Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O).

MATHEMATICAL CENTRE TRACTS 82

FOUNDATIONS OF COMPUTER SCIENCE II

K.R. APT (ed.)
J.W. DE BAKKER (ed.)

PART 2

MATHEMATISCH CENTRUM AMSTERDAM 1979

AMS(MOS) subject classification scheme (1970): 68A05, 68A30, 68A10

ACM - Computing Reviews - categories: 5.24, 5.23, 5.25

ISBN 90 6196 141 6

First printing 1976
Second printing 1979

PROGRAM SEMANTICS AND MECHANIZED PROOF by R. MILNER

MATHEMATICAL CENTRE TRACTS 82, 1976, 3-44

PROGRAM SEMANTICS AND MECHANIZED PROOF

R. MILNER
University of Edinburgh, Edinburgh, U.K.

1. INTRODUCTION

In the last seven or eight years strong advances have been made in the mathematical description of the meaning of programming languages. Before this, the semantic description (in contrast to the syntactic description, which was quite formal and was often given greater weight) was presented rather informally, and inevitably contained ambiguities and left out details, and the result was that the same language acquired different meanings in different implementations.

The work of Strachey and Scott and their followers has brought about an enormous improvement. Strachey was dissatisfied both with the informality of the existing language descriptions and with their dependence on the notion of evaluation, and when Scott provided the mathematical models which he was looking for it was a rather short time before the whole of then-existing languages, such as ALGOL 60, could be mathematically defined in an elegant manner. And because the description is mathematical, it is now possible both to study concepts underlying programming languages and to conduct proofs concerning (for example) the equivalence of different constructs in one language, or of constructs in different languages.

The current literature contains quite a few examples of mathematical descriptions of languages, but it is less easy to find reports of proofs about languages. This is perhaps because proofs about real languages tend to be long and difficult both to present and to read. The aim of this paper is to remedy this deficiency to some extent. We take a very simple language, which admittedly illustrates only some of the techniques which have been developed for language description, and in the next section we study its operational semantics (semantics by abstract machine, or by evaluation).

In section 3, using that small part of Scott's work on semantic domains which is reported in my paper "Models of LCF", we present its denotational semantics in the style originated by Strachey, and show that the two semantic descriptions are indeed equivalent in an appropriate sense.

Section 4 gives an alternative semantics for the language, using the technique of continuations; it is also demonstrated that this technique can more naturally handle certain extensions to the language (in particular the introduction of error exits, and other features which allow the "normal" flow of control to be diverted). We again give the proof that - for the non-extended language - the new semantics is equivalent to the old.

Finally in section 5, using as a basis the formal deductive calculus described in the second half of "Models of LCF", we discuss the problem of mechanizing the proof of section 4. The emphasis throughout the paper is on the detail of the proofs, since we wish to convince the reader as far as possible, by leaving as few gaps as possible, that the proof strategy of section 5 will actually work. I hope that the reader will also be encouraged to believe that proofs about larger languages will indeed be amenable to similar strategies; it is an unfortunate fact that proofs about programs and languages are on the whole so long and tedious in comparison to their intellectual content that no human being is likely to have the patience to convince himself (even less, to convince others) that they are *correct* proofs.

Not many references to the literature are given in the main part of the paper; instead, I have discussed some of the relevant papers in the final section.

2. OPERATIONAL SEMANTICS

2.1. Discussion

We will consider throughout these lectures a simple programming language L which is well-understood by everyone, and indeed possesses very few features of interest. May aim is to consider *styles* of proof about languages rather than sophisticated language "features", since I believe that these styles are also appropriate to more complex languages. It will be apparent that even for such a simple language as L the detailed proofs are nor particularly

easy to read; this is not so much because of their length as because of the high ratio of technical manipulation to real mathematical content. To put it more crudely, the proofs are tedious. It has been remarked more than once that no correctness proof of a program provides greater certainty of the program's correctness than does thorough "debugging", unless the proof is mechanically checked, and this is equally true of proofs about programming languages. If we first examine the mechanizability of some proofs about a simple language, we hope to reach a position from which we can advance to proofs about more complex languages.

In this section we introduce L and describe it operationally - that is, using an abstract machine and its state transitions. The technique derives from Landin [10] and is essentially the basis of the method used to define PL/1 [11,12]. We then describe an alternative operational model, using a method learnt from Plotkin and employed by him in [13] for various λ-calculi. The purpose of this second model is as an intermediary between the abstract machine model and the denotational semantic description to be studied in following sections.

Operational and denotational semantics play complementary roles, and I believe both will continue to be necessary. An operational definition gives a guide to implementation, and as such it is likely to be in the language designer's mind more or less explicitly from the outset, since he is always aiming at a language which admits an efficient implementation. On the other hand, to describe a language by giving an abstract denotation for each phrase is at least a guard against redundancy and "adhocness"; more importantly, the language is thus defined independently of the structure of an abstract machine (which however abstract, inevitably contains some arbitrary structure), and the denotational definition is more succinct - often by a factor of three or more in length. And perhaps most importantly, the denotational definition admits *proofs* about the language, which are either impossible or very cumbersome in terms of its operational definition.

2.2. The language L

Constants : 0,1,2,...,true,false
Variables : $x_0, x_1, \ldots$
Integer operations: +,-,×,...
Boolean operations: =,>,<,...

We use c,x,b,e,p,iop,bop to range over respectively constants, variables, boolean expressions, integer expressions, programs, integer operations, boolean operations.

Expressions : e ::= x|n|e_1 iop e_2

Boolean expressions: b ::= e_1 bop e_2|true|false

Programs : p ::= null|x:=e|p_1;p_2|if b then p_1 else p_2|
while b do p_1

(We have omitted the use of parentheses; we are concerned not with parsing but with the phrase structure which results from parsing).

These constitute the three types of phrase in language L.

2.3. The S,M,C machine

A machine state is a triple <S,M,C>, where

- the value stack S $\in$ (Phrases)*
- the memory M $\in$ (Constants)$^\infty$
- the control stack C $\in$ (Phrases $\cup$ Operations $\cup$ {if,assign,while})*.

The *value stack* holds results (i.e. constants) and is also used to keep phrases whose execution is deferred.

The *memory* M = $m_0, m_1, \ldots$ holds current values of the variables $x_0, x_1, \ldots$.

The *control stack* holds phrases and operations awaiting execution.

The *Transition rules* of the machine are as follows, given by a relation $\Rightarrow$ over states (in these rules " $\cdot$ " prefixes an element to a stack, and M[n/i] means $m_0, \ldots, m_{i-1}, n, m_{i+1}, \ldots$).

I$\Rightarrow$ <S,M,c$\cdot$C> $\Rightarrow$ <c$\cdot$S,M,C>.

<S,M,$x_i\cdot$C> $\Rightarrow$ <$m_i\cdot$S,M,C> .

II$\Rightarrow$ <$n_2\cdot n_1\cdot$S,M,+$\cdot$C> $\Rightarrow$ <$n_1+n_2\cdot$S,M,C>

... etc. for all iops.

III$\Rightarrow$ <$n_2\cdot n_1\cdot$S,M,=$\cdot$C> $\Rightarrow$ <t$\cdot$S,M,C> where t is true if $n_1 = n_2$,
false otherwise

... etc. for all bops.

IV$\Rightarrow$ <S,M,null$\cdot$C> $\Rightarrow$ <S,M,C>

<S,M,(x_i:=e)$\cdot$C> $\Rightarrow$ <i$\cdot$S,M,e$\cdot$assign$\cdot$C>

<S,M,(p_1;p_2)$\cdot$C> $\Rightarrow$ <S,M,$p_1\cdot p_2\cdot$C>

$\langle S,M,(\underline{\text{if}}\ b\ \underline{\text{then}}\ p_1\ \underline{\text{else}}\ p_2)\cdot C\rangle \Rightarrow \langle p_2\cdot p_1\cdot S,M,b\cdot\underline{\text{if}}\cdot C\rangle$

$\langle S,M,(\underline{\text{while}}\ b\ \underline{\text{do}}\ p_1)\cdot C\rangle \Rightarrow \langle p_1\cdot b\cdot S,M,b\cdot\underline{\text{while}}\cdot C\rangle$.

V⇒ $\langle \underline{n}\cdot\underline{i}\cdot S,M,\underline{\text{assign}}\cdot C\rangle \Rightarrow \langle S,M[\underline{n}/i],C\rangle$

$\langle \underline{\text{true}}\cdot p_2\cdot p_1\cdot S,M,\underline{\text{if}}\cdot C\rangle \Rightarrow \langle S,M,p_1\cdot C\rangle$

$\langle \underline{\text{false}}\cdot\ldots\ldots\ldots\ldots\rangle \quad \langle\ldots\ldots p_2\ldots\rangle$

$\langle \underline{\text{true}}\cdot p_1\cdot b\cdot S,M,\underline{\text{while}}\cdot C\rangle \Rightarrow \langle S,M,(p_1;\underline{\text{while}}\ b\ \underline{\text{do}}\ p_1)\cdot C\rangle$

$\langle \underline{\text{false}}\cdot\ldots\ldots\ldots\ldots\ldots\rangle \Rightarrow \langle S,M,\underline{\text{null}}\cdot C\rangle$.

Note that ⇒ is deterministic. We use $\overset{n}{\Rightarrow}$ to denote the n-th power of the relation ⇒, and $\overset{*}{\Rightarrow}$ for its transitive reflexive closure.

We define EVAL: Programs × Memories → Memories by:

$$EVAL(p,M) = M' \quad \text{iff} \quad \langle \varepsilon,M,p\cdot\varepsilon\rangle \overset{*}{\Rightarrow} \langle \varepsilon,M',\varepsilon\rangle\ ,$$

where ε stands for an empty stack. Notice that EVAL is a partial function.

2.4. The reduction relation →

The S,M,C machine is abstract, but rather arbitrary in the method chosen to control evaluation - that is, one might have used other structures in preference to a pair of stacks and a memory vector. Some of this arbitrariness may be removed by axiomatizing a reduction relation → over Phrases × Memories. We give the axioms and rules of a simple formal deductive system.

I→ $x_i,M \to \underline{m}_i,M$.

II→ $(e_1\ \text{op}\ e_2),M \to (e_1'\ \text{op}\ e_2),M$ if $e_1,M \to e_1',M$

$(\underline{n}\ \text{op}\ e),M \to (\underline{n}\ \text{op}\ e'),M$ if $e,M \to e',M$

$(\underline{n}_1+\underline{n}_2),M \to \underline{n_1+n_2},M$

... etc, for all iops.

$(\underline{n}_1=\underline{n}_2),M \to \underline{\text{true}},M$
$\to \underline{\text{false}},M$ } according as $n_1 = n_2$ or not,

... etc, for all bops.

III→ $(x_i:=e),M \to \underline{\text{null}},M[\underline{n}/i]$ if $e,M \overset{*}{\to} \underline{n},M$.

IV→ $(p_1;p_2),M \to (p_1';p_2),M'$ if $p_1,M \to p',M'$

$(\underline{\text{null}};p),M \to p,M$.

V→ $(\underline{\text{if}}\ b\ \underline{\text{then}}\ p_1\ \underline{\text{else}}\ p_2),M \to p_1,M$ if $b,M \overset{*}{\to} \underline{\text{true}},M$

..... $\to p_2,M$ if $b,M \overset{*}{\to} \underline{\text{false}},M$.

VI→ $(\underline{\text{while}}\ b\ \underline{\text{do}}\ p_1),M \to (p_1;\underline{\text{while}}\ b\ \underline{\text{do}}\ p_1),M$ if $b,M \overset{*}{\to} \underline{\text{true}},M$

..... $\to \underline{\text{null}},M$ $\underline{\text{if}}\ b,M \overset{*}{\to} \underline{\text{false}},M$

REMARK. To be fully formal, we should be clear that the sentences of this formal system are of the form

$$\phi,M \to \phi',M' ,$$

where ϕ,ϕ' are program phrases; $\phi,M \overset{*}{\to} \phi',M'$ is not a sentence. Thus rule III→ has not just a single hypothesis $e,M \overset{*}{\to} \underline{n},M$, but k hypotheses (for some $k \geq 0$) of the form

$$e_i,M \to e_{i+1},M \qquad (0 \leq i < k),$$

where $e_i = e$, $e_k = \underline{n}$. The same remark holds for V→ and VI→.

We sketch the proof that → is deterministic. First, one shows that for each e,M there is at most one pair e',M' such that

$$e,M \to e',M'$$

and that M' = M. It follows that in the case of $\overset{*}{\to}$, for each e,M there is at most one pair $\underline{n},M'$ such that

$$e,M \overset{*}{\to} \underline{n},M'$$

and that M' = M. Similar results hold for boolean expressions. These proofs proceed by induction on the structure of expressions. Analogously, one then shows that if $p,M \to p',M'$ then p',M' is unique.

It follows that the partial function

$$\text{eval: Programs} \times \text{Memories} \to \text{Memories}$$

is well-defined as follows:

$$\mathrm{eval}(p,M) = M' \text{ iff } p,M \overset{*}{\to} \underline{\mathrm{null}},M' .$$

2.5. EVAL = eval

One might hope to prove the equivalence of EVAL and eval by induction on the structure of programs. This fails just because of the while construct, and we have to resort to induction on the length of computation.

LEMMA 1.

(i) *If* $e,M \overset{k}{\to} \underline{n},M'$ *then* $M' = M$ *and* $\langle S,M,e\cdot C\rangle \overset{*}{\Rightarrow} \langle \underline{n}\cdot S,M',C\rangle$.

(ii) *If* $b,M \overset{k}{\to} \left\{\begin{matrix}\underline{\mathrm{true}}\\ \underline{\mathrm{false}}\end{matrix}\right\},M'$ *then* $M' = M$ *and* $\langle S,M,b\cdot C\rangle \overset{*}{\Rightarrow} \langle\left\{\begin{matrix}\underline{\mathrm{true}}\\ \underline{\mathrm{false}}\end{matrix}\right\}\cdot S,M'.C\rangle$.

(iii) *If* $p,M \overset{k}{\to} \underline{\mathrm{null}},M'$, *then* $\langle S,M,p\cdot C\rangle \overset{*}{\Rightarrow} \langle S,M',C\rangle$.

Proof. We shall omit the proofs of (i) and (ii) and prove (iii) by induction on k (parts (i) and (ii) are simpler).

Basis k = 0. In this case p = null, M = M' and use IV(⇒).

Step. Assume (iii) for all k' < k, and assume

(1) $\quad p,M \overset{k}{\to} \mathrm{null},M' .$

Argue by cases

(a) p is null. Impossible, since $\underline{\mathrm{null}},M \not\to$.

(b) p is $(x_i:=e)$. Then k = 1 and (1) must have been inferred by III (→), so $e,M \overset{*}{\to} \underline{n},M$ and $M' = M[\underline{n}/i]$. So we have

$$\begin{aligned}&\langle S,M,(x_i:=e)\cdot C\rangle \Rightarrow \langle \underline{i}\cdot S,M,e\cdot\underline{\mathrm{assign}}\cdot C\rangle\\ &\overset{*}{\Rightarrow} \langle \underline{n}\cdot\underline{i}\cdot S,M,\underline{\mathrm{assign}}\cdot C\rangle \quad \text{by Lemma 1(i)}\\ &\Rightarrow \langle S,M[\underline{n}/i]\rangle \quad \text{by V (⇒)},\end{aligned}$$

as required.

(c),(d) We leave the cases p is $(p_1;p_2)$ or p is if b then p_1 else p_2 as an exercise for the reader.

(e) p is (while b do p_1). Then we have

$$p,M \to \left\{\begin{matrix}p_1;p\\ \underline{\mathrm{null}}\end{matrix}\right\},M \overset{k-1}{\to} \underline{\mathrm{null}},M', \text{ with resp. } b,M \overset{*}{\to} \left\{\begin{matrix}\underline{\mathrm{true}}\\ \underline{\mathrm{false}}\end{matrix}\right\},M$$

and Lemma 1(ii) then allows us to deduce

$$\begin{aligned}
\langle S,M,p\cdot C\rangle &\Rightarrow \langle p_1\cdot b\cdot S,M,b\cdot \underline{\text{while}}\cdot C\rangle && \text{by IV } (\Rightarrow)\\
&\Rightarrow \left\{\begin{matrix}\underline{\text{true}}\\ \underline{\text{false}}\end{matrix}\right\}\cdot p_1\cdot b\cdot S,M,\underline{\text{while}}\cdot C\rangle && \text{by Lemma 1(ii)}\\
&\Rightarrow \langle S,M,\left\{\begin{matrix}p_1;p\\ \underline{\text{null}}\end{matrix}\right\}\cdot C\rangle && \text{by V } (\Rightarrow)\\
&\overset{*}{\Rightarrow} \langle S,M',C\rangle && \text{by the ind.hypoth. at } k-1.
\end{aligned}$$

This concludes the proof of Lemma 1. □

Lemma 1 is half of our equivalence theorem. For the other half we introduce a definition.

DEFINITION. The reduction $\langle S,M,t\cdot C\rangle \overset{k}{\rightarrow} \langle S',M',C\rangle$ is *perfect* if the control stack in each intermediate state is a proper extension of C; that is, C is first "uncovered" at the last step.

LEMMA 2.

(i) *If* $\langle S,M,e\cdot C\rangle \overset{k}{\Rightarrow} \langle S',M',C\rangle$ *is perfect, then* $S' = \underline{n}\cdot S$ *for some* $\underline{n}$, $M' = M$ *and* $e,M \overset{*}{\rightarrow} \underline{n},M$.

(ii) *If* $\langle S,M,b\cdot C\rangle \overset{k}{\Rightarrow} \langle S',M',C\rangle$ *is perfect, then* $S' = \underline{\text{true}}\cdot S$ *or* $S' = \underline{\text{false}}\cdot S$, $M' = M$ *and* $b,M \overset{*}{\rightarrow} \left\{\begin{matrix}\underline{\text{true}}\\ \underline{\text{false}}\end{matrix}\right\},M$ *resp.*

(iii) *If* $\langle S,M,p\cdot C\rangle \overset{k}{\Rightarrow} \langle S',M',C\rangle$ *is perfect, then* $S' = S$ *and* $p,M \overset{*}{\Rightarrow} \underline{\text{null}},M'$.

Proof. Again, we omit the proofs of (i) and (ii), and deal with representative parts of (iii), for which we induce on k.

Basis k = 0. Impossible.

Step. Assume (iii) for all $k' < k$, and that

$$(2) \qquad \langle S,M,p\cdot C\rangle \overset{k}{\Rightarrow} \langle S',M',C\rangle$$

is perfect.

Argue by cases:

(a) p is $\underline{\text{null}}$. Then k must be 1 since the reduction (2) is perfect, and by IV $(\Rightarrow)$ we see that M' = M, S' = S. The rest is trivial.

(b) p is $(x_i := e)$. Then from (2)

$$\left.\begin{array}{l} \langle S,M,p\cdot C\rangle \Rightarrow \langle \underline{i}\cdot S,M,e\cdot \underline{assign}\cdot C\rangle \\ \qquad\qquad \overset{k-2}{\Rightarrow} \langle S'',M'',\underline{assign}\cdot C\rangle \end{array}\right] \text{ perfect}$$

$$\Rightarrow \langle S',M',C\rangle \qquad \text{by V } (\Rightarrow),$$

where by Lemma 2(i), S" = $\underline{n}\cdot S$, M" = M and $e,M \overset{*}{\rightarrow} \underline{n},M$. Hence by V ($\Rightarrow$) S" = S, M' = $[\underline{n}/i]M$; so $p,M \rightarrow \underline{null},M'$ by III ($\rightarrow$).

(c),(d) p is $(p_1;p_2)$ or $(\underline{if}\ b\ \underline{then}\ p_1\ \underline{else}\ p_2)$. Exercise.

(e) p is $(\underline{while}\ b\ \underline{do}\ p_1)$. Then from (2)

$$\left.\begin{array}{l} \langle S,M,p\cdot C\rangle \Rightarrow \langle p\cdot b\cdot S,M,b\cdot \underline{while}\cdot C\rangle \\ \qquad\qquad \overset{k_1}{\Rightarrow} \langle S'',M'',\underline{while}\cdot C\rangle \end{array}\right] \text{ perfect}$$

(3) $$= \langle \left\{\begin{matrix}\underline{true}\\ \underline{false}\end{matrix}\right\}\cdot p\cdot b\cdot S,M,\underline{while}\cdot C\rangle \text{ where } b,M \overset{*}{\rightarrow} \left\{\begin{matrix}\underline{true}\\ \underline{false}\end{matrix}\right\},M \text{ resp.}$$

$$\Rightarrow \langle S,M,\left\{\begin{matrix}(p_1;p)\\ \underline{null}\end{matrix}\right\}\cdot C\rangle \text{ by V } (\Rightarrow)$$

$$\overset{k_2}{\Rightarrow} \langle S',M',C\rangle \qquad \text{perfectly, by assumption.}$$

Now $k_2 < k$, so using Lemma 2(iii) at k_2, for the last reduction, we infer that resp.

(4) $$\left\{\begin{matrix}(p_1;p)\\ \underline{null}\end{matrix}\right\},M \overset{*}{\rightarrow} \underline{null},M'.$$

But from (3), by VI ($\rightarrow$) we obtain respectively

$$p,M \rightarrow \left\{\begin{matrix}p_1;p\\ \underline{null}\end{matrix}\right\},M,$$

whence $p,M \overset{*}{\rightarrow} \underline{null},M'$ by (4).

This completes the proof of Lemma 2. □

THEOREM. EVAL = eval.

Proof. We need to establish that

$$\langle \varepsilon, M, p \cdot \varepsilon \rangle \overset{*}{\Rightarrow} \langle \varepsilon, M', \varepsilon \rangle \quad \text{iff} \quad p', M \overset{*}{\rightarrow} \underline{\text{null}}, M'.$$

($\Rightarrow$) Suppose $\langle \varepsilon, M, p \cdot \varepsilon \rangle \overset{*}{\Rightarrow} \langle \varepsilon, M', \varepsilon \rangle$. Thus must be a perfect reduction, since there is no production of the form

$$\langle S, M, \varepsilon \rangle \Rightarrow \ldots .$$

So Lemma 2 provides the rest.
($\Leftarrow$) This is a simple application of Lemma 1. □

3. DENOTATIONAL SEMANTICS

3.1. Semantic domains

In this section we give a semantic description of L in terms of cpo's, using the definitions and results of the first two sections of "Models of LCF" [28]. But first we need two further cpo-preserving domain operations (we have already seen that if D and E are cpos, so is the continuous function domain $[D \to E]$, which we shall abbreviate henceforward to just $D \to E$).

Cartesian product. If we define the ordering $\sqsubseteq$ over

$$D \times E = \{\langle x, y \rangle \mid x \in D,\ y \in E\}$$

by

$$\langle x, y \rangle \sqsubseteq \langle x', y' \rangle \text{ iff } x \sqsubseteq x' \text{ and } y \sqsubseteq y',$$

the following are easily verified:

(i) $D \times E$ is a cpo; indeed, $\perp_{D \times E} = \langle \perp_D, \perp_E \rangle$ and for the chain $\{\langle x_i, y_i \rangle \mid i \geq 0\}$ in $D \times E$, $\bigsqcup_i \langle x_i, y_i \rangle = \langle \bigsqcup_i x_i, \bigsqcup_i y_i \rangle$.

(ii) The pairing function $\lambda x \cdot \lambda y \cdot \langle x, y \rangle \in D \to E \to D \times E$ is continuous, and so are the selector functions

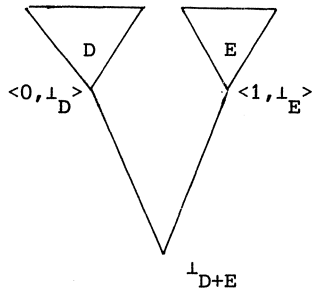
$$\text{fst} = \lambda\langle x,y\rangle \cdot x \in D \times E \to D$$
$$\text{snd} = \lambda\langle x,y\rangle \cdot y \in D \times E \to E.$$

Disjoint sum. Let us define

$$D + E = \{\bot\} \cup \{\langle 0,x\rangle \mid x \in D\} \cup \{\langle 1,x\rangle \mid x \in E\}$$

and the ordering $\sqsubseteq$ over $D + E$ by $z \sqsubseteq z'$ iff either $z = \bot$, or $z = \langle 0,x\rangle$ and $z' = \langle 0,x'\rangle$ and $x \sqsubseteq x'$ in D, or $z = \langle 1,y\rangle$ and $z' = \langle 1,y'\rangle$ and $y \sqsubseteq y'$ in E.

A diagram makes it clear:



The flags 0 and 1 are merely for disjoining D from E, so that for example $D + D$ contains two distinct copies of D.

Associated with + are the following functions:

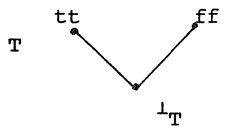
(i) The *injection* functions $\iota_0 : D \to D + E = \lambda x{:}D \cdot \langle 0,x\rangle$
$\iota_1 : D \to D + E = \lambda y{:}E \cdot \langle 1,y\rangle$.

(ii) The *projection* functions $\pi_0 : D + E \to D$
$\pi_1 : D + E \to E$,

where $\pi_0 z = \begin{cases} x \text{ if } z = \langle 0,x\rangle \\ \bot_D \text{ if } z = \langle 1,y\rangle \text{ or } \bot_{D+E} \end{cases}$,

and $\pi_1 z$ is similar.

(iii) *The discriminator functions* $\left.\begin{matrix}\delta_0: \\ \delta_1:\end{matrix}\right\} D + E \to T$



where $\delta_0 z = \begin{cases} tt \text{ if } z = \langle 0,x\rangle \\ ff \text{ " " } = \langle 1,y\rangle \\ \bot_T \text{ " " } = \bot_{D+E} \end{cases}$,

and $\delta_1 z$ is similar with tt, ff interchanged.

It is a simple exercise to show that these are all continuous functions, and that $\forall x \in D$

(a) $\pi_0(\iota_0 x) = x$,

(b) $\pi_1(\iota_0 x) = \perp_E$,

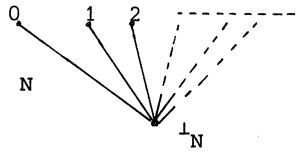
(c) $\delta_0(\iota_0 x) = tt$,

and many similar identities. Of course the binary + can be generalised to n-ary and even infinitary +. If for the latter we choose the non-negative integers 0,1,... as flags, we may also define

$$D^* = \sum_{i=0}^{\infty} D^i = \{\cdot\} + D + (D \times D) + (D \times D \times D) + \ldots$$

(where $\{\cdot\}$ is the one element domain), which is a domain of finite sequences of elements of D. Then $\iota_0(\cdot)$ is just the null sequence, and

$$\text{length: } D^* \to N$$



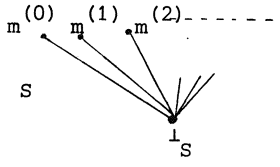
is just the "flag selecting" function. It is easy enough to define all the normal list processing operations - head, tail, cons, null - in terms of pairing and selecting and the ι_i, δ_i, π_i.

3.2. Denotation of language L

Our aim is to acribe directly (rather than via a machine) a function: $S \to S$ to each program in L, as its meaning or denotation, where S is now a cpo of (abstract) memories or stores. We choose S to respect convention: it is accidental that S stood also for a stack in our operational semantics, but we shall henceforward use it only in the new sense.

Though there are alternatives, we choose S to be a flat cpo of vectors of non-negative integers. More precisely

$$S = \{\perp_S\} \cup \{\langle m_i \rangle \mid \langle \underline{m}_i \rangle \in \text{Memories}\}$$



and we shall allow s to vary over S, but will adopt the convention that m varies over $S - \{\perp_S\}$ - i.e. it stands always for a *defined* store; also we will use m for the abstract counterpart of M:

$$M = \underline{m}^\bullet, \underline{m}_1, \ldots \iff m = m_0, m_1, \ldots \quad .$$

The only operations we need on stores are

$$\text{update: } N \to (N \times S \to S)$$
$$\text{select: } N \to (S \to N) \ ,$$

where

$$\text{update } i(n,s) = \begin{cases} \perp_S \text{ if } i,n \text{ or } s \text{ is undefined} \\ [n/i]s \qquad \text{otherwise} \end{cases} ,$$

$$\text{select } i\ s \quad = \begin{cases} \perp_N \text{ if } i \text{ or } s \text{ is undefined} \\ s_i \qquad \text{otherwise.} \end{cases}$$

These functions are easily shown continuous, and we can also shown

LEMMA 1. *If* i,j,n_1,n_2 *are all defined then*

(1) $$\text{select } i \text{ (update } j(n,m)) \quad = \begin{cases} n, \text{ if } i = j \\ \text{select } i\ m \text{ if } i \neq j \ , \end{cases}$$

(2) $$\text{update } i \text{ (select } i\ m, m) \quad = \quad m \ ,$$

(3) $$\text{update } i \ (n_1, \text{update } j(n_2, m)) = \begin{cases} \text{update } i\ (n_1, m) \text{ if } i = j \\ \text{update } j\ (n_2, \text{update } i\ (n_1, m)) \text{ otherwise.} \end{cases}$$

Proof. Omitted. □

REMARK. Part (3) holds even for undefined i,j,n_1,n_2 and for $\perp_S$ in place of m.

3.3. Semantics of expressions

We first assign to each expression e a function $\in S \to N$ as its meaning, and to each boolean expression b a function $\in S \to T$. The meaning $E[\![e]\!]$ of e

is given as follows:

$$\begin{aligned}
&\mathcal{E}[\![e]\!]\perp_S &&= \perp_N\\
&\mathcal{E}[\![\underline{n}]\!]m &&= n\\
&\mathcal{E}[\![x_i]\!]m &&= \text{select } i\ m\ (= m_i)\\
&\mathcal{E}[\![e_1+e_2]\!]m &&= \mathcal{E}[\![e_1]\!]m + \mathcal{E}[\![e_2]\!]m\\
&\ldots \text{ etc. for all iops.}
\end{aligned}$$

[Note that the right-hand + stands for a function $\in N \to (N \to N)$, while the left-hand + is a symbol of L.]

The brackets $[\![\ ]\!]$ are to distinguish syntactic objects. For later work we shall need to consider a cpo E, which consists of all Expressions together with certain "partially-defined" expressions; then it will be possible to discuss $\mathcal{E}$ as a member of the domain $E \to (S \to N)$. Such a domain as E would indeed be important if we were discussing expressions as data objects (on a par with N) - as they would be for a compiler for example. But here we need do no more than remark that $\mathcal{E}[\![e]\!]$ is defined inductively on the structure of e. The same remark applies to our later semantic functions $\mathcal{B}$ and $\mathcal{P}$.

For boolean expressions, we define $\mathcal{B}[\![b]\!] \in S \to T$ thus:

$$\begin{aligned}
&\mathcal{B}[\![b]\!]\perp_S &&= \perp_S\\
&\mathcal{B}[\![\underline{\text{true}}]\!]m &&= \text{tt}, \quad \mathcal{B}[\![\underline{\text{false}}]\!]m = \text{ff}\\
&\mathcal{B}[\![e_1=e_2]\!]m &&= \begin{cases} \perp_T & \text{if either } \mathcal{E}[\![e_1]\!]m \text{ or } \mathcal{E}[\![e_2]\!]m \text{ is } \perp_N\\ \text{tt} & \text{if they are equal}\\ \text{ff} & \text{otherwise.}\end{cases}
\end{aligned}$$

Before dealing with programs, we state without proof a simple lemma which relates the denotational semantics of expressions to their operational semantics.

LEMMA 2.

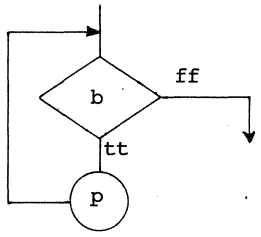
(1) $e,M \overset{*}{\to} \underline{n},M$ iff $\mathcal{E}[\![e]\!]m = n \neq \perp_N$

(2) $b,M \overset{*}{\to} \left\{\begin{matrix}\underline{\text{true}}\\ \underline{\text{false}}\end{matrix}\right\},M$ iff $\mathcal{B}[\![b]\!]m = \left\{\begin{matrix}\text{tt}\\ \text{ff}\end{matrix}\right\}$.

Proof. By induction on the structure of expressions. Recall our convention that m is the abstract counterpart of M. □

3.4. Semantics of programs

We proceed as with expressions to define $\mathcal{P}[\![p]\!] \in S \to S$ inductively on the structure of programs. But first the while construct deserves special attention. What function $f \in S \to S$ is $\mathcal{P}[\![\underline{\text{while}}\ b\ \underline{\text{do}}\ p_1]\!]$? The diagram



suggests that f satisfies

$$fs = \mathcal{B}[\![b]\!]s \to f(\mathcal{P}[\![p_1]\!]s), s$$

so that we choose for f the least fixed point fix Φ of the functional

$$\Phi = \lambda f \cdot \lambda s \cdot (\mathcal{B}[\![b]\!]s \to f(\mathcal{P}[\![p_1]\!]s), s) \in [[S \to S] \to [S \to S]].$$

That the *least* fixed point of Φ is right will be justified by our theorem.

In defining $\mathcal{P}[\![e]\!]$ we choose not to *define* $\mathcal{P}[\![e]\!]\perp_S = \perp$ since it *follows* from our definition as an easy lemma.

$$\begin{aligned}
&\mathcal{P}[\![\underline{\text{null}}]\!]s && = s\\
&\mathcal{P}[\![x_i := e]\!]s && = \text{update } i\ (\mathcal{E}[\![e]\!]s, s)\\
&\mathcal{P}[\![p_1 ; p_2]\!]s && = \mathcal{P}[\![p_2]\!](\mathcal{P}[\![p_1]\!]s)\\
&\mathcal{P}[\![\underline{\text{if}}\ b\ \underline{\text{then}}\ p_1\ \underline{\text{else}}\ p_2]\!]s && = \mathcal{B}[\![b]\!]s \to \mathcal{P}[\![p_1]\!]s, \mathcal{P}[\![p_2]\!]s\\
&\mathcal{P}[\![\underline{\text{while}}\ b\ \underline{\text{do}}\ p_1]\!] && = \text{fix}(\lambda f \cdot \lambda s' \cdot \mathcal{B}[\![b]\!]s' \to f(\mathcal{P}[\![p]\!]s'), s').
\end{aligned}$$

LEMMA 3. $\mathcal{P}[\![p]\!]\perp_S = \perp_{S'}$.

Proof. By induction on the structure of p. Use the definition of update, and of $\mathcal{B}[\![b]\!]$, and also that fix $\Phi s = \Phi(\text{fix } \Phi)s$. □

3.5. Equivalence of operational and denotational semantics for L

We now prove the theorem:

THEOREM. $\mathcal{P}[\![p]\!]m = m'$ *iff* $p,M \overset{*}{\to} \underline{null},M'$.

As an easy corollary of this, we have that $\mathcal{P}[\![p]\!]m = \perp_S$ iff the reduction of p,M under $\to$ fails to terminate. It is easier to divide the theorem into two lemmas.

LEMMA 4. *If* $p,M \to p',M'$ *then* $\mathcal{P}[\![p]\!]m = \mathcal{P}[\![p']\!]m'$.

Proof. Induction on p.

Basis. (i) p is null. Nothing to prove, since $null,M \not\to$.

(ii) p is $(x_i:=e)$. Then by the rules of $\to$, p' is null, and $M' = [\underline{n}/i]M$, where $e,M \overset{*}{\to} \underline{n},M$. So Lemma 2 gives $\mathcal{E}[\![e]\!]m = n$, whence $\mathcal{P}[\![p]\!]m = \text{update } i\ (n,m) = m[n/i] = m'$, while $\mathcal{P}[\![p']\!]m' = \mathcal{P}[\![\underline{null}]\!]m' = m'$.

Step. (iii) p is $(p_1;p_2)$.

If p_1 *is* null, then $p',M' = p_2,M$ by rule IV ($\to$). But then $\mathcal{P}[\![p]\!]m = \mathcal{P}[\![p_2]\!](\mathcal{P}[\![\underline{null}]\!]m) = \mathcal{P}[\![p_2]\!]m = \mathcal{P}[\![p']\!]m'$.

Otherwise $p' = (p_1';p_2)$ by IV ($\to$), where $p_1,M \to p_1',M'$, so by Induction Hypothesis $\mathcal{P}[\![p_1]\!]m = \mathcal{P}[\![p_1']\!]m'$. But then $\mathcal{P}[\![p]\!]m = \mathcal{P}[\![p_2]\!](\mathcal{P}[\![p_1]\!]m) = \mathcal{P}[\![p_2]\!](\mathcal{P}[\![p_1']\!]m') = \mathcal{P}[\![p']\!]m'$.

(iv) p is (if b then p_1 else p_2). Exercise.

(v) p is (while b do p_1). Then

$$p',M' = \begin{Bmatrix} p_1;p \\ \underline{null} \end{Bmatrix},M \text{ where resp. } b,M \overset{*}{\to} \begin{Bmatrix} \underline{true} \\ \underline{false} \end{Bmatrix},M.$$

So Lemma 2 gives resp. $\mathcal{B}[\![b]\!]m = \begin{Bmatrix} tt \\ ff \end{Bmatrix}$, whence

$$\begin{aligned} \mathcal{P}[\![p]\!]m &= (\text{fix } \Phi)m = \Phi(\text{fix } \Phi)m \\ &= \mathcal{B}[\![b]\!]m \to (\text{fix } \Phi)(\mathcal{P}[\![p_1]\!]m),m \\ &= \text{resp.} \left\{ \begin{matrix} \text{fix } \Phi(\mathcal{P}[\![p_1]\!]m) = \mathcal{P}[\![p]\!](\mathcal{P}[\![p_1]\!]m) \\ m = \mathcal{P}[\![\underline{null}]\!]m \end{matrix} \right\} \\ &= \text{in each case } \mathcal{P}[\![p']\!]m'. \quad \square \end{aligned}$$

Since this lemma was only concerned with one step reductions, we needed no induction to handle the while construct.

