

DEPT. OF COMPUTER SCIENCE,  
UNIVERSITY OF WATERLOO,  
WATERLOO, ONTARIO N2L 3G1

Department of Applied Analysis  
and Computer Science

Research Report CS-74-05

April 1974

RENAMINGS IN  
PARALLEL PROGRAM SCHEMAS

by

Luigi Logrippo

**Faculty of Mathematics**  
**University of Waterloo**  
**Waterloo, Ontario**  
**Canada**



**Department of Applied Analysis**  
**&**  
**Computer Science**

Department of Applied Analysis  
and Computer Science

Research Report CS-74-05

April 1974

RENAMINGS IN  
PARALLEL PROGRAM SCHEMAS

by

Luigi Logrippo

RENAMINGS  
IN  
PARALLEL PROGRAM SCHEMAS

by

Luigi Logrippo

A thesis  
presented to  
the  
University of Waterloo  
in partial fulfillment  
of the requirements  
for the degree of

DOCTOR OF PHILOSOPHY

Department of Applied Analysis & Computer Science

Faculty of Mathematics

February 1974

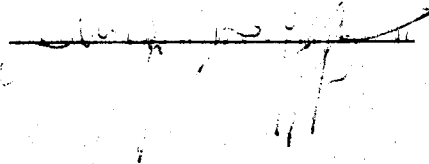
The University of Waterloo requires  
the signature of all persons using  
this thesis. Please sign below and  
give address and date.

© Luigi Logrippo, 1974

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend it to other institutions or individuals for the purpose of scholarly research.

Signature

A handwritten signature in black ink, appearing to be "John A. ...", written over a horizontal line.

ABSTRACT

In this thesis, we study the effects of changing the names of the variables (i.e., performing renamings) in parallel program schemas (hereafter simply called "schemas"). First, we give a method for performing renamings in such a way that the resulting schema computes step by step the same values as the original one. We call "proper renamings" the renamings that are performed according to this method. We show that there exists a class of schemas such that any two schemas in the class for which one is not a proper renaming of the other do not compute step by step the same values. In this sense our method of renaming is the only one that is generally valid.

Some applications of this technique are then studied. First, we show that, given any schema  $S$ , there exists a procedure for obtaining another schema  $S'$  that is a proper renaming of  $S$  and uses no more variables than any proper renaming of  $S$ . Furthermore, we show that, by changing the control structure of  $S$ , it is possible to obtain a schema  $S''$ , with computations identical to those of  $S$ , which can be renamed to use the smallest possible number of variables. The number of variables in the resulting schema is less than or equal to the number of variables in the original schema.

Finally, we study the effects of renamings on parallelism. It is shown that the amount of parallelism present in a schema can be enhanced by a procedure that

involves both proper renamings and transformation of control structure. Iteration of this procedure tends towards a limit that can be thought of as the maximally parallel version of the original schema.

## ACKNOWLEDGEMENT

I wish to express my gratitude to my thesis advisors, Professor J.A. Brzozowski and Professor E.A. Ashcroft, for their very helpful suggestions and criticism. These are too many to be individually acknowledged. However, I am particularly indebted to Professor Brzozowski for his contributions to the definition of the model and to Professor Ashcroft for providing the idea of the proofs of theorems 3.6.4 and 5.2.9.

I am indebted to Drs. D.P. Bovet, R.M. Keller, R.E. Miller for stimulating discussions, and to the members of the examining committee, Drs. R.E. Miller, E.L. Robertson and H.S. Shank for their useful comments.

I wish to thank Mrs. Claudette Henderson and Mrs. Teresa Miao for the skilful typing of the manuscript.

This research was supported in part by the Consiglio Nazionale delle Ricerche, Comitato per la Matematica (Italy), and in part by the National Research Council of Canada under Grant No. A-1617.



TABLE OF CONTENTS

CHAPTER 1 - INTRODUCTION	.....	1
§1.1 Generalities	.....	1
§1.2 The Model	.....	1
§1.3 Overview of problems and results	.....	11
§1.4 Previous research	.....	19
CHAPTER 2 - NOTATION AND TERMINOLOGY	.....	21
§2.1 Basic notation and terminology	.....	21
§2.2 List of definitions	.....	25
§2.3 Numbering system	.....	31
CHAPTER 3 - RENAMINGS IN TERMINATOR STRINGS	.....	33
§3.1 Terminator strings, interpretations, histories.....		34
§3.2 Equivalence and similarity	.....	44
§3.3 Segments	.....	47
§3.4 Renamings	.....	51
§3.5 Proper renamings and values computed	.....	58
§3.6 The converse results	.....	64
CHAPTER 4 - RENAMINGS IN SCHEMAS	.....	73
§4.1 Schemas	.....	74
§4.2 Areas in schemas	.....	79
§4.3 Renamings in schemas	.....	83
§4.4 Computations and renamings in schemas	.....	91

§4.5	A converse of #3.4. Tree schemas	..... 95
§4.6	A converse of #4.4.	.....103
CHAPTER 5 - MEMORY ECONOMY		.....108
§5.1	Minimum memory requirements for schemas	.....108
§5.2	Schemas with minimum memory in a set of L-equivalent schemas	.....113
§5.3	It is decidable whether two finite, consistent schemas have the same language up to renaming	.....126
§5.4	Dynamic memory allocation	.....129
CHAPTER 6 - PARALLELISM		.....131
§6.1	Enhancing parallelism in schemas: the first step	.....132
§6.2	Enhancing parallelism in schemas: the further steps	.....144
§6.3	Maximal parallelism	.....155
DIRECTIONS OF FURTHER RESEARCH		.....161
BIBLIOGRAPHY		.....164

## CHAPTER 1

### INTRODUCTION

#### §1.1 Generalities.

Among the transformations that are known to preserve the results of programs, an important place is occupied by transformations that affect the names of the variables, i.e. require a renaming. For example, such transformations are used in the optimization phase of some compilers [Gri]. The aim of this thesis is to develop a theory of renamings in the framework of a theory of program schemas inspired by the work of Karp and Miller [K&M].

First, we shall consider properties of renamings, and then their two main applications: we may want to rename in order to decrease the amount of working storage used by a program; or we may want to rename in order to increase the parallelism of a program, by eliminating bottlenecks caused by the fact that two instructions or two procedures share some storage space unnecessarily.

In this chapter, we shall introduce informally the concepts discussed in the thesis.

#### §1.2 The Model.

The development of the theory of program schemas has produced a considerable number of models and formalisms. Some of the most notable differences have concerned the

representation of parallelism. We have decided, not without regret, to produce yet another model, a variation of those studied in [K&M] and [Kel]. In the following chapters we shall explain why the departures from those models were deemed necessary.

A parallel program schema, or simply a schema, can be seen in two different ways: as a control mechanism for parallel processing, or as an acceptor of computations. The first interpretation is discussed in [Kel] and we refer the reader to that source for further explanations. We shall only note that the concepts of "enabling operations" and "controlling automaton" introduced in that work are compatible with our model, and that the reader who prefers that interpretation may certainly adopt it. For a formal treatment, it is not necessary to refer to the concepts above, and a schema can be viewed simply as an acceptor of computations, as we shall now show.

Consider the program of Fig. 1, where parallelism is represented in terms of the well-known operations of "fork" and "join" [Con] and the integers represent memory addresses (we call this kind of representation "flow-chart representation"). First of all, since we are studying programs from the point of view of control flow, rather than from the point of view of functions computed, we replace functions with function variables. This operation of abstraction yields the schema of the program, represented in Fig. 2. We say that  $(f(0) \rightarrow 1, 2)$ ,  $(k(0, 1) \rightarrow 0)$ ,  $(g(3, 2) \rightarrow 4)$ ,  $(h(0, 3))$  are the operations

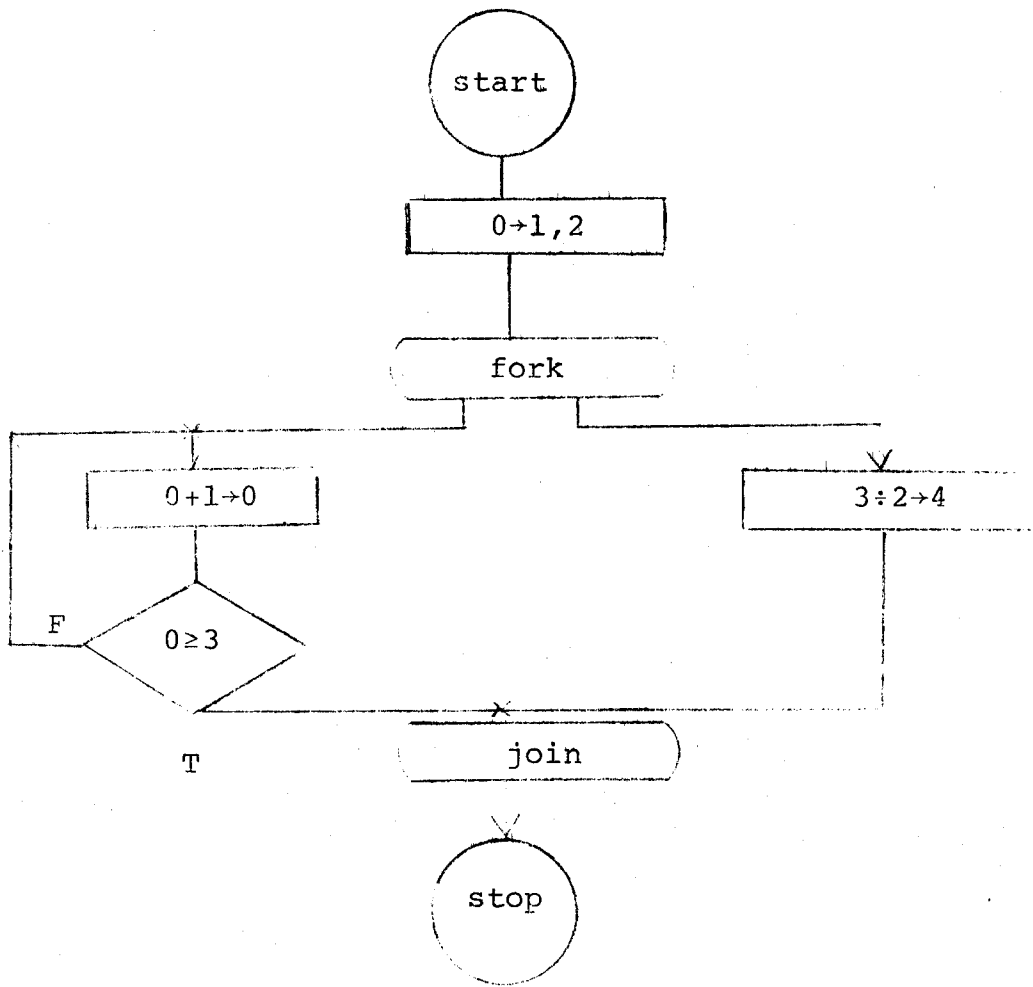


Fig. 1. A parallel flow-chart

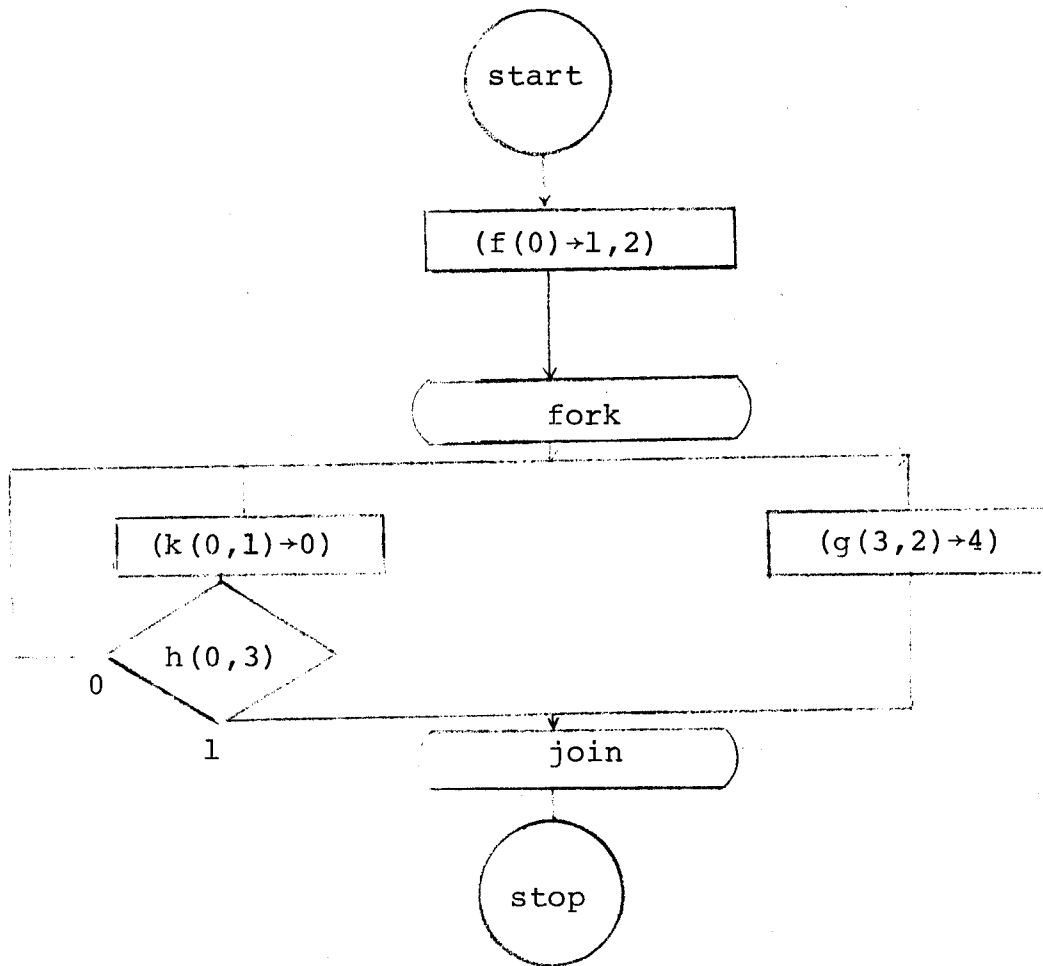


Fig. 2. A schema for the flow-chart of Fig. 1.

1.2

of the schema. These operations have the following ordered sets of domain variables:  $(0)$ ,  $(0,1)$ ,  $(3,2)$ ,  $(0,3)$  and the following ordered sets of range variables:  $(1,2)$ ,  $(0)$ ,  $(4)$ ,  $\emptyset$ . In our model, every operation embodies a test, possibly a trivial one that always gives the same result. The result of a test is called its outcome. In our example, all operations have just one outcome, except  $(h(0,3))$  which has two: 0 and 1.

Given a schema, we go back to a program by means of an interpretation, i.e. by replacing function variables with actual functions and tests. For convenience, we shall also include in the interpretation the specification of the initial memory values.

Let us now consider the set of all possible instruction sequences of programs whose schema is that of Fig. 2. If we denote with  $a^i$  the execution of operation  $a$  with outcome  $i$ , and assume that operations with only one outcome always have outcome 0, we see that some of these instruction sequences are:

$$(f(0) \rightarrow 1,2)^0 (g(3,2) \rightarrow 4)^0 (k(0,1) \rightarrow 0)^0 (h(0,3))^1$$

$$(f(0) \rightarrow 1,2)^0 (k(0,1) \rightarrow 0)^0 (h(0,3))^1 (g(3,2) \rightarrow 4)^0$$

etc.

One can verify that the set of all finite instruction sequences allowed by  $S$  is the language of the finite automaton of Fig. 3. This is an incompletely specified Moore automaton: the vertices of the graph of Fig. 3 represent its states, while for clarity we have enclosed the labels of the edges in

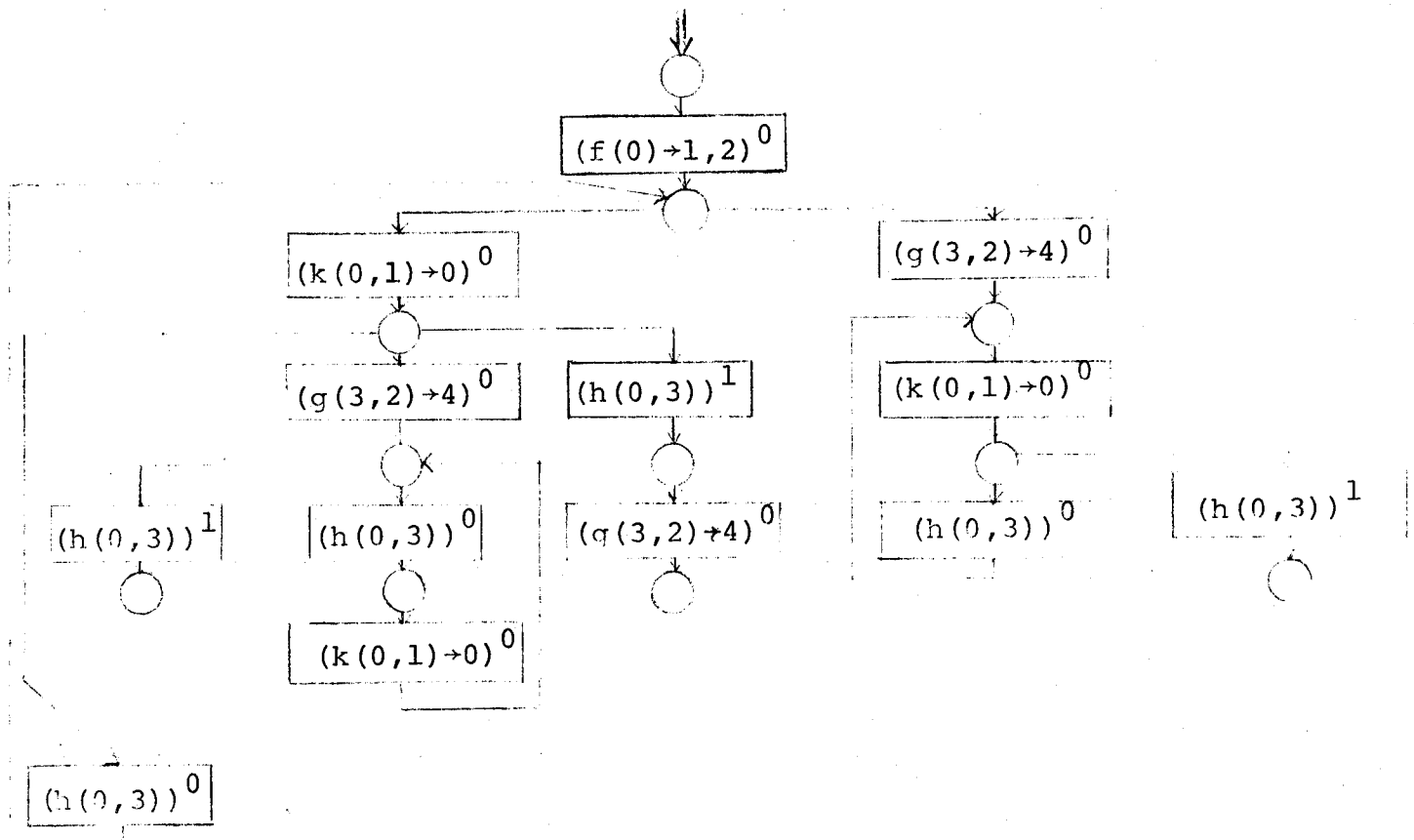


Fig. 3. Representation of the schema in Fig. 2 as an automaton.



boxes (graphs with such long labels might otherwise be difficult to read). Every state of the automaton is taken to be an accepting state.

In this thesis, we shall define schemas as such automata, rather than as flow-charts, mainly because in this way we can consider schemas as acceptors of instruction sequences, and can use the terminology and some of the results of automata theory. Furthermore, the automaton representation is more general than the flow-chart representation: Ashcroft and Manna [Man] [A&M] give a procedure for transforming every schema in the latter representation into a schema in the former, while one easily verifies that the converse is not always possible.

To be capable of computing, a schema must be interpreted: we have already seen that an interpreted schema is a program. An interpretation consists of three elements:

a domain, i.e. a set of possible variable values;

the initial content of each memory variable;

an assigned meaning for each function variable.

The assigned meaning consists of two functions: a function taking its arguments and results in the domain, and a function taking its arguments in the domain and its results in the set  $\{0,1,\dots\}$ . This second function is the one that computes the outcome.

Concerning the schema just considered, a possible interpretation  $I_0$  for it is the following.

The domain is the set of nonnegative integers.

The initial variable values are fixed as follows:

the initial value of 0 is '2', the initial value of 3 is '6', and the initial value of the remaining variables is '0'.

The function variables are interpreted as in Fig. 1:

f is the identity function, k is an addition, g is a division, and h is a test. The outcome for each operation has value 0, except for the function h, which has outcomes 0 and 1;  $h(m,n)$  has outcome 1 iff the value of m is greater or equal than the value of n.

A possible instruction sequence for this interpretation is:

$(f(0) \rightarrow 1, 2)^0$	i.e.	$0 \rightarrow 1, 2$
$(k(0, 1) \rightarrow 0)^0$		$0 + 1 \rightarrow 0$
$(h(0, 3))^0$		$0 < 3$
$(k(0, 1) \rightarrow 0)^0$		$0 + 1 \rightarrow 0$
$(h(0, 3))^1$		$0 \geq 3$
$(g(3, 2) \rightarrow 4)^0$		$3 \div 2 \rightarrow 4$

In this sequence,  $(g(3, 2) \rightarrow 4)$  is executed last; however, this instruction could have been executed at any time after the execution of  $(f(0) \rightarrow 1, 2)$ .

We see that the successive memory values for this sequence are as follows:

		variables					
		0	1	2	3	4	...
initial value	2	0	0	6	0	0	
	4	2	2	6	0	0	
successive values	4	2	2	6	0	0	
	6	2	2	6	0	0	
	6	2	2	6	0	0	
	6	2	2	6	0	0	
	6	2	2	6	3	0	

If only the changes in memory values are shown, we get

	0	1	2	3	4	...
2	0	0	6	0	0	
4	2	2		3		
6						

This is what we call history of the sequence. It is interesting to note that this concept of history is independent of the concept of schema, in that a history is uniquely defined by an instruction sequence and an interpretation. We say that an instruction sequence, finite or infinite, is an I-computation for an interpretation I if a history is defined for the sequence under I. For example, the string

$$(f(0) \rightarrow 1, 2)^0 (k(0, 1) \rightarrow 0)^0 (h(0, 3))^1$$

is not an  $I_0$ -computation since the test  $h(0, 3)$  at that point can only have outcome 0, i.e.  $\leftarrow$ .

We say that an instruction sequence is a computation

for a given schema  $S$  under a given interpretation  $I$  if the sequence is an  $I$ -computation and is either

i) finite and it ends in a state of the schema where no further transition is defined

or ii) infinite and all its prefixes are accepted by the schema.

We take the results of a computation to be its whole history. This makes it possible to consider infinite computations, thus allowing modeling of systems, such as operating systems or real-time systems, that have legitimate infinite computations.

Returning to our example, we note that all  $I_0$ -computations have the same history, i.e. are equivalent with respect to  $I_0$ . This concept of equivalence can be generalized as follows. We say that two computations are  $I$ -similar for a given interpretation  $I$  if their histories for that interpretation contain the same values, not necessarily in the same order. Two computations are similar if they are  $I$ -similar for all  $I$ .

A computation is repetition-free if it never computes the same value or the same test twice by applying the same functions in the same order. Thus in a certain sense, repetition-free computations are reduced. A repetition-free schema has only repetition-free computations.

Concerning the adequacy of this model for describing the complexities of real-life computational systems, we refer the reader to [P&H]. These authors show that models of the type described above are inadequate to express recursion. However, while in a model allowing recursion many of our

results would not hold, there are areas where our model is meaningful. One of these is the modeling of internal computer operations and microprogramming, where everything is done by using absolute addresses. Another is the modeling of large systems, such as operating systems, where each operation can be taken as a subsystem.

### §1.3 Overview of problems and results

#### Renamings in computations and schemas

We have found it convenient to attack the problem of renaming by first considering computations, then schemas. As regards computations, we want to find a renaming rule with the following properties:

- i) By renaming a computation  $x$  according to such a rule, we obtain a computation  $x'$  such that, for the same interpretation,  $x$  and  $x'$  compute step by step the same values.
- ii) There exists a computation  $x$  such that, for any renaming violating the renaming rule, the renamed computation  $x'$  does not always compute step by step the same values.

We first show how our renaming rule works by means of example. Consider the following computation:

$$(f(2,3) \rightarrow 0)^0 \quad (g(3) \rightarrow 1)^0 \quad (h(0,1) \rightarrow 0,3)^0 \quad (g(3) \rightarrow 3)^0$$

It is clear that, if we replace variable 0 in the substring within square bracket with some other variable (say 4) that is not "occupied" in that substring, then the resulting compu-

tation computes step by step the same values as the original computation:

$$(f(2,3) \rightarrow 4)^0 (g(3) \rightarrow 1)^0 (h(4,1) \rightarrow 0,3)^0 (g(3) \rightarrow 3)^0$$

We say that the substring above is a segment of the variable 0. In general, a segment of a variable  $m$  in a computation  $x$  is a substring  $M$  in  $x$  such that:

- i) at the beginning of  $M$ ,  $m$  is assigned a new value;
- ii) the value assigned to  $m$  at the beginning of the segment is not changed within the segment; and
- iii) at the end of  $M$ , this value is used for the last time in  $x$ .

A simple store of a new value in  $m$  that will never be used again in  $x$  is also a (trivial) segment.

The desired renaming rule for computations is found to be the following:

A variable  $m$  can be renamed as  $n$  in any segment  $M$  of  $m$  in the computation, provided that  $M$  does not intersect any segment of  $n$ .  $n$  must replace  $m$  uniformly throughout  $M$ .

Renamings abiding by this rule will be called proper. Furthermore, we shall show that if a computation  $x'$  is obtained by renaming a repetition-free computation  $x$  without following the rule above, then  $x$  and  $x'$  do not compute step by step the same values.

As regards schemas, we say that two schemas  $S$  and

$S'$  compute step by step the same values if for any computation  $x$  of  $S$  there exists a computation of  $S'$  that for all interpretations computes step by step the same values as  $x$ , and vice versa. We want to find a renaming rule with the following properties:

- i) By renaming a schema  $S$  according to the rule, we obtain a schema  $S'$  that computes step by step the same values as  $S$ .
- ii) There exists a class of schemas such that any two schemas in the class that cannot be obtained the one from the other by the rule do not compute step by step the same values.

To do this, we first notice that if any computation follows a particular path through a schema then the computation is simply the sequence of instructions labeling the edges of that path. For a schema  $S$ , we define the areas of a variable  $m$  to be particular subgraph-like objects, corresponding to segments of  $m$  in computations of  $S$ . The desired renaming rule is found to be:

A variable  $m$  can be renamed as  $n$  in any area  $M$  of  $m$  in the schema, as long as  $M$  does not intersect any area of  $n$  and as long as the renaming is performed uniformly in  $M$ .

Renamings abiding by this rule are called proper.

#### First application: memory economy

It is easy to find two schemas for which one is a proper

renaming of the other but which use different numbers of variables. This phenomenon is studied in Chapter 5 as the most obvious application of renamings.

We define the incompatibility graph of a schema  $S$  to be a graph having as many vertices as there are areas in  $S$  and where two vertices are joined by an edge if the corresponding two areas intersect. A proper renaming of  $S$  can be obtained by associating variables with vertices of the graph in such a way that no two vertices connected by an edge are associated with the same variable, and then correspondingly renaming the areas of  $S$ . Thus, it is seen that the problem of renaming a schema in such a way that a minimum amount of memory is used reduces to the well-known problem of coloring a graph with a minimum number of colors.

It is also interesting to note that two schemas that accept the same language may have minimum renamings that use different amounts of memory. This problem is connected to a problem that in compiler theory is known as "register mismatch" problem. We show that, given a finite schema  $S$ , it is possible to construct a schema  $S'$  that accepts the same language and can be properly renamed with the smallest possible number of variables. The construction is of a kind that could be included in an optimizing compiler.

The above provides a theory of static storage allocation. Dynamic storage allocation is also briefly considered, and it is shown that it makes possible an even better utilization of memory.



Second application: parallelism.

Renamings can also be used in order to increase the amount of potential parallelism of a schema. A trivial example of this fact is shown in Fig. 4. Assume that a schema contains the sequence of operations shown in A. The two operations could not be executed in parallel, nor could the second one be executed before the first, without changing the results of the program. However, there is no intrinsic reason why the second operation should wait for the first one, since the second operation does not use any value computed by the first one. We have here a bottleneck caused by the fact that the second operation happens to store its results in the same memory locations from which the first one fetches. This bottleneck can be eliminated by renaming, as in B, and then the two operations can be executed in either order, as in C. The last part of this work (Chapter 6) deals with these problems.

Two schemas are similar if for every computation of one there exists a similar computation of the other. A schema S is more parallel than a schema S' if S and S' are similar and for every computation of S' there is a computation of S that is a renaming of the computation of S' but not vice versa: in an intuitive way, this amounts to saying that S has more freedom in choosing the order of the operations than S'. A schema S is hyperclosed (i.e. maximally parallel) if there is no schema S' that is more parallel than S. Another "natural" definition of maximal parallelism

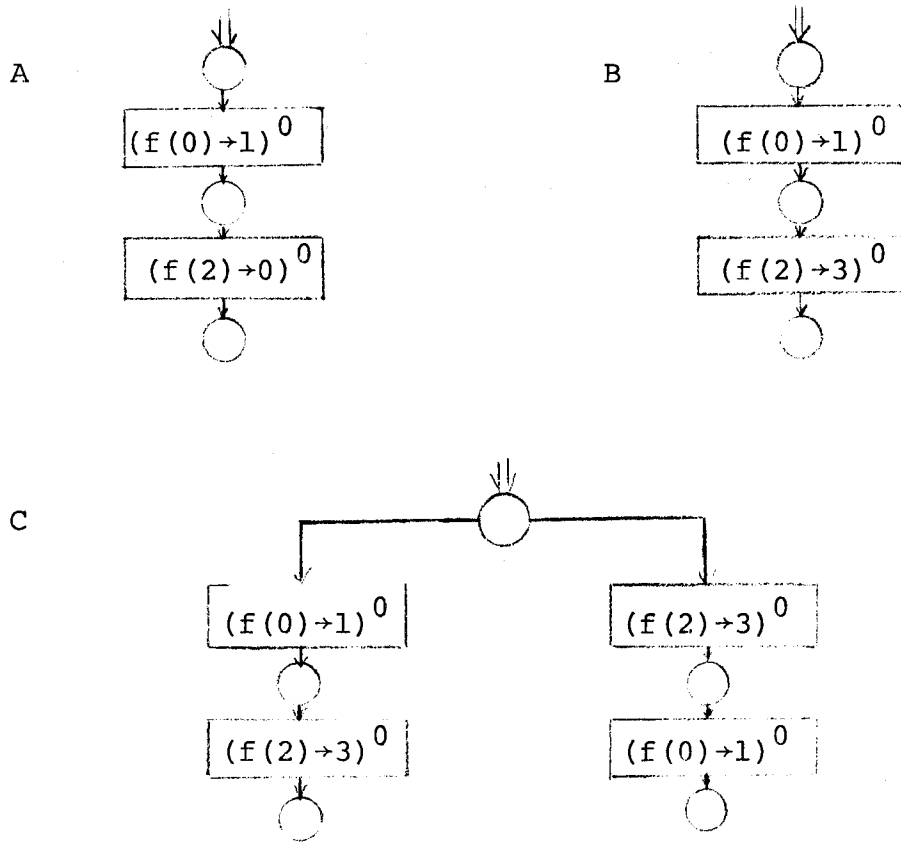


Fig. 4. Renaming to enhance parallelism.

is compared with this one and found to be equivalent, under reasonable assumptions.

The following properties are found to be true for a large class of schemas, called restricted schemas:

- i) For finite schemas, the property of being hyperclosed is decidable.
- ii) If a schema  $S$  is not hyperclosed, then it is possible to transform  $S$  into a schema  $S'$  that is more parallel than  $S$ ; furthermore, if  $S$  is finite, so is  $S'$ .
- iii) For every schema  $S$  there is a similar hyperclosed schema  $\tilde{S}$  called hyperclosure of  $S$ , that is usually not finite, even if  $S$  is.  $\tilde{S}$  can be thought of as the limit of the iterated application of ii). No algorithm is known for obtaining  $\tilde{S}$  from  $S$  even in the case where  $\tilde{S}$  is finite. However,  $\tilde{S}$  can be simulated by a sort of "look-ahead" interpreter.
- iv) If  $S$  and  $S'$  are similar schemas, then  $\tilde{S}$  accepts the same language as some proper renaming of  $\tilde{S}'$ .

A result related to the one of Paterson [Pat] about the decidability of the equivalence problem for progressive schemas is shown to follow easily from the results above.

Concerning the interpretation of these results, note that the fact that schemas usually do not have finite hyper-

closures (that may at first seem disappointing) lends itself to the following interesting interpretation. Given any program that is not maximally parallel, it is possible to enhance its parallelism by generating a larger, more parallel program. This process can be continued, until all available memory is used. This method of approximating hyperclosures could be called "static" and is suitable for use in a compiler.

Another way of approximating a hyperclosure is to "simulate" it dynamically by means of an instruction look-ahead interpreter. Such an interpreter works as follows: at the beginning of the program, and after executing each instruction, it looks ahead in the instruction stream to see whether there are any instructions that have become executable, i.e. instructions that only depend on data and tests that have already been computed. All these instructions are automatically "moved up" in the instruction stream, so that they can actually be executed at the earliest point where this is possible without losing similarity. To perform this correctly, renaming may be necessary (refer again to the example in Fig. 4). Such an interpreter may also run out of memory, either working memory or memory for tables needed during the simulation process. However, its use of working memory would in general be better than the use of memory in the static method, and here too, the larger the computer the higher the amount of parallelism that can be achieved. Interpreters using this sort of idea, but without facilities for renaming, are implemented in the hardware of a number of high-performance computers. An inter-

preter using renamings has been proposed by Stone [Sto].

#### §1.4 Previous research

The amount of work that has been done in the subjects touched by this thesis is very large, and relates to such different areas as program schema theory, hardware design, software design, and programming languages. This section will limit itself to mentioning the work that has most directly influenced this thesis.

As we have already stated, we have been influenced mainly by the papers of Karp and Miller [K&M] and Keller [Kel]. These authors proposed a model for parallel computing systems that has shown itself suitable for considering many properties of these systems that had escaped formal treatment before. We have tried to carry this investigation somewhat further.

The ideas of areas of a variable in a schema and of proper renamings were, to the knowledge of the author, first introduced by S. S. Lavrov in [Lav] (see also [Yel]). These works remained unnoticed in the West for a decade, while instead Russian authors widely acknowledged them and further exploited their implications (for an account of several related papers see [Y&L]). Lavrov defined proper renamings and showed that they are the only ones that preserve the area structure of a schema. We apply these ideas to parallel schemas and further extend them. We prove that a properly renamed schema computes the same values as the original schema,

and that in some sense proper renamings are the only ones having this property.

Chapter 6 bears some analogy with the works of Kotov [Kot] and Slutz [Slu]. However, the models used by these authors are very different from ours. The material of Chapter 6 is instead closely related to the work of Keller [Kel], and is a generalization of that work, in that we have introduced renamings into his theory of parallelism and generalized many of his results. In particular, it is interesting to note that both Keller and the author have found what can be considered a theoretical model of instruction look-ahead: however, while Keller's look-ahead interpreter is of the conventional type, similar to the ones implemented in the IBM 7030 [Buc], CDC 6600 [Lor] or IBM 360/91 [A,S&T], the author's look-ahead interpreter has dynamic storage allocation facilities, like that of [Sto].

