrix times vector in the form

$$\begin{pmatrix} \lambda_1 x+\lambda_2 y & \lambda_2 x-\lambda_1 y \end{pmatrix}.\begin{pmatrix} u \\ v \end{pmatrix}$$

Winograd's theorem can now be applied, with $F = \mathbb{R}(x,y)$, $G = \mathbb{R}$. Since $\{\lambda_1 x+\lambda_2 y, \lambda_2 x-\lambda_1 y\}$ are $\mathbb{R}$-independent for any $\lambda_1,\lambda_2 \in \mathbb{R}$, not both zero, the theorem yields a contradiction. □

We saw in section 2 that three multiplications are sufficient, so this result is the best possible. A further corollary of Fiduccia's theorem is that at least seven multiplications/divisions are required to compute the product of two quaternions presented in the usual form.

7. MATRIX PRODUCT OVER OTHER STRUCTURES

We have so far only considered matrices over rings. The operation of matrix product is readily generalized however to other structures. In this section we shall consider some structures in which this operation has useful applications. Given the binary operators $\otimes$ and $\oplus$ over some structure S, the product of two matrices A,B, over S is given by

$$c_{ij} = \underset{k}{} a_{ik} \otimes b_{kj} \qquad \text{for all } i, j.$$

The derived operator (analogous to $\sum$ being derived from +) is well-defined if $\oplus$ is associative. In all the structures and pairs of operators we consider, $\oplus$ is also commutative and $\otimes$ distributes over $\oplus$ (i.e. $a\otimes(b\oplus c) = (a\otimes b)\oplus(a\otimes c)$.) It is easy to verify that these properties on the basic operators induce the same properties for matrix product and matrix sum, where the sum C = A $\oplus$ B is given by

$$c_{ij} = a_{ij} \oplus b_{ij} \qquad \text{for all } i, j.$$

We usually denote matrix product by A.B for brevity, but use the fuller form $A \begin{bmatrix} \otimes \\ \oplus \end{bmatrix} B$ when it is necessary to mention the operators explicitly. In all but one of the structures considered $\otimes$ is also associative and induces the same property for the matrix product.

Examples.

	$\otimes$	$\oplus$	Domain
1.	$\times$	+	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{Z}$, $\mathbb{Z}_k$ and other *rings*.
2.	$\wedge$	$\vee$	{true, false} - Boolean algebra
3.	$\vee$	$\wedge$	{true, false}
4.	$\wedge$	$\not\equiv$	
5.	$\equiv$	$\wedge$	
6.	+	minimum	$\mathbb{R}_{\geq 0} \cup \{\infty\}$
7.	min	max	$\mathbb{R} \cup \{+\infty, -\infty\}$
8.	$\cdot$ (concatenation)	$\cup$	subsets of $\sum^*$, 'languages' - regular algebra.

A further example concerned with context-free grammars will be considered in section 11.

Using the definition of matrix product directly, one may compute an $I \times K \times J$ product over any of these structures using IJK $\otimes$'s and $IJ(K-1)$ $\oplus$'s. The identities of STRASSEN (or WINOGRAD) cannot be used unless there is an inverse operator to $\oplus$, a 'minus', but it is clear that Strassen's algorithm works over an arbitrary ring. Winograd's algorithm also requires commutativity of $\otimes$.

The first product, $\left[\begin{smallmatrix}\wedge\\ \vee\end{smallmatrix}\right]$, over the 2-element Boolean algebra will be called *Boolean product*. The $\left[\begin{smallmatrix}\vee\\ \wedge\end{smallmatrix}\right]$ product is a dual operation to this. If $\neg A$ denotes the matrix got by complementing each entry of A, it can be seen that

$$A \left[\begin{smallmatrix}\vee\\ \wedge\end{smallmatrix}\right] B = \neg((\neg A) \left[\begin{smallmatrix}\wedge\\ \vee\end{smallmatrix}\right] (\neg B))$$

To avoid confusion this product will not be mentioned again. We shall denote *true* and *false* by 1 and 0 respectively. In this notation, the operations of $\wedge$ and $\times$ are identical over the domain $\{0,1\}$. Similarly $\not\equiv$ and $+_{\mathrm{mod}2}$ are identical. Hence the $\left[\begin{smallmatrix}\wedge\\ \not\equiv\end{smallmatrix}\right]$ product is isomorphic to the product over $\mathbb{Z}_2$, the ring of integers modulo 2. Strassen's algorithm could be used.

The $\left[\begin{smallmatrix}\equiv\\ \wedge\end{smallmatrix}\right]$ product can be related to a string matching problem. Given two binary n-vectors $\underline{a}$, $\underline{b}$, regarding $\underline{a}$ as a row vector and $\underline{b}$ as a column vector,

$$\underline{a} \left[\begin{smallmatrix}\equiv\\ \wedge\end{smallmatrix}\right] \underline{b} = 1 \text{ if } \underline{a}, \underline{b} \text{ are identical}$$
$$= 0 \text{ otherwise.}$$

Thus, given two sets $\mathcal{A}$, $\mathcal{B}$, of binary n-vectors, all matching pairs of vectors from $\mathcal{A}$ and $\mathcal{B}$ can be found by computing $A \left[\begin{smallmatrix}\equiv\\ \wedge\end{smallmatrix}\right] B$ where $\mathcal{A}$ is the set of rows of A, and $\mathcal{B}$ is the set of columns of B.
The computation of $\left[\begin{smallmatrix}\equiv\\ \wedge\end{smallmatrix}\right]$ product can be reduced to two $\left[\begin{smallmatrix}\wedge\\ \vee\end{smallmatrix}\right]$ products and low-order operations by the identity

$$A \left[\begin{smallmatrix}\equiv\\ \wedge\end{smallmatrix}\right] B = \neg((A \left[\begin{smallmatrix}\wedge\\ \vee\end{smallmatrix}\right] \neg B) \vee (\neg A \left[\begin{smallmatrix}\wedge\\ \vee\end{smallmatrix}\right] B)).$$

A *unit* is an element $\underline{e}$ such that

$$\underline{e} \otimes x = x = x \otimes \underline{e} \qquad \text{for all } x.$$

A *zero* is an element $\underline{z}$ such that

$$\underline{z} \otimes x = \underline{z} = x \otimes \underline{z} \text{ and } \underline{z} \oplus x = x = x \oplus \underline{z} \qquad \text{for all } x.$$

All the structures given, except (5), have a unit and a zero.

