OXFORD UNIVERSITY COMPUTING LABORATORY
PROGRAMMING RESEARCH GROUP
45 BANBURY ROAD
OXFORD

M. Clint.

for inclusion in file.

Department of Computer Science,

University College of Swansea.

Computers and Logic Research Group.

Memorandum No. 14: October 1970

These are informal working papers.

A FORMAL NOTION OF SIMULATION BETWEEN PROGRAMS

R. Milner

This work was carried out under a grant from the Science Research Council.

A FORMAL NOTION OF SIMULATION BETWEEN PROGRAMS

Sections 1. Introduction. 2. An illustration of program verification. 3. Simulation; a formal treatment. 4. An illustration of program simulation. 5. Discussion and conclusion.

1. INTRODUCTION

Programmers often choose an unnatural representation for their data objects, even when a more natural choice exists in the language they are using. Their reason may be that they do not wish to incur the space/time overheads of the more natural choice of data types, particularly when they only use frequently a subset of the operations available over those types — and when that subset is efficiently carried out in terms of a less obvious choice of data representation.

An example of this is in the storage of clauses in a resolution-based theorem-prover. Clauses are naturally sets of formulae, and a formula is naturally a tree or nested list. But because (i) clauses once formed are not normally altered or extended, and (ii) the frequent operations on clauses — e.g. unification, substitution — are efficiently done on a string or vector representation, one would not except for pedagogic reasons use for clauses the data types set, list even in a language which had them.

Whatever the reason for choosing an unnatural

representation, programmers continue to think in terms of the natural data objects too; it follows that they are aware of the mapping — or whatever it is — that exists between them, however implicitly. Part of the purpose of this paper is to formulate whatever it is explicitly: we show how the real program simulates (in a precise sense) the ideal program that never got written.

Motivation comes from the fact that proving that an 'unnatural' program works — a task certainly too hard for complete mechanisation at present except for the most trivial cases — or even expressing what it is supposed to do, appears to involve making the simulation at least partly explicit.

We start in section 2 by illustrating this last remark. In section 3 we define simulation formally, and derive some consequences of the definition. In section 4 we continue with the example of section 2, in the light of the theory.

I hope that the theory of section 3 also has some independent interest. For one thing, it is related to some concepts in automata theory (see the remarks after Theorem 1). For another, since the simulation relation defined is a quasi-ordering, it gives rise to certain equivalence classes of program. These classes provide a tentative answer to the question "What is the algorithm expressed by a program?" (see the end of section 3.1).

2. AN ILLUSTRATION OF PROGRAM VERIFICATION

The program of Figure 1 is supposed, given a string σ in elements $1, 2, \cdots, hl-1$ of character array s, a string τ in elements $1, 2, \cdots, kl-1$ of array t, and a character c in x, to generate in elements $1, 2, \cdots, hh-1$ of array ss a string consisting of σ with all occurrences of c replaced by τ. The verification of a similar program was discussed in detail in Milner [1] (it was in fact part of a resolution-based theorem-prover written in POP2), but we repeat what we need of the discussion here.