## CHAPTER 2

### NOTATION AND TERMINOLOGY

#### §2.1 Basic notation and terminology.

The reader is assumed to be familiar with the basic results of the theory of computing. Most of our basic notation and terminology is standard in that field. Some additional notation is derived from [K&M] and [Kel].

A review of the basic notation follows.

'iff'	if and only if
{a,b,c,...}	is the set containing exactly the elements a,b,c,...
{a $\in$ A : P(a)}	is the subset of A containing exactly those elements for which P(a) is true
$\omega$	set of nonnegative integers
$\in \notin$	set membership and its negation
$\subseteq \not\subseteq$	set inclusion and its negation
$\subset$	proper inclusion
$\emptyset$	empty set
$\cup$	union
$\cap$	intersection
-	relative complement
$\times$	cartesian product
A	where A is a set, is the number of elements in A
$2^A$	is the set of all subsets of a set A

Sets consisting of couples of elements are also called relations.

If  $R$  is such a set, the two notations  $(a,b) \in R$  and  $aRb$  are equivalent. We also write  $a \not R b$  for  $(a,b) \notin R$ .

If two expressions have an equal sign '=' in the middle then the right-hand side is defined iff the left-hand side is defined and then they are equal.

$f: A \rightarrow B$  means that  $f$  is a (total or partial) function from a set  $A$  to a set  $B$ .  $A$  and  $B$  are respectively the domain and range of  $f$ .

$f$  is an identity function if for all  $a$  in the domain of  $f$ ,  $f(a) = a$ .

$(f(a))$  means:  $f(a)$  is defined

$f = g$  is true iff  $f$  and  $g$  are two functions with the same domain  $A$  and, for all  $a \in A$ ,  $f(a) = g(a)$ .

$f \subseteq g$  is true iff  $f$  and  $g$  are two functions such that the domain and range of  $f$  are included in respectively the domain and range of  $g$  and for all  $a$  that are in the domain of  $f$ ,  $f(a) = g(a)$ .

$f^{-1}$  for all  $a$  in the range of function  $f$ ,  $f^{-1}(a) = \{b : f(b) = a\}$

an injection is a function that is one-to-one

a bijection is an injection that is onto



- $B^A$  is the set of all total functions from  $A$  to  $B$ .
- $(a_1, \dots, a_n)$  ordered set of elements  $a_1, \dots, a_n$
- $A[I]$  where  $A$  is an ordered set taking indexes in  $\omega$ , and  $I = (i_1, \dots, i_n)$  is an ordered subset of  $\omega$ , is the ordered set  $(A_{i_1}, \dots, A_{i_n})$ .  $A[I]$  is defined iff all its elements are defined.
- strings we take the definition current in automata and language theory.
- $\Sigma^*$  for a set  $\Sigma$ , called alphabet,  $\Sigma^*$  is the set of all finite strings over  $\Sigma$ .
- $\Sigma^\omega$  is the set of all countably-infinite strings over  $\Sigma$ .
- $\hat{\Sigma}$  is  $\Sigma^* \cup \Sigma^\omega$
- $|x|$  (where  $x \in \hat{\Sigma}$ ) is the length of  $x$  (undefined iff  $x \in \Sigma^\omega$ ).
- $\lambda$  is the string of length 0.
- $xy$  or  $x \cdot y$  where  $x \in \Sigma^*$  and  $y \in \hat{\Sigma}$  is the concatenation of strings  $x$  and  $y$ , as usually defined in automata and language theory.
- $x \leq y$  (where  $x \in \Sigma^*$ ,  $y \in \hat{\Sigma}$ ) is true iff there exists  $z \in \hat{\Sigma}$  such that  $y = xz$ . In such a case,  $x$  is said to be a prefix of  $y$ .
- $x < y$  where  $x \in \Sigma^*$  and  $y \in \hat{\Sigma}$ , is true iff  $x \leq y$  and  $x \neq y$ . In this case,  $x$  is said to be a proper prefix of  $y$ .

For  $L \subseteq \hat{\Sigma}$ ,  $x$  is called minimal (maximal) in  $L$  if for any  $y \in L$ ,  $y \leq x$  ( $x \leq y$ ) implies  $x = y$ .  
 $x$  is shortest (longest) if for any  $y \in L$ ,  
 $|y| \geq |x|$  ( $|y| \leq |x|$ ).

$n^x$  where  $x \in \hat{\Sigma}$  and  $|x| \geq n$ , denotes the  $y$  such that  $y \leq x$  and  $|y| = n$ ; is undefined otherwise.

$x[n]$  where  $x \in \hat{\Sigma}$  and  $|x| \geq n$ , is the  $n$ -th element of  $x$ ; is undefined otherwise.

$a \in x$  is true if there exists  $n$  such that  $a = x[n]$ .

Ordering, well-ordering and lexicographic ordering are as in [Knu] p. 20.

A graph is a pair  $G = (V, E)$ , where  $V$  is a set of elements called vertices and  $E$  is a subset of  $V \times V$  called the set of edges. For any two  $a, b \in V$  whenever we say that  $(a, b) \in E$  we imply  $(b, a) \in E$ . A graph is finite if  $V$  is finite. A path is an ordered set of vertices  $(v_0, \dots, v_n)$  such that for each  $i$  ( $0 \leq i < n$ )  $(v_i, v_{i+1}) \in E$ . A cycle is a path where the first and last vertex coincide. A graph is acyclic if it does not contain any cycles. A self-loop is a cycle of one vertex. A graph is connected if for any two vertices  $v, v'$  there exists a path  $(v, \dots, v')$ . For a graph  $G = (V, E)$ , a subgraph of  $G$  is a  $G' = (V', E')$  where  $V' \subseteq V$  and  $E' \subseteq E \cap (V' \times V')$ . A tree

is a connected, acyclic graph.

## §2.2 List of definitions

### A) Terminology

- affect ... 136
- area ... 79
- area map ... 81
- assignment ... 36
- canonical renaming function ... 120
- characteristic ... 59
- closed area ... 81
- closed segment ... 49
- coloring ... 110
- computation ... 37
- compute step by step the same values (strings) ... 58
- compute step by step the same values (schemas) ... 93
- consistency ... 103
- corresponding element ... 55
- critical set ... 145
- decision-free ... 106
- determinate schema ... 94
- domain variables ... 34
- equivalence of strings ... 44
- equivalence of schemas ... 44
- fetches from ... 35
- finite schema ... 76
- finitely-branching schema ... 75

finitely-branching language ... 78  
free schema ... 103  
function symbol ... 34  
head of segment ... 49  
head of route ... 98  
head of area ... 118  
h-interpretation ... 40  
history ... 39  
history permutation ... 45  
hyperclosed schema ... 155  
hyperclosure ... 156  
incompatibility graph ... 110  
initial memory state ... 37  
interpretation ... 36  
language of interpretation ... 37  
language of schema ... 75  
language-equivalence ... 79  
liberality ... 65  
losslessness ... 65  
memory-reduced schema ... 111  
memory requirements ... 111  
memory state transition function ... 37  
memory states ... 36  
minimal coloring ... 110  
minimal renaming ... 111  
minimum memory ... 111  
number of arguments ... 34

number of outcomes of function ... 34  
number of outcomes of operation ... 35  
number of results ... 34  
occupied variable ... 118  
open area ... 81  
open segment ... 49  
operation ... 34  
parallel program schema ... 75  
prefix-closed ... 78  
program ... 91  
progressive schema ... 159  
prompt schema ... 155  
proper renamings of strings ... 54  
proper renamings of schemas ... 84  
quasi-determinate schema ... 94  
range variables ... 35  
renaming ... 51  
renaming function of string ... 54  
renaming function of schema ... 84  
repetition-free string ... 65  
repetition-free schema ... 103  
restricted schema ... 133  
route ... 79  
schema ... 75  
schema states ... 75  
schema state transition function ... 75  
scope ... 48

- segment ... 49
- segment of a route ... 79
- segment map ... 54
- set of variables of schema ... 111
- similarity of strings ... 45
- similarity of schemas ... 93
- simple test ... 65
- stores in ... 35
- string ... 35
- terminator ... 35
- terminator string ... 35
- term ... 40
- totally-defined schema ... 133
- tree schema of language ... 78
- tree schema of schema ... 79
- tree schema ... 98
- values ... 36
- values fetched, stored ... 37
- value of variable ... 37

## B) Notation

### a) Latin notation

- A ... 35
- Adv<sup>1</sup> ... 136
- Adv ... 145
- Amap ... 81
- Ar ... 83
- Area ... 80

c ... 37  
 $c^H$  ... 41  
Char ... 59  
Comp ... 91  
Crit ... 145  
D ... 34  
 $\mathcal{D}$  ... 34  
F ... 35  
Free ... 118  
 $G_S$  ... 111  
 $h^L$  ... 85  
 $h^Q$  ... 84  
 $h^q$  ... 84  
 $h^\Sigma$  ... 59  
 $g^\Omega$  ... 45  
 $L_I$  ... 37  
 $L_S$  ... 75  
K ... 35  
K ... 34  
Mem ... 111  
Min ... 111  
Pref ... 91  
 $q_0$  ... 75  
 $q_x$  ... 75  
R ... 34  
R ... 34  
Rut ... 79

$S_C$  ... 124  
 $S_{cr}$  ... 128  
 Seg ... 49  
 Smap ... 54  
 S-string ... 91  
 $T_C$  ... 122  
 U ... 36  
 $Ult^1$  ... 136  
 Ult ... 145  
 Var ... 111

b) Greek notation

$\Gamma$  ... 36  
 $\gamma$  ... 39  
 $\delta$  ... 75  
 $\mu$  ... 37  
 $\bar{v}_q$  ... 118  
 $\rho$  ... 137  
 $\Sigma$  ... 35  
 $\phi$  ... 75  
 $\Psi$  ... 36  
 $\Omega$  ... 39  
 $\bar{\Omega}$  ... 39  
 $\Omega^H$  ... 41

c) Special characters

$\equiv$  (as applied to strings) ... 44  
 $\equiv$  (as applied to schemas) ... 93



$\sim$	(as applied to strings) ...	45
$\sim$	(as applied to schemas) ...	93
$\downarrow$	...	54
$\overline{\downarrow}$	...	84
$\leftrightarrow$	(as applied to strings) ...	58
$\leftrightarrow$	(as applied to schemas) ...	91
$\leftrightarrow$	(as applied to languages) ...	85
$\Leftarrow$	(as applied to strings) ...	58
$\Leftarrow$	(as applied to schemas) ...	93
$\simeq$	...	79
/a/	...	58
$/\phi(q)/$	...	136
$\cdot >$	...	118
$\circ$	...	134
$\geq, >$	(as applied to schemas) ...	134

### §2.3 Numbering system.

We refer to definitions, lemmas, properties and theorems as follows. Every reference is preceded by the symbol #. If the reference contains three numbers, these represent in order the chapter, section, number within the section. If the reference contains two numbers, these are the section number and the number within the section, while it is understood that the chapter is the current chapter. If the reference contains only one number, this is a number within the current section.

For figures we use a similar system, except for the

fact that figures are numbered consecutively within a chapter.

For example, Definition 3.2.1 will be referred to as #3.2.1 outside Chapter 3, as #2.1 within Chapter 3, as #1 within Section 3.2.

## CHAPTER 3

### RENAMINGS IN TERMINATOR STRINGS

#### Introduction

Even though our main concerns are properties of schemas, a number of results on schemas can be conveniently approached by first considering the behaviour of computations under some interpretation, and then extending the reasoning to schemas. This chapter is dedicated to such results.

The first section introduces the basic concepts of terminators, strings, interpretations, computations, and histories. The second section introduces the concepts of equivalence and similarity. Section 3 introduces segments, the basic units in which renamings are performed in terminator strings. Section 4 defines proper renamings and introduces some of their "syntactic" properties, i.e. properties that can be detected without any consideration of interpretations. The most important of these properties is the fact that proper renamings preserve the segment structure of the string.

Sections 5 and 6 are the core of the chapter. In section 5 we show that, by properly renaming a computation, we obtain another computation that has the property of computing step by step the same values as the original computation. In section 6 we derive a partial converse of this result, by showing that if two repetition-free computations compute step by step the same values then one is a proper renaming of the other. In other words, if two repetition-free

computations are such that one is an improper renaming of the other, then the two computations do not compute step by step the same values. Thus, in some sense, proper renamings are the only class of renamings in which we are interested.

### §3.1 Terminator strings, interpretations, histories.

The idea of an instruction sequence, as used in the introduction, can be approached by thinking of a recording of the sequence of operations executed during a run of a program. Each element in the sequence tells which operation has been executed, and the outcome of its execution. We call these elements "terminators".

1. Definition. A function symbol is a label  $f$  or  $f_{i,j,k}$  together with three integers:

$D(f) = i$ , a nonnegative integer, is the number of arguments of  $f$ ,

$R(f) = j$ , a nonnegative integer, is the number of results of  $f$ ,

$K(f) = k$ , a positive integer, is the number of outcomes of  $f$ ,

where the following holds:  $R(f) = 0$  implies  $K(f) > 1$  ("do nothing" operations not allowed).

2. Definition. An operation is a triple

$a = (f, (d_1, \dots, d_i), (r_1, \dots, r_j))$  where:  $F(a) = f_{i,j,k}$  is a function symbol;  $D(a) = (d_1, \dots, d_i)$  is an ordered set of  $i$  distinct elements of  $\omega$ , the domain variables of  $a$ ;

$R(a) = (r_1, \dots, r_j)$  is an ordered set of  $j$  distinct elements

of  $\omega$ , the range variables of  $a$ .

With the operation  $a$  are associated:

an integer  $K(a) = k$ , the number of outcomes  
of  $a$ ,

a set  $\Sigma(a) = \{a^0, \dots, a^{k-1}\}$  of symbols called  
terminators of  $a$ .

An operation  $a$  will be written

$(f(d_1, \dots, d_i) \rightarrow (r_1, \dots, r_j))$ . We say that  $a$  fetches from  
 $(d_1, \dots, d_i)$  and stores in  $(r_1, \dots, r_j)$ .

The condition that for all  $a$  any two elements of  $D(a)$  must be distinct is rather inessential, and is only introduced because it allows some simplifications in the proofs.

In the following we assume that  $F$  is a finite set of function symbols and  $A(F)$  (simply written  $A$ , where  $F$  is understood) is the set of all operations whose function symbols are in  $F$ . Let also  $\Sigma(A(F))$  (or simply  $\Sigma$ ) be  $\{a^i: a^i \in \Sigma(a) \text{ for some } a \in A\}$ . Any finite or infinite string over  $\Sigma$  is called a terminator string, or simply a string.

In this chapter, terminator strings will be considered in relation to a special kind of automaton, called "interpretation". For most authors, an interpretation simply fixes the set of possible values of the memory variables, and associates functions on these values with function symbols. We have found it convenient to define an interpretation as an infinite automaton, whose states are all possible memory con-

tents, and which operates as an acceptor of terminator strings. This automaton also incorporates an interpretation in the previously mentioned sense, that we call "assignment".

An interpretation starts with a certain designated memory contents, and for each terminator  $(f(d_1, \dots, d_i) \rightarrow (r_1, \dots, r_j))^k$  in a string it performs the following: it fetches from memory the values contained in  $d_1, \dots, d_i$ ; it computes the outcome according to an assigned function  $\Gamma_f$  and, if this computed outcome is not  $k$ , it rejects the string (this will be expressed by saying that the next memory state is not defined). If instead the computed outcome is  $k$ , an assigned function  $\Psi_f$  is computed on the values fetched, and the resulting values are stored in memory locations  $r_1, \dots, r_j$ , thus causing a change in the memory state.

This is now made more precise.

3. Definition. For  $f \in F$ , an assignment for a function symbol  $f$  in a set of values  $U$  consists of two total functions:

$$\Psi_f: U^{\mathcal{D}(f)} \rightarrow U^{\mathcal{R}(f)}$$

$$\Gamma_f: U^{\mathcal{D}(f)} \rightarrow \{0, \dots, K(f)-1\}$$

Given a set of values  $U$  and an assignment in  $U$  for each  $f \in F$ , an interpretation of  $F$  is a triple  $I = (U^\omega, c_0, \mu)$  where:

$U^\omega$ , the set of all infinite sequences of elements of  $U$  indexed by  $\omega$ , is the set of memory states;

$c_0 \in U^\omega$  is the initial memory state;

$\mu$  is a partial function  $\mu: U^\omega \times \Sigma \rightarrow U^\omega$ , the memory state transition function.

Let  $F(a) = f$ . Then  $(\mu(c, a^k))$  iff  $\Gamma_f(c[D(a)]) = k$  and in this case it is defined to be  $c'$  as follows: for all  $m \in \omega$

$$c'[m] = \begin{cases} c[m] & \text{if } m \notin R(a) \quad (\text{a does not store in } m) \\ \Psi_f(c[D(a)])[s] & \text{if } m = R(a)[s] \quad (\text{a stores in } m). \end{cases}$$

We extend  $\mu$  to a partial function  $\mu: U^\omega \times \Sigma^* \rightarrow U^\omega$  in the obvious way, i.e.

$$\mu(c, \lambda) = c$$

$$\text{and, for } x \in \Sigma^*, \quad \mu(c, xa^k) = \begin{cases} \mu(\mu(c, x), a^k) & \text{if this is defined} \\ \text{undefined otherwise} \end{cases}$$

For  $x \in \Sigma^*$ , let  $c_x = \mu(c_0, x)$ ;  $c_x[m]$  is the value (or content) of variable  $m$  after  $x$ . Note that  $c_x$  is not defined for  $x \in \Sigma^\omega$ . When we talk of the (ordered set) of values fetched by  $x[i]$ , we take this to mean  $c_{i-1}^{x[D(x[i])]}$ , while the (ordered set of) values stored by  $x[i]$  means  $c_x[R(x[i])]$  (thus even if  $x[i] = y[j]$  it does not necessarily follow that the values fetched (or stored) by  $x[i]$  and  $y[j]$  are the same).

The language  $L_I$  of the interpretation  $I$  is the set  $\{x \in \Sigma^*: (c_x)\}$ .  $x \in \hat{\Sigma}$  is an I-computation if for all

$y \leq x, y \in L_I$ .  $x \in \hat{\Sigma}$  is a computation if it is an I-computation for some I.

Wherever we use a symbol such as  $U, c_0, \mu$ , etc., it is understood that we refer to the set of values, initial state, next state function, etc., of the interpretation under consideration at that point. Wherever more than one interpretation is being considered, we shall use subscripts and superscripts to indicate the interpretation to which we are referring: for example,  $U_I^\omega$  will be the set of states of the interpretation I,  $c_x^I = \mu^I(c_0^I, x)$ , and so on. Other similar conventions will be introduced tacitly later.

The reader will have noticed that, while our model can be considered a variety of Karp and Miller's model [K&M], the introduction of function variables in the definition of operations is a step towards models of the kind studied by Paterson [Pat]. This variation of Karp and Miller's model has been introduced to provide a way of interpreting consistently an operation and its renaming with the same function. In Karp and Miller's model, any two distinct operations can be interpreted with distinct functions. In our model, we are able to express the constraint that two operations that are distinct as concerns domain and range variables are required to compute the same function under any interpretation.

The next concept is the "history" of a computation. Following [K&M], in our model we take the results of a computation to be all the values stored by the computation into memory at each step (rather than just the final results as in



[Pat]). The set of values stored by a computation is called the "history" of the computation.

In the following definition, for a given interpretation, take  $\gamma_m$  to be a function  $\gamma_m: \Sigma^* \rightarrow U \cup \{\lambda\}$  as follows:

$$\gamma_m(\lambda) = \lambda$$

$$\gamma_m(xa^j) = \begin{cases} \lambda & \text{if } m \notin R(a) \text{ and } (c_{xa^j}) \\ c_{xa^j}[m] & \text{if } m \in R(a) \text{ and } (c_{xa^j}) \\ \text{undefined otherwise} & \end{cases}$$

In other words,  $\gamma_m(x)$  gives the value stored in  $m$  by the last terminator of string  $x$ . Note that  $\gamma_m(x)$  is not defined for  $x \in \Sigma^{\omega}$ . However for  $x \in \Sigma^*$ ,  $(\gamma_m(x))$  iff  $(c_x[m])$ .

We now define the "history" of a variable in a computation to be a vector where all the successive values stored in the variable are recorded, while the "history" of a computation is a two-dimensional array, whose columns are the histories of the variables.

For  $x \in \hat{\Sigma}$ , take  $\bar{\Omega}_m(x) = (c_0[m], \gamma_m(1^x), \gamma_m(2^x), \dots)$ .  $\bar{\Omega}_m(x)$  is defined iff each one of its elements is defined. If defined,  $\bar{\Omega}_m(x)$  is of the same length as  $x$  if  $x$  is finite, it is infinite otherwise.

4. Definition. For an interpretation  $I$ ,  $m \in \omega$ ,  $x \in \hat{\Sigma}$ , we define:

$\Omega_m(x)$ , the history of  $m$  in computation  $x$  as the sequence obtained by eliminating from  $\bar{\Omega}_m(x)$  all the  $\lambda$  elements.  $\Omega_m(x)$  is defined iff

$\bar{\Omega}_m(x)$  is defined.

$\Omega(x)$ , the history of  $x$  as the infinite sequence  
 $(\Omega_0(x), \Omega_1(x), \Omega_2(x), \dots)$  that is defined iff  
 each one of its elements is defined.

Observe that  $\Omega(x)$  is defined iff for each  $y \leq x$ ,  
 $c_y$  is defined. Thus  $x$  is an I-computation iff  $\Omega(x)$  is  
 defined for I.

We represent  $\Omega(x)$  as a two-dimensional array,  
 whose columns are the  $\bar{\Omega}_m(x)$ . By an element of  $\Omega(x)$  we  
 mean any element in the array, and the element  $\bar{\Omega}_m(x)[n]$  is  
 denoted  $\Omega(x)[n,m]$ .

Among the interpretations, a special role is played  
 by that class of interpretations that have been called "one-  
 to-one" in [K&M], "free" in [Pat], "Herbrand interpretations"  
 in [A,M&P]; we shall call them "h-interpretations".  
 h-interpretations are constructed in such a way that each value  
 stored tells how the value itself has been computed, i.e.  
 values are formulas that specify the sequence of functions  
 that has been computed.

5. Definition. An h-interpretation for a set  $F$  of functions  
 is an interpretation  $H = (U^\omega, c_0, \mu)$  where:

$U$  is a set of strings of symbols called terms defined  
 as follows:

- i) each 'm' for  $m \in \omega$ , is a term;
- ii) if  $f \in F$  and  $t_1, \dots, t_{D(f)}$  are terms, then

' $f^n(t_1, \dots, t_{\mathcal{D}(f)})$ ' is also a term, for  
 $n \in \{1, \dots, \mathcal{R}(f)\}$ .

for all  $f \in F$ :

$$\Psi_f(t_1, \dots, t_{\mathcal{D}(f)})[n] = 'f^n(t_1, \dots, t_{\mathcal{D}(f)})'.$$

$\Gamma_f$  is arbitrary.

$c_0$  is the infinite sequence ('0', '1', '2', ...).

Note that quotes have been used to stress the fact that terms are literally strings of symbols.

Note also that two  $h$ -interpretations can only differ in the choice of  $\Gamma_f$ . From this observation, we get the following: for all  $x \in \Sigma^*$  and  $h$ -interpretations  $H, H'$ ,  $(c_x^H)$  and  $(c_x^{H'})$  implies  $c_x^H = c_x^{H'}$ . We then can uniquely define:

for  $x \in \Sigma^*$ ,  $c_x^H = c_x^H$  for an  $h$ -interpretation such that  $(c_x^H)$ .

for  $x \in \hat{\Sigma}$ ,  $\Omega^H(x) = \Omega^H(x)$  for an  $h$ -interpretation such that  $(\Omega^H(x))$ .

The importance of these definitions is due to the following:

6. Proposition. For all  $x, y \in \hat{\Sigma}$  and all interpretations  $I$ , the following hold:

- A) There exists an  $h$ -interpretation  $H$ , depending only on  $I$ , such that  $x$  is an  $H$ -computation iff  $x$  is an  $I$ -computation, and  $c_x^H[m] = c_y^H[n]$  implies  $c_x^I[m] = c_y^I[n]$ .
- B) If  $(c_x^I)$  then  $(c_x^H)$  and if  $(\Omega^I(x))$  then  $(\Omega^H(x))$ .

- C)  $(c_x^I), (c_y^I)$  and  $c_x^H[m] = c_y^H[n]$  imply  $c_x^I[m] = c_y^I[n]$ .  
 D)  $(\Omega^I(x)[n,m]), (\Omega^I(y)[j,i])$  and  $\Omega^H(x)[n,m] = \Omega^H(y)[j,i]$   
 imply  $\Omega^I(x)[n,m] = \Omega^I(y)[j,i]$ .

Proof. The proof of A) is well-known (see [Pat], [K&M]), and the reader may refer to #7 for an example showing how for any interpretation I it is possible to construct an h-interpretation that "simulates" I. B), C), and D) follow.

7. Example. Consider a set of functions  $F = \{f, g, h\}$ , where:

$$\begin{array}{lll} \mathcal{D}(f) = 2 & R(f) = 0 & K(f) = 2 \\ \mathcal{D}(g) = 2 & R(g) = 1 & K(g) = 1 \\ \mathcal{D}(h) = 1 & R(h) = 1 & K(h) = 1 \end{array}$$

Let

$$x = (f(1,2))^0 (g(1,2) \rightarrow 1)^0 (h(0) \rightarrow 0)^0 (f(1,2))^0 (g(1,2) \rightarrow 1)^0 (h(0) \rightarrow 0)^0 (f(1,2))$$

Let I be an interpretation as follows:

U is the set of the integers

$c_0$  is defined as follows:

$$c_0[1] = 6, c_0[2] = 3, c_0[i] = 0 \text{ for } i \neq 1, 2$$

$\Psi_f$  is the function with empty result;

$$\Gamma_f(m, n) = \begin{cases} 1 & \text{if } m < n \\ 0 & \text{otherwise} \end{cases}$$

$\Psi_g(m, n) = m - n$ , and  $\Gamma_g$  is constantly 0.

$\Psi_h(m) = m + 1$ , and  $\Gamma_h$  is constantly 0.

Then  $\Omega(x)$  can be represented as follows:

variables	{	0	1	2	3	4		
		0	6	3	0	0	...	
successive		1	3					
values		2	0					

We see that  $\Omega(x)$  is defined or, equivalently, that  $x$  is accepted by the interpretation  $I$ .  $x$  would not have been accepted if its first terminator had been  $(f(1,2))^1$ , as can be seen from the definition of  $\Gamma_f$ . We now give an  $h$ -interpretation  $H$  for  $F$ . The only thing that we are free to choose is  $\Gamma_f$ , and we choose it as follows:

$$\Gamma_f(t_1, t_2) = \begin{cases} 1 & \text{if } v(t_1) < v(t_2) \\ 0 & \text{otherwise} \end{cases}$$

where, if  $t, t_1, t_2$  are terms,  $v(t)$  is defined as follows:

$$v('1') = 6, \quad v('2') = 3, \quad v('i') = 0 \quad \text{for } i \neq 1, 2$$

$$v('g^1(t_1, t_2)') = v(t_1) - v(t_2)$$

$$v('h^1(t)') = v(t) + 1$$

Then  $\Omega(x)$  is:

0	1	2					
0	1	2					...
$h^1(0)$	$g^1(1, 2)$						
$h^1(h^1(0))$	$g^1(g^1(1, 2), 2)$						

The reader will note that we have chosen  $H$  so that  $L_H = L_I$ .

Before closing, we want to attract the attention of the reader to the fact that, if a computation  $x$  has a prefix  $y$  that is in the language of an interpretation  $I$ , and a prefix  $z > y$  that is not in that language, then of course  $x$  is not an  $I$ -computation and the history of  $x$  is undefined, even though the history of  $y$  is defined.

### §3.2 Equivalence and similarity.

We have already noted that in our model we take the results of a computation to be what we have called the history of the computation. Equivalence is defined accordingly: two computations are equivalent if they have the same histories.

1. Definition. For an interpretation  $I$ , we say that  $x, y \in \hat{\Sigma}$  are  $I$ -equivalent, written  $x \equiv_I y$ , if  $\Omega(x) = \Omega(y)$ ,  $x, y$  are equivalent, written  $x \equiv y$ , if they are  $I$ -equivalent for all interpretations  $I$ .

This very strict concept of equivalence, requiring that the two computations compute the same intermediate values and store them in the same memory variables in the same order, can be generalized in a number of ways. In our work, we shall concentrate on a generalization obtained by relaxing the constraint that the two computations must store the same values in the same variables. Two computations are said to be similar if they compute the same values, independently of where they store them. This amounts to saying that the history array of

one computation is a permutation of the history array of the other. Clearly, equivalence implies similarity.

2. Definition. For an interpretation  $I$ , we say that  $x$  and  $y$  in  $\hat{\Sigma}$  are  $I$ -similar, written  $x \sim_I y$ , if neither of them is an  $I$ -computation or they are both  $I$ -computations and there exists a bijection  $g^\Omega: \omega \times \omega \rightarrow \omega \times \omega$  (called history permutation) such that

$$\Omega(x)[m,n] = \Omega(y)[g^\Omega(m,n)].$$

$x$  and  $y$  in  $\hat{\Sigma}$  are similar (written  $x \sim y$ ) if they are  $I$ -similar for all  $I$ .

3. Example. Consider the string  $x$  and the interpretation  $I$  introduced in #1.7. The following string  $y$  is  $I$ -similar to  $x$ :

$$y = (h(0) \rightarrow 0)^0 (h(0) \rightarrow 0)^0 (h(0) \rightarrow 0)^0.$$

In fact,  $\Omega(y)$  is as follows:

	0	1	2	3	
	0	6	3	0	...
1					
2					
3					

We now show that for similar computations a history permutation  $g^\Omega$  can be chosen independently of the interpretation.

4. Proposition. If two computations  $x, y$  are similar then there exists a bijection  $g^\Omega: \omega \times \omega \rightarrow \omega \times \omega$  such that for all interpretations  $I$ ,

$$(*) \quad \Omega(x)[m, n] = \Omega(y)[g^\Omega(m, n)],$$

whenever both histories are defined.

Proof. If  $x$  and  $y$  are  $I$ -computations for some  $I$ , then by #1.6.A) there exists an  $h$ -interpretation  $H$  such that  $x, y$  are  $H$ -computations. By definition  $\Omega^H(x) = \Omega^H(x)$  and  $\Omega^H(y) = \Omega^H(y)$ . Take a bijection  $g_H^\Omega$  which satisfies  $(*)$  for  $H$ . Consider any interpretation  $I$  for which  $x$  and  $y$  are  $I$ -computations. Then  $(\Omega^I(x))$  and  $(\Omega^I(y))$ . By #1.6.D), the bijection  $g_H^\Omega$  satisfies  $(*)$  for  $I$  and is therefore the desired bijection.

The following proposition states that in some sense we only need to consider the behaviour of strings under  $h$ -interpretations.

5. Proposition.  $x, y \in \hat{\Sigma}$  are similar iff they are  $H$ -similar for all  $h$ -interpretations  $H$ .

Proof. Suppose that  $x$  and  $y$  are  $H$ -similar for all  $h$ -interpretations  $H$ . If neither  $x$  nor  $y$  are  $H$ -computations for any  $h$ -interpretation  $H$  then by #1.6.A) they cannot be  $I$ -computations for any interpretation  $I$ , hence they are similar. Otherwise there exists an  $h$ -interpretation  $J$  such that both  $x$  and  $y$  are  $J$ -computations. Therefore a bijection  $g^\Omega$  exists between  $\Omega^J(x) = \Omega^H(x)$  and  $\Omega^J(y) = \Omega^H(y)$ . If  $I$  is



any interpretation such that  $(\Omega^I(x))$  and  $(\Omega^I(y))$  then by #1.6.D)  $x \sim_I y$ . Finally suppose  $(\Omega^I(x))$  but not  $(\Omega^I(y))$ . By #1.6.A) there exists an h-interpretation G such that  $(\Omega^G(x))$  but not  $(\Omega^G(y))$ . But this contradicts the fact that x and y are G-similar.

As a consequence of these results, in order to find out whether x and y are similar, it is sufficient to consider the sets of h-interpretations for which x and y are computations, and the arrays  $\Omega^H(x)$ ,  $\Omega^H(y)$ .

6. Corollary. Two computations x,y are similar iff for all h-interpretations H, x is an H-computation iff y is an H-computation, and a history permutation  $g^\Omega$  as in #2 exists for  $\Omega^H(x)$  and  $\Omega^H(y)$ .

One verifies that similarity and equivalence are equivalence relations.

### §3.3 Segments

In this section, we shall introduce the concept of a segment of a variable in a terminator string. This corresponds to two related concepts:

the concept of a variable being "occupied" at a certain point in a computation, since the value contained in it is to be fetched by some successive computation step;

the concept of two variable names being "bound together" in a computation, in the sense that if one of them is changed, the other must be changed in the same way.

Examples of the latter are a store on a memory variable and a fetch from the same variable such that between the store and the fetch no new value is stored in the variable; or two fetches from the same variable that are not separated by any store on the variable.

Before introducing an example, we agree that in this chapter we shall omit the indication of outcomes in terminator strings when they are not necessary.

Consider the following string:

$$x = (f_1(0) \rightarrow \underline{1, 3}) (f_2(2, 1) \rightarrow 2) (f_3(2, 3) \rightarrow 2) (f_4(3, 1) \rightarrow 5, 6)$$

$\bar{0}$                        $\bar{1}$                        $\bar{2}$                        $\bar{3}$                        $\bar{4}$

We see that variable 3 is occupied by a value in the underlined portion of  $x$ , and that all occurrences of the name 3 in the portion are bound together. The fact that variable 3 is occupied from the end of the first operation to the beginning of the fourth will be expressed by saying that  $\bar{1}, \bar{2}, \bar{3}$  are in the same segment of 3 in  $x$ , where  $\bar{i}$  can be thought as referring to the interval between terminator  $i$  and terminator  $i+1$ . We call scope the set  $\{\bar{i} : i \text{ is a nonnegative integer}\}$ .

We shall now define these concepts formally. First, we extend the functions  $D, R, F$  to symbols in  $\Sigma$  in the following natural way: for  $a^i \in \Sigma$ ,  $D(a^i) = D(a)$  etc. Also,

$$K(a^i) = i.$$

1. Definition. For  $x \in \hat{\Sigma}$ , a segment  $M$  of  $m \in \omega$  in  $x$  is a maximal (finite or infinite) set of consecutive scope elements:  $M = \{\bar{i}, \overline{i+1}, \overline{i+2}, \dots\}$  such that:

A)  $m \notin R(x[j])$  for  $j > i$ ,  $j \in M$  and

either

B)  $i > 0$ ,  $m \in R(x[i])$  and for each  $\bar{h} \in M$ ,  $h > i$ , there exists  $k \geq h$ ,  $k \in M$  such that  $m \in D(x[k+1])$ .

Such a segment is called closed.

or

C)  $i = 0$  and for each  $\bar{h} \in M$  there exists  $k \geq h$ ,  $k \in M$  such that  $m \in D(x[k+1])$ . Such a segment is called

open.

$\bar{i}$  is the head of segment  $M$ .

$\text{Seg}_m(x)$  is the set of all segments of  $m$  in  $x$ .

The reader will verify that, for each  $m \in \omega$ ,  $\text{Seg}_m(x)$  is either empty, or a set of nonintersecting subsets of the scope. Each set  $M$  in  $\text{Seg}_m(x)$  belongs to one of the following types:

A) Trivial segments, where some value is stored in the variable  $m$ , and is never subsequently fetched.

These segments consist of a single element of the scope. As an example, consider the segment  $(\bar{3})$  of 2 in  $x$  above.

B) Segments in which  $m$  is assigned a value with the

initial memory assignment and which contain a finite number of fetches from  $m$  up to a final fetch. The latter is a fetch that either is the last appearance of  $m$  in the string, or is such that the next appearance of  $m$  in the string is in a store. These segments consist of a prefix of the scope. An example is the segment  $(\bar{0}, \bar{1})$  of 2 in  $x$ .