	$\underline{e}$	$\underline{z}$	
1.	1	0	
2.	1	0	
3.	0	1	
4.	1	0	
5.	1		no element has the $\otimes$ property of $\underline{z}$
6.	0	∞	
7.	∞	$-\infty$	
8.	$\{\lambda\}$	$\emptyset$	i.e. the set containing only the empty string, and the empty set respectively.

A further important property, satisfied in only (2), (3), (6) and (7) above, is that

$$\underline{e} \oplus x = \underline{e} \qquad \text{for all } x.$$

This *absortive* property of the unit will be used later. Structures with a unit and zero have a *unit matrix* $\underline{I}$, and a *zero matrix* $\underline{0}$, with appropriate properties, given by

$$I_{ij} = \underline{e} \text{ if } i = j \quad = \underline{z} \text{ if } i \neq j \qquad 0_{ij} = \underline{z} \qquad \text{for all } i, j.$$

8. BOOLEAN MATRICES

Boolean matrices have important computational applications as representations of binary relations on finite sets, or, equivalently, finite directed graphs, where

A_{ij} = 1 if i is related to j ((i,j) is an arc)

= 0 otherwise.

The $\begin{bmatrix} \wedge \\ \vee \end{bmatrix}$ product corresponds to composition of relations. Provided $\otimes$ is associative, as it is for Boolean product, we can define *powers* of a square matrix A by

$$A^0 = I$$

$$A^{n+1} = A.A^n \qquad \text{for } n \geq 0.$$

If A represents a directed graph, then

$$(A^2)_{ij} = 1 \Leftrightarrow \exists \text{ path of length 2 from i to j}$$

and more generally, for all $k \geq 0$,

$$(A^k)_{ij} = 1 \Leftrightarrow \exists \text{ path of length k from i to j,}$$

where by convention there is a 'path of length 0' from i to i, for all i. Thus the connectedness relation ("there is a path from i to j") is given by the matrix

$$A^* = I \vee A \vee A^2 \vee A^3 \quad \ldots$$

For relations, A^* is the (reflexive and) *transitive closure* of the relation A.

Since there is a path from i to j in an n-node directed graph if and only if there is such a path of length less than n, we have for an $n \times n$ matrix A,

$$A^* = I \vee A \vee A^2 \vee \ldots \vee A^{n-1}$$

We can further show that

$$A^* = (I \vee A)^{n-1} = (I \vee A)^m \qquad \text{for all } m \geq n-1,$$

so that one fairly efficient way to compute A^* is to form $I \vee A$ and then

square the result $\lceil \log_2(n-1) \rceil$ times in succession. There are even better methods however.

LEMMA 1. $(A \vee B)^* = (A^*.B)^*.A^*$

'PROOF'. $(A \vee B)^* = I \vee (A \vee B) \vee (A \vee B)^2 \vee \ldots$

= {all finite products of A's & B's} arranged by length

= {all finite products of A's & B's} arranged by number of B's

$= A^* \vee A^*.B.A^* \vee A^*.B.A^*.B.A^* \vee \ldots$

$= (A^*.B)^*.A^*$ □

LEMMA 2. *If* A *is the partitioned matrix* $\left(\begin{array}{c|c} A_{11} & 0 \\ \hline A_{21} & 0 \end{array}\right)$ *where* A_{11} *is square then*

$$A^* = \left(\begin{array}{c|c} A_{11}^* & 0 \\ \hline A_{21}A_{11}^* & I \end{array}\right)$$

'PROOF'.

$$A^0 = \begin{pmatrix} I & 0 \\ 0 & I \end{pmatrix}, \quad A^1 = \begin{pmatrix} A_{11} & 0 \\ A_{21} & 0 \end{pmatrix}, \quad A^2 = \begin{pmatrix} A_{11}^2 & 0 \\ A_{21}A_{11} & 0 \end{pmatrix}, \quad A^3 = \begin{pmatrix} A_{11}^3 & 0 \\ A_{21}A_{11}^2 & 0 \end{pmatrix},$$

$\ldots$ □

We can now derive an expression for the closure of a partitioned matrix in terms of the closures of sub-matrices.

THEOREM. *If* $A = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right)$ *where* A_{11}, A_{22} *are square then*

$$A^* = \left(\begin{array}{c|c} A_{11}^* \vee A_{11}^* A_{12} E^* A_{21} A_{11}^* & A_{11}^* A_{12} E^* \\ \hline E^* A_{21} A_{11}^* & E^* \end{array}\right)$$

where $E = A_{21}A_{11}^*A_{12} \vee A_{22}$.