LEMMA 5. *If* $\mathcal{P}[\![p]\!]m = m'$ *then* $p,M \xrightarrow{*} \underline{null},M'$.

Proof. Again, by induction on p. We leave all the cases to the reader as an exercise, except for the while construct. Here, we assume the lemma for p_1 and assume $\mathcal{P}[\![p]\!]m = m'$, where p is (while b do p_1). Now $\mathcal{P}[\![p]\!] = \text{fix } \Phi$ (Φ as before). Let

$$\left.\begin{aligned} f_0 &= \perp_{s\to s} \\ f_{i+1} &= \Phi\, f_i \end{aligned}\right\} \text{ so that } \mathcal{P}[\![p]\!] = \bigsqcup_i f_i \ .$$

We shall prove by induction on i (i.e. induction on the *iterates* of Φ) that

(#) $\qquad$ if $f_i m = m'$ then $p,M \xrightarrow{*} null,M'$.

Further, from $\mathcal{P}[\![p]\!]m = \bigsqcup_i (f_i m) = m'$ we can infer that for *some* i, $f_i m = m'$; the rest follows from (#).

[Note: this inference is a consequence of the fact that S is a *flat* cpo; $\langle f_i m\rangle$ is a chain in S with lub m', and hence some member of the chain is itself equal to m'.]

Proof of (#). For the basis i = 0 there is nothing to prove, since $f_0 m = \perp m = \perp$, while m' is by convention *defined*.

Step. Assume # for i, and assume $f_{i+1} m = m'$. Now

$$f_{i+1} m = \Phi f_i m = (\mathcal{B}[\![b]\!]m \to f_i(\mathcal{P}[\![p_1]\!]m), m) = m'.$$

But since m' is defined

either a) $\mathcal{B}[\![b]\!]m = tt$ (whence $b,M \xrightarrow{*} \underline{true},M$ by Lemma 2) and
$\quad f_{i+1} m = f_i(\mathcal{P}[\![p_1]\!]m) = m'$,

or b) $\mathcal{B}[\![b]\!]m = ff$ (whence $b,M \xrightarrow{*} \underline{false},M$) and
$\quad f_{i+1} m = m = m'$.

In the case of b), $p,M \xrightarrow{*} \underline{null},M'$ follows easily. For case a), we first

note that $P[\![p_1]\!]m$ must be defined, = m" say. (If not, then we have $m' = f_i(\perp) \sqsubseteq P[\![p]\!]\perp = \perp$, a contradiction.)

So by the inductive assumption of Lemma 5 for p_1, we have that $p_1,M \overset{*}{\to} \underline{null},M"$; and by the present inductive assumption for f_i we have that $p,M" \overset{*}{\to} \underline{null},M'$. It follows that

$$\begin{aligned} p,M &\to (p_1;p),M \quad \text{(since } b,M \overset{*}{\to} \underline{true},M\text{)} \\ &\overset{*}{\to} (\underline{null},p),M" \\ &\to p,M" \\ &\overset{*}{\to} \underline{null},M' \end{aligned}$$

as required.

Proof of Theorem. (⇒) Directly from Lemma 5.
(⇐) Let $p,M = p^{(0)},M^{(0)} \to \ldots \to p^{(n)},M^{(n)} = \underline{null},M'$. Then Lemma 4 tells us that $\{P[\![p^{(i)}]\!]m^{(i)}\}$ are all equal, whence $P[\![p]\!]m = P[\![\underline{null}]\!]m' = m'$ as required. □

4. CONTINUATION SEMANTICS

4.1. Continuations

Hitherto we have tacitly assumed of the language L that the execution of each program construct, if it terminates at all, terminates "naturally" - i.e. control passes out at the end of the construct. Thus it was safe to write

$$P[\![p_1;p_s]\!]s = P[\![p_2]\!](P[\![p_1]\!]s)$$

indicating that p_2 will always be executed after the termination of p_1.

Various language constructs do not admit this assumption; jumps are the obvious example, and error exits (trapped or not) are another. We shall consider the latter only, and show how the device of *continuations* - introduced independently by L. Morris and C. Wadsworth - can handle abnormal exits. Extension of the technique to jumps involves no further concepts - it is just rather tedious, essentially because jumps spoil the structured

nature of programs.

To avoid too much detail, let us from now on forget the details of assignments (or other basic non-compound instructions) and expressions in L, merely assuming that there are certain Boolean expressions $b_1, b_2, \ldots$ with corresponding semantics $\beta_i = \mathcal{B}[\![b_i]\!] \in S \to T$, and certain basic instructions c_i - including assignments (but excluding null) with corresponding semantics $\gamma_i = \mathcal{C}[\![c_i]\!] \in S \to S$, where in particular $\mathcal{C}[\![x_i := e]\!]s = \text{update } i\ (\mathcal{E}[\![e]\!]s, s)$. (We are thus preventing consideration of abnormal exit from expressions or basic instructions; the technique of continuations adapts easily to allow this.)

Consider now adding a single extra instruction "error", whose effect is supposed to be to abort the whole program and deliver an error message (which may depend on the current store). There is no easy way to fit error into the semantic equations for $\mathcal{P}$.

So we proceed as follows. First, assume a cpo O containing all possible end results of programs, including error messages. We should need something like O anyway - hitherto we have taken the meaning of a program in the domain $S \to S$, but we are not often interested in the state of the whole store at the end of the whole program. With O we may imagine extracting the final output by applying to $\mathcal{P}[\![p]\!]s$ some output function $\in S \to O$.

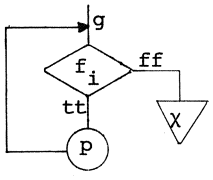
(Aside: we are still not considering programs which can output - or input - information during execution; it turns out that continuations also make this easy to handle.)

But now it appears that if χ is such an output function, then $\chi \circ \mathcal{P}[\![p]\!]$ is a function in the same domain $S \to O$, respecting the work done by executing p and then "outputting". If we call $C = S \to O$ the domain of *continuations*, we can equally well specify the meaning of a program p by defining the effect of "prefixing" its work to our arbitrary continuation χ to yield another continuation. So our new semantic function will be $\mathcal{Q}$, and we define $\mathcal{Q}[\![p]\!] \in C \to C$ below.

To emphasize the back-to-front way of working, consider the while construct again. If p is while b_i do p_1, then we want to specify the *output* $\mathcal{Q}[\![p]\!]\chi s$ which results from starting p with s and continuing after p with χ.

Let $\mathcal{Q}[\![p]\!] = g$. Then $g\chi$ satisfies

$$g\chi s = \beta_i s \to \mathcal{Q}[\![p_1]\!](g\chi)s,\ \chi s$$



and our equation below settles on the least fixed point of this recursive equation for g.

$$\begin{aligned}
&\mathcal{Q}[\![\underline{\text{null}}]\!]\chi s &&= \chi s\\
&\mathcal{Q}[\![c_i]\!]\chi s &&= \chi(\gamma_i s)\\
&\mathcal{Q}[\![p_1;p_2]\!]\chi s &&= \mathcal{Q}[\![p_1]\!](\mathcal{Q}[\![p_2]\!]\chi)s\\
&\mathcal{Q}[\![\underline{\text{if}}\ b_i\ \underline{\text{then}}\ p_1\ \underline{\text{else}}\ p_2]\!]\chi s &&= \beta_i s \rightarrow \mathcal{Q}[\![p_1]\!]\chi s, \mathcal{Q}[\![p_2]\!]\chi s\\
&\mathcal{Q}[\![\underline{\text{while}}\ b_i\ \underline{\text{do}}\ p_1]\!] &&= \text{fix}(\lambda g\cdot\lambda\chi\cdot\lambda s\cdot\beta_i s \rightarrow \mathcal{Q}[\![p_1]\!](f\chi)s, \chi s),
\end{aligned}$$

and finally we add

$$\mathcal{Q}[\![\underline{\text{error}}]\!]\chi s = \text{dump } s\ ,$$

where dump is a special error continuation - we may imagine that it prints the whole store if we like. The vital point is that because error chooses to ignore the normal continuation χ, any program containing error may also choose to ignore its normal continuation. (Write out the meaning of the program "error;p" to emphasize this, and also notice the difference between $\mathcal{P}[\![p_1;p_2]\!]$ and $\mathcal{Q}[\![p_1;p_2]\!]$.)

In these lectures we do not propose to develop the semantics of more complex languages (they can be found in the literature) but rather to look at proofs about simple languages; in fact we shall only do *one* proof, since we also want to examine the possibility of mechanizing it. But before leaving errors, it is worth while sketching an extension to the language and its semantics to allow for trapping errors.

Suppose then that error is to invoke not a fixed continuation "dump", but an error continuation η which has somehow been established by the program. That is, to give the meaning of error, our new semantic function $\mathcal{R}$ needs an η as well as a χ as argument - i.e. $\mathcal{R}[\![p]\!] \in C \rightarrow C \rightarrow C$, and we write

$$\mathcal{R}[\![\underline{\text{error}}]\!]\chi\eta s = \eta s \qquad \text{(cannot succeed!)}$$

and naturally

$$\mathcal{R}[\![\underline{\text{null}}]\!]\chi\eta s = \chi s \qquad \text{(cannot fail!)} \quad .$$

But how are errors to be trapped - or (to ask the same question differently) how are error continuations established? The simplest possible answer is to add the program construct

$$p_1 \ \underline{\text{orelse}} \ p_2 \ ,$$

whose effect is to be as p_1 if p_1 terminates normally, otherwise to execute p_2 as soon as p_1 commits an error. (Thus, the whole construct can only err if p_2 errs.)

Exercise. Give the semantic equation

$$\mathcal{R}[\![p_1 \ \underline{\text{orelse}} \ p_2]\!] \chi n s = \ldots .$$

and complete the definitions of $\mathcal{R}$. You should then be able to *prove*

(i) orelse is associative, i.e.

$$\mathcal{R}[\![p_1 \ \underline{\text{orelse}} (p_2 \ \underline{\text{orelse}} \ p_3)]\!] = \mathcal{R}[\![(p_1 \ \underline{\text{orelse}} \ p_2) \underline{\text{orelse}} \ p_3]\!] \ .$$

(ii) error is a left zero for " ; ", i.e.

$$\mathcal{R}[\![\underline{\text{error}} ; p]\!] = \mathcal{R}[\![\underline{\text{error}}]\!] \ .$$

(Is it a right zero?)

(iii) error is a left identity for orelse, i.e.

$$\mathcal{R}[\![\underline{\text{error}} \ \underline{\text{orelse}} \ p]\!] = \mathcal{R}[\![p]\!] \ .$$

(Is it a right identity? Is there a left or right zero for orelse?)

Does orelse distribute over " ; " ? ... over if b_i then -- else -- ?
If not always, then under what conditions?

Of course the most useful errors are those which return some kind of value, but we shall have to omit this kind of extension. Again, the techniques require no really new idea.

4.2. Techniques for proof about continuous functions

Most interesting properties of language semantics involve fixed-points, and their proofs depend upon the fact that fix Φ denotes the *least* fixed point of Φ. (The simple properties of error and orelse mentioned above are an exception.) The fundamental method is to prove that the required property holds (or something similar holds) when fix Φ is replaced by each of the iterates $f_i = \Phi^i(\perp)$ of Φ. Let us take as an example a general property of fixed points:

$$F(\mathrm{fix}(G\circ F)) = \mathrm{fix}(F\circ G),$$

provided F,G are continuous.

Proof. Let

$$H = F\circ G, \quad J = G\circ F$$

and let

$$h_i = H^i(\bot), \quad j_i = J^i(\bot).$$

We then show by induction on i that

$$(*) \qquad F(j_i) \sqsubseteq \mathrm{fix}(F\circ G),$$

whence

$$F(\mathrm{fix}\ J) = F(\bigsqcup_i j_i) = \bigsqcup_j (F(j_i)) \sqsubseteq \mathrm{fix}(F\circ G),$$

and a similar induction on the iterates h_i yields the other half of the required result.

To prove (*):

when $i = 0$, $F(j_0) = F(\bot) \sqsubseteq F(G(\mathrm{fix}\ H)) = (F\circ G)(\mathrm{fix}(F\circ G)) = \mathrm{fix}(F\circ G)$;

otherwise $F(j_{i+1}) = F((G\circ F)j_i) = (F\circ G)(Fj_i) \sqsubseteq (F\circ G)(\mathrm{fix}(F\circ G))$ (by induction)

$= \mathrm{fix}(F\circ G)$.

Hence the induction is complete. □

Exercises. Prove similarly

(i) $\mathrm{fix}\ F = \mathrm{fix}(F\circ F)$;

(ii) if $F(\bot) = G(\bot) = \bot$ and $F\circ G = G\circ F$ then $\mathrm{fix}\ F = \mathrm{fix}\ G$;

(iii) ditto, replacting the second condition by $F\circ F\circ G = G\circ F$.

The method of such proofs is to prove first that $\bar{F}[f_i]$ holds for all i, and then step to $\bar{F}[\mathrm{fix}\ \Phi]$ where $f_i = \Phi^i\bot$. For this step to be valid, the predicate $\bar{F}[\ ]$ has to be *directed-complete*; that is, for *any* chain $\langle f_i \rangle$,

$$(\forall_i \cdot \bar{F}[f_i]) \Rightarrow \bar{F}[\bigsqcup_i f_i].$$

Now any equational formula - that is $t = t'$ - or inequality $t \sqsubseteq t'$ is easily shown to be directed - complete considered as a predicate of some free variable x in the formula, provided that t and t' are built by application and abstraction from continuous functions. (This is the import of Prop. 3.1 in "Models of LCF".) It is also easy to show that if $F[x]$ is directed complete in x, so are

$$\forall y \cdot F[x]$$

$$G \Rightarrow F[x] \ ,$$

provided that x is not a free variable of G. The class of directed-complete formulae may be extended further; we merely emphasize here the importance of the notion.

4.3. The equivalence of direct and continuation semantics

We have presented two semantic definitions of language L, both guided by our intuition about what L *should* mean, and one of them (the *direct* semantics P) further substantiated by a proof of its equivalence with an operational definition. We must therefore answer the question: *in what sense do P and Q give the same meaning to* L? In some sense they simulate one another - their definitions are structurally similar - but we cannot simply claim $P = Q$, or $P[\![p]\!] = Q[\![p]\!]$, since the domains are different.

The simulation relation between direct and continuation semantics for a more complex language was exhibited by Reynolds. He used more powerful techniques than we have developed here, but for L they are unnecessary.

We will give a rather simple proof of the appropriate relationship between P and Q, and then proceed to examine how the proof might be formalized and performed interactively with a machine.

We must first omit from L the error command, since it was not handled by P. Then in view of our discussion when Q was first defined, it is natural to expect that

$$(1) \qquad \forall \chi \cdot Q[\![p]\!]\chi = \chi \circ P[\![p]\!]$$

and indeed this is readily verified from the semantic equations when p is c_i or null.

But suppose that p_0 is in some program like

$$\underline{\text{while}}\ \underline{\text{true}}\ \underline{\text{do}}\ p_1$$

which never terminates? It is not hard to verify that whatever χ, $\mathcal{Q}[\![p_0]\!]\chi = \perp$; on the other hand if we pick $\chi = \lambda s \cdot o$ for some $o \neq \perp \in O$ (i.e. χ is a constant function) then we also have $\chi \circ \mathcal{P}[\![p]\!] = \lambda s \cdot o \neq \perp$.

So some restriction on equation (1) is required. It is not hard to accept that χ should be a *strict* function - that is, it should satisfy $\chi\perp = \perp$; intuitively the continuation should be patient enough to wait for p to introduce <u>some</u> information (which in our case means a fully defined store). We therefore formulate our simulation relation thus

$$(2) \qquad \forall p \cdot \forall \chi \cdot \chi \text{ strict} \Rightarrow \mathcal{Q}[\![p]\!]\chi = \chi \circ \mathcal{P}[\![p]\!] \ ,$$

which we shall prove inductively on the structure of programs. We also need to assume that the meanings of basic instructions and boolean expressions are strict functions:

$$(3) \qquad \forall i \ \beta_i(\perp_S) = \perp_T, \quad \gamma_i(\perp_S) = \perp_S \ .$$

The following Lemma is needed:

LEMMA. $\forall p \cdot \forall \chi \cdot \chi$ strict $\Rightarrow \mathcal{Q}[\![p]\!]\chi$ strict.

Proof. By induction on the structure of p. Assume χ strict.

Basis. (i) $p = \underline{\text{null}}$. Then $(\mathcal{Q}[\![p]\!]\chi)\perp = \chi\perp = \perp$.

(ii) $p = c_i$. Then $(\mathcal{Q}[\![p]\!]\chi)\perp = \chi(\gamma_i\perp) = \chi\perp$ by (3)
$= \perp$.

Step. Assume the lemma for all subprograms of p

(iii) $p = (p_1;p_2)$. Then $\mathcal{Q}[\![p]\!]\chi = \mathcal{Q}[\![p_1]\!]\chi'$ where $\chi' = \mathcal{Q}[\![p_2]\!]\chi$. But by the Lemma for p_2 χ' is strict, hence by the lemma for p_1 so is $\mathcal{Q}[\![p]\!]\chi$.

(iv) $p = (\underline{\text{if}}\ b_i\ \underline{\text{then}}\ p_1\ \underline{\text{else}}\ p_2)$. Then $\mathcal{Q}[\![p]\!]\chi\perp = \beta_i\perp \rightarrow \ldots\ , \ldots = \perp$ by (3).

(v) $p = (\underline{\text{while}}\ b_i\ \underline{\text{do}}\ p_1)$. Then $\mathcal{Q}[\![p]\!]\chi\perp = (\text{fix}\ \Phi)\chi\perp$ $(\Phi = \lambda g \cdot \lambda\chi \cdot \lambda s \cdot \beta_i s \rightarrow \ldots\ ,$
$\ldots) = \Phi(\text{fix}\ \Phi)\chi\perp = \beta_i\perp \rightarrow \ldots\ , \ldots = \perp$ by (3). □

REMARK. In this proof the while construct caused no difficulty because it executes the test before the body. You may like to try the following exercise, in which you may need an inner induction on the iterates of a functional like Φ.

Exercise. Formulate the obvious continuation semantics of the construct

$$\underline{\text{do}}\ p_1\ \underline{\text{until}}\ b_i$$

and prove the corresponding case of the lemma. Also prove that

$$\mathcal{Q}[\![\underline{\text{if}}\ b_i\ \underline{\text{then}}\ (\underline{\text{do}}\ p_1\ \underline{\text{until}}\ b_i)\ \underline{\text{else}}\ \underline{\text{null}}]\!]$$

$$= \mathcal{Q}[\![\underline{\text{while}}\ b_i\ \underline{\text{do}}\ p_1]\!] \ .$$

(For the last part you may need to look at the style of proof of the following Theorem.)

SIMULATION THEOREM. $\forall p \cdot \forall \chi \cdot \chi$ strict $\Rightarrow \mathcal{Q}[\![p]\!]\chi = \chi \circ \mathcal{P}[\![p]\!]$.

Proof. By induction on the structure of p.

Basis. (i) $p = \underline{\text{null}}$, (ii) $p = c_i$. Both trivial.

Step. Assume the theorem for all subprograms of p, and assume χ strict.

(iii) $p = (p_1;p_2)$. Then $\mathcal{Q}[\![p]\!]\chi = \mathcal{Q}[\![p_1]\!]\chi'$ where $\chi' = \mathcal{Q}[\![p_2]\!]\chi$
$= \chi' \circ \mathcal{P}[\![p_1]\!]$ by the theorem for p_1, since χ' is strict by the Lemma,
$= \chi \circ \mathcal{P}[\![p_2]\!] \circ \mathcal{P}[\![p_1]\!]$ by the theorem for p_2
$= \chi \circ \mathcal{P}[\![p]\!]$.

(iv) $p = (\underline{\text{if}}\ b_i\ \underline{\text{then}}\ p_1\ \underline{\text{else}}\ p_2)$. Then $\mathcal{Q}[\![p]\!]\chi s = \beta_i s \to \mathcal{Q}[\![p_1]\!]\chi s, \mathcal{Q}[\![p_2]\!]\chi s$
while $(\chi \circ \mathcal{P}[\![p]\!])s = \chi(\beta_i s \to \mathcal{P}[\![p_1]\!]s, \mathcal{P}[\![p_2]\!]s)$.
The result follows by considering the three cases $\beta_i s = tt, ff, \perp_T$.

(v) $p = (\underline{\text{while}}\ b_i\ \underline{\text{do}}\ p_1)$. Then $\mathcal{Q}[\![p]\!]\chi = (\text{fix}\ \Psi)\chi$ and $\chi \circ \mathcal{P}[\![p]\!] = \chi \circ \text{fix}\ \Phi$
where $\Psi = \lambda g \cdot \lambda \chi' \cdot \lambda s' \cdot \beta_i s' \to \mathcal{Q}[\![p_1]\!](g\chi')s', \chi's'$
$\Phi = \lambda f \cdot \lambda s' \cdot \beta_i s' \to (f \circ \mathcal{P}[\![p_1]\!])s', s'$.

We need only show that if f_i, g_i are the iterates of Φ, Ψ respectively, then

for each $i \geq 0$

(4) $\quad \forall \chi' \cdot \chi' \text{strict} \Rightarrow g_i\chi' = \chi' \circ f_i$.

For then, when χ is strict $\mathcal{Q}[\![p]\!]\chi = (\bigsqcup_i g_i)\chi = \bigsqcup_i (g_i\chi) = \bigsqcup_i (\chi \circ f_i) = \chi \circ \bigsqcup_i f_i = \chi \circ \mathcal{P}[\![p]\!]$.

To prove (4); when $i = 0$ $g_0\chi' = \bot\chi' = \bot = \chi' \circ \bot$ (χ' strict) $= \chi' \circ f_0$. Now assume (4) for i, and let χ' be strict. Then

$$g_{i+1}\chi' = \beta_i s \to \mathcal{Q}[\![p_1]\!](g_i\chi')s, \chi's$$

$$= \beta_i s \to (g_i\chi' \circ \mathcal{P}[\![p_1]\!])s, \chi's \text{ by the theorem for } p_1, \text{ since}$$
$$g_i\chi' \sqsubseteq (\text{fix } \Psi)\chi' = \mathcal{Q}[\![p]\!]\chi' \text{ which is strict}$$
$$\text{by the lemma, so } g_i\chi' \text{ is also strict,}$$

$$= \beta_i s \to (\chi' \circ f_i \circ \mathcal{P}[\![p_1]\!])s, \chi's \text{ by inductive assumption for i.}$$

On the other hand

$$(\chi' \circ f_{i+1})s = \chi'(f_{i+1}s) = \chi'(\beta_i s \to (f_i \circ \mathcal{P}[\![p_1]\!])s, s)$$

and equality follows by case analysis on $\beta_i s$. □

5. MECHANIZED SEMANTICS

5.1. Deductive systems

The aim of this section is to explore the possibility of mechanizing the proof of the simulation theorem, using a formal deductive calculus which is an extension of that described in "Models of LCF", Sections 3 & 4. The extension is in two directions; more logical connectives, and more types. This system has been presented in detail in Milner, Morris and Newey [24], and we shall be more informal about it here.

Well-formed formulae (wffs). Wffs are formed from the atomic wffs (awffs) by normal use of the connectives &, $\Rightarrow$, $\forall$ (conjunction, implication and universal quantification), and a sentence is $\Gamma \vdash A$, where Γ is a set of

wffs and A is a wff. As rules of inference we add

Conjunction $\dfrac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \,\&\, B}$ *Selection* $\dfrac{\Gamma \vdash A \,\&\, B}{\Gamma \vdash A \quad \Gamma \vdash B}$

Deduction $\dfrac{\Gamma \cup \{A\} \vdash B}{\Gamma \vdash A \Rightarrow B}$ *Modus Ponens* $\dfrac{\Gamma \vdash A \quad \Delta \vdash A \Rightarrow B}{\Gamma \cup \Delta \vdash B}$

Generalization $\dfrac{\Gamma \vdash A}{\Gamma \vdash \forall x \cdot A}$ *Specialization* $\dfrac{\Gamma \vdash \forall x \cdot A}{\Gamma \vdash A\{t/x\}}$

(x not free in Γ)

Assumption $\dfrac{}{\{A\} \vdash A}$

[Note: these rules actually replace INCL, CONT and CUT of "Models of LCF".]

Types. Instead of just two basic types IND and TR, we allow any number of basic types (including TR which we rename T), and types may be built using the binary connectives + × and →. From what has gone before, we know that there is a domain (a cpo) for each type, whose name is just that type.

Two further ingredients are necessary; (a) Reflexive types, and (b) Polymorphic types. From Scott, we know that any family of recursive domain equations

$$D_1 = F_1(D_1,\ldots,D_n)$$

$$D_n = F_n(D_1,\ldots,D_n)$$

has a solution for the D_i (strictly the equality is an isomorphism) where the F_i are built from the D_i, and possibly other domains, by + × and →.

We therefore allow any family of such equations as relations over our type constants (domain names); more precisely then a type is any equivalence class of type expressions induced by these relations, and it may be named by any member of the class. These are our reflexive types.

But there are many operations which make sense at an infinity of types. Examples are the conditional and fixed point operations, and the operations fst, snd, δ_0, δ_1, ι_0, ι_1, π_0, π_1 introduced earlier. To allow

these operations to have types, we introduce type variables $\alpha,\alpha_1,\ldots,$ $\beta,\beta_1,\ldots$ and then for example fst: $\alpha\times\beta \to \alpha$, ι_0: $\alpha \to \alpha+\beta$ etc.

Simple and natural rules apply for a term or wff to be well-typed, and we do not go into them here. Type variables have no binding quantifier, but we add to our deduction rules the following

Type Instantiation $\dfrac{\Gamma \vdash A}{\Gamma \vdash A\{\tau/\alpha\}}$ where α is a type variable not in Γ, and τ is any type (possibly including variables).

This rule is invoked whenever we wish to use a "polymorphic" theorem, such as

$$\forall x\cdot \pi_0(\iota_0 x) \equiv x$$

at a particular instance of its type.

This is the basis which is intended to formalize our informal reasoning within a typed framework; from now on we shall omit types almost everywhere (except when discussing them, rather than the objects which possess them), and we would expect any tolerable mechanization to allow these omissions but to supply and check types internally.

Note: We will use "fix" rather than "Y" for the fixed point combinator. Also we shall use the constant " $\perp$ " - instead of "UU" as in "Models of LCF" - to denote $\perp$. It may be worth remarking here that the reasons for choosing TT, FF, UU in LCF were (i) t, f are far too often used as variables; similarly with T,F. (ii) In addition, the Stanford LCF was superimposed on LISP, which does quite surprising things with the atom T. (iii) TT and FF are much quicker to type than true, false. However, I suggest we pronounce TT,FF,UU "true", "false", "bottom".

5.2. Formalizing the syntax of language L

Our calculus discusses cpo's; so to discuss both the syntax *and* the semantics of L, the syntactic objects as well as the semantic ones must be found in suitable cpo's. (This will mean introducing things like the undefined program, and possibly infinite programs; these do not get in the way however!)

We define a set of mutually reflexive types

(1)
$$\begin{aligned} \text{PROGM} &= \text{NULL} + \text{INSTN} + \text{COMPD} + \text{CONDL} + \text{ITERN} \\ \text{NULL} &= \cdot \\ \text{INSTN} &= \ldots \\ \text{COMPD} &= \text{PROGM} \times \text{PROGM} \\ \text{CONDL} &= \text{BEXP} \times \text{PROGM} \times \text{PROGM} \\ \text{ITERN} &= \text{BEXP} \times \text{PROGM} \\ \text{BEXP} &= \ldots \end{aligned}$$

Assume that + and × associate to the right.

" · " is our name for the domain with a single element, which we shall denote by the constant (). This domain is exiomatized easily by

$$\vdash \forall x_0 \cdot x_0 \equiv () \qquad !$$

We have left out the definitions of INSTN and BEXP, consistent with our previous treatment. They could indeed by left unspecified, and for our theorem all we shall need is that the functions

$$\mathcal{C}\colon \text{INSTN} \to S \to S \quad \text{and} \quad \mathcal{B}\colon \text{BEXP} \to S \to T$$

satisfy

(2)
$$\vdash \forall c_{\text{INSTN}} \cdot \mathcal{C}[\![c]\!]\perp_S \equiv \perp_S$$
$$\vdash \forall b_{\text{BEXP}} \cdot \mathcal{B}[\![b]\!]\perp_S \equiv \perp_T$$

Note that our type equations really give the *abstract* syntax of L; all that is said of a while program is that it is a pair consisting of a boolean expression and a program, which is all that matters to us.

But we would like mnemonic names for the discriminators, constructors and destructors of this abstract syntax; we *define*

Discriminators
$$\begin{aligned} &\vdash \text{isnull} \equiv \delta_0 \\ &\vdash \text{isinstn} \equiv \lambda p \cdot \delta_1 p \to \delta_0(\pi_1 p), FF \\ &\vdash \text{iscompd} \equiv \lambda p \cdot \delta_1 p \to (\delta_1 \pi_1 p \to \delta_0(\pi_1(\pi_1 p)), FF), FF \\ &\text{etc.} \end{aligned}$$

all of type PRGM → T .

Constructors (3)

$$\begin{aligned}
&\vdash \text{mknull} \equiv \iota_0 && : \text{NULL} \to \text{PROGM}\\
&\vdash \text{mkinstn} \equiv \iota_1 \circ \iota_0 && : \text{INSTN} \to \text{PROGM}\\
&\vdash \text{mkcompd} \equiv \iota_1 \circ \iota_1 \circ \iota_0 && : \text{PROGM} \times \text{PROGM} \to \text{PROGM}\\
&\quad \text{etc.}
\end{aligned}$$

Destructors

$$\begin{aligned}
&\vdash \text{destnull} \equiv \pi_0 && : \text{PROGM} \to \text{NULL}\\
&\vdash \text{destinstn} \equiv \pi_0 \circ \pi_1 && : \text{PROGM} \to \text{INSTN}\\
&\vdash \text{destcompd} \equiv \pi_0 \circ \pi_1 \circ \pi_1 && : \text{PROGM} \to \text{PROGM} \times \text{PROGM}\\
&\quad \text{etc.}
\end{aligned}$$

We have worked with a *binary* disjoint sum operation (rather than a 5-*ary* one) which makes these definitions lengthy. But once we have proved standard theorems for our new operations, their definitions need never be seen again. Such theorems are

(4)

$$\begin{aligned}
&\vdash \quad \text{isnull(mknull())} && \equiv \text{TT}\\
&\vdash \forall b \forall p_1 \forall p_2 \cdot \text{iscondl(mkcondl(b,}p_1\text{,}p_2\text{))} && \equiv \text{TT}\\
&\vdash \forall b \forall p_1 \forall p_2 \cdot \text{destcondl(mkcondl(b,}p_1\text{,}p_2\text{))} && \equiv \text{TT}\\
&\vdash \quad \forall i \cdot \text{isnull(mkinstn(i))} && \equiv \text{FF}\\
&\quad \text{etc., etc..}
\end{aligned}$$

We shall use them later as simplification rules (which we shall describe) in proving the main theorem - indeed, these theorems themselves are proved by using the earlier definitions as simplification rules, i.e. they are proved completely automatically.

Structural Induction for L

We wish to derive, from the standard rule of computation induction given in "Models of LCF", the following inference rule for programs of L:

(5)

$$\frac{\begin{array}{lll}\vdash F[\bot] & \vdash F[\text{mknull()}] & \vdash F[\text{mkinstn(i)}]\\ F[p_1], F[p_2] \vdash F[\text{mkcompd}(p_1,p_2)] & & \\ F[p_1], F[p_2] \vdash F[\text{mkcondl}(b,p_1,p_2)] & & \\ F[p_1] \vdash F[\text{mkitern}(b,p_1)] & & \end{array}}{\vdash F[p]} \ ,$$

where extra assumptions on the left of the turnstiles have been left out

for clarity, but may occur (with their union occurring in the conclusion of the rule) provided they contain none of i,b,p_1,p_2,p free.

We now have to face a problem of all reflexive domain equations, that they do not necessarily have a unique solution. Our rule will only follow if the domains PROGM,... are in some sense the *least* solution of equations (1). The following axiom ensures this; what it expresses is roughly that every program is well-founded - i.e. if we analyse it into its primitive components (in NULL, INSTN and BEXPN) and then build it up again, we have back our original program. The axiom is made more concise with the following functional #:

$$\vdash \forall f \forall g \forall x \forall y \cdot (f\#g)(x,y) \equiv (f(x),g(y)).$$

The axiom is:

(6) $\quad \vdash \lambda p \cdot p \equiv \text{fix progfun},$

where

$$\begin{aligned} \vdash \text{progfun} \equiv \lambda f \cdot \lambda p \cdot\ & \text{isnull } p \rightarrow \text{mknull(destnull } p), \\ & \text{isinstn } p \rightarrow \text{mkinstn(destinstn } p), \\ & \text{iscompd } p \rightarrow \text{mkcompd((f\#f)(destcompd } p)), \\ & \text{iscondl } p \rightarrow \text{mkcondl(((}\lambda b \cdot b)\#(f\#f))\text{(destcondl } p)), \\ & \text{isitern } p \rightarrow \text{mkitern(((}\lambda b \cdot b)\#f)\text{(destitern } p)), \\ & \perp. \end{aligned}$$

Now we are at last ready to derive the structural induction rule (5), for an arbitrary formula $F[p]$.

We take the instance of the computational induction rule on the functional progfun, in which $G[f]$ is $\forall p \cdot F[f(p)]$:

(7) $$\frac{\vdash G[\perp] \quad G[f] \vdash G[\text{progfun}(f)]}{\vdash G[\text{fix(progfun)}]} \quad .$$

Our task is to prove the two hypotheses of (7) from the six hypotheses of (5); (7) then allows us to infer (together with (6)) that $\vdash G[\lambda p \cdot p]$, i.e. $\vdash \forall p \cdot F[p]$, whence the conclusion of (5) follows by specialization.

Basis. $G[\perp]$ is $\forall p \cdot F[\perp(p)]$, or $\forall p \cdot F[\perp]$ which is the generalization of the first hypothesis of (5).

Step. Assume

(8) $G[f]$, that is $\forall p \cdot F[f(p)]$.

We require to prove $\forall p \cdot F[\text{progfun}(f)p]$, so we attempt to prove $F[\text{progfun}(f)p]$, for some arbitrary p.
For this, it is enough to consider the truth values of the five conditions isnull(p),isinstn(p),... . Now from the definition of progfun, the only cases which do not yield progfun(f)p $\equiv \perp$ (when we are done) are when some condition is TT and the earlier ones are all FF. In each case, progfun(f)p is equal to one of the five expressions at the right of $\rightarrow$ in (6). Consider just the fourth case. Then we are trying to prove

$$F[\text{mkcondl}(((\lambda b \cdot b)\#(f\#f))(\text{destcondl}(p)))]$$

that is (for some b,p_1 and p_2)

$$F[\text{mkcondl}(b,f(p_1),f(p_2))],$$

which follows readily, by (8), from the fifth hypothesis of (5). □

5.3. Strategies

We now attempt to describe the kind of proof strategy which can relieve one of a morass of technical detail in performing proofs interactively with a machine, using the Simulation Theorem as an example. Because induction (either structural or computational) appears to be the major creative ingredient in such proofs, there is some hope that without too much ill-directed search the machine can automatically dispose of large parts of a proof, given only an initial hint of what induction to perform. Indeed, once this is achieved one may expect to design strategies which make an intelligent search for the right induction; such strategies have already been studied with some success (though in restricted problem domains) by Boyer and Moore [25], Aubin [26] and von Henke [27] among others.

A major part of such strategies will be the use of equational formulae as *simplification rules* (which we abbreviate to *simprules*). More

precisely, any theorem of the form

$$\Gamma \vdash \forall x_1,\ldots,x_n \cdot t[x_1,\ldots,x_n] \equiv t'[x_1,\ldots,x_n],$$

which belongs to the current *simplification set* (*simpset*) will be used in the following way when simplification is explicitly called for in a strategy: to transform a *goal* F (formula to be proved) into a simpler goal F', any subterm having the form $t[u_1,\ldots,u_n]$ is replace by $t'[u_1,\ldots,u_n]$. If after all possible such replacements F' is discovered to be a simple tautology, then the goal is achieved. As the strategy proceeds, transforming the original goal into a list of subgoals, each subgoal attains a simpset appropriate for its proof.

Other components of a suitable strategy for our example will emerge in the following analysis; the resulting strategy will be seen to be not especially oriented to the example, though we would certainly not claim universal applicability for it.

As a starting point, let us now specify our main goal $G_0[p]$ as follows:

$$(G_0) \qquad \forall \chi \cdot \chi\bot \equiv \bot \Rightarrow \forall s \cdot Q[\![p]\!]\chi s \equiv \chi(P[\![p]\!]s)$$

and suppose that we have in the initial simpset the following theorems:

1. Each clause of the definitions of P and Q, in the form*)
   $\vdash \forall b,p_1,p_2,s \cdot P[\![\mathrm{mkcondl}(b,p_1,p_2)]\!]s \equiv \mathcal{B}[\![b]\!]s \to P[\![p_1]\!]s, P[\![p_2]\!]s$, together with the strictness clauses $P[\![\bot_{\mathrm{PROGM}}]\!] = \bot$, $Q[\![\bot_{\mathrm{PROGM}}]\!] = \bot$.
2. All the theorems (4) above concerning the syntax of L.
3. The strictness theorems (2) for $\mathcal{B}$ and C.
4. Standard rules such as β conversion, conditional conversion ($\vdash \forall x,y \cdot TT \to x,y \equiv x$, etc.), minimality ($\forall x \cdot \bot x \equiv x$) and rules concerning fst, snd, δ_i, π_i, ι_i.