- C) Segments where  $m$  is initialized as in B, but where there is no last fetch, since the same value is fetched from  $m$  an infinite number of times. These segments consist of the whole scope of the string. It is clear that they can only occur in infinite strings, and thus there is no example of them in  $x$ .
- D) Segments where  $m$  is initialized by a store, and contain a finite number of fetches from  $m$  up to a final fetch. These segments consist of a finite, proper subset of the scope. An example is the segment  $(\bar{1}, \bar{2}, \bar{3})$  of 3 or 1 in  $x$ .
- E) Segments where  $m$  is initialized as in D, but there is no last fetch, since, as in C, we keep fetching from  $m$  an infinite number of times. These segments consist of a suffix of the scope.

Segments of the types B and C above, where the variable is initialized with the initial memory assignment, contain  $\bar{0}$  and are open. Segments of the types A, D and E do not contain  $\bar{0}$  and are closed.

A complete tabulation of the segments in  $x$  above is:

Segments of 0	: $(\bar{0})$	Type	B
of 1	: $(\bar{1}, \bar{2}, \bar{3})$		D
of 2	: $(\bar{0}, \bar{1}), (\bar{2}), (\bar{3})$		B, D, A
of 3	: $(\bar{1}, \bar{2}, \bar{3})$		D
of 5	: $(\bar{4})$		A
of 6	: $(\bar{4})$		A

### §3.4 Renamings.

Let  $a^i, b^j$  be terminators:  $a^i$  is a renaming of  $b^j$  if  $i = j$  and  $F(a) = F(b)$ . Let  $x, y \in \hat{\Sigma}$ :  $x$  is a renaming of  $y$  if  $|x| = |y|$  and for all  $i$  such that  $(x[i])$ ,  $x[i]$  is a renaming of  $y[i]$ . In general, the histories of  $y$  may be totally different from those of  $x$ . However, we shall show that, if the renaming is performed according to certain rules,  $y$  computes step by step the same results as  $x$ .

To understand these rules, consider again the string  $x$  presented in #1.7:

$x = (f(1,2)) (g(1,2) \rightarrow \underline{1}) (h(0) \rightarrow 0) (f(1,2) \rightarrow 0) (g(1,2) \rightarrow 1) \dots$

and consider the string  $y$  obtained by replacing 1 by 3 in the underlined segment of  $x$ :

$y = (f(1,2)) (g(1,2) \rightarrow 3) (h(0) \rightarrow 0) (f(3,2) \rightarrow 0) (g(3,2) \rightarrow 1) \dots$

The strings  $x, y$  perform the same sequence of functions on the same data in the same order, and store the same values in the same order: they only differ in their use of memory

variables. The same would not have been true if  $l$  had not been replaced consistently in all its appearances in the segment.

Also, consider what would have happened if  $l$  had been replaced by, say,  $0$ , giving  $z$ :

$$z = (f(1,2))(g(1,2) \rightarrow 0)(h(0) \rightarrow 0)(f(0,2) \rightarrow 0)(g(0,2) \rightarrow 0) \dots$$

The third terminator in  $z$  now fetches the value of  $0$  computed by the second terminator, rather than fetching the value of  $0$  fixed by the initial memory state, as in  $x$ . Under an  $H$ -interpretation, that terminator would store ' $h^1(0)$ ' in  $x$ , ' $h^1(g^1(1,2))$ ' in  $z$ , and thus in no possible sense could we say that  $x$  and  $z$  compute step by step the same results, as  $x$  and  $y$  do.

Similarly, we could not rename a variable in an open segment, since the values of variables in open segments are fixed by the initial memory state, i.e. by the interpretation.

We are now ready to understand the three rules for those renamings that we shall call "proper":

- 1) the renaming must be consistent over a segment;
- 2) the name of a variable  $m$  cannot be changed in an open segment of  $m$ ;
- 3) if a segment  $M$  of  $m$  intersects a segment  $N$  of  $n$ ,  $m$  and  $n$  cannot be identified in  $M, N$  by the renaming.

Rule 2) deserves some further consideration.

Assume that a computation  $y$  is obtained from a computation  $x$  by a renaming that abides by 1) and 3), but not 2) :

$y$  does not, in general, have the property of computing step by step the same values as  $x$ . However, given any interpretation  $I$ , one can find an interpretation  $I'$  such that  $y$  under  $I'$  computes step by step the same values as  $x$  under  $I$ . Assume for example that  $y$  has been obtained from  $x$  by renaming consistently some variable  $m$  as  $n$  in some open segment of  $m$  in  $x$  that does not intersect any segment of  $n$ .  $I'$  can be obtained by modifying  $I$  so that  $I'$  assigns to  $n$  the same initial value as  $I$  assigns to  $m$ . This example shows that, to handle such renamings, we would have to separate the initial memory state from the other elements of the interpretation, and introduce the concept of permutation of values in the initial memory state. However, for any renaming  $y$  of a string  $x$  that satisfies 1) and 3) there exists a string  $y'$  that can be obtained from  $x$  by a proper renaming, and that can be obtained from  $y$  by a simple one-to-one replacement of names over the whole string. Therefore, there is no loss of generality if condition 2) is added.

In this section, we shall define proper renamings and show some of their basic properties. It is especially important to note that they preserve the segment structure of the string.

We define a renaming as a global procedure that operates on the whole string. A renaming is performed according to a renaming function, that tells which variables get which names in which segments. This function is defined on the list of the segments of all the variables in the string, the seg-

ment map of the string: for  $x \in \hat{\Sigma}$ ,  $\text{Smap}(x)$ , the segment map of  $x$  is the set  $\{(M,m) : M \in \text{Seg}_m(x)\}$ .

1. Definition. A renaming function of  $x$  is a function  $v: \text{Smap}(x) \rightarrow \omega$ , satisfying the following conditions. For all  $(M,m), (N,n) \in \text{Smap}(x)$ :

- A)  $v(M,m) = m$  for every open segment  $M$  of  $m$  in  $x$ ;
- B)  $v(M,m) \neq v(N,n)$  whenever  $M \cap N \neq \emptyset$  and  $m \neq n$ .

Given a renaming function, we use it to perform renamings as follows: if  $v(M,m) = n$  then  $m$  is replaced by  $n$  everywhere in  $M$ , as precisely specified by the following:

2. Definition. (Renaming rule) Let  $x, y \in \hat{\Sigma}$  be such that  $y$  is a renaming of  $x$ . We say that  $y$  is the proper renaming of  $x$  for the renaming function  $v$  and we write  $x \stackrel{v}{\sim} y$  if the following is true for all  $i, j$ :

- A) if  $R(x[i])[j] = m$  and  $\bar{i}$  is in a segment  $M$  of  $m$  such that  $v(M,m) = n$  then  $R(y[i])[j] = n$ .
- B) if  $D(x[i])[j] = m$  and  $\overline{i-1}$  is in a segment  $M$  of  $m$  such that  $v(M,m) = n$  then  $D(y[i])[j] = n$ .

3. Example. We represent  $x$  and the scope as follows:

$(f(1,2)) (g(1,2) \rightarrow 1) (h(0) \rightarrow 0) (f(1,2)) (g(1,2) \rightarrow 1) (h(0) \rightarrow 0) (f(1,2))$   
 $\bar{0} \quad \bar{1} \quad \quad \bar{2} \quad \quad \bar{3} \quad \quad \bar{4} \quad \quad \bar{5} \quad \quad \bar{6} \quad \quad \bar{7}$

We proceed to compute  $\text{Smap}(x)$  and we define a renaming function as follows:



$$\begin{aligned}
v((\bar{0}, \bar{1}, \bar{2}), 0) &= 0 & v((\bar{3}, \bar{4}, \bar{5}), 0) &= 3 & v((\bar{6}), 0) &= 4 \\
v((\bar{0}, \bar{1}), 1) &= 1 & v((\bar{2}, \bar{3}, \bar{4}), 1) &= 5 & v((\bar{5}, \bar{6}), 1) &= 6 \\
v((\bar{0}, \bar{1}, \bar{2}, \bar{3}, \bar{4}, \bar{5}, \bar{6}), 2) &= 2
\end{aligned}$$

Following #2 we obtain the  $y$  such that  $x \stackrel{v}{\sim} y$ :

$$(f(1,2))(g(1,2) \rightarrow 5)(h(0) \rightarrow 3)(f(5,2))(g(5,2) \rightarrow 6)(h(3) \rightarrow 4)(f(6,2))$$

Note that in this renaming we have used as many different names as possible.

A fact that is not shown in this example is that some segments could well be segments of more than one variable.

The reader will easily verify that for all  $x \in \hat{\Sigma}$  there exists a unique  $y \in \hat{\Sigma}$  such that  $x \stackrel{v}{\sim} y$ . The only thing that needs to be checked is that #1.2 cannot be violated by identifying two distinct elements of the domain and/or range locations of some operation: this is not possible because of #1.B.

We shall now prove that, if  $y$  is obtained by properly renaming  $x$ , then  $y$  has the same segment map as  $x$ , up to renaming. For example, the segment map of  $y$  in #3 is:

$$\begin{aligned}
((\bar{0}, \bar{1}, \bar{2}), 0) & & ((\bar{3}, \bar{4}, \bar{5}), 3) & & ((\bar{6}), 4) \\
((\bar{0}, \bar{1}), 1) & & ((\bar{2}, \bar{3}, \bar{4}), 5) & & ((\bar{5}, \bar{6}), 6) \\
((\bar{0}, \bar{1}, \bar{2}, \bar{3}, \bar{4}, \bar{5}, \bar{6}), 2)
\end{aligned}$$

We introduce the following terminology: let  $a$  be a renaming of  $b$ ,  $D(a)[k] = m$ ,  $D(b)[k] = n$  for some  $k$ : then  $n$  is said to be the corresponding element in  $D(b)$  of

$m$  in  $D(a)$ . Similarly for range variables.

4. Proposition. (Proper renamings preserve segment maps).

Let  $x, y \in \hat{\Sigma}$ ,  $x \preceq y$ . Then there exists a bijection

$h: \text{Smap}(x) \rightarrow \text{Smap}(y)$  such that for all  $(M, m) \in \text{Smap}(x)$ ,  
 $h(M, m) = (M, n)$  iff  $v(M, m) = n$ .

Proof. We first prove:

(\*) if  $M \in \text{Seg}_m(x)$ , and  $v(M, m) = n$ , then  $M \in \text{Seg}_n(y)$ .

Suppose that  $M$  is closed, and let  $\bar{i}$  be its head.

By the renaming rule,  $n \in R(y[i])$  and also for each  $\bar{j} \in M$  such that  $m \in D(x[j+1])$ ,  $n \in D(y[j+1])$ . Suppose that for some  $\bar{k} \in M$ ,  $k \neq i$ ,  $n \in R(y[k])$ . Then if the corresponding element in  $R(x[k])$  is  $e$ ,  $\bar{k}$  is in some segment  $E$  of  $e$  in  $x$ . Hence  $E \cap M \neq \emptyset$ , and  $v(E, e) = n = v(M, m)$ , contradicting #1.B). Now  $M$  satisfies #3.1.A) and B) with respect to  $n$  and hence is contained in some closed segment  $N$  of  $n$  in  $y$  whose head is  $\bar{i}$ .

Suppose that  $M$  is open. By the renaming rule for each  $\bar{j} \in M$  such that  $m \in D(x[j+1])$  we have  $n \in D(y[j+1])$ . By the argument used previously  $n \in R(y[k])$  is false for all  $\bar{k} \in M$ . Hence  $M$  satisfies #3.1.A) and C) with respect to  $n$  and is contained in some open segment  $N$  of  $n$  in  $y$ .

Suppose in either case ( $M$  open or closed), that  $M \not\subseteq N$ : this implies that  $M$  is finite. Let  $M = \{\bar{i}, \dots, \bar{t}\}$ . If  $\bar{j} \in N - M$  then by #3.1.B) and C) there exists  $k \geq j$  such that  $\bar{k} \in N$ ,  $n \in D(y[k+1])$ . Let the corresponding element in  $D(x[k+1])$  be  $e$ . Then  $\bar{k}$  is in some segment  $E$  of  $e$

in  $x$  and one of the following must be true:

- i) There exists  $s$  such that  $t < s \leq k$ ,  $e \in R(x[s])$  and  $\bar{s}$  and  $\bar{k}$  are in the same segment of  $e$ . Then by #2.A)  $n \in R(y[s])$  contradicting the fact that  $M \not\subseteq N$ .
- ii)  $t \in M \cap E$  in  $x$ . However, this is impossible by #1.B) since we have:  $v(M,m) = v(E,e) = n$ .

This proves (\*).

To prove the Proposition, for  $(M,m) \in \text{Smap}(x)$  define  $h(M,m) = (M,n)$  iff  $v(M,m) = n$ . By (\*)  $(M,n) \in \text{Smap}(y)$ . To show that  $h$  is a bijection we must only verify that if  $m \neq k$  then  $h(M,m) \neq h(M,k)$  (since obviously  $h(M,m) \neq h(J,j)$  if  $M \neq J$ ). Suppose indeed  $h(M,m) = h(M,k)$ . Since  $M \cap M \neq \emptyset$  we must have  $v(M,m) \neq v(M,k)$ , a contradiction.

At this point, it is not difficult to prove two more interesting properties of proper renamings.

### 5. Corollary.

- A) (Proper renamings are closed under composition) For  $x, y, z \in \hat{\Sigma}$  if there exist renaming functions  $v, v'$  such that  $x \xrightarrow{v} y \xrightarrow{v'} z$  then there exists a renaming function  $v''$  such that  $x \xrightarrow{v''} z$ .
- B) (Proper renamings have inverses) For  $x, y \in \hat{\Sigma}$ , there exists  $v$  such that  $x \xrightarrow{v} y$  iff there exists  $v^{-1}$  such that  $y \xrightarrow{v^{-1}} x$ .

Proof.

- A) If  $v(M,m) = n$ ,  $v'(M,n) = r$  then take  $v''(M,m) = r$ . By #4 one easily verifies that  $v''$  is a renaming function such that  $x \stackrel{v''}{\sim} z$ .
- B) For all  $(M,m) \in \text{Smap}(x)$ , if  $v(M,m) = n$ , take  $v^-(M,n) = m$ .

The latter result justifies the introduction of the following notation:  $x \leftrightarrow y$  iff there exists a renaming function  $v$  such that  $x \stackrel{v}{\sim} y$ ;  $x \not\leftrightarrow y$  otherwise. Note that the relation  $\leftrightarrow$  is an equivalence relation.

### §3.5 Proper renamings and values computed.

In this section we shall prove that two computations one of which is a proper renaming of the other have the property of computing step by step the same values.

1. Definition. We say that  $x, y \in \hat{\Sigma}$  compute step by step the same values for the interpretation I, written  $x \stackrel{I}{\sim} y$ , if for all  $i \in \omega$ ,  ${}_i x \stackrel{I}{\sim} {}_i y$ . We say that  $x, y$  compute step by step the same values, written  $x \approx y$ , if they compute step by step the same values for all interpretations I.

It is easily seen that  $x \approx y$  implies  $x \sim y$ .

We now need some further notation. For an operation  $a$ , let  $/a/ = (F(a), D(a))$ . If  $a = (f(d_1, \dots, d_i) \rightarrow (r_1, \dots, r_j))$ , we write  $/a/ = f(d_1, \dots, d_i)$ .

2. Definition. Let  $x \in \hat{\Sigma}$ ,  $x[i] = a^k$ . The characteristic of the  $i$ -th terminator in  $x$ , in symbols  $\text{Char}(x,i)$ , is the triple  $(F(a), c_{i-1}^H x[D(a)], k)$ . If  $F(a) = f$  and  $c_{i-1}^H x[D(a)] = (t_1, \dots, t_n)$ , then we write  $\text{Char}(x,i) = f(t_1, \dots, t_n)^k$ .

We then have:

3. Proposition. Let  $x, y \in \hat{\Sigma}$ . If there exists a bijection  $h^\Sigma: \omega \rightarrow \omega$  such that for all  $i \in \omega$ ,  $\text{Char}(x,i) = \text{Char}(y, h^\Sigma(i))$  then  $x \sim y$ .

Proof. We shall use the characterization of similarity in terms of  $h$ -interpretations introduced in #2.6.

We first prove that if  $h^\Sigma$  exists then

(\*) for all  $h$ -interpretations  $H$ ,  $x$  is an  $H$ -computation iff  $y$  is an  $H$ -computation.

Assume that  $x$  is not an  $H$ -computation for  $H$ . This means that there exists  $i$  such that  $\text{Char}(x,i)$  is  $f(t_1, \dots, t_n)^k$  but computation of the function  $\Gamma_f$  on  $(t_1, \dots, t_n)$  does not give outcome  $k$ . However, by the existence of  $h^\Sigma$ , there must also exist  $y[j]$  such that  $\text{Char}(y,j) = f(t_1, \dots, t_n)^k$ . Hence,  $y$  is not an  $H$ -computation. The same argument holds in the direction from  $y$  to  $x$ , so (\*) is true.

Next, we show the existence of the bijection  $g^\Omega$ .

If  $h^\Sigma(i) = j$  then  $x[i]$  and  $y[j]$  fetch the same values under an  $h$ -interpretation, and then by #1.5 also store the

same values. Since  $h^\Sigma$  is a bijection,  $g^\Omega$  clearly exists.

The following is then immediate:

4. Corollary. Let  $x, y \in \hat{\Sigma}$ . If for all  $i \in \omega$ ,  $\text{Char}(x, i) = \text{Char}(y, i)$  then  $x \simeq y$  (and  $x \sim y$ ).

Hence, to prove that properly renamed strings compute step by step the same values, we only need to prove that corresponding terminators in the two strings have the same characteristics. To do this, we use the following result. If  $x$  is a proper renaming of  $y$ , then the values that are in the occupied variables of memory at a certain point of the computation of  $x$ , are also in memory at the corresponding point of the computation of  $y$ . Furthermore, corresponding operations in  $x$  and  $y$  fetch the same ordered sets of values.

5. Theorem. (Proper renamings preserve memory values and values fetched). Let  $x, y \in \hat{\Sigma}$  be such that  $x \simeq y$ . Then for all  $j \in \omega$  the following hold:

A) For all  $k \in \omega$  such that  $\bar{j} \in K$ , for some  $K \in \text{Seg}_k(x)$  such that  $v(K, k) = k'$ ,  $c_{j^x}[k] = c_{j^y}[k']$ .

B)  $c_{j^x}[D(x[j+1])] = c_{j^y}[D(y[j+1])]$ .

Proof. By induction on  $j$ . We prove simultaneously A) and B).

Induction basis. If  $j = 0$ ,  $c_0[k] = c_0[k']$  since

$k = k'$  (the variable  $k$  cannot be renamed in an open segment of  $k$ ), and the interpretation is the same in both cases. This proves A). Also,  $D(x[1]) = D(y[1])$  and B) follows by the same reasoning.

Induction hypothesis. Assume that A) and B) hold for  $j$ .

Induction step. We first show that

$$(*) \quad (c_{j+1}^x) \text{ iff } (c_{j+1}^y)$$

By the induction hypothesis,  $(c_j^x) \text{ iff } (c_j^y)$ . Thus we must show that

$$(**) \quad (\mu(c_j^x, x[j+1])) \text{ iff } (\mu(c_j^y, y[j+1])).$$

Since  $x[j+1] \neq y[j+1]$ , both terminators have the same function symbol  $f$  and the same outcome. Since  $\Gamma_f$  in any interpretation depends only on the ordered sets of values fetched and these are identical by induction hypothesis B), the outcomes computed by  $\Gamma_f$  for  $x[j+1]$  and  $y[j+1]$  are identical. Thus (\*\*) follows by the definition of  $\mu$ , and so (\*) is proved.

Returning to the proof of the theorem, if  $c_{j+1}^x$  and  $c_{j+1}^y$  are undefined the theorem holds. Otherwise they are both defined. Consider any  $k \in \omega$  such that  $\overline{j+1} \in K$ , where  $K \in \text{Seg}_k(x)$ . If  $x[j+1]$  does not store in  $k$ , then  $c_j^x[k] = c_{j+1}^x[k]$  and  $\overline{j} \in K$ . By the induction hypothesis,  $c_j^x[k] = c_j^y[k']$ . If  $c_{j+1}^y[k'] = c_j^y[k']$  then  $y[j+1]$  stores in  $k'$  showing that  $\overline{j+1}$  is the head of a segment of  $k'$  in  $y$ . This contradicts #4.4, and  $y[j+1]$  does not

store in  $k'$ . If  $x[j+1]$  stores in  $k$  then  $\overline{j+1}$  is the head of  $K$  and  $y[j+1]$  stores in  $k'$ . By induction hypothesis B)  $x[j+1]$  and  $y[j+1]$  fetch the same values. By definition of  $\Psi_f$  they both store the same values. This proves the induction step of A).

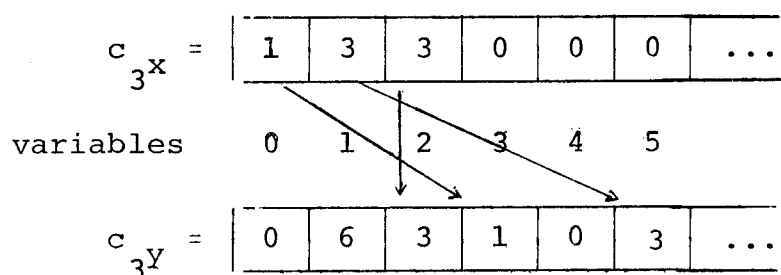
Concerning the induction step of B), consider that for any  $k \in D(x[j+2])$ ,  $\overline{j+1} \in K$  for some  $K \in \text{Seg}_k(x)$ . If  $k'$  is the corresponding element in  $D(y[j+2])$  then  $v(K, k) = k'$ . Hence, the result follows by the induction step of A).

6. Example. Let us see what happens with the strings  $x, y$  presented in #4.3:

$x = (f(1,2)) (g(1,2) \rightarrow 1) (h(0) \rightarrow 0) (f(1,2)) (g(1,2) \rightarrow 1) (h(0) \rightarrow 0) (f(1,2))$

$y = (f(1,2)) (g(1,2) \rightarrow 5) (h(0) \rightarrow 3) (f(5,2)) (g(5,2) \rightarrow 6) (h(3) \rightarrow 4) (f(6,2))$

and consider the memory contents after the third terminator under the interpretation presented in the first part of #1.7:



Now,  $\overline{3}$  is:

in a segment of  $\overline{2}$  where  $\overline{2}$  has been identically renamed,

and in fact  $c_{3^x}[\overline{2}] = c_{3^y}[\overline{2}]$



in a segment of 0 where 0 has been renamed as 3,  
and in fact  $c_{3^x}^{[0]} = c_{3^y}^{[3]}$

in a segment of 1 where 1 has been renamed as 5,  
and in fact  $c_{3^x}^{[1]} = c_{3^y}^{[5]}$ .

Values of those variables such that  $\bar{3}$  is not in any of their segments may or may not be in  $c_{3^y}$ . The whereabouts of those values are not of interest to us, because they are not needed at this point.

The following is a consequence of #5:

7. Proposition. (Proper renamings preserve characteristics)

Let  $x \leftrightarrow y$ . Then for all  $j \in \omega$ ,  $\text{Char}(x, j) = \text{Char}(y, j)$ .

Proof. Assume that  $\text{Char}(x, j)$  is not defined. Then for no h-interpretation  $(c_{j-1^x}^{[D(x[j])]})$ . Assume that, on the contrary,  $(\text{Char}(y, j))$ . Then  $(c_{j-1^y}^{[D(y[j])]})$  for some h-interpretation, and by #5.B for the same h-interpretation  $(c_{j-1^x}^{[D(x[j])]})$ , a contradiction. Therefore,  $\text{Char}(y, j)$  is also not defined. By the same reasoning, if  $(\text{Char}(x, j))$  then  $(\text{Char}(y, j))$ .

Assume that they are both defined, and let  $\text{Char}(x, j) = f(t_1, \dots, t_n)^k$ ,  $\text{Char}(y, j) = g(t'_1, \dots, t'_m)^h$ . Since  $x[j] \leftrightarrow y[j]$  then  $f = g$ ,  $k = h$ ,  $n = m$ . Now, by #5.B)  $c_{j-1^x}^{[D(x[j])]} = c_{j-1^y}^{[D(y[j])]}$  for any h-interpretation

such that  $(c_{j-1}^x)$  and  $(c_{j-1}^y)$ , hence  $(t_1, \dots, t_n) = (t_1', \dots, t_m')$ .

The main result of this section follows then without difficulty:

8. Theorem. (Properly renamed strings compute step by step the same values).  $x \leftrightarrow y$  implies  $x \simeq y$  (and  $x \sim y$ ).

Proof. #7 and #4.

9. Example. Consider  $x$  and  $y$  presented in #4.3 and #6, under the interpretation defined in the first part of #1.7. We show the ordered sets of values fetched and stored by both  $x$  and  $y$  at the various steps:

$x = (f(1,2)) (g(1,2) \rightarrow 1) (h(0) \rightarrow 0) (f(1,2)) (g(1,2) \rightarrow 1) (h(0) \rightarrow 0) (f(1,2))$

$y = (f(1,2)) (g(1,2) \rightarrow 5) (h(0) \rightarrow 3) (f(5,2)) (g(5,2) \rightarrow 6) (h(3) \rightarrow 4) (f(6,2))$

fetch:

(6,3) (6,3) (0) (3,3) (3,3) (1) (0,3)

store:

$\emptyset$  (3) (1)  $\emptyset$  (0) (2)  $\emptyset$

It is also easily seen that  $x$  and  $y$  are similar, as are all of their prefixes of equal lengths.

### §3.6 The converse results.

We are now interested in finding out whether #5.8

has a converse, that is, how far we can go towards a characterization of proper renamings in terms of results computed. We shall find that such a characterization can be obtained for repetition-free strings, where  $x \in \hat{\Sigma}$  is repetition-free if  $x$  is a computation and  $\text{Char}(x,i) \neq \text{Char}(x,j)$  whenever they are both defined and  $i \neq j$ . The reader will note that this definition of repetition-freeness is equivalent to the definitions of [K&M] [Kel]. In other words, a repetition-free computation never computes the same values or the same test twice under an  $h$ -interpretation. Note also that  $x$  is repetition-free iff all  $y$  such that  $y \leq x$  are repetition-free, and that, by #5.8,  $x \leftrightarrow z$  for a repetition-free  $x$  implies that  $z$  is also repetition-free.

In the following results, an important role will be played by the fact that every repetition-free computation is liberal, where a computation  $x$  is said to be liberal if no two elements of  $\Omega^H(x)$  are equal.

For repetition-free computations it is possible to obtain a converse of #5.3. We shall use the following terminology: an operation  $a$  is lossless if  $R(a) \neq \emptyset$ ; is a simple test otherwise.

1. Proposition. If two repetition-free computations  $x, y$  are similar then there exists a bijection  $h^\Sigma: \omega \rightarrow \omega$  such that  $\text{Char}(x,i) = \text{Char}(y, h^\Sigma(i))$  for all  $i \in \omega$ .

Proof. Assume that  $\text{Char}(x,i) = f(t_1, \dots, t_n)^k$ .

Let  $x[i]$  be a lossless operation and let

$t = 'f^r(t_1, \dots, t_n)'$  be any value stored by  $x[i]$  under an  $h$ -interpretation. By the liberality of  $x$ , there is no other  $t$  in  $\Omega^H(x)$ . By the liberality of  $x$  and similarity of  $x, y$ ,  $\Omega^H(y)$  must also contain a single occurrence of  $t$ . Let this occurrence be stored by  $y[j]$ : then we define  $h^\Sigma(i) = j$ . The characteristic of  $y[j]$  must also be  $f(t_1, \dots, t_n)^s$  for some  $s$ , and one verifies that  $j$  is unique and independent of the choice of  $r$ . Assume now that  $k \neq s$ , and consider any  $h$ -interpretation  $H$  such that  $x$  is an  $H$ -computation. Since  $\Gamma_f(t_1, \dots, t_n) = k \neq s$  in  $H$ ,  $y$  is not an  $H$ -computation, contradicting the similarity of  $x, y$ . Thus  $k = s$  and  $\text{Char}(x, i) = \text{Char}(y, j)$ .

Let  $x[i]$  be a simple test. We show that there must exist a unique  $j$  such that  $\text{Char}(x, i) = \text{Char}(y, j)$ . Assume in fact that such  $j$  does not exist, and consider any  $h$ -interpretation  $H$  such  $x, y$  are both  $H$ -computations (such an interpretation must exist by the similarity of  $x, y$ ). Since by #1.1 the number of outcomes of  $f$  is greater than 1, we can define an  $h$ -interpretation  $H'$  that is as  $H$ , except for the fact that  $\Gamma_f(t_1, \dots, t_n) \neq k$ .  $H'$  can be chosen in such a way that  $y$  is still an  $H'$ -computation, but certainly  $x$  is not an  $H'$ -computation. This contradicts the similarity of  $x, y$ . By the repetition-freedom of  $y$  we know that  $j$  is unique.

We have then proved that for each  $i$  such that  $(\text{Char}(x, i))$  there exists a unique  $j$  such that  $\text{Char}(x, i) = \text{Char}(y, j)$ . The same argument holds in the direction

from  $y$  to  $x$ , and thus the desired bijection exists.

For repetition-free computations there exists a simple characterization of the concept of computing step by step the same values:

2. Proposition. If two repetition-free computations  $x, y$  are such that  $x \sim y$  then for all  $i \in \omega$   $\text{Char}(x, i) = \text{Char}(y, i)$ .

Proof. Assume that  $x, y$  are repetition-free and such that for all  $i \in \omega$ ,  ${}_i x \sim {}_i y$ . We show that for all  $i \in \omega$   $\text{Char}(x, i) = \text{Char}(y, i)$  by induction on  $i$ . If  $i = 0$   $\text{Char}(x, 0)$  and  $\text{Char}(y, 0)$  are both undefined. Let  ${}_{i+1} x \sim {}_{i+1} y$  but  $\text{Char}(x, i+1) \neq \text{Char}(y, i+1)$ . By induction hypothesis, for all  $j \leq i$ ,  $\text{Char}(x, j) = \text{Char}(y, j)$ . Also, by definition of repetition-freedom for all  $j, k \leq i+1$ ,  $j \neq k$  implies  $\text{Char}(x, j) \neq \text{Char}(x, k)$  and  $\text{Char}(y, j) \neq \text{Char}(y, k)$ . Therefore, the bijection  $h^\Sigma$  required by #1 does not exist for  ${}_{i+1} x$  and  ${}_{i+1} y$ , and  ${}_{i+1} x \not\sim {}_{i+1} y$ , a contradiction.

3. Lemma. Let  $x, y \in \hat{\Sigma}$ ,  $x$  be a renaming of  $y$ .  $x \leftrightarrow y$  iff for all  $i \in \omega$  there exists  $v_i$  such that  $x \xrightarrow{v_i} y'$  for some  $y'$  such that  ${}_i y = {}_i y'$ .

Proof. If  $x \xrightarrow{v} y$ , just take  $v_i = v$  for all  $i$ .

Conversely, assume that there exist the  $v_i$  defined above. We agree that if a segment is called  $M$  in  $x$ , then the corresponding segment or portion of segment in  $x' \leq x$

(if there exists one) is also called  $M$ . For all  $i$ , let  $R_i(x)$  be the subset of  $\text{Smap}(x)$  consisting of all the couples  $(M,m)$  such that for some  $j \leq i$ ,  $\bar{j} \in M$ , and let  $v_i'$  be the restriction of  $v_i$  to  $R_i(x)$  (note that  $v_i'$  is not necessarily a renaming function of  ${}_i x$  since it may be defined for some couples that are not in  $\text{Smap}({}_i x)$ ). For each  $i$ ,  $v_i'$  must be a restriction of  $v_{i+1}'$ , whenever the latter is defined (in fact,  ${}_i y'$  and  ${}_{i+1} y'$  are both prefixes of  $y$ , while no renaming is possible in open segments).

Define now a function  $v: \text{Smap}(x) \rightarrow \omega$  as follows: for each  $(M,m) \in \text{Smap}(x)$ , let  $v(M,m) = v_j'(M,m)$  where  $v_j'$  is any of the above defined restrictions such that  $(v_j'(M,m))'$ .

We claim that  $v$  is a renaming function. Assume in fact that it is not so. Then either  $v$  does not comply with #4.1.A) or  $v$  does not comply with #4.1.B). As concerns A), assume that for some open segment  $M$  of  $m$  in  $x$ ,  $v(M,m) \neq m$ .  $\bar{0} \in M$  and  $v_0(M,m) = v_0'(M,m) = v(M,m)$  by definition. Thus  $v_0(M,m) \neq m$ , a contradiction with the hypothesis that  $v_0$  is a renaming function. Assume instead that for two segments  $M, N$  such that  $M \cap N \neq \emptyset$  we have:  $v(M) = v(N)$ . Take  $\bar{1} \in M \cap N$ . Again, by definition of  $v_i', v'$  we have:  
 $v_i(M,m) = v_i'(M,m) = v(M,m)$  and:  $v_i(N,n) = v_i'(N,n) = v(N,n)$ .  
 Thus,  $v_i(M,m) = v_i(N,n)$ , a contradiction with the hypothesis that  $v_i$  is a renaming function.

As a consequence,  $v$  is a renaming function and clearly  $x \xrightarrow{v} y$ .

The reader should be aware of the fact that the following result holds for liberal strings, even though for

reasons of economy of exposition we only state it for repetition-free  $x, y$ .

4. Theorem. (Repetition-free computations that have the same characteristics are one a proper renaming of the other). Let  $x, y$  be repetition-free computations such that for all  $s \in \omega$ ,  $\text{Char}(x, s) = \text{Char}(y, s)$ . Then  $x \leftrightarrow y$ .

Proof. We show that if  $x \not\leftrightarrow y$  then there exists  $s$  such that  $\text{Char}(x, s) \neq \text{Char}(y, s)$ .

Assume first that  $y$  is not a renaming of  $x$ . Then either  $|x| \neq |y|$ , or there exists  $s$  such that  $F(x[s]) \neq F(y[s])$  or  $K(x[s]) \neq K(y[s])$ . In the first case, there exists  $s$  such that  $\text{Char}(x, s) \neq \text{Char}(y, s)$  but not  $\text{Char}(y, s) \neq \text{Char}(x, s)$  or vice-versa, as desired. In the second case,  $\text{Char}(x, s) \neq \text{Char}(y, s)$  by definition.

Assume instead that  $y$  is a renaming of  $x$ , but not a proper one. We prove that there exists  $s$  such that  $c_{s-1}^H x[D(x[s])] \neq c_{s-1}^H y[D(y[s])]$ . This implies  $\text{Char}(x, s) \neq \text{Char}(y, s)$  by definition.

By the contrapositive of #3, if  $x \not\leftrightarrow y$  then for some  $j$  there does not exist  $v_j$  and  $y'$  such that  $x \xrightarrow{v_j} y'$  and  $v_j y = v_j y'$ . Consider the smallest  $j$  such that  $v_j$  as above does not exist,  $v_{j-1}$  exists. Let  $z$  be such that  $x \leftrightarrow z$  and  $v_{j-1} z = v_{j-1} y$ . Since  $v_j z \neq v_j y$ ,  $z[j] \neq y[j]$ . Under the assumption that  $y$  is a renaming of  $x$ , either it is true that  $D(z[j]) \neq D(y[j])$ , or this is false, and it is

true that  $R(z[j]) \neq R(y[j])$ . These are our cases 1) and 2).