PROOF.

$$A^* = \left(\begin{pmatrix} A_{11} & 0 \\ A_{21} & 0 \end{pmatrix} \vee \begin{pmatrix} 0 & A_{12} \\ 0 & A_{22} \end{pmatrix}\right)^*$$

$$= \left(\begin{pmatrix} A_{11} & 0 \\ A_{21} & 0 \end{pmatrix}^* \begin{pmatrix} 0 & A_{12} \\ 0 & A_{22} \end{pmatrix}\right)^* \begin{pmatrix} A_{11} & 0 \\ A_{21} & 0 \end{pmatrix}^* \qquad \text{by Lemma 1}$$

$$= \left(\begin{pmatrix} A_{11}^* & 0 \\ A_{21}A_{11}^* & I \end{pmatrix} \begin{pmatrix} 0 & A_{12} \\ 0 & A_{22} \end{pmatrix}\right)^* \begin{pmatrix} A_{11}^* & 0 \\ A_{21}A_{11}^* & I \end{pmatrix} \qquad \text{by Lemma 2}$$

$$= \begin{pmatrix} 0 & A_{11}^*A_{12} \\ 0 & E \end{pmatrix}^* \begin{pmatrix} A_{11}^* & 0 \\ A_{21}A_{11}^* & I \end{pmatrix}$$

$$= \begin{pmatrix} I & A_{11}^*A_{12}E^* \\ 0 & E^* \end{pmatrix} \begin{pmatrix} A_{11}^* & 0 \\ A_{21}A_{11}^* & I \end{pmatrix} \qquad \text{by a symmetric version of Lemma 2}$$

= 'result claimed'. □

The formula given in the theorem, taking A_{11} to be an $\lceil n/2 \rceil \times \lceil n/2 \rceil$ submatrix, together with the fact that the closure of a 1×1 matrix is 1, yields a recursive algorithm for the transitive closure of Boolean matrices. If $C(n)$, $P(n)$, are the numbers of Boolean operations required for the transitive closure, product, respectively, of $n \times n$ matrices, we get the recurrence relation

$$C(n) \leq 2C(\lceil n/2 \rceil) + 0(P(\lceil n/2 \rceil)) + 0(n^2).$$

Hence for all $\alpha \geq 2$, if $P(n) = 0(n^\alpha)$ then $C(n) = 0(n^\alpha)$, see also [Mun 71] for a different method.
The converse is trivial since

$$\begin{pmatrix} 0 & A & 0 \\ 0 & 0 & B \\ 0 & 0 & 0 \end{pmatrix}^* = \begin{pmatrix} I & A & A.B \\ 0 & I & B \\ 0 & 0 & I \end{pmatrix}$$

so that if $C(n) = 0(n^\alpha)$, then $P(n) \leq C(3n) = 0(n^\alpha)$. So we have shown that transitive closure and product are of the same order of complexity. Using

the naive algorithm for Boolean product, we get an $O(n^3)$ algorithm for the transitive closure of n × n matrices.

An interesting observation due to FURMAN [Fur 70], and M.FISCHER & MEYER [Fis 71] is that an indirect application of Strassen's algorithm to Boolean product is possible, although the structure concerned is not a ring. With the notations 0,1, for *false* and *true* respectively, ∧ and × are identifiable for 0,1-valued arguments while ∨ and + differ in that $1 \vee 1 = 1$ but $1 + 1 = 2$. However it is apparent that if we multiply together two 0,1 matrices over the integers and then in the product replace each *non-zero* entry by 1, we get exactly the Boolean product of the matrices. If the matrices are of size n × n we have a product algorithm which uses $O(n^{\log_2 7})$ *arithmetic* operations, and hence a closure algorithm of the same order. This is not quite the result we want since for a Boolean product we should insist on *Boolean* basic operations. For n × n matrices, no entry in the product can exceed n, though intermediate values in the recursive calls of Strassen's algorithm may perhaps be considerably larger. If the arithmetic is done in $\mathbb{Z}_{n+1}$ then the true integer result must be uniquely determined since it is known to lie in the range [0,n]. We can represent elements in $\mathbb{Z}_{n+1}$ by binary vectors of length $\lceil \log_2(n+1) \rceil$ and perform ring addition in O(log n) Boolean operations (by simulating ordinary digital circuitry). Ring multiplication done in the conventional way takes $O((\log n)^2)$ operations but this could be improved to $O(\log n.\text{loglog } n.\text{logloglog } n)$ using the methods in [Sch 71]. We have shown therefore

THEOREM. *Using the operations* ∧, ∨, ¬ *(or any other complete basis), for any* $\varepsilon > 0$, n × n × n *Boolean product and* n × n *transitive closure can be computed in* $O(n^{\log_2 7}.(\log n)^{1+\varepsilon})$ *basic operations.*

It is of interest that Warshall's $O(n^3)$ algorithm for transitive closure [War 62] corresponds approximately to Jordan elmination for matrices over fields.

Symmetric Boolean Matrices

This important special case of Boolean matrices correponds to undirected graphs and symmetric relations. It is simple to show that for almost any structure the product for symmetric matrices is of the same order

of complexity as the general product, since the square of the symmetric matrix

$$\begin{pmatrix} 0 & A & 0 \\ A^T & 0 & B \\ 0 & B^T & 0 \end{pmatrix} ,$$

where M^T is the transpose of M, contains the product A.B as a sub-matrix. The proof, in this section, above, showing product and closure to be of the same order of complexity, breaks down if the extra condition of symmetricity is imposed. Indeed there is a rather simple $O(n^2)$ algorithm for the transitive closure of symmetric Boolean matrices.

Given an n × n symmetric Boolean matrix A, define the function

$$\text{Next}(i) = \text{least } j > i \text{ such that } A_{ij} = 1$$
$$= 0 \text{ if none such .}$$

The following informal program computes A^*.

```
for i = 1(1)n
    [ j := next(i)
      if j ≠ 0 then [ Row(j) := Row(j) ∨ Row(i) ]
               else [ A_ii := 1
                      for k = 1(1) i-1
                          [ if A_ki = 1 then Row(k) := Row(i) ] ]
```

The reader is invited to carry out the non-trivial proof that this algorithm is correct. The $0(n^2)$ bound can be shown as follows. The only parts which require attention are the row operations, which take 0(n) basic operations each time. The ∨ operation is only executed at most n times. The row copying instruction is also executed at most n times since each row is copied *into* at most once.