---

*) although we continue to use [[ ]] as decorative brackets, they have no formal significance to distinguish them from ( ).

First tactic

Try structural induction on p, then simpify the resulting six subgoals; for each remaining subgoal add its assumptions to the simpset for that goal.

Applying this to G_0 gives (before simplification) the six hypotheses of rule (5) (with G_0 for F). Simplification however eliminates the first three - those with no assumptions - provided a wide enough class of simple tautologies is detected. The first hypothesis, for example, becomes

$$\forall\chi\cdot\chi\bot \equiv \bot \Rightarrow \forall s\cdot\bot \equiv \chi\bot$$

and the other two yield even simpler tautologies. We are therefore left with

(G_1) $\quad \forall\chi\cdot\chi\bot \equiv \bot \Rightarrow \forall s\cdot Q[\![p_1]\!](Q[\![p_2]\!]\chi)s \equiv \chi(P[\![p_2]\!](P[\![p_1]\!]s))$

with $G_0[p_1]$ and $G_0[p_2]$ in the simpset,

(G_2) $\quad \forall\chi\cdot\chi\bot \equiv \bot \Rightarrow \forall s\cdot B[\![b]\!]s \rightarrow Q[\![p_1]\!]\chi s, Q[\![p_2]\!]\chi s \equiv \chi(B[\![b]\!]s \rightarrow P[\![p_1]\!]s, P[\![p_2]\!]s)$

with $G_0[p_1]$ and $G_0[p_2]$ in the simpset,

(G_3) $\quad \forall\chi\cdot\chi\bot \equiv \bot \Rightarrow \forall s\cdot \mathrm{fix}\,\Psi\chi s \equiv \chi(\mathrm{fix}\,\Phi\, s)$ *)

with $G_0[p_1]$ in the simpset.

But the reader will have already noticed that the various formulae $G_0[p_1]$ added to the simpsets are not equational, and we must therefore extend our notion of simprule to justify what we have done. We introduce the notion of a *conditional* simprule. Any theorem of the form

$$\Gamma \vdash \forall \underline{y}(t_2[\underline{y}] \equiv t_2'[\underline{y}] \Rightarrow \forall x\cdot t_1[\underline{x},\underline{y}] \equiv t_1'[\underline{x},\underline{y}])$$

(in which $\underline{y}$, $\underline{x}$ stand for vectors of variables) may be used in simplifying a goal F as follows: any subterm of F having the form $t_1[\underline{u},\underline{v}]$ may be

---

*) We use Ψ and Φ here as *abbreviations* for our familiar functionals; they are not variables of the calculus.

replaced by $t_1'[\underline{u},\underline{v}]$ (where $\underline{u},\underline{v}$ are vectors of terms) *provided that*

(*) $\quad t_2[\underline{v}] \equiv t_2'[\underline{v}]$

is first proved by simplification. For pragmatic purposes we add one further constraint: that each of the terms v_i should be *free in* F, i.e. no variable occurrence free in v_i is bound in F. This constraint is not applied to the u_i, either here or in ordinary simplification; it is present to prevent conditional simplification setting up for itself too many unachievable subsidiary goals of the form of (*) above. We will see how it works later. [Aside: even our ordinary simplification mechanism (without conditional simprules) runs the resk of non-termination, unless some constraint is placed on the *form* of simplification rules. Without this, perhaps it should be called *computation* rather than simplification; the point is that some automatic equational transformation is needed, and it can always be bounded artificially in some way.]

With this extended notion, we may now assume that our strictness lemma

$$\vdash \forall p \cdot \chi \cdot \chi\bot \equiv \bot \Rightarrow Q[\![p]\!]\chi\bot \equiv \bot$$

is present in the simpset throughout.

Second tactic

Each of G_i (i = 1,2,3) is a quantified implication, and we now iterate the process of *stripping* quantifiers and *assuming* antecedents. These are normal informal proof techniques, and are justified respectively by the rules of generalisation and deduction given earlier. We choose arbitrary new variables for those which become unbound.

We then add the assumed antecedent into the simpset for each subgoal, and apply simplification.

What happens to G_1 under this tactic? Before the simplification it becomes

$$Q[\![p_1]\!](Q[\![p_2]\!]\chi_1)s_1 \equiv \chi_1(P[\![p_2]\!](P[\![p_1]\!]s_1))$$

with $G_0[p_1]$, $G_0[p_2]$ and $\chi_1\bot \equiv \bot$ in the simpset.

Now the subterm $\mathcal{Q}[\![p_2]\!]\chi_1$, being *free*, is an admissible instance for χ in the conditional simprule $G_0[p_1]$, enabling the left hand side to become

$$(*) \qquad \mathcal{Q}[\![p_2]\!]\chi_1(\mathcal{P}[\![p_1]\!]s),$$

provided that

$$\mathcal{Q}[\![p_2]\!]\chi_1\perp \equiv \perp$$

can be proved by simplification. But this is done by use of the strictness lemma, and by $\chi_1\perp \equiv \perp$. Returning to (*), it is similarly transformed further (using $G_0[p_2]$) into the right hand side.

Thus G_1 has become a trivial equation, and is achieved.

What happens to G_2 under the second tactic? Similar use of conditional simplification easily reduces it (as the reader may like to check) to

$$(G_4) \qquad \mathcal{B}[\![b]\!]s_2 \to \chi_2(\mathcal{P}[\![p_1]\!]s_2),\chi_2(\mathcal{P}[\![p_2]\!]s_2) \equiv \chi_2(\mathcal{B}[\![b]\!]s_2 \to \mathcal{P}[\![p_1]\!]s_2,\mathcal{P}[\![p_2]\!]s_2)$$

with $G_0[p_1]$, $G_0[p_2]$ and $\chi_2\perp \equiv \perp$ in the simpset.

The simple task of proving G_4 we leave to the third tactic.

What happens to G_3 under the second tactic? Before simplification it becomes

$$(G_5) \qquad \mathrm{fix}(\lambda g\cdot\lambda\chi'\cdot\lambda s'\cdot(\mathcal{B}[\![b]\!]s' \to \mathcal{Q}[\![p_1]\!](g\chi')s',\chi's'))\chi_3 s_3 \equiv \chi_3(\mathrm{fix}(\lambda f\cdot\lambda s'\cdot\mathcal{B}[\![b]\!]s' \to f(\mathcal{P}[\![p_1]\!]s'),s'))s_3)$$

with $G_0[p_1]$ and $\chi_3\perp \equiv \perp$.

Now in this case, conditional simplification by $G_0[p_1]$ is not possible, since the required instance $g\chi'$ of χ in that rule is not free in G_5. (If we relaxed this constraint on conditional simplification, because χ' is bound in the left hand side of G_5 we would need to prove $\chi'\perp \equiv \perp$ for an *arbitrary* χ' to achieve the subsidiary goal of the conditional simplification, and this is not valid.)

So simplification does nothing for G_5.

Third tactic

A tactic which will achieve G_4 and is of wide application is: find any term t of type T which is free in a goal G, and produce three subgoals, each consisting of G with respectively t ≡ TT, t ≡ FF and t ≡ ⊥ in the simpset. Then simplify.

G_4 yields very simply to this tactic - the only candidate for t is $\mathcal{B}[\![b]\!]s_2$ - and the strictness of χ_2 disposes of the third subgoal. On the other hand G_5 is not amenable to the tactic, since $\mathcal{B}[\![b]\!]s'$ is not free in G_5.

The third tactic is of wider applicability than to the type T; one may perform case analysis on any term denoting a member of a finite domain.

5.4. Discussion

Let us suppose that our overall strategy is to apply the three tactics in sequence, each to all the subgoals remaining after applying the previous one. The effect for our example is to reduce the original goal G_0 to a single subgoal G_5, for which as we saw earlier a further induction is required. Let us briefly discuss G_5, then consider strategies in general.

The form of G_5 is

$$(\text{fix } \Psi)\chi_3 s_3 \equiv \chi_3((\text{fix } \Phi)s_3),$$

with $G_0[p_1]$ and $\chi_3\bot \equiv \bot$ in the simpset. Now our informal proof consisted in an inductive proof that for each i ≥ 0

$$(*) \qquad \forall\chi\cdot\chi\bot \equiv \bot \Rightarrow \forall s\cdot g_i\chi s \equiv \chi(f_i s),$$

where g_i, f_i are the iterates of Ψ, Φ. Formally we might therefore apply the rule of *parallel* induction:

$$\frac{F[\bot,\bot] \quad F[f,g] \vdash F[\Phi f,\Psi g]}{[F \text{ fix } \Phi, \text{fix } \Psi]}$$

(which is easily derived from the standard rule), taking for $F[f,g]$ the formula (*) with suffix i removed. (In passing we may remark that the

principle reason for using such induction rules rather than *mathematical* induction on the index of the iterates is to avoid formalizing arithmetic solely for this purpose.)

Without troubling with details, we claim that our original strategy, but with structural induction replaced by parallel induction will indeed achieve the goal G_3 (from which G_5 came); but it is worth remarking that the generation of G_5, resulting from uniform application of all three tactics, was wasted work.

The detailed study of this example leaves us with the impression that strategies for certain classes of problems may often be built from rather general purpose tactical material, but that it would be unwise to pursue the ideal of a single general purpose strategy.

For this very reason, a user of an interactive proof system must have the ability to extend not only his repertoire of Theorems, but also his repertoire of tactics, of ways of composing strategies from them, and hence of strategies themselves. This sounds very like programming; the problem then is to give him a programming language (a meta-LCF) in which it can all be done tolerably, and in which however badly he programs he cannot "prove" non-theorems. He will then not need to study a strategy at length in the abstract before typing it in and trying it; if he thinks that

> *structural induction on p, then stripping quantifiers and antecedents and doing case analysis, all mixed up with simplification*

is a recipe worth trying, then he can type it in, at perhaps no greater length than the above sentence, and see what happens.

Work with LCF at Stanford [18,19,20,21,22,23] has motivated the design of such a programming meta-language, and at Edinburgh we have implemented one which appears to allow plenty of scope for experiment in strategy-building.

As for the deductive calculus, we would not claim that the one outlined in this paper is the final answer, even for problems in the rather special area of programming language semantics; we expect to continue to find problems whose expression or solution is either impossible or repulsive however much we enrich the calculus. But in contrast to this, it *is* reasonable to expect that many pragmatic aspects of interactive proof-finding remain constant as the calculus varies. That is to say, a

good *meta*-language for proof may not be so far away. (This project has involved four people - initially L. Morris and M. Newey, and currently M. Gordon and C. Wadsworth - besides myself; much of the work remains still to be reported, but I would like to acknowledge here the very able and persistent work of these colleagues.)

6. LITERATURE

Scott and Strachey [1] give a starting point for the study of denotational semantics. Scott [2] provided the underlying models; in [3] he gives an outline without too much technical detail, and in [4] he carries the theory further.

Further studies in denotational semantics are given by Tennent [5], which contains both a good introduction and a presentation of the semantics of Reynolds' GEDANKEN. Mosses [6] presents ALGOL 60, and Gordon [7] presents LISP.

For operational semantics, Landin [10] gives a starting point. The description of PL/1 is by Lucas and Walk [11]; also Wegner [12] gives a very readable account of the Vienna Definition Language which was invented for the description of PL/1. Plotkin [13] at a more fundamental level discusses evaluation in the λ-calculus.

The continuation technique of Wadsworth and Morris is presented by Reynolds [14], who also gives a mathematical discussion of the directed complete relations which are employed in his analogue of our simulation theorem. Strachey and Wadsworth [15] illustrate the continuation technique.

For a study of the syntactic properties of formulae which express directed complete relations - i.e. formulae which admit the use of computation induction - see Klebansky et al [16] and Igarashi [17]. The forerunner of the computation induction was "recursion induction" given by McCarthy in [8]. Scott's rule was also discovered independently by Park [9].

The implementation of LCF carried out at Stanford in 1971-2 is described in Milner [18], and studies in its use are Milner and Weyhrauch [19], Weyhrauch and Milner [20], Newey [22], Aiello, Aiello and Weyhrauch [21] and von Henke [23]. The extended formal calculus used in the present paper is given in full detail in Milner, Morris and Newey [24].

Work on strategies for proof by induction can be found in Boyer and

Moore [25], Aubin [26] and von Henke [27]. The models of the original LCF are in Milner [28].

This is by no means a full list of the relevant papers, but should help the reader to explore further the different aspects of work in the field of semantics and proof.

REFERENCES.

[1] D. SCOTT & C. STRACHEY, *Towards a mathematical Semantics for Computer Languages,* Proc. Symposium on Computers and Automata, Microwave Res. Inst. Symposia series, Vol. 21, Polytechnic Institute of Brooklyn, 1971.

[2] D. SCOTT, *Lattice Theoretic Models for Various Type-free Calculi,* Proc. IV-th International Congress for Logic, Methodology and the Philosophy of Science, Bucharest, 1972.

[3] D. SCOTT, *Outline of a Mathematical Theory of Computation,* Proc. Fourth Annual Princeton Conference on Information Sciences and Systems, 1970.

[4] D. SCOTT, *Data Types as Lattices,* Unpublished Lecture, University of Amsterdam, 1973.

[5] R. TENNENT, *The Denotational Semantics of Programming Languages,* Comm. A.C.M., Vol. 19, No. 8, 1976.

[6] P. MOSSES, *The Mathematical Semantics of ALGOL 60,* Technical Monograph PRG-12, Oxford University Computing Laboratory, Programming Research Group, 1974.

[7] M. GORDON, *Models of pure LISP,* Experimental Programming Report 37, School of Artificial Intelligence, University of Edinburgh, 1973.

[8] J. McCARTHY, *A Basis for a Mathematical Theory of Computation,* Computer Programming and Formal System, D. Braffort & D. Hirshberg eds., North-Holland, Amsterdam, 1963.

[9] D. PARK, *Fixpoint Induction and Proofs of Program Properties,* Machine Intelligence 5, B. Meltzer & D. Michie eds., Edinburgh University Press, 1969.

[10] P. LANDIN, *The Mechanical Evaluation of Expressions*, Computer Journal 6, 4, 1964.

[11] P. LUCAS & K. WALK, *On the formal Descriptions of PL/]*, Annual Review in Automatic Programming 6, 3, 1969.

[12] P. WEGNER, *The Vienna Definition Language*, ACM Computing Surveys, 4, 1, 1972.

[13] G. PLOTKIN, *Call by name, call by value and the λ-calculus*, Theoretical Computer Science, 1, 2, 1975.

[14] J. REYNOLDS, *On the relation between direct and continuation semantics*, Proc. 2-nd Colloquium on Automata, Languages and Programming, Saarbrücken, 1974.

[15] C. STRACHEY & C. WADSWORTH, *Continuations: A mathematical semantics for handling full jumps*, Technical Monograph PRG-11, Oxford University, Computing Laboratory, Programming Research Group, 1974.

[16] B. KLEBANSKY, Z. MANNA & A. PNUELI, *Formulas admissible for Induction*, Dept. of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel, 1973.

[17] S. IGARASHI, *Admissibility of Fixed-Point Induction in First Order Logic of Typed Theories*, AI Memo AIM-168, Computer Science Dept., Stanford University, 1972.

[18] R. MILNER, *Logic for Computable Functions; Description of a Machine Implementation*, AI Memo No. 169, Computer Science Dept., Stanford, 1972.

[19] R. MILNER & R. WEHRAUCH, *Proving compiler correctness in a Mechanized Logic*, in Machine Intelligence 7, ed. D. Michie, Edinburgh University Press, 1972.

[20] R. WEYHRAUCH & R. MILNER, *Program semantics and correctness in a Mechanized Logic*, Proc. USA - Japan Computer Conference, Tokyo, 1972.

[21] L. AIELLO, M. AIELLO & R. WEYHRAUCH, *The semantics of PASCAL in LCF*, AIM-221 Computer Science Dept., Stanford University, 1974.

[22] M. NEWEY, *Formal Semantics of LISP with applications to program correctness*, AIM-243, Stanford University, Computer Science Dept., 1975.

[23] F. VON HENKE, *Notes on Automating theorem proving in LCF*, forthcoming memorandum, 1976.

[24] R. MILNER, L. MORRIS & M. NEWEY, *A Logic for Computable Functions with Reflexive and Polymorphic Types*, Proc. Conference on Proving and Improving Programs, Arc-et-Senans, 1975.

[25] R. BOYER & J. MOORE, *Proving Theorems about LISP functions*, J. ACM 22, 1975 (pp.129-144).

[26] R. AUBIN, *Some generalization heuristics in proofs by induction*, Proc. Conference on Proving and Improving Programs, Arc-et-Senans, 1975.

[27] F. VON HENKE, *On Automating Proofs by Induction*, unpublished paper, 1976.

[28] R. MILNER, *Models of LCF*, AI Memo No. 186, Computer Science Dept., 1973 (also in this volume; see p.49).

MODELS OF LCF by R. MILNER

MATHEMATICAL CENTRE TRACTS 82, 1976, 49-63

MODELS OF LCF*

R. MILNER
University of Edinburgh, Edinburgh, U.K.

1. INTRODUCTION

The logic of computable functions proposed by Dana Scott in 1969, in an unpublished note, has since been the subject of an interactive proof-checking program designed as a first step in formally based machine-assisted reasoning about computer programs. This implementation is fully documented in [1], and its subsequent applications are reported in later papers [2,3,4 and 5]. However the model theory of the logic, which Scott originally supplied, is not discussed in those papers, and the purpose of this Memorandum is to present that theory. Nothing is added here to Scott's work. The concept of a continuous function, which is central to the theory, has since been developed by him to provide models for the λ-calculus and to yield his mathematical theory of continuous lattices; the interested reader can follow these topics in Scott [6]. However, since LCF is only a version of the *typed* λ-calculus, these developments are not necessary for the present purpose, and the present paper contains all that is needed to understand LCF.

2. CONTINUOUS FUNCTION DOMAINS

In this section we define a particular sort of partially ordered domain, called a complete partial order (cpo), and the concept of continuous

---

*) This paper appeared first as Memo AI-186 and Report CS-332 of the Computer Science Department, Stanford University, California, and was written while the author worked at the Artificial Intelligence Laboratory there in 1972. The research was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense, USA, under Contract No. SD-183.

function. We prove some propositions for later use; in particular, that if D and E are cpo's, then the set of continuous functions from D to E is itself a cpo.

DEFINITION 2.1. A *partial order* (po) is a pair $(D,\sqsubseteq)$ where D is any set (domain) and $\sqsubseteq$ is a transitive, reflexive, antisymmetric relation over D.

DEFINITION 2.2. For a po $(D,\sqsubseteq)$, a set $X \subseteq D$ is a *chain* if $X = \{x_i \mid i \geq 0\}$ and $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \ldots$.

DEFINITION 2.3. A po $(D,\sqsubseteq)$ is a *complete partial order* (cpo) if

(1) it has a minimum element, which we denote by $\perp_D$, or just $\perp$ if there is no confusion;

(2) every chain $X \subseteq D$ has a least upper bound (lub) in D, which we denote by $\sqcup X$.

DEFINITION 2.4. If D and E are cpo's, then a function $f: D \to E$ is *continuous* if every chain $X \subseteq D$ satisfies $\sqcup\{f(x) : x \in X\} = f(\sqcup X)$.

Thus a continuous function is one which preserves the lubs of chains. Note that the set on the lefthand side of the above equation is a chain, since if $X = \{x_0,x_1,\ldots\}$ and $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$ then we also have $f(x_0) \sqsubseteq f(x_1) \sqsubseteq \ldots$. To see this, we only need to observe that any continuous function is monotonic - that is, $x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$, and this is true because if Y is the chain $\{x,y\}$ then $\sqcup y = y$, so we have

$$f(x) \sqsubseteq \sqcup\{f(x),f(y)\} = f(\sqcup Y) = f(y).$$

We should also note that there is an alternative (more restrictive) definition of a cpo which uses the concept of *directed set* (X is directed iff $x,y \in X \Rightarrow \exists z \in X \cdot x,y \sqsubseteq z$) instead of a chain. This, in turn, leads to an alternative (more restrictive) definition of continuous function. We have chosen the less restrictive alternative, but we remark that the theory can be done equally well (as far as we are here concerned) with either definition.

Notice that we use the same symbol $\sqsubseteq$ for the relation in every po under discussion. This should give no difficulty. We also use names like

D and E both for po's and for their domains.

DEFINITION 2.5. We denote the set of continuous functions from D to E, where these are cpo's, by $[D \to E]$.

PROPOSITION 2.1. *If* D *and* E *are cpo's then* $F = [D \to E]$ *is a cpo under the relation*

$$f \sqsubseteq g \quad \textit{iff} \quad \forall x \cdot f(x) \sqsubseteq g(x).$$

Proof. First, f *is* a po under this relation (check reflexivity, transitivity and antisymmetry). Second, the minimum element $\perp_F$ of F is easily seen to be $\lambda x \cdot \perp_E$. Finally, we need that any chain $Z \subseteq F$ has a lub $\sqcup Z \in F$. *Define*

$$\sqcup Z = \lambda x \cdot \sqcup\{f(x) : f \in Z\}.$$

This is a well-defined function since for each $x \in D$, $\{f(x) : f \in Z\}$ is easily seen to be a chain in E. Next, it bounds above every $f \in Z$, since for each $x \in D$, $f(x) \sqsubseteq \sqcup\{f(x) : f \in Z\} = (\sqcup Z)(x)$. Further, it is a lub, since if h is any other upper bound for Z, then for each $x \in D$ and $f \in Z$, we have $f(x) \sqsubseteq h(x)$; it follows that $(\sqcup Z)(x) \sqsubseteq h(x)$, and hence $\sqcup Z \sqsubseteq h$.

But we must also show that $\sqcup Z \in F$, i.e., $\sqcup Z$ is *continuous*. Let $X \subseteq D$ be a chain. We require

$$(\sqcup Z)(\sqcup X) = \sqcup\{(\sqcup Z)(x) : x \in X\},$$

but

$$\begin{aligned}(\sqcup Z)(\sqcup X) &= \sqcup\{f(\sqcup X) : f \in Z\} \qquad \text{by the definition of } \sqcup Z,\\ &= \sqcup\{f(x) : f \in Z,\ x \in X\}\\ &= \sqcup\{(\sqcup Z)(x) : x \in X\}.\end{aligned}$$

This completes the proof. □

PROPOSITION 2.2. *For any cpo* D, *every* $f \in [D \to D]$ *has a minimum fixed-point* $Yf \in D$, *i.e. we have* $f(Yf) = Yf$ *and for all* $x \in D$, $f(x) = x$ *implies* $Yf \sqsubseteq x$.

REMARK. This proposition ensures the existence of the least fixed-point operator $Y : [D \to D] \to D$. The next proposition shows that Y is continuous, i.e. $Y \in [[D \to D] \to D]$.

Proof. The set $S = \{f^i(\perp_D) : 0 \leq i\}$ is a chain by the monotonicity of f. Define

$$Yf = \sqcup S.$$

By the continuity of f, we have $f(Yf) = \sqcup\{f^{i+1}(\perp_D) : 0 \leq i\} = Yf$, so Yf is a fixed-point of f. Let x be any other fixed-point. Now by the monotonicity of f we have

$$f(\perp_D) \sqsubseteq f(x) = x,$$

and, by induction on i we can show

$$f^i(\perp_D) \sqsubseteq x \quad \text{for all } i \geq 0,$$

so

$$Yf = \sqcup\{f^i(\perp_D) : 0 \leq i\} \sqsubseteq x,$$

and thus Yf is the minimum fixed-point of f. □

PROPOSITION 2.3. Y *is continuous, so* $Y \in [[D \to D] \to D]$.

Proof. Let Z be any chain $\subseteq [D \to D]$. We must show that $Y(\sqcup Z) = \sqcup\{Yf : f \in Z\}$. In one direction ($\sqsupseteq$) proof is easy since for each $f \in Z$, $\sqcup Z \sqsupseteq f$, so $Y(\sqcup Z) \sqsupseteq Yf$ by the monotonicity of Y which in turn follows directly from the definition of Yf. In the other direction we only need to show that $\sqcup\{Yf : f \in Z\}$ is a fixed-point of $\sqcup Z$, since then it dominates the least such, which is $Y(\sqcup Z)$. Now

$$\begin{aligned}
\sqcup Z(\sqcup\{Yf : f \in Z\}) &= \sqcup\{g(\sqcup\{Yf : f \in Z\}) : g \in Z\} \\
&= \sqcup\{g(Yf) : g \in Z,\ f \in Z\} \text{ by continuity of } g, \\
&= \sqcup\{f(Yf) : f \in Z\}, \qquad \text{since} \\
&\qquad g(Yf) \sqsubseteq h(Yh) \text{ where } h = \max(g,f), \\
&= \sqcup\{Yf : f \in Z\},
\end{aligned}$$

which is the required fixed-point property. This completes this proof. □

3. PURE LCF: TERMS

In this section we give the term syntax of Pure LCF, and then after defining a standard interpretation as a function from identifiers into the union of a family of cpo's, we show how such an interpretation is extended uniquely to a function from *all* terms into the same range. The terms of Pure LCF are just those of a typed λ-calculus.

Types.

(1) ind and tr are (basic) types.

(2) If β1, β2 are types then (β1 → β2) is a type.

(3) These are all the types.

We use β, β1, β2,... to denote types, and frequently omit parentheses, assuming that " → " associates to the right, so that β1 → β2 → β3 abbreviates (β1 → (β2 → β3)).

Terms.

Each term has a well defined type. We use s,t,u to denote terms, and write s : β to mean that s has type β.

(1) Any identifier is an (atomic) term. We do not need to describe them, except to say that there are infinitely many at each type, that the type of each is determined in some way (perhaps by explicit subscripting), and that they include TT : tr, FF : tr and the families (indexed by type)

$$UU_{\beta},\ \supset_{tr \to \beta \to \beta \to \beta} \quad \text{and} \quad Y_{(\beta \to \beta) \to \beta}.$$

These identifiers are special only in that each standard interpretation will assign a particular element to each of them. We use x,y to denote arbitrary identifiers.

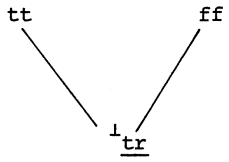
(2) If s : β1 → β2 and t : β1 are terms then s(t) : β2 is a term. If x : β1 is an identifier and s : β2 is a term, then [λx•s] : β1 → β2 is a term.

(3) These are all the terms.

REMARK. In the machine implementation of LCF, and often for intelligibility, we have written terms of the form $\supset(s)(t)(u)$ and $Y([\lambda x \cdot s])$ respectively as $(s \to t,u)$ and $[\alpha x \cdot s]$, and have dispensed with $\supset$ and Y. It is clear that every term of implemented LCF is then a transcription of a term of Pure LCF, and it therefore suffices to discuss the semantics of the latter.

Semantics.

A *standard model* (of LCF) is a family $\{D_\beta\}$ of cpo's, one for each type β, where $D_{\underline{ind}}$ is an arbitrary cpo, $D_{\underline{tr}}$ is the cpo $\{tt,ff,\perp_{\underline{tr}}\}$ under the partial order given by the diagram



and $D_{\beta 1 \to \beta 2} = [D_{\beta 1} \to D_{\beta 2}]$. Note that $D_{\underline{ind}}$ completely determines a standard model.

Let I be the set of identifiers of Pure LCF. A *standard interpretation* (of LCF) is a standard model $\{D_\beta\}$ together with a *standard assignment*, which is a function

$$A : I \to U\{D_\beta\}$$

which satisfies the further conditions

(1)*) $A[\![x : \beta]\!] \in D_\beta$,

(2) The value of A for the special identifiers is given by the following:

---

*) We write the (syntactic) arguements of A in decorated brackets as an aid to the eye.

$$A[\![TT]\!] = tt, \quad A[\![FF]\!] = ff,$$

$$A[\![UU_\beta]\!] = \perp_\beta,$$

$$A[\![\supset_{\underline{tr}} \to \beta \to \beta \to \beta]\!] = \lambda\xi \in D_{\underline{tr}} \cdot \lambda\eta \in D_\beta \cdot \lambda\chi \in D_\beta \cdot (\xi \to \eta, \chi),$$

and

$$A[\![Y_{(\beta \to \beta) \to \beta}]\!] = Y_{(\beta \to \beta) \to \beta},$$

where $(\xi \to \eta, \chi)$ - the conditional - takes the values $\perp, \eta, \chi$ according as $\xi = \perp_{\underline{tr}}$, tt, ff, and where we have subscripted the fixed-point operator Y on the right to indicate that it belongs to $[[D_\beta \to D_\beta] \to D_\beta]$. Note that the Y on the left is an identifier, and the Y on the right a function. It is easy to check that $A[\![\supset]\!]$ is a continuous function, and Proposition 2.3 has assured us that $A[\![Y]\!]$ is also continuous.

If A satisfies condition (1) above, but not necessarily condition (2), we call it just an *assignment*, yielding an *interpretation* (not necessarily standard). We also confuse the terms assignment and interpretation, since we have no occasion to discuss here different standard models.

We write $A_{\xi/x}$ to indicate the assignment differing from A only in that its value at x is ξ; clearly we have that

$$(A_{\xi/x})_{\eta/y} = \begin{cases} A_{\eta/y} & \text{if } x = y, \\ (A_{\eta/y})_{\xi/x} & \text{otherwise.} \end{cases}$$

We now show how to extend the domain of an assignment A to all terms, preserving the condition that

$$A[\![s : \beta]\!] \in D_\beta$$

which states not only that A respects types, but also that (for composite types) it yields a *continuous* function over the appropriate domains.

We define A by induction on the structure of terms, as follows:

$$A[\![s(t)]\!] = A[\![s]\!](A[\![t]\!])$$

$$A[\![[\lambda x \cdot s]]\!] = \lambda\xi \cdot A_{\xi/x}[\![s]\!].$$

That A respects types is obvious. That $A[\![s]\!] \in D_\beta$ for all β and $s : \beta$ is a

corollary of the following

PROPOSITION 3.1. *For each assignment* A *and for each* $x : \beta 1$, $s : \beta 2$, $\lambda\xi \in D_{\beta 1} \cdot A_{\xi/x}[\![s]\!] \in [D_{\beta 1} \to D_{\beta 2}]$.

Proof. First, suppose s is an atomic term, i.e. an identifier. Either $s = x$, in which case $\beta 1 = \beta 2$ and $\lambda\xi \cdot A_{\xi/x}[\![s]\!]$ is the identity function over $D_{\beta 1}$, or $s \neq x$ in which case it is a constant function from $D_{\beta 1}$ to $D_{\beta 2}$. In either case it is a continuous function, hence $\in [D_{\beta 1} \to D_{\beta 2}]$.

Next suppose s is t(u), $t : \beta 3 \to \beta 2$ and $u : \beta 3$. Assume the proposition for t and u. We have to show that for any chain $X \subseteq D_{\beta 1}$,

$$\sqcup\{A_{\xi/x}[\![t(u)]\!] : \xi \in X\} = A_{\sqcup X/x}[\![t(u)]\!];$$

that is, that

$$\sqcup\{A_{\xi/x}[\![t]\!](A_{\xi/x}[\![u]\!]) : \xi \in X\} = A_{\sqcup X/x}[\![t]\!](A_{\sqcup X/x}[\![u]\!]).$$

Now if we denote $\lambda\xi \cdot A_{\xi/x}[\![t]\!]$ and $\lambda\xi \cdot A_{\xi/x}[\![u]\!]$ by f and g, the inductive assumption tells us that $f \in [D_{\beta 1} \to [D_{\beta 3} \to D_{\beta 2}]]$ and $g \in [D_{\beta 1} \to D_{\beta 3}]$, and the required equation merely states that for such f and g, $\lambda\xi \cdot f(\xi)(g(\xi))$ is continuous. The proof of this we leave to the reader; it is hardly more than proving that for a chain X, $\{f(\xi)(g(\xi)) : \xi \in X\}$ and $\{f(\xi)(g(\eta) : \xi,\eta \in X\}$ are cofinal chains.

Finally suppose s is $[\lambda y \cdot t]$, $y : \beta 3$, $t : \beta 4$ and $\beta 2 = \beta 3 \to \beta 4$. We need to show that

$$\lambda\xi \in D_{\beta 1}, A_{\xi/x}[\![[\lambda y \cdot t]]\!] \in [D_{\beta 1} \to [D_{\beta 3} \to D_{\beta 4}]],$$

that is, that for any chain $X \subseteq D_{\beta 1}$,

$$\sqcup\{\lambda\eta \in D_{\beta 3} \cdot (A_{\xi/x})_{\eta/y}[\![t]\!] : \xi \in X\} = \lambda\eta \in D_{\beta 3} \cdot (A_{\sqcup X/x})_{\eta/y}[\![t]\!].$$

Now in the case $x = y$, we have

$$(A_{\xi/x})_{\eta/y} = (A_{\sqcup X/x})_{\eta/y} = A_{\eta/y}$$

and the equation reduces to a tautology. If $x \neq y$, then

$$(A_{\xi/x})_{\eta/y} = (A_{\eta/y})_{\xi/x},$$

and the inductive hypothesis (that the proposition is true for t) tells us that $\lambda\xi\cdot(A_{\eta/y})_{\xi/x}[\![t]\!]$ is continuous - hence monotonic - so $\{(A_{\xi/x})_{\eta/y}[\![t]\!]\}$ is a chain in $D_{\beta 4}$, for each η. Moreover, the inductive hypothesis also tells us that for each ξ $\lambda\eta\cdot(A_{\xi/x})_{\eta/y}[\![t]\!]$ is in $[D_{\beta 3} \to D_{\beta 4}]$, and by the previous remark the set of these functions - as ξ ranges over X - is a chain in $[D_{\beta 3} \to D_{\beta 4}]$. Thus by the definition of $\sqcup$ for function spaces (Proposition 2.1) we can replace the lefthand side of the desired equation by

$$\lambda\eta \in D_{\beta 3}\cdot\sqcup\{(A_{\eta/y})_{\xi/x}[\![t]\!] : \xi \in X\} = \lambda\eta \in D_{\beta 3}\cdot(A_{\eta/y})_{\sqcup X/x}[\![t]\!] =$$

$$= \lambda\eta \in D_{\beta 3}\cdot(A_{\sqcup X/x})_{\eta/y}[\![t]\!]$$

since $x \neq y$, and we are done. We have therefore proved the proposition by induction on the structure of terms. □

COROLLARY 3.2. *For every assignment* A, *type* β, *and term* $s : \beta$, $A[\![s]\!] \in D_\beta$.

Proof. For atomic terms the corollary is assured by the definition of an assignment. For λ-terms, the proposition gives the corollary directly. For an application term $s(t) : \beta$, the proposition tells us that

$$\lambda\xi \in D_{\beta 1}\cdot A_{\xi/x}[\![s(t)]\!] \in [D_{\beta 1} \to D_\beta],$$

so by application to $A[\![x]\!]$ we get

$$A[\![s(t)]\!] = A_{A[\![x]\!]/x}[\![s(t)]\!] \in D_\beta$$

as required. □

4. PURE LCF: FORMULAE, SENTENCES, RULES AND VALIDITY

In this section we define the remainder of the syntax of Pure LCF, extending the domain of assignments A still further, and after defining the concept of validity of a sentence we give the rules of inference and show that

they preserve validity.

Atomic well-formed formulae (awffs).
If $s,t : \beta$ are terms, then $s \subset t$ is an awff. Let us add the truth values T,F (not to be confused with TT, FF) to the range of an assignment, and extend any A to awffs by

$$A[\![s \subset t]\!] = \begin{cases} T & \text{if } A[\![s]\!] \sqsubseteq A[\![t]\!], \\ F & \text{otherwise.} \end{cases}$$

Well-formed formulae (wffs).
A wff is a set of awffs. We use P,Q,P1,Q1,... to denote arbitrary wffs. Extend A to wffs by

$$A[\![P]\!] = \begin{cases} T & \text{if } A \in P \Rightarrow A[\![A]\!] = T, \\ F & \text{otherwise.} \end{cases}$$

We use $s \equiv t$ to abbreviate $\{s \subset t,\ t \subset s\}$.

Sentences.
If P,Q are wffs, then $P \vdash Q$ is a sentence (if $P = \emptyset$, we just write $\vdash Q$). Extend A to sentences by

$$A[\![P \vdash Q]\!] = \begin{cases} F & \text{if } A[\![P]\!] = T,\ A[\![Q]\!] = F, \\ T & \text{otherwise.} \end{cases}$$

We say that $P \vdash Q$ is *false in* A, *true in* A respectively. We say that a sentence is *valid* iff it is true in all standard interpretations.

We now introduce the rules of inference of Pure LCF, accompanying each by a proof - often very trivial - that it is valid (a rule is valid if whenever its hypotheses are valid its conclusion is valid). The proofs will rely on two facts about assignments which are fairly easy to prove (we omit their proofs). First, if A is any syntactic entity in the domain of an assignment A, and x is not free in A, then $A[\![A]\!]$ is independent of $A[\![x]\!]$;

more precisely, $A_{\xi/x}[\![A]\!] = A[\![A]\!]$. Second, in specifying the inference rules we use A{t/x} to mean: Substitute t for x in A with suitable changes of bound variables so that no identifier free in t becomes bound after the substitution, and we need the fact that

$$A[\![A\{t/x\}]\!] = A_{A[\![t]\!]/x}[\![A]\!].$$

Rules of Inference.

We write the hypotheses of each rule above a solid line. If there are none, we omit the solid line. We use the same names for rules as in [1].

INCL $P \vdash Q$ $(Q \subseteq P)$

Clearly P true in A implies Q true in A.

CONJ $\dfrac{P \vdash Q1 \qquad P \vdash Q2}{P \vdash Q1 \cup Q2}$

Clearly valid.

CUT $\dfrac{P1 \vdash P2 \qquad P2 \vdash P3}{P1 \vdash P3}$

Clearly valid.