1) Assume that  $D(z[j]) \neq D(y[j])$ . By #5.5.B) we have  $c_{j-1}^H[z][D(z[j])] = c_{j-1}^H[x][D(x[j])]$ , i.e.  $z[j]$  and  $x[j]$  fetch the same ordered sets of values. Now,  $j-1^z = j-1^y$  implies  $c_{j-1}^H[z] = c_{j-1}^H[y]$ , i.e. after  $j-1^z$  and  $j-1^y$  the memory states are identical. Also,  $j-1^z$  and  $j-1^y$  are both repetition-free and liberal, thus any two values in memory are distinct. Hence,  $D(z[j]) \neq D(y[j])$  implies  $c_{j-1}^H[y][D(y[j])] \neq c_{j-1}^H[z][D(z[j])]$ , i.e. the ordered sets of values fetched by  $y$  and  $z$  at the  $j$ -th step are different. Thus  $j$  is the  $s$  that we wanted.

2) Assume instead that 1) is false, but  $R(z[j]) \neq R(y[j])$ . Then there must exist  $n$  such that all the following are true:

- i)  $n \in R(y[j])$
- ii) the corresponding element in  $R(z[j])$  is not  $n$ ;  
assume it is  $m$ .
- iii)  $\overline{j-1}, \overline{j} \in N$ , for some  $N \in \text{Seg}_n(z)$ .

In fact, i) and ii) must be trivially true. iii) must also be true, otherwise there would be  $y'$  such that  $x \leftrightarrow y'$  and  $j^y = j^{y'}$ : such a  $y'$  could be obtained from  $x$  by the same renaming function by which  $z$  is obtained from  $x$ , modified in such a way that  $R(y'[j]) = R(y[j])$ .

Then by #3.1 there exists  $r \geq j$  such that  $n \in D(z[r+1])$  and for no  $k$  such that  $r \geq k > j$ ,  $n \in R(z[k])$ . Since  $x$  is a computation, by #5.8  $z$  is a computation, and



by #5.5.B)  $c_r^H[x][D(x[r+1])]$  and  $c_r^H[z][D(z[r+1])]$  are both defined and are the same, i.e.  $x[r+1]$  and  $z[r+1]$  fetch the same ordered sets of values. If  $y[r+1]$  fetches a different ordered set of values, then  $r$  is the desired  $s$ . Thus we assume that  $y[r+1]$  fetches the same ordered sets of values as  $x[r+1]$  and  $z[r+1]$ . In particular, let  $z[r+1]$  fetch value  $t$  from  $n$ . Since by #3.1  $\bar{j}-1$ ,  $\bar{j}$  and  $\bar{r}$  all belong to the same segment of  $n$  in  $z$ ,  $t$  must have been stored in  $n$  by some  $z[h]$ , where  $h < j$ ; since  $j-1^z = j-1^y$ ,  $t$  must also have been stored in  $n$  by  $y[h]$ . However, since  $n \in R(y[j])$ , and  $y[j]$  stores the same values as  $x[j]$ , by the liberality of  $x$ ,  $t \notin c_j^y$ , i.e.  $t$  is not in memory after  $y[j]$ . Therefore, there must be  $d$  such that  $j < d \leq r$  and  $y[d]$  stores  $t$ . Now,  $z[d]$  cannot store  $t$ , since  $z$  is liberal by the liberality of  $x$  and #5.8. Therefore  $z[d]$  and  $y[d]$  store different sets of values and, as a consequence, also fetch different ordered sets of values. Since by #5.5.B)  $x[d]$  and  $z[d]$  fetch the same ordered sets of values,  $d$  is the desired  $s$ .

Note that if  $x$  and  $y$  are also lossless, then for some interpretation  $x$  actually stores in memory some value that is different from the value stored by  $y$  at the same step.

The desired converse of #5.8 follows then immediately:

5. Theorem. (If two repetition-free computations compute step by step the same values then they are one a proper renaming of the other). For repetition-free  $x$  and  $y$ ,  $x \sim y$  implies  $x \leftrightarrow y$ .

Proof. #2, #4.

Together with #5.8, the result above provides the following characterization: For repetition-free  $x$  and  $y$ ,  $x \leftrightarrow y$  iff  $x \sim y$ .

To see that this result does not hold in general, not even for liberal computations, consider the following  $x$  and  $y$ :

$$x = f(1)^0 f(0)^0 f(0)^0$$

$$y = f(1)^0 f(0)^0 f(1)^0$$

We have:  $x \sim y$  but  $x \not\leftrightarrow y$ .

## CHAPTER 4

### RENAMINGS IN SCHEMAS

#### Introduction

In the previous chapter we have established the fact that proper renamings (and only proper renamings) of terminator strings have certain desirable properties. In this chapter, we show that it is possible to extend the concept of proper renamings to schemas, and to derive corresponding properties for proper renamings of schemas.

In the first section, we introduce schemas as acceptors of terminator strings. In the second section we extend to schemas the concept of a segment, obtaining the concept of "area" of a variable in a schema. In the third section, we define proper renamings of schemas. This definition is analogous to the definition of proper renamings in terminator strings, and the same role that in that definition was played by the concept of segment of a variable in a computation, is played here by the concept of area of a variable in a schema. We then show that if two schemas are one a proper renaming of the other, the sets of strings accepted by them are the same up to proper renaming. The fourth section extends these concepts to sets of computations and shows that two schemas that are one a proper renaming of the other have the property of computing step by step the same results.

The second part of the chapter is dedicated to the proof of a partial converse of the latter result. In the fifth section we show that if two schemas whose state transition

diagram is a tree accept the same languages up to proper renaming, then they are one a proper renaming of the other. Finally, in section five, by imposing further restrictions on the sets of schemas in consideration, we obtain increasingly stronger converses of the results of section four. The strongest result proved is that there exists a class of schemas such that if two schemas in the class are one an improper renaming of the other then for some interpretation the two schemas actually store in memory some different values at some computation step. Thus, for schemas as for strings, the only generally acceptable procedure for renaming is the one that we have called "proper".

#### §4.1 Schemas.

In this work, schemas are defined as incompletely specified automata that accept terminator strings. This approach enables us to deal with them by using the familiar terminology of automata theory, and some automata-theoretical results. The occurrence of a terminator in a string causes either a state transition in the schema, or the rejection of the string, if the next state is not defined for that terminator. Each state in a schema is an accepting state.

Some readers may consider as more natural the following point of view, which clearly shows the applications of this theory to problems of systems design: in each state a number of operations (those operations for which the next state function is defined) are enabled for execution, and some

control device nondeterministically chooses which of these operations is executed next. The execution of an operation causes a state transition dependent upon the outcome of the operation. We shall occasionally refer to this point of view for explanatory remarks.

1. Definition. A parallel program schema over a function set  $F$ , or simply a schema is a triple  $S = (Q, q_0, \delta)$  where:

$Q$  is a nonempty, countable set of schema states;

$q_0 \in Q$  is the initial schema state;

$\delta$  is a partial function  $\delta: Q \times \Sigma \rightarrow Q$ , the schema state transition function;

For  $q \in Q$ , we write  $\phi(q) = \{a^k: (\delta(q, a^k))\}$ . We assume that the set  $\phi(q)$  is finite for all  $q$  (finitely branching property).

We extend  $\delta$  to a partial function  $\delta: Q \times \Sigma^* \rightarrow Q$  in the obvious way, i.e.

$$\delta(q, \lambda) = q.$$

$$\text{and, for } x \in \Sigma^*, \delta(q, xa^k) = \begin{cases} \delta(\delta(q, x), a^k) & \text{if this is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

For  $x \in \Sigma^*$ , let  $q_x = \delta(q_0, x)$ . We usually assume that  $S$  is connected, i.e. that for all  $q \in Q$ ,  $q = q_x$  for some  $x \in \Sigma^*$ .

The language  $L_S$  of a schema  $S$  is the set of all those finite terminator strings that are accepted by  $S$ , i.e.

$$L_S = \{x \in \Sigma^* : (q_x)\}.$$

We say that a schema is finite if  $Q$  is a finite set. Clearly, if  $S$  is a finite schema,  $L_S$  is a finite-state (or regular) language.

Schemas will be represented as shown in Fig. 1, according to a symbolism that is reminiscent of both automata theory and flow-charts, and should be self-explanatory (see §1.2).

The reader will note the main differences between this model and other models that have been used in related research. As opposed to Karp and Miller's model [K&M] and as in Keller [Kel] we do not consider queuing and we do not have the concepts of initiators and terminators. The latter limitation means that in our model all operations are assumed to have a null execution time, and that the execution of two operations cannot overlap in time. An earlier version of this work was written by using a model with initiators and terminators (but not queuing). The model was revised when it was found that the same results could be obtained, with much simpler proofs, in the present version.

Another major difference between our model and the models of [K&M] and [Kel], is the fact that here a number of axioms and conditions have been eliminated. Some of these axioms and conditions, however, will have to be introduced later in order to obtain certain results.

We are able to represent both tests (see for example the branches exiting states  $q_1$  and  $q_4$  in Fig. 1), and

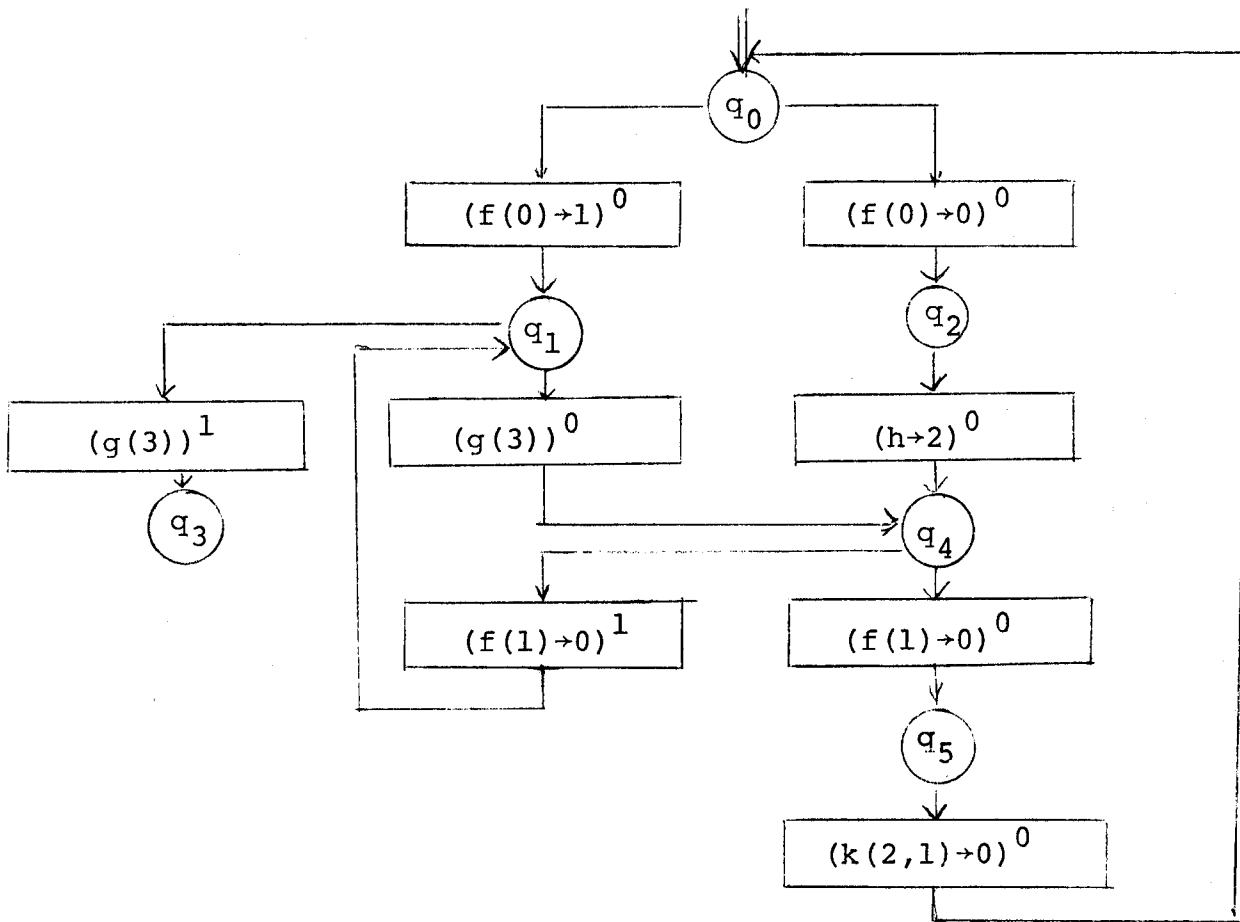


Fig. 1. Example of a schema.

situations as represented in the two branches leaving state  $q_0$ , where either one of the two branches can be taken whenever application of test  $f$  to the value of variable  $0$  gives outcome  $0$ . In a certain sense, we are here just one step away from a nondeterministic model as defined in automata theory. As concerns parallelism, the reader may consider that two operations  $a$  and  $b$  are allowed to be executed in parallel if for some  $x$ , both  $xa^k b^h$  and  $xb^h a^k$  are in  $L_S$ , i.e. if  $a$  and  $b$  can be executed in either order (see Fig. 1.4.C). This notion will be developed in Chapter 6.

We say that  $P \subseteq \hat{\Sigma}$  is prefix-closed if  $x \in P$  and  $y \leq x$  imply  $y \in P$ . A prefix-closed  $P$  is finitely-branching if for all  $x \in P$ , there exists a finite number of  $a^k \in \Sigma$  such that  $xa^k \in P$ . The following proposition characterizes those sets of strings that are languages for some schema.

2. Proposition.  $P = L_S$  for some schema  $S$  iff  $P$  is a non-empty, prefix-closed, finitely-branching subset of  $\hat{\Sigma}$ .

Proof. The direct direction immediately follows from the definitions. Conversely, we define a schema  $S$  such that  $P = L_S$  as follows:  $Q = \{ \langle x \rangle : x \in P \}$ ;  $q_0 = \langle \lambda \rangle$ ; for all  $\langle x \rangle \in Q$  and  $a^k \in \Sigma$ ,  $\delta(\langle x \rangle, a^k) = \langle xa^k \rangle$  if  $xa^k \in P$ ,  $\delta(\langle x \rangle, a^k)$  is undefined otherwise. Clearly,  $Q$  is nonempty since  $P$  is nonempty, and is countable since  $P$  is finitely-branching and thus countable.

For a language  $P$ , the schema  $S$  defined above is called the tree schema of  $P$ .



3. Definition. (Language equivalence) Let  $S = (Q, q_0, \delta)$  and  $S' = (Q', q'_0, \delta')$  be schemas. We say that  $S$  and  $S'$  are L-equivalent, written  $S \simeq S'$ , if  $L_S = L_{S'}$ . We say that  $q \in Q$ ,  $q' \in Q'$  are L-equivalent, written  $q \simeq q'$ , iff  $L_{S''} = L_{S'''}$ , where  $S'' = (Q, q, \delta)$ ,  $S''' = (Q', q', \delta')$ .

For any schema  $S$ , let  $L_S$  be the language of  $S$  and let  $T$  be the tree schema of  $L_S$ : clearly  $S \simeq T$ . We say that  $T$  is the tree schema of  $S$ . Tree schemas will have an important role in this research.

#### §4.2 Areas in schemas.

Areas of variables in schemas play the same role in the theory of renamings of schemas as segments of variables play in the theory of renamings of terminator strings. It is convenient to introduce areas by using the notion of route, that provides a link between the concept of segment and the concept of area.

1. Definition. Let  $S = (Q, q_0, \delta)$  be a schema. We say that  $\bar{R} \subseteq Q$  is a route of  $m \in \omega$  in  $S$  if there exists  $x \in L_S$  and  $M \in \text{Seg}_m(x)$  such that  $\bar{R} = \{q_{i,x} : \bar{i} \in M\}$ . In this case,  $M$  is said to be a segment of  $\bar{R}$ , and  $\bar{R}$  is the route of  $M$ .

$\text{Rut}_m(S)$  is the set of all routes of  $m$  in  $S$ .

We say that  $\bar{M}$  is an area of  $m$  in  $S$  if  $\bar{M}$  is a minimal subset of  $Q$  having the following properties:

A)  $\bar{M}$  contains a route of  $m$  in  $S$ .

B) for any route  $\bar{R}$  of  $m$  in  $S$ , if  $\bar{M} \cap \bar{R} \neq \emptyset$  then  $\bar{R} \subseteq \bar{M}$ .

$\text{Area}_m(S)$  is the set of all areas of  $m$  in  $S$ .

For each  $m$  and finite schema  $S$ ,  $\text{Area}_m(S)$  is clearly computable.  $\text{Area}_m(S) \neq \emptyset$  iff there exists  $q$  and  $a^k$  such that  $(\delta(q, a^k))$  and  $m \in D(a) \cup R(a)$ . We now give an algorithm for finding  $\text{Area}_m(S)$  for such  $m$  and a finite  $S$ .

2. Algorithm. for finding  $\text{Area}_m(S)$ .

- i) Take  $A_m^1$  to be the set of those singleton subsets  $\{q\}$  of  $Q$  such that  $m \in D(a^k)$  for some  $a^k$  such that  $(\delta(q, a^k))$ .
- ii) Given  $\{q\} \in A_m^1$ , let  $\bar{M}^q$  be the set of all those  $p$  such that  $\delta(p, x) = q$  for an  $x$  such that for all  $b^j \in x$ ,  $m \notin R(b^j)$ . Take  $A_m^2$  to be the set of all such  $\bar{M}^q$ .
- iii) Given  $A_m^2$ , take  $A_m^3 = A_m^2 \cup \{\{q\} : \delta(p, a^k) = q \text{ for some } a^k \text{ such that } m \in R(a^k) \text{ and some } p\}$ .
- iv) Given  $A_m^3$ , take  $q, p \in Q$  to belong to the same element of  $\text{Area}_m(S)$  iff there exist  $\bar{M}_1, \dots, \bar{M}_n \in A_m^3$  such that  $q \in \bar{M}_1$ ,  $p \in \bar{M}_n$  and  $\bar{M}_i \cap \bar{M}_{i+1} \neq \emptyset$  for all  $i \in \{1, \dots, n-1\}$ .

Since  $Q$  is a finite set, the procedure clearly terminates.

3. Example. We compute the areas in schema  $S$  presented in Fig 2. First of all, we know that  $\text{Area}_m(S) = \emptyset$  for  $m \notin \{0,1,2,3\}$ . Let us compute  $\text{Area}_0(S)$ . Step i) yields:  $A_0^1 = \{\{q_2\}\}$ . Step ii) yields:  $A_0^2 = \{\{q_1, q_2\}\}$ . Step iii) yields:  $A_0^3 = \{\{q_1, q_2\}, \{q_1\}, \{q_2\}, \{q_4\}\}$ . Finally by iv)  $\text{Area}_0(S) = \{\{q_1, q_2\}, \{q_4\}\}$ .

Similarly, we find:

$$\text{Area}_1(S) = \{\{q_2, q_3\}\}$$

$$\text{Area}_2(S) = \{\{q_0, q_1, q_2, q_3\}\}$$

$$\text{Area}_3(S) = \{\{q_0, q_1, q_2, q_3\}, \{q_4\}\}$$

In Fig. 2 we have drawn the areas of 0 in a way that we shall find useful to understand the following developments on renamings, i.e. by including in an area containing a state  $q$  the second half of those terminators that lead to  $q$  and the first half of those terminators that lead away from  $q$ . The correlation between the concept of segment and the concept of area is readily understood. Roughly speaking, states play for areas the same role that elements of the scope play for segments.

The concept of area map corresponds exactly to the concept of segment map: for a schema  $S$ ,  $\text{Amap}(S)$ , the area map of  $S$  is the set  $\{(\bar{M}, m) : \bar{M} \in \text{Area}_m(S)\}$ . The following proposition states the relation existing between segment maps and area maps. Note that an area or a route is open if it contains  $q_0$ , closed otherwise.

For a schema  $S$ ,  $x \in L_S$  and  $(M, m) \in \text{Smap}(x)$  we

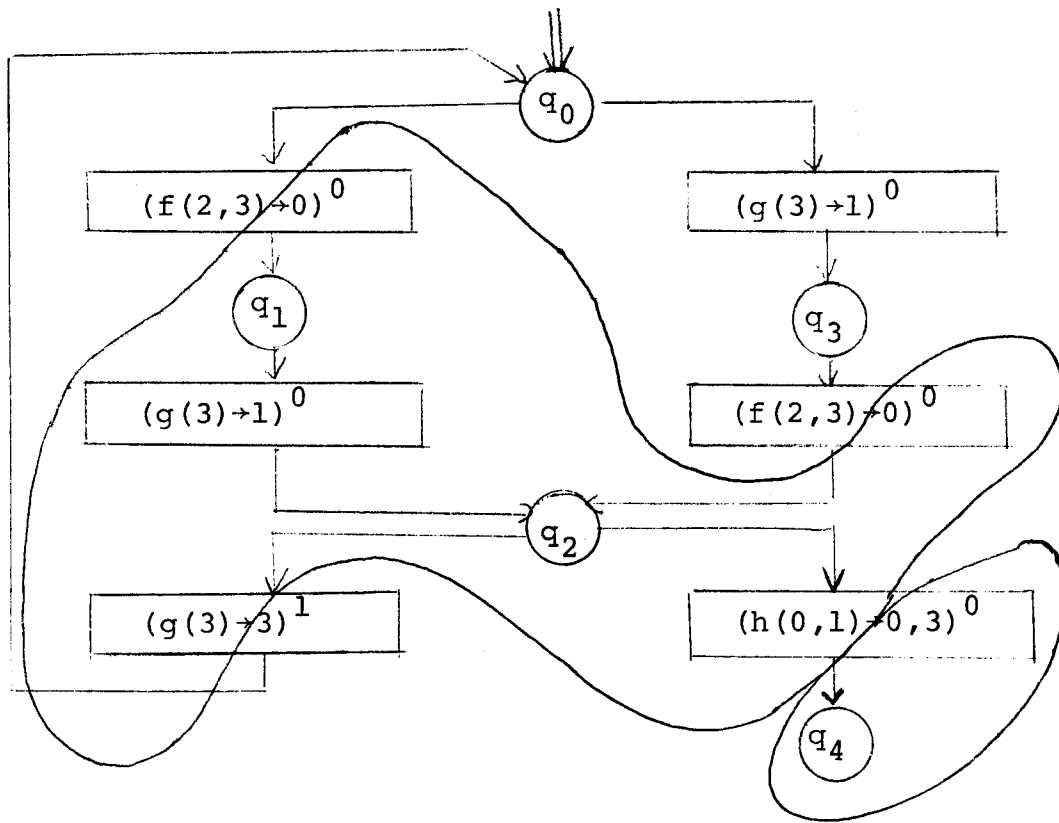


Fig. 2. Schema S and areas of 0 in S.

take  $\text{Ar}(M,m) = (\bar{M},m)$  iff  $\bar{M}$  is the area of  $m$  that contains the route of  $M$ . The following is then immediate:

4. Proposition. Let  $S$  be a schema,  $x \in L_S$ . Then for all  $(M,m), (N,n) \in \text{Smap}(x), (\bar{M},m), (\bar{N},n) \in \text{Amap}(S)$  the following hold:

- A) If  $M$  is an open segment of  $m$  in  $x$  and  $\text{Ar}(M,m) = (\bar{M},m)$ , then  $\bar{M}$  is an open area of  $m$  in  $S$ .
- B) If  $M \cap N \neq \emptyset$  for two segments  $M$  of  $m$ ,  $N$  of  $n$  in  $x$ , and  $\text{Ar}(M,m) = (\bar{M},m), \text{Ar}(N,n) = (\bar{N},n)$  then  $\bar{M} \cap \bar{N} \neq \emptyset$ .

As an example, consider the schema  $S$  of Fig. 2, and let  $x = (f(2,3) \rightarrow 0)^0 (g(3) \rightarrow 1)^0 (h(0,1) \rightarrow 0,3)^0$ : one verifies that  $\text{Ar}((\bar{1},\bar{2}),0) = (\{q_1, q_2\}, 0)$ .

#### §4.3 Renamings in schemas.

This section is devoted to the definition of proper renamings in schemas and to the proof of some of their basic properties. The most important of these is stated in #5: if one schema is a proper renaming of another, then there is a one-to-one correspondence between the languages of the two schemas such that corresponding strings are one a proper renaming of the other: in other words, the two languages are the same up to proper renamings. Using this fact, we derive for proper renamings of schemas several properties corresponding to

properties of proper renamings of terminator strings: most important, the facts that proper renamings of schemas are closed under composition, and have inverses.

The definition of proper renamings in schemas follows closely the model of the definition of proper renamings in terminator strings.

1. Definition. Let  $S$  be a schema. A renaming function of  $S$  is a function  $\bar{v}: \text{Amap}(S) \rightarrow \omega$  satisfying the following conditions: for all  $(\bar{M}, m), (\bar{N}, n) \in \text{Amap}(S)$

- A)  $\bar{v}(\bar{M}, m) = m$  for every open area  $\bar{M}$  of  $m$  in  $S$ .
- B)  $\bar{v}(\bar{M}, m) \neq \bar{v}(\bar{N}, n)$  whenever  $\bar{M} \cap \bar{N} \neq \emptyset$  and  $m \neq n$ .

Now, for each state  $q$  we rename the second half of those terminators that lead to  $q$  and the first half of those terminators that lead away from  $q$ .

2. Definition. (Renaming rule). Let  $S = (Q, q_0, \delta)$ ,  $S' = (Q', q'_0, \delta')$  be schemas. We say that  $S'$  is the proper renaming of  $S$  for a renaming function  $\bar{v}$  and we write  $S \xrightarrow{\bar{v}} S'$  if there exists a bijection  $h^Q: Q \rightarrow Q'$  and bijections  $h^q: \phi(q) \rightarrow \phi(h^Q(q))$  for all  $q \in Q$  such that the following hold for all  $q \in Q$  and  $a^k \in \phi(q)$ :

- A)  $h^Q(q_0) = q'_0$
- B)  $h^Q(\delta(q, a^k)) = \delta'(h^Q(q), h^q(a^k))$
- C)  $h^q(a^k)$  is the renaming of  $a^k$  that satisfies the following:

if  $R(a)[i] = m$ ,  $\delta(q, a^k) \in \bar{M}$  for some  $\bar{M} \in \text{Area}_m(S)$ ,  
 and  $\bar{v}(\bar{M}, m) = n$  then  $R(h^q(a^k))[i] = n$ ;  
 if  $D(a)[i] = m$  and  $q \in \bar{M}$  for some  $m \in \text{Area}_m(S)$   
 then  $D(h^q(a))[i] = \bar{v}(\bar{M}, m)$ .

Note that for each  $S$  and renaming function  $\bar{v}$  either there exists exactly one  $S'$  such that  $S \xrightarrow{\bar{v}} S'$ , or there does not exist any such  $S'$ . Fig. 3 shows an example where such an  $S'$  exists, while Fig. 4 shows an example where such an  $S'$  does not exist.

We now introduce a notation for an important relation between two sets of strings, a relation that holds iff the sets are the same up to proper renaming.

3. Definition. Let  $P, P' \subseteq \hat{\Sigma}$ . We write  $P \leftrightarrow P'$  iff there exists a bijection  $h^L: P \rightarrow P'$  such that:

for all  $x \in P$ ,  $h^L(x) \leftrightarrow x$ ;

$x, {}_i x \in P$  implies  $h^L({}_i x) = {}_i(h^L(x))$ .

From this point on, whenever we shall use the notation  $P \leftrightarrow P'$  defined above, we shall also use the notation  $h^L$  to denote the bijection existing between  $P$  and  $P'$ .

Thus we can state the fundamental property of proper renamings of schemas as follows:

4. Proposition. (Properly renamed schemas have the same languages up to proper renaming). If  $S \xrightarrow{\bar{v}} S'$  then  $L_S \leftrightarrow L_{S'}$ .

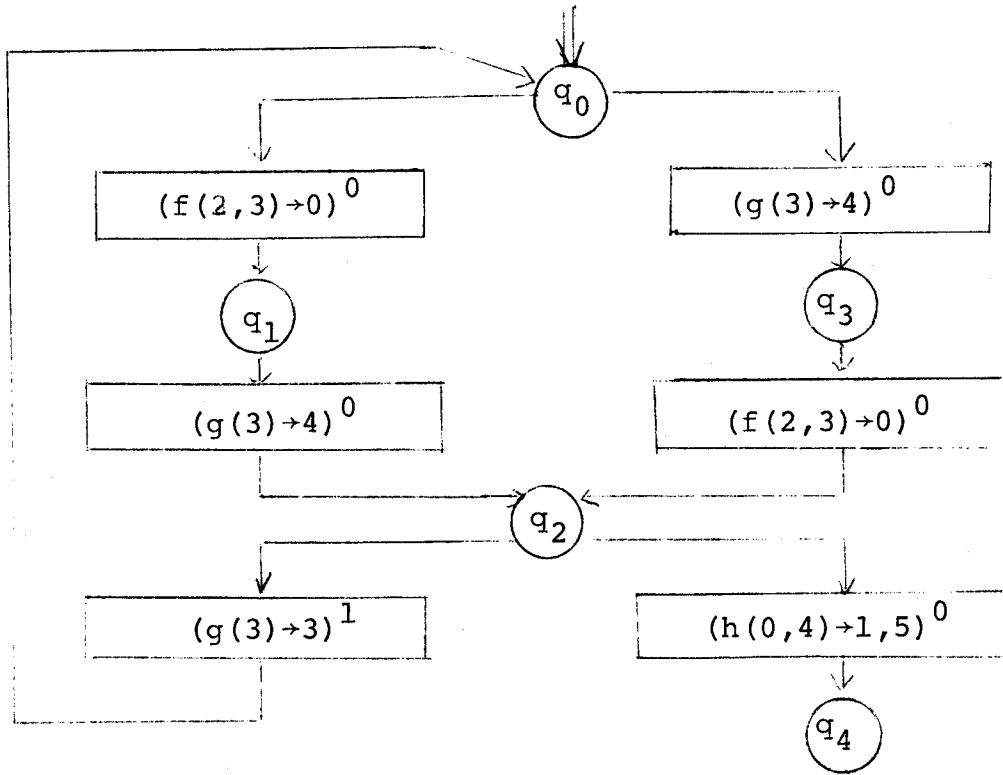


Fig. 3. Renaming of schema S in Fig. 2 according to the following renaming function:

$$\bar{v}(\{q_1, q_2\}, 0) = 0$$

$$\bar{v}(\{q_4\}, 0) = 1$$

$$\bar{v}(\{q_2, q_3\}, 1) = 4$$

$$\bar{v}(\{q_0, q_1, q_2, q_3\}, 2) = 2$$

$$\bar{v}(\{q_4\}, 3) = 5$$

$$\bar{v}(\{q_0, q_1, q_2, q_3\}, 3) = 3$$



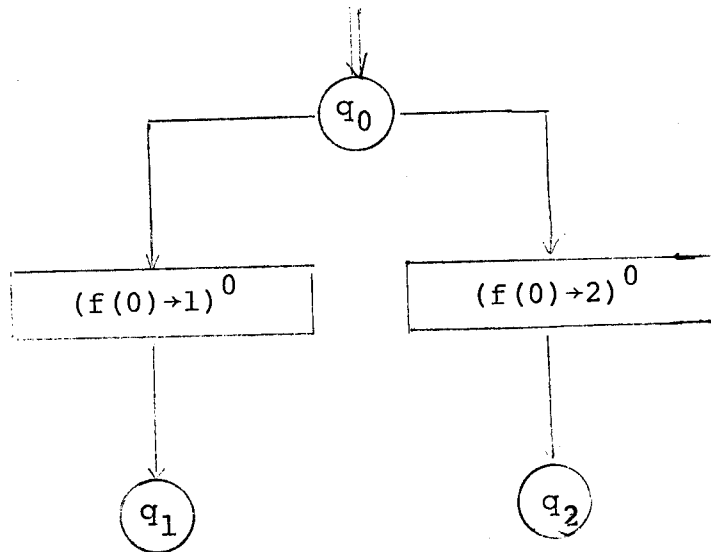


Fig. 4. Any renaming function  $\bar{v}$  such that  $\bar{v}(\{q_1\}, 1) = \bar{v}(\{q_2\}, 2)$  would identify the two terminators. Therefore, for no schema  $S'$ ,  $S \xrightarrow{\bar{v}} S'$ .

Proof. We define the bijection  $h^L$  required by #3 as follows:

$$h^L(\lambda) = \lambda$$

for  $xa^k \in L_S$ ,  $h^L(xa^k) = (h^L(x) \cdot h^q(a^k))$  where  $h^q$  is the bijection defined in #2 for  $q = q_x$ .

Clearly, if  $x \in L_S$  then  $h^L(x) \in L_S$ . Also,  $h^L$  is a bijection since  $h^q$  is a bijection.  $h^L$  satisfies the second condition of #3 by definition. We shall now show that it satisfies the first one.

For any  $x \in L_S$ , consider a renaming function  $v_x$  defined as follows: for all  $(M,m) \in \text{Smap}(x)$ ,  $v_x(M,m) = \bar{v}(\text{Ar}(M,m))$ .  $v_x$  is a function since both  $\text{Ar}$  and  $\bar{v}$  are functions. To see that  $v_x$  is a renaming function, assume  $v_x(M,m) \neq m$  for some open segment  $M$  of  $m$ . Then  $\bar{v}(\text{Ar}(M,m)) \neq m$ , a contradiction with the fact that by #2.4.A) if  $\text{Ar}(M,m) = (\bar{M},m)$  then  $\bar{M}$  is open, and with the fact that  $\bar{v}$  is a renaming function. Assume  $v_x(M,m) = v_x(N,n)$  where  $M \cap N \neq \emptyset$ ,  $m \neq n$ . Then  $\bar{v}(\text{Ar}(M,m)) = \bar{v}(\text{Ar}(N,n))$ , a contradiction with the fact that by #2.4.B) if  $\text{Ar}(M,m) = (\bar{M},m)$  and  $\text{Ar}(N,n) = (\bar{N},n)$  then  $\bar{M} \cap \bar{N} \neq \emptyset$ .

We now show that for all  $x \in L_S$ ,  $x \xrightarrow{v_x} h^L(x)$ . By definition of  $h^L$  above and by definition of  $h^q$  it is immediate that for all  $i \in \omega$ ,  $x[i]$  is a renaming of  $h^L(x)[i]$ . For any  $j \in \{1, \dots, |D(x[i])|\}$  assume  $D(x[i])[j] = m$ ,  $D(h^L(x)[i])[j] = n$ . By definition of  $h^q$  this is only possible if  $q_{i-1}^x \in \bar{M}$ , where  $\bar{v}(\bar{M},m) = n$ . Let  $\bar{i-1} \in M$ , where  $M \in \text{Seg}_m(x)$ .  $\text{Ar}(M,m) = (\bar{M},m)$  and  $v_x(M,m) = n$  as desired. A similar argument holds for any element in  $R(x[i])$ .

Thus  $x \xrightarrow{v} x, h^L(x)$  as desired.

The following is now obvious:

5. Corollary. Let  $S \xrightarrow{\bar{v}} S'$ ,  $x \in L_S$ ,  $h^L(x) = y$ : then  $h^Q(q_{i^x}) = q_{i^y}$  (the mappings  $h^L$ ,  $h^Q$  are here those defined for  $S, S'$  in #3, #2).

6. Example. Consider the two schemas of Fig. 2 and Fig. 3. We show the correspondence between some of the strings in the two languages:

$\lambda$	$\lambda$
$(f(2,3) \rightarrow 0)^0$	$(f(2,3) \rightarrow 0)^0$
$(g(3) \rightarrow 1)^0$	$(g(3) \rightarrow 4)^0$
$(f(2,3) \rightarrow 0)^0 (g(3) \rightarrow 1)^1$	$(f(2,3) \rightarrow 0)^0 (g(3) \rightarrow 4)^1$
$(g(3) \rightarrow 1)^0 (f(2,3) \rightarrow 0)^0$	$(g(3) \rightarrow 4)^0 (f(2,3) \rightarrow 0)^0$
$(f(2,3) \rightarrow 0)^0 (g(3) \rightarrow 1)^1 (g(3) \rightarrow 3)^1$	$(f(2,3) \rightarrow 0)^0 (g(3) \rightarrow 4)^1 (g(3) \rightarrow 3)^1$
etc.	