9. TRANSITIVE CLOSURE IN OTHER STRUCTURES

Most of the results and formulae of Section 8 carry over to other structures we have described. The general definition of (reflexive and) *transitive closure* is of course

$$A^* = \sum_{m \geq 0} A^m \text{ , provided } A^0 = I \text{ is defined.}$$

For structures 2, 3, 6, 7, where the unit has the absorbtive property, we can prove the identities

$$A^* = \sum_{0 \leq m < n} A^m = (I \oplus A)^m \qquad \text{for any } m \geq n - 1.$$

For those of the structures (all except (5)) with a unit and a zero, the partitioned matrix closure formula holds whenever both sides are well-defined. The problem here is that the closure of 1×1 matrices, i.e. single elements may not be defined. With an absorbtive unit, $x^* = \underline{e}$ for all x. For (8), x^* always exists as a set of strings and is denoted by 'x^*'. In rings such as in (1) and (4), $\underline{e}^* = \underline{e} + \underline{e} + \underline{e} + \ldots$ is undefined. In a field, for any $x \neq 1$,

$$x^* = 1 + x + x^2 + \ldots$$

can be taken to be $(1-x)^{-1}$. It is the strong similarity between A^* and $(I-A)^{-1}$ for matrices over a field which accounts for the correspondence between the matrix closure recurrence and the block LDU factorization which has undoubtedly struck the reader. The principal difference between the form of algorithms for matrix inversion over fields and for transitive closures over other structures is because of the need in the former case to avoid singular sub-matrices and elements. The usual JORDAN elimination algorithm for matrix inversion therefore uses pivoting, while the algorithms of WARSHALL [War 62] and FLOYD [Flo 62] are quite analogous over structures (2) and (6) respectively except that they do not need to pivot. The structures (6), (7), (8), can all be regarded as generalizations of (2), and they are all isomorphic to (2) when their domains are restricted to just the zero and the unit. Whereas matrices in (2) correspond to directed graphs, in these three structures we have directed graphs labelled with

elements of the structure. Associated with any path in the graph is, in (6) the sum of the labels of its edges, in (7) the minimum of its labels and in (8) the concatenation of its labels.

$\left[\begin{smallmatrix}+\\ \min\end{smallmatrix}\right]$ product. Regarding the lables as direct distances between pairs of nodes, the transitive closure is the matrix giving the *shortest path* between all pairs of nodes. Floyd's algorithm [Flo 62] is a simple, 'in place', algorithm for this with a complexity of $0(n^3)$. The recursive closure formula provides a family of different $0(n^3)$ methods for the reflexive and transitive closure. An algorithm which uses $0(n^3)$ comparisons and $0(n^5/2)$ addition/*subtractions* is given in [Hof 72].

$\left[\begin{smallmatrix}\min\\ \max\end{smallmatrix}\right]$ product. This structure is somewhat similar to the preceding one. An example of an application is the problem of transporting a wide load between points of a transportation network in which there are bridges of varying widths. The maximum width for a given pair of points is given by the maximum over all paths of the minimum width along each path.

$\left[\begin{smallmatrix}\cdot\\ \cup\end{smallmatrix}\right]$ product. An application is in the theory of finite automata. A finite automaton is a finite directed graph labelled with subsets of a finite alphabet $\sum$. The *language accepted* by an automaton is the set of strings over $\sum$ which label all possible paths from an initial node to one of a set of final nodes. This is of course a union of entries of the transitive closure of the matrix describing the automaton. A *regular set* can be defined as a set of strings formed from subsets of a finite alphabet using the operations of union, concatenation and transitive closure of sets. One half of Kleene's theorem states that the language accepted by a finite automaton is a regular set. An easy inductive proof of this follows from the recursive formula for matrix closure. This was shown by CONWAY [Con 71] from whom I first heard of this useful formula. In this structure there seems little to be said about the "computational complexity" unless it would be related to the expression length of the representation obtained for regular languages.

An axiomatic treatment of the generalization of Floyd's closure algorithm to other algebraic structures can be found in [Bru 72]. Many examples of such structures are given by [Pai 70], and a useful survey of this area appears in [Bru 74].

10. LOWER BOUNDS FOR BOOLEAN PRODUCT OVER A MONOTONE BASIS

THEOREM. *Any algorithm computing the Boolean product of an* I × K *matrix and an* K × J *matrix using only binary* ∧ *and* ∨ *as basic operations requires at least* IJK ∧'s *and* IJ(K-1) ∨'s. *Furthermore any algorithm achieving both these lower bounds is equivalent, using only the associativity of* ∨ *and the commutativity of* ∧ *and* ∨, *to the naive algorithm.*

We shall give here just an outline of the IJK lower bound for ∧'s to illustrate the proof methods. (A full proof of the theorem is given in [Pat 75].)

The proof is by induction on K. The result is trivial for K = 0. Suppose now K > 0 and the result has been proved already for K - 1. The inputs to the algorithm are the elements of the two matrices, $a_{11},\ldots,a_{IK}$, $b_{11},\ldots,b_{KJ}$ say. The final results are the values

$$c_{ij} = \bigvee_{1\le k\le K} a_{ik} \wedge b_{kj} \qquad \text{for } i = 1,\ldots,I \text{ and } j = 1,\ldots,J.$$

We consider 'straight-line' programs with operations ∧ and ∨. We use 0,1 for *false* and *true*, and regard Boolean expressions as sets, identifying ∧, ∨, with intersection and union respectively. Thus we could write

$$0 \subseteq a_{11} \wedge b_{11} \subseteq a_{11} \subseteq a_{11} \vee b_{11} \subseteq 1.$$

Suppose we are considering an algorithm for I × K × J product with the minimum number of ∧ and ∨ operations. We refer to initial inputs and the values computed at each step as *issues*.

LEMMA 1. *If for some issue* s, *and for some* i,i', i ≠ i', $a_{i1} \vee a_{i'1} \subseteq s$ *then* s *can be replaced by* 1 *without affecting the outputs of the algorithm. The same is true for* $b_{1j} \vee b_{1j'}$ (j≠j') *and* $a_{i1} \vee b_{1j}$.