APPL $t \subset u \vdash s(t) \subset s(u)$.

If $A[\![t]\!] \sqsubseteq A[\![u]\!]$, then $A[\![s(t)]\!] = A[\![s]\!](A[\![t]\!]) \sqsubseteq A[\![s]\!](A[\![u]\!]) = A[\![s(u)]\!]$, using the monotonicity of $A[\![s]\!]$.

REFL $\vdash s \subset s$

Clearly valid, by reflexivity of $\sqsubseteq$.

TRANS $s \subset t,\ t \subset u \vdash s \subset u$

Clearly valid, by transitivity of $\sqsubseteq$.

MIN1 $\vdash UU \subset s$

Clearly valid, by the minimality of $\perp_\beta$.

MIN2 $\vdash UU(s) \subset UU$

Clearly valid, by the definition $\perp_{\beta 1 \to \beta 2} = \lambda \xi \in \beta 1 \cdot \perp_{\beta 2}$.

Note that in the last two rules we have omitted the type subscripts from UU, intending that they be supplied in such a way as to yield a proper awff - i.e. that the terms on either side should have the same type. We could have written $UU_{\beta 1 \to \beta 2}(s : \beta 1) \subset UU_{\beta 2}$. Similarly we will omit subscripts from $\supset$ and Y.

CONDT $\quad \vdash \supset (TT)(s)(t) \equiv s$

CONDU $\quad \vdash \supset (UU)(s)(t) \equiv UU$

CONDF $\quad \vdash \supset (FF)(s)(t) \equiv t$

These rules are justified by the standard interpretation of $\supset$.

ABSTR $\quad \dfrac{P \vdash s \subset t}{P \vdash [\lambda x \cdot s] \subset [\lambda x \cdot t]}$ $\qquad$ x not free in P.

Let A be such that $A[\![P]\!] = T$. Since x is not free in P, we have also $A_{\xi/x}[\![P]\!] = T$ for any ξ. So the hypotheses of the rule assures us that for each ξ in D_β, where $x : \beta$, $A_{\xi/x}[\![s]\!] \sqsubseteq A_{\xi/x}[\![t]\!]$. Hence

$$\lambda\xi \cdot A_{\xi/x}[\![s]\!] \sqsubseteq \lambda\xi \cdot A_{\xi/x}[\![t]\!],$$

which is to say that

$$A[\![[\lambda x \cdot s] \subset [\lambda x \cdot t]]\!] = T$$

as required.

CONV $\quad \vdash [\lambda x \cdot s](t) \equiv s\{t/x\}$

We have that $A[\![[\lambda x \cdot s](t)]\!] = (\lambda\xi \cdot A_{\xi/x}[\![s]\!])(A[\![t]\!]) = A_{A[\![t]\!]/x}[\![s]\!]$, which is equal to $A[\![s\{t/x\}]\!]$ by the second of the facts about assignments which we have assumed.

ETACONV $\quad \vdash [\lambda x \cdot y(x)] = y$, y distinct from x

$A[\![[\lambda x \cdot y(x)]]\!] = \lambda\xi \cdot A_{\xi/x}[\![y(x)]\!] = \lambda\xi \cdot A_{\xi/x}[\![y]\!](A_{\xi/x}[\![x]\!]) = \lambda\xi \cdot A[\![y]\!](\xi)$ (since x is distinct from y, so does not occur free in y), $= A[\![y]\!]$.

CASES $$\frac{P, s \equiv TT \vdash Q \quad P, s \equiv UU \vdash Q \quad P, s \equiv FF \vdash Q}{P \vdash Q}$$

Let A be such that $A[\![P]\!] = T$. Since $s : \underline{tr}$, $A[\![s]\!]$ must take one of the values $\{tt, \perp_{\underline{tr}}, ff\}$, so that one of $A[\![s \equiv TT]\!]$, $A[\![s \equiv UU]\!]$, $A[\![s \equiv FF]\!]$ takes the value T. The validity of the appropriate hypothesis ensures $A[\![Q]\!] = T$.

FIXP $\vdash Y(x) \equiv x(Y(x))$

Clearly valid, by the standard interpretation of Y.

INDUCT $$\frac{P \vdash Q\{UU/x\} \qquad P \cup Q \vdash Q\{s(x)/x\}}{P \vdash Q\{Y(s)/x\}}$$ x not free in P or s.

For simplicity, we consider just the case that Q is an awff. Moreover we can assume that it is of the form $t(x) \subset u(x)$ where x is not free in t or u, since for any term t', $A[\![t']\!] = A[\![[\lambda y \cdot t'\{y/x\}](x)]\!]$, y distinct from x, and then x is not free in $[\lambda y \cdot t'\{y/x\}]$. Let A be a standard assignment, $A[\![P]\!] = T$, and assume that $A[\![s]\!] = f$, $A[\![t]\!] = g$, $A[\![u]\!] = h$. We first show by induction on i that for each $i \geq 0$, $g(f^i(\perp_\beta)) \sqsubseteq h(f^i(\perp_\beta))$, where $x : \beta$. For $i = 0$, the first hypothesis gives that

$$A_{\perp_\beta/x}[\![Q]\!] = T,$$

that is $A[\![t]\!](\perp_\beta) \sqsubseteq A[\![u]\!](\perp_\beta)$ (since x is not free in t,u), so

$$g(\perp_\beta) \sqsubseteq h(\perp_\beta)$$

Now assume the inequality for i. That is, we assume

$$A_{f^i(\perp_\beta)/x}[\![Q]\!] = T.$$

Since x is not free in P, we also have

$$A_{f^i(\perp_\beta)/x}[\![P]\!] = T,$$

and we deduce from the second hypothesis that

$$A_{f^i(\perp_\beta)/x}[\![Q\{s(x)/x\}]\!] = T.$$

Now

$$A_{f^i(\perp_\beta)/x}[\![s(x)]\!] = f(f^i(\perp_\beta)),$$

since x is not free in s, $= f^{i+1}(\perp_\beta)$, so from the second fact which we assumed for assignments we deduce that

$$A_{f^{i+1}(\perp_\beta)/x}[\![Q]\!] = T,$$

that is

$$g(f^{i+1}(\perp_\beta)) \sqsubseteq h(f^{i+1}(\perp_\beta)).$$

So the induction is complete. Now

$$A[\![Q\{Y(s)/x\}]\!] = A_{Y(f)/x}[\![Q]\!],$$

which we require to take the value T. That is, we require $g(Y(f)) \sqsubseteq h(Y(f))$. But

$$g(Y(f)) = \sqcup\{g(f^i(\perp_\beta)) : i \geq 0\} \quad \text{(by the continuity of g)},$$

$$\sqsubseteq \sqcup\{h(f^i(\perp_\beta) : i \geq 0\} \quad \text{(by what we have proved)},$$

$$\sqsubseteq h(Y(f)) \quad \text{by the monotonicity of h},$$

and the justification is complete.

This completes also our justification of the validity of the Rules of LCF.

REFERENCES.

[1] R. MILNER, *Logic for Computable Functions. Description of a Machine Implementation*, Artificial Intelligence Laboratory Memo No. AIM-169, Computer Science Department, Stanford University (1972).

[2] R. MILNER, *Implementation and Applications of Scott's Logic for Computable Functions*, Proc. ACM Conference on Proving Assertions about Programs, New Mexico State University, Las Cruces, New Mexico (1972).

[3] R. WEYHRAUCH & R. MILNER, *Program Semantics and Correctness in a Mechanized Logic*, Proc. USA-Japan Computer Conference, Tokyo (1972).

[4] R. MILNER & R. WEYHRAUCH, *Proving Compiler Correctness in a Mechanized Logic*, Machine Intelligence 7, ed. D. Michie, Edinburgh University Press (1972).

[5] R. MILNER, *A Calculus for the Mathematical Theory of Computation*, International Symposium on Theoretical Programming, Novosibirsk, USSR (1972), Springer-Verlag Lecture Notes in Comp. Sc. 5.

[6] D. SCOTT, *Continuous Lattices*, Proc. 1971 Dalhousie Conference, Springer Lecture Note Series, Springer-Verlag, Heidelberg.

L SYSTEMS: A PARALLEL WAY OF LOOKING AT FORMAL LANGUAGES.
NEW IDEAS AND RECENT DEVELOPMENTS

by A. SALOMAA

(A Finnish motto for each chapter is given in parentheses.)

MATHEMATICAL CENTRE TRACTS 82, 1976, 67-107

L SYSTEMS: A PARALLEL WAY OF LOOKING AT FORMAL LANGUAGES. NEW IDEAS AND RECENT DEVELOPMENTS

A. SALOMAA
University of Turku, Turku, Finland

1. L BASICS

The theory of L systems originated from the work of Lindenmayer, [L]. The original aim of this theory was to provide mathematical models for the development of simple filamentous organisms. At the beginning L systems were defined as linear arrays of finite automata, later however they were reformulated into the more suitable framework of grammar-like constructs. From then on, the theory of L systems was developed essentially as a branch of formal language theory. It constitutes today one of the most vigorously investigated areas of formal language theory: so far the yearly growth in the number of papers has been exponential with base 2, and the number of people joining the "L crowd" has grown linearly with a decent factor. Indeed, following [vL1] we can say that L systems as a theory of parallel rewriting constitutes a non-parallelled theory of languages.

The purpose of these notes is to discuss recent results in the theory of L systems. By "recent" we mean things that have happened after the latest major L systems conference at Noordwijkerhout in April 1975. These notes are not intended to be self-contained. For unexplained notions concerning automata and formal languages we refer to [Sa1]. In the first two chapters of the notes we try to explain to some extent the basic notions and techniques in L systems. However, the exposition will be rather brief and sketchy. For more details and background, the reader is referred to [HR] and [RS2] (the former is more comprehensive but contains material only roughly up to 1973, the latter contains also material from 1973-74), and to [RS1] and [LR] (these are collections of articles). The present notes discuss L systems only from the mathematical and formal language theory point of view. For biological aspects, the reader is referred to [HR], [LR] and [L1].

The essential feature about L systems, as opposed to grammars, is that the rewriting of a string happens in a parallel manner, contrary to the sequential rewriting in grammars. This means that at every step of the rewriting process according to an L system every letter has to be rewritten. One step of the rewriting process according to a grammar changes only some part of the string considered.

Let us consider a very simple example. Assume that we are dealing with a context-free grammar containing the production $S \to SS$. Then, starting from S, we get any string of the form S^n, where $n \geq 1$. This follows because at one step of the rewriting process we can replace one occurrence of S by SS and leave the other occurrences unchanged. Assume next that we are dealing with an L system containing the production $S \to SS$. Then, starting from S, we get by this production only strings of the form S^{2^n}, $n \geq 0$. This follows becuase we cannot leave occurrences of S unchanged. Thus, if we are rewriting the string SS, we obtain at one step the string SSSS = S^4, and not the string S^3. On the other hand, if our L system contains also the production $S \to S$ then we can derive any string of the form S^n, $n \geq 1$.

This parallelism in rewriting reflects the basic biological motivation behind L systems. We are trying to model the development of an organism. The development takes place in a parallel way, simultaneously everywhere in the organism. Sequential rewriting is not suitable for this modeling.

The simplest version of L systems assumes that the development of a cell is free of influence of other cells. This type of L systems is customarily called a <u>0L system</u> ("0" stands for zero-sided communication between cells.) By definition, a 0L system is a triple $G = (\Sigma, P, \omega)$, where Σ is an alphabet, ω is a word over Σ, and P is a finite set of rewriting rules of the form

$$a \to x, \quad a \in \Sigma, \quad x \in \Sigma^*.$$

(It is also assumed that P contains at least one rule for each letter of Σ.) The language of G consists of all words which can be derived from ω using rules of P in the parallel way. (The meaning of this should be clear enough. The formal definition in terms of the yield-relation is left to the reader.)

As an example, consider the 0L system

({a,b},a,{a→b,b→ab}).

The first few words in the generated language are

a,b,ab,bab,abbab,bababbab,abbabbababbab.

Since the system is *deterministic* (there is only one production for each letter), its language is generated as a *sequence* in a unique way. (Deterministic systems are denoted by the letter D.) The mathematically minded reader will also notice that the lengths of the words in this sequence form the famous Fibonacci sequence. In fact, our 0L system provides a very simple way to generate the Fibonacci sequence, when compared to other possible devices in automata and formal language theory. Our system is also *propagating* (abbreviated P): there are no erasing productions, where a letter goes to the empty word λ.

In L systems with interactions, abbreviated IL systems, the productions have the form (y,a,z) → x. Such a production can be applied to rewrite the letter a in the context yaz as x. If in all productions the length of y (resp. z) equals k (resp. l), we speak of a system with <k,l> interactions. (From the biological point of view, this means that an individual cell may communicate with k of its left and l of its right neighbours.) Near the ends of the string, the missing neighbours are provided by a special letter g. For instance, the string aaa may be rewritten as bbaba by the (1,1) productions (g,a,a) → bb, (a,a,a) → ab, (a,a,g) → a.

An L system with *tables* (abbreviated T) has several sets of rewriting rules instead of just one set. At one step of the rewriting process, rules belonging to the same set have to be applied. For an L system of any type, systems of the same type and with tables may be considered. The biological motivation for introducing tables is that one may want different rules to take care of different environmental conditions (heat, light, etc.) or of different stages of development.

When defining the language generated by an L system, we have so far considered only the exhaustive approach: all words derivable from the axiom by the rules in a parallel way belong to the language. The families of languages obtained in this fashion (for instance, the family of 0L languages which we denote simply by 0L) have very weak closure properties. In addition to the exhaustive approach, various selective approaches are possible.

In such a selective approach, some "filtering mechanism" is applied like the mechanism of taking the intersection with the set of words over some terminal alphabet Δ. (This mechanism is always applied in ordinary phrase structure grammars.) Thus, an 0L system $G_1 = (\Sigma,P,\omega)$ is extended to a construct $G_2 = (\Sigma,P,\omega,\Delta)$, referred to as an E0L system (E for "extended"). The language of G_2 equals the language of G_1 intersected with Δ^*. Similarly, we may speak of ETOL systems and languages. We may also apply a homomorphism (resp. a letter-to-letter homomorphism also called a coding) to $L(G_1)$, obtaining an H0L (resp. a C0L) language. According to the well-known Ehrenfeucht-Rozenberg Theorem, all of these mechanisms coincide as far as the generative capacity is concerned: E0L = H0L = C0L and ET0L = HT0L = = CT0L. (The first equations are proved in the next chapter.) This result is very representative for L systems because nothing similar can be obtained in the sequential case. For other selective approaches in the definition of L languages, we refer to [RS2]. A particularly interesting result is that the "adult" language consisting of words deriving themselves and themselves only) are exactly the same as context-free languages.

We repeat the main items from the dictionary of L systems which will be used frequently in the sequel:

0 - context free,
T - with tables,
P - propagating,
D - deterministic (in connection with tables, each table is deterministic),
I - with interactions,
E - extensions (intersection with Δ^*),
H - homomorphic images,
C - codings (i.e., letter-to-letter homomorphic images).

Combinations are possible. Thus, we speak of EPDT0L systems and languages (whose family is denoted simply by EPDT0L).

2. L PROOFS

Because of the parallel mode in rewriting, ordinary techniques used in language theory are not as such applicable for L systems. Consider, for instance, the "pumping lemma" for context-free languages. The proof is based

on the fact that in big enough derivation trees paths with repetitions must occur. This enables us to pump because the other parts of the tree do not develop further but keep waiting for us. The last statement does not hold if rewriting happens in a parallel way and, thus, the argument is not applicable for L systems.

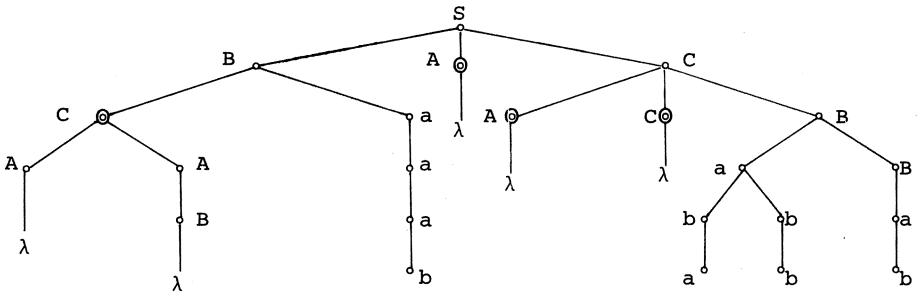
A whole bunch of new techniques have been invented for proofs dealing with L systems. As a typical example, we give in this section a proof for the Ehrenfeucht-Rozenberg Theorem EOL = HOL = COL. Our exposition runs along the lines of [RS2].

First we will prove an auxiliary result: A language K is an EOL language if and only if there exists an EPOL system G such that $K - \{\lambda\} = L(G)$.

If there exists an EPOL system G such that $K - \{\lambda\} = L(G)$, then clearly K is an EOL language.

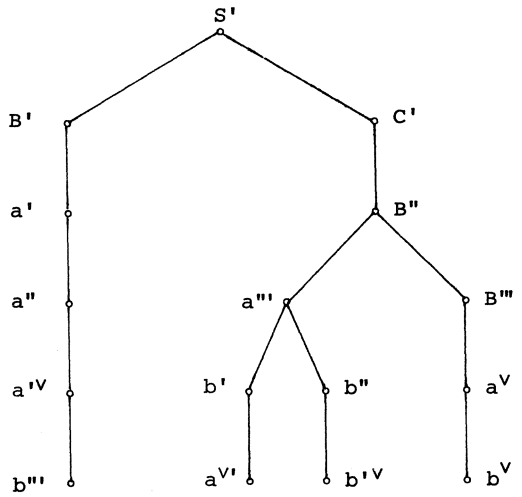
The more difficult part of the proof is to show that if K is an EOL language then there exists an EPOL system G such that $K - \{\lambda\} = L(G)$. Let $K = L(H)$ for an EOL system $H = \langle\Sigma,P,S,\Delta\rangle$ and let us assume that L(H) is infinite and P contains erasing productions (otherwise the result holds trivially). We assume that $S \in \Sigma - \Delta$. We also assume that $\Delta \subseteq \Sigma$. (This can be clearly done without loss of generality.) The idea underlying our proof can be explained rather simply. We want to construct an EPOL system G which would simulate derivations in H in such a way that in corresponding derivation trees (in G) the occurrences of symbols which do not contribute anything to the final product (word) of a tree will not be introduced at all.

Let us assume that the following tree T is a derivation tree (for a word babb) in H:



In simulating this tree in G we want to avoid the situation in which we will be forced to apply an erasing production and so we want to delete every subtree which does not "contribute" to the final result babb. Hence we want to delete subtrees with double circled roots.

We would like then to be able to produce in G a derivation tree of this form

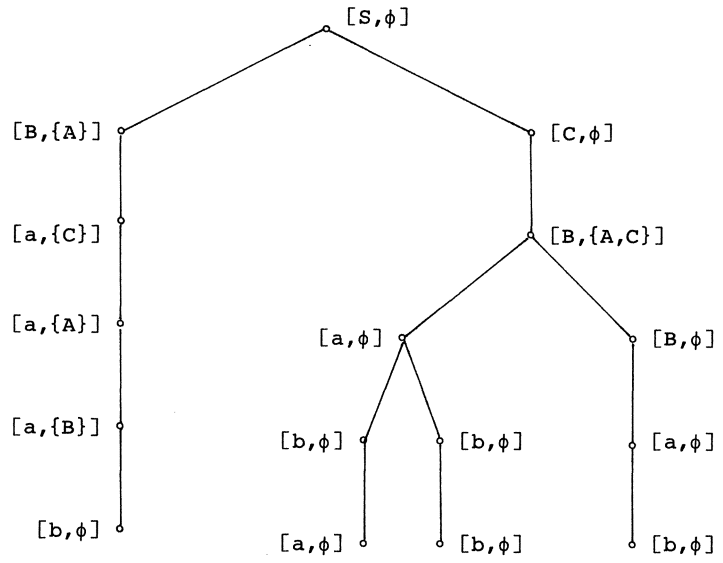


where S', B'-B'" , C',a',...,$a^{v'}$,b',...,b^{v} are some "representations" of symbols S,B,C,a,b.

In other words we are "killing" non-productive occurrences as early (going top-down) as possible. But, in general, there is no relation whatsoever between the level on which we delete (in G) a subtree at its root and the level (in H) on which this subtree really vanishes. Thus we have to carry along some information which would allow us to say (in G) at a certain moment: the considered subtree vanishes (in H). Fortunately for this purpose we can carry finite information only: it is enough to remember

the minimal subalphabet Min(x) of a word x derived so far in the considered subtree rather than the word itself. We will carry this information as the second component in two-component letters of the form $[\sigma,Z]$ where $\sigma \in \Sigma$ and $Z \subseteq \Sigma$.

Thus in our particular example we will have the following tree in G.



Now inspecting words on all levels of this tree we notice that only the last word

$$[b,\phi][a,\phi][b,\phi][b,\phi]$$

should be transformed to the terminal word (babb) because only on this level all subtrees that we have decided to delete (in G) really vanished (in H).

How to perform such a transformation within the system G itself?

To this aim we introduce a rather simple trick called "the synchronization".

For each letter of the form $[\sigma,\phi]$ with σ in Δ we introduce a production $[\sigma,\phi] \to \sigma$ and a production $\sigma \to F$ where F is a distinguished nonterminal symbol in G for which the only production in G is $F \to F$. Assuming that no other productions have terminal symbols on their right hand sides, this trick does the job. To see this observe that

$$[b,\phi][a,\phi][b,\phi][b,\phi] \underset{G}{\Rightarrow} babb \underset{G}{\Rightarrow} F^4 \underset{G}{\Rightarrow} F^4 \underset{G}{\Rightarrow} \ldots$$

but if we attempt to use these terminating productions too early then we fail to obtain a terminal word.

Now the reader should easily understand the following construction.

Let $G = \langle V,R,[S,\phi], \Delta\rangle$ be the EOL system defined by

1. $V = V_1 \cup \{F\} \cup \Delta$, where $V_1 = \{[\sigma,Z]: \sigma \in \Sigma \text{ and } Z \subseteq \Sigma\}$, and F is a new symbol.
2. R consists of the following productions:

2.1. If $A \to B_1 \ldots B_k$ is in P with $k \geq 2$, $B_1,\ldots,B_k \in \Sigma$ then, for every $Z \subseteq \Sigma$,

$[A,Z] \to [B_{i_1},Z_{i_1}][B_{i_2},Z_{i_2}] \ldots [B_{i_p},Z_{i_p}]$ is in R if

$1 \leq i_1 < i_2 < \ldots < i_p \leq k$ and,

for $2 \leq j \leq p-1$, $Z_{i_j} = \mathrm{Min}(B_{i_j+1}B_{i_j+2}\ldots B_{i_{j+1}-1})$,

$Z_{i_1} = \mathrm{Min}(B_{i_1+1}B_{i_1+2}\ldots B_{i_2-1}) \cup \mathrm{Min}(B_1B_2\ldots B_{i_1-1})$,

$Z_{i_p} = \mathrm{Min}(B_{i_p+1}B_{i_p+2}\ldots B_k) \cup Z'$

providing that $Z' \in \mathrm{Suc}_H(Z)$, where

$\mathrm{Suc}_H(Z) = \{U \subseteq \Sigma$: there exist x,y in Σ^* with $\mathrm{Min}(x) = Z$,

$\mathrm{Min}(y) = U$ where $x \underset{H}{\Rightarrow} y\}$.

2.2. If $A \to B$ is in P with B in Σ, then, for every $Z \subseteq \Sigma$, $[A,Z] \to [B,Z']$ is in R providing that $Z' \in \mathrm{Suc}_H(Z)$.

2.3. $[\sigma,\phi] \to \sigma$ is in R for all σ in Δ.

2.4. $F \to F$ is in R, and so is $\sigma \to F$ for all σ in Δ.

The reader should be able to convince himself that $L(G) = L(H) - \{\lambda\}$, and this ends the proof.

Now we present a proof for the Ehrenfeucht-Rozenberg Theorem. We also make the definitional convention that whenever a language L belongs to some of the families considered, then also $L \cup \{\lambda\}$ belongs to the same family, and vice versa. This convention is tacitly applied also many times in the later chapters, i.e., we do not care if some specific construction causes us to loose or gain the empty word.

By definition, it follows that COL $\subseteq$ HOL. After having read this chapter so far the reader should be able to produce easily the proof of

the containment HOL $\subseteq$ EOL. If one considers only non-erasing homomorphisms then the synchronization trick suffices on its own. However if one considers an erasing homomorphism then this trick by itself does not work. But then one can construct an equivalent EOL system in which the symbols which are to be erased by the considered homomorphism will not be introduced at all. Technically it can be done in exactly the same way as avoiding (occurrences of) symbols which are to be erased in the given system.

Thus we have to prove only that EOL $\subseteq$ COL. By our auxiliary result, it suffices to prove that EPOL $\subseteq$ COL.

If A is an ultimately periodic set of non-negative integers then *thres*(A) denotes the smallest integer j for which there exists a positive integer q such that, for all $i \geq j$, i is in A if and only if (i+q) is in A. The smallest positive integer q such that, for all $i \geq thres(A)$, whenever i is in A also i+q is in A, is denoted by *per*(A).

If G is an EOL system with a terminal alphabet Δ and σ is a letter in the alphabet of G, then the *spectrum of* σ in G, denoted as Spec(σ,G), is defined by $Spec(\sigma,G) = \{n \geq 0: \sigma \overset{n}{\underset{G}{\Rightarrow}} w \text{ for some } w \text{ in } \Delta^*\}$.

It is easy to see that all spectra of letters in an EOL system are ultimately periodic.

Now we need some further terminology and notation.

Let $G = \langle \Sigma,P,S,\Delta \rangle$ be an EOL system and let $\sigma \in \Sigma$. We say that σ is *vital*, if for every $k > 0$ there exists an $\ell > k$ such that $a \overset{\ell}{\Rightarrow} w$ for some $w \in \Delta^*$. (We will use A_G to denote the set of all vital symbols from Σ).

Once we have noticed that each symbol in an EPOL system G contributes terminal subwords to terminal words in G in an ultimately periodic fashion we are trying to decompose G into a (finite) number of component systems in each of which one can consider only terminal contributions at the same moments of time.

Let $G = \langle \Sigma,P,S,\Delta \rangle$ be an EPOL system. We define the *uniform period* of G, denoted as m_G, to be the smallest positive integer such that

(i) for all $k \geq m_G$, if a is in $\Sigma - A_G$ and $a \overset{k}{\underset{G}{\Rightarrow}} w$, then $w \notin \Delta^*$,

(ii) for all a in A_G, $m_G > thres(Spec(G,a))$ and $per(Spec(G,a))$ divides m_G.

Now our starting point is to consider all words that can be derived from S in m_G steps. (We will loose in this way all terminal words that can be derived in less than m_G steps from G but this is a finite set and, as we will see, easy to handle.)

Then we will divide the words in this set into (not necessarily disjoint) subsets in each of which we can view all derivations going "according to the same clock" or, in more mathematical terms, conforming to the same (ultimately periodic) spectrum.

Here is thus our basic construction.

CONSTRUCTION. Let $0 \leq k < m_G$ and let $Ax(G,k) = \{w \in A_G^+: S \overset{m_G}{\underset{G}{\Rightarrow}} w$ and, for all a in Min(w), $m_G + k$ is in Spec(G,a)$\}$. If $Ax(G,k) \neq \phi$, then, for all w in Ax(G,k) define a 0L system $G(k,w) = \langle \Sigma_{k,w}, R_{k,w}, w \rangle$ as follows:

(i) $\Sigma_{k,w} = \{a \in A_G: m_G + k$ is in Spec(G,a) and, for some $\ell \geq 0$, a is in Min(y) for some y such that $w \overset{\ell \cdot m_G}{\underset{G}{\Longrightarrow}} y\}$,

(ii) $a \to \alpha$ is in $R_{k,w}$ if and only if $a \overset{m_G}{\underset{G}{\Rightarrow}} \alpha$ with $a \in \Sigma_{k,w}$ and $\alpha \in \Sigma_{k,w}^+$.

Hence we have the following situation. If $w \in Ax(G,k)$ then the derivation in G(k,w) goes as follows:

$$w \overset{\text{1 step in } G(k,w)}{\Rightarrow} \ldots \overset{m_G \text{ steps in } G}{\Rightarrow} w_1 \overset{\text{1 step in } G(k,w)}{\Rightarrow} \ldots \overset{m_G \text{ steps in } G}{\Rightarrow} w_2 \Rightarrow \ldots$$

Now using the fact that all symbols appearing in words in L(G(k,w)) contain $m_G + k$ in their spectra we can squeeze the language from G(k,w) in the following way.

Define M(G(k,w)) by

$M(G(k,w)) = \{x \in \Delta^*:$ there exists y in L(G(k,w)) such that $y \overset{m_G+k}{\underset{G}{\Longrightarrow}} x\}$.

We shall now show that the union of the languages M(G(k,w)) over all $k < m_G$ and w in Ax(G,k) is identical (modulo a finite set) to L(G).

CLAIM 1. $L(G) = \{w \in \Delta^+: S \overset{l}{\underset{G}{\Rightarrow}} w$ for some $l < 2m_G\} \cup \bigcup_{k<m_G} \bigcup_{w \in Ax(G,k)} M(G(k,w))$.

Proof. Obviously the right side is included in the left side. Now let us assume that x is in L(G).

(a) If x can be derived in less than $2m_G$ steps, then x is in the first set in union.

(b) If x is derived in at least $2m_G$ steps, then let $D = (S, x_1, \ldots, x_{m_G}, \ldots, x_p = x)$ be a derivation in G where $p = l_p \cdot m_G + k_p$

for some $l_p \geq 2$ and $0 \leq k_p < m_G$.
For all l, $1 \leq l < l_p$ and a in $\mathrm{Min}(x_{l \cdot m_G})$, we have

(i) a is vital, since $a \overset{t}{\underset{G}{\Rightarrow}} \bar{x}$ for some word $\bar{x}$ in Δ^+, where $t = (l_p \cdot m_G + k_p) - l \cdot m_G \geq m_G$.

(ii) $m_G + k_p$ is in $\mathrm{Spec}(G,a)$, since $(l_p - l)m_G + k_p$ is in $\mathrm{Spec}(G,a)$ and $\mathrm{Spec}(G,a)$ is an ultimately periodic set with period m_G and threshold smaller than m_G.

Therefore x is in $M(G(k_p, x_{m_G}))$ and hence x is in our union of languages on the right side.

Therefore Claim 1 holds.

Now the reader should note that we have already proven a quite significant result: each EP0L language is the result of a finite substitution on a 0L language!

However we want to replace the finite substitution mapping by a coding. To this aim we shall prove now that each component language in the "union formula for L(G)" as given in the statement of Claim 1 is a finite union of codings of 0L languages.

CLAIM 2. Assume that $\mathrm{Ax}(G,k) \neq \phi$ and let w be in $\mathrm{Ax}(G,k)$. Then there exist 0L systems $H_1, \ldots, H_f$ and a coding h such that $M(G(k,w)) = \bigcup_{i=1}^{f} h(L(H_i))$.

Proof. Let $w = b_1 \ldots b_t$ where b_i is in A_G for $1 \leq i \leq t$. For all a in $\Sigma_{k,w}$ let $U(a,k) = \{x \in \Delta^+ : a \overset{m_G+k}{\underset{G}{\Longrightarrow}} x\} = \{\alpha_{a,k,1}, \alpha_{a,k,2}, \ldots, \alpha_{a,k,U(a,k)}\}$, say, and $\bar{\Sigma}_{k,w} = \{[a,b], [\bar{a},\bar{b}] : a \in \Sigma_{k,w} \text{ and } b \in \Delta\}$.

Let $W(w) = \{[b_1,c_{11}][b_1,c_{12}] \ldots [\overline{b_1,c_{1r_1}}][b_2,c_{21}] \ldots [\overline{b_2,c_{2r_2}}] \ldots [b_t,c_{t1}] \ldots [\overline{b_t,c_{tr_t}}] : c_{j1} \ldots c_{jr_j} \in U(b_j,k) \text{ for } 1 \leq j \leq t\}$.

Let $\bar{R}_{k,w} = \{[a,b] \to \lambda : a \in \Sigma_{k,w}, b \in \Delta\} \cup \{[\overline{a,b}] \to [c_1,d_{11}][c_1,d_{12}] \ldots [\overline{c_1,d_{1v_1}}] \ldots [c_s,d_{s1}] \ldots [\overline{c_s,d_{sv_s}}] : b \in \Delta, a \to c_1 \ldots c_s \text{ is in } R_{k,w} \text{ and } d_{j1} \ldots d_{jv_j} \in U(c_j,k) \text{ for } 1 \leq j \leq s\}$.

Let, for every z in $W(w)$, $G(k,w,z)$ be the 0L system $\langle \bar{\Sigma}_{k,w}, \bar{R}_{k,w}, z \rangle$ and let h be a coding from $\bar{\Sigma}_{k,w}$ into Δ such that $h([a,b]) = h([\overline{a,b}]) = b$.

We leave to the reader the obvious, but tedious proof of the fact that

$$M(G(k,w)) = \bigcup_{z \in W(w)} h(L(G(k,w,z))).$$

Thus Claim 2 holds.

Now we state two obvious results.

(I) If K is a finite language, then there exist a 0L system G and a coding h such that $K = h(L(G))$.

(II) If $H_1,\ldots,H_f$ are 0L systems, $h_1,\ldots,h_f$ are codings and $K = \bigcup_{i=1}^{f} h_i(L(H_i))$, then there exist a 0L system G and a coding $\overline{h}$ such that $K = \overline{h}(L(G))$.

Thus, we can collect together all the component languages of G by means of one 0L system and one coding. This ends the proof of the equations E0L = C0L = H0L.

3. L FAMILIES

It is apparent on the basis of the notions considered in Chapter 1 that it is possible and natural within L systems theory to define quite a number of different language families. We mention [NRSS] as a typical paper along these lines. It is also clear that the pure families, i.e., families obtained by the exhaustive definition are mathematically somewhat awkward because of their weak closure properties. Undoubtedly one can say that E0L and ET0L are the basic and most thoroughly investigated L families. Because of a number of reasons (for instance, cf. [Sa2]), they are also very natural from the formal language theory point of view.

In this chapter we consider some recently introduced L families. Context-free languages of finite index (also referred to as derivation-bounded, quasirational, semilinear and superlinear languages) are quite widely studied, cf. [Sa1]. The notion of finite index has been extended to ET0L systems in the following way. Let us call a letter a in an ET0L system G *active* provided a derives according to G some word $\alpha \neq a$. Let A(G) be the set of all active letters of G and let k be a positive integer. We say that G is of *index* k iff every word x in L(G) has a derivation in which every word has at most k occurrences of active letters, G is of *finite index* iff it is of index k for some k. An ET0L language is of index k (resp. of finite index) iff it is generated by an ET0L system of index k (resp. of finite index). The corresponding language families are denoted by $ET0L_{FIN(k)}$ and $ET0L_{FIN}$. The notations $E0L_{FIN}$, $EDT0L_{FIN(k)}$, etc., are

used in the same way.

It is easy to see that, for a given ETOL (resp. EDTOL) system, one can construct an equivalent system G where A(G) equals the set of nonterminals of G. (Systems of the latter type are said to be in active normal form.) Also it is clear that many constructions, such as the well-known construction of replacing an ETOL system by an equivalent EPTOL system, preserve the index. Also the following transition from nondeterministic to deterministic systems can be made.

THEOREM 3.1. *For any given ETOL system of index* k, *one can find an equivalent EPDTOL system of index k in active normal form.*

The proof of Theorem 3.1 is carried out by replacing each nonterminal A of the original system with k "descendants" $A_1,\ldots,A_k$. Because of the assumption concerning the index, the tables for the descendants can be chosen deterministic. The only nondeterminism required is in picking up the right descendants but this can be affected by introducing different tables. (Note that the same construction does not work for EOL and EDOL systems.) As an immediate corollary we get the following result.

THEOREM 3.2. $\text{ETOL}_{\text{FIN(k)}} = \text{EPDTOL}_{\text{FIN(k)}}$ *and* $\text{ETOL}_{\text{FIN}} = \text{EPDTOL}_{\text{FIN}}$.

Once can also consider the subclass of ETOL systems of finite index consisting of systems in which every derivation leading to a terminal word satisfies the finite index restriction. Formally, an ETOL system G is of *uncontrolled index* k iff whenever x is a word belonging to some derivation of a word y in L(G), then x contains at most k occurrences of active letters. The notion of an uncontrolled finite index, as well as the notations $\text{ETOL}_{\text{UFIN(k)}}$, $\text{ETOL}_{\text{UFIN}}$, etc., are defined similarly as before. Note the analogy between ETOL systems of finite index and uncontrolled finite index on one hand, and context-free grammars of finite index and ultralinear grammars on the other hand, cf. [Sa1]. It is well-known that there are context-free languages of finite index which are not ultralinear, for instance, the language $(\{a^n b^n | n\geq 0\}c)^*$. However, in case of ETOL systems the situation is quite the opposite, and one can obtain the following rather surprising result.

THEOREM 3.3. *For any given* ETOL *system of index* k, *one can find an equivalent* EPDTOL *system of uncontrolled index* k (*and in active normal form*).