Also of interest is the property corresponding to #3.4.4 for renamings in schemas, that will enable us to easily obtain the correspondent of #3.4.5.

7. Proposition. (Proper renamings preserve area maps). Let  $S \xrightarrow{\bar{v}} S'$ . Then the relation  $h$  such that for all  $(\bar{M}, m) \in \text{Amap}(S)$ ,  $h(\bar{M}, m) = (\bar{M}', n)$  iff  $\bar{v}(\bar{M}, m) = n$  and  $\bar{M}' = \{q \in Q' : q = h^Q(p) \text{ for some } p \in \bar{M}\}$  is a bijection mapping  $\text{Amap}(S) \rightarrow \text{Amap}(S')$ .

Proof. We show that, for all  $p, q \in Q$ , we have:  $p, q \in \bar{R}$ , for some  $\bar{R} \in \text{Rut}_m(S)$  in  $\bar{M}$  iff  $h^Q(p), h^Q(q) \in \bar{P}$  for some  $\bar{P} \in \text{Rut}_n(S')$  (so that the one-to-one correspondence is in fact between routes). From this, the proposition follows without difficulty, by the definition of areas.

Consider any segment  $M$  of  $\bar{R}$  and assume that  $M \in \text{Seg}_m(x)$  for some  $x$ . Then  $p, q \in \bar{R}$  iff there exist  $\bar{i}, \bar{j}$  such that  $p = q_{\bar{i}x}$ ,  $q = q_{\bar{j}x}$ , and  $\bar{i}, \bar{j} \in M$ . Let  $h^L(x) = y$ . By #4 and #3.4.4 the above is true iff  $\bar{i}, \bar{j} \in N$ , for  $N \in \text{Seg}_n(y)$ . By #2.1 again this is true iff  $q_{\bar{i}y}$ ,  $q_{\bar{j}y} \in \bar{P}$  for some  $\bar{P} \in \text{Rut}_m(S')$ . By #5,  $q_{\bar{i}y} = h^Q(q_{\bar{i}x})$ ,  $q_{\bar{j}y} = h^Q(q_{\bar{j}x})$ .

By a reasoning similar to the one used to prove #3.4.5 it is now easy to see that proper renamings of schemas are closed under composition, and that every proper renaming has an inverse:

#### 8. Corollary.

- A) If there exist renaming functions  $\bar{v}, \bar{v}'$  such that  $S \xrightarrow{\bar{v}} S' \xrightarrow{\bar{v}'} S''$  then there exists a renaming function  $\bar{v}''$  such that  $S \xrightarrow{\bar{v}''} S''$ .
- B) For all schemas  $S, S'$ , there exists  $\bar{v}$  such that  $S \xrightarrow{\bar{v}} S'$  iff there exists  $\bar{v}^-$  such that  $S' \xrightarrow{\bar{v}^-} S$ .

In view of #8.B, we can extend to schemas the nota-

tion used for strings:  $S \leftrightarrow S'$  iff there exists a proper renaming function  $\bar{v}$  such that  $S \xrightarrow{\bar{v}} S'$ .  $\leftrightarrow$  is then an equivalence relation.

#### §4.4 Computations and renamings in schemas.

Up to this point, we have discussed schemas as automata, i.e. as recognizers of finite strings. This has allowed us to derive a number of results that will make it straightforward to deal with schemas as recognizers of computations.

We define a program as the parallel connection of two automata: a schema and an interpretation. The application-oriented reader may consider that the schema represents the control element of a computing system, while the interpretation represents the arithmetic unit and the memory.

For a schema  $S$ , we say that  $x \in \hat{\Sigma}$  is an S-string if for all  $y \leq x$ ,  $y \in L_S$ , and  $\phi(q_x) = \emptyset$  whenever  $x \in \Sigma^*$ .

1. Definition. Given a finite set of function symbols  $F$ , a schema  $S$  over  $F$ , and an interpretation  $I$  of  $F$ , a program is a couple  $(S, I)$ . For a program  $(S, I)$  we define:

$$\text{Comp}(S, I) = \{x \in \hat{\Sigma} : x \text{ is an } S\text{-string and an } I\text{-computation}\}$$

$$\text{Pref}(S, I) = \{x \in \Sigma^* : x \leq y \text{ for some } y \in \text{Comp}(S, I)\}$$

$$\text{Comp}(S) = \{x \in \hat{\Sigma} : x \in \text{Comp}(S, I) \text{ for some interpretation } I\}$$

$$\text{Pref}(S) = \{x \in \Sigma^* : x \in \text{Pref}(S, I) \text{ for some interpretation } I\}$$

Note that for any program  $(S, I)$ ,  $\text{Pref}(S, I) \subseteq L_S \cap L_I$

(equality holds for totally defined schemas, as will be shown in Chapter 6).

#3.4 can now be extended to sets of computations and prefixes.

2. Lemma. (Properly renamed schemas have properly renamed computations and prefixes). Let  $S \leftrightarrow S'$ : then  $\text{Comp}(S, I) \leftrightarrow \text{Comp}(S', I)$  and  $\text{Pref}(S, I) \leftrightarrow \text{Pref}(S', I)$ .

Proof. In this proof, we call  $h^L$  the bijection that in #3.4 we have shown to exist between  $L_S$  and  $L_{S'}$ ,  $h^C$  the bijection that we now want to prove to exist between  $\text{Comp}(S, I)$  and  $\text{Comp}(S', I)$ , and  $h^P$  the bijection that we want to prove to exist between  $\text{Pref}(S, I)$  and  $\text{Pref}(S', I)$ .

To see that  $h^C$  exists, for  $x \in \text{Comp}(S, I)$  define  $h^C(x)$  to be the  $x'$  such that for all  $i \in \omega$ ,  $h^L(i, x) = i, x'$ .  $h^C$  is a bijection as a consequence of the fact that  $h^L$  is a bijection. To see that  $x'$  is an  $S'$ -string, consider that, by definition of  $h^L$ , for all  $y \leq x'$ ,  $y \in L_{S'}$ . Furthermore, if  $x' \in \Sigma^*$  then  $x \in \Sigma^*$  and  $\phi(q_x) = \emptyset$ ; by #3.5 and #3.2  $\phi(q_{x'}) = \emptyset$ . Finally, for any  $x \in \hat{\Sigma}$ , by #3.6.3  $x \leftrightarrow x'$  and then by #3.5.8 and #3.2.2  $x'$  is an  $I$ -computation.

Next, for  $x \in \text{Pref}(S, I)$ , take  $h^P(x) = h^L(x)$ .

Since  $x \in \text{Pref}(S, I)$ , there exists  $y \in \text{Comp}(S, I)$  such that  $x \leq y$ .  $x \in L_S$ , and by definition of  $h^C$   $h^L(x) \leq h^C(y)$ , where  $h^C(y) \in \text{Comp}(S', I)$ . Thus  $h^L(x) \in \text{Pref}(S', I)$  as desired.

The same reasoning holds from  $S'$  to  $S$ , thus  $h^C$

and  $h^P$  are bijections.

We shall extend to schemas some concepts that were first introduced for strings.

3. Definition. Schemas  $S, S'$  are equivalent, written  $S \equiv S'$  (similar, written  $S \sim S'$ ) [compute step by step the same values, written  $S \approx S'$ ] if for all  $x \in \text{Comp}(S)$  there exists  $y \in \text{Comp}(S')$  such that  $x \equiv y$  ( $x \sim y$ ) [ $x \approx y$ ] and vice versa.

Karp and Miller [K&M] use a weaker definition of equivalence, where  $S$  and  $S'$  are said to be equivalent if for all interpretations  $I$  and  $x \in \text{Comp}(S, I)$  there exists  $y \in \text{Comp}(S', I)$  such that  $x \equiv_I y$ , and vice versa. Results #4 and #6.4 below are still true if such a definition, together with analogous definitions of similarity and computing step by step the same results, are used. However, in Chapter 6 we shall need the stronger definition presented above. Similar remarks hold for our definition of determinacy, to be introduced later in this section.

It is now not difficult to obtain the correspondent of #3.5.8 for schemas.

4. Theorem. (Properly renamed schemas compute step by step the same values). If  $S \leftrightarrow S'$  then  $S \approx S'$  (which implies  $S \sim S'$ ).

Proof. By #3 for each  $x \in \text{Comp}(S, I)$  there exists  $h^C(x) \in \text{Comp}(S', I)$  such that  $x \leftrightarrow h^C(x)$ . By #3.5.8  $x \sim h^C(x)$ . The same argument holds in the direction from  $S'$  to  $S$ .

We conclude this section by proving another property of proper renamings.

A schema will be said to be determinate if any two of its computations under any given interpretation are the same up to equivalence, quasi-determinate if any two such computations are the same up to similarity. Previous research [K&M] [Kel] has stressed the concept of determinacy, while in our work the concept of quasi-determinacy will be more important.

5. Definition. A schema is determinate [quasi-determinate] if for all interpretations  $I$ ,  $x, y \in \text{Comp}(S, I)$  implies  $x \equiv y$  [ $x \sim y$ ].

Clearly, determinacy implies quasi-determinacy.

We shall now show that proper renamings preserve quasi-determinacy, but not necessarily determinacy.

6. Proposition. Let  $S \leftrightarrow S'$ ; then  $S$  is quasi-determinate iff  $S'$  is quasi-determinate.

Proof. Consider  $x, y \in \text{Comp}(S', I)$  for some  $I$ . By the quasi-determinacy of  $S$ , for  $h^C(x), h^C(y) \in \text{Comp}(S, I)$  we have:



and  $h^P$  are bijections.

We shall extend to schemas some concepts that were first introduced for strings.

3. Definition. Schemas  $S, S'$  are equivalent, written  $S \equiv S'$  (similar, written  $S \sim S'$ ) [compute step by step the same values, written  $S \approx S'$ ] if for all  $x \in \text{Comp}(S)$  there exists  $y \in \text{Comp}(S')$  such that  $x \equiv y$  ( $x \sim y$ ) [ $x \approx y$ ] and vice versa.

Karp and Miller [K&M] use a weaker definition of equivalence, where  $S$  and  $S'$  are said to be equivalent if for all interpretations  $I$  and  $x \in \text{Comp}(S, I)$  there exists  $y \in \text{Comp}(S', I)$  such that  $x \equiv_I y$ , and vice versa. Results #4 and #6.4 below are still true if such a definition, together with analogous definitions of similarity and computing step by step the same results, are used. However, in Chapter 6 we shall need the stronger definition presented above. Similar remarks hold for our definition of determinacy, to be introduced later in this section.

It is now not difficult to obtain the correspondent of #3.5.8 for schemas.

4. Theorem. (Properly renamed schemas compute step by step the same values). If  $S \leftrightarrow S'$  then  $S \approx S'$  (which implies  $S \sim S'$ ).

Proof. By #3 for each  $x \in \text{Comp}(S, I)$  there exists  $h^C(x) \in \text{Comp}(S', I)$  such that  $x \leftrightarrow h^C(x)$ . By #3.5.8  $x \sim h^C(x)$ . The same argument holds in the direction from  $S'$  to  $S$ .

We conclude this section by proving another property of proper renamings.

A schema will be said to be determinate if any two of its computations under any given interpretation are the same up to equivalence, quasi-determinate if any two such computations are the same up to similarity. Previous research [K&M] [Kel] has stressed the concept of determinacy, while in our work the concept of quasi-determinacy will be more important.

5. Definition. A schema is determinate [quasi-determinate] if for all interpretations  $I$ ,  $x, y \in \text{Comp}(S, I)$  implies  $x \equiv y$  [ $x \sim y$ ].

Clearly, determinacy implies quasi-determinacy.

We shall now show that proper renamings preserve quasi-determinacy, but not necessarily determinacy.

6. Proposition. Let  $S \leftrightarrow S'$ ; then  $S$  is quasi-determinate iff  $S'$  is quasi-determinate.

Proof. Consider  $x, y \in \text{Comp}(S', I)$  for some  $I$ . By the quasi-determinacy of  $S$ , for  $h^C(x), h^C(y) \in \text{Comp}(S, I)$  we have:

$h^C(x) \sim h^C(y)$  (where  $h^C$  is as defined in #3). By the fact that  $x \leftrightarrow h^C(x)$  and  $y \leftrightarrow h^C(y)$  and by #3.5.8 we have:  $x \sim h^C(x)$  and  $y \sim h^C(y)$ . Thus  $x \sim y$ .

On the contrary, Fig. 5 presents two schemas  $S, S'$  such that  $S \leftrightarrow S'$  but such that  $S$  is determinate, while  $S'$  is not determinate (in fact,  $S'$  is quasi-determinate).

#### §4.5 A converse of #3.4. Tree schemas.

In order to obtain a converse of #4.4 we shall first derive the following partial converse of #3.4: if two tree schemas have the same language up to proper renaming then they are one a proper renaming of the other.

A full converse of #3.4 does not hold. Consider in fact the two schemas  $S$  and  $S'$  in Fig. 6. Clearly, there exists the desired one-to-one correspondence, but no renaming of  $S$  could possibly yield  $S'$ : in fact, two independent areas of 0,1 in  $S$  are merged in a single area of 0 in  $S'$ ,

The other purpose of this section is to introduce the concept of tree schema. A tree schema is a schema whose state transition diagram is a tree: we have already considered such schemas in §1. We shall have several occasions of using properties of tree schemas in this thesis, the most important of these properties being the fact that any schema has an L-equivalent tree schema.

The proof of #4, the converse of #3.4, requires a number of preliminary results and definitions. We start by

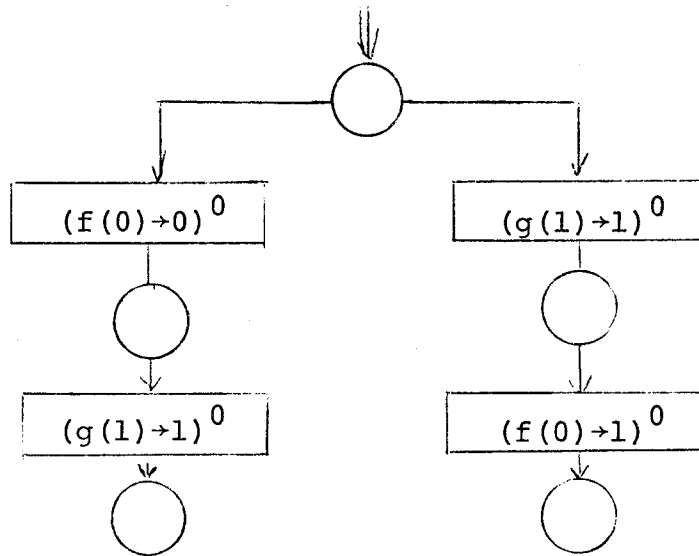
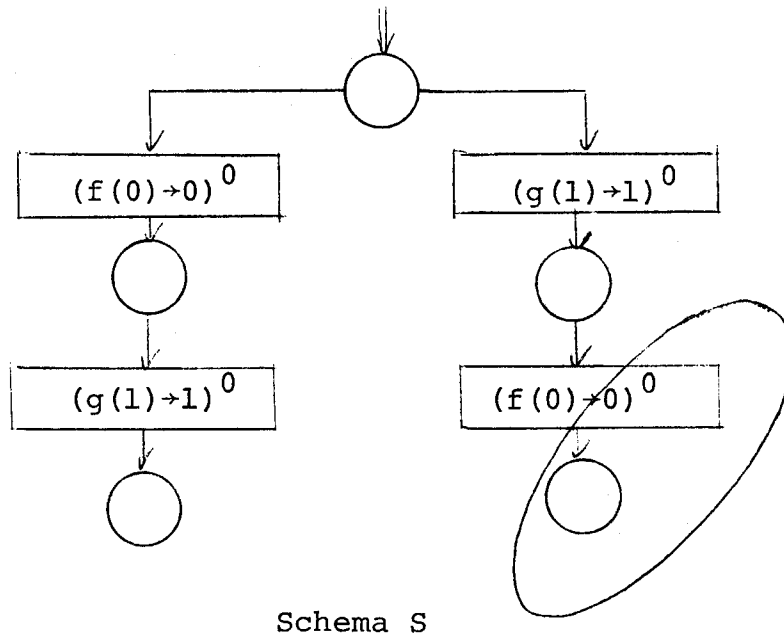
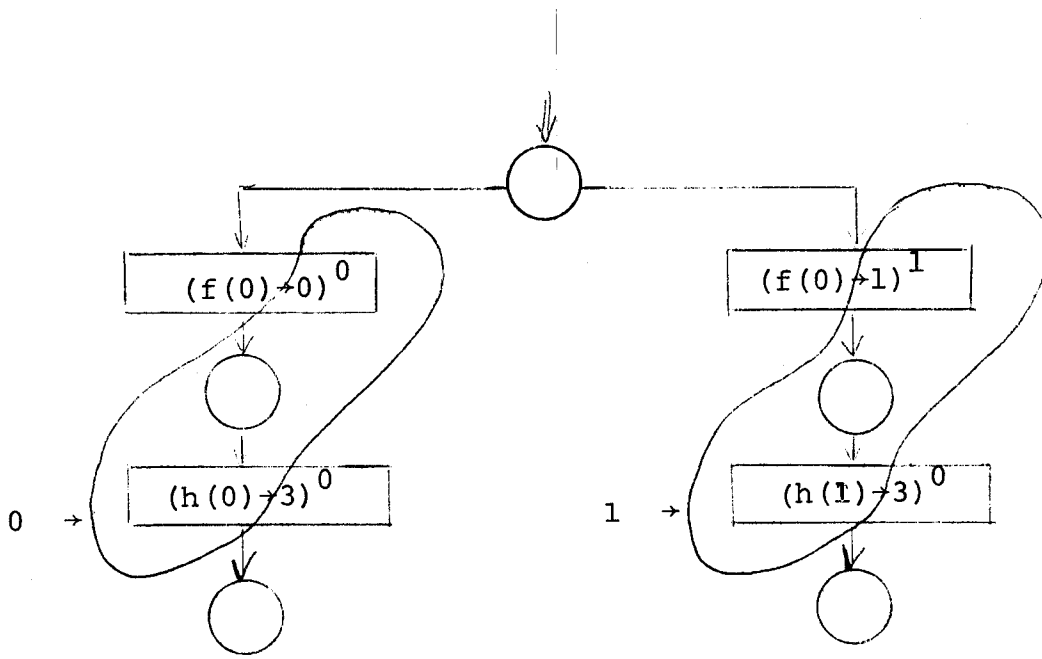
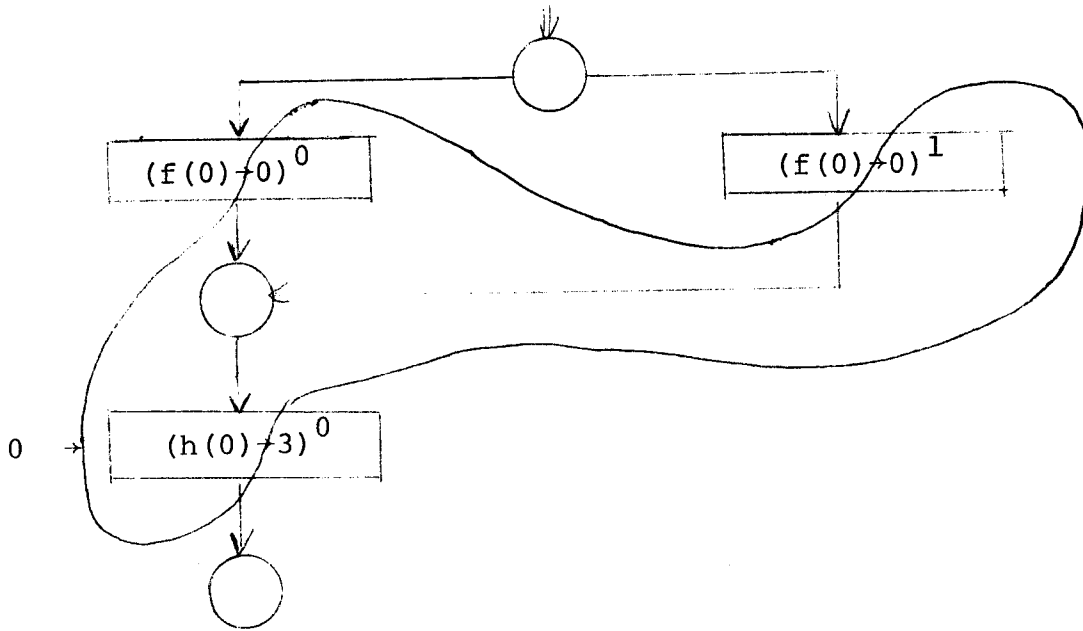


Fig. 5. Showing that proper renamings do not necessarily preserve determinacy.



Schema S



Schema S'

Fig. 6. A counterexample to a full converse of #3.4.

proving some properties of tree schemas.

1. Definition. We say that a schema  $T$  is a tree schema if for all  $q_x, q_y \in Q$ ,  $q_x = q_y$  implies  $x = y$ .

In a tree schema, let  $\bar{R}$  be a route of  $m$ :  
 $q_{i^x} \in \bar{R}$  is said to be the head of the route if for all  $q_{j^x} \in \bar{R}$  we have:  $j \geq i$  (in other words, the head is the first state of the route, and  $q_{j^x}$  is the head of the route of segment  $M$  of  $m$  in  $x$  iff  $\bar{j}$  is the head of  $M$ ).

2. Lemma. Let  $T$  be a tree schema. Then the following hold:

- A) Two routes  $\bar{R}, \bar{R}'$  in  $T$  are such that  $\bar{R} \cap \bar{R}' \neq \emptyset$  iff the head of  $\bar{R}$  is in  $\bar{R}'$  or vice versa.
- B) For two  $(\bar{M}, m), (\bar{N}, n) \in \text{Amap}(T)$  such that  $\bar{M}, \bar{N}$  are not both open and  $\bar{M} \cap \bar{N} \neq \emptyset$  there exists  $x \in L_T$  and  $(M, m), (N, n) \in \text{Smapp}(x)$  such that  $\text{Ar}(M, m) = (\bar{M}, m)$ ,  $\text{Ar}(N, n) = (\bar{N}, n)$  and  $M \cap N \neq \emptyset$ .
- C) Let  $\bar{R}, \bar{R}'$  be two routes of  $m$  in  $T$ ,  $q$  be the head of  $\bar{R}$ ,  $q'$  be the head of  $\bar{R}'$ ;  $\bar{R} \cap \bar{R}' \neq \emptyset$  iff  $q = q'$  (in other words, all routes in an area have a common head).
- D) Let  $x, y \in L_T$ ,  $M \in \text{Seg}_m(x)$ ,  $M' \in \text{Seg}_m(y)$ ,  $\text{Ar}(M, m) = \text{Ar}(M', m) = (\bar{M}, m)$ . Then there exists  $\bar{k} \in M \cap M'$  such that  $\bar{k}$  is the head of  $M$  and  $M'$  and  $k^x = k^y$ .

- E) Let  $x \in L_T$ ,  $M \in \text{Seg}_m(x)$ ,  $\text{Ar}(M, m) = (\bar{M}, m)$ : then  $M$  is open iff  $\bar{M}$  is.

Proof.

- A) Assume that  $q = q_{i^x}$  is the head of  $\bar{R}$ ,  $q' = q_{j^y}$  is the head of  $\bar{R}'$  and  $q \notin \bar{R}'$ ,  $q' \notin \bar{R}$ . Let  $p \in \bar{R} \cap \bar{R}'$ : then  $p = q_{k^x} = q_{h^y}$  where  $k > i$ ,  $h > j$ . By #1 this implies  $k^x = h^y$ , thus  $q_{j^y} = q_{j^x}$ . Now, either  $j \leq i$  or  $i < j$ : it is easily seen that in both cases  $q_{i^x}$ ,  $q_{j^x}$  and  $q_{k^x}$  all belong to the same route, opposite to the assumption.
- B) If  $\bar{M} \cap \bar{N} \neq \emptyset$  then there are routes  $\bar{R} \subseteq \bar{M}$ ,  $\bar{R}' \subseteq \bar{N}$  such that  $\bar{R} \cap \bar{R}' \neq \emptyset$ . By A) either  $\bar{R}$  contains the head of  $\bar{R}'$  or vice versa, and either only  $\bar{R}$  is open, or only  $\bar{R}'$  is, or neither is. This makes six cases, but they all reduce to the following: assume that  $\bar{R}$  contains the head of  $\bar{R}'$ ,  $\bar{R}$  is open or closed and  $\bar{R}'$  is closed. Assume that  $M$  is a segment of  $\bar{R}$ , where  $M \in \text{Seg}_m(x)$ . If  $q$  is the head of  $\bar{R}$  then  $q = q_{i^x}$  where  $i \in M$ . Since  $q \in \bar{R}'$ , also  $q = q_{j^y}$  and  $j \in N$  for some  $N \in \text{Seg}_n(y)$  and some  $y$ . However, by #1  $i^x = j^y$ , so  $i \in M \cap N$ , where  $M$  and  $N$  are segments of  $m$  and  $n$  in  $x$  (or, equivalently, in  $y$ ).
- C) Assume  $\bar{R} \cap \bar{R}' \neq \emptyset$ , but  $q \neq q'$ . By A) either  $\bar{R}$  contains  $q'$ , or  $\bar{R}$  contains  $q$ ; the two cases can obviously be treated in the same way, so we assume

that  $q' \in \bar{R}$ . If  $M \in \text{Seg}_m(x)$  is a segment of  $\bar{R}$ , then  $q' = q_{i,x}$ ,  $\bar{i} \in M$ ,  $q = q_{j,x}$ ,  $\bar{j} \in M$  and  $j < i$ .

Since  $q_{i,x}$  is the head of  $R'$  we must have

$m \in R(x[i])$ , a contradiction with the fact that  $\bar{i}, \bar{j} \in M$ .

- D) The routes of  $M$  and  $M'$  both belong to  $\bar{M}$  and thus by C) have a common head  $q_{k,x} = q_{k,y}$ . By #1,  $k^x = k^y$ .
- E) Assume that  $q_0 \in \bar{M}$ : by C) each route in  $\bar{M}$  contains  $q_0$ , therefore for each route, any segment of that route contains  $\bar{0}$ . The converse is #2.4.A).

The last preliminary step is the proof of a property of the bijection  $h^L$ .

3. Lemma. Let  $P, P'$  be two countable, prefix-closed subsets of  $\hat{\Sigma}$  such that  $P \leftrightarrow P'$ , and let  $x, y \in P$  be such that  $h^L(x) = x'$ ,  $h^L(y) = y'$  and  $k^x = k^y$ ; then  $k^{x'} = k^{y'}$ .

Proof. By #3.3  $k(h^L(y)) = h^L(k^y) = h^L(k^x) = k(h^L(x)) \leq h^L(x)$ .

The following is the desired partial converse of #3.4:

4. Proposition. (If two tree schemas have the same language up to proper renaming then they are one a proper renaming of



the other). If  $L_T \leftrightarrow L_{T'}$ , for tree schemas  $T$  and  $T'$  then  $T \leftrightarrow T'$ .

Proof. We claim that  $T \xrightarrow{\bar{v}} T'$  where  $\bar{v}$  is defined as follows:  
 $\bar{v}(\bar{M}, m) = n$  iff there exists  $x \in L_T$  and  $M \in \text{Seg}_m(x)$  such that  $\text{Ar}(M, m) = (\bar{M}, m)$  in  $T$ ,  $x \xrightarrow{v} h^L(x)$  and  $v(M, m) = n$ .  
 We shall now check that  $\bar{v}$  is well-defined as a function, that  $\bar{v}$  is a renaming function, and that in fact  $T \xrightarrow{\bar{v}} T'$ .

- 1) To verify that  $\bar{v}$  is well-defined as a function, we must verify that if there exist  $x, y \in L_T$  and  $M \in \text{Seg}_m(x)$ ,  $M' \in \text{Seg}_m(y)$  such that  $\text{Ar}(M, m) = \text{Ar}(M', m) = (\bar{M}, m)$ , and  $x \xrightarrow{v} h^L(x)$  for some  $v$  such that  $v(M, m) = n$ ,  $y \xrightarrow{v'} h^L(y)$  for some  $v'$  such that  $v'(M', m) = r$ , then  $n = r$ .  
 We have two cases:  $\bar{M}$  is an open area, or  $\bar{M}$  is a closed area. If  $\bar{M}$  is an open area then by #2.E)  $M$  and  $M'$  are both open and  $n = r = m$  by #3.4.1.A). If instead  $\bar{M}$  is closed then by #2.E)  $M$  and  $M'$  are also closed. By #2.D) there exists  $\bar{k} \in M \cap M'$  such that  $\bar{k}$  is the head of  $M$  and  $M'$  and  $k^x = k^y$ . Thus  $m = R(x[k])[j] = R(y[k])[j]$  for some  $j$ . By #3  $k(h^L(x)) = k(h^L(y))$ ; this implies  $R(h^L(x)[k])[j] = R(h^L(y)[k])[j]$  and  $v(M, m) = v'(M', m)$ .

- 2) We must now check that  $\bar{v}$  is a renaming function. The fact that #3.1.A) is true follows by the fact that, by #2.E), for any open area  $\bar{M}$  of  $m$  in  $T$  and for any  $(M, m)$  such that  $\text{Ar}(M, m) = (\bar{M}, m)$ ,  $M$  is an open segment. It remains to verify #3.1.B). Assume that  $\bar{M} \cap \bar{N} \neq \emptyset$  where  $\bar{M} \in \text{Area}_m(T)$ ,

$\bar{N} \in \text{Area}_n(T')$ . By #2.B) there exists  $x \in L_T$  and  $(M,m), (N,n) \in \text{Smap}(x)$  such that  $\text{Ar}(M,m) = (\bar{M},m)$ ,  $\text{Ar}(N,n) = (\bar{N},n)$  and  $M \cap N \neq \emptyset$ . Thus by #3.4.1.B)  $v(M,m) \neq v(N,n)$  for any renaming function  $v$  of  $x$ , and by 1) above  $\bar{v}(\bar{M},m) \neq \bar{v}(\bar{N},n)$ .

- 3) The last thing that we must check is that in fact  $T \xrightarrow{\bar{v}} T'$ , i.e. that the bijections  $h^Q$  and  $h^q$  defined in #3.2 exist. For each  $q_x \in Q$ , we define  $h^Q(q_x) = q_{x'}$  iff  $h^L(x) = x'$ . For each  $a^k \in \phi(q_x)$ , we define  $h^{q_x}(a^k) = b^k$  iff  $h^L(xa^k) = x'b^k$ . A) and B) in #3.2 follow immediately. To see C), assume  $R(a)[i] = m$ ,  $q_{xa^k} \in \bar{M}$  for some  $x$  and  $\bar{M} \in \text{Area}_m(T)$  such that  $\bar{v}(\bar{M},m) = n$ . Let  $\overline{|xa^k|} \in M$ , where  $M \in \text{Seg}_m(x)$ . Then  $\text{Ar}(M,m) = (\bar{M},m)$  and by definition of  $\bar{v}$ ,  $v(M,m) = n$ . Hence, if  $h^L(xa^k) = x'b^k$  then  $R(b^k)[i] = n$ . But  $b^k = h^{q_x}(a^k)$ . Thus  $R(h^{q_x}(a^k))[i] = n$  as desired. Similarly for  $D(a)$ .

The following restatement of #4 will be used in the next chapter:

5. Corollary. Let  $S, S'$  be schemas such that  $L_S \leftrightarrow L_{S'}$ , and let  $T$  be the tree schema of  $S$ ,  $T'$  be the tree schema of  $S'$ . Then  $T \leftrightarrow T'$ .

The reader should be aware of the crucial importance of the fact that the sets of strings in consideration are

prefix-closed. Proposition 4 would not hold if the bijection existed between non-prefix-closed sets.

For finite schemas, a related result will be presented in #5.3.2.

#### §4.6 A converse of #4.4.

The partial converse of #4.4 is the extension of #5.4 to sets of computations and prefixes. This converse only holds for tree, free schemas, where a schema  $S$  is said to be free if  $\text{Pref}(S) = L_S$ , i.e. every path in the schema corresponds to a prefix for some computation.

First, we prove some intermediate results.

1. Proposition. (If two free, tree schemas have the same set of prefixes up to proper renaming then they are one a proper renaming of the other). If  $\text{Pref}(T) \leftrightarrow \text{Pref}(T')$  for free, tree schemas  $T$  and  $T'$  then  $T \leftrightarrow T'$ .

Proof. Immediate by #5.4 and the definition of freedom.

Note that  $\text{Pref}(S) \leftrightarrow \text{Pref}(S')$  implies  $\text{Comp}(S) \leftrightarrow \text{Comp}(S')$  (but not vice versa).

We say that a schema  $S$  is repetition-free if for all  $x \in L_S$ ,  $x$  is repetition-free. Since any repetition-free string is a computation, a repetition-free schema is free.

2. Definition. We say that a schema  $S$  is consistent if the following holds: for all  $q \in Q$  and any two distinct

$a^k, b^k \in \phi(q)$  we have:  $/a^k/ \neq /b^k/$  (or: if  $xa^k, xb^k \in L_S$  and  $a^k \neq b^k$  then  $/a^k/ \neq /b^k/$ ). We say that  $P \subseteq \hat{\Sigma}$  is consistent if for all  $x, y \in P$ ,  $x \leftrightarrow y$  implies  $x = y$ .

For example, the schema in Fig. 1 is not consistent, since both  $(f(0) \rightarrow 1)^0$  and  $(f(0) \rightarrow 0)^0$  are in  $\phi(q_0)$ .

The following Lemma shows that consistent schemas have consistent languages and shows other related properties.

### 3. Lemma.

- A) A schema  $S$  is consistent iff  $L_S$  is consistent.
- B) If a schema  $S$  is consistent then  $\text{Comp}(S)$  is consistent.
- C) Let  $P, P' \subseteq \hat{\Sigma}$  be consistent, prefix-closed. If there exists a bijection  $h^L: P \rightarrow P'$  such that for all  $x \in P$ ,  $x \leftrightarrow h^L(x)$  then  $P \leftrightarrow P'$ .
- D) For consistent schemas  $S$  and  $S'$   $\text{Comp}(S) \leftrightarrow \text{Comp}(S')$  iff  $\text{Pref}(S) \leftrightarrow \text{Pref}(S')$ .

### Proof.

- A) Let  $x, y \in L_S$  be such that  $x \leftrightarrow y$ . Consider the smallest  $i$  such that  $_i x \neq _i y$ , and let  $_i x = x'a^k$ ,  $_i y = x'b^k$ . We have:  $x'a^k, x'b^k \in L_S$ ,  $a^k \neq b^k$ : however, since  $x'a^k \leftrightarrow x'b^k$ ,  $/a^k/ = /b^k/$ , contradicting the consistency of  $S$ .

Conversely, it is easily seen that if  $S$  is not consistent then  $L_S$  contains two distinct strings that are one a proper renaming of the other.

- B) Follows by the same reasoning of A).
- C)  $h^L$  satisfies by definition the first condition of #3.3. To see that for consistent, prefix-closed sets the second condition must also be satisfied, assume that  $h^L(x) = x'$  but for some  $i$ ,  $h^L(i x) = i z \neq i x'$ . We must have  $i z \leftrightarrow i x'$ , a contradiction with the consistency of  $P'$ .
- D) Assume that it is false that  $\text{Pref}(S) \leftrightarrow \text{Pref}(S')$ , and assume that for some  $x \in \text{Pref}(S)$  it is false that there exists exactly one  $y \in \text{Pref}(S')$  such that  $x \leftrightarrow y$ . If no  $y$  exists then for no computation  $z$  of  $S$  having  $x$  as a prefix there can be  $z' \in \text{Comp}(S')$  such that  $z \leftrightarrow z'$ . If more than one  $y$  exists then all these  $y$  must be one a renaming of the other, a contradiction with A). Hence for each  $x \in \text{Pref}(S)$  there exists  $y \in \text{Pref}(S')$  such that  $x \leftrightarrow y$  and vice versa, and a bijection  $h^L: \text{Pref}(S) \rightarrow \text{Pref}(S')$  as required by C) above exists. Therefore,  $\text{Pref}(S) \leftrightarrow \text{Pref}(S')$ .