Of course a conclusion that an issue can be changed to 1 contradicts the minimality of the algorithm.

For each i,j, define the predicate Q_{ij} on issues by

$$Q_{ij}(s) \Leftrightarrow a_{i1} \wedge b_{1j} \subseteq s \quad \text{but} \quad a_{i1} \not\subseteq s \quad \text{and} \quad b_{1j} \not\subseteq s.$$

An *initial occurrence* of Q_{ij} is an instruction for which the result satisfies Q_{ij} but neither of the arguments does. The set of initial occurrences of Q_{ij} is denoted by $I(Q_{ij})$.

LEMMA 2. *Any instruction in* $I(Q_{ij})$ *must be an* $\wedge$ *with arguments* x,y, *satisfying*: $a_{i1} \leq x$, $b_{1j} \leq y$.

Suppose $I(Q_{ij}) \cap I(Q_{i'j'}) \neq \emptyset$ and $(i,j) \neq (i',j')$, then Lemmas 1 and 2 imply together that one of the arguments of any instruction in the intersection can be replaced by 1, contradicting the minimality assumption. Also each $I(Q_{ij})$ is non-empty since no input satisfies Q_{ij} but c_{ij} does satisfy Q_{ij}.

If the valuation $a_{i1} = 1$ for all i, $b_{1j} = 0$ for all j, is imposed on the inputs, then all the $\wedge$ instructions in all the $I(Q_{ij})$ can be eliminated because of the "$a_{i1} \leq x$" condition of Lemma 2. Thus we get a new algorithm with at least IJ fewer $\wedge$-instructions, and the (i,j) output is now clearly $\bigvee_{k=2}^{K} a_{ik} \wedge b_{kj}$. The new algorithm therefore computes an I × (K-1) × J product. By the inductive hypothesis this algorithm still has at least I.J.(K-1) $\wedge$-instructions, so the original algorithm had at least IJK. The lower bound for $\vee$-instructions is proved similarly.

As we remarked in the previous section, $\boxed{\begin{matrix}+\\ \min\end{matrix}}$ product and $\boxed{\begin{matrix}\min\\ \max\end{matrix}}$ product are isomorphic to Boolean product when the domain is restricted, therefore the results of the theorem carry over to analogous results in those structures. This theorem for Boolean product implies an $0(n^3)$ lower bound for Boolean transitive closure. The results are of particular interest in juxtaposition with the fast algorithms derived from Strassen's algorithm which are possible when complementation is permitted, and with the $\boxed{\begin{matrix}+\\ \min\end{matrix}}$ product algorithm using subtractions given in [Hof 72].

11. CONTEXT-FREE LANGUAGE RECOGNITION

In this section I shall introduce the sub-cubic time context-free language recognition algorithm recently discovered by VALIANT [Val 74]. For this purpose I shall simplify the presentation of context-free grammars by taking them to be in Chomsky normal form and by not distinguishing between terminal and non-terminal symbols. This involves no real loss of generality. A *context-free grammar* (cfg) is a finite alphabet $\sum = \{A_1,\ldots,A_q\}$

with a finite set of *productions* P, each of the form

$$A_i \to A_j A_k \qquad \text{for some } i,j,k.$$

For any finite strings w_1, w_2 over $\sum$ and any production $A_i \to A_j A_k$ in P, we write

$$w_1 A_i w_2 \Rightarrow w_1 A_j A_k w_2.$$

If $\overset{*}{\Rightarrow}$ is the (reflexive and) transitive closure of $\Rightarrow$, we define for each $A_i \in \sum$,

$$L_{A_i} = \{w \mid A_i \overset{*}{\Rightarrow} w\}$$

i.e. the set of strings over $\sum$ derivable from A_i. The *recognition problem* considered here is the following: "*Given a* cfg G *and a string* w, *is* $w \in L_{A_1}$?". (In the usual terminology we are recognizing arbitrary sentential forms.)

We define $\otimes$ and $\oplus$ for the *context-free matrix product with respect to a* cfg G. The domain is all subsets of the alphabet $\sum$, and $\oplus$ is set union. $\otimes$ is defined by

$$S_1 \otimes S_2 = \{A_i \mid \exists A_j \in S_1,\ A_k \in S_2 \text{ and } A_i \to A_j A_k\}$$

which is roughly the "inverse of the production relation" applied to the concatenation of S_1, S_2. Thus $\emptyset$, the empty set is the 'zero' and there is not necessarily any unit. If required, a natural unit could be adjoined by augmenting $\sum$ with λ, the empty string, and adding the productions $A \to A$ for all $A \in \sum \cup \{\lambda\}$, so that $\{\lambda\}$ becomes a unit. We do not do this here.

An unusual feature of this $\otimes$ is that it is not associative, and so the corresponding matrix product is not. We must therefore give a new definition of the transitive closure since matrix powers are not uniquely defined. We define

$$A^{(1)} = A,$$

$$A^{(n)} = \bigoplus_{0<i<n} A^{(i)}.A^{(n-i)} \qquad \text{for } n > 1,$$

and finally,

$$A^+ = \bigcup_{n>0} A^{(n)}$$

is the (non-reflexive) *transitive closure*. The previous closure algorithms described here all rely on associativity and so are not applicable in this case.

It can be proved from the definitions of $\otimes$ and $\overset{*}{\Rightarrow}$ that for all $X,X_1,\ldots,X_k \in \sum$, $(k\geq 1)$ $X \overset{*}{\Rightarrow} X_1\ldots X_k$ if and only if a product of the sets $\{X_1\}\otimes\ldots\otimes\{X_k\}$, *associated in some way*, contains X. If $w = X_1\ldots X_{n-1} \in \sum^{n-1}$, let M(w) be the $n \times n$ matrix with

$$M(w)_{i,i+1} = X_i \qquad \text{for } i = 1,\ldots,n-1,$$

and all other entries $\emptyset$, i.e. w is written in the diagonal immediately above the main diagonal.