Proof. By Theorem 3.1 we may assume that the given ETOL system G is, in fact, an EPDTOL system and in active normal form. Let G contain m nonterminals. We construct an equivalent EPDTOL system H of uncontrolled index k as follows. The nonterminals of H are of form $A(i_1,\ldots,i_m)$, where A is a nonterminal of G, $0 \le i_j \le k$, and $i_1+\ldots+i_m \le k$. The vector $(i_1,\ldots,i_m)$ keeps track of the numbers of occurrences of nonterminals in a derivation according to G, and whenever there is an overflow, a garbage symbol is introduced. Thus, the initial symbol of H is $S(1,0,\ldots,0)$, where S is the initial symbol of G. A production $A \to \alpha_1 A_1 \alpha_2 \ldots \alpha_t A_t \alpha_{t+1}$ (A's are nonterminals) in a table T of G is changed into the set of productions (one for each vector)

$$A(i_1,\ldots,i_m) \to \alpha_1 A_1 (i_1,\ldots,i_m)^T \alpha_2 \ldots \alpha_t A_t (i_1,\ldots,i_m)^T \alpha_{t+1},$$

where $(i_1,\ldots,i_m)^T$ is the m-dimensional Parikh vector of nonterminals obtained from the vector $(i_1,\ldots,i_m)$ by the use of table T. (Note that $(i_1,\ldots,i_m)^T$ is unique because T is deterministic.) If the sum of components in $(i_1,\ldots,i_m)^T$ is greater than k, then instead of the production indicated we let $A(i_1,\ldots,i_m)$ go into a garbage symbol.

Combining Theorems 3.2 and 3.3, we get the equations

$$\mathrm{ETOL}_{\mathrm{FIN}(k)} = \mathrm{ETOL}_{\mathrm{UFIN}(k)} = \mathrm{EPDTOL}_{\mathrm{FIN}(k)} = \mathrm{EPDTOL}_{\mathrm{UFIN}(k)},$$

$$\mathrm{ETOL}_{\mathrm{FIN}} = \mathrm{ETOL}_{\mathrm{UFIN}} = \mathrm{EPDTOL}_{\mathrm{FIN}} = \mathrm{EPDTOL}_{\mathrm{UFIN}}.$$

The above results are from [RV1], where the study of the family $\mathrm{ETOL}_{\mathrm{FIN}}$ was initiated. It turns out that under the finite index restriction some hierarchies of language families collapse into one family, namely, $\mathrm{ETOL}_{\mathrm{FIN}}$. In particular, if the finite index restriction is introduced for context-free programmed grammars (i.e., for the families P_{ac} and P^{λ}_{ac} in the notation of [Sa1]), the resulting family equals $\mathrm{ETOL}_{\mathrm{FIN}}$. That the resulting family contains $\mathrm{ETOL}_{\mathrm{FIN}}$ is obvious. The reverse inclusion is shown by constructing an ETOL system H of index k for a language generated by a context-free programmed grammar G of index k. The construction is possible

since, because of the finite index restriction, we can carry complete information about the nonterminals occurring (even the number of occurrences and their order), in a derivation step according to G, attached to nonterminals of H. There are only finitely many different derivation steps (i.e., where not both the rule and the entire sequence of nonterminals are the same), and each of them is simulated by one table of H. The construction resembles the one in the proof of Theorem 3.1 but is a little more complicated because we have to take care of both the success and failure fields of the original rule in G. The formal details can be found in [RV2].

It can now be inferred that if L is any family lying between ETOL and the family of context-free programmed languages, then $L_{FIN} = ETOL_{FIN}$. (Examples of such families L are given in [Pe].) The only thing one has to show in each particular case is that the simulation by context-free programmed grammars of the generative devices yielding the family L preserves the finiteness of the index. This is straightforward in most cases. One case is considerd in detail in [R1]. Thus one can say that a number of quite different approaches give rise to the family $ETOL_{FIN}$.

However, $ETOL_{FIN}$ contains properly EOL_{FIN}. For instance, the language $\{a^n b^m a^n \mid m \geq n \geq 1\}$ is in $ETOL_{FIN}$ but is not even EOL. The following results are established in [RV1].

THEOREM 3.4. *The family* $ETOL_{FIN(1)}$ *equals the family of linear languages. Moreover,*

$$ETOL_{FIN(1)} \subset\neq ETOL_{FIN(2)} \subset\neq \cdots \subset\neq ETOL_{FIN}.$$

$ETOL_{FIN}$ *is a substitution closed full* AFL. *There is an algorithm for deciding whether an arbitrary given* ETOL *system* G *is of uncontrolled finite index but no algorithm for deciding whether* G *is of finite index.*

suits concerning L families. A characterization of the family of ETOL languages, as well as its subfamilies EOL, EDTOL, EDOL, in terms of context-free programmed grammars of restricted types is given in [RV2]. It is also shown in the quoted paper that ETOL systems added with an "exactly one occurrence checking" mechanism (i.e., certain tables P can be applied only to strings having exactly one occurrence of a specific letter A_p) yield the full generative capacity of (λ-free) context-free programmed

grammars. Thus, ETOL systems with an additional mechanism yield context-free programmed languages. One can in a sense reverse this way of thinking and study context-free programmed grammars with an additional mechanism suggested by ETOL systems. This is done in [RS3] and leads to a variation of programmed grammars, where control is imposed over sets of productions ("tables") rather than over single productions. Such programmed grammars generate exactly the family of context-sensitive languages, the result being true even if one considers control of one of the two very simple types which correspond to programmed grammars with empty failure fields and those with unconditional transfer.

Some new definitional mechanisms ("squeezing" mechanisms) are considered in [RS4]. They can be applied to L systems of the pure type, such as OL, DOL, PDOL, TOL, DTOL systems. In the *production-universal* (resp. *production-existential*) definition of the language, only those words are accepted which possess a derivation such that at the last step every production (resp. at least one production) applied belongs to a specified set of "good" productions. Both of these mechanisms are special cases of the more general *production- subset* definition: only those words are accepted which possess a derivation such that at the last step the set of productions applied equals some set in a specified finite collection of "good" sets. *Letter-universal* and *letter-existential* mechanisms are defined similarly by specifying the set of "good" letters whose presence is requested in the last word. (Thus, the letter-universal mechanism coincides with the E-mechanism.) From a biological point of view, one is led naturally to these definitions if one wants to consider only certain stages in the development. From a generative capacity point of view, it turns out that if the underlying L structure is strong enough (such as a TOL structure) then all of these definitional mechanisms are of equal power, whereas uncomparability and strict inclusion results are obtained for weaker underlying L structures.

We mention briefly two other squeezing mechanisms recently introduced: time delay and fragmentation introduced in [W] and [RRS], respectively. With OL as the underlying L structure, we get using these two mechanisms the families VOL and JOL. In a VOL system letters carry natural numbers as delay indicators. In one derivation step, each delay indicator is reduced by 1, and new development can take place after the indicator has become 0. The language of the system is obtained by stripping off the

delay indicators. Thus the VOL system with the axiom a_0 and production $a_0 \to a_0a_1$ yields at first the sequence

$$a_0, a_0a_1, a_0a_1a_0, a_0a_1a_0a_0a_1, a_0a_1a_0a_0a_1a_0a_1a_0, \cdots$$

Hence, the language of the system equals $\{a^q | q \text{ Fibonacci}\}$. In a JOL system, right sides of the productions may contain occurrences of a special letter q which induces a cut in the resulting word. The derivation may then continue from one of the parts thus obtained. For instance, using the productions $a \to aqa$ and $b \to ab$, we get from the word bab in one derivation step the words aba and aab. These two mechanisms can of course also be used in connection with underlying L structures other than OL. (A feature common to both mechanisms is the method by which the abbreviations V and J were chosen.)

It is well known that the celebrated LBA-problem can also be expressed in terms of L systems. For various ways of doing this, the reader is referred to [Vi2]. In most cases, one has to compare the generative capacity of some nondeterministic and deterministic L systems with interactions. The most interesting result along these lines is in [Vi2], where comparison is made between two deterministic L systems (although nondeterminism comes into the picture in form of table systems which are always in a sense nondeterministic). More specifically, it is shown in [Vi2] (which continues the work begun in [Vi1]) that the family of context-sensitive languages equals EP1L (the proof of this result is based on the left context-sensitive normal form) and, furthermore, using the previous result, that the family of context-sensitive languages equals also $EPDT_2 1L$ (where T_2 refers to table systems with only two tables). In [Vi1], it is shown that the family of deterministic context-sensitive languages equals EPD2L. Therefore, the LBA problem amounts to solving the problem of whether or not a tradeoff is possible between one-sided context with two tables and two-sided context with one table for λ-free deterministic L systems using nonterminals.

4. L GENERALIZATIONS

A number of attempts have been made towards a uniform framework for L

systems. Such a framework usually presents also a generalization of the individual systems considered. The basic idea underlying L systems is that of iterated substitution. The corresponding formal model is a K-iteration grammar which has turned out to be a very useful general framework for discussing L systems.

Let K be a family of languages which is closed under alphabetical variance and contains a language containing a nonempty word. By a K-substitution over an alphabet Σ we mean a substitution σ defined on Σ such that, for each $a \in \Sigma$, $\sigma(a)$ is a language over Σ belonging to K. A *K-iteration grammar* is a quadruple $G = (\Sigma,P,S,\Delta)$, where Σ and Δ are alphabets, $\Delta \subseteq \Sigma$, $S \in \Sigma - \Delta$, and $P = \{\sigma_1,\ldots,\sigma_n\}$ is a finite set of K-substitutions over Σ. We write $x \overset{k}{\Rightarrow} y$, for x and y in Σ^*, iff there are $\sigma_{i_1},\ldots,\sigma_{i_k}$ in P such that

$$y \in \sigma_{i_k} \ldots \sigma_{i_1}(x).$$

The language generated by G is defined by

$$L(G) = \{w \in \Delta^* \mid S \overset{k}{\Rightarrow} x, \quad \text{for some } k\}.$$

Languages of this form are called *hyper-algebraic* over K, and the family consisting of them the *hyper-algebraic extension* of K, in symbols, H(K). For $m \geq 1$, $H_m(K)$ denotes the family of languages generated by K-iteration grammars such that the cardinality of P does not exceed m.

Following [As1], we call a family of languages K a *prequasoid* iff K is closed under finite substitution and intersection with regular languages. A *quasoid* is a pre-quasoid containing at least one infinite language. It is easy to see (for details, cf. [AS2]) that every pre-quasoid (resp. quasoid) contains all finite (resp. regular) languages and that the family of finite languages is the only pre-quasoid which is not a quasoid.

By definition, a pre-quasoid K is a *hyper-AFL* iff H(K) = K, i.e., K is hyper-algebraically closed. (It is shown in [As2] that this definition is equivalent to the definition given in [RS1]. Thus, ETOL is the smallest hyper-AFL). The following two theorems are from [As2].

THEOREM 4.1. *Assume that* K *contains a language* {a}, *where a is a letter. Then* $H(K) = H_m(K) = H_2(K)$ *for each* $m \geq 2$.

Proof. Since K is closed under alphabetical variance, it contains all languages consisting of a word of length 1. Clearly, it suffices to prove the inclusion $H_m(K) \subseteq H_2(K)$, for an arbitrary $m \geq 2$. (This implies $H(K) \subseteq H_2(K)$, and the reverse inclusions are obvious.) Thus, consider a K-iteration grammar G with $m \geq 3$ substitutions $\sigma_1,\ldots,\sigma_m$. We simulate G by a K-iteration grammar H with two substitutions τ_1 and τ_2, defined as follows. To get the alphabet of H, we add to the alphabet of G a garbage symbol F and "descendants" $a_0,a_1,\ldots a_m$, for each letter a in the alphabet Σ of G. The terminal alphabet of H equals that of G, and the initial letter of H is $h_0(S)$, where S is the initial letter of G and h is the homomorphism defined on Σ which sends every letter a to the descendant a_0 (i.e., $h_0(S) = S_0$). The K-substitutions τ_1 and τ_2 are defined by

$$\tau_1(a_i) = a_{i+1}, \quad \tau_2(a_0) = \{a\}, \quad \tau_2(a_j) = h_0(\sigma_j(a)),$$

where $a \in \Sigma$, $0 \leq i \leq m-1$, $1 \leq j \leq m$, and τ_1 and τ_2 assume the value $\{F\}$ for all other letters. □

As a corollary of Theorem 4.1 we get the known result that every ETOL language is generated by an ETOL system with two tables. The proof of Theorem 4.1 uses the idea of "cyclic tables" familiar from ETOL systems.

THEOREM 4.2. *For every pre-quasoid* K, *the family* H(K) *is a hyper-AFL.* (*Moreover,* H(K) *contains* ETOL *and is the smallest hyper-AFL containing* K.)

The essential point in Theorem 4.2 is that H(K) is hyper-algebraically closed, i.e., H(H(K)) = H(K). This is an extension of the result of Christensen, [RS1], that ETOL is hyper-algebraically closed.

THEOREM 4.3. *There exists an infinite chain* K_i *of hyper* AFL's *strictly in between the families of contex-free and context-sensitive languages*:

$$CF \subsetneq K_0 \subsetneq K_1 \subsetneq K_2 \subsetneq \ldots \subsetneq K_i \subsetneq \ldots \subsetneq CS.$$

We outline the proof, the details of which can be found in [AL]. We consider ETOL systems with a control language on the use of tables. The family of languages generated by ETOL systems with control languages belonging to

the family K is denoted by (K)ETOL. Define now

$$K_0 = \text{ETOL}, \quad K_i = (K_{i-1})\text{ETOL} \qquad (i>0).$$

One can show that each (K_{i-1})ETOL is hyper-algebraically closed, whence it follows that each K_i is a hyper-AFL. Clearly, for all i, $K_i \subseteq K_{i+1}$. The strictness of the inclusion is seen by considering functions

$$f_0(x) = 2^x, \quad f_i(x) = 2^{f_{i-1}(x)} \qquad (i>0)$$

and languages

$$L_i = \{a^{f_i(x)} \mid x \geq 0\} \qquad (i\geq 0).$$

By induction on i, it is immedaitely seen that $L_i \in K_i$. That $L_i \notin K_{i-1}$ (i>0) follows because the growth rate of $f_i(x)$ exceeds the rate possible for languages in K_{i-1}. This again is seen inductively by noting first that the growth rate in ETOL is at most exponential. In the inductive step, it is useful to note that we may restrict attention to λ-free ETOL systems.

To complete the proof, one shows by a complexity argument that K_i is properly contained in CS. □

Especially interesting is the limit family $K_\omega = \bigcup_i K_i$. K_ω is a hyper-AFL which is not principal. Furthermore, it is the smallest hyper-AFL satisfying the "fixed-point" condition (K)ETOL = K.

K-iteration grammars with a control language (in the same sense as for ETOL systems) have been investigated in [As2]. Given K and the family Γ of control languages, we denote by H(K,Γ) the family of languages generated by K-iteration grammars with a control language in Γ. Under certain quite general assumptions concerning Γ, results analogous to Theorems 4.1 and 4.2 are valid for the family H(K,Γ).

We mention, finally, the work begun in [R2] concerning selective substitution grammars. This is a general model which yields K-iteration grammars (and, hence, various L systems) as well as the main types of grammars met in sequential rewriting as special cases.

5. L PARSING

We consider parsing of EOL and ETOL languages. It was shown in [vL2] that the membership problem for ETOL is NP-complete. In fact, ETOL systems constitute perhaps the simplest grammatical device for generating the language SAT_3 crucial for NP-completeness. The following ETOL system G generates SAT_3 (consisting of satisfiable formulas of propositional calculus in 3-conjunctive normal form with unary notation for variables.) The alphabet of G consists of the letters S (initial), $\vee$ (disjunction), $\neg$ (negation), t ("true"), f ("false"), (,) (parentheses), F (garbage), 1 (variables). We have the following table

$$[S \to (\alpha \vee \beta \vee \gamma)S,\ S \to (\alpha \vee \beta \vee \gamma)]$$

where (α,β,γ) ranges over all combinations of $(t,\neg t,f,\neg f)$ which do not consist entirely of $\neg$t's and f's, as well as the table

$$[S \to F,\ t \to 1t,\ f \to 1f,\ t \to 1]$$

and the table obtained from this by replacing $t \to 1$ with $f \to 1$. (The letters not listed in the tables go into themselves.)

The same system shows also that the membership problem for TOL languages is NP-complete. As regards DTOL systems, the problem is still open, although some results (in [Ha]) make it very likely that membership for DTOL is polynomial time.

The fastest known algorithm for recognizing EOL languages is the one given in [vL3]. The algorithm works in time $O(n^{3,81})$, and is based on the construction of appropriate data-structures such that Valiant's fast algorithm for computing the transitive closure of matrices over non-associative domains can be applied.

Questions concerning parsing are closely related to the problem of finding suitable machine models for L systems. Combining the idea of a checking stack automaton and his earlier pre-set pushdown automaton, van Leeuwen, [vL4], has defined the notion of an *augmented checking stack aotumaton* and shown that the family of accepted languages equals ETOL. As a consequence of this result, it follows that ordinary checking stack languages are in ETOL.

An augmented checking stack automaton uses both a checking stack and a synchronously operating pushdown tape. The input tape is one-way, there is a finite control as usual, and the machine is in general nondeterministic. Initially, the machine writes information on top of the checking stack. (This information cannot be altered later on but can only be used for "checking".) After that, the pushdown tape becomes operational in the ordinary way except that there is a double pointer pointing to both pushdown and checking stack. The checking stack interrupts the computation if the pointer reaches its top (beyond which it is not allowed to go). Thus, one step in a computation of the machine begins by noting the current input symbol, the current state and the symbols pointed at in the storage (i.e., the top symbol of the pushdown tape and the contents of the opposite square on the checking stack), and results in (perhaps in a nondeterministic fashion) moving the input head 0 or 1 squares to the right, changing the internal state, and popping or pushing a symbol on the pushdown, together with an adjustment of the double pointer. The latter never moves beyond the area allocated for the checking stack. Clearly, both the pre-set pushdown automaton and the ordinary checking stack automaton are degenerate cases of this model.

6. L GROWTH

We now turn into the discussion of some recent results, all in the area of informationless L systems, concerning growth of word length. The theory of growth functions has been extensively discussed in [HR], [RS1], [RS2], [LR], and [HV], the last-mentioned reference being a recently published survey article on this area. The results discussed below are from [So1], [So2], [So3], [Sa3] and [SaSo]. We begin with the discussion of some undecidability results. In fact, one can claim that some of the problems listed below, for instance Problem (4), are the most "innocent looking" problems concerning L systems which have been shown to be undecidable. Of course, there are even more innocent looking problems whose decidability status is open.

Consider functions f mapping the set V^* of all words over a finite alphabet $V = \{a_1,\ldots,a_k\}$ into the set Z of all integers. Such a function is termed *Z-rational* (resp. *N-rational*) iff there is a row vector π, a

column vector η, and square matrices $M_1,\ldots,M_k$, all of the same dimension m and with integral (resp. nonnegative integral) entries, such that for any word $x = a_{i_1}\ldots a_{i_t}$,

$$f(x) = \pi M_{i_1}\ldots M_{i_t}\eta.$$

(If x equals the empty word λ, this matrix representation reduces to $f(\lambda) = \pi\eta$.) An N-rational function is termed DT0L iff all entries in η equal 1. Finally, a DT0L function is termed PDT0L iff every row in each of the matrices $M_1,\ldots,M_k$ contains at least one element greater than zero. In the special case of a one-letter alphabet, k = 1, DT0L functions (resp. PDT0L functions) are referred to as D0L (resp. PD0L) functions. In this case, the argument is written simply n, instead of a_1^n.

From the L systems point of view, these definitions can be interpreted as follows. Consider a DT0L system with k tables and m letters in the alphabet. (Thus, the alphabet V will be the alphabet of the tables, whereas the dimension of the matrices gives the cardinality of the alphabet of the system itself.) The matrix M_i is the growth matrix associated with the table a_i, and π indicates the distribution of the letters in the axiom. The function value $f(a_{i_1}\ldots a_{i_t})$ gives the length of the filament resulting by applying the sequence of tables $a_{i_1}\ldots a_{i_t}$ to the axiom. PDT0L functions correspond in the same way to PDT0L systems, and N-rational functions to HDT0L systems.

N-rational functions over a one-letter alphabet give the length sequence of an HD0L system. In case of one-letter alphabet, we often speak of sequences instead of functions. In this chapter, we always mean length sequences rather than word sequences.

The following theorem which strengthens analogous results in [E] is our basic tool for establishing undecidability.

THEOREM 6.1. *Consider an alphabet* V = {a,b} *consisting of two letters. The following problems are undecidable for* Z*-rational functions* f *defined on* V^*:

(i) *Does* f *assume the value* 0 *at least once?*

(ii) *Does* f *assume the value* 0 *infinitely many times?*

(iii) *Are all values of* f *nonnegative?*

(iv) *Does there exist a* t *such that* f(x) *is nonnegative for all words* x *with length greater than* t?

Proof. We show first that if we could decide (i), we would be solving Hilbert's Tenth Problem. For this purpose we construct, for a given integer polynomial P with u variables, a Z-rational function r satisfying the identity

$$r(a^{n_1}ba^{n_2}b\ldots a^{n_u}b) = P(n_1,\ldots,n_u) \qquad (n_i \geq 0).$$

For $i = 1,\ldots,u$, let r_i be the N-rational function defined by the following (i+1)-dimensional vectors and matrices:

$$\pi_i = (10\ldots0), \quad \eta_i = (0\ldots01)^T,$$

$$M_i(a) = \begin{pmatrix} 1 & & & \\ & 1 & 0 & \\ & & \ddots & 0 \\ & & & 1 \\ \hline & & & \begin{matrix} 1 & 1 \\ 0 & 1 \end{matrix} \end{pmatrix}, \quad M_i(b) = \begin{pmatrix} 0 & 1 & & \\ & 0 & 1 & \\ & & \ddots & 0 \\ & & & 1 \\ \hline & & & \begin{matrix} 0 & 0 \\ 0 & 1 \end{matrix} \end{pmatrix}.$$

It is easy to see that r_i satisfies

$$r_i(a^{n_1}b\ldots a^{n_u}b) = n_i.$$

The function r can now be constructed from the functions r_i by the rational operations used in the definition of the polynomial P.

Let now L_u be the complement of the (regular) language $a^*(ba^*)^{u-1}$, and s the (Z-rational) characteristic function of L_u. Denote Hadamard product by 0. Then the Z-rational function

$$s_1 = r \,0\, r + s$$

assumes the value 0 iff $P(n_1,\ldots,n_u) = 0$ has a solution in nonnegative integers n_i.

Because all values of the function $s_1 - 1$ are nonnegative iff s_1 does not assume the value 0, we see that also (iii) is undecidable. The undecidability of (ii) and (iv) is seen by an easy modification of the argument above. □

The next two theorems establish useful interconnections between Z-rational

and DT0L functions. The theorems are obtained by the techniques of merging and dominant terms considered in my article in [LR]. For the statement of the latter theorem, we need the operator ODD, defined for words of odd length (over any alphabet) as follows:

$$ODD(b_1b_2\ldots b_{2n+1}) = b_1b_3\ldots b_{2n+1}.$$

For words x over even length, ODD(x) is undefined.

THEOREM 6.2. *For any* Z*-rational function* f, *there exists a number* u_0 *such that, for all integers* $u \geq u_0$, *the function* $f_1(x) = u^{lg(x)+1} + f(x)$ *is a* PDT0L *function.*

THEOREM 6.3. *For any* Z*-rational function* f, *there is a number* u_0 *such that, for any integer* $u \geq u_0$, *the function* g *defined in the following way is a* DT0L *function:*

$$g(x) = u^{n+1} \quad \textit{for} \quad lg(x) = 2n,$$

$$g(x) = u^{n+1} + f(ODD(x)) \quad \textit{for} \quad lg(x) = 2n+1.$$

We now list some decision problems for DT0L and D0L functions. In the statement of the problems, f(x) (resp. $f_P(x)$) is a DT0L (resp. PDT0L) function over a two-letter alphabet (i.e., the corresponding table system consists of *two tables only*). Furthermore, g(n) is a PD0L function, x is a word over $V = \{a_1,a_2\}$ and b ranges over $\{a_1,a_2\}$.

(1) (Comparison between PD0L and PDT0L growth) Given g and f_P, decide whether $g(n) \leq f_P(x)$ holds for all n and x with lg(x) = n.

(2) (Monotonicity of DT0L growth) Given f, decide whether $f(x) \leq f(xb)$ holds for all x and b.

(3) (Existence of equal size between PD0L and PDT0L growth) Given g and f_P, decide whether there exist an n and x with lg(x) = n such that $g(n) = f_P(x)$.

(4) (Constant level in DT0L growth) Given f satisfying for all x and b $f(x) \leq f(xb)$, decide whether there exist x and b such that f(x) = f(xb).

The remaining problems (5)-(8) are modifications of (1)-(4) respectively.

(5) (Ultimate comparison between PD0L and PDT0L growth) Given g and f_p, decide whether there exists a u such that $g(n) \leq f_p(x)$ holds for all $n \geq u$ and all x with $lg(x) = n$.

(6) (Ultimate monotonicity of DT0L growth) Given f, decide whether there exist a u such that $f(x) \leq f(xb)$ holds for all x and b with $lg(x) \geq u$.

(7) (Equal size between PD0L and PDT0L growth infinitely often) Given g and f_p, decide whether $g(n) = f_p(x)$ holds for infinitely many pairs (n,x) with $lg(x) = n$.

(8) (Constant level in DT0L growth infinitely often) Given f satisfying for all x and b $f(x) \leq f(xb)$, decide whether $f(x) = f(xb)$ holds for infinitely many pairs (x,b).

The intuitive meaning from the L systems point of view is indicated in connection with each problem. For instance, as regards problem (1), this can be expanded as follows. We have modeled two developmental processes, one by a PD0L system and the other by a PDT0L system with two tables. We want to know whether it is possible to use the tables in such a way that, at some time instant, the filament obtained in the latter process is smaller than the one obtained in the former process.

THEOREM 6.4. *All of the Problems* (1)-(8) *are undecidable.*

Proof. The undecidability of (i) in Theorem 6.1 gives by Theorem 6.2 (resp. Theorem 6.3) the undecidability of (3) (resp. (4)). Similarly, (ii) is utilized to prove the undecidability of (7) and (8), (iii) to prove that of (1) and (2), and (iv) to prove that of (5) and (6). □

We consider next problems whose decidability has recently been established. The decision methods also give a complete characterization of PD0L growth functions within the class of D0L growth functions, of D0L growth functions within the class of HD0L growth functions, as well as of HD0L growth functions within the class of Z-rational functions. Hence, we can always find out whether or not a given function in one of these classes belongs to some other class. We get also a complete solution for the D0L synthesis problem.

The three characterization results referred to above are contained in the next three theorems. For proofs, we refer to [So1], [So2], [So3], or to [SaSo]. Since we are dealing with one-letter alphabet, we prefer to

state the theorems for sequences r_n determined by values of functions: $r_n = r(n)$.

THEOREM 6.5. *A sequence of integers* r_n *is a* PD0L *sequence iff* $r_0 > 0$ *and the sequence* $s_n = r_{n+1} - r_n$ *is* N-*rational.*

THEOREM 6.6. *Let* r_n *be a* N-*rational sequence such that* $r_n \neq 0$, *for every* n, *and the quotient* r_{n+1}/r_n *remains bounded. Then* r_n *is a* D0L *sequence.*

THEOREM 6.7. *Let* r_n *be a sequence of integers satisfying the following two conditions:*

(i) $r_n \geq 0$ *for every* n,

(ii) *there are numbers* m *and* p *such that whenever* $0 \leq j \leq p-1$ *then the numbers* $s_n^{(j)} = r_{m+j+np}$ *have an expression* $s_n^{(j)} = P(n)\alpha^n + \Sigma_i P_i(n)\alpha_i^n$, *where* $\alpha \geq 0$, $\alpha > \max_i |\alpha_i|$, *and* P *and* P_i *are polynomials such that* P *is not identically* 0. (*For different values of* j, *the numbers* α, α_i *as well as the polynomials* P, P_i *may be different.*)
Then r_n *is an* N-*rational sequence.*

That Theorems 6.6 and 6.7 give a necessary and sufficient condition for D0L-ness and N-rationality, respectively, follows because also their converses hold. Clearly, for every D0L sequence r_n (which does not become ultimately 0), the quotient r_{n+1}/r_n remains bounded. On the other hand, by Berstel's Theorem (cf. [E]), every N-rational sequence satisfies conditions (i) and (ii) of Theorem 6.7.

According to Theorem 6.7, the condition characteristic for N-rationality of a given Z-rational sequence r_n is that the poles of its generating function (if there are any) are of the form $\rho\xi$, where $\rho > 0$ and ξ is a root of unity. By considering if necessary decompositions of the given sequence r_n, to test the N-rationality of a sequence, it suffices to be able to test the validity of the following conditions:

(i) the generating function is a polynomial or has only one pole with minimum modulus (and that pole is positive),

(2) $r_n \geq 0$ for every n.

As shown in [So2], these conditions are decidable. It is shown in [So1] how to decide the condition for D0L-ness stated in Theorem 6.6. Hence, Theorems 6.5-6.7 imply the following theorem.

THEOREM 6.8. *It is decidable whether or not a given* Z*-rational sequence is* N*-rational, a* D0L *sequence, or a* PD0L *sequence.*

As we already pointed out, according to Theorem 6.7 the condition characterizing N-rational sequences among Z-rational ones deals with the poles of minimal modulus of the generating function. Using this criterion and the fact that (3+4i)/5 is not a root of unity, we see that the following Z-rational sequence (consisting of positive terms) is not N-rational:

$$r_n = n \cdot 5^n + (3+4i)^n + (3-4i)^n.$$

Combining this with Theorem 6.5, we can easily construct strictly growing D0L sequences which are not PD0L sequences. (The existence of such sequences was an open problem for a long time.) For instance, the sequence s_n obtained from our sequence r_n above as follows:

$$s_{2n} = 10^n$$
$$s_{2n+1} = 10^n + r_n,$$

is such a D0L sequence. (It is an instance of merging, of which Theorem 6.3 gives a more general result.)

It is well-known that N-rational sequences a_n may consist of differently growing parts. For instance, a_{2n} may grow linearly and a_{2n+1} exponentially. (Since N-rational sequences coincide with HD0L sequences, we see this clearly by letting the homomorphism erase some letters.) Obviously, such differently growing parts are not possible in a D0L sequence. Theorem 6.6 shows that this is, in fact, the only difference between D0L and HD0L sequences. It is also a consequence of Theorem 6.6 that the quotient a_n/b_n of two D0L sequences is itself a D0L sequence, provided always $b_n \neq 0$ and b_n divides a_n. For a more detailed discussion regarding these matters, we refer to three quoted papers by Soittola or to [SaSo].

Although most of the problems concerning growth in informationless L systems have been solved, there are still some open problems which probably are very hard. (Note that there really is no mathematical theory, apart from some tricky examples and undecidability results, concerning DIL growth.) Using the terminology of Theorem 6.1, these problems can be stated simply as follows. Assume that the basic alphabet V consists of one letter only.

Are the problems (i), (iii), (iv) decidable? ((ii) is known to be decidable by a result of Berstel and Mignotte, cf. [LR].) A further discussion concerning these problems can be found in [LR]. We mention here only that an equivalent formulation for (i) (resp. (iii)) is the following: Given a DOL sequence r_n, decide whether or not there exists an i such that $r_i = r_{i+1}$ (resp. Decide whether or not a given DOL sequence r_n is monotonic). It would be extremely surprising if one of these problems would turn out to be undecidable.

We mention, finally, that in [Da] the following function $d_G(n)$ is introduced for DTOL systems G: $d_G(n)$ equals the number of distint words derivable in exactly n steps according to G. (Thus, for DOL systems G, $d_G(n)$ is identically 1.) The functions $d_G(n)$ might have some interconnections with growth functions although it is not clear whether or not they are even Z-rational.

7. L FORMS

The notion of a grammar form was introduced in [CG] as an attempt to define families of structurally similar grammars by means of one underlying grammar called a "grammar form", and an "interpretation" mechanism defining an infinite family of grammars related to the given grammar form. This notion of a grammar form and its interpretations has turned out to be a powerful tool for the study of grammatical properties of both theoretical and practical significance. Despite the youth of this area of language theory, much promising work has already been done in the field.

The notion of a grammar form can be introduced for L systems as well. This seems to be well motivated because it will certainly aid to the understanding of the structure of "L grammars" (as regards problems such as what types of EOL systems suffice to generate all EOL languages). Furthermore, from a biological point of view, a family of related L systems can be interpreted as a "family" or "species" of organisms.

The study of L forms, i.e., grammar forms for L systems has been initiated in [MSW] which contains all the results mentioned in this chapter. The results concern EOL forms only. However, work concerning other types of forms, in particular ETOL forms, is in progress.

By definition, an EOL *form* F is an EOL system, $F = (\Sigma,P,S,\Delta)$. An EOL system $F' = (\Sigma',P',S',\Delta')$ is called an *interpretation* of F modulo μ,

symbolically $F' \lhd F(\mu)$ iff μ is a substitution defined on Σ such that the following conditions are satisfied:

(i) $\mu(A) \subseteq \Sigma' - \Delta'$ for each $A \in \Sigma - \Delta$,

(ii) $\mu(a) \subseteq \Delta'$ for each $a \in \Delta$,

(iii) $\mu(\alpha) \cap \mu(\beta) = \phi$ for any $\alpha \neq \beta$,

(iv) $P' \subseteq (P)$ (where $\mu(P) = \bigcup_{\alpha \to x \text{ in } P} \mu(\alpha) \to \mu(x)$),

(v) $S' \in \mu(S)$.

The family $Gr(F) = \{F' \mid F' \lhd F\}$ is referred to as the grammar family of F, and the family $La(F) = \{L(F') \mid F' \lhd F\}$ as the language family generated by F. Two EOL forms F_1 and F_2 are termed *equivalent* (resp. *strictly equivalent*) iff $La(F_1) = La(F_2)$ (resp. $Gr(F_1) = Gr(F_2)$).

For the readers familiar with the theory of (ordinary) grammar forms, we would like to point out that our definition of interpretation differs from the ordinary one with respect to terminals: according to (ii), terminals are interpreted by terminal letters rather than terminal words, and condition (iii) is extended to concern also terminals. In addition of our definition being more natural mathematically than the ordinary definition, it has also several advantages from the point of view of L systems which have been explained in [MSW]. Moreover, the main reason for the exceptional definition of interpretation of terminals in the ordinary theory of grammar forms (the obtained language families become semi-AFL's) is not an important issue in L systems theory.

For the EOL form

$$F_1 = (\{S,a\},\{S \to Sa, S \to a, a \to a\},S,\{a\}).$$

the language family $La(F_1)$ equals the family of regular languages, as shown in [MSW]. (In fact, we obtain only λ-free regular languages but, according to our earlier convention, we consider the equality of languages modulo λ.)

For the EOL form

$$F_2 = (\{S,a\},\{S \to SS, S \to S, S \to a, a \to S\},S,\{a\}),$$

the language family equals the family of all EOL languages. Note, however, that if F_1 and F_2 are viewed as EOL systems, they generate the same language.

Results from the theory of ordinary grammar forms carry over to the grammar families Gr(F). (For instance, it is decidable whether two EOL forms generate the same grammar family.) This is quite natural because the parallelism in derivations is not used at all in the definition of the family Gr(F). As regards language families La(F), the situation is quite different. The basic lemma concerning ordinary grammar forms, according to which $L(F) \subseteq L(F')$, provided for every production $A \to x$ in F there is a derivation $A \Rightarrow^* x$ according to F', is not valid for EOL forms, the reason being that because of the parallelism, the derivations $A \Rightarrow^* x$ should be of the same length and they should also not introduce terminal words at the intermediate steps. The following EOL forms F and F' constitute a counter example (we list the productions only, the capital letters being nonterminals and small letters terminals):

$$F\colon S \to aa,\ a \to a; \qquad F'\colon S \to b,\ b \to aa,\ a \to a.$$

The productions of F can be simulated by derivations according to F' (even by derivations of the same length 2) but La(F) is not contained in La(F') because the language {aa} is in La(F) - La(F').

For EOL forms, the situation concerning the basic lemma discussed in the previous paragraph is much more complicated. Several substitutes for the basic lemma have been established in [MSW].

Several "reduction" or "normal form" results for EOL forms are known, i.e., results allowing us to replace a given form by an equivalent simpler one. We mention the following one.

THEOREM 7.1. *For every* EOL *form an equivalent* EOL *form can be constructed such that the productions in the latter are of the types*

$$A \to a,\ A \to BC,\ A \to B,\ a \to A,\ A \to \lambda,$$

where A, B, C *are nonterminals and* a *is terminal.*

An EOL form is termed *complete* iff its language family equals the whole family of EOL languages. Although no exhaustive characterization for completeness of EOL forms is known, we have a number of results which enable us to decide the matter in most cases. Theorem 7.2 gives some necessary,

and Theorem 7.3 sufficient conditions for completeness. Let us call an EOL system *looping* (resp. *expansive*) iff it has a letter α, reachable from the initial letter, which derives itself in a positive number of steps (resp. derives a word containing two occurrences of α).

THEOREM 7.2. *Assume that* F *is a complete* EOL *form. Then* F *viewed as an* EOL *system, is looping and expansive. Furthermore,* F *contains a production sending a nonterminal to a word over the terminal alphabet, as well as a production sending a terminal to a word containing one nonterminal.*

THEOREM 7.3. *Assume that* F *is an* EOL *form such that, for some* $t \geq 1$, *the following derivations are possible according to* F (*viewed as an* EOL *system*):

$$S \overset{t}{\Rightarrow} S,\ S \overset{t}{\Rightarrow} SS,\ S \overset{t}{\Rightarrow} a,\ a \overset{t}{\Rightarrow} xSy,$$

for some words x *and* y *and terminal letter* a, *and that no words strictly over the terminal alphabet appear in these derivations at the intermediate steps. Then* F *is complete.*

The following forms F_1-F_9 (resp. H_1-H_6) provide some typical examples of complete EOL forms (resp. of EOL forms which are not complete).