The converse is true for any schema. For  $x \in \text{Comp}(S)$  define  $h^C(x)$  to be the  $y$  such that for all  $i x$ ,  $h^P(i x) = i y$  ( $h^C$  and  $h^P$  are as defined in #4.2.).

We are now ready to prove the partial converse of #4.4: if two repetition-free, consistent, tree schemas compute step by step the same values then they are one a proper renaming of the other.

4. Theorem. Let  $T$  and  $T'$  be two repetition-free, tree, consistent schemas. Then  $T \simeq T'$  implies  $T \leftrightarrow T'$ .

Proof. Assume that  $T \not\leftrightarrow T'$ . Then by the contrapositive of #1  $\text{Pref}(T) \not\leftrightarrow \text{Pref}(T')$ . By #3.D)  $\text{Comp}(T) \not\leftrightarrow \text{Comp}(T')$ . Since no computation of  $T$  ( $T'$ ) is a proper prefix of a computation of  $T$  ( $T'$ ), by #3.3 this is only possible if there does not exist a bijection  $h^C: \text{Comp}(T) \rightarrow \text{Comp}(T')$  such that for all  $x \in \text{Comp}(T)$ ,  $x \leftrightarrow h^C(x)$ . If  $T$  and  $T'$  are consistent then for a given  $x \in \text{Comp}(T)$  there could not be more than one  $y \in \text{Comp}(T')$  such that  $x \leftrightarrow y$ : in fact, if there were more than one, all such  $y$  would be one a proper renaming of the other, a contradiction with #3.B). Assume then that  $x$  is such that there is no  $y \in \text{Comp}(T')$  such that  $x \leftrightarrow y$ , but there is  $z \in \text{Comp}(T')$  such that  $x \simeq z$ . By #3.6.5 we have  $x \leftrightarrow z$ , a contradiction. The same argument holds from  $T'$  to  $T$ . Hence  $T \simeq T'$  is false.

By combining the above with #4.4 we obtain the following characterization result: for repetition-free, tree, consistent schemas  $T$  and  $T'$  we have:  $T \leftrightarrow T'$  iff  $T \simeq T'$ .

Finally, a stronger result can be obtained if one further condition is added. We say that a schema is decision-free if every function has just one outcome. If  $S$  is a decision-free schema then every  $S$ -string is an  $I$ -computation for any interpretation  $I$ .

5. Theorem. Let  $T$  and  $T'$  be repetition-free, tree, con-

sistent, decision-free schemas such that for all  $x \in \text{Comp}(T)$  and all interpretations  $I$  there exists  $y \in \text{Comp}(T')$  such that  $x \stackrel{I}{\sim} y$ . Then  $T \leftrightarrow T'$ .

Proof. Assume  $T \not\leftrightarrow T'$ . By #4  $T \sim T'$  is false. Let  $x \in \text{Comp}(T)$  be such that there exists no  $y \in \text{Comp}(T')$  such that  $x \stackrel{I}{\sim} y$ , and assume that for some h-interpretation  $H$  and  $z \in \text{Comp}(T')$ ,  $x \stackrel{H}{\sim} z$ . By #3.2.6 and the fact that  $x$  and  $z$  are both computations for all interpretations we have  $x \sim z$ , a contradiction.

Note also that by #3.1.1 a decision-free computation is lossless. Thus by what we have noted after the proof of #3.6.4 two schemas in this class that are not one a proper renaming of the other actually store in memory some different values at some step for some interpretation.

## CHAPTER 5

### MEMORY ECONOMY

#### Introduction

In this chapter, we shall consider a first application of renamings: memory economy.

In Section 1 we shall study the problem of renaming a schema in such a way that in the renamed schema the smallest possible number of variables is used. We shall find that this problem is equivalent to a well-known problem of graph theory. In Section 2 we shall show that further memory economy can be obtained by unwinding the loops of a schema. In the third section, we apply the main result of Section 2 to show that for a large class of schemas it is decidable whether two schemas in the class have the same language up to proper renaming. We conclude with some remarks on dynamic memory allocation (Section 4).

#### §5.1 Minimum memory requirements for schemas

One of the implications of the ideas developed in the previous chapter is that two distinct variables  $m, n$  in two areas  $M$  of  $m$  and  $N$  of  $n$  in a schema can be identified if and only if  $M$  and  $N$  do not intersect. We shall see in this section that this fact is relevant for the problem of finding the minimum memory requirements for schemas (or minimal renamings of schemas).



The results of this section have been known in the software literature for years, under the general heading of static storage allocation principles [Ye2] [Wil]. However, it is useful to briefly revisit them in the framework of our theory.

We shall show that the problem of finding the minimum memory requirements for a schema reduces to the following well-known problem of graph theory: given a graph without self-loops, color its vertices in such a way that:

- i) no two vertices connected by an edge are colored by the same color;
- ii) the total number of colors used is minimum .

For an account of work on the graph coloring problem, see [Rin].

First of all, we want to make our model somewhat more manageable, by limiting our consideration to a family of schemas which is closed under arbitrary proper renamings (we have pointed out in §4.3 that not every family of schemas has this property). The following result shows that such a family is the family of consistent schemas, introduced in #4.6.2.

1. Proposition. For any consistent schema  $S$  and for any renaming function  $\bar{v}$  of  $S$  there exists a schema  $S'$  such that  $S \xrightarrow{\bar{v}} S'$ .

Proof. We only need to verify that for any  $\bar{v}$  a bijection  $h^q$  as defined in #4.3.2 exists, because then it follows that  $S'$  is a schema. In fact, we show that for any two distinct  $a^k, b^j \in \phi(q)$ , if  $c^k$  and  $d^j$  are obtained by renaming

respectively  $a^k$  and  $b^j$  according to #4.3.2.C), then  $c^k \neq d^j$ . Since  $a^k \neq b^j$  then by the consistency of  $S$  either  $F(a) \neq F(b)$ , or  $k \neq j$ , or  $D(a) \neq D(b)$ . If  $F(a) \neq F(b)$  or  $k \neq j$  then by definition of renaming  $c^k \neq d^j$ . If  $D(a) \neq D(b)$  then  $D(a)[i] = m \neq D(b)[i] = n$  for some  $i, m, n$ . Thus  $q$  is in areas of both  $m$  and  $n$  and  $D(c)[i] \neq D(d)[i]$  by #4.3.1.B).

The reduction of the problem of finding the minimum memory requirements for a schema  $S$  to the problem of finding the minimum coloring of a graph can now be performed by constructing a graph, that we shall call "incompatibility graph" of  $S$ , having a vertex for each element of the area map of  $S$ , and where two distinct vertices are joined by an edge iff the corresponding areas intersect. We shall now show the details of the construction.

We first recall some graph-theoretical concepts. Let  $G = (V, E)$  be a graph, and  $C$  any set (called the set of colors). A mapping  $c: V \rightarrow C$  is called a coloring of  $G$  by set  $C$  if for any two vertices  $v, w$ ,  $c(v) = c(w)$  implies  $(v, w) \notin E$ . A coloring is a k-coloring if the number of colors used is  $k$ .  $G$  is said to be k-colorable if there exists a  $k$ -coloring of  $G$ . A coloring of  $G$  is minimal if it is a  $k$ -coloring and for no  $i < k$ , is  $G$   $i$ -colorable.

2. Definition. The incompatibility graph  $G_S$  of schema  $S$  is a graph  $(V, E)$  where:

$$V = \text{Amap}(S)$$

$$E = \{((\bar{M}, m), (\bar{N}, n)) : \bar{M} \cap \bar{N} \neq \emptyset \text{ and } m \neq n\}$$

Fig. 1 presents a schema and its incompatibility graph.

Next, we define the quantities that we want to minimize:

3. Definition. For a schema  $S$ ,  $\text{Var}(S)$ , the set of variables of  $S$  is defined as follows:

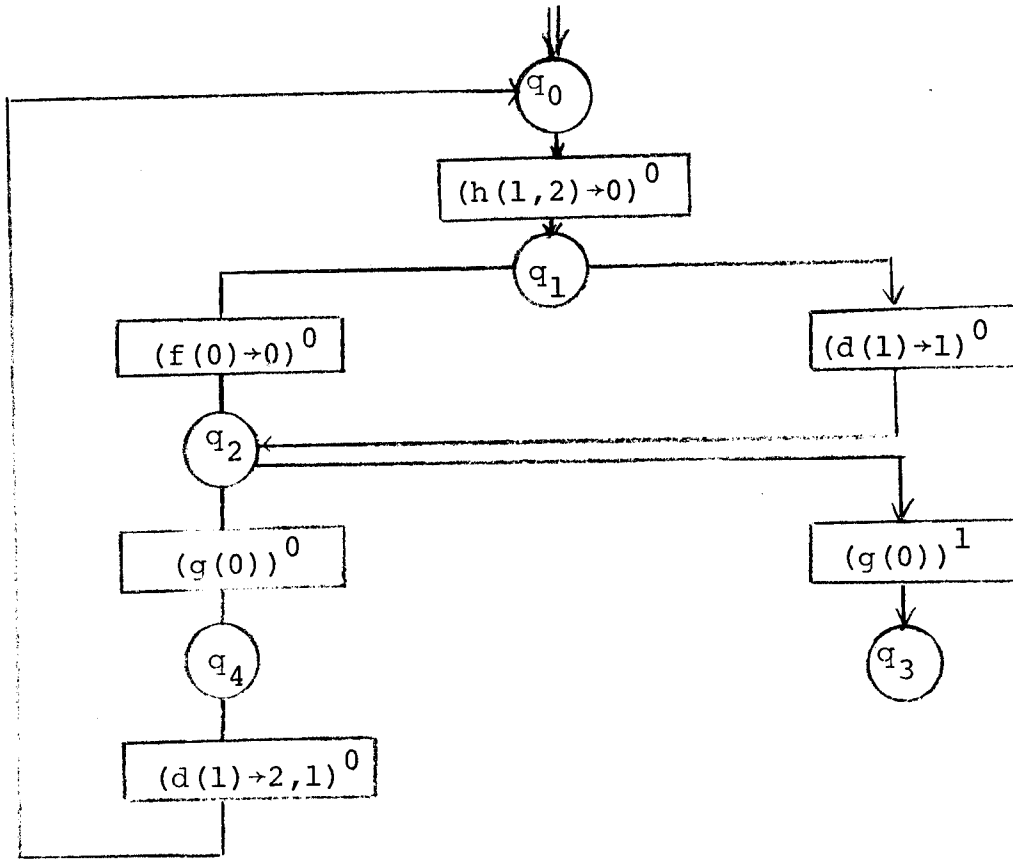
$$\text{Var}(S) = \{m \in \omega : \text{Area}_m(S) \neq \emptyset\}$$

$\text{Mem}(S)$ , the memory requirements of  $S$  is  $|\text{Var}(S)|$ .

$\text{Min}(S)$ , the minimum memory of  $S$  is the smallest  $k$  such that for some schema  $S'$ ,  $S \leftrightarrow S'$  and  $\text{Mem}(S') = k$ . A schema  $S$  is memory-reduced if  $\text{Mem}(S) = \text{Min}(S)$ . We say that  $S'$  is a minimal renaming of  $S$  if  $S \leftrightarrow S'$  and  $S'$  is memory-reduced.

It is then not difficult to see that the following result, first proved in [Lav] and [Yel], holds for our model. The proof involves simply the definition of a suitable correspondence between the set of colors and a subset of  $\omega$ .

4. Theorem. Let  $S$  be a consistent schema. Then for every  $k$ -coloring of  $G_S$  there exists a proper renaming  $S'$  of  $S$  having memory requirements  $k$  and vice versa.



Areas of 0:  $\{q_1\}, \{q_2\}$

Areas of 1:  $\{q_0, q_1, q_2, q_4\}$

Areas of 2:  $\{q_0\}$

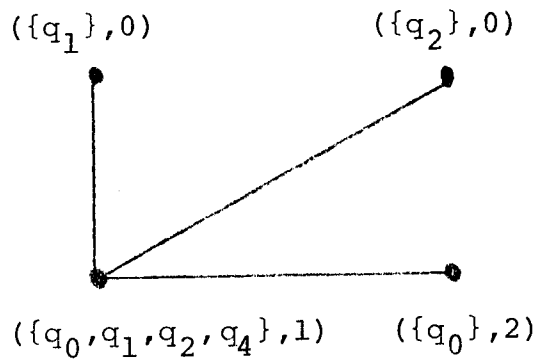


Fig. 1. A schema and its incompatibility graph.

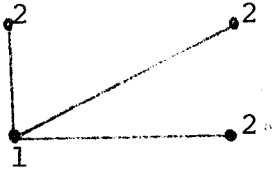
As a consequence, for a consistent schema  $S$  the problem of finding a minimal renaming of  $S$  is equivalent to the problem of finding a minimal coloring of  $G_S$ . If the incompatibility graph of  $S$  is finite and can be constructed, this problem can be solved with the algorithms for finding a minimal coloring of a finite graph. Fig. 2 shows an example.

It is interesting to note that in [Mar] it has been proved that for any connected, finite graph  $G$  without self-loops there exists a schema  $S$  such that  $G$  is the incompatibility graph of  $S$ . That proof can be adapted to our model. Therefore, the "minimal renaming problem" is of the same degree of computational complexity as the "minimal coloring problem".

#### §5.2 Schemas with minimum memory in a set of L-equivalent schemas.

The following questions present themselves: do any two L-equivalent schemas have the same minimum memory, and, if not, can we find, given a finite schema, an L-equivalent finite schema that has the smallest possible minimum memory? This section provides a negative answer to the first question, and a positive answer to the second one.

To see that two L-equivalent schemas do not need to have the same minimum memory, consider schemas  $S_1$  and  $S_2$  presented in Fig. 3. The reader will note that in no computation of  $S_1$  more than two variables are ever simultaneously occupied. Thus, the fact that  $S_1$  requires three variables



$$\bar{v}(\{q_1\}, 0) = 2$$

$$\bar{v}(\{q_2\}, 0) = 2$$

$$\bar{v}(\{q_0, q_1, q_2, q_4\}, 1) = 1$$

$$\bar{v}(\{q_0\}, 2) = 2$$

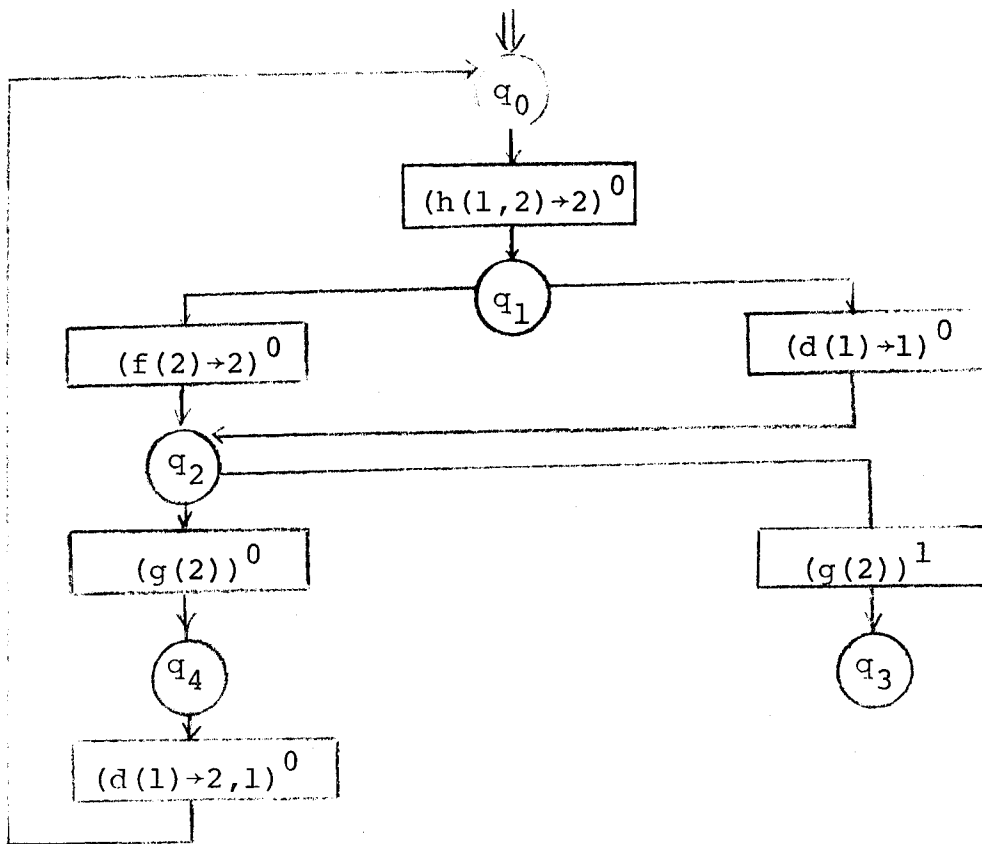


Fig. 2. A minimal coloring of the incompatibility graph of Fig. 1 by the set of colors  $\{1, 2\}$ , the related renaming function, and the renamed schema.

5.2

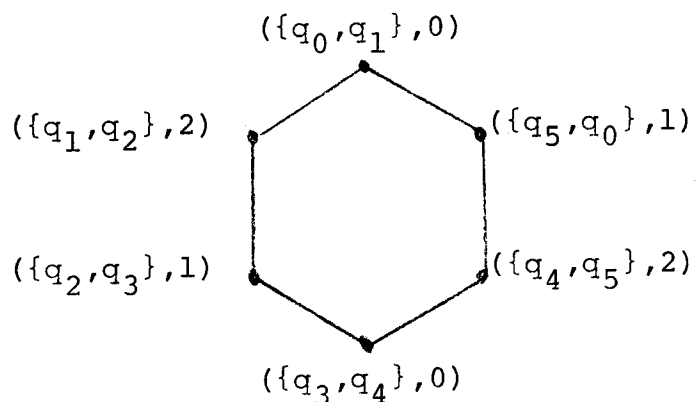
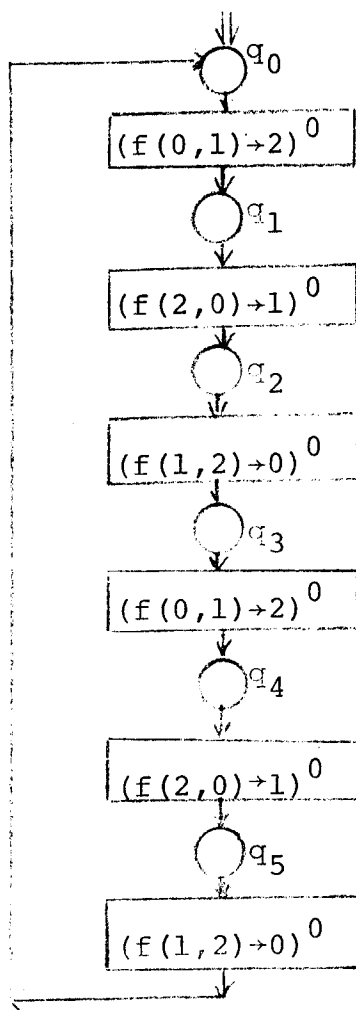
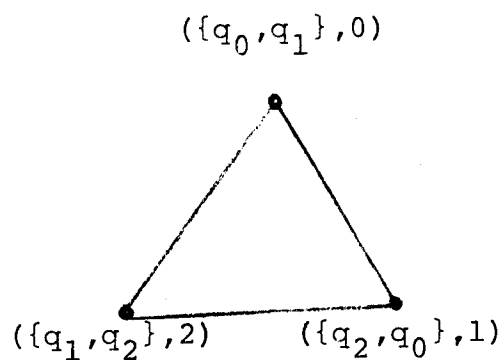
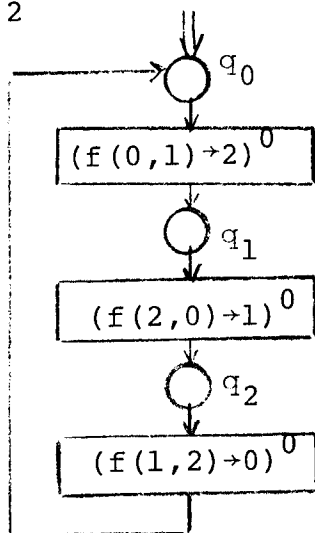


Fig. 3.A). Schema  $S_1$  (top) and schema  $S_2$  are L-equivalent, but the incompatibility graph of  $S_2$  can be colored with two colors, while the incompatibility graph of  $S_1$  can only be colored with three.

by the previous observation that  $\bar{v}_q$  is one-to-one for all  $q$ .

Continuing Example 3, we define the canonical renaming function of schema  $T$  as follows. Let  $(A_i, m)$  be the element of  $\text{Amap}(T)$  such that  $A_i$  is the area of  $m$  whose head is  $q_i$ .

$$\begin{aligned} \bar{v}(A_0, 1) &= 1; & \bar{v}(A_7, 1) &= 1. \\ \bar{v}(A_0, 2) &= 2; & \bar{v}(A_3, 2) &= 2; & \bar{v}(A_8, 2) &= 1; & \bar{v}(A_9, 2) &= 1. \\ \bar{v}(A_0, 3) &= 3; & \bar{v}(A_2, 3) &= 2; & \bar{v}(A_6, 3) &= 3. \\ \bar{v}(A_3, 4) &= 1; & \bar{v}(A_7, 4) &= 2. \end{aligned}$$

The result of the application of this renaming function to  $T$  is shown at the bottom of Fig. 4. For a tree schema  $T$ , we shall denote by  $T_c$  the schema obtained from  $T$  by canonical renaming. A schema  $T_c$  exists for any consistent  $T$ .

By the observation that in a canonical renaming the  $s$ -th variable is first used only when all previous  $s-1$  are occupied, we obtain the following:

6. Proposition. For any consistent, tree schema  $T$ ,  $T_c$  is memory-reduced.

Proof. We prove that  $\text{Mem}(T_c) = \text{Min}(T)$ . Take  $\underline{\omega}$  to be the set  $\omega$  ordered under the relation  $<$ . If  $\text{Mem}(T_c) = s$  then there must exist  $q$  in the set of states of  $T$  and  $m \in \text{Var}(T)$  such that  $\bar{v}_q(m) = n$ , where  $n = \underline{\omega}[s]$ . Consider a shortest  $x$  such that  $\bar{v}_{q_x}(m) = n$ . If  $x = \lambda$  then in  $T$



there must be  $s$  open areas. Otherwise  $x = ya^k$  for some  $ya^k$  and  $n = \text{Free}(q_{ya^k})[i]$  where  $i = |R(a)|$ . It is easily seen from #3.B) that this is only possible if  $q_{ya^k}$  is in the intersection of  $s$  distinct areas in  $T$ . In both cases there are  $s$  mutually intersecting areas in  $T$  and any proper renaming of  $T$  requires at least  $s$  variables.

For example, the canonical renaming of the schema in Fig. 4 requires 3 variables because state  $q_6$  is in the intersection of areas of 1,2,3.

Two technical Lemmas:

7. Lemma. Let  $p$  and  $q$  be states of  $T$  such that  $p \simeq q$  and  $\bar{v}_p = \bar{v}_q$ . Then  $h^Q(p) \simeq h^Q(q)$  in  $T_c$  (where  $h^Q$  is the bijection defined for  $T$  and  $T_c$  in #4.3.2).

Proof. The two subschemas of  $T$  having respectively  $p$  and  $q$  as initial states are identical. By the definition of canonical renaming function it is then clear that a canonical renaming function  $\bar{v}$  such that  $\bar{v}_p = \bar{v}_q$  renames both subschemas in the same way.

8. Lemma. Let  $T$  be the tree schema of a finite schema  $S$ . Then for any state  $q$  of  $T$

- A) it is decidable whether  $q$  belongs to an area of  $m$  in  $T$  for any  $m$ .
- B)  $\bar{v}_q$  is computable.

Proof.

- A) Clearly,  $q$  belongs to an area of  $m$  in  $T$  iff the following is true:  $\delta(p, a^h) = q$  for some  $p$  and  $a^h$  such that  $m \in R(a)$  or there exists  $xb^j$  such that  $(\delta(q, xb^j))^{-1}$ ,  $m \in D(b)$ , and for no  $c^k \in x$ ,  $m \in R(c)$ . The existence of such  $p$  and  $a^h$  is obviously decidable. Furthermore, by elementary automata theory, if  $x$  as above exists, there must exist one of length less than the number of states in the state set of  $S$ .
- B) Follows immediately.

We can finally conclude:

9. Theorem. Let  $S$  be a finite, consistent schema, and let  $T_c$  be the canonical renaming of the tree schema of  $S$ . Then there exists an effectively computable finite schema  $S_c$  such that  $T_c \simeq S_c$ .

Proof. Let  $T$  be the tree schema of  $S$ . For all  $q$  in the state set of  $T$ ,  $\bar{v}_q$  is a partial mapping from  $\text{Var}(S)$  into  $\text{Var}(T_c)$ . Since by #1 and #6  $\text{Var}(T_c) \leq \text{Var}(S)$  and  $\text{Var}(S)$  is a finite set, there is only a finite number, say  $n$ , of such mappings. By #7 any two states  $h^Q(p)$  and  $h^Q(q)$  of  $T_c$  such that  $p \simeq q$  and  $\bar{v}_p = \bar{v}_q$  are L-equivalent. Hence, the L-equivalence relation partitions the set of states of  $T_c$  in at most  $n|Q|$  sets of mutually L-equivalent states (where  $Q$  is the set of states of  $S$ ) and, by elementary finite automata theory,  $T_c$  is L-equivalent to a finite schema

$S_c$  with at most  $n|Q|$  states.

We shall now give an algorithm for computing  $S_c$ . Note that the algorithm uses the decidability and computability results of #8.

10. Algorithm for computing  $S_c$ . The algorithm considers three schemas: schema  $S$ , schema  $T$ , schema  $S_c$ . We assume for simplicity that  $S$  is reduced, so for each state  $q$  of  $T$  there exists exactly one state  $p$  of  $S$  such that  $p \simeq q$ : such a state is called "the state of  $S$  corresponding to  $q$ ". The states of  $S_c$  are of the form  $(p, \bar{v}_q)$ , where  $\bar{v}_q$  is one of the  $n$  mappings considered above, and  $p$  is the state of  $S$  corresponding to  $q$ . At the beginning,  $S_c$  is empty. We proceed enumerating the states of  $T$  in such a way that if  $|x| > |y|$ , then  $q_x$  is taken into consideration after  $q_y$ . For each state  $q_x$  of  $T$  that is enumerated we place a state in  $S_c$ , called the "state of  $S_c$  corresponding to  $q_x$ ".

We start taking into consideration state  $q_\lambda$  of  $T$ , and then we place in  $S_c$  a state  $(q_0, \bar{v}_{q_\lambda})$ . For a state  $q_{xa}^k$  of  $T$ , such that the corresponding state in  $S$  is  $p$ , consider  $(p, \bar{v}_{q_{xa}^k})$ , and assume that the state of  $S_c$  corresponding to  $q_x$  is  $(p', \bar{v}_{q_x})$ . Take  $b^k$  to be the renaming of  $a^k$  such that  $n = D(b)[i]$  ( $n = R(b)[i]$ ) for some  $i$  iff  $m = D(a)[i]$  ( $m = R(a)[i]$ ) and  $\bar{v}_{q_x}(m) = n$  ( $\bar{v}_{q_{xa}^k}(m) = n$ ). If there is no state  $(p, \bar{v}_{q_{xa}^k})$  in  $S_c$  then such a state is placed in  $S_c$ , and we define  $\delta((p', \bar{v}_{q_x}), b^k) = (p, \bar{v}_{q_{xa}^k})$ . Otherwise,  $(p, \bar{v}_{q_{xa}^k})$  is already

in  $S_c$  and we only need to define  $\delta((p', \bar{v}_{q_x}), b^k) = (p, \bar{v}_{q_{xa^k}})$ . In the latter case, we do not need to take into consideration any  $q_z$  such that  $z > xa^k$ . By the proof of #9, the algorithm will terminate before we take into consideration any state  $q_u$ , where  $|u| = n|Q|$ .

Note that  $S_c$  does not need to be automata-theoretically reduced.

The algorithm could be included in a compiler, providing a method for obtaining optimal memory assignments without running into the difficulty of the "register mismatch" noted at the beginning of the section.

11. Example. One verifies that schema  $S_3$  in Fig. 3 can be obtained from schema  $S_1$  by means of Algorithm 9. If we represent the mappings  $\bar{v}_q$  as in Example 3, the states of  $S_3$  are as follows:

$$\begin{array}{lll} q'_0 = (q_0, \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix}) & q'_1 = (q_1, \begin{pmatrix} 0 & 2 \\ 0 & 1 \end{pmatrix}) & q'_2 = (q_2, \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}) \\ q'_3 = (q_0, \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}) & q'_4 = (q_1, \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}) & q'_5 = (q_2, \begin{pmatrix} 1 & 2 \\ 1 & 0 \end{pmatrix}) \end{array}$$

§5.3 It is decidable whether two finite, consistent schemas have the same language up to renaming.

We shall now apply #9 to prove a decidability result.

In the following proof, we shall use the notation  $\bar{v}_q$  in a general sense, as follows: for any renaming function  $\bar{v}$  of a tree schema  $T$ , we write  $\bar{v}_q(m) = n$  iff  $q$  is in

an area  $\bar{M}$  of  $m$  such that  $\bar{v}(\bar{M}, m) = n$ .

1. Lemma. For consistent, tree schemas  $T$  and  $T'$ ,  $T \leftrightarrow T'$  implies  $T_c = T'_c$ .

Proof. Let  $\bar{v}$  be the renaming function such that  $T \xrightarrow{\bar{v}} T'$ ,  $\bar{v}$  the renaming function such that  $T \xrightarrow{\bar{v}} T_c$ ,  $\bar{v}'$  the renaming function such that  $T' \xrightarrow{\bar{v}'} T'_c$ . For any  $x \in L_T$  we write  $h^L(x)$  in  $L_{T'}$  as  $x'$ . Also, we take  $T = (Q, q_0, \delta)$ ,  $T' = (Q', q'_0, \delta')$ ,  $q_x = \delta(q_0, x)$ ,  $q'_x = \delta'(q'_0, x')$ . We show by induction on the length of  $x$  that

(\*) for all  $x \in L_T$  and  $m$ , if  $\bar{v}_{q_x}(m) = n$  then  $\bar{v}'_{q'_x}(m) = \bar{v}_{q'_x}(n)$ .

From (\*) it follows immediately that  $T_c = T'_c$ .

First of all, we note that there exists an open area of  $m$  in  $T$  iff such an open area exists in  $T'$  and vice versa. Thus the ordering  $\prec$  of  $\omega$  is the same for both  $T$  and  $T'$ .

If  $x = \lambda$ , for all  $m$ ,  $\bar{v}_{q_\lambda}(m) = m$  and  $\bar{v}'_{q'_\lambda}(m) = m = \bar{v}_{q'_\lambda}(m)$  as desired.

Assume that (\*) is proved for some  $x \in L_T$ . We prove it for  $xa^k \in L_T$ . Assume that both  $q_x$  and  $q_{xa^k}$  are in the same area of some variable  $m$ , and let  $\bar{v}_{q_x}(m) = n$ . By definition of renaming  $q'_x$  and  $q'_{xa^k}$  are in the same area of  $n$ . By induction hypothesis  $\bar{v}_{q_x}(m) = \bar{v}_{q'_x}(n) = s$  for some  $s$ , and thus by #2.2.A)  $\bar{v}_{q_{xa^k}}(m) = \bar{v}_{q'_{xa^k}}(n) = s$ . The reasoning is symmetrical, and thus  $\text{Free}(q_{xa^k}) = \text{Free}(q'_{xa^k})$ . Consider now any  $m$  such that  $m = R(a)[i]$  for some  $i$ . If  $\bar{v}_{q_{xa^k}}(m) = n$  then by

#2.2.B)  $\overline{v}_{q_{xa}k}(m) = \text{Free}(q_{xa}k)[i] = \text{Free}(q'_{xa}k)[i] = \overline{v}_{q'_{xa}k}(n)$   
 as desired.

For a schema  $S_c$ , we denote by  $S_{cr}$  the (automata-theoretically) reduced version of  $S_c$ .

2. Lemma. Let  $S$  and  $S'$  be finite, consistent schemas such that  $L_S \leftrightarrow L_{S'}$ . Then  $S_{cr} = S'_{cr}$ .

Proof. By #4.5.5,  $S \simeq T \leftrightarrow T' \simeq S'$  for the tree schemas  $T$  and  $T'$  of  $S$  and  $S'$ . By #1 the canonical renamings of  $T$  and  $T'$  are the same schema, that we shall call  $T_c$ . By #2.9 we have  $S_c \simeq T_c \simeq S'_c$ , and by elementary automata theory  $S_{cr} = S'_{cr}$ .

Our decidability result follows immediately:

3. Theorem. For finite, consistent schemas  $S$  and  $S'$  it is decidable whether  $L_S \leftrightarrow L_{S'}$ .

Proof. Clearly,  $S_{cr}$  and  $S'_{cr}$  are effectively computable from  $S$  and  $S'$ .

Note also that #2 holds if  $S$  and  $S'$  are finite, free, consistent schemas such that  $\text{Pref}(S) \leftrightarrow \text{Pref}(S')$ . For such schemas it is therefore decidable whether  $\text{Pref}(S) \leftrightarrow \text{Pref}(S')$ .

#### §5.4 Dynamic memory allocation.

We can consider what we have done up to this point as a model of mechanisms for allocating variable names in an optimal way at the time of the compilation of a program. At program run time, one of the following two memory allocation strategies can be chosen. In the method known as static memory allocation, one memory location is assigned to each variable in the program as soon as the program is loaded into memory. If such a method is chosen, the number of memory locations used by a program is equal to  $\text{Mem}(S)$ , where  $S$  is the schema of the program. In the method known as dynamic memory allocation, a memory location is assigned to a variable only when the variable is actually assigned a value during the computation. The location is taken from a pool of free memory locations, is kept occupied as long as the value is needed, and is returned to the pool after the value is referenced for the last time in the computation.

In this thesis, we shall not attempt a theory of dynamic memory allocation: however, we wish to consider briefly how such a theory could be developed. A program could be run under the control of a procedure that would both simulate the program's behaviour, and implement the dynamic memory allocation mechanism. The latter function would be performed by upkeeping the pool of free locations, and by looking ahead in the program to see whether the various values in memory are still needed. We have seen in #2.8 that this is decidable.

This process essentially corresponds to computing

the canonical renaming of the tree schema  $T$  of the schema  $S$  of the program. Therefore, at no point in the computation, a dynamic memory allocation mechanism will use more memory than  $\text{Min}(T)$ . However, dynamic memory allocation may well be more economical than that, since only variables referenced in computations that are computations for the given interpretation need to be allocated memory.

We can conclude that dynamic memory allocation for a program whose schema is  $S$  requires amounts of memory that vary with the interpretation, but never exceed the amount of memory needed for minimally renaming any schema  $L$ -equivalent to  $S$ .



CHAPTER 6  
PARALLELISM

Introduction.