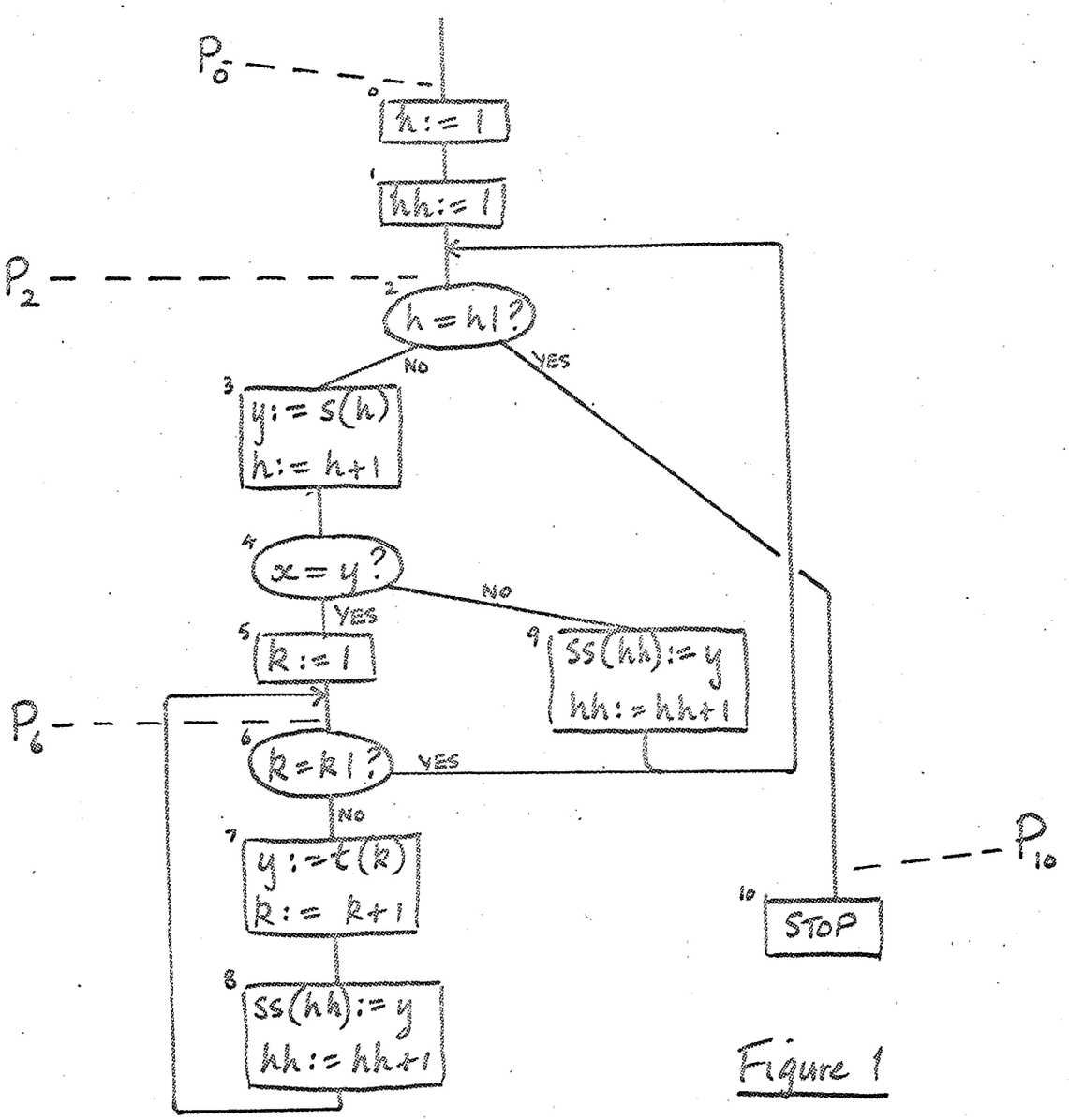


Figure 1

Floyd's [2] method of program verification (also presented formally in section 3) is as follows:
Suppose the state vector of the program is $\underline{x}$. Attach assertions — i.e. truth-valued formulae with free variables x_i — to sufficient arcs of the program that there are no assertion-free cycles, and also assertions $P_0(\underline{x})$, $P_N(\underline{x})$ to entry and exit.
If for every assertion-free path π between two assertions P_i, P_j we can prove

$$P_i(\underline{x}) \wedge \pi(\underline{x}) \Longrightarrow P_j(f_\pi(\underline{x})) \qquad ---(1)$$

where $\pi(\underline{x})$ is the condition that path π will be followed starting with state $\underline{x}$ and f_π is the function which changes the state along π,
Then for any input satisfying P_0 the program will compute output (if any) satisfying P_N. Others have discussed this method; see for example Cooper [3]. This technique does not prove termination: Floyd gives another technique for this, and Manna [4,5] and Burstall [6] deal with both verification and termination. The present Floyd technique however is particularly close to the notion of simulation of programs defined in section 3.