F_1: S → a, S → S, S → SS, a → S

F_2: S → a, S → S, S → Sa, a → S

F_3: S → a, S → S, S → aS, a → S

F_4: S → a, S → S, S → SS, a → SS

F_5: S → a, S → λ, S → S, S → SSS, a → S

F_6: S → A, A → S, A → SS, A → a, a → A

F_7: S → a, S → SSA, S → S, a → S, A → λ

F_8: S → a, S → S, S → SS, a → N, N → N

F_9: S → a, S → S, S → SS, a → N, N → NN

H_1: S → a, S → S, S → aa, a → S

H_2: S → a, S → S, a → SS

H_3: S → a, S → S, a → S, S → SSS

H_4: S → a, a → S, a → SS, a → a

H_5: S → a, a → S, a → a

H_6: S → A, A → S, S → SS, A → a, a → A.

Note the similarity between F_6 and H_6. However, F_6 is complete and H_6 is not complete.

It is an open problem whether or not there exists an EOL form whose language family equals the family of context-free languages. However, if we consider *uniform* interpretations (i.e., when taking interpretations of productions, the substitution μ is uniform on terminals), then the language family obtained from the form

F: S → SS, S → a, a → a

equals the family of context-free languages. On the other hand, the language family La(F) contains non-context-free languages because, for instance, the EOL system with the productions

S → SS, S → a, a → c, c → c

is an interpretation of F. (This system is not a uniform interpretation of F because a → c cannot result from a → a under uniform interpretation.)

Instead of EOL forms, one can also consider so-called *pure* forms having just one alphabet (as OL systems have). The distinction between terminals and nonterminals will be made in interpretations only. Various interconnections between pure forms and EOL forms can be stated. The problem of completeness is easier for pure forms. For instance, one can show that a pure form over a one-letter alphabet {S} is complete iff it contains the productions S → S and S → SS.

It is well-known that some L families have very weak, and some others strong closure properties. Therefore, it is not surprising that, for EOL forms F, the family La(F) is sometimes an AFL, sometimes an anti-AFL. We already gave an example of the former possibility, where the language family was the family of regular languages. For the following form F:

S → a, S → cc, S → AAAA, A → AA, A → b, a → a, b → b, c → c,

the language family La(F) is an anti-AFL.

8. L DECIDABILITY

We review in this chapter briefly some recent language-theoretic decidability results concerning L systems. The problems we have in mind are in a sense comparative ones: we compare two language families K and K', where in most cases one of the families is "sequential" and the other "parallel". More specifically, we are interested in the following two decision problems for fixed language families K and K'.

(1) The *equivalence* problem between K and K': Given languages L in K and L' in K', one has to decide whether or not L = L'. (In case K = K', we speak of the equivalence problem for K.)

(2) The K'-*ness* problem for K: Given a language L in K, one has to decide whether or not L is in K'. (If K' is the family of regular languages, we speak of the regularity problem for K, etc.)

Let us call TOL systems with a one-letter alphabet "unary", and denote them as well as the generated languages by TUL (in accordance with UL systems and languages). Generalizing the earlier results concerning UL languages, Latteux has established the following results in [La1] and [La2].

THEOREM 8.1. *The* TUL*-ness problem is decidable for regular languages, and so are the regularity and* UL*-ness problems for* TUL *languages. Consequently, the equivalence problem between* TUL *languages and* UL *languages, as well as the equivalence problem between* TUL *languages and regular languages are decidable.*

The following results are established in [Sa4].

THEOREM 8.2. *The regularity and context-freeness problems are decidable for the family* DOL *and so is the equivalence problem between* DOL *and context-free languages.*

The equivalence problem between regular and OL languages, the equivalence problem between OL and DOL languages, the regularity problem for OL languages, and the OL-ness problem for regular languages are all open. However, the following result is established in [Li].

THEOREM 8.3. *The* DOL*-ness problem for context-free languages is decidable.*

The proof of Theorem 8.3 proceeds by showing that, for any context-free DOL language L, one can find a constant k such that in any DOL system generating L the lengths of the axiom and the right sides of all productions are bounded by k. Theorem 8.3 follows from this observation by Theorem 8.2.

9. L PROBLEM

What we call "L problem" is easy to guess: The DOL equivalence problem. Since this problem is perhaps the most simply stated combinatorial problem whose decidability status is still open, let us repeat here the problem in a formulation understandable to any mathematician (just in case somebody not knowing anything about formal languages happens to see this):

L PROBLEM. (Problem of Iterated Morphism). Consider quadruples K = = (V,x,g,h), where V is a finite set, x is an element of the free monoid V^* generated by V, and g and h are endomorphisms of V^*. Is there an algorithm which decides of a given K whether or not $g^i(x) = h^i(x)$ holds for all i?

It is generally believed that L Problem is decidable although some people have the opposite opinion. The latter is perhaps due to the fact that so far the Problem has resisted all efforts of finding an algorithm for its solution. Some partial results are know, cf. [RS2]. The more recent results are briefly reviewed below.

THEOREM 9.1. *The* DOL *equivalence problem is decidable for polynomially bounded* DOL *sequences.*

The first detailed proof of Theorem 9.1 appears in [Ka], although the theorem was announced earlier by Ehrenfeucht and Rozenberg. The proof in [Ka] is based on a reduction on the degree of the polynomial, leading finally to linear growth, where results like Theorem 8.2 become available. Although some notions about how the proof might be carried out are clear to everybody who has worked on the problem, it still takes a lot of careful

analysis and effort to complete the proof. Therefore, for anyone trying to get a feeling on the L problem, we recommend that he works through [Ka] (or finds a better argument himself). If he eventually thinks that he has solved the L Problem (the argument in [Ka] cannot be extended to the general case), we also hope that he goes through and presents the details carefully. (No arm waving please.)

We say that a pair (G,H) of DOL systems has a k-*bounded balance* ($k \geq 0$) iff the condition

$$|lg(h(x)) - lg(g(x))| \leq k$$

is satisfied for every word x appearing as a prefix in some word in L(G). (g and h are the homomorphisms defining the systems G and H.) The pair (G,H) has a bounded balance iff it has a k-bounded balance, for some k. A family of DOL systems is *smooth* iff every pair of (sequence) equivalent systems from the family has a bounded balance. The following result is established in [C] and [Va].

THEOREM 9.2. *Equivalence problem is decidable for every smooth family of* DOL *systems.*

Proof. Let G and H be systems from a smooth family K. For $k \geq 0$, we construct a finite automaton M_k such that $L_k = L(M_k) \cap L(G) = \phi$ iff the pair (G,H) has k-bounded balance and G and H are (sequence) equivalent. (After reading a prefix x of an input, M_k remembers, using its states, which of the words g(x) and h(x) is longer and by what word, provided the length difference does not exceed k and provided one of g(x) and h(x) is a prefix of the other.) Each L_k is an EOL language and, hence, its emptiness is decidable. Because K is smooth, testing the emptiness of $L_0, L_1, \ldots$ constitutes a partial decision procedure for equivalence. Since a partial decision procedure for nonequivalence is obvious, we obtain the required full decision procedure by running the partial ones concurrently. □

Denote by K_1 the family of DOL systems G such that, for some t, all entries in the t:th power of the growth matrix of G are positive. The following result is established in [C].

THEOREM 9.3. *The family* K_1 *is smooth. Consequently, equivalence is decidable for systems in* K_1.

THEOREM 9.4. DOL *equivalence is decidable for systems over a two-letter alphabet.*

Theorem 9.4 is easily established by a case analysis concerning growth matrices because the difficult cases are taken care of by Theorem 9.3. Theorem 9.4 appears also in [Va].

It is well-known that one can decide whether or not two DOL systems generate the same sequence of Parikh vectors. Instead of Parikh vectors, one can consider some numerical information telling more about a word than its Parikh vector. [Ru] is a comprehensive study along these lines. As regards DOL systems, it is shown in [Ru] that the equivalence with respect to many types of combinatorial mappings is decidable. For instance, this is true for the *power sum* mappings. Let $i \geq 0$ be an integer and b a letter. Define a mapping ps_b^i for words w by

$$ps_b^i(w) = \sum_j (j-1)^i,$$

where j ranges over all numbers such that b occurs as the j:th letter in w. (Thus, for i = 0, we get a component in the Parikh mapping.) For any fixed i and b, it is decidable whether or not

$$ps_b^i(w_n) = ps_b^i(w'_n)$$

holds for all n, where w_n and w'_n are two given DOL word sequences.

10. L FUTURE

So far the yearly growth in the number of papers dealing with L systems has been exponential with base 2 but it seems that during this year we cannot keep up with this rate any more (going from 256 to 512). Undoubtedly, L systems theory constitutes today a central area within formal language theory, an area in many respects richer than the theory of phrase structure grammars ever was. I leave it to the biologists to discuss and

decide about the importance of L systems for "real biology". Overtures about possibilities for applications in other areas (operating systems, linguistics) have been heard lately. Also the purely theoretical research seems to continue vigorously.

Other letters come and go, L is here to stay!

11. L REFERENCES

[AL] P. ASVELD & J. VAN LEEUWEN, *Infinite chains of hyper-AFL's*, Technische Hogeschool Twente, Memorandum nr. 99 (1975).

[AS1] P. ASVELD, *Rational, algebraic and hyperalgebraic extensions of families of languages*, Ibid., nr. 90 (1975).

[AS2] P. ASVELD, *Controlled iteration grammars and full hyper-AFL's*, Ibid., nr. 114 (1976).

[C] K. CULIK II: *On the decidability of the sequence equivalence problem for* D0L-*systems*, Theoretical Computer Science, to appear.

[CG] A. CREMERS & S. GINSBURG, *Context-free grammar forms*, Journal of Computer and Systems Sciences, 11 (1975) 86-117.

[DA] J. DASSOW, *Eine neue Funktion von* DT0L-*Systemen*, EIK, to appear.

[E] S. EILENBERG, *Automata, Languages and Machines*, Vol. A. Academic Press (1974).

[Ha] T. HARJU, *On the complexity of some* L *systems*, In preparation.

[HR] G. HERMAN & G. ROZENBERG, *Developmental Systems and Languages*, North-Holland (1975).

[HV] G. HERMAN & P. VITANYI, *Growth functions associated with biological development*, American Mathematical Monthly 83 (1976) 1-15.

[Ka] J. KARHUMÄKI, *The decidability of the equivalence problem for polynomially bounded* D0L *sequences*, Submitted for publication.

[L] A. LINDENMAYER, *Mathematical models for cellular interactions in development*, I-II. Journal of Theoretical Biology 18 (1968) 280-315.

[L1] A. LINDENMAYER, *Developmental algorithms for multi-cellular organisms: A survey of* L-*systems*, Ibid. 54 (1975) 3-22.

[La1] M. LATTEUX, *Sur les* TOL-*systèmes unaires,* Laboratoire de Calcul de l'Univ. Lille, Publ. nr. 48 (1975).

[La2] M. Latteux, *Problemes decidables concernant les* TOL-*languages unaires,* Discrete Mathematics, to appear.

[Li] M. LINNA, *The* DOL-*ness for context-free languages is decidable,* Submitted for publication.

[LR] A. LINDENMAYER & G. ROZENBERG (eds.), *Automata, Languages, Development,* At the Crossroads of Biology, Mathematics and Computer Science. North-Holland (1976).

[MSW] H. MAURER, A. SALOMAA & D. WOOD, EOL *forms,* Institut für Angewandte Informatik und Formale Beschreibungsverfahren, Univ. Karlsruhe (1976).

[NRSS] M. NIELSEN, G. ROZENBERG, A. SALOMAA & S. SKUYM, *Nonterminals, homomorphisms and codings in different variations of* OL *systems,* I-II. Acta Informatica 3, 357-364 and 4, 87-106 (1974).

[Pe] M. PENTTONEN, ETOL *grammars and* N-*grammars,* Information Processing Letters 4 (1974) 11-13.

[R1] G. ROZENBERG, *More on* ETOL *systems versus random context grammars,* UIA, Antwerpen, Publ. 75-15 (1975).

[R2] G. ROZENBERG, *Selective substitution grammars,* I. Ibid. 76-02 (1976).

[RRS] G. ROZENBERG, K. RUOHONEN & A. SALOMAA, *Developmental systems with fragmentation,* International Journal of Computer Mathematics, to appear.

[RS1] G. ROZENBERG & A. SALOMAA (eds.), L *Systems,* Springer Lecture Notes in Computer Science 15 (1974).

[RS2] G. ROZENBERG & A. SALOMAA, *The mathematical theory of* L *systems.* To appear in J. Tou (ed.), Advances in Information Systems Science, Vol. 6.

[RS3] G. ROZENBERG & A. SALOMAA, *Context-free grammars with graph controlled tables,* To appear in Journal of Computer and Systems Sciences.

[RS4] G. ROZENBERG & A. SALOMAA, *New squeezing mechanisms for* L *systems,* UIA, Antwerpen, Publ. 76-06 (1976).

[RV1] G. ROZENBERG & D. VERMEIR, *On* ETOL *systems of finite index,* Ibid. 75-13 (1975).

[RV2] G. ROZENBERG & D. VERMEIR, *On the relationship between context-free programmed grammars and* ETOL *systems,* Ibid. 75-14 (1975).

[Ru] K. RUOHONEN, *Some combinatorial mappings of words,* Ann. Acad. Scient. Fennicae, Ser. AI Dissertationes 9 (1976).

[Sa1] A. SALOMAA, *Formal Languages,* Academic Press (1973).

[Sa2] A. SALOMAA, *Parallelism in rewriting systems,* Springer Lecture Notes in Computer Science, 14 (1974) 523-533.

[Sa3] A. SALOMAA, *Undecidable problems concerning growth in information-less Lindenmayer systems,* EIK, to appear.

[Sa4] A. SALOMAA, *Comparative decision problems between sequential and parallel rewriting,* IEEE 75 CH1052-0C, pp. 62-66.

[SaSo] A. SALOMAA & M. SOITTOLA, *Automata-Theoretic Aspects of Formal Power series,* Springer Verlag, in preparation.

[So1] M. SOITTOLA, *Remarks on* DOL *growth sequences,* Revue Francaise Informatique Theoretique, to appear.

[So2] M. SOITTOLA, *Positive rational sequences,* To appear in Theoretical Computer Science.

[So3] M. SOITTOLA, *On* DOL *synthesis problem,* To appear in [LR].

[Va] L. VALIANT, *The equivalence problem for* DOL *systems and its decidability for binary alphabets,* Univ. of Leeds, Centre for Computer Studies Report nr. 74 (1975).

[Vi1] P. VITANYI, *Deterministic LIndenmayer languages, nonterminals and homomorphisms,* To appear in Theoretical Computer Science.

[Vi2] P. VITANYI, *Context-sensitive table Lindenmayer languages and a relation to* LBA-*problem,* Mathematisch Centrum, Amsterdam, prepublication 49/75 (1975).

[vL1] J. VAN LEEUWEN, *Parallel rewriting - a non-parallelled theory of languages,* Manuscript (1976).

[vL2] J. VAN LEEUWEN, *The membership question for* ETOL*-languages is polynomially complete,* Information Processing Letters 3 (1975) 138-143.

[vL3] J. VAN LEEUWEN, *Deterministically recognizing* EOL*-languages in time* $O(n^{3.81})$, Mathematisch Centrum, Amsterdam, prepublication 9/75 (1975).

[vL4] J. VAN LEEUWEN, *One-way machines using a checking stack,* Manuscript (1976).

[W] D. WOOD, *Time-delayed* OL *languages and sequences,* Information Sciences 8 (1975) 271-281.

Note added in proof.

The decidability of the DOL equivalence problem has been shown by K. Culik II and I. Fris. Their paper will appear in Information and Control.

THREE HARDEST PROBLEMS by W.J. SAVITCH

MATHEMATICAL CENTRE TRACTS 82, 1976, 111-149

THREE HARDEST PROBLEMS

W.J. SAVITCH*
Mathematical Centre, Amsterdam, The Netherlands

1. INTRODUCTION

In its most traditional form the theory of computation concerns itself with classifying functions as either computable or noncomputable. Researchers in theoretical computer science have refined this theory so as to be able to classify functions as either efficiently computable or not. The notions of efficiency most often studied, and the ones studied here, are time and storage. This general research area, which studies the efficiency of computation, is usually referred to as complexity theory. The work in this area has been quite successful at discovering functions which can be computed by efficient algorithms. The work has met with less success in its attempts to discover which functions cannot be computed by realistically efficient algorithms. Some striking successes at computing lower bounds on the time and storage requirements of computable functions have been achieved. However, it is still true that we do not yet have the techniques to prove good lower bounds for the time and storage needs of very many common functions. Recent work in complexity theory has produced results which go a long way toward isolating the problem of deciding which functions can be done by realistically efficient algorithms. A common theme in much recent work in complexity theory is to consider some class of interesting problems, say for example the recognition problem for context-free languages, and to try to determine the minimum time and storage bounds needed to solve these problems on a computer. The work has not been a total success but many such efforst have met with striking partial success. Many of these partial successes follow a similar pattern. They do not determine the time and storage needs of the class of problems in question; however, they do often produce examples of

*) Permanent address: University of California, San Diego, USA.

problems which are the hardest problem in that class. For example, there is a hardest context-free language. It is hardest in the sense that if it can be recognized with a given time or storage bound, then all context-free language can be recognized within that same time or storage bound. In these lectures we will look at three examples of problem classes that have hardest problems. The three problems considered are: the context-free language recognition problem, the problem of converting an arbitrary nondeterministic storage-bounded procedure to an efficient deterministic storage-bounded procedure, and the analogous problem for time-bounded procedures. Rather than survey the lists of hardest problems for each problem class, we will try to explain the techniques used to discover and prove that a problem is hardest for a given class. We will therefore look at one very key and representative problem for each of the three classes and will discuss in some detail the proof that these problems are among the hardest problems in their class.

All our proofs require, in an essential way, that we have a formally defined model of a computer and a formally defined notion of the time and storage requirements of a procedure. We have chosen to use the most commonly studied model; namely, the Turing machine. This model was chosen because it yields notation and proofs which are particularly simple, and we wish to exhibit the techniques of the proofs as much as the final results. However, the results presented here are model-independent. Within the bounds of accuracy of our results, all of our results apply to any reasonable model of a computer. This point is discussed in more detail when we define the model itself.

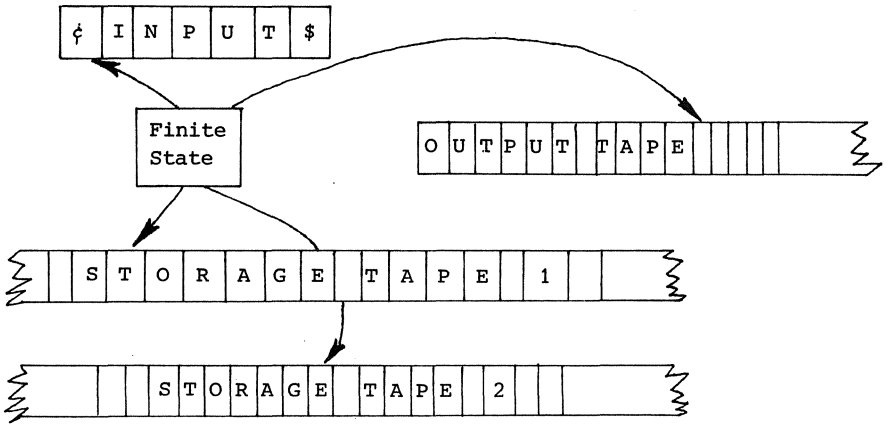


Fig. 1
Turing Machine with Two Storage Tapes

2. A MODEL FOR TIME AND STORAGE

2.1. Machine model

The Turing machine model was first introduced by Turing in 1936. Since that time Turing's model and variations on it have been central to much of the theoretical development in computer science. We will here give an informal definition of the variant of Turing's model that will serve as our model of a computer. The reader who finds this description to imprecise can consult any of a number of standard references. ([Hopcroft & Ullman 1969] for example.) A *Turing machine* is a finite state machine attached to a read-only input tape, finitely many read/write storage tapes, and a write-only output tape. The finite state machine is referred to as the *finite state control*, or sometimes just *the control*. The tapes are divided into squares. Each square of a storage tape is capable of holding any one symbol from a specified finite storage tape alphabet. The storage tapes are infinitely long in both directions. The output tape has a left-hand end but extends infintely long to the right. There is a specified finite input alphabet and a specified finite output alphabet. The input consists of a string over the input alphabet and is placed on the input tape. The input tape is provided with two distinguished end markers, one at each end of the input string. Each tape has one *tape head* communicating with finite state control. The situation is diagramed in Fig. 1. The machine in Fig. 1 has two storage tapes, and the end markers are denoted by ¢ and $.

At any point in time, each head will be scanning one square on its tape and the finite state control will be in one state. Depending on this state and the symbols scanned by the input and storage tape heads, the machine will, in one step, do all of the following:

(i) Overwrite a symbol on the scanned square of each of its storage tapes (it is, of course, permissible to overwrite a symbol by itself and so leave it unchanged),

(ii) shift its input head and each storage tape head either left one square, right one square or not at all (different heads may get different instructions),

(iii) possibly, write a symbol on the output tape. In this case the output tape head is advanced one square to the right so that it is ready to write the next output symbol,

(iv) change the state of the finite state control.

The finite state control is designed so that the input head will never leave the segment of tape containing the input string and end markers.

The machine is said to be *deterministic* if there is only one possible action at each step. It is said to be *nondeterministic* if there are finitely many possible actions at each step. So, for example, a nondeterministic Turing machine with one storage tape may have the following as an instruction:

> If the finite state control is in stape p and the input head is scanning symbol a and the storage tape head is scanning symbol c, then replace c by d, move the storage tape head left one square, move the input tape head left one square, change the finite state control to state q, and output the symbol e.

Since it is nondeterministic, it may have another instruction which is applicable in the same situation. For example, it may also have the following instruction:

> If the finite state control is in the state p and the input head is scanning symbol a and the storage tape head is scanning symbol c, then replace c by f, move the storage tape head right one square, move the input head right one square, change the finite state control to state r, and output the symbol g.

What happens when the machine is in a situation where both instructions apply? There are two (or more) possible next steps. Both next steps are considered equally valid. The machine arbitrarily chooses one of the applicable steps. If it makes the right arbitrary choices then it will successfully complete the computation. If it makes cther than the right arbitrary choices, then the computation yields no information. A formal definition of nondeterministic computations will be given later on. For now, we will give only an informal description of how these machines compute. The informal description is given in terms of parallel computations rather than "arbitrary choices", but the two concepts are equivalent.

One way to view a Turing machine is as a list of instructions of the type above. If the machine is nondeterministic, then there can be more than one instruction for a given situation. When a machine gets to a siatuation where $m \geq 2$ instructions apply, one can think of the machine as dividing into m copies of itself; each copy follows one of the m instructions. These m copies then compute in parallel and may later divide themselves. The computation will be considered successful if at least one path through this

tree of dividing machines leads to a successful outcome.

One state of the finite state control is designated as a *start state* and finitely many states are designated to be *accepting states*. One special tape symbol is designated as the *blank* symbol. At the start of a computation, the input string is placed on the input tape and delimited by end markers, the input tape head is set scanning the left end marker, the finite state control is put into the start state, the output tape and storage tapes are blank, and the output tape head is placed at the left end of the output tape.

A deterministic machine is said to *compute* a *function* f from input strings to output strings, provided that starting in the designated start configuration with input w, the machine eventually halts in an accepting state with f(w) written on its output tape. If f is a partial function then it is usual to insists that the machine not halt on any input w for which f(w) is undefined. This point will not, however, be important to what we will be doing. One can define nondeterministic machines that compute functions in a more or less similar manner. However, we will have no need for such machines here; we will use nondeterministic machines only in the restricted manner described in the next paragraph.

To simplify the discussion we will usually confine ourself to situations where the input is in some sense either accepted or rejected. In these cases there is no need for an output tape. From now on, we will assume, unless otherwise stated, that our Turing machines have no output tape. If the machine reaches an accepting state, that will designate acceptance. Thus, we say that the machine *accepts the input* w provided that there is some computation of the machine on input w which reaches an accepting state. The meaning of this for deterministic machines should be clear. For nondeterministic machines this definition of acceptance is to be interpreted as follows: The input w is accepted provided that there is some sequence of steps each of which follows one of its permitted instructions and this sequence of step ends up in an accepting state. (In terms of the dividing tree described above, this means that there is at least one path through the tree that leads to an accepting state.) Notice that these machines do not have any "rejecting" states. So it is not possible to have one sequence of moves that leads to an accepting state and have another sequence of moves, on the same input, that leads to a rejecting state. In this way we avoid inconsistencies that could otherwise arise.

In order to make the above definition of acceptance more precise and

in order to develop some notation which will be useful later on we define a concept called an *id for a Turing machine with input* w or just *id* for short. ("id" stands for "instantaneous description"). An id is a string of symbols that completely describes the configuration of the Turing machine with the input w at an instant of time. Specifically, if the machine has k storage tapes then an id is a string of the form $pm\not{c}\alpha_1\triangleright\beta_1\not{c}\alpha_2\triangleright\beta_2\not{c}\ldots\not{c}\alpha_k\triangleright\beta_k$ where p is a symbol standing for the state of the finite state control, m is a binary numeral telling the number of squares between the input tape head and the left end of the tape, and each of the strings $\alpha_i\triangleright\beta_i$ describe a storage tape configuration. $\alpha_i\triangleright\beta_i$ means that the i-th tape contains $\alpha_i\beta_i$ and that the tape head is scanning the first symbol of β_i; we always assume leading and trailing blanks are trimmed from $\alpha_i\triangleright\beta_i$. The symbols $\not{c}$ and $\triangleright$ are presumed to be new symbols. If I_1 and I_2 are ids of the type described above we will write $I_1 \vdash_w I_2$ provided that, with input w and in the configuration described by I_1, the machine has some applicable instruction that will allow it to change to the configuration represented by I_2. Notice that, if the machine is deterministic than, for any id I_1, there is at most one id I_2 such that $I_1 \vdash_w I_2$. If the machine is nondeterministic there may be finitely many I_2 such that $I_1 \vdash_w I_2$. $I_1 \vdash_w^* I_2$ provided either $I_1 = I_2$ or else there are $J_1, J_2, \ldots, J_\ell$ such that $I_1 = J_1$, $I_2 = J_\ell$ and $J_i \vdash_w J_{i+1}$ for $1 \leq i < \ell$. The definition of acceptance can now be rephrased as follows: The machine *accepts* the input w provided there is some id I_a such that I_a has the finite control in an accepting state and such that $I_0 \vdash_w^* I_a$, where I_0 is the id for the start machine configuration. An id that contains an accepting state is called an *accepting id*. (When w is clear from context we will write $\vdash$ and $\vdash^*$ for $\vdash_w$ and $\vdash_w^*$.) The *language accepted* by the machine is the set of all input strings accepted by the machine.

In practice we will state all our algorithms informally and omit the fairly straightforward but very tedius task of converting our informal algorithms into Turing machine instructions of the type described above.

2.2. Time and storage defined

We now introduce the measures of time and storage that we will use for our model of computing. In all case the time and storage will be measured as a function of the length of the input. So that, rather than having a fixed constant allotment of time or storage, we will allow more resources for

bigger inputs. If w is a string of symbols $\|w\|$ will denote the length of w.

DEFINITION. Let M be a Turing machine (deterministic or nondeterministic), let A be a set of strings over the input alphabet of M, and let both T(n) and S(n) be functions on the natural numbers.

(1) M is said to *accept* A *within time* T(n) proved that
   (i) M accepts exactly those input strings which are in A and
   (ii) for each string w in A, there is at least one accepting computation of M on w which takes $T(\|w\|)$ or fewer steps. That is, there are id's $I_0, I_1, \ldots, I_\ell$ such that I_0 is the start id, $I_i \vdash_w I_{i+1}$ for $0 \leq i < \ell$, I_ℓ is an accepting id and $\ell \leq T(\|w\|)$.

(2) M is said to *accept* A *within storage* S(n) provided that
   (i) M accepts exactly those input strings which are in A and
   (ii) for each string w in A, there is an accepting computation of M on w that uses no more than $S(\|w\|)$ squares of storage tape. More precisely there are $I_0, I_1, \ldots, I_\ell$ such that I_0 is the start id, $I_i \vdash_w I_{i+1}$ for $0 \leq i < \ell$, I_ℓ is an accepting id and for each I_i if $I_i = pm\not\alpha_1 \triangleright \beta_1 \not\alpha_2 \triangleright \beta_2 \not\ldots \not\alpha_k \triangleright \beta_k$, then

$$\sum_{j=1}^{k} \|\alpha_j \beta_j\| \leq S(\|w\|).$$

   When discussing storage, we will always assume that the machine M never overwrites a nonblank symbol by a blank symbol. There is no loss of generality in this assumption, since we can always add one extra symbol that is not the blank but it treated like a blank in doing computations. Thus the sum $\sum_j \|\alpha_j \beta_j\|$ is the total number of squares that have been scanned so far in the computation.

There are a number of observations to be made about these two definitions. First note that we are only measuring time and storage on those inputs which are accepted. If an input is not accepted, then the machine may use any amount of time and storage. This may seem peculiar for deterministic machines. However, for the kinds of well behaved time and storage bounds we will be discussing here it can be shown that the above definition is equivalent to one that requires that the machine always operates within the bound specified. The reason for phrasing the definition in this way is to accommodate nondeterministic machines. We want to say the machine operates in time T(n) or storage S(n) provided that the most efficient accepting computation

operates in time T(n) or storage S(n). This requires that our definition ignore all but the most efficient accepting computations and as a result ignore all non accepting computations. It should also be noted that if the machine is deterministic, then in parts (ii) of both (1) and (2) there will be exactly one computation to look at. Finally, we should note that the above definition can easily be extended to accommodate deterministic machines that have an output tape and which compute some partial function. When measuring storage no charge is made for the output (or input) tape, only the storage tape is charged for.

2.3. Worth of the model

The natural question to ask is how faithfully do these abstract measures of time and storage model time and storage needs of real computers. The answer is that the Turing machine is only an approximate measure. The quality of the Turing machine measure of time and storage can be described as *coarse but correct*. Suppose one writes an algorithm to solve some problem and then implements this algorithm both on a Turing machine and on some other abstract model of a computer or on some real computer. The time and storage requirements of the different implementations will be approximately the same. Let us first consider deterministic computations and let us consider time and storage separately. The easiest to analyze is storage. It can be shown that, if a set is accepted by a Turing machine within storage S(n), then we find another machine that accepts the same language in storage cS(n), for any constant c, no matter how small. To accomplish this all we need to do is enlarge the tape alphabet so many symbols can be coded as a single symbol. On the other hand, even huge changes in the model change the storage used by only a constant multiple. So saying something can be done within storage S(n) on a Turing machine, means that it can be done in order of magnitude S(n) storage on any reasonable machine. Define $S_2(n) = O(S_1(n))$ to mean $S_2(n) \leq cS_1(n)$ for some c and all but finitely many n. Saying something is doable in deterministic storage S(n) really conveys no more information than to say it is doable in deterministic storage O(S(n)). The following result makes this more precise. The proof is left as an exercise.

THEOREM 1. *If* A *is accepted by a deterministic (respectively nondeterministic) Turing machine within storage* S(n), *and if* c *is any constant greater*

than zero, then we can find another deterministic (respectively nondeterministic) Turing machine that accepts A *within storage* cS(n).

A result similar to Theorem 1 can be proven for time. Again the proof is omitted. (Theorems 1 and 2 are from [Hartmanis, Lewis & Stearns 1965; Hartmanis & Stearns 1965], and can now be found in many introductory texts.)

THEOREM 2. *If* A *is accepted by a deterministic (respectively nondeterministic) Turing machine within time* T(n) *and* c *is any positive constant, then we can find a deterministic (respectively nondeterministic) Turing machine that accepts* A *within time* $T_2(n) = \max\{cT(n), n+1\}$, *provided* $\inf_{n\to\infty} T(n)/n = \infty$.

This last theorem might lead one to believe that, just as with storage, Turing machines measure time up to a constant multiple. Unfortunately, time is not quite as well behaved as storage. If we change the model, then the time needed to implement a given algorithm can go from T(n) to a polynomial in T(n). For example, going from our model to a model with only one tape may raise the time from T(n) to $T^2(n)$. It is however true that, for most realistic models, changing the model will not change the run time from T(n) to anything worst than a polynomial in T(n). Define $T_2(n) = P(T_1(n))$ to mean that $T_2(n) \leq (T_1(n))^c$ for some c and all but finitely many n. Then to say something is doable in deterministic time T(n) really conveys little more information than to say it is doable in deterministic time P(T(n)).

The situation for nondeterministic models is similar. To say that something is doable in nondeterministic storage S(n) [respectively time T(n)] conveys little more information than to say it is doable in nondeterministic storage O(S(n)) [respectively time P(T(n))]. The comparison of deterministic and nondeterministic costs is much harder and is the topic of the last two major subdivisions of this paper.

3. CONTEXT FREE LANGUAGES

3.1. Preliminary Definitions

In this section we will describe a context-free language which is in a very

strong sense the hardest context-free language. Before going on to describe this language we will first review the definitions of context-free grammar and context-free language. We shall simplify the presentation by taking our context-free grammars to be in Greibach normal form, by assuming that our languages do not contain the empty string and by assuming that all of our derivations are leftmost derivations. These simplifications cause no loss of generality, since every context-free language can be generated by a grammar in Greibach normal form, since the complexity of a language is not affected by adding or deleting the empty string to the language, and since every derivation has an equivalent leftmost derivation. So, for our purposes, we define a *context-free grammar* to consist of four distinguished items (Σ,N,S,P) where: Σ and N are disjoint finite sets of symbols called the *terminal* and *nonterminal* alphabets; S is a nonterminal symbol referred to as the *start symbol*; P is a finite set of rewrite rules of the form $A \rightarrow aX_1X_2\ldots X_n$ where $A,X_1,X_2,\ldots,X_n$ are all nonterminals and a is a terminal symbol. (The possibility of $n = 0$ is allowed. In that case, the rule is $A \rightarrow a$.) These rewrite rules are frequently referred to as *productions*. We write $\mu A\nu \Rightarrow \mu aX_1X_2\ldots X_n\nu$ provided μ is a string of terminal symbols, ν is a string of symbols from $\Sigma \cup N$, and $A \rightarrow aX_1X_2\ldots X_n$ is one of the productions in P. The symbolism $\overset{*}{\Rightarrow}$ denotes the reflexive, transitive closure of $\Rightarrow$. So $\alpha \overset{*}{\Rightarrow} \beta$, provided that either $\beta = \alpha$ or β can be obtained from α by finitely many applications of $\Rightarrow$. The language generated by G is denoted $L(G)$ and is defined to be the set of all strings over the terminal alphabet which can be obtained from the start symbol by repeated application of the rewrite rules. That is, $L(G) = \{w \mid S \overset{*}{\Rightarrow} w \text{ and } w \in \Sigma^*\}$. ($\Sigma^*$ denotes the set of all strings made up from symbols in Σ.)

The hardest context-free language which we are about to describe has the property that every context-free language can be obtained from it by a very simple type of transformation called an inverse homomorphism. Suppose Σ and Δ are finite alphabets. A *homomorphism* of Σ^* into Δ^* is defined to be a mapping h which maps each symbol of Σ onto a string in Δ^*. The domain of the mapping h is extended from Σ to Σ^* by defining $h(\text{the empty string}) = \text{the empty string}$ and by defining $h(a_1a_2\ldots a_n) = h(a_1)h(a_2)\ldots h(a_n)$ where the a_i are individual symbols. If L is a language over Δ, then $h^{-1}(L)$ is the language over Σ which is defined by $h^{-1}(L) = \{w \mid h(w) \in L\}$. $h^{-1}(L)$ is said to be obtained from L by *inverse homomorphism*.

3.2. A hardest context free language

We now proceed to describe a language discovered by Greibach and shown by her to be the hardest context-free language. We need one preliminary definition before we can define the language in question.

DEFINITION. The *Dyck set on two letters* is the set of all strings over the four symbols (,), [and] which have the property that all parenthesis and brackets match in the usual way. The Dyck set on two letters will be denoted by D. A more precise way to describe D is as follows. A string w is in D provided that w can be transformed into the empty string by repeated application of the two rewrite rules: (1) for any x and y, x()y rewrites to xy and (2) for any x and y, x[]y rewrites to xy.

We now describe a language L_0 which will turn out to be the hardest context-free language. L_0 will not be described in such a way that it is immediately obvious that it is a context-free language. It is, however, a not too difficult exercise to write a context-free grammar G such that $L(G) = L_0$.

DEFINITION. Let $T = \{(,),[,],c\}$. Let $L_0 = \{x_1cy_1cz_1dx_2cy_2cz_2d\ldots dx_\ell cy_\ell cz_\ell d \mid \ell \geq 1,\ x_i,z_i \in T^*$ for all i and $[(y_1y_2\ldots y_\ell \in D\}$.

L_0 can be described less formally as follows. The only things that are candidates for membership in L_0 are strings of the form $w_1dw_2d\ldots w_\ell d$ such that each w_i is a string of the form $y_1^icy_2^ic\ldots cy_{n(i)}^i$ where each y_j^i is a string consisting of parenthesis and brackets. That is, each y_j^i is made up of the symbols (,), [and]. In order to be in L_0, such a candidate string must pass the additional test that there is some way of choosing one y from each w in such a way that [(, followed by the concatenation of the chosen y's is a string in D; that is, such that all parenthesis and brackets match. For example, the candidate

$$)c]]c]]]d()c()[d(()]c]]c]]$$

is in L_0, since [( ) ()[ ]] is in D. The "y's" chosen to get a correct cancellation are underlined. The next result is the main theorem of this section; it is from Greibach [1973].

THEOREM 3. *If* L *is any context-free language, then there is some homomor-*

phism h *such that* $L = h^{-1}(L_0)$.

Before proving Theorem 3, we will derive some of its consequences. The next two theorems are corollaries of Theorem 3. They say that, in terms of both time and storage, L_0 is in a very strong sense the hardest context-free language.