In this chapter, we consider another major application of renamings, by showing that they can be used to make a schema more parallel. The theory developed here is a generalization of the one presented by Keller [Kell]. In that paper, a criterion for comparing the amount of parallelism present in equivalent schemas is given, and a schema  $S$  is said to be "closed" if there is no schema  $S'$  equivalent to  $S$ , which is more parallel than  $S$ . Keller then proves the following to be true for a large class of schemas:

- i) For finite schemas, the property of being closed is decidable.
- ii) If a schema  $S$  is not closed, then it is possible to transform  $S$  into a schema  $S'$  that is more parallel than  $S$ .
- iii) For every schema  $S$  there is an equivalent closed schema  $\tilde{S}$ . No procedure is known for obtaining  $\tilde{S}$  from  $S$  even in the case when  $\tilde{S}$  is finite. However,  $\tilde{S}$  can be "simulated" by a sort of "look-ahead" interpreter.
- iv) If  $S$  and  $S'$  are two equivalent schemas, then  $\tilde{S}$  and  $\tilde{S}'$  are L-equivalent.

We shall show that, by using the concept of renaming, it becomes possible to extend Keller's criterion and compare

the amount of parallelism present in two similar schemas. Accordingly, we define a schema to be "hyperclosed" if there is no schema  $S'$  similar to  $S$ , which is more parallel than  $S$ . It is interesting to note that all the facts established by Keller for his definition of parallelism based on equivalence have correspondents in our formulation.

In the first section of this chapter, we introduce the class of the "restricted" schemas, a class of schemas that are particularly well-behaved for parallelism. Whenever we shall use the word "schema" in the rest of this introduction, we shall mean "restricted schemas". We then introduce a criterion for comparing the amount of parallelism present in two similar schemas. Finally, we show that if in a schema  $S$  there is an operation that can be advanced one step ahead of some other operation, then there exists a schema  $S'$  where this operation is in fact enabled one step ahead, and such that  $S'$  is more parallel than  $S$ . In the second section, we show that this process can be extended to any finite number of steps. In section three we introduce two "natural" definitions of maximal parallelism in schemas, and we show their equivalence. We call "hyperclosed" a schema that is maximally parallel. We derive several properties of hyperclosed schemas, the most important of these being the fact that the hyperclosure of any finite schema can be approximated by a "look-ahead" mechanism.

#### §6.1 Enhancing parallelism in restricted schemas: the first step.

In this chapter we shall mainly be concerned with a

class of schemas that is considerably narrower than the class considered up to this point. We obtain it by combining some of the restrictions that we have already met with one more, that we shall now introduce.

A restriction that is quite natural (and that in fact is normally observed in programming systems) is to ask that whenever the next state is defined for some outcome of an operation, it is defined for all such outcomes. A schema is totally defined if for all  $x \in L_S$  and  $a^i, a^j \in \Sigma$ ,  $xa^i \in L_S$  implies  $xa^j \in L_S$ ; in other words, for all  $q \in Q$ ,  $(\delta(q, a^i))$  implies  $(\delta(q, a^j))$ .

We can now define the class of schemas that will be studied in this chapter: a schema is restricted if it is repetition-free, consistent, quasi-determinate and totally defined.

Next, we introduce a notion that will be adequate for a characterization of the idea of parallelism. Consider two similar schemas  $S$  and  $S'$ , such that for any computation  $x$  of  $S$  there exists a computation  $x'$  of  $S'$  such that  $x' \leftrightarrow x$ , but not vice versa. It is natural to think that  $S'$  has more freedom in computing than  $S$ , and that  $S'$  is more parallel than  $S$  (for a formal justification of a similar idea, see [Kel]). From this, we get the following definition.

For  $P \subseteq \hat{\Sigma}$ , let  $P^0$  be the set  $\{x: x \leftrightarrow y, \text{ for some } y \in P\}$ .

1. Definition. Let  $S$  and  $S'$  be consistent and similar schemas. We write  $S \geq S'$  if for all interpretations  $I$ ,  $\text{Comp}(S', I)^0 \subseteq \text{Comp}(S, I)^0$ . We write  $S > S'$  ( $S$  is more parallel than  $S'$ ) if  $S \geq S'$  and there exists  $I$  such that  $\text{Comp}(S', I)^0 \neq \text{Comp}(S, I)^0$ .

The definition above is equivalent to a definition in terms of prefixes. To see this, we need three preliminary Lemmas.

2. Lemma. Let  $P, L \subseteq \hat{\Sigma}$ ,  $x \in P^0 - L^0$ . Then there exists  $u$  such that  $u \leftrightarrow x$ ,  $u \in P$  and  $u \notin L^0$ .

Proof. If  $x \in P^0 - L^0$ , then  $x \in P^0$ ,  $x \notin L^0$ . If  $x \in P^0$ , there must exist  $u$  such that  $u \leftrightarrow x$ ,  $u \in P$ . Now,  $u \in L^0$  implies  $x \in L^0$ , a contradiction.

3. Lemma. Let  $P, L \subseteq \hat{\Sigma}$  be prefix-closed and consistent.

$P^0 \subseteq L^0$  iff there exists  $P' \subseteq L$  such that  $P \leftrightarrow P'$ .

Proof. Assume that  $P^0 \subseteq L^0$ , and consider any  $x \in P$ . Clearly, there must exist  $x' \in L$  such that  $x \leftrightarrow x'$ . Let  $P'$  be the set of all those  $x' \in L$  such that for some  $x \in P$ ,  $x \leftrightarrow x'$ : we show that  $P'$  is prefix-closed. For  $x' \in P'$  as above, consider  ${}_i x'$  for any  $i$ . Since  $P$  and  $L$  are prefix-closed,  ${}_i x \in P$  and  ${}_i x' \in L$ . Since  ${}_i x \leftrightarrow {}_i x'$ ,  ${}_i x' \in P'$ . Next, we see that  $P'$  is consistent: in fact, since  $P' \subseteq L$ , the inconsistency of  $P'$  implies the inconsistency of  $L$ . By the above and #4.6.2 there exists a bijection  $h^L$  between  $P$  and

$P'$  as required by #4.6.3.C) and thus  $P \leftrightarrow P'$ .

Conversely, assume that there exists  $P' \subseteq L$  such that  $P \leftrightarrow P'$ , and let  $x \in P^\circ$ . Some  $y$  such that  $y \leftrightarrow x$  is in  $P$ , and some  $z$  such that  $z \leftrightarrow y$  is in  $P'$ , i.e. in  $L$ . But  $z \leftrightarrow x$ , thus  $x \in L^\circ$ .

4. Lemma. Let  $S$  and  $S'$  be consistent schemas. Then  $\text{Comp}(S, I)^\circ \subseteq \text{Comp}(S', I)^\circ$  iff  $\text{Pref}(S, I)^\circ \subseteq \text{Pref}(S', I)^\circ$ .

Proof. We prove that if there exists  $x \in \text{Pref}(S, I)^\circ$  which is not in  $\text{Pref}(S', I)^\circ$  then there exists  $y$  that is in  $\text{Comp}(S, I)^\circ$  but not in  $\text{Comp}(S', I)^\circ$ . By #2,  $x \in \text{Pref}(S, I)^\circ - \text{Pref}(S', I)^\circ$  implies that there exists  $u$  such that  $u \leftrightarrow x$ ,  $u \in \text{Pref}(S, I)$ , and  $u \notin \text{Pref}(S', I)$ . Consider any  $y \in \text{Comp}(S, I)$  such that  ${}_i y = u$  for some  $i$ . Assume that  $y \in \text{Comp}(S', I)^\circ$ , i.e. that there exists  $y' \in \text{Comp}(S', I)$  such that  $y \leftrightarrow y'$ . Clearly  ${}_i y' \in \text{Pref}(S', I)$ . Since  ${}_i y' \leftrightarrow {}_i y = u$ ,  $u \in \text{Pref}(S', I)^\circ$ , a contradiction.

Conversely, assume that  $\text{Pref}(S, I)^\circ \subseteq \text{Pref}(S', I)^\circ$ , i.e. by #3 assume that there exists  $P \subseteq \text{Pref}(S', I)$  such that  $P \leftrightarrow \text{Pref}(S, I)$ , and let  $h^P$  be the correspondence existing between  $P$  and  $\text{Pref}(S, I)$ . Then for any  $x \in \text{Comp}(S, I)$  there exists  $x' \in \text{Comp}(S', I)$  such that  $x \leftrightarrow x'$ : just define  $x'$  to be such that for all  $i$ ,  $h^P({}_i x) = {}_i x'$  (where  $h^P$  is as defined in #4.4.2). Hence there exists a correspondence between  $\text{Comp}(S, I)$  and a subset of  $\text{Comp}(S', I)$  such that corresponding strings are one a proper renaming of the other, and  $\text{Comp}(S, I)^\circ \subseteq \text{Comp}(S', I)^\circ$ .

We then immediately have:

5. Proposition. For consistent schemas  $S$  and  $S'$ ,  $S \geq S'$  iff for all interpretations  $I$ ,  $\text{Pref}(S', I)^{\circ} \subseteq \text{Pref}(S, I)^{\circ}$ ;  $S > S'$  iff  $S \geq S'$  and there exists  $I$  such that  $\text{Pref}(S', I)^{\circ} \neq \text{Pref}(S, I)^{\circ}$ .

We shall now see how it is possible to enhance the parallelism of a schema that is not already maximally parallel.

We introduce the following notation: for a state  $q$ ,

$/\phi(q)/ = \{ /a/ : a^k \in \phi(q) \text{ for some } k \}$ .

For some  $u \in \Sigma^*$  and  $a \in A$ , we say that  $u$  affects  $D(a)$  iff for some  $b^j \in u$ ,  $R(b) \cap D(a) \neq \emptyset$ .

We say that  $\text{Ult}^1(/a/, x)$  is true in schema  $S$  if  $x \in \text{Pref}(S)$  and for all  $y \in \text{Comp}(S)$  such that  $y \geq x$  there exists  $c^j$  such that  $y > xc^j$ ,  $/a/ \in / \phi(q_{xc^j}) /$  and  $c^j$  does not affect  $D(a)$ . We say that  $\text{Adv}^1(/a/, x)$  is true in  $S$  if  $\text{Ult}^1(/a/, x)$  is true in  $S$  and  $/a/ \notin / \phi(q_x) /$ .

If  $\text{Adv}^1(/a/, x)$  is true in a restricted  $S$  then  $/a/$  can be advanced by one step ahead of some other operations. In fact, the following construction shows how this can be done for the tree schema  $T$  of  $S$  (for which  $\text{Adv}^1(/a/, x)$  is also true). In the resulting schema  $S'$ ,  $/a/$  is enabled immediately after  $x$  (the word "enable" will be used in the sense specified at the beginning of §4.1:  $/b/$ , where  $/b/ = /a/$ , or  $a$ , or  $a^k$  are "enabled" immediately after  $x$ , or at  $q_x$ , iff  $(\delta(q_x, a^k))$ ).

6. Construction. (Desequencing). Consider a restricted tree schema  $T$  where  $\text{Adv}^1(/a/,x)$  is true. If  $|R(a)| = n$  then take  $n$  arbitrary elements  $r_1, \dots, r_n$  from the set  $\omega - \text{Var}(S)$  (we can clearly assume without loss of generality that this set is nonempty). Define a renaming function  $\bar{v}$  of  $T$  as follows: for each area  $\bar{M}$  of  $m$  in  $T$  where state  $q_{xc}^j b^k$  is in  $\bar{M}$ ,  $/b/ = /a/$ , and  $R(b^k)[i] = m$ , let  $\bar{v}(\bar{M}, m) = r_i$ ;  $\bar{v}(\bar{N}, n) = n$  otherwise. It can be verified that, since  $T$  is a tree schema,  $\bar{v}$  satisfies #4.3.1.B), while clearly  $\bar{v}$  satisfies #4.3.1.A). Now, let  $T'' = (Q'', q_0'', \delta'')$  be such that  $T \xrightarrow{\bar{v}} T''$ . For all  $c^j$ , all terminators  $b^k$  (such that  $/b/ = /a/$ ) in  $\phi(q_{xc}^j)$  have been renamed in  $T''$  to be terminators  $a^k$  of the same operation  $a$ . Since  $T$  is consistent,  $T''$  is a schema.

Finally, define  $S' = (Q', q_0', \delta')$  from  $T''$  as follows:  $Q'' \subseteq Q'$ ,  $q_0'' = q_0'$ ,  $\delta'' \subseteq \delta'$ . Furthermore,  $Q'$  contains a new state  $\delta'(q_x, a^k) = q_{xa}^k$  for each terminator  $a^k$  of  $a$ , and for each such  $q_{xa}^k$  and each  $c^j$  such that  $(q_{xc}^j)$ ,  $\delta'(q_{xa}^k, c^j) = q_{xc}^j a^k$ .  $S'$  does not contain anything else.

The construction of schema  $S'$  from schema  $T''$  is demonstrated in Fig. 1.

We shall now proceed to show that Construction 6 produces in fact the desired results. The following is immediate from the definitions.

7. Lemma. Let  $T, S'$  and  $xc^j b^k$  be as defined above. Then  $\text{Comp}(S')$  contains exactly the following strings:

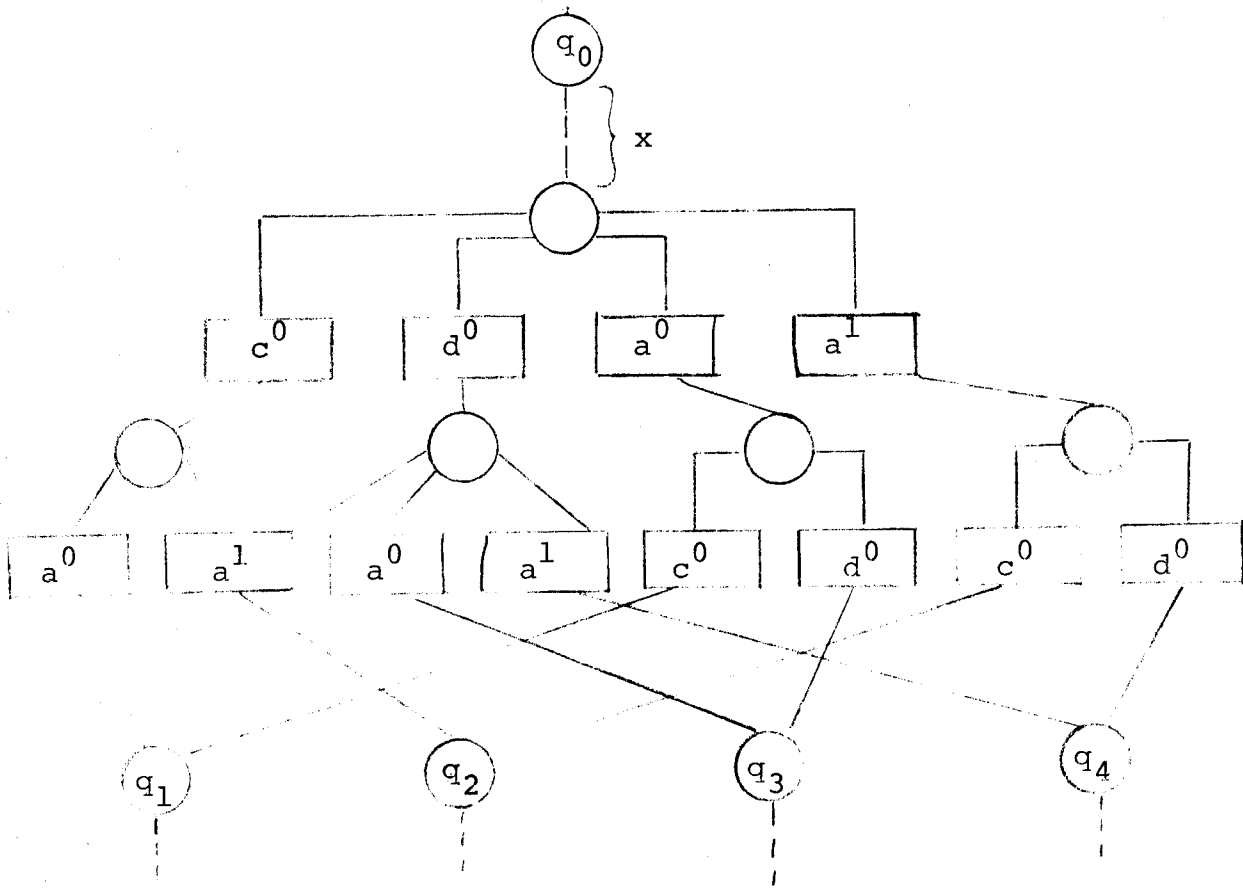
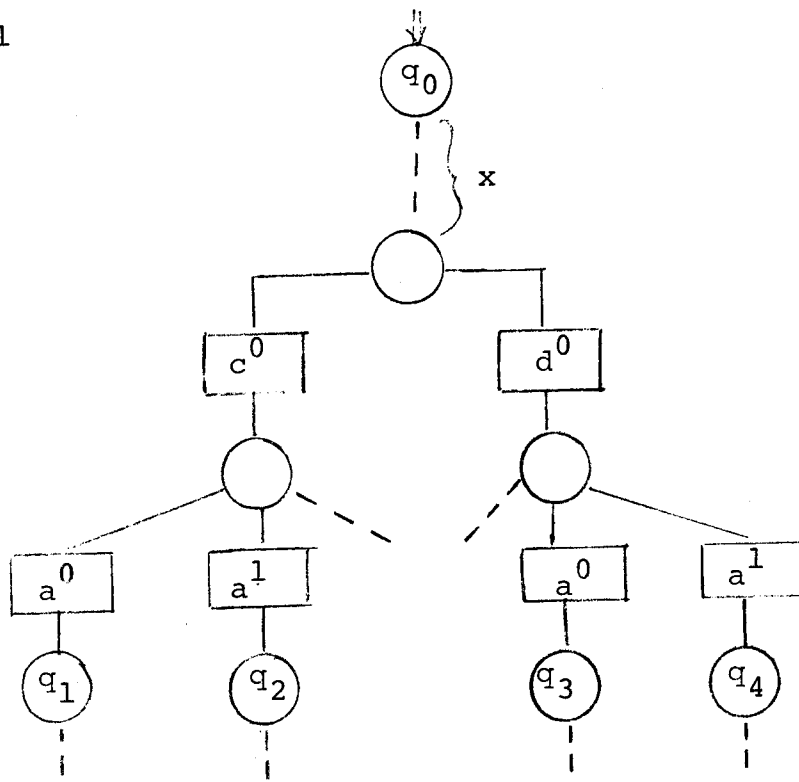


Fig. 1 - Demonstrating part of Construction 6.



all  $z \in \text{Comp}(T)$  such that  $xc^{j_b^k} \leq z$  is false  
 for each string  $xc^{j_b^k}y \in \text{Comp}(T)$ , a string  $xc^{j_a^k}y'$  and  
 a string  $xa^{k_c^j}y'$ , where  $xc^{j_a^k}y' \leftrightarrow xc^{j_b^k}y$ .

From [K&M] we have the following definition. Let  
 $a, c \in A$ . We write  $a \rho c$  iff  $(D(a) \cap R(c)) \cup (R(a) \cap D(c))$   
 $\cup ((Ra) \cap R(c)) \neq \emptyset$ . As a direct consequence of results proved  
 in [K&M] and [Kel] we have:

8. Lemma. Let  $a \not\rho c$  for any  $a, c \in A$ . Then  $xc^{j_a^k}y \equiv xa^{k_c^j}y$   
 for all  $x \in \Sigma^*$ ,  $y \in \hat{\Sigma}$ ,  $a^{k_c^j} \in \Sigma$ .

We can now prove that  $S'$  is actually more parallel  
 than  $T$  and  $S$ , and is restricted.

9. Theorem. If  $S$  is a restricted schema such that  $\text{Adv}^1(/a/, x)$   
 is true and  $S'$  is defined from the tree schema of  $S$  by  
 Construction 6, then  $S'$  is restricted and  $S' > S$ .

Proof. We first check that  $S'$  is restricted. It is easily  
 seen that  $S'$  is repetition-free and consistent if  $S$  is so.

As concerns quasi-determinacy, note that by #4.4.6,  
 $T''$  is quasi-determinate as a consequence of the fact that  $T$   
 is quasi-determinate. If  $u$  is any string in  
 $\text{Comp}(S') - \text{Comp}(T'')$ ,  $u = xa^{k_c^j}y'$ , where  $xc^{j_a^k}y' \in \text{Comp}(T'')$ .  
 However, by the definition of  $\text{Adv}^1$  we know that  
 $R(c) \cap D(a) = \emptyset$  and since by Construction 6,  $R(a)$  contains  
 variables that do not appear in any other segment in  $u$ , we

have  $a \notin c$ . Hence, by #8,  $xc^j a^k y' \equiv xa^k c^j y'$  and  $S'$  is quasi-determinate.

The facts that  $S'$  is totally defined and finitely branching derive easily from Construction 6.

Finally, by definition we have  $S \stackrel{\sim}{=} T \leftrightarrow T''$ , thus by #4.4.4  $S \sim T''$ . By the argument above we know that for each  $u \in \text{Comp}(S') - \text{Comp}(T'')$  there exists  $u' \in \text{Comp}(T'')$  such that  $u' \equiv u$ . Hence  $T'' \equiv S'$  and  $S \sim S'$ . Clearly by #7  $S' > S$ .

Construction 6 has the disadvantage of almost always yielding an infinite schema, even in the case where  $S$  is finite. Fortunately however, if  $S$  is finite the construction can be replaced by an algorithm that produces a finite schema. The algorithm uses an intermediate finite schema  $S_F$  in place of the (normally infinite) tree schema  $T$ .  $S_F$  is of the form shown in Fig. 2 :  $S$  is expanded into a tree from the initial state to each state  $q_{xc^j b^k}$ . All the transitions that do not lead to such states end into distinct copies of  $S$ .

10. Algorithm. Let  $\text{Adv}^1(/a/,x)$  be true in a finite schema  $S$ . Construct  $S_F$  from  $S$  as follows. For each  $y$  such that  $y < xc^j b^k$  for some  $xc^j b^k$  such that  $/b/ = /a/$  and  $c^j$  does not affect  $b^k$ ,  $S_F$  contains a state  $\langle y \rangle$ . Furthermore, for each minimal  $y$  such that  $y < xc^j b^k$  is false,  $S_F$  contains a set of states  $\{q^Y : q \text{ is a state of } S\}$ . The initial state of  $S_F$  is  $\langle \lambda \rangle$ . The state transition function of  $S_F$  is as

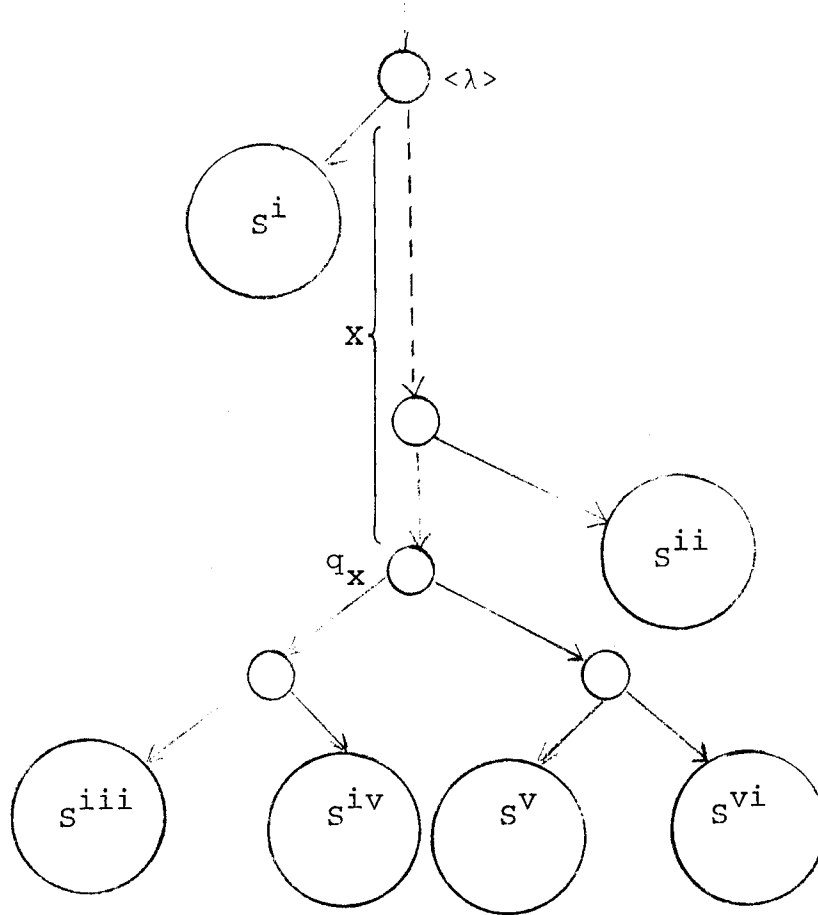


Fig. 2 - Schema  $S_F$ .  $s^i, \dots, s^{vi}$  are copies of  $S$ .

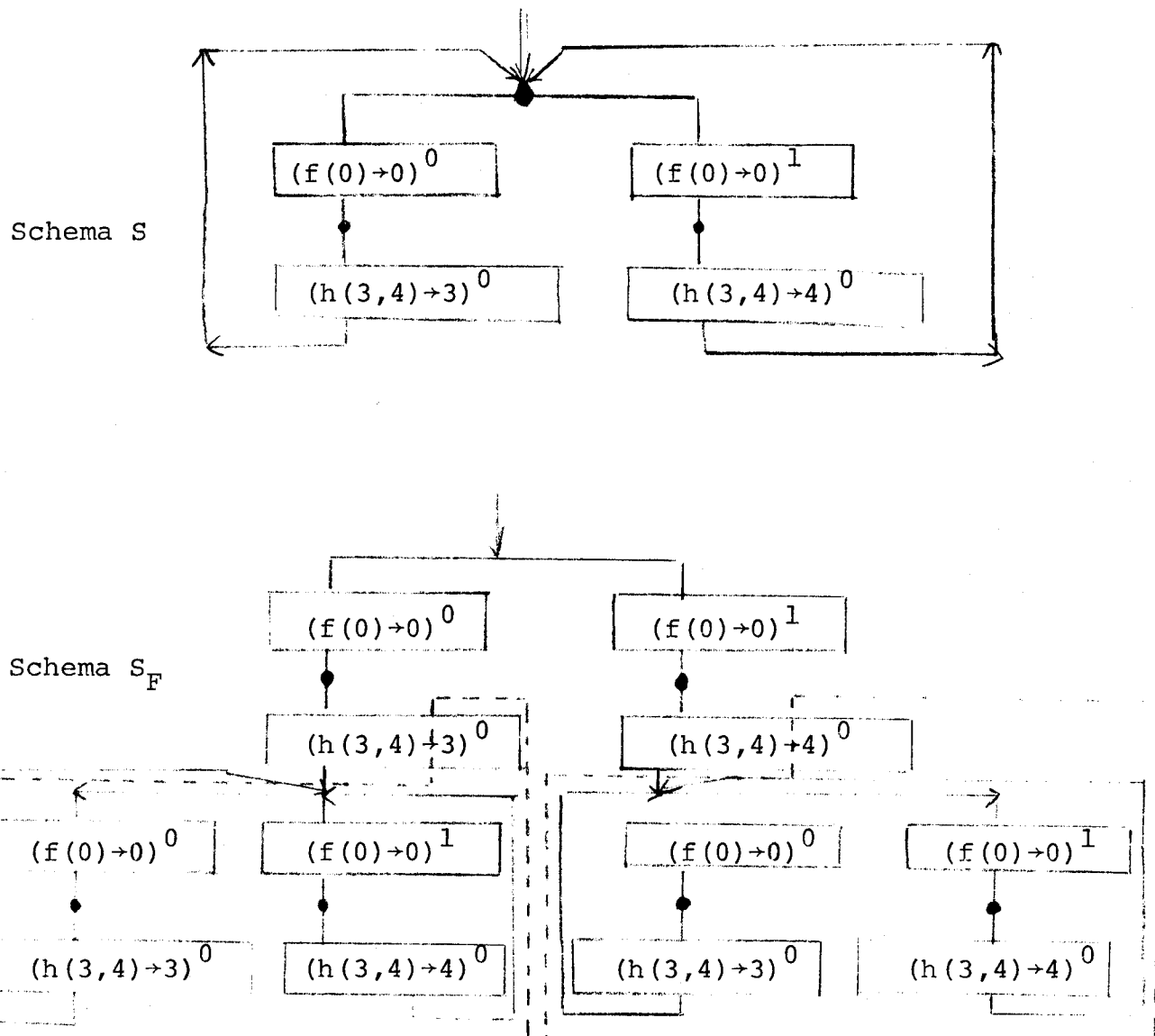


Fig. 3.A) Demonstrating Algorithm 10.

$\text{Adv}^1(h(3,4), \lambda)$  is true in schema S. At the first step, we unwind the loops so that the appropriate renaming can be performed. Subsequently, 3 is renamed as 5 in the left dotted area, and 4 as 5 in the right dotted area. In this case, the renaming is used to identify two operations, rather than to eliminate a memory conflict.

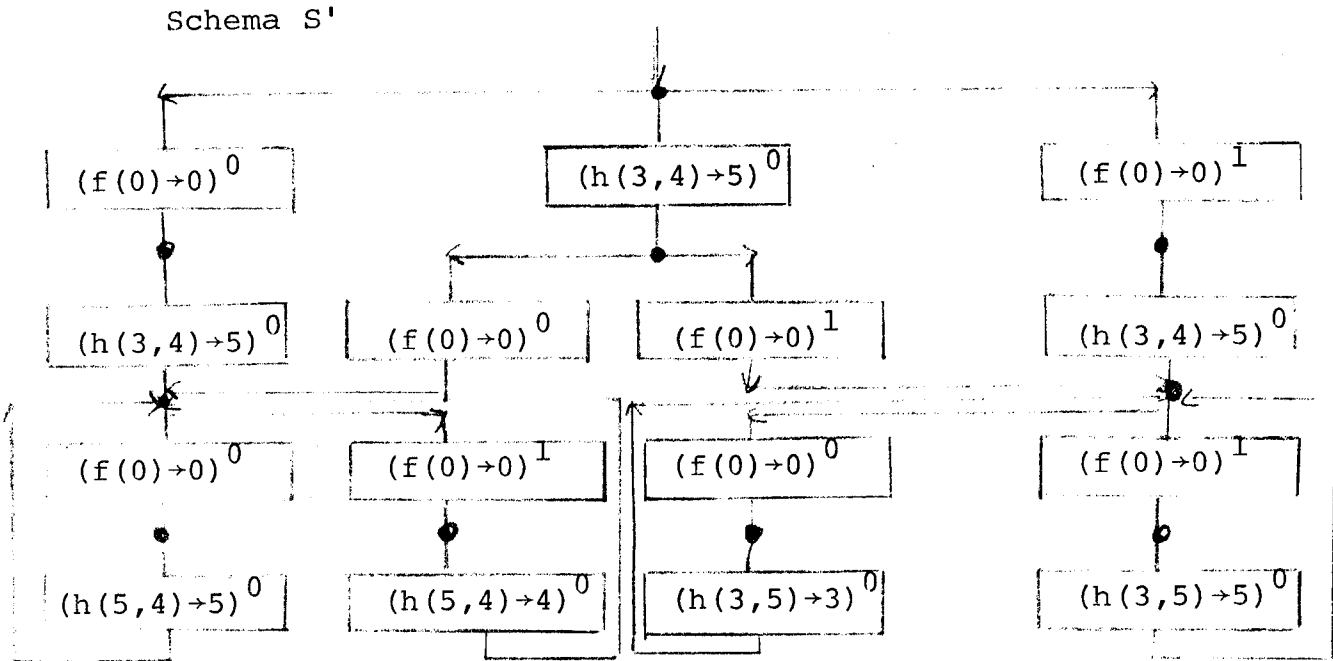


Fig. 3.B) Completing Algorithm 10, the last step.

$(h(3,4) \rightarrow 5)^0$  is now enabled at the initial state.

follows:  $\delta(\langle y \rangle, a^k) = \langle ya^k \rangle$  if  $\langle ya^k \rangle$  is a state of  $S_F$ . Otherwise  $\delta(\langle y \rangle, a^k) = p^Y$  where  $p = q_{ya^k}$  in  $S$ . Finally, for all states  $q^Y$  as above.  $\delta(q^Y, b^j) = p^Y$  iff  $\delta(q, b^j) = p$  in  $S$ . It is easily seen that  $S \sim S_F$ . Also, one verifies that the rest of Construction 6 can be modified to apply to  $S_F$  instead of  $T$ , and that  $S_F$  has all the properties needed for the construction to be well-defined. We call the resulting schema  $S'_F$ .

The algorithm is demonstrated in Fig. 3.

Since #7 holds for  $S_F$  and  $S'_F$ , we get the following result by the reasoning used to prove #9.

11. Proposition. If  $S$  is a finite, restricted schema such that  $\text{Adv}^1(/a/,x)$  is true, and  $S'_F$  is constructed from  $S$  by applying Algorithm 10, then  $S'_F$  is finite, restricted and such that  $S'_F > S$ .

### §6.2 Enhancing parallelism in schemas: the further steps.

The next question is whether iteration of the constructions discussed in the previous section on a restricted schema where an operation can be advanced by more than just one step eventually yields a schema where the operation is actually enabled as early as possible. We shall answer this question affirmatively. First of all, we need several preliminary results.

We say that Ult(/a/,x) is true in schema  $S$  if

$x \in \text{Pref}(S)$  and for all  $y \in \text{Comp}(S)$  such that  $y \geq x$  there exists  $u$  such that  $y > xu$ ,  $/a/ \in / \phi(q_{xu}) /$  and  $u$  does not affect  $D(a)$ .

We say that  $\text{Adv}(/a/,x)$  is true in  $S$  if  $\text{Ult}(/a/,x)$  is true and  $/a/ \notin / \phi(q_x) /$ , i.e.  $/a/$  cannot be immediately executed after  $x$ .

In other words,  $\text{Ult}(/a/,x)$  is true if  $/a/$  is ultimately enabled after  $x$  and is not affected by intervening operations.  $\text{Adv}(/a/,x)$  is true if  $\text{Ult}(/a/,x)$  is true, but  $/a/$  is not enabled immediately after  $x$ . We shall see that in this case  $/a/$  can be advanced to immediately after  $x$ .

1. Definition. For a schema  $S$ ,  $a \in A$ ,  $x \in \text{Pref}(S)$ , we define the critical set of  $/a/$  and  $x$  in  $S$ ,  $\text{Crit}(/a/,x)$ , to be the set  $\{y : y \geq x \text{ and for all } z \text{ such that } y \geq z \geq x, \text{Adv}(/a/,z)\}$ .

This means that the critical set is the set of all those strings  $y \geq x$  such that  $\text{Ult}(/a/,x)$  is true but  $/a/$  is not enabled at any point between  $x$  and  $y$ , nor immediately after  $y$ .

2. Proposition. For a repetition-free schema  $S$ ,  $\text{Crit}(/a/,x)$  is a nonempty, finite subset of  $\Sigma^*$  iff  $\text{Adv}(/a/,x)$  is true.

Proof. If  $\text{Crit}(/a/,x)$  is nonempty then by definition  $\text{Adv}(/a/,x)$  is true.

Conversely, assume that  $\text{Adv}(/a/,x)$  is true. Then

$\text{Crit}(/a/,x)$  contains  $x$  and is nonempty. To show that  $\text{Crit}(/a/,x)$  is finite we use an argument similar to the one used in the proof of Konig's Lemma [Knu]. Assume that  $\text{Crit}(/a/,x)$  is infinite. Since  $S$  is finitely-branching there exists  $b_1^k$  such that  $xb_1^k$  is in  $\text{Crit}(/a/,x)$  and is the prefix of an infinite number of elements in  $\text{Crit}(/a/,x)$ . Again, there exists  $b_2^j$  such that  $xb_1^k b_2^j$  is the prefix of an infinite number of elements in  $\text{Crit}(/a/,x)$  and the process does not terminate. By the repetition-freedom (and freedom) of  $S$ ,  $xb_1^k \dots b_i^h$  for all  $i \in \omega$  is in  $\text{Pref}(S)$  and there exists  $xw \in \text{Comp}(S)$  such that for no  $u < w$  is  $/a/$  an element of  $/\phi(q_{xu})/$ , a contradiction with the fact that  $\text{Ult}(/a/,x)$  is true. It also follows that every  $y \in \text{Crit}(/a/,x)$  is in  $\Sigma^*$ .