THEOREM 1. *For all* $A_i \in \sum$ *and* u,v, $(1\leq u<v\leq n)$

$$A_i \overset{*}{\Rightarrow} X_u\ldots X_{v-1} \Leftrightarrow A_i \in (M(w))^+_{uv} .$$

PROOF. By induction on $v - u$. □

COROLLARY. *The recognition problem can be solved by computing* $(M(w))^+$ *and checking whether* A_1 *is a member of the* $(1,n)$ *entry.*

We also have:

THEOREM 2. *For any* cfg, *the corresponding* cf *matrix product requires the same number of operations to within a constant factor as for Boolean product.*

PROOF. Immediate from the next two Lemmas. □

LEMMA 1. *If the* cfg *has only the single production* $A \to AA$, *then the* cf *product is equivalent to Boolean product.*

LEMMA 2. *If* G_1, G_2, G_3 *are* cfg's *over* $\sum$ *and have sets of productions* P_1, P_2 *and* $P_1 \cup P_2$ *respectively, then for matrices* C, D,

$$C \otimes_3 D = (C\otimes_1 D) \cup (C\otimes_2 D)$$

where $\otimes_i$ *is the product operation for* G_i.

VALIANT [Val 74] describes a very ingenious, simple, recursive algorithm for computing the transitive closure of triangular matrices. His main theorem implies:

THEOREM 3. *For context-free product, if* $P(n) = O(n^{\alpha})$ *for some* $\alpha > 2$ *then* $C(n) = O(n^{\alpha})$, *where* C *is the complexity of triangular transitive closure.*

COROLLARY. *The recognition problem can be solved in time* $O(n^{2.81})$. All previously known algorithms require at least $O(n^3)$. (See for example, Younger's algorithm [You 67].) An alternative account of Valiant's algorithm can be found in [Pat 74].

REFERENCES

Bre 70 BRENT R., *'Algorithms for matrix multiplication'*, STAN-CS-70-157 (March 1970) Computer Science Dept., Stanford U.

Bru 72 BRUCKER P., *'R-Netzwerke und Matrixalgorithmen'*, Computing 10 (1972) 271-283.

Bru 74 BRUCKER P., *'Theory of matrix algorithms'*, Mathematical Systems in Economics 13 (Verlag Anton Hain KG, 1974).

Bun 72 BUNCH J. & J. HOPCROFT, *'Triangular factorization and inversion by fast matrix multiplication'*, TR 72-152 (1972) Computer Science Dept., Cornell U.

Con 71 CONWAY J., *Regular Algebra & Finite Machines*. (Chapman and Hall, 1971).

Fid 71 FIDUCCIA C., *'Fast matrix multiplication'*, Proc. 3rd Annual ACM Symp. on Theory of Computing (1971), 45-49.

Fid 72 FIDUCCIA C., *'On obtaining upper bounds on the complexity of matrix multiplication'*, Complexity of Computer Computations, eds. R. Miller & J. Thatcher (1972 Plerum Press, N.Y.) 31-40.

Fis 71 FISHER M. & A. MEYER, *'Boolean matrix multiplication and transitive closure'*, IEEE Conf. Record of 12th Annual Symposium on Switching and Automata Theory (1971), 129-131.

Flo 62 FLOYD R., *'Algorithm 97, Shortest Path'*, CACM 5, 6, (June, 1962) 345-345.

Fur 70 FURMAN M., *'Application of a method of fast multiplication of matrices in the problem of finding the transitive closure of a graph'*, Dokl. Akad. Nauk SSSR 194 (1970) p.524 (Russian). English translation in Soviet Math. Dokl. 11 (1970) p.1252.

Hof 72 HOFFMAN A. & S. WINOGRAD, *'Finding all shortest distances in a directed network'*, IBM Journ. of R. and D., 16 4 (July 1972) 412-414.

Hop 71 HOPCROFT J. & L.KERR, *'On minimizing the number of multiplications necessary for matrix multiplication'*, SIAM J. Appl. Math. 20 (1971), 30-35.

Hop 73 HOPCROFT J. & J. MUSINSKI, *'Duality applied to the complexity of matrix multiplication and other bilinear forms'*, SIAM J. Computing, 2, 3, (Sept. 1973) 159-173.

Mun 71 MUNRO I., *'Efficient determination of the transitive closure of a directed graph'*, Information Processing Letters 1 (1971) 56-58.

Pai 70 PAIR C., *'Mille et un algorithmes pour les problèmes de cheminement dans les graphes'*, Revue Française d'Informatique et de Recherche opérationelle, B-3, (1970) 125-143.

Pat 74 PATERSON M., *'Complexity of product and closure algorithms for matrices'*, Proc. Int. Congress of Math., Vancouver, 1974.

Pat 74 PATERSON M., *'Complexity of monotone networks for Boolean matrix product'*, to appear in Theoretical Computer Science 1 (1975).

Pro 75 PROBERT R., *'Additive symmetry in matrix product computations'*, Report 75-1 (January 1975) Dept. of Comp. Sci., U. of Saskatchewan, Saskatoon.

Sch 71 SCHÖNHAGE A. & V. STRASSEN, *'Fast multiplication of large numbers'*, Computing 7 (1971) 281-292 (German with English summary).

Str 69 STRASSEN V., *'Gaussian elimination is not optimal'*, Numer. Math. 13 (1969) 354-356.

Str 72 STRASSEN V., *'Evaluation of rational functions'*, Complexity of Computer Computations, eds. R. Miller & J. Thatcher (1972 Plenum Press, N.Y.) 1-10.

Val 74 VALIANT L., *'General context-free recognition in less than cubic time'*, to appear in JCSS.

War 62 WARSHALL S., *'A theorem on Boolean matrices'*, JACM 9 (1962) 11-12.

Win 70 WINOGRAD S., *'On the number of multiplications necessary to compute certain functions'*, Comm. Pure Appl. Math. 23 (1970) 165-179.