THEOREM 4. *Let* T(n) *be any monotone, nondecreasing time bound. Suppose* L_0 *is accepted by some deterministic (respectively nondeterministic)* T(n) *time-bounded Turing machine and that* L *is any other context-free language. Then there is a constant* c *depending only on* L *such that* L *is accepted by some deterministic (respectively nondeterministic)* T(cn) *time-bounded Turing machine.*

A special case may help to illustrate the importance of Theorem 4.

COROLLARY. *If* L_0 *is accepted within deterministic time* $O(n^{\alpha})$, *where* α *is a non-negative real number, then every context-free language can be accepted within deterministic time* $O(n^{\alpha})$.

The corollary is immediate from Theorem 4. The fastest known general algorithm for accepting arbitrary context-free languages is due to [Valiant 1975]. This algorithm runs in time $O(n^{2.81})$. If the single language L_0 could be accepted in n^{α} for $\alpha < 2.81$, then we would have a faster algorithm for general context-free language acceptance. It should be noted that Valiant's algorithm was not stated for Turing machines and it is not clear that it can be made to run on a Turing machine in time $O(n^{2.81})$. However, the proof of Theorem 4 remains true if we replace Turing machines by any other "reasonable" model of a computer. In particular, Theorem 4 and Valiant's results both hold true for a random-access type model of a computer and such random-access type models actually look more, rather than less, like the architecture of current real computers. (For a discussion of such random-access type machines, see [Cook & Reckhow 1973].) It should also be noted that Theorem 4 is really of no great interest for nondeterministic machines. This is because all context-free languages can be recognized in nondeterministic time $T(n) = n+1$. (See most any introductory text on the subject. For example [Hopcroft & Ullman 1969].) However, we will include the non-

deterministic machines in the proof, so that we can draw analogies to nondeterministic storage. We now give the proof of Theorem 4.

Proof of Theorem 4. Suppose L_0 is accepted by a Turing machine M_0 within time T(n). Let L be any contex-free language. By Theorem 3, there is a homomorphism h such that $L = h^{-1}(L_0)$. Let c be the maximum length of h(a), as a varies over all symbols in the domain of h. We will describe a Turing machine M that accepts L within time T(cn); if M_0 is deterministic, then M will also be deterministic. The algorithm for M is fairly straightforward. Given input w, M must decide if w is in L. Now w is in $L = h^{-1}(L_0)$ if and only if h(w) is in L_0. So all M needs to do is to simulate M_0 operating on h(w). If the simulation accepts h(w), then h(w) is in L_0, so w is in L and M accepts w. One way to implement this would be to have M first compute h(w), write h(w) on a scratch tape and then simulate M_0 on h(w). This will take time cn to compute h(w) plus about T(cn) to simulate M_0. So the total time would be at most cn + T(cn). By a more efficient implementation, we can get the total time down to T(cn). The more efficient implementation does not compute h(w) all at once but merely computes parts of h(w) as needed. Specifically, the more efficient implementation proceeds as follows. Given $w = a_1a_2\ldots a_n$, M's input is always scanning some a_i (or an end marker). When M is scanning a_i, it pretends it is scanning $h(a_i)$. To accomplish this you can think of the finite state control of M as having inside of it a finite buffer that can hold up to c symbols. As part of the first single move on input a_i, M can put $h(a_i)$ inside this buffer. M then does a step-by-step simulation of M_0. This simulation will tell if M_0 accepts h(w) and so will tell if w is in $L = h^{-1}(L_0)$. Since no extra time is needed to compute h(w), M runs in the time needed to simulate M_0 on h(w). So M runs in time $T(\|h(w)\|) \leq T(cn)$. □

A result analogous to that of Theorem 4 holds for storage. The proof is also analogous to that of Theorem 4 and so is left as an exercise.

THEOREM 5. *Let* S(n) *be any monotone nondecreasing storage bound. Suppose* L_0 *is accepted by some deterministic (respectively nondeterministic)* S(n) *storage bounded Turing machine, and that L is any other context-free language. Then there is a constant* c *depending only on* L *such that* L *is accepted by some deterministic (respectively nondeterministic)* S(cn) *storage bounded Turing machine.*

To date, the most storage efficient method for general context-free recognition uses $O((\log_2 n)^2)$ storage. This is the best known bound for both deterministic and nondeterministic machines. The algorithm can be implemented on a Turing machine or most any other standard model and will still run in storage $O((\log_2 n)^2)$. The algorithm is due to Lewis, Stearns & Hartmanis [1965]. Finding a more efficient algorithm for the particular language L_0 would lower this bound. Having described the importance of Theorem 3, we now give its proof.

Proof of Theorem 3. Let L be a context-free language and let G be a context-free grammar for L. (Recall that we are assuming that L does not include the empty string and that, as we defined context-free grammars, G is in what is commonly called Greibach normal form.) We will explicitely describe a homomorphism h such that $L = h^{-1}(L_0)$. The homomorphism h is described in terms of G. Let $A_1, A_2, \ldots, A_m$ be an enumeration, without repetition, of the nonterminals of G. Also, let this enumeration be such that A_1 is the start symbol. We will describe a method for coding every production of G as a string of parenthesis and brackets. We will then use this code to describe h.

If $A_i \to a$ is a production, where a is a terminal, then define

$$\text{Code}(A_i \to a) =)^i];$$

recall that $)^i$ means) written i times.

If $A_i \to aA_{j_1}A_{j_2}\ldots A_{j_r}$ is a production of G, where a is a terminal, then define

$$\text{Code}(A_i \to aA_{j_1}A_{j_2}\ldots A_{j_r}) =)^i][(^{j_r}[(^{j_{r-1}}\ldots[(^{j_1}.$$

Using this Code function, we can now define h. We need to define a string h(a), for each terminal symbol a of G. Let a be an arbitrary terminal symbol of G. Let $p_1, p_2, \ldots, p_\ell$ be an enumeration of all productions such that a occurs on the right-hand side of the production. Define

$$h(a) = \text{Code}(p_1)c\text{Code}(p_2)c\ldots c\text{Code}(p_\ell)d.$$

This completely describes the homomorphism h.

It remains to show that $L = h^{-1}(L_0)$. In order to establish this equality, we will show two claims. It will help in understanding the claims

if you recall that, by our definition of context-free grammar, one terminal symbol is produced at each step of a derivation and that, since we always rewrite the leftmost nonterminal, the terminal symbols are produced left to right.

CLAIM 1. Suppose $a_1a_2...a_n$ is a string of terminal symbols and $p_1,p_2,...,p_n$ is a sequence of rewrite rules such that a_i appears on the right-hand side of p_i, $i = 1,2,...,n$. Then the following two statements are equivalent:

(1) $A_1 \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow ... \Rightarrow \alpha_n = a_1a_2...a_n$ by applying productions $p_1,p_2,...,p_n$ (to the leftmost nonterminal),

(2) $[(Code(p_1)Code(p_2)...Code(p_n) \in D$.

CLAIM 2. w is in L if and only if h(w) is in L_0.

Once Claim 2 is established, we will have completed the proof. We will first derive Claim 2 from Claim 1. We will then go back and prove Claim 1. To see Claim 2, suppose w is in L. We then know that $A_1 \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow ... \Rightarrow \alpha_n = w = a_1a_2...a_n$ by applying some productions $p_1,p_2,...,p_n$. (Recall that A_1 is the start symbol.) Now, by the way we defined context-free grammars, the symbol a_j appears on the right-hand side of production p_j, $j = 1,2,...,n$. So $h(a_j) = x_jcCode(p_j)cz_jd$. So

(3) $h(w) = x_1cCode(p_1)cz_1dx_2cCode(p_2)cz_2d...dx_ncCode(p_n)cz_nd$.

But by Claim 1,

(4) $[(Code(p_1)Code(p_2)...Code(p_n)$ is in D.

So by (3), (4) and the definition of L_0, h(w) is in L_0. Conversely, suppose h(w) is in L_0. By reversing the above argument, we can show that w is in L. Thus Claim 2 follows from Claim 1.

It remains to show Claim 1. In order to do this we will prove a slightly stronger statement. (To get Claim 1 from Claim 1', set $s = 1$ and $i_1 = 1$.)

CLAIM 1'. Assume the hypothesis of Claim 1. If $A_{i_1}A_{i_2}...A_{i_s}$ are any nonterminals, then the following two statements are equivalent.

(1') $A_{i_1}A_{i_2}...A_{i_s} \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow ... \Rightarrow \alpha_n = a_1a_2...a_n$, by applying productions $p_1,p_2,...,p_n$.

(2') $[(^{i_s}[(^{i_{s-1}}[...[(^{i_1}$ $Code(p_1)Code(p_2)...Code(p_n) \in D$.

Before proving Claim 1', it might help to see an example. Suppose (1') is $A_1 \Rightarrow aA_3 \Rightarrow abA_2A_1 \Rightarrow abcA_1 \Rightarrow abcd$. In this case, the string in (2') is

$$[(\ )][(((\ )))][([((\ ))]\ )]$$

which is clearly in D. To see where the string came from, we have underlined every other string of the form $Code(p_i)$. In the general case, it is not hard to see that if a rule $A_i \to aA_{j_1}A_{j_1}A_{j_2}\ldots A_{j_r}$ is applied then the prefix $)^{i}]$ of $Code(A_i \to aA_{j_1}A_{j_2}\ldots A_{j_r}) =)^{i}][(^{j_r}[(^{j_{r-1}}\ldots[(^{j_1}$ will cancel leaving $[(^{j_r}[^{j_{r-1}}\ldots[(^{j_1}$. This string will in turn cancel out because the production applied to A_{j_1} has a code beginning with $)^{j_1}]$ and this $)^{j_1}]$ cancels with the $[(^{j_1}$; the production applied to A_{j_2} has a code beginning with $)^{j_2}]$, and so forth. We now give a formal proof of this intuitive idea.

Claim 1' is proven by induction on n. We first show that (1') implies (2'). Suppose (1') is true and consider the base case n = 1. In this case, (1') reduces to $A_{i_1} \Rightarrow a_1$ by $p_1 = A_{i_1} \to a_1$. But $[(^{i_1}Code(p_1) = [(^{i_1})^{i_1}]$ which is in D. So (2') is true. Next consider the inductive step of the proof. Suppose (1') is true and suppose that Claim 1' is true for all derivations of less than n steps. The production p_1 must be of the form $A_{i_1} \to a_1A_{j_1}A_{j_2}\ldots A_{j_t}$. So, by (1') we conclude that

$$a_1A_{j_1}A_{j_2}\ldots A_{j_t}A_{i_2}A_{i_3}\ldots A_{i_s} = \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n = a_1a_2\ldots a_n$$

and so we may conclude

(1") $$A_{j_1}A_{j_2}\ldots A_{j_t}A_{i_2}A_{i_3}\ldots A_{i_s} \Rightarrow \alpha_2' \Rightarrow \alpha_3' \ldots \Rightarrow \alpha_n' = a_2a_3\ldots a_n$$

by applying productions $p_2, p_3, \ldots, p_n$, where $\alpha_i = a_1\alpha_i'$, $i = 2,3,\ldots,n$. Now consider (2'), which we are trying to prove. Since p_1 is $A_{i_1} \to a_1A_{j_1}A_{j_2}\ldots A_{j_t}$, $Code(p_1) =)^{i_1}][(^{j_t}[(^{j_{t-1}}\ldots[(^{j_1}$. So substituting this in for $Code(p_1)$ in (2') and cancelling $[(^{i_1})^{i_1}]$, we see that (2') is equivalent to

(2") $$[(^{i_s}[(^{i_{s-1}}\ldots[(^{i_2}[(^{j_t}[(^{j_{t-1}}\ldots[(^{j_1}Code(p_2)Code(p_3)\ldots Code(p_n) \in D.$$

In order to complete the inductive step we must show (2') of Claim 1'. To do this it will suffice to show (2"). But we know (1") is true and we know that, by induction hypothesis (2") follows from (1"). So (2") and hence (2') is true and the inductive step is completed. This completes the proof that (1') implies (2') of Claim 1'. The proof that (2') implies (1') is

similar and will be left as an exercise. So Claim 1' is established and this completes the proof of Theorem 3. □

4. NONDETERMINISTIC STORAGE

The next class of problems we wish to consider is that of simulating an arbitrary nondeterministic algorithm by a storage efficient deterministic algorithm. In order to motivate the basic definitions and to get some practice in simulating Turing machine computations, we first present one efficient simulation algorithm. We then go on to produce a hardest problem in this class and to discuss the possibility of producing a more efficient simulation.

4.1. An efficient simulation algorithm

It is well known that any nondeterministic algorithm can be simulated by a deterministic algorithm. All the deterministic algorithm need do on an input w is to simulate all possible computations of the nondeterministic algorithm on input w. However, if this idea is implemented in the obvious way, then the deterministic algorithm will use an amount of storage that is exponential in the amount used by the nondeterministic algorithm. However, there is a way of checking all possible computations that uses much less storage than that. The simulation algorithm is from [Savitch 1970].

THEOREM 6. *If a set* A *is accepted by a nondeterministic Turing machine* M *within storage*[*)] S(n) *and* $S(n) \geq \log_2 n$, *then* A *is accepted by a deterministic Turing machine* M_D *within storage* $S^2(n)$.

Proof. To simplify the proof it will help to assume that the storage bound S(n) is well behaved. Functions that are well behaved in the sense we need are called storage constructable.

---

*) For our purposes, we make the convention $\log_b 0 = 0$, for all $b \geq 2$. We also assume $S(n) \geq 1$, for all storage bounds S(n).

DEFINITION. A function S(n) is said to be *storage constructable* if there is a deterministic Turing machine which, given an input of length n, will mark off exactly S(n) squares on a storage tape and, furthermore, will use no more than S(n) storage in the process.

Assume that S(n) is storage constructable. Later we will note that this assumption can be dropped. The theorem is proven by exhibiting an algorithm whereby M_D can simulate the computation of M. To understand the simulation, we will need to recall some facts about id's of M. Consider a fixed input w to M and let n be the length of w. Consider an id which could occur in an accepting computation of M and that uses no more than S(n) storage. There is a constant b, depending only on M, with the property that all such id's have length at most bS(n). This is because the portion that codes a storage tape has length at most S(n) + 1, the portion that codes the input head position has length at most the length of n + 2 in binary and so has length at most $O[\log_2 n] = O[S(n)]$. So these pieces, plus the state symbol and punctuation symbols ¢, have total length at most bS(n) for some suitable b.

Having obtained a bound on the length of an id, we can use this bound to derive a bound on the running time of M. Suppose M accepts an input w of length n. Then there is an accepting computation of M on w such that no id has length greater than bS(n). Now there are at most $a^{bS(n)}$ such id's where a-1 is the number of symbols needed to write id's. From this we can conclude that no more than $a^{bS(n)}$ distinct id's occur in the accepting computation. Now if the accepting computation takes more than $a^{bS(n)}$ steps, then some id is repeated. We can eliminate the steps between the repeated id and get a shorter accepting computation. If the shorter computation still takes more than $a^{bS(n)}$ steps, then we can again obtain a still shorter computation. If we proceed in this way, we eventually obtain an accepting computation which contains at most $a^{bS(n)}$ steps and in which all id's have length at most bS(n).

We can always find an integer c such that $a^{bS(n)} \leq 2^{cS(n)}$ and $bS(n) \leq cS(n)$. It will be helpful to consider the number of steps in a computation to be a power of two. Let I_0 denote the start id of the nondeterministic machine M. In order to tell if M accepts w, it will suffice for M_D to test for all accepting id's I_a whether or not $I_0 \vdash_w^* I_a$ on M by a computation of at most $2^{cS(n)}$ steps, in which all id's have length at most cS(n). After testing each I_a, it can clear storage for the next tests. So if it can make each of these tests in storage $O(S^2(n))$, then M_D can tell

if M accepts w and can do it all in deterministic storage $0(S^2(n))$. In order to see that M_D can efficiently make the tests to see if $I_0 \vdash_w^* I_a$, for start and accepting id's, it will suffice to establish,

CLAIM: There is an algorithm with the following properties.

Input : I_1, I_2 two id's for M such that $\|I_1\| \leq cS(n)$ and $\|I_2\| \leq cS(n)$.

Output: "yes" if $I_1 \vdash_w^* I_2$ on M by a computation which takes at most 2^m steps and in which each id has length at most $cS(n)$. Otherwise the output is "no".

Storage used : $mcS(n)$ [not counting the storage to hold I_1 and I_2 themselves].

To see that the theorem follows from the claim, set $m = cS(n)$. Then all the tests can be done in $c^2S^2(n) = 0(S^2(n))$ storage as desired. In order to prove the claim, we will give the algorithm. The algorithm is stated recursively. However, it can be converted to a nonrecursive algorithm of the type required by the definition of Turing machine. The algorithm follows.

ALGORITHM A

Case: $m > 0$.

I. For each id J such that $\|J\| \leq cS(n)$ do the following:
   1. Call the algorithm recursively to test whether or not $I_1 \vdash_w^* J$ on M in at most 2^{m-1} steps by id's of length at most $cS(n)$.
   2. Clear storage and then again call the algorithm recursively to test whether or not $J \vdash_w^* I_2$ on M in at most 2^{m-1} steps by id's of length at most $cS(n)$.

II. If for at least one such J we get affirmative answers to both 1 and 2 above, then $I_1 \vdash_w^* J \vdash_w^* I_2$ and so output "yes". Otherwise output "no".

Case: $m = 0$.

This requires testing to see if $I_1 \vdash_w I_2$ or $I_1 = I_2$, which can easily be done with no storage (besides that used to hold I_1 and I_2 themselves).

In order to implement Algorithm A we must be able to go through all id's J such that $\|J\| \leq cS(n)$. However, since S(n) is storage constructable, this is easy to do within storage $cS(n)$. For example, we can run through

all strings of length cS(n) in lexicographic order and test each one to see if it is an id.

To see that Algorithm A runs in storage mcS(n), proceed by induction. The base case is easy. Proceeding inductively, suppose m > 0. The algorithm needs cS(n) storage to hold the id's J that it is testing as intermediate points. For each such J it makes two tests:

(i) $I_1 \vdash_w^* J$ in at most 2^{m-1} steps,

(ii) $J \vdash_w^* I_2$ in at most 2^{m-1} steps.

Storage is cleared between tests. So the total storage needed is cS(n) to store the J plus (m-1) cS(n) for the recursive tests. So the total storage is cS(n) + (m-1)cS(n) = mcS(n), as desired. The bookkeeping required to implement this recursion on a Turing machine is rather messy but uses only standard techniques.

Thus the proof is complete in the case where S(n) is storage constructable. Most all reasonable functions are storage constructable. However, the proof can be made to work even for those S(n) which do not have this property. The interested reader is referred to [Savitch 1970], if he wishes to see this extra bit of generality. □

4.2. Storage hard and complete

The previous theorem said that we can simulate nondeterministic storage S(n) by deterministic storage $S^2(n)$. Whether or not one can do a more efficient simulation than that is an open question. We can, however, exhibit a language with the following properties: The language can be accepted in nondeterministic storage $\log_2 n$; furthermore, any deterministic algorithm to efficiently accept this one particular language can be used to produce a storage-efficient deterministic simulation of any nondeterministic algorithms. Thus if we are interested in converting arbitrary nondeterministic algorithms to deterministic algorithms, then it will be sufficient to study this one particular language. Actually, there are many such languages but we will focus on one particularly simple and representative one. First we will give some definitions so that we can make these remarks precise. In the following definition, L may be any class of languages whatsoever. However, it will help to think of it as the class of all languages that can be accepted within nondeterministic storage $(\log_2 n)^\alpha$, for some $\alpha \geq 1$.

DEFINITION. Let $\mathcal{L}$ be a class of languages and L be a particular language (L may or may not be in $\mathcal{L}$). L is said to be *storage hard* for the class $\mathcal{L}$ provided the following holds for all storage bounds of the form $(\log_2 n)^\alpha$, where $\alpha \geq 1$. If there is some deterministic Turing machine that accepts L within storage $(\log_2 n)^\alpha$ then, for any language L_2 in $\mathcal{L}$, there is some deterministic Turing machine that accepts L_2 within the same storage bound $(\log_2 n)^\alpha$. L is said to be *storage complete* for the class $\mathcal{L}$ provided L is storage hard for $\mathcal{L}$ and L is in $\mathcal{L}$.

Restricting ourselves to storage bounds of the form $(\log_2 n)^\alpha$ may seem quite arbitrary. However, there are two good reasons for this choice. The first reason is that we will be considering classes $\mathcal{L}$ for which it is already known that everything in $\mathcal{L}$ can be done in storage $(\log_2 n)^\alpha$, for some α. So the goal is to see how small a value of α will suffice to accept all languages in $\mathcal{L}$. These classes $\mathcal{L}$ will also have the property that they contain some languages which cannot be accepted in storage $(\log_2 n)^\alpha$, for any $\alpha < 1$. Thus the lowest value of α we can hope for is $\alpha = 1$. Therefore, for the classes $\mathcal{L}$ which we will be considering, to say that L is storage complete for $\mathcal{L}$ is to say that L is the hardest problem in $\mathcal{L}$, provided storage is our measure of difficulty. For example, suppose we take $\mathcal{L}$ to be the class of context-free languages. It is known that everything in $\mathcal{L}$ can be accepted in deterministic storage $(\log_2 n)^2$. It is also known that there are context-free languages which cannot be accepted in storage $(\log_2 n)^\alpha$, for any $\alpha < 1$. So a storage complete language for $\mathcal{L}$ would be a hardest context-free language. Theorem 5 says, among other things, that the language L_0 of Greibach is storage complete for the class of context-free languages. The second reason for considering bounds of the form $(\log_2 n)^\alpha$ is that the proofs we will give only work for bounds which are of this form or similar to this form. At the end of this chapter we will, however, see that many of our results generalize to arbitrary storage bounds.

4.3. A storage complete language

We now describe a language which is storage complete for the class of all languages which are accepted within nondeterministic storage $O(\log_2 n)$. Later on we will see that this language is the key to simulating nondeterministic storage in general; the importance of this language is not limited to the storage bound $\log_2 n$.

DEFINITION. A *maze* is a finite directed graph with a single distinguished node called the *start node* and a finite number of distinguished nodes called *goal nodes*. The maze is said to be *threadable* if there is a path from the start node to some goal node. We will always assume that a maze with k nodes has its nodes numbered one through k.

A (base b) *coding* of a threadable maze with k nodes is a string of the following form

$$s[1*y_1^1*y_2^1*\ldots*y_{n(1)}^1][2*y_1^2*y_2^2*\ldots*y_{n(2)}^2]\ldots$$

$$\ldots[k*y_1^k*y_2^k\ldots*y_{n(k)}^k]*u_1*u_2*\ldots*u_g$$

where the numbers $1,2,\ldots,k$ are written in base b; s, the y's and the u's are base b numerals standing for the node they number; s is the start node; $u_1,u_2,\ldots,u_g$ are the goal nodes; and, for each i, $y_1^i,y_2^i,\ldots,y_{n(i)}^i$ is an enumeration, without repetition, of all nodes y such that there is a directed arc from i to y. [,] and * are three new symbols. M_b will denote the set of all base b codings of threadable mazes.

THEOREM 7. *For any* $b \geq 2$, *the set* M_b, *of all codings of threadable mazes, can be accepted by a nondeterministic Turing machine within storage* $\log_2 n$.

Proof. The algorithm for the Turing machine is quite simple. Given an input string, the machine first checks to see if the input is the coding of some maze (possibly not threadable). This it can do deterministically in storage $0(\log_2 n)$. (The details are messy but not difficult.) If the input is not the coding of some maze, then the computation is aborted. The heart of the problem is to tell if a given maze is threadable. So, assume the algorithm has determined that the input is the coding of some maze. It then proceeds as follows. The machine writes down the number of the start node on a storage tape. It then locates the block which describes the arcs leaving the start node. (In the notation above, it finds the s-th block which is enclosed in brackets.) It then nondeterministically chooses one of the nodes that can be reached from the start node by traversing a single directed arc. (In the notation above, it chooses one y_j^s.) It replaces the start node by this newly chosen node. It then finds the block corresponding to this newly chosen node, nondeterministically chooses one node that can be reached by

traversing another arc and replaces the number of the old node by the number of this newly chosen node. It proceeds in this way to nondeterministically trace out a path through the graph. When it chooses each new node, it checks to see if it is a goal node. If a goal node is ever reached, then the machine accepts the input. If the maze is threadable, then clearly there is some sequence of nondeterministic moves that will lead to a goal node. Thus this algorithm accepts M_b. So it remains to estimate the storage. If the input has length n, then there can be no more than n nodes in the graph. So every node is named by a number which is at most n. A base b numeral which is no bigger than n can be written down in $0(\log_b n)$ storage. But the algorithm needs to store the number for only one node at a time. Hence, the whole algorithm runs in storage $0(\log_b n)$. By Theorem 1, the algorithm can be made to work in storage $\log_2 n$, since $\log_2 n = 0(\log_b n)$. □

By Theorem 7, we can conclude that M_b can be accepted in nondeterministic storage log n. In what follows it will be convenient to know that the difficulty of accepting M_b is independent of the base b. The proof of the following lemma is left as an exercise. [Notice that for any bases b and d which are at least 2, $\log_b n = 0(\log_d n)$. So we will often write log n omitting the base. This is because, by Theorem 1, any such storage bounds are equally powerful.]

LEMMA 1. *Let* b *and* d *be any two bases which are greater than or equal to two. Let* S(n) *be any storage bound such that* $S(n) \geq \log_2 n$. *If there is a deterministic Turing machine that accepts* M_b *within storage* S(n), *then we can find another deterministic Turing machine that accepts* M_d *within storage* S(n).

THEOREM 8. *For any* $b \geq 2$, *the set* M_b, *of all codings of threadable mazes, is storage complete for the class of languages that can be accepted within nondeterministic storage* log n.

Proof. Let L denote the class of all languages that can be accepted within nondeterministic storage log n. By Theorem 7, we know that M_b is in L. So it will suffice to show that M_b is storage-hard for L. Let α be a real number which is greater than or equal to one. We will show that: If we have a deterministic algorithm that accepts M_b within storage $(\log n)^\alpha$ and L is any language in L, then we can write a deterministic algorithm to accept L

within storage $(\log n)^{\alpha}$.

We first describe a deterministic algorithm that accepts L. This algorithm will not run in storage $(\log n)^{\alpha}$. Later on we will indicate how the algorithm can be modified to run in storage $(\log n)^{\alpha}$. Let Z be a nondeterministic Turing machine that accepts L within storage $\log_2 n$. Let w be an input string to Z of length n. We associate a maze m(w) with w. The nodes of m(w) are all those id's of Z which use no more than $\log_2 n$ squares of storage tape, and which assume an input of length at most n. That is, all id's $p\ell\not\alpha_1\triangleright\beta_1\not\alpha_2\triangleright\beta_2\not\ldots\not\alpha_k\triangleright\beta_k$ such that ℓ is the numeral for a number which is at most $n+2$ and such that $\Sigma_i \|\alpha_i\beta_i\|$ is at most $\log_2 n$. (k is the number of tapes of Z.) There is a directed arc from id I_1 to id I_2 provided Z, with input w, can go from configuration I_1 to configuration I_2 in one step. That is, there is an arc from I_1 to I_2, provided $I_1 \vdash_w I_2$ on Z. The start node of m(w) is the start id of Z. An id is a goal node if it contains an accepting state. With this definition of m(w), note that

(1) Z accepts w if and only if m(w) is threadable.

Let d be the number of distinct symbols needed to write id's of Z. If we identify these d symbols with the digits 0 through d - 1, then each id can be considered a base d numeral. This gives us a natural way of numbering the nodes of the maze m(w). We number them one through h, where h is the largest number whose base d numeral is an id of the type being considered. (Of course, some numerals will not be id's. These numbers can be considered isolated points of the graph and ignored.) Using this numbering we can write a coding of m(w). In order to keep from getting buried in notation, we will let m(w) denote both the maze m(w) and the base d coding of m(w). Now we can rewrite (1) as

(2) Z accepts w if and only if m(w) is in M_d.

We can now give a deterministic algorithm to simulate Z.

ALGORITHM B

Input: String w

1. Construct m(w)
2. Test if m(w) is in M_d
3. If m(w) is in M_d, then accept w.

By (2) we know that algorithm B accepts exactly the same input strings as Z. Unfortunately, algorithm B does not run in storage $(\log n)^{\alpha}$. If implemented in a straightforward fashion, Step 1 would require $\|m(w)\|$ storage. Let us estimate $\|m(w)\|$. The string m(w) consists of a list of id's separated by some punctuation symbols. If $n = \|w\|$, then there are at most n^s distinct id's that appear in m(w), where s is a constant depending only on Z. In general, id's will occur more than once. Still, there is a constant t such that this list will contain a total of at most n^t id's. Each id can be written down in $O(\log n)$ space. Thus the total length of m(w) is n^c, for some constant c depending only on Z. Now n^c is clearly much larger than $(\log n)^{\alpha}$. So we will need to implement Step 1 in other than the most obvious way. Before saying more about Step 1, let us analize Step 2. We know M_b can be accepted in deterministic storage $(\log n)^{\alpha}$. So by Lemma 1, we know there is a deterministic Turing machine Z_d that accepts M_d within storage $(\log n)^{\alpha}$. So the test in Step 2 can be done in storage $(\log\|m(w)\|)^{\alpha} = O[(\log n^c)^{\alpha}] = O[(\log n)^{\alpha}]$. So, except for the storage needed to construct and hold m(w), the algorithm runs in storage $(\log n)^{\alpha}$. In order to modify the algorithm so that it runs in total storage $(\log n)^{\alpha}$, we will need one claim.

CLAIM. There is a deterministic Turing machine Z_m which, given input w, will output m(w) and, furthermore, Z_m will use only $O(\log_2 n)$ storage.

We first see how the claim allows us to modify algorithm B to run in storage $(\log n)^{\alpha}$. After that we will prove the claim. The unmodified algorithm B would be implemented on a Turing machine as follows. Simulate Z_m and, in this way, write m(w) on a storage tape. Then use this storage tape as a simulated input tape and simulate Z_d to test if m(w) is in M_d. Aside from the tape holding m(w), the algorithm runs in storage $(\log n)^{\alpha}$. Now the simulated input head of Z_d never sees more than one symbol of m(w) at a time. So we can modify algorithm B as follows and still have it work. The tape that previously held m(w) will now hold two things, the single symbol being scanned by the simulated input head of Z_d and a binary numeral telling how many symbols there are between the left end of m(w) and the simulated tape head of Z_d. Every time the simulation of Z_d requires that the input head of Z_d move, this symbol-numeral pair must be updated. It is easy to update the numeral; simply add or subtract one from it, whichever is appropriate. Let q be the new numeral so obtained. In order to

obtain the new symbol, Z_m is simulated from the beginning. The output is never written any where but a count is kept (on an extra scratch tape) to see how many symbols would have been outputed. When this count gets to q, then the q-th symbol of m(w) is produced and entered as the updated symbol. The simulation of Z_d can then proceed.

Let us estimate the storage used by this modified algorithm B on an input w of length n. Steps 1 and 2 are no longer clearly separated in time and it is best to talk about them as routines rather than steps. One routine effectively constructs symbols of m(w) as needed. This routine must store one symbol plus two numbers, both of which are at most $\|m(w)\|$, and it must also simulate Z_m on input w. It takes $O(\log \|m(w)\|) = O(\log n)$ to store the symbols and numbers. By the claim, it takes $O(\log n)$ storage to simulate Z_m. So the routine that generates the sumbols of m(w) uses a total of $O(\log n)$ storage. The only other storage is that used to simulate Z_d to see if m(w) is in M_d. As in the unmodified algorithm, this takes storage $O[(\log \|m(w)\|)^{\alpha}]$. But $\alpha \geq 1$. So the total storage for the modified algorithm B is $O(\log n) + O[(\log n)^{\alpha}] = O[(\log n)^{\alpha}]$, as desired.

It remains to prove the claim. Let c' be a constant such that all id's which use $\log_2 n$ storage or less have length bounded above by $c'\log_2 n$. We will describe how Z_m operates on an input w of length n. Z_m first of all outputs the start id. On one storage tape Z_m then generates, in numerical order, all base d numerals of length at most $c'\log_2 n$. (Recall that id's have been identified with base d numerals.) For each such numeral i, it finds the list $y_1^i, y_2^i, \ldots, y_{n(i)}^i$ of id's such that $i \vdash_w y_j^i$ and y_j^i uses storage at most $\log_2 n$. (If i is not an id, then the list is empty.). Z_m then outputs $[i*y_1^i*y_2^i*\ldots*y_{n(i)}^i]$. Z_m then again generates, in numerical order, all base d numerals of length at most $c'\log_2 n$. Z_m tests each i to see if it is an accepting id; if it is Z_m output * followed by the id. Z_m clearly computes m(w). Furthermore, Z_m operates within storage $O(\log_2 n)$. This completes the proof of the claim and the proof of the theorem. □

The previous theorem says that any storage efficient algorithm that deterministically accepts M_b can be used to convert any nondeterministic log n storage algorithm to a storage efficient deterministic one. The same techniques for converting from nondeterministic to deterministic storage apply to all storage bounds $S(n) \geq \log_2 n$ and not just to the case $S(n) = \log_2 n$. In particular, by mimicking the proof of Theorem 8, we can prove the next result. The proof is omitted but can fairly easily be constructed by

following the proof of Theorem 8 as a model.

THEOREM 9. *Suppose there is a deterministic Turing machine* Z *and a base* $b \geq 2$ *such that* Z *accepts* M_b, *the set of all codings of threadable mazes, and such that* Z *runs in storage* $(\log n)^{\alpha}$. *In this case every set accepted by a nondeterministic* S(n) *storage bounded Turing machine is also accepted by some deterministic* $(S(n))^{\alpha}$ *storage bounded Turing machine, provided* $S(n) \geq \log_2 n$ *and* $\alpha \geq 1$.

Other storage complete languages are discussed by Sudborough [1975a,b] and Jones [1975]. (They do not actually prove that their languages are storage complete as we have defined it here. However, it is not difficult to see that their languages are storage complete.) Sudborough exhibits a context-free language (in fact, a linear context-free language), which is storage complete for the class of languages accepted in nondeterministic storage log n. This means that a new more storage-efficient algorithm for general context-free language recognition would have the side effect of producing a more storage-efficient algorithm for deterministic simulation of arbitrary nondeterministic algorithms. In [Sudborough 1976] a language is shown to be storage complete for the class of deterministic context-free languages. [Hartmanis & Hunt 1974] give a language which is in a natural sense "storage hardest" for the class of context-sensitive languages.

5. NONDETERMINISTIC TIME

5.1. Polynomial time

There are problems which are known to have algorithmic solutions but yet, in practice, cannot be solved on computers; the algorithms cost too much to run. They would exhaust computing resources long before the algorithms terminate. Many such problems are intrinsically that difficult. So there is no hope of designing completely general and yet practical algorithms for these problems. Proofs for the existence of such problems can be found in most texts on complexity theory. (See [Aho, Hopcroft & Ullman 1974]. A good discussion of some particularly interesting problems appeared in [Rabin 1974] and in works by Fischer, Meyer and Stockmeyer.) It would be nice to determine a time-bound T(n) such that any algorithm that runs in time T(n) is efficient

enough for practical implementation, and any algorithm that runs in time which is significantly greater than T(n) is not efficient enough for practical implementation. Since computer technology is continually changing, it seems that there is no hope of finding such a T(n) which will remain valid over time. If, however, one asks for only a very approximate calculation of such a bound, then there is some hope. Our introductory remarks indicated that the measures of time that we are discussing are only accurate to within a polynomial bound; that is, if $T_1(n) = P(T_2(n))$ then, within our limits of accuracy, $T_1(n)$ and $T_2(n)$ can be considered the same bound. This suggests that the things doable in polynomial time are a likely candidate for the class of practically doable problems. It is a widely held view that, given forseeable advances in computer technology, any algorithm that cannot be made to run in polynomial time is too slow for practicle implementation. On the other hand, an algorithm that runs in Time $T(n) = n$ or $T(n) = n^2$ is efficient enough to be considered practical.

DEFINITION. Let P be the class of all languages which can be accepted by deterministic Turing machines within a polynomial time bound.

By our above remarks, P would appear to be the class of practically doable problems. (The class P was first proposed as an important class to study by Cobham [1964].) The argument that things which are too difficult to be in P are too difficult to solve by a computer appears to be a reasonably sound argument. The argument that everything in P can be solved within practical time bounds is quite false. (Is something doable in time n^c practical if c is astronomically large? For that matter, is cn a practical time bound for very large c?) However, given the accuracy of our model, P is the best approximation to the class of practically doable problems that we have so far been able to produce. The true boundary between practically doable and impractically difficult problems lies somewhere in the class P and probably is a fluid boundary that will continue to change as technology changes. In any event, P seems to be a very natural class to look at. The class remains the same whether we define it in terms of Turing machines, random access-type machines or most any reasonable model of a computer. Also, we can say with some confidence that things not in P are too difficult for practical solutions, given current technology. So P can at least be thought of as those problems for which there is some hope of a practical solution. ([Hartmanis & Simon 1974 and Savitch & Stimson 1976] have given models

which appears to be much more powerful than most models. These machines appear to be able to do things which are not in the class P, and to do them in polynomial time. However, these models use unbounded parallism in an explicite or implicite way. Here we are discussing only serial computations when we refer to deterministic polynomial time.)

By analogy to the definition of P, we define a similar class for nondeterministic computations.

DEFINITION. Let NP be the class of all languages which can be accepted by nondeterministic Turing machines within a polynomial time bound.

5.2. Time hard and complete

Whether or not $P = NP$ is a major open question. We do know of a large number of problems which are in NP and for which practical algorithms would be very useful. So the question of how efficiently a nondeterministic algorithm can be converted to a deterministic algorithm is an important one. To give a perspective on what is in NP, we now list a few problems which are in NP. (These problems are not stated as languages to be accepted but as problems to be solved. The difference is one of notation and the translation can easily been made.)