To formulate the correctness criterion $P_N(\underline{x})$ for the program in figure 1 we need to make precise the first sentence of the present section, which is about "strings held in segments of arrays". It is natural to introduce a function

seq: arrays × integers × integers ⟶ strings

such that for $i \leq j$, $seq(a,i,j)$ is the string of characters $a(i), a(i+1), \cdots a(j-1)$, and for $i > j$ seq is arbitrarily

null

defined (to keep it Total), and a function

$$Sub : strings \times characters \times strings \longrightarrow strings$$

such that $Sub(\sigma, c, \tau)$ is the result of replacing c by τ everywhere in σ. Now the correctness criterion is succinctly written

$$P_{10} : \quad seq(ss, 1, hh) = Sub\left(seq(s, 1, h1), x, seq(t, 1, k1)\right)$$

(one may also add as extra conjuncts $1 \le hh, 1 \le h1, 1 \le k1$), whereas without seq and sub the criterion would be lengthy. Of course the price to be paid is some axioms or assumptions about seq and sub to prove the appropriate implications (1). We leave it to the reader to ascertain that with

$$P_0 : \quad 1 \le h1 \ \wedge \ 1 \le k1$$

$$P_2 : \quad seq(ss, 1, hh) = sub\left(seq(s, 1, h), x, seq(t, 1, k1)\right) \ \wedge \ 1 \le h \le h1 \ \wedge \ 1 \le hh \ \wedge \ 1 \le k1$$

$$P_6 : \quad seq(ss, 1, hh) <> seq(t, k, k1) = sub\left(seq(s, 1, h), x, seq(t, 1, k1)\right) \ \wedge \ 1 \le h \le h1 \ \wedge \ 1 \le k \le k1 \ \wedge \ 1 \le hh$$

all the appropriate instances of (1) follow, with the following assumptions (<> is concatenation, ε is string identity):

$$\left\{\begin{array}{l} seq(a, i, i+1) = a(i) \\ i \le j \Rightarrow \left(seq(a, i, j) = \varepsilon \Leftrightarrow i = j\right) \\ i \le j \le k \Rightarrow \left(seq(a, i, j) <> seq(a, j, k) = seq(a, i, k)\right) \\ Sub(\varepsilon, c, \tau) = \varepsilon \\ Sub(c, c, \tau) = \tau \\ d \ne c \Rightarrow Sub(d, c, \tau) = d \quad [d \text{ a character}] \\ Sub(\sigma_1 <> \sigma_2, c, \tau) = Sub(\sigma_1, c, \tau) <> Sub(\sigma_2, c, \tau) \end{array}\right. \tag{2}$$

For example, it must be proved that

$$P_2 \ \wedge \ h \ne h1 \ \wedge \ x = s(h) \Longrightarrow P_6\left[h+1/h, s(h)/y, 1/k\right]$$

(2)

where the square brackets indicate simultaneous substitutions.

If the programmer has in mind, while writing this program, an 'ideal' program with a similar flowchart and with string variables, then the values assumed by the latter will roughly correspond to the values of the 'seq(....)' subexpressions in the assertions. In fact one may hope for a kind of homomorphism from the statespace of the real program to that of the ideal program, since the latter's variables take values which are functions of those in the former. In the next section we see exactly to what extent this is true; and that in general the requirement of a homomorphism is unnecessary, and indeed unlikely to be satisfied.

? ||

However, independently of what follows the present example shows that program verification is most naturally done in terms of the natural data objects. I have found this true in the only other two programs I have examined in which the natural data objects were not directly represented: in one case they were records and sets of records, and in the other case a special subclass of algebraic expressions.