Win 71 WINOGRAD S., *'On multiplication of 2 × 2 matrices'*, Linear algebra and its applications 4 (1971) 381-388.

You 67 YOUNGER D., *'Recognition and parsing of context-free languages in time n^3'*, Inf. and Control. 10 (1967) 189-208.

OTHER TITLES IN THE SERIES MATHEMATICAL CENTRE TRACTS

A leaflet containing an order-form and abstracts of all publications mentioned below is available at the Mathematisch Centrum, Tweede Boerhaavestraat 49, Amsterdam-1005, The Netherlands. Orders should be sent to the same address.

MCT 1 T. VAN DER WALT, *Fixed and almost fixed points*, 1963. ISBN 90 6196 002 9.

MCT 2 A.R. BLOEMENA, *Sampling from a graph*, 1964. ISBN 90 6196 003 7.

MCT 3 G. DE LEVE, *Generalized Markovian decision processes, part I: Model and method*, 1964. ISBN 90 6196 004 5.

MCT 4 G. DE LEVE, *Generalized Markovian decision processes, part II: Probabilistic background*, 1964. ISBN 90 6196 006 1.

MCT 5 G. DE LEVE, H.C. TIJMS & P.J. WEEDA, *Generalized Markovian decision processes, Applications*, 1970. ISBN 90 6196 051 7.

MCT 6 M.A. MAURICE, *Compact ordered spaces*, 1964. ISBN 90 6196 006 1.

MCT 7 W.R. VAN ZWET, *Convex transformations of random variables*, 1964. ISBN 90 6196 007 X.

MCT 8 J.A. ZONNEVELD, *Automatic numerical integration*, 1964. ISBN 90 6196 008 8.

MCT 9 P.C. BAAYEN, *Universal morphisms*, 1964. ISBN 90 6196 009 6.

MCT 10 E.M. DE JAGER, *Applications of distributions in mathematical physics*, 1964. ISBN 90 6196 010 X.

MCT 11 A.B. PAALMAN-DE MIRANDA, *Topological semigroups*, 1964. ISBN 90 6196 011 8.

MCT 12 J.A.TH.M. VAN BERCKEL, H. BRANDT CORSTIUS, R.J. MOKKEN & A. VAN WIJNGAARDEN, *Formal properties of newspaper Dutch*, 1965. ISBN 90 6196 013 4.

MCT 13 H.A. LAUWERIER, *Asymptotic expansions*, 1966, out of print; replaced by MCT 54.

MCT 14 H.A. LAUWERIER, *Calculus of variations in mathematical physics*, 1966. ISBN 90 6196 020 7.

MCT 15 R. DOORNBOS, *Slippage tests*, 1966. ISBN 90 6196 021 5.

MCT 16 J.W. DE BAKKER, *Formal definition of programming languages with an application to the definition of ALGOL 60*, 1967. ISBN 90 6196 022 3.

MCT 17 R.P. VAN DE RIET, *Formula manipulation in ALGOL 60, part 1*, 1968. ISBN 90 6196 025 8.

MCT 18 R.P. VAN DE RIET, *Formula manipulation in ALGOL 60, part 2*, 1968. ISBN 90 6196 038 X.

MCT 19 J. VAN DER SLOT, *Some properties related to compactness*, 1968. ISBN 90 6196 026 6.

MCT 20 P.J. VAN DER HOUWEN, *Finite difference methods for solving partial differential equations*, 1968. ISBN 90 6196 027 4.

MCT 21 E. WATTEL, *The compactness operator in set theory and topology*, 1968. ISBN 90 6196 028 2.

MCT 22 T.J. DEKKER, *ALGOL 60 procedures in numerical algebra, part 1*, 1968. ISBN 90 6196 029 0.

MCT 23 T.J. DEKKER & W. HOFFMANN, *ALGOL 60 procedures in numerical algebra, part 2*, 1968. ISBN 90 6196 030 4.

MCT 24 J.W. DE BAKKER, *Recursive procedures*, 1971. ISBN 90 6196 060 6.

MCT 25 E.R. PAERL, *Representations of the Lorentz group and projective geometry*, 1969. ISBN 90 6196 039 8.

MCT 26 EUROPEAN MEETING 1968, *Selected statistical papers, part I*, 1968. ISBN 90 6196 031 2.

MCT 27 EUROPEAN MEETING 1968, *Selected statistical papers, part II*, 1969. ISBN 90 6196 040 1.

MCT 28 J. OOSTERHOFF, *Combination of one-sided statistical tests*, 1969. ISBN 90 6196 041 X.

MCT 29 J. VERHOEFF, *Error detecting decimal codes*, 1969. ISBN 90 6196 042 8.

MCT 30 H. BRANDT CORSTIUS, *Excercises in computational linguistics*, 1970. ISBN 90 6196 052 5.

MCT 31 W. MOLENAAR, *Approximations to the Poisson, binomial and hypergeometric distribution functions*, 1970. ISBN 90 6196 053 3.

MCT 32 L. DE HAAN, *On regular variation and its application to the weak convergence of sample extremes*, 1970. ISBN 90 6196 054 1.

MCT 33 F.W. STEUTEL, *Preservation of infinite divisibility under mixing and related topics*, 1970. ISBN 90 6196 061 4.

MCT 34 I. JUHÁSZ, A. VERBEEK & N.S. KROONENBERG, *Cardinal functions in topology*, 1971. ISBN 90 6196 062 2.

MCT 35 M.H. VAN EMDEN, *An analysis of complexity*, 1971. ISBN 90 6196 063 0.

MCT 36 J. GRASMAN, *On the birth of boundary layers*, 1971. ISBN 90 6196 064 9.