1. Given an integer matrix A and a vector d, does there exist a 0 - 1 vector x such that $Ax = d$?
2. Given a graph G and a positive number n, determine if the nodes of G can be colored with n colors so that no two adjacent nodes have the same color.
3. Given a directed graph G, does G have a directed cycle which includes each node exactly once ?
4. Given a Boolean expression, is there some assignment of truth values to the variables which makes the expression true ?

A discussion of these, and many other such problems, can be found in [Karp 1972]. All of these languages are in NP. Also, they all have the interesting property that, if any one of them is in P, then they are all in P. In fact, if any one of them is in P, then $P = NP$. So they are, in some sense, the hardest languages in NP. This brings us to our next definition.

DEFINITION. Let L be any class of languages and let L be any particular language. L is said to be *time hard* for L provided that the following holds:

If L is accepted by some deterministic Turing machine within polynomial time, then every language in L is accepted within deterministic polynomial time. L is said to be *time complete* for L provided that L is time hard for L and L is in L.

The four languages listed above are time complete for the class NP. The hardest one to show complete is the fourth one, the satisfiable Boolean expressions. It is also the key complete problem. The proof of its completeness is used implicitely or explicitely in the proofs of completeness for all problems known to be time complete for NP. We will show the completeness of this problem. The proof is from [Cook 1971]. The proofs of the other complete problems referred to can be found in [Karp 1972]. (What we are calling time complete for the class NP is the same concept that Karp calls NP *complete*.)

5.3. A time complete language

Notation. Let SAT denote the set of all satisfiable Boolean expressions. That is, SAT is the set of all (parenthesized) Boolean expressions such that there is some assignment of truth values to the variables which makes the value of the entire expression true. We will use $\wedge$, $\vee$ and $\neg$ for "and", "or" and "not" respectively. Formally, variables will be strings Pm where m is a binary numeral. In this way SAT is a language over a finite alphabet. We will, however, abbreviate freely and use any convenient notation to abbreviate variables.

THEOREM 10. SAT *is in* NP.

Proof. The following algorithm can tell if a string is in SAT.

ALGORITHM C

Input: A string to be tested for membership in SAT.

1. Test if the input is a Boolean expression. If it is not, then abort the computation; otherwise, continue.
2. For each variable Pm do the following: Nondeterministically choose either "True" or "False". Replace each occurrence of Pm by the truth value chosen.
3. Evaluate the Boolean expression produced by 2. If it evaluates to

"True", then the string is in SAT.

Clearly, Algorithm C accepts exactly the satisfiable Boolean expressions. It remains to show that the algorithm runs in polynomial time. Step 1 requires testing to see that the expression has balanced parenthesis, that each binary operation ($\wedge$ and $\vee$) has two arguments and that each $\neg$ has one argument. It is not difficult to write a deterministic polynomial time algorithm for this. Step 2 requires a loop to iterate as many times as there are variables. Each iteration requires one pass over the input. If the input has length n, then Step 2 requires at most n iterations of a loop and each iteration takes time a polunomial in n. So Step 2 requires time a polynomial in n. It is straightforward to write a polynomial time algorithm for Step 3. So the total time is the sum of three polynomials and hence is itself a polynomial. □

THEOREM 11. SAT *is time complete for the class* NP.

Proof. By Theorem 10 we know SAT is in NP. So it will suffice to show that SAT is time hard for NP. In order to show that we will need one lemma.

LEMMA 2. *Suppose* L *is accepted by a nondeterministic Turing machine* Z *within polynomial time. We can find a deterministic Turing machine* Z_f *which runs in polynomial time, has the same input alphabet as* L *and computes a function* f *with the following property: For any string* w *over the input alphabet of* Z, Z *accepts* w *if and only if* f(w) *is in* SAT,

We now assume that the lemma is true and complete the proof of Theorem 11. After completing the proof of Theorem 11, we will go back and prove the lemma. Suppose SAT is accepted by a deterministic Turing machine Z_S within polynomial time. Suppose L is in NP. We must produce a deterministic algorithm for L that runs in polynomial time. Since L is in NP, there is a nondeterministic Turing machine Z that accepts L in polynomial time. Let Z_f be as in the lemma.

ALGORITHM D

Input: A string w to be tested for membership in L.

1. Simulate Z_f to compute f(W)
2. Simulate Z_S to test if f(w) is in SAT
3. If f(w) is in SAT, then accept w.

By the lemma, w is in L if and only if f(w) is in SAT. So Algorithm D accepts exactly those strings which are in L. Step 1 takes time n^c for some c, where $n = \|w\|$. Since Step 1 takes time n^c, we know that $\|f(w)\| \leq n^c$. Also, by hypotheses, Z_S runs in time n^b, for some b. So, Step 2 takes time at most $\|f(w)\|^b \leq n^{bc}$. Step 3 takes a constant amount of time. So the total time for Algorithm D is bounded by a polynomial in $n = \|w\|$. Since L was an arbitrary element of *NP*, it follows that SAT is time hard for *NP* and so time complete for *NP*. So, once Lemma 2 is proven, the proof of Theorem 11 will be complete.

Proof of Lemma 2. We will assume that Z has only one storage tape. The proof is the same if Z has finitely many storage tapes but the notation is much simpler if we assume that Z has just one storage tape. (Actually, it is fairly easy to show that it is possible to simulate finitely many tapes by one tape and still have the algorithm run in polynomial time. In any event, there is no loss of generality in assuming there is just one storage tape.) We will also assume that Z never reaches a halting configuration. To insure this, add some trivial instructions to Z that is applicable when no other instruction is applicable. So, after accepting, Z does some irrelevant series of moves. We will explicitly give the algorithm for f(w). The algorithm is quite intuitive, even though the details are very tedious. The Boolean formula f(w) will be designed so that it in some sense says "Z accepts w". In order to define f(w), we will need quite a bit of notation. We now tabulate this notation.

S = the set of states of Z.

s = the start state of Z.

Y = the set of accepting states of Z.

A = the storage tape alphabet of Z.

b = the blank symbol for Z.

$a_0a_1a_2\ldots a_na_{n+1}$ denotes the string of symbols on the input tape. So $w = a_1a_2\ldots a_n$, where the a_i are individual symbols; a_0 and a_{n+1} are end markers.

p(n) is a polynomial such that Z runs in time p(n).

The storage tape head can move a maximum of p(n) squares from its original position within time p(n). So only p(n) squares of storage tape on either side of the tape head's initial position need be considered. We number these squares as follows. The square the tape

head starts out on is numbered zero. The p(n) squares to the right of this square are numbered 1,2,...p(n). The p(n) squares to the left of this square are numbered -1,-2,...-p(n).

f(w) contains a number of variables. Below we list the abbreviation for these variables and their intuitive meaning. Each variable is meant to stand for a proposition which is true if and only if the intuitive meaning is true.

$T^a_{s,t}$ "means" storage tape square number s contains symbol a at time t; $a \in A$, $-p(n) \leq s \leq +p(n)$, $0 \leq t \leq p(n)$.

$H_{s,t}$ "means" the storage tape head is scanning square number s at time t; $-p(n) \leq s \leq -p(n)$, $0 \leq t \leq p(n)$.

S^q_t "means" the finite state control is in state q at time t; $q \in S$, $0 \leq t \leq p(n)$.

$I_{i,t}$ "means" the input head is scanning the square with symbol a_i at time t, $0 \leq i \leq n+1$, $0 \leq t \leq p(n)$.

$f(w) = S \wedge I \wedge H \wedge T \wedge B \wedge M \wedge A$, where S, I, H, T, B, M and A are Boolean formulae with the following intuitive meanings. All references are to computations with input w.

S "says" that at each instant of time the finite state control of Z is in one and only one state.

I "says" that at each instant of time the input head of Z is at one and only one square.

H "says" that at each instant of time the storage tape head is at one and only one storage tape square.

T "says" that at each instant of time each storage tape square contains one and only one symbol.

B "says" that at time zero, Z is in the initial configuration.

M "says" that at each instant of time, in any configuration, Z follows one of its instructions; that is it "says" that symbols and states change in the way that the finite control of Z might change them.

A "says" that at some time, Z reaches an accepting state.

Given the intuitive meanings of the variables, it is not too difficult to

build Boolean expressions which have the above described intuitive meaning. For example, consider S. For a fixed t, $\vee_{q\in S} S_t^q$ "says" that at time t Z is in some state $q \in S$. ($\vee_{q\in S} S_t^q$ is an abbreviation for $S_t^{q_1} \vee S_t^{q_2} \vee \ldots \vee S_t^{q_e}$ where $S = \{q_1, q_2, \ldots, q_e\}$). So $\wedge_{t=0}^{p(n)} (\vee_{q\in S} S_t^q)$ "says" that, at all times, Z is in some state. For a fixed t, $\wedge_{r\neq q} \neg (S_t^r \wedge S_t^q)$ "says" that at time t, Z is not in two different states. So $\wedge_{t=0}^{p(n)} (\wedge_{r\neq q} \neg (S_t^r \wedge S_t^q))$ "says" that, at all times, Z is in at most one state. So the correct formula for S is

$$S = \left[\bigwedge_{t=0}^{p(n)} \left(\bigvee_{q\in S} S_t^q\right)\right] \wedge \left[\bigwedge_{t=0}^{p(n)} \left(\bigwedge_{r\neq q} \neg (S_t^r \wedge S_t^q)\right)\right] .$$

The intuition for formulas I, H, T, B and A are similar. So we give them without explanation.

$$I = \left[\bigwedge_{t=0}^{p(n)} \left(\bigvee_{i=0}^{n+1} I_{i,t}\right)\right] \wedge \left[\bigwedge_{t=0}^{p(n)} \left(\bigwedge_{i\neq j} \neg (I_{i,t} \wedge I_{j,t})\right)\right] ,$$

$$H = \left[\bigwedge_{t=0}^{p(n)} \left(\bigvee_{s=-p(n)}^{+p(n)} H_{s,t}\right)\right] \wedge \left[\bigwedge_{t=0}^{p(n)} \left(\bigwedge_{s\neq r} \neg (H_{s,t} \wedge H_{r,t})\right)\right] ,$$

$$T = \left[\bigwedge_{t=0}^{p(n)} \bigwedge_{s=-p(n)}^{+p(n)} \left(\vee_{a\in A} T_{s,t}^a\right)\right] \wedge \left[\bigwedge_{t=0}^{p(n)} \bigwedge_{s=-p(n)}^{+p(n)} \bigwedge_{a\neq c} \neg (T_{s,t}^a \wedge T_{s,t}^c)\right],$$

$$B = I_{0,0} \wedge H_{0,0} \wedge S_0^{\delta} \wedge \bigwedge_{s=-p(n)}^{+p(n)} T_{s,0}^b ,$$

$$A = \bigvee_{t=0}^{p(n)} (\vee_{q\in Y} S_t^q) .$$

The formula for M is arrived at by the same kind of intuition but is a bit more complicated. So we will describe it in terms of subformulae. For each input head position i, the i-th input symbol is some letter a. So for each input head position i, each state r and each storage tape symbol c, there are one or more instructions of the following form which are applicable when the machine has its input head at position i, its finite state control in state r and its storage tape head scanning symbol c.

> If the finite state control is in state r and the input head is scanning symbol a and the storage tape head is scanning symbol c, then replace c by d, move the storage tape head x squares, move the input tape head y squares and change the finite state control to state q. (x and y may independently be -1, 0 or +1.)

Let this be a fixed but arbitrary such instruction and call this instruction α. We will construct a formula $M(\alpha,s,t)$ which "says" that, in the transition from time t to time $t+1$, instruction α is followed. $M(\alpha,s,t)$ is constructed assuming that, at time t, the storage tape head is at square s, the input tape head is at square i and the instruction α is applicable.

$$M(\alpha,s,t) = T^{d}_{s,t+1} \wedge H_{s+x,t+1} \wedge I_{i+y,t+1} \wedge S^{q}_{t+1} .$$

Notice that $M(\alpha,s,t)$ does not say that α is applicable only that it is followed. Notice also that $M(\alpha,s,t)$ says nothing about storage tape squares other than s. Both of these considerations will be dealt with later on as we build up M.

Let i be an arbitrary input head position, r an arbitrary state, c an arbitrary storage tape symbol, s an arbitrary storage tape square and t an arbitrary time. We next construct a formula $M(i,r,c,s,t)$ which "says" that: if at time t the input head is in position i, the finite state control is in state r, and the storage tape head is in position s scanning symbol c, then some applicable instruction is followed. First let $\alpha_1,\alpha_2,\ldots,\alpha_k$ be a list of all the instructions which are applicable in the situation under discussion, then define

$$M(i,r,c,s,t) = (I_{i,t} \wedge S^{r}_{t} \wedge H_{s,t} \wedge T^{c}_{s,t}) \rightarrow$$

$$(M(\alpha_1,s,t) \vee M(\alpha_2,s,t) \vee \ldots M(\alpha_k,s,t)).$$

The symbol $\rightarrow$ is being used for the Boolean connective "implies". So $A \rightarrow B$ is an abbreviation for $\neg A \vee B$.

Finally we can define the formula M in terms of subformulae defined above.

$$M = \bigwedge_{t=0}^{p(n)-1} \bigwedge_{s=-p(n)}^{+p(n)} \bigwedge_{i,r,c} \{M(i,r,c,s,t) \wedge$$

$$[H_{s,t} \rightarrow (\bigwedge_{a} \bigwedge_{v\neq s} (T^{a}_{v,t} \rightarrow T^{a}_{v,t+1}))]\}$$

The subformula in square brackets says that, except for the storage tape square scanned at time t, the contents of the storage tape is unchanged

from time t to time t + 1. Given the intuitive meaning of the propositional variables, the intuitive meaning of M should be clear. M says that, at every time t, the id for time t + 1 is obtained from the id at time t by following some applicable instruction of Z.

The algorithm for f(w) has now been completely described. It remains to show two things.

CLAIM 1. f(w) can be computed in polynomial time.

CLAIM 2. Z accepts w if and only if f(w) is in SAT.

First consider Claim 1. Since p(n) is a polynomial each of the subformulae S, I, H, T, B, M and A are of length at most $p_2(n)$, where $p_2(n)$ is some polynomial depending only on p(n) and Z. Also, each of them can be constructed by algorithms that essentially just write them down in one left-to-right pass. There is some bookkeeping involved but the algorithms can easily be made to all run within some polynomial time bound $p_3(n)$. So the time needed to produce all of f(w) is some polynomial $p_4(n)$.

Finally consider Claim 2. If Z accepts w then there is some computation of Z on w that passes through an accepting state. Assign to $T^a_{s,t}$ the value true if in this computation storage square s contains symbol a at time t; otherwise assign $T^a_{s,t}$ the value False. Assign $H_{s,t}$ the value True if in this computation the storage tape head is scanning square s at time t; otherwise assign $H_{s,t}$ the value False. Similarly, assign truth values to the S^q_s and $I_{i,t}$ according to whether or not their intuitive meaning holds in this computation. Clearly, this assignment of truth values makes f(w) true. Thus, if Z accepts w then f(w) is in SAT.

Conversely, suppose f(w) is in SAT. Then there is some assignment of truth values to the variables $T^a_{s,t}$, $H_{s,t}$, S^q_t and $I_{i,t}$ that makes f(w) true. Conisder such an assignment of truth values. Since the assignment makes f(w) true, it makes S true. So for each time t, exactly one S^q_t was assigned true. Associate this unique state q with time t. Since this assignment makes f(w) true, it makes T true. So for each time $t = 0,1,\ldots,$ p(n) and each storage tape square s, $-p(n) \leq s \leq +p(n)$, there is one and only one symbol a such that $T^a_{s,t}$ is true. In this way we can, for each time t, assign a unique symbol a to each storage tape square s. Since this assignment makes f(w) true, it also makes I and H true; so in a similar way we can use this assignment to associate a unique input tape head position

and a unique storage head position with each instant of time $t = 0,1,\ldots,p(n)$. In this way, the given assignment of truth values associates a unique id of Z to each time unit t from zero to p(n). If we can show that this sequence of id's is an accepting computation of Z on input w, then we will know Z accepts w and we will have shown Claim 2. But this assignment of truth values makes B, M and A each true. Since B is true, the id at time zero is the correct start id; at each instant of time t, the id at time $t+1$ is the result of some instruction of Z, since this is what M says and, finally, since A is true some id includes an accepting state. So this sequence of id's is an accepting computation of Z on w. Thus, if f(w) is in SAT, then Z accepts w. This completes the proof of Claim 2 and the proof of Lemma 2. □

An analysis of the proof of Theorem 11 shows that we have actually proven something stronger than Theorem 11. Consider the formula

$$f(w) = S \wedge I \wedge H \wedge T \wedge B \wedge M \wedge A ,$$

constructed in Lemma 2. The subformulae S, I, H, T, B and A are all either in conjunctive normal form or can be converted to conjunctive normal form by some trivial operations. Thus, if we convert M to conjunctive normal form, then f(w) will be in conjunctive normal form. In order to convert M to conjunctive normal form, we need only convert each of the subformulae M(i,r,c,s,t) to this form, use the definition of $\rightarrow$ to replace the remaining $\rightarrow$'s by $\neg$ and $\vee$, and finally apply the distributive law to the $\neg H_{s,t}$. But the subformulae M(i,r,c,s,t) are all fairly simple. So these subformulae can easily be converted to conjunctive normal form. Thus we can define f(w) so that it is in conjunctive normal form and is still computable within deterministic polynomial time. This means that, instead of Claim 2, we could have proved

CLAIM 2'. Z accepts w if and only if f(w) is a satisfiable Boolean formula in conjunctive normal form.

This small modification to the proof yields,

THEOREM 12. *The set of satisfiable Boolean formulae in conjunctive normal form is time complete for the class NP.*

5.4. Conclusions

Unfortunately, there is no known deterministic polynomial time algorithm to accept the satisfiable Boolean expressions. The most widely held view is that none exists and that P does not equal NP. So it appears that SAT and all the many interesting problems that are time complete for NP are just too difficult to do within practical time limits. The best known deterministic algorithms for these problems take time c^n, where c is a constant depending on the problem.

REFERENCES

A.V. AHO, J.E. HOPCROFT, J.D. ULLMAN (1974), The Design and Analysis of Computer Algorithms, Addison-Wesley.

A. COBHAM (1964), *The intrinsic computational difficult of functions*, Proceedings of the 1964 International Congress for Logic, Methodology and Philosophy of Science, North-Holland, 24-30.

S.A. COOK (1971), *The complexity of theorem-proving procedures*, Proceedings of the Third Annual ACM Symposium on Theory of Computing, 151-158.

S.A. COOK, R.A. RECKHOW (1973), *Time bounded random access machines*, JCSS 7, 354-375.

M.J. FISCHER, M.O. RABIN (1974), *Super-exponential complexity of presburger arithmetic*, MIT MAC Technical Memorandum 43.

S.A. GREIBACH (1973), *The hardest context-free language*, SIAM J. on Computing 2, 304-310.

J. HARTMANIS, H. HUNT (1974), *The lba problem and its importance in the theory of computing*, SIAM-AMS Proceedings 7, 1-26.

J. HARTMANIS, P.M. LEWIS II, R.E. STEARNS (1965), *Hierarchies of memory limited computations*, IEEE Conference Record on Switching Circuit Theory and Logical Design, 179-190. (The material cited can be found as Theorem 10.1 of Hopcroft & Ullman (1969).)

J. HARTMANIS, R.E. STEARNS (1965), *On the computational complexity of algorithms*, Trans. AMS 117, 285-306. (The material cited can be found as Theorem 10.3 of Hopcroft & Ullman (1969).)

J. HARTMANIS, J. SIMON (1974), *On the power of multiplication in random access machines*, Proceedings IEEE 15th Annual Symposium on Switching and Automata Theory, 13-23.

J.E. HOPCROFT, J.D. ULLMAN (1969), Formal Languages and Their Relations to Automata, Addison-Wesley.

N.D. JONES (1975), *Space-bounded reducibility among combinatorial problems*, JCSS 11, 68-85.

R.M. KARP (1972), *Reducibility among combinatorial problems*, in R.E. Miller & J.W. Thatcher (Eds.), Complexity of Computer Computations, Plenum Press, 85-103.

R.M. KARP (1975), *On the computational complexity of combinatorial problems*, Networks 5, 45-68.

P.M. LEWIS II, R.E. STEARNS, J. HARTMANIS (1965), *Memory bounds for the recognition of context-free and context-sensitive languages*, IEEE Conference Record on Switching Circuit Theory and Logical Design, 191-202. (Much of the material also appeared in Hopcroft & Ullman (1969).)

A. MEYER, L. STOCKMEYER (1972), *The equivalence problem for regular expressions with squaring requires exponential space*, Conf. Record IEEE Thirteenth Annual Symposium on Switching and Automata Theory, 125-129.

M.O. RABIN (1974), *Theoretical Impediments to artificial intelligence*, Proceedings IFIP Congress 74, Stockholm, 615-619.

W.J. SAVITCH (1970), *Relationships between nondeterministic and deterministic tape complexities*, JCSS 4, 177-192.

W.J. SAVITCH, M.J. STIMSON (1976), *The complexity of time bounded recursive computations*, Proceedings 1976 Conference on Information Sciences and Systems, The Johns Hopkins University, 42-46.

I.H. SUDBOROUGH (1975a), *On tape-bounded complexity classes and multihead finite automata*, JCSS 10, 62-76.

I.H. SUDBOROUGH (1975b), *A note on tape-bounded complexity classes and linear context-free languages*, JACM 4, 499-500.

I.H. SUDBOROUGH (1976), *On deterministic context-free languages, multihead automata and the power of auxiliary pushdown force*, Proceedings of the Eighth Annual ACM Symposium on Theory of Computing.

A.M. TURING (1936), *On computable numbers with an application to the entscheidungs problem*, Proc. London Math., Soc. 2-42, 230-265 (a correction, *ibid*, 43, 544-546).

L.G. VALIANT (1975), *General context-free recognition in less than cubic-time*, JCSS 10, 308-315.

OTHER TITLES IN THE SERIES MATHEMATICAL CENTRE TRACTS

A leaflet containing an order-form and abstracts of all publications mentioned below is available at the Mathematisch Centrum, Tweede Boerhaavestraat 49, Amsterdam-1005, The Netherlands. Orders should be sent to the same address.

MCT 1 T. VAN DER WALT, *Fixed and almost fixed points*, 1963. ISBN 90 6196 002 9.

MCT 2 A.R. BLOEMENA, *Sampling from a graph*, 1964. ISBN 90 6196 003 7.

MCT 3 G. DE LEVE, *Generalized Markovian decision processes, part I: Model and method*, 1964. ISBN 90 6196 004 5.

MCT 4 G. DE LEVE, *Generalized Markovian decision processes, part II: Probabilistic background*, 1964. ISBN 90 6196 005 3.

MCT 5 G. DE LEVE, H.C. TIJMS & P.J. WEEDA, *Generalized Markovian decision processes, Applications*, 1970. ISBN 90 6196 051 7.

MCT 6 M.A. MAURICE, *Compact ordered spaces*, 1964. ISBN 90 6196 006 1.

MCT 7 W.R. VAN ZWET, *Convex transformations of random variables*, 1964. ISBN 90 6196 007 X.

MCT 8 J.A. ZONNEVELD, *Automatic numerical integration*, 1964. ISBN 90 6196 008 8.

MCT 9 P.C. BAAYEN, *Universal morphisms*, 1964. ISBN 90 6196 009 6.

MCT 10 E.M. DE JAGER, *Applications of distributions in mathematical physics*, 1964. ISBN 90 6196 010 X.

MCT 11 A.B. PAALMAN-DE MIRANDA, *Topological semigroups*, 1964. ISBN 90 6196 011 8.

MCT 12 J.A.TH.M. VAN BERCKEL, H. BRANDT CORSTIUS, R.J. MOKKEN & A. VAN WIJNGAARDEN, *Formal properties of newspaper Dutch*, 1965. ISBN 90 6196 013 4.

MCT 13 H.A. LAUWERIER, *Asymptotic expansions*, 1966, out of print; replaced by MCT 54 and 67.

MCT 14 H.A. LAUWERIER, *Calculus of variations in mathematical physics*, 1966. ISBN 90 6196 020 7.

MCT 15 R. DOORNBOS, *Slippage tests*, 1966. ISBN 90 6196 021 5.

MCT 16 J.W. DE BAKKER, *Formal definition of programming languages with an application to the definition of ALGOL 60*, 1967. ISBN 90 6196 022 3.

MCT 17 R.P. VAN DE RIET, *Formula manipulation in ALGOL 60, part 1*, 1968. ISBN 90 6196 025 8.

MCT 18 R.P. VAN DE RIET, *Formula manipulation in ALGOL 60, part 2*, 1968. ISBN 90 6196 038 X.

MCT 19 J. VAN DER SLOT, *Some properties related to compactness*, 1968. ISBN 90 6196 026 6.

MCT 20 P.J. VAN DER HOUWEN, *Finite difference methods for solving partial differential equations*, 1968. ISBN 90 6196 027 4.

MCT 21 E. WATTEL, *The compactness operator in set theory and topology*, 1968. ISBN 90 6196 028 2.

MCT 22 T.J. DEKKER, *ALGOL 60 procedures in numerical algebra, part 1*, 1968. ISBN 90 6196 029 0.

MCT 23 T.J. DEKKER & W. HOFFMANN, *ALGOL 60 procedures in numerical algebra, part 2*, 1968. ISBN 90 6196 030 4.

MCT 24 J.W. DE BAKKER, *Recursive procedures*, 1971. ISBN 90 6196 060 6.

MCT 25 E.R. PAERL, *Representations of the Lorentz group and projective geometry*, 1969. ISBN 90 6196 039 8.

MCT 26 EUROPEAN MEETING 1968, *Selected statistical papers, part I*, 1968. ISBN 90 6196 031 2.

MCT 27 EUROPEAN MEETING 1968, *Selected statistical papers, part II*, 1969. ISBN 90 6196 040 1.

MCT 28 J. OOSTERHOFF, *Combination of one-sided statistical tests*, 1969. ISBN 90 6196 041 X.

MCT 29 J. VERHOEFF, *Error detecting decimal codes*, 1969. ISBN 90 6196 042 8.

MCT 30 H. BRANDT CORSTIUS, *Excercises in computational linguistics*, 1970. ISBN 90 6196 052 5.

MCT 31 W. MOLENAAR, *Approximations to the Poisson, binomial and hypergeometric distribution functions*, 1970. ISBN 90 6196 053 3.

MCT 32 L. DE HAAN, *On regular variation and its application to the weak convergence of sample extremes*, 1970. ISBN 90 6196 054 1.

MCT 33 F.W. STEUTEL, *Preservation of infinite divisibility under mixing and related topics*, 1970. ISBN 90 6196 061 4.

MCT 34 I. JUHÁSZ, A. VERBEEK & N.S. KROONENBERG, *Cardinal functions in topology*, 1971. ISBN 90 6196 062 2.

MCT 35 M.H. VAN EMDEN, *An analysis of complexity*, 1971. ISBN 90 6196 063 0.

MCT 36 J. GRASMAN, *On the birth of boundary layers*, 1971. ISBN 90 6196 064 9.

MCT 37 J.W. DE BAKKER, G.A. BLAAUW, A.J.W. DUIJVESTIJN, E.W. DIJKSTRA, P.J. VAN DER HOUWEN, G.A.M. KAMSTEEG-KEMPER, F.E.J. KRUSEMAN ARETZ, W.L. VAN DER POEL, J.P. SCHAAP-KRUSEMAN, M.V. WILKES & G. ZOUTENDIJK, *MC-25 Informatica Symposium*, 1971. ISBN 90 6196 065 7.

MCT 38 W.A. VERLOREN VAN THEMAAT, *Automatic analysis of Dutch compound words*, 1971. ISBN 90 6196 073 8.

MCT 39 H. BAVINCK, *Jacobi series and approximation*, 1972. ISBN 90 6196 074 6.

MCT 40 H.C. TIJMS, *Analysis of (s,S) inventory models*, 1972. ISBN 90 6196 075 4.

MCT 41 A. VERBEEK, *Superextensions of topological spaces*, 1972. ISBN 90 6196 076 2.

MCT 42 W. VERVAAT, *Success epochs in Bernoulli trials (with applications in number theory)*, 1972. ISBN 90 6196 077 0.

MCT 43 F.H. RUYMGAART, *Asymptotic theory of rank tests for independence*, 1973. ISBN 90 6196 081 9.

MCT 44 H. BART, *Meromorphic operator valued functions*, 1973. ISBN 90 6196 082 7.

MCT 45 A.A. BALKEMA, *Monotone transformations and limit laws*, 1973. ISBN 90 6196 083 5.

MCT 46 R.P. VAN DE RIET, *ABC ALGOL, A portable language for formula manipulation systems, part 1: The language*, 1973. ISBN 90 6196 084 3.

MCT 47 R.P. VAN DE RIET, *ABC ALGOL, A portable language for formula manipulation systems, part 2: The compiler*, 1973. ISBN 90 6196 085 1.

MCT 48 F.E.J. KRUSEMAN ARETZ, P.J.W. TEN HAGEN & H.L. OUDSHOORN, *An ALGOL 60 compiler in ALGOL 60, Text of the MC-compiler for the EL-X8*, 1973. ISBN 90 6196 086 X.

MCT 49 H. KOK, *Connected orderable spaces*, 1974. ISBN 90 6196 088 6.

MCT 50 A. VAN WIJNGAARDEN, B.J. MAILLOUX, J.E.L. PECK, C.H.A. KOSTER, M. SINTZOFF, C.H. LINDSEY, L.G.L.T. MEERTENS & R.G. FISKER (Eds), *Revised report on the algorithmic language ALGOL 68*, 1976. ISBN 90 6196 089 4.

MCT 51 A. HORDIJK, *Dynamic programming and Markov potential theory*, 1974. ISBN 90 6196 095 9.

MCT 52 P.C. BAAYEN (ed.), *Topological structures*, 1974. ISBN 90 6196 096 7.

MCT 53 M.J. FABER, *Metrizability in generalized ordered spaces*, 1974. ISBN 90 6196 097 5.

MCT 54 H.A. LAUWERIER, *Asymptotic analysis, part 1*, 1974. ISBN 90 6196 098 3.

MCT 55 M. HALL JR. & J.H. VAN LINT (Eds), *Combinatorics, part 1: Theory of designs, finite geometry and coding theory*, 1974. ISBN 90 6196 099 1.

MCT 56 M. HALL JR. & J.H. VAN LINT (Eds), *Combinatorics, part 2: graph theory, foundations, partitions and combinatorial geometry*, 1974. ISBN 90 6196 100 9.

MCT 57 M. HALL JR. & J.H. VAN LINT (Eds), *Combinatorics, part 3: Combinatorial group theory*, 1974. ISBN 90 6196 101 7.

MCT 58 W. ALBERS, *Asymptotic expansions and the deficiency concept in statistics*, 1975. ISBN 90 6196 102 5.

MCT 59 J.L. MIJNHEER, *Sample path properties of stable processes*, 1975. ISBN 90 6196 107 6.

MCT 60 F. GÖBEL, *Queueing models involving buffers*, 1975. ISBN 90 6196 108 4.

* MCT 61 P. VAN EMDE BOAS, *Abstract resource-bound classes, part 1*. ISBN 90 6196 109 2.

* MCT 62 P. VAN EMDE BOAS, *Abstract resource-bound classes, part 2*. ISBN 90 6196 110 6.

MCT 63 J.W. DE BAKKER (ed.), *Foundations of computer science*, 1975. ISBN 90 6196 111 4.

MCT 64 W.J. DE SCHIPPER, *Symmetric closed categories*, 1975. ISBN 90 6196 112 2.

MCT 65 J. DE VRIES, *Topological transformation groups 1 A categorical approach*, 1975. ISBN 90 6196 113 0.

MCT 66 H.G.J. PIJLS, *Locally convex algebras in spectral theory and eigenfunction expansions*, 1976. ISBN 90 6196 114 9.

* MCT 67 H.A. LAUWERIER, *Asymptotic analysis, part 2.*
ISBN 90 6196 119 X.

MCT 68 P.P.N. DE GROEN, *Singularly perturbed differential operators of second order*, 1976. ISBN 90 6196 120 3.

MCT 69 J.K. LENSTRA, *Sequencing by enumerative methods*, 1977.
ISBN 90 6196 125 4.

MCT 70 W.P. DE ROEVER JR., *Recursive program schemes: semantics and proof theory*, 1976. ISBN 90 6196 127 0.

MCT 71 J.A.E.E. VAN NUNEN, *Contracting Markov decision processes*, 1976.
ISBN 90 6196 129 7.

MCT 72 J.K.M. JANSEN, *Simple periodic and nonperiodic Lamé functions and their applications in the theory of conical waveguides*, 1977.
ISBN 90 6196 130 0.

MCT 73 D.M.R. LEIVANT, *Absoluteness of intuitionistic logic*, 1979.
ISBN 90 6196 122 x.

MCT 74 H.J.J. TE RIELE, *A theoretical and computational study of generalized aliquot sequences*, 1976. ISBN 90 6196 131 9.

MCT 75 A.E. BROUWER, *Treelike spaces and related connected topological spaces*, 1977. ISBN 90 6196 132 7.

MCT 76 M. REM, *Associons and the closure statement*, 1976. ISBN 90 6196 135 1.

MCT 77 W.C.M. KALLENBERG, *Asymptotic optimality of likelihood ratio tests in exponential families*, 1977 ISBN 90 6196 134 3.

MCT 78 E. DE JONGE, A.C.M. VAN ROOIJ, *Introduction to Riesz spaces*, 1977.
ISBN 90 6196 133 5.

MCT 79 M.C.A. VAN ZUIJLEN, *Empirical distributions and rankstatistics*, 1977.
ISBN 90 6196 145 9.

MCT 80 P.W. HEMKER, *A numerical study of stiff two-point boundary problems*, 1977. ISBN 90 6196 146 7.

MCT 81 K.R. APT & J.W. DE BAKKER (eds), *Foundations of computer science II*, part I, 1976. ISBN 90 6196 140 8.

MCT 82 K.R. APT & J.W. DE BAKKER (eds), *Foundations of computer science II*, part II, 1976. ISBN 90 6196 141 6.

* MCT 83 L.S. VAN BENTEM JUTTING, *Checking Landau's "Grundlagen" in the AUTOMATH system*, ISBN 90 6196 147 5.

MCT 84 H.L.L. BUSARD, *The translation of the elements of Euclid from the Arabic into Latin by Hermann of Carinthia (?) books vii-xii*, 1977.
ISBN 90 6196 148 3.

MCT 85 J. VAN MILL, *Supercompactness and Wallman spaces*, 1977.
ISBN 90 6196 151 3.

MCT 86 S.G. VAN DER MEULEN & M. VELDHORST, *Torrix I*, 1978.
ISBN 90 6196 152 1.

* MCT 87 S.G. VAN DER MEULEN & M. VELDHORST, *Torrix II*,
ISBN 90 6196 153 x.

MCT 88 A. SCHRIJVER, *Matroids and linking systems*, 1977.
ISBN 90 6196 154 8.

MCT 89 J.W. DE ROEVER, *Complex Fourier transformation and analytic functionals with unbounded carriers*, 1978. ISBN 90 6196 155 6.

* MCT 90 L.P.J. GROENEWEGEN, *Characterization of optimal strategies in dynamic games*, . ISBN 90 6196 156 4.

* MCT 91 J.M. GEYSEL, *Transcendence in fields of positive characteristic*, . ISBN 90 6196 157 2.

* MCT 92 P.J. WEEDA, *Finite generalized Markov programming*, ISBN 90 6196 158 0.

MCT 93 H.C. TIJMS (ed.) & J. WESSELS (ed.), *Markov decision theory*, 1977. ISBN 90 6196 160 2.

MCT 94 A. BIJLSMA, *Simultaneous approximations in transcendental number theory*, 1978 . ISBN 90 6196 162 9.

MCT 95 K.M. VAN HEE, *Bayesian control of Markov chains*, 1978. ISBN 90 6196 163 7.

* MCT 96 P.M.B. VITÁNYI, *Lindenmayer systems: structure, languages, and growth functions*, . ISBN 90 6196 164 5.

* MCT 97 A. FEDERGRUEN, *Markovian control problems; functional equations and algorithms*, . ISBN 90 6196 165 3.

MCT 98 R. GEEL, *Singular perturbations of hyperbolic type*, 1978. ISBN 90 6196 166 1

MCT 99 J.K. LENSTRA, A.H.G. RINNOOY KAN & P. VAN EMDE BOAS, *Interfaces between computer science and operations research*, 1978. ISBN 90 6196 170 X.

MCT 100 P.C. BAAYEN, D. VAN DULST & J. OOSTERHOFF (Eds), *Proceedings bicentennial congress of the Wiskundig Genootschap, part 1*, 1979. ISBN 90 6196 168 8.

MCT 101 P.C. BAAYEN, D. VAN DULST & J. OOSTERHOFF (Eds), *Proceedings bicentennial congress of the Wiskundig Genootschap, part 2*, 1979. ISBN 90 9196 169 6.

MCT 102 D. VAN DULST, *Reflexive and superreflexive Banach spaces*, 1978. ISBN 90 6196 171 8.

MCT 103 K. VAN HARN, *Classifying infinitely divisible distributions by functional equations*, 1978 . ISBN 90 6196 172 6.

* MCT 104 J.M. VAN WOUWE, *Go-spaces and generalizations of metrizability*, . ISBN 90 6196 173 4.

* MCT 105 R. HELMERS, *Edgeworth expansions for linear combinations of order statistics*, . ISBN 90 6196 174 2.

AN ASTERISK BEFORE THE NUMBER MEANS "TO APPEAR"