For finite schemas, we have the following decidability result:

**3. Proposition.** For a repetition-free finite schema, and for any  $a \in A$  and  $x \in \Sigma^*$  it is decidable whether  $\text{Adv}(/a/,x)$  is true. Furthermore,  $\text{Crit}(/a/,x)$  is computable.

Proof. A repetition-free schema  $S$  is free, and thus  $\text{Pref}(S) = L_S$  where, if  $S$  is finite,  $L_S$  is a regular language. By elementary finite automata theory,  $x \in \text{Pref}(S)$  is decidable. Furthermore, if for some  $y \in \text{Comp}(S)$  such that  $y \geq x$  there exists  $u$  such that  $y > xu$ ,  $/a/ \in / \phi(q_{xu}) /$  and  $u$  does not affect  $D(a)$ , then there



exists such  $u$  of length not exceeding  $|Q| - 1$ . Therefore, if  $S$  is a repetition-free and finite schema such that  $\text{Ult}(/a/,x)$  is true, for all  $y \in \text{Comp}(S)$  such that  $y \geq x$  there must exist  $u$  of length not exceeding  $|Q| - 1$  such that  $y > xu$ ,  $/a/ \in / \phi(q_{xu}) /$ , and  $u$  does not affect  $D(a)$ . An algorithm to decide whether  $\text{Adv}(/a/,x)$  is true needs only to enumerate, according to some lexicographic ordering, all  $w \in \Sigma^*$  such that  $xw \in \text{Pref}(S)$  but for all  $z$  such that  $z \leq w$ ,  $/a/ \notin / \phi(q_{xz}) /$ , and  $w$  does not affect  $D(a)$ . If during the enumeration process we obtain a string  $w$  such that  $|w| = |Q|$ , or a string  $w$  that affects  $D(a)$ , or we reach a final state, then  $\text{Ult}(/a/,x)$  is false, and the process is interrupted. Otherwise, if the process reaches the end, the set of strings that has been enumerated is  $\text{Crit}(/a/,x)$ , and if this set is nonempty  $\text{Adv}(/a/,x)$  is true.

In practice, the computation can be performed by constructing a tree as shown in Fig. 4.

An operation can be advanced iff it can be advanced by just one step:

4. Proposition. Let  $S$  be a schema,  $y \in \text{Pref}(S)$ .  $\text{Adv}(/a/,y)$  is true iff  $\text{Adv}^1(/a/,x)$  is true for all maximal  $x \in \text{Crit}(/a/,y)$ .

Proof. If  $\text{Adv}(/a/,y)$  is true and  $x \in \text{Crit}(/a/,y)$  then

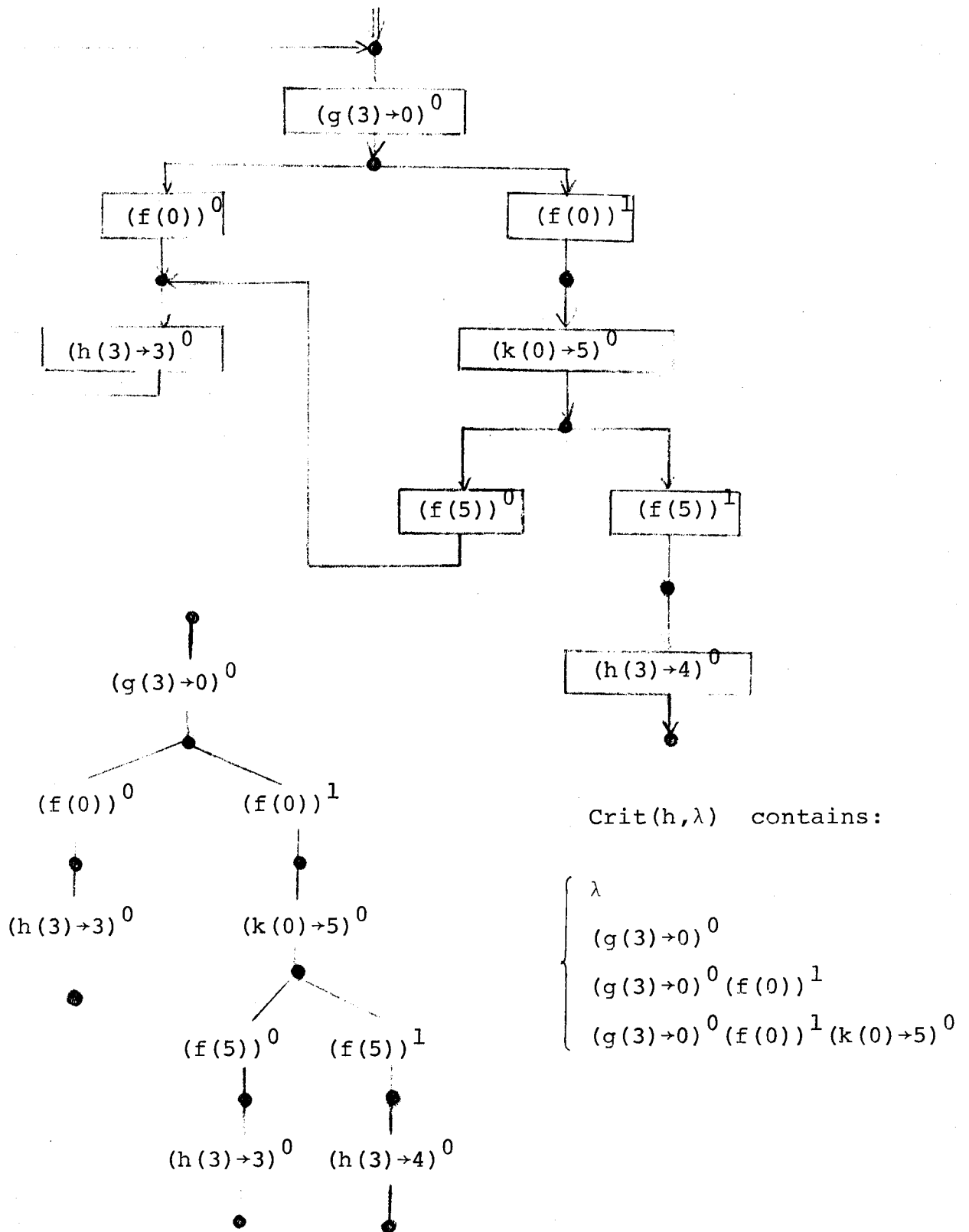


Fig. 4. A schema and the tree used in the computation of  $\text{Crit}(h(3), \lambda)$ . It is then seen that  $\text{Adv}(h(3), \lambda)$  is true.

$\text{Adv}(/a/,x)$  and  $\text{Ult}(/a/,x)$  are true, i.e. for all  $z \in \text{Comp}(S)$  such that  $z \geq x$  there exists a nonempty  $u$  such that  $z > xu$ ,  $/a/ \in / \phi(q_{xu}) /$  and  $u$  does not affect  $D(a)$ . Suppose  $u = b^k w$  for some  $b^k$ , where  $/a/ \notin / \phi(q_{xb^k}) /$  and  $x$  is maximal in  $\text{Crit}(/a/,y)$ . Then  $\text{Ult}(/a/,xb^k)$  and  $\text{Adv}(/a/,xb^k)$  are true and  $xb^k \in \text{Crit}(/a/,x)$ , contradicting the maximality of  $x$ . Hence  $/a/ \in / \phi(q_{xb^k}) /$  and  $\text{Ult}^1(/a/,x)$ ,  $\text{Adv}^1(/a/,x)$  are true.

Conversely, it is obvious that if  $\text{Adv}^1(/a/,y)$  is true, so is  $\text{Adv}(/a/,y)$ .

Referring to the example of Fig. 4, one verifies that  $\text{Adv}^1(h(3), (g(3)+0)^0 (f(0))^1 (k(0)+5)^0)$  is true.

The next thing to be verified is the fact that by iterating Construction 1.6 (or Algorithm 1.10), the number of critical elements is in fact decreased. This is our #10 and its proof will require several preliminary results.

5. Lemma. Let  $x$  and  $x'$  be repetition-free, similar computations such that  $x = v u a^k z$  where  $u$  does not affect  $D(a)$ , and  $x' \geq v'$  where  $v \leftrightarrow v'$ . Then  $x' = v' u' b^k z'$  where  $u'$  does not affect  $D(b)$  and  $v a^k \leftrightarrow v' b^k$ .

Proof. If  $x$  and  $x'$  are repetition-free and similar then there must exist the bijection  $h^\Sigma$  required by #3.6.1. By #3.5.7,  $h^\Sigma$  is the identity for  $v$  and  $v'$ . Now, let  $a^k$  be the  $i$ -th terminator in  $x$ : there must exist  $j > |v'|$

such that  $\text{Char}(x,i) = \text{Char}(x',j)$ , i.e. the set of values fetched by  $x[i]$  and  $x'[j]$  are the same. Since the set of values fetched by  $x[i]$  is in memory after  $v$ , by #3.5.5 the set of values fetched by  $x'[j]$  must also be in memory after  $v'$ . Hence by the liberality of  $x'$  no  $x'[n]$  affects  $D(x'[j])$ , for  $n \in \{|v| + 1, \dots, j - 1\}$ . If  $x'[j] = b^k$  then there must exist an identity bijection between the sets of characteristics of  $va^k$  and  $v'b^k$  and by #3.6.4,  $va^k \leftrightarrow v'b^k$  as desired.

6. Proposition. Let  $S$  be a totally defined schema. Then for all interpretations  $I$ ,  $\text{Pref}(S,I) = L_S \cap L_I$ .

Proof.  $\text{Pref}(S,I) \subseteq L_S \cap L_I$  is true for any schema by #4.4.1. For the converse, we see that for all  $x \in L_S \cap L_I$  either  $x$  is an  $S$ -string, in which case by #4.4.1  $x \in \text{Comp}(S,I)$  and  $x \in \text{Pref}(S,I)$ , or  $x$  is not an  $S$ -string. In the latter case, there must exist  $b^i \in \phi(q_x)$ . If  $\Gamma_{F(b)}(c_x[D(b)]) = j$  then by the fact that  $S$  is totally defined  $b^j \in \phi(q_x)$ , hence  $xb^j \in L_S \cap L_I$ . By iterating this argument, we see that either we eventually reach a final state, in which case  $x$  is the prefix of a finite element of  $\text{Comp}(S,I)$ , or we always have a way to continue the computation, in which case  $x$  is the prefix of an infinite element of  $\text{Comp}(S,I)$ .

As a consequence, the class of restricted schemas enjoys of a property that is similar to the "persistency" of [K&M], [Kell]: if  $a^i$  becomes enabled at a certain point of

a computation  $y$  then some  $b^k$  such that  $/b/ = /a/$  is ultimately executed. Part B) of the following proposition further strengthens the result, by showing that ultimately enabled implies ultimately executed.

7. Proposition. Let  $S$  be a repetition-free, quasi-determinate, totally defined schema. Then the following hold:

A) If  $va^i \in \text{Pref}(S)$  then for all  $y \in \text{Comp}(S)$  such that  $v \leq y$ ,  $vub^k \leq y$  for some  $u$  that does not affect  $D(b)$  and some  $b^k$  such that  $/b/ = /a/$  (this implies the truth of  $\text{Ult}(/a/,v)$ ).

B) If  $\text{Ult}(/a/,v)$  is true then for all  $y \in \text{Comp}(S)$  such that  $y \geq v$ ,  $y = vuc^k$  for some  $u$  that does not affect  $D(c)$  and some  $c^k$  such that  $/c/ = /a/$ .

Proof.

A) Assume  $y \in \text{Comp}(S,I)$  for some interpretation  $I$ , and let  $y = vc^h w$  for some  $c^h$ . If  $/c/ = /a/$  then we are of course finished. Otherwise, since  $va^i \in \text{Pref}(S)$  and  $S$  is totally defined, there must be  $a^k$  such that  $va^k \in L_S \cap L_I$ . By #6,  $va^k \in \text{Pref}(S,I)$  and  $va^k z \in \text{Comp}(S,I)$  for some  $z$ . By the quasi-determinacy of  $S$ ,  $va^k z \sim vc^h w = y$ . Thus applying #5 with  $u = \lambda$  and  $v = v'$ ,  $vc^h u'' b^k \leq y$  for some  $c^h u'' = u'$  that does not affect  $D(b)$  and some  $b^k$  as desired.

B) If  $\text{Ult}(/a/,v)$  is true then for all  $y \in \text{Comp}(S)$  such that  $y \geq v$  there exists  $u'$  such that  $y > vu'$ ,  $/a/ = / \phi(q_{vu'}) /$  and  $u'$  does not affect  $D(a)$ . Then

$vu'b^i \in \text{Pref}(S)$  for some  $b$  such that  $/b/ = /a/$  and since  $vu' < y$ , by A) we have  $wu'u''c^k \leq y$  for some  $u''$  that does not affect  $D(c)$  or  $D(a)$  and some  $c^k$  such that  $/c/ = /a/$ , as desired.

As an immediate consequence of the definitions of similarity and quasi-determinacy we have:

8. Proposition. For any two similar, quasi-determinate schemas  $S$  and  $S'$  and all interpretations  $I$ , if  $x \in \text{Comp}(S, I)$  and  $y \in \text{Comp}(S', I)$  then  $x \sim y$ .

Proof. By the similarity of  $S$  and  $S'$  for all  $x \in \text{Comp}(S, I)$  there must exist  $z \in \text{Comp}(S', I)$  such that  $x \sim z$ . By the quasi-determinacy of  $S'$ ,  $z \sim y$ ; hence  $x \sim y$ .

Lemma 5 can then be extended to schemas.

9. Proposition. Let  $S$  and  $S'$  be similar, repetition-free, quasi-determinate, totally defined schemas, and let  $v \in \text{Pref}(S)$ ,  $\bar{v}' \in \text{Pref}(S')$  and  $a, b \in A$  be such that  $va^k \leftrightarrow v'b^k$ . Then  $\text{Ult}_S(/a/, v)$  is true iff  $\text{Ult}_{S'}(/b/, \bar{v}')$  is true.

Proof. Consider any  $x'$  such that  $x' \geq \bar{v}'$ , and  $x' \in \text{Comp}(S', I)$  for some  $I$ . By #3.5.8  $v \sim \bar{v}'$ , hence both  $v$  and  $\bar{v}'$  are in  $L_I$  and (by #6)  $v \in \text{Pref}(S, I)$ . Let  $x \in \text{Comp}(S, I)$  be such that  $x \geq v$ . By #8,  $x \sim x'$ . If  $\text{Ult}_S(/a/, v)$  is true then by #7.B)  $x = vuc^k z$  for some  $u$  that does not affect

$D(c)$  and some  $c^k$  such that  $/c/ = /a/$ . Hence by #5  $x' = v'u'd^kz'$  where  $u'$  does not affect  $D(d)$  and  $vc^k \leftrightarrow v'd^k$ . Clearly  $/d/ = /b/$ . Since this is true for any  $x' \geq v'$ ,  $\text{Ult}_{S'}(/b/,v')$  is true.

The same argument holds in the direction from  $S'$  to  $S$ .

Finally, the following Lemma provides the induction step for the main result of this section.

10. Lemma. Let  $\text{Adv}(/a/,y)$  be true in a restricted schema  $S$ , and let  $x$  be a maximal element in  $\text{Crit}_S(/a/,y)$ . Then  $\text{Adv}_S^1(/a/,x)$  is true and if  $S'$  is the schema defined from  $S$  by Construction 1.6 (or Algorithm 1.10 if  $S$  is finite), the following holds:  $\text{Crit}_{S'}(/a/,y) = \text{Crit}_S(/a/,y) - \{x\}$ . Proof. Note that if  $\text{Adv}_S(/a/,y)$  is true then by #2  $\text{Crit}_S(/a/,y)$  is a nonempty, finite subset of  $\Sigma^*$  and contains a maximal element  $x$ . By #4,  $\text{Adv}_S^1(/a/,x)$  is true. Thus Construction 1.6 (or Algorithm 1.10) can be applied to schema  $S$ .

First of all, we prove that every element of  $\text{Crit}_S(/a/,y)$ , except  $x$ , is in  $\text{Crit}_{S'}(/a/,y)$  (the fact that  $x \notin \text{Crit}_{S'}(/a/,y)$  is obvious by #1.6). Since  $y \in \text{Pref}(S')$ ,  $(\text{Crit}_{S'}(/a/,y))'$ . Consider now any  $u \in \text{Crit}_S(/a/,y) - \{x\}$ . For all  $z$  such that  $u \geq z \geq y$ ,  $\text{Ult}_{S'}(/a/,z)$  is true by #9 and the fact that  $\text{Ult}_S(/a/,z)$  is true. Next, assume that for some such  $z$  it is false

that  $\text{Adv}_S(/a/,z)$ . This means that  $/a/ \in / \phi(q_z) /$ , i.e.  $zb^k \in \text{Pref}(S')$  for some  $b^k$  such that  $/b/ = /a/$ . However, since  $\text{Adv}_S(/a/,z)$  is true, for all  $c$  such that  $/c/ = /a/$ ,  $zc^k \notin \text{Pref}(S)$ . Thus  $zb^k \in \text{Pref}(S')$  contradicts #1.7. Hence  $\text{Adv}_S(/a/,z)$  is true and  $u \in \text{Crit}_S(/a/,y)$ . This proves that  $\text{Crit}_S(/a/,y) - \{x\} \subseteq \text{Crit}_S(/a/,y)$ .

Next, we want to check that every element of  $\text{Crit}_S(/a/,y)$  is in  $\text{Crit}_S(/a/,y)$ . Assume  $u \in \text{Crit}_S(/a/,y)$ . By definition,  $u$  cannot be such that  $xc^j a^k \leq u$ , or such that  $xa^k \leq u$  (where  $c^j$  and  $a^k$  are as defined in Construction 1.6). Thus by #1.7,  $u \in \text{Pref}(S)$ . Also, by #9 for all  $z$  such that  $u \geq z \geq y$ ,  $\text{Ult}_S(/a/,z)$  is true. Assume now that for some such  $z$  it is false that  $\text{Adv}_S(/a/,z)$ . This means that  $/a/ \in / \phi(q_z) /$  i.e.  $zb^k \in \text{Pref}(S)$  for some  $b^k$  such that  $/b/ = /a/$ . But then by #1.7  $zb^k \in \text{Pref}(S')$  also, a contradiction.

The proof for Algorithm 1.10 follows by the same reasoning since #1.7 holds for  $S$  and  $S'_F$ .

By induction on the number of elements in  $\text{Crit}_S(/a/,y)$  it is then not difficult to obtain the main result of this section:

11. Theorem. (Long-distance desequencing) Let  $S$  be a restricted schema such that  $\text{Adv}_S(/a/,x)$  is true. Then there exists a restricted schema  $S'$  such that  $/a/ \in / \phi(q_x) /$  and  $S' > S$ . If  $S$  is finite, then there exists a finite,



restricted schema  $S''$  that has the same properties, and that can be effectively computed from  $S$ .

### §6.3 Maximal parallelism.

We shall now introduce two "natural" definitions of the notion of maximal parallelism in schemas, and we shall show that for restricted schemas these definitions are equivalent.

We say that a restricted schema  $S$  is hyperclosed if for all restricted  $S'$  such that  $S' \sim S$ ,  $S \geq S'$ .  $S$  is prompt if for all  $x \in \text{Pref}(S)$  and  $a \in A$ ,  $\text{Adv}(/a/,x)$  is false.

Thus a restricted schema  $S$  is hyperclosed if no restricted schema  $S'$  is more parallel than  $S$ , where "more parallel" is taken in the sense of #1.1. A schema is prompt if it enables every operation as early as possible.

1. Theorem. A restricted schema is hyperclosed iff it is prompt.

Proof. The forward direction follows immediately from #2.11, where we have seen that if a restricted schema  $S$  is not prompt then there exists a restricted schema  $S'$  such that  $S' > S$ , thus implying that  $S$  is not hyperclosed.

For the converse, assume that  $S$  is not hyperclosed, and for some restricted  $S'$  such that  $S' > S$  consider a shortest  $x'b^k \in \text{Pref}(S',I) - \text{Pref}(S,I)^0$  for any  $I$  (there must be such  $x'b^k$  by #1.5 and #1.2). Clearly

$x' \in \text{Pref}(S', I) \cap \text{Pref}(S, I)^0$ , i.e. there exists  $x \in \text{Pref}(S, I)$  such that  $x \leftrightarrow x'$ . By #2.7.A),  $\text{Ult}_S(/b/, x')$  is true, and by #2.9  $\text{Ult}_S(/a/, x)$  is true for some  $a$  such that  $xa^k \leftrightarrow x'b^k$ . However, by the way in which we have chosen  $x$ , for all such  $a$ , we have  $/a/ \notin / \phi(q_x) /$  in  $S$ . Thus  $\text{Adv}_S(/a/, x)$  is true and  $S$  is not prompt.

The following is then immediate:

2. Proposition. It is decidable whether a finite, restricted schema is prompt (and hence hyperclosed).

Proof. If  $q_x \simeq q_z$  in a restricted schema  $S$  then for all  $y \in \text{Comp}(S)$ ,  $y \geq x$  if  $y \geq z$ . Thus for all  $a$ ,  $\text{Ult}(/a/, x)$  iff  $\text{Ult}(/a/, z)$  and  $\text{Adv}(/a/, x)$  iff  $\text{Adv}(/a/, z)$ . As a consequence, if  $q_x = q_z$ ,  $\text{Adv}(/a/, x)$  iff  $\text{Adv}(/a/, z)$ .

An algorithm for deciding whether a finite, restricted schema is prompt is therefore the following. For each state  $q \in Q$ , if  $q = q_x$  for some  $x$ , decide whether there exists  $/a/$  such that  $\text{Adv}(/a/, x)$  is true. Only those  $/a/$  such that  $(\delta(q', a^k))$  for some  $q' \in Q$  and some  $k$  need to be considered. By #2.3 the predicate  $\text{Adv}$  is decidable. By definition, a schema is prompt iff such a  $q$  and  $/a/$  cannot be found.

For a restricted schema  $S$ , we say that a restricted schema  $S'$  is a hyperclosure of  $S$  if for all  $S''$  such that  $S'' \sim S$ ,  $S' \geq S''$ . The following result states that two similar

schemas must have the same hyperclosure, up to proper renaming.

3. Proposition. Let  $S$  and  $S'$  be restricted schemas such that  $S \sim S'$ , and let  $\tilde{S}$  be any hyperclosure of  $S$ ,  $\tilde{S}'$  be any hyperclosure of  $S'$ . Then:

- A)  $\text{Pref}(\tilde{S})^0 = \text{Pref}(\tilde{S}')^0$ ;
- B)  $\text{Pref}(\tilde{S}) \leftrightarrow \text{Pref}(\tilde{S}')$ ;
- C)  $T \leftrightarrow T'$  for the tree schemas  $T$  and  $T'$  of  $\tilde{S}$  and  $\tilde{S}'$ .

Proof.  $\tilde{S} \sim S \sim S' \sim \tilde{S}'$  : thus  $\tilde{S} \geq \tilde{S}'$  and  $\tilde{S}' \geq \tilde{S}$ , i.e. by #1.5  $\text{Pref}(\tilde{S})^0 \subseteq \text{Pref}(\tilde{S}')^0$  and  $\text{Pref}(\tilde{S}')^0 \subseteq \text{Pref}(\tilde{S})^0$ . B) follows by #1.3 and C) by #4.6.1.

The main result of this section will show in part A) that any finite, restricted schema has a hyperclosure. Part B) of the result can be interpreted in the following sense: there exists a procedure that, fed with any such schema, will behave in the same way as one of its hyperclosures. This procedure realizes the look-ahead mechanism that we have discussed in Chapter 1, in that operations are systematically "moved up" as far as possible in the schema one at a time.

First of all, we need a Lemma, that can be proved by a reasoning very similar to the one used in the proof of #2.3:

4. Lemma. In a restricted, finite schema, for all  $x \in \text{Pref}(S)$  the set of those  $/a/$  such that  $\text{Ult}(/a/,x)$  is true is a finite and effectively computable set.

5. Theorem. Let  $S$  be a restricted, finite schema. Then:

- A) A hyperclosure  $\tilde{S}$  of  $S$  exists.
- B) For all  $x \in \Sigma^*$ ,  $\phi(q_x)$  in  $\tilde{S}$  is computable.

Proof. We shall show that there exists a tree schema  $S'$  such that B) is true and then we shall show that  $S'$  is the desired hyperclosure. For each  $x \in \Sigma^*$  we define  $\phi(q_x)$  in  $S'$ , that we shall write  $\phi'(q_x)$ , by induction on the length of  $x$ . Our definition yields an algorithm for computing  $\phi'(q_x)$  for an  $x$  of arbitrary length.

To compute  $\phi'(q_\lambda)$  consider the set  $A^\lambda = \{ /a/ : \text{Ult}(/a/, \lambda) \text{ is true in } S \}$ . By #4 this set is finite and computable. By repeated use of #2.11 we know that there exists an effectively computable finite schema  $S^\lambda$ , with initial state  $q_0$ , such that  $S^\lambda \sim S$  and  $/a/ \in / \phi(q_0) /$  for all  $/a/ \in A^\lambda$  (note that by #2.9  $\text{Ult}(/a/, \lambda)$  is true for some  $/a/$  in  $S$  iff it is true in  $S^\lambda$  and in all the intermediate schemas obtained during the construction of  $S^\lambda$ ). Take then  $\phi'(q_\lambda) = \phi(q_0)$ , where  $q_0$  is the initial state of  $S^\lambda$ .

Assume that for some  $x \in \Sigma^*$  we have computed  $\phi'(q_x)$  and a finite schema  $S^x$  such that  $S^x \sim S$  and for all  $/a/$  such that  $\text{Ult}(/a/, x)$  in  $S^x$ ,  $/a/ \in / \phi(q_x) /$ .  $q_{xb^k}$  for some  $b^k$  is defined in  $S'$  iff  $b^k \in \phi'(q_x)$ , i.e.  $b^k \in \phi(q_x)$  in  $S^x$ . Assume then that  $\phi'(q_{xb^k})$  is defined. Again, the set  $A^{xb^k} = \{ /a/ : \text{Ult}(/a/, xb^k) \text{ is true in } S^x \}$  is finite, and it is possible to compute a finite schema  $S^{xb^k}$  such that  $S^{xb^k} \sim S$  and for all  $/a/$  in  $A^{xb^k}$ ,

$/a/ \in / \phi(q_{xb^k}) /$  in  $S^{xb^k}$ . We then take  $\phi'(q_{xb^k}) = \phi(q_{xb^k})$ .

Finally, we shall show that if a hyperclosure  $\tilde{S}$  of  $S$  exists, then  $\text{Pref}(S')^0 = \text{Pref}(\tilde{S})^0$ , thus implying, by #1.5, that  $S'$  is also a hyperclosure of  $S$ . First of all, note that  $x \in \text{Pref}(S')^0$  iff there exists  $S^Y$  such that  $x \leftrightarrow y$  and  $x \in \text{Pref}(S^Y)^0$ . Since  $S^Y \sim S \sim \tilde{S}$ ,  $\tilde{S} \geq S^Y$  and by #1.5  $x \in \text{Pref}(\tilde{S})^0$ . Conversely, assume that  $\text{Pref}(\tilde{S})^0 \subseteq \text{Pref}(S')^0$  is false. By #1.2 there exists a shortest  $x \in \text{Pref}(\tilde{S}) - \text{Pref}(S')^0$ . Then if  $x = za^k$ ,  $z \in \text{Pref}(S')^0$  and  $z \in \text{Pref}(S^V)^0$  for some  $v \leftrightarrow z$ . By #2.7.A),  $\text{Ult}(/a/,z)$  is true in  $S$ , while  $S^V \sim S \sim \tilde{S}$ . Hence, by #2.9,  $\text{Ult}(/b/,v)$  is true in  $S^V$  for some  $b$  such that  $vb^k \leftrightarrow za^k$  and by definition of  $S^V$   $/b/ \in / \phi(q_v) /$  in  $S^V$ , a contradiction.

Before closing this section, we want to note an application of #3 and #5.3.3. In [Pat] it is shown, with a rather long proof, that the equivalence problem is solvable for the class of progressive schemas. We show that, with the changes in definitions that are necessary in order to adapt the problem to our model, this solvability result is a corollary of #3.

We say that a restricted schema is progressive if for all  $x \in L_S$  and all  $i \in \{1, \dots, |x| - 1\}$  the following is true:  $R(x[i]) \cap D(x[i+1]) \neq \emptyset$ . We then have immediately:

6. Lemma. A restricted, progressive schema is prompt and

hyperclosed.

7. Theorem. It is decidable whether two finite, progressive, restricted schemas are similar.

Proof. If  $S$  and  $S'$  are progressive, similar and restricted then by the preceding Lemma and #3 we have:  $L_S \leftrightarrow L_{S'}$ . By #5.3.3 this is decidable for any two restricted, finite schemas.

#### §6.4 Concluding remarks.

The procedure of approximating the hyperclosure of a restricted schema, outlined in the proof of #3.5, does not need to terminate. In fact, by extending an argument due to Keller [Kel] it is possible to show that restricted, finite schemas that do not have finite hyperclosures are not exceptional. Also, there are restricted schemas such that the minimum memory of their hyperclosure is infinite. Worse yet, if a finite schema has a finite hyperclosure, our procedure will not necessarily yield it (this can also be shown by extending an argument of Keller [Kel]). The problem of characterizing the class of restricted schemas that have finite hyperclosures is open. Note that if this class could be characterized, and an algorithm to obtain a finite hyperclosure for the schemas in the class could be given, we would also have characterized a class of schemas where the similarity problem is decidable (this would hold by the same reasoning used in #3.7).

However, the look-ahead procedure mentioned above gives us a way of effectively "simulating" the hyperclosure of any restricted schema, a result of some practical meaning. Note that this result becomes true because of the fact that, for any state of any finite restricted schema, there is only a finite number of operations that can be advanced to that state (see #3.4).

### DIRECTIONS OF FURTHER RESEARCH

We have studied certain classes of transformations of parallel program schemas, in various ways related to the concept of renaming. To do this, we have defined some families of program schemas that were particularly suitable for our investigation. The conditions under which our results could be extended to other families of schemas remain to be studied, and in many cases the extension is not straightforward.

Another area of study are the possible applications of our theory. The most immediate application seems to be in microprogramming. Most microprograms are made of register-to-register transfers, together with computations of elementary functions, and our schemas seem suitable to represent such control structures. Application of our results on memory economy to the design of microprograms could have some practical importance, since the number of registers available to the microprogrammer is usually small. Similar remarks apply to the techniques for increasing program parallelism.

Concerning research of longer range, the author believes that the concepts of segment and area have applications that go far beyond the subjects investigated in this thesis. Itkin [Itk] has recently shown the utility of these concepts for solving a special case of the equivalence problem of schemas, and one can expect further results in the same general direction.

Finally, another application of the same concepts



can be seen in connection with a theory of data structures, as presented by Rosenberg [Ros]. The whole subject of data structures seems to be closely related to the subject of renaming. One possible direction of research is the following. In our thesis, we have considered a "one-state" memory, where the values of all the variables are always accessible. It seems possible to generalize such a concept to the concept of a "memory automaton" in which each variable is capable of several states, as: "accessible" (i.e. the variable's value can be fetched), "available" (i.e. the variable is not accessible, but holds some value and is capable of becoming accessible), and "not available" (i.e., not holding any value). In real-life computing systems, an accessible variable is a variable in main memory, while an available variable is a variable that is held in some kind of back-up memory (as drums, disks, or similar). It is immediate that a variable  $m$  must be accessible when an operation affecting  $m$  is executed, and available when the control is in an area of  $m$ . In such a theory, it would be possible to investigate properties of structured memory systems, and such questions as whether a given memory system is "adequate" for a certain program, whether two memory systems are in some sense "equivalent", and so on.

Another direction of further research has been suggested in Section 5.4 and is a theory of dynamic memory allocation. One could develop the kind of reasoning needed for a formal treatment of the subject, and investigate the

amounts of memory needed by dynamic memory allocation mechanisms under various conditions.

## BIBLIOGRAPHY

- [A&M] E.A. Ashcroft and Z. Manna, Formalization of Properties of Parallel Programs, Machine Intelligence 6(1971), 17-41.
- [A,M&P] E.A. Ashcroft, Z. Manna and A. Pnueli, Decidable Properties of Monadic Functional Schemas, J. of the Assoc. for Computing Machinery, 20(1973), 489-99.
- [A,S&T] D.W. Anderson, F.J. Sparacio and R.M. Tomasulo, Machine Philosophy and Instruction Handling, IBM Journal of Research and Development, 11(1967), 8-24.
- [Buch] W. Buchholz, "Planning a Computer System", McGraw-Hill, New York, 1962.
- [Con] M.E. Conway, A Multiprocessor System Design, Proceedings Fall Joint Computer Conf., 24(1963), 139-46.
- [Gri] D. Gries, "Compiler Construction for Digital Computers", Wiley, New York, 1971.
- [Itk] V.E. Itkin, Logiko-termal'naya Ekvivalentnost Schem Programm, Kibernetika (1972), 5-28.
- [Kel] R.M. Keller, Parallel Program Schemata I and II, J. of the Assoc. for Computing Machinery, 20(1973), 514-37 and 696-710.
- [Knu] D.E. Knuth, "The Art of Computer Programming", Vol.1, Addison Wesley, Reading, 1968.
- [Kot] V.E. Kotov, Preobrazovaniye Operatornykh Schem v Asinchronnye Programmy, Akademiya Nauk SSSR, Novosibirsk, 1971.
- [K&M] R.M. Karp and R.E. Miller, Parallel Program Schemata, J. of Computer and System Sciences, 2(1969), 147-95.
- [Lav] S.S. Lavrov, Economy of Memory in Closed Operator Schemes, U.S.S.R. Computational Mathematics and Mathematical Physics 1 (1961), 810-28.
- [Lor] H. Lorin, "Parallelism in Hardware and Software", Prentice-Hall, Englewood Cliffs, 1972.
- [Man] Z. Manna, The Correctness of Nondeterministic Programs, Artificial Intelligence 1(1970), 1-26.

- [Mar] V.V. Martinyuk, On the Economical Distribution of a Store, U.S.S.R. Computational Mathematics and Mathematical Physics 2(1962), 469-81.
- [Pat] M.S. Paterson, Equivalence Problems in a Model of Computation, MIT AI Lab. Memo N.211, 1970.
- [P&H] M.S. Paterson and C.E. Hewitt, Comparative Schematology, MIT AI Lab. Memo No.201, 1970.
- [Rin] G. Ringel, Färbungsprobleme auf Flächen und Graphen, Berlin, 1959.
- [Ros] A.L. Rosenberg, Data Graphs and Addressing Schemes, J. of Computer and System Sciences 5(1971), 193-238.
- [Slu] D.R. Slutz, The Flow-Graph Schemata Model of Parallel Computation, Project MAC, MIT, Doc. MAC-TR-53, 1968.
- [Sto] H.S. Stone, A Pipeline Push-down Stack Computer, in: Hobbs (ed.), "Parallel Processor Systems, Technologies and Applications", Spartan Books, New York, 1970, 235-49.
- [Wil] T.C. Wilson, A Graph-Theoretical Approach to Some Problems in Compiler Code Optimization, Report CSRR 2069, University of Waterloo, 1972.
- [Yel] A.P. Yershov, Reduction of the Problem of Memory Allocation in Programming to the Problem of Coloring the Vertices of a Graph, Soviet Mathematics 3(1962), 163-65.
- [Ye2] A.P. Yershov, "The Alpha Automatic Programming System", Academic Press, New York, 1971.
- [Y&L] A.P. Yershov and A.A. Lyapunov, Formalisation of the Concept of Program, Cybernetics 3(1967), 35-49.