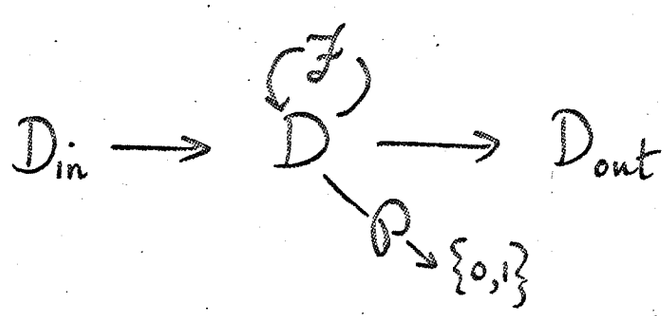
3. Simulation; A formal treatment

3.1. Definitions

We now introduce a formal notion of simulation between programs and prove some properties of it.

First, some notational conventions. If $R \subseteq A \times B$ is a relation between the domains A, B we write $R(a,b)$ for "a and b are in the relation R". Denote $\{b \mid R(a,b)\}$ by $R(a)$, and for $A_1 \subseteq A$ denote $\{b \mid \exists a \in A_1 . R(a,b)\}$ by $R(A_1)$. Denote the inverse relation of R by R^{-1}. Denote $R(A), R^{-1}(B)$ by $\operatorname{ran} R$, $\operatorname{dom} R$ — the range and domain of R. We consider partial functions $A \to B$ as relations. If f is a partial function we allow $f(a)$ which is at most a singleton set also to denote the member (if any) of that set. For $R_1 \subseteq A \times B$, $R_2 \subseteq B \times C$ denote $\{\langle a,c \rangle \mid \exists b (R_1(a,b) \,\&\, R_2(b,c))\}$ by $R_1 \circ R_2$. Denote the identity relation on any set A by I_A.

Definition A machine $\mathcal{M} = \langle D_{in}, f_{in}, D, f_{out}, D_{out}, \mathcal{F}, P \rangle$, which may be represented



consists of

An input domain D_{in}
A computation domain D
An output domain D_{out}
An input function $f_{in} : D_{in} \to D$
An output function $f_{out} : D \to D_{out}$
A family of computation functions $\mathcal{F} \subseteq D \to D$.
A family of predicates $\mathcal{P} \subseteq D \to \{0,1\}$

we really need multiply families of input/output fns.

For this paper we regard f_{in}, f_{out}, $f \in \mathcal{F}$, $p \in \mathcal{P}$ as total on their domains, and single valued — i.e. we are concerned with deterministic programs.

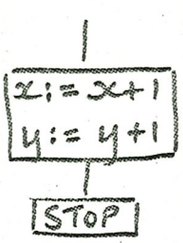
This definition of machine follows de Bakker and Scott [7].

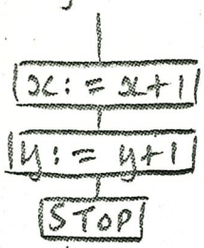
Definition A program scheme $\mathcal{S} = \langle N, \mathcal{G} \rangle$ is a directed graph $\mathcal{G}$ with $N+1$ nodes labelled $0, 1, \ldots, N$. Node 0 is the entry node. For $0 \leq n < N$ node n either is an f-node and has one successor node denoted n^+ or is a p-node and has two successors denoted n^+ and n^-. Node N is the exit node and has no successors.

When refering to a component of a particular machine M or program scheme $\mathcal{S}$ we use a subscript, e.g. $f_{in}^{(M)}$ or $\mathcal{G}^{(\mathcal{S})}$.

Definition A program $\mathcal{A} = \langle \mathcal{S}, M, K \rangle$ is a program scheme $\mathcal{S}$, a machine M, and an assignment K to each node n $(0 \leq n < N^{(\mathcal{S})})$ of either a function $f_n \in \mathcal{F}^{(M)}$ if n is an f-node or a predicate $p_n \in \mathcal{P}^{(M)}$ if n is a p-node. The $\langle M, K \rangle$ part of $\mathcal{A}$ may be called the interpretation; thus a program is an interpreted program scheme.

In the programs that we consider below D will be the possible states of the program variables and a member of $\mathcal{F}$ will normally be an assignment statement or group of assignment statements. One point is worth noting; the programs roughly indicated by the following two diagrams





are formally distinct by our definition. They have different schemes S and assignments K. We require this distinction to formulate the definition of simulation.

From now on to avoid superscripts we assume that $\mathcal{A} = \langle S, \mathcal{M}, K \rangle$ where $S = \langle N, \xi \rangle$ and $\mathcal{M} = \langle D_{in}, f_{in}, D, f_{out}, D_{out}, \mathcal{F}, P \rangle$ and that $\mathcal{A}'$ is the same with primes throughout.

Definition The computation of $\mathcal{A}$ under $x \in D_{in}$ is either the infinite sequence

$$x, \langle n_0, d_0 \rangle, \langle n_1, d_1 \rangle, \ldots\ldots$$

or the finite sequence

$$x, \langle n_0, d_0 \rangle, \langle n_1, d_1 \rangle, \ldots\ldots, \langle n_k, d_k \rangle, y$$

where $n_0 = 0$, $d_0 = f_{in}(x)$ and in the second case $n_k = N$, $y = f_{out}(d_k)$ while for all other n_j, $0 \leq n_j < N$ and either n_j is an f-node and $\langle n_{j+1}, d_{j+1} \rangle = \langle n_j^+, f_{n_j}(d_j) \rangle$ or n_j is a p-node and $\langle n_{j+1}, d_{j+1} \rangle = \langle n_j^-, d_j \rangle$ or $\langle n_j^+, d_j \rangle$ according as $p_{n_j}(d_j) = 0$ or 1.

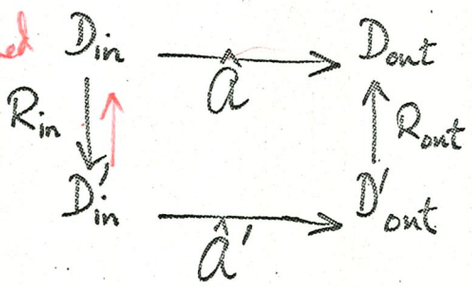
Definition The execution path of $\mathcal{A}$ under x is the infinite or finite node sequence

$$n_0, n_1, \ldots\ldots$$

occurring in the computation under x.

Definition The partial function $\hat{\mathcal{A}} : D_{in} \to D_{out}$ of program $\mathcal{A}$ is the set of $\langle x, y \rangle$ which begin and end a finite computation of $\mathcal{A}$.

Definition If $R_{in} : D_{in} \to D'_{in}$ and $R_{out} : D'_{out} \to D_{out}$ are single valued and R_{in} is total, then $\mathcal{A}'$ covers $\mathcal{A}$ w.r.t. R_{in}, R_{out} if $R_{in} \circ \hat{\mathcal{A}}' \circ R_{out} = \hat{\mathcal{A}}$, i.e. the following diagram commutes:



R_{in} should be defined the other way round (or be multivalued)

From now on we assume the stated restrictions on R_{in}, R_{out} wherever they are mentioned. If $\mathcal{A}'$ covers $\mathcal{A}$ w.r.t some such pair, then $\mathcal{A}'$ covers $\mathcal{A}$.

If $\mathcal{A}'$ covers $\mathcal{A}$, then it is at least as powerful as $\mathcal{A}$ in the sense that we can "use it to do $\mathcal{A}$" with suitable input/output conversion. But we are interested in a more refined relation than covering, because we want to include the idea of following the same execution path in order to formalize the relation between real and ideal program suggested in Section 2.

NO.

I prefer covering to simulation

Definition $\mathcal{A}'$ simulates $\mathcal{A}$ w.r.t R_{in}, R_{out} if

(i) $S = S'$

(ii) $\mathcal{A}'$ covers $\mathcal{A}$ w.r.t. R_{in}, R_{out}

(iii) The execution paths of $\mathcal{A}$ under x, $\mathcal{A}'$ under $R_{in}(x)$ are identical for all $x \in D_{in}$

If $\mathcal{A}'$ simulates $\mathcal{A}$ w.r.t some R_{in}, R_{out} then $\mathcal{A}'$ simulates $\mathcal{A}$.

The restriction that $\mathcal{A}, \mathcal{A}'$ must have the same program scheme is not so stringent as it appears; note that the assigned f's and p's may differ widely in complexity between $\mathcal{A}$ and $\mathcal{A}'$. What S does is to indicate the level of detail to which the simulation is faithful. In other words, if S is big and the assigned f's and p's are simple then the simulation is detailed: if S is small and the f's and p's are complex then it is coarse. In the limit S may have only one f-node, so that $\hat{\mathcal{A}} = f_{in} \circ f_0 \circ f_{out}$ and $\hat{\mathcal{A}}' = f'_{in} \circ f'_0 \circ f'_{out}$ where f_0, f'_0 may be as complex as we wish; here simulation has reduced to mere covering, loosely speaking. this is sad: we want one to be simple and the other to be complex.

The decision to require Total computation functions and predicates should be justified. First, it simplifies the discussion of execution paths (they must either terminate with output or be infinite); second, a rather less appealing definition of simulation is required to make the following theorem, or a variant of it, go through for partial functions; and third, even if the computation functions used in practice are complex — e.g. subroutines — it is good programming practice to arrange that they terminate somehow (even Theorem provers have space bounds!).

Formally, there are one or two simple consequences of the definition of simulation. Programs as defined here are the objects of a category with pairs $\langle R_{in}, R_{out} \rangle$ as morphisms, under the composition ($\cdot$) rule

$$\langle R_{in}^{(1)}, R_{out}^{(1)} \rangle \cdot \langle R_{in}^{(2)}, R_{out}^{(2)} \rangle = \langle R_{in}^{(1)} \circ R_{in}^{(2)}, R_{out}^{(2)} \circ R_{out}^{(1)} \rangle .$$

A related remark is that programs form a quasi-ordering under simulation, since transitivity is obvious. Suppose we restrict R_{in}, R_{out} to be identity relations, so that

$$A' \text{ simulates } A \text{ w.r.t. } R_{in}, R_{out} \Longrightarrow D_{in} = D'_{in}, D_{out} = D'_{out} .$$

Simulation is now symmetric — and is always reflexive clearly — so it is now an equivalence relation. If the algorithm expressed by a program is regarded as a property of the program, it may be argued that it is invariant under the above equivalence relation. Put another way: if algorithms are equivalence classes under some equivalence relation on programs, then the latter relation includes (is implied by) the former.

wrong for practical purposes.

no — the "abstractest" machine should be a canonical form, with all concrete realisations subordinate to it.

3.2 Theory. The following Theorem, which is the main one, throws further light on the definition of simulation, and also prepares the way for proof of simulation in particular cases, which in turn allows a simulating (real) program to be proved correct by proving the simulated (ideal) program correct.

Theorem 1 (The Simulation Theorem). If $S = S'$, then $\mathcal{A}'$ simulates $\mathcal{A}$ w.r.t R_{in}, R_{ont} if and only if there is a set of relations $\{R_n \mid 0 \in n \in N\}$, $R_n \subseteq D \times D'$, such that the following hold:

$$R_{in} \circ f'_{in} \subseteq f_{in} \circ R_0 , \qquad \text{---- (1)}$$

For $0 \in n < N$ $\begin{cases} R_n \circ f'_n \subseteq f_n \circ R_{n^+} \text{ if } n \text{ is an f-node} \quad \text{... (2f)} \\ R_n \circ I_{p'_n} \subseteq I_{p_n} \circ R_{n^+} \text{ and} \\ \qquad R_n \circ I_{\sim p'_n} \subseteq I_{\sim p_n} \circ R_{n^-} \text{ if } n \text{ is a p-node} \end{cases}$ ... (2p)

and $\quad R_N \circ f'_{font} \subseteq f_{font} \circ R_{ont}^{-1} \qquad \text{---- (3)}$

[Note: for a predicate p we use $p, \sim p$