MCT 37 J.W. DE BAKKER, G.A. BLAAUW, A.J.W. DUIJVESTIJN, E.W. DIJKSTRA, P.J. VAN DER HOUWEN, G.A.M. KAMSTEEG-KEMPER, F.E.J. KRUSEMAN ARETZ, W.L. VAN DER POEL, J.P. SCHAAP-KRUSEMAN, M.V. WILKES & G. ZOUTENDIJK, *MC-25 Informatica Symposium*, 1971. ISBN 90 6196 065 7.

MCT 38 W.A. VERLOREN VAN THEMAAT, *Automatic analysis of Dutch compound words*, 1971. ISBN 90 6196 073 8.

MCT 39 H. BAVINCK, *Jacobi series and approximation*, 1972. ISBN 90 6196 074 6.

MCT 40 H.C. TIJMS, *Analysis of (s,S) inventory models*, 1972. ISBN 90 6196 075 4.

MCT 41 A. VERBEEK, *Superextensions of topological spaces*, 1972. ISBN 90 6196 076 2.

MCT 42 W. VERVAAT, *Success epochs in Bernoulli trials (with applications in number theory)*, 1972. ISBN 90 6196 077 0.

MCT 43 F.H. RUYMGAART, *Asymptotic theory of rank tests for independence*, 1973. ISBN 90 6196 081 9.

MCT 44 H. BART, *Meromorphic operator valued functions*, 1973. ISBN 90 6196 082 7.

MCT 45 A.A. BALKEMA, *Monotone transformations and limit laws*, 1973. ISBN 90 6196 083 5.

MCT 46 R.P. VAN DE RIET, *ABC ALGOL, A portable language for formula manipulation systems, part 1: The language*, 1973. ISBN 90 6196 084 3.

MCT 47 R.P. VAN DE RIET, *ABC ALGOL A portable language for formula manipulation systems part 2: The compiler*, 1973. ISBN 90 6196 085 1.

MCT 48 F.E.J. KRUSEMAN ARETZ, P.J.W. TEN HAGEN & H.L. OUDSHOORN, *In ALGOL 60 compiler in ALGOL 60, Text of the MC-compiler for the EL-X8*, 1973. ISBN 90 6196 086 X.

MCT 49 H. KOK, *Connected orderable spaces*, 1974. ISBN 90 6196 088 6.

MCT 50 A. VAN WIJNGAARDEN, B.J. MAILLOUX, J.E.L. PECK, C.H.A. KOSTER, M. SINTZOFF, C.H. LINDSEY, L.G.L.T. MEERTENS & R.G. FISKER (eds.), *Revised report on the algorithmic language ALGOL 68*. ISBN 90 6196 089 4.

MCT 51 A. HORDIJK, *Dynamic programming and Markov potential theory*, 1974. ISBN 90 6196 095 9.

MCT 52 P.C. BAAYEN (ed.), *Topological structures*, 1974. ISBN 90 6196 096 7.

MCT 53 M.J. FABER, *Metrizability in generalized ordered spaces*, 1974. ISBN 90 6196 097 5.

MCT 54 H.A. LAUWERIER, *Asymptotic analysis, part 1*, 1974. ISBN 90 6196 098 3.

MCT 55 M. HALL JR. & J.H. VAN LINT (eds.), *Combinatorics, part 1: Theory of designs finite geometry and coding theory*, 1974. ISBN 90 6196 099 1.

MCT 56 M. HALL JR. & J.H. VAN LINT (eds.), *Combinatorics, part 2: Graph theory; foundations, partitions and combinatorial geometry*, 1974. ISBN 90 6196 100 9.

MCT 57 M. HALL JR. & J.H. VAN LINT (eds.), *Combinatorics, part 3: Combinatorial group theory*, 1974. ISBN 90 6196 101 7.

MCT 58 W. ALBERS, *Asymptotic expansions and the deficiency concept in statistics*, 1975. ISBN 90 6196 102 5.

MCT 59 J.L. MIJNHEER, *Sample path properties of stable processes*, 1975. ISBN 90 6196 107 6.

MCT 60 F. GÖBEL, *Queueing models involving buffers*. ISBN 90 6196 108 4.

* MCT 61 P. VAN EMDE BOAS, *Abstract resource-bound classes, part 1*. ISBN 90 6196 109 2.

* MCT 62 P. VAN EMDE BOAS, *Abstract resource-bound classes, part 2*. ISBN 90 6196 110 6.

MCT 63 J.W. DE BAKKER (ed.), *Foundations of computer science*, 1975. ISBN 90 6196 111 4.

MCT 64 W.J. DE SCHIPPER, *Symmetrics closed categories*, 1975. ISBN 90 6196 112 2.

MCT 65 J. DE VRIES, *Topological transformation groups 1 A categorical approach*, 1975. ISBN 90 6196 113 0.

* MCT 66 H.G.J. PIJLS, *Locally convex algebras in spectral theory and eigenfunction expansions*. ISBN 90 6196 114 9.

* MCT 67 H.A. LAUWERIER, *Asymptotic analysis, part 2.* ISBN 90 6196 119 X.

* MCT 68 P.P.N. DE GROEN, *Singulary pertlibed differential operators of second order.* ISBN 90 6196 120 3.

* MCT 69 J.K. LENSTRA, *Sequencing by enumerative methods.* ISBN 90 6196 125 4.

* MCT 70 W.P. DE ROEVER JR., *Recursive program schemes: semantics and proof theory.* ISBN 90 6196 127 0.

* MCT 71 J.A.E.E. VAN NUNEN, *Contracting Markov decision processes.* ISBN 90 6196 129 7.

* MCT 72 J.K.M. JANSEN, *Simple periodic and nonperiodic Lamé functions and their applications in the theory of eletromagnetism.* ISBN 90 6196 130 0.

* MCT 73 D.M.R. Leivant, *Absoluteness of intuitionistic logic.* ISBN 90 6196 122 x.

* MCT 74 H.J.J. Te Riele, *A theoretical and computational study of generalized aliquot sequences.* ISBN 90 6196 131 9.

An asterisk before the number means "to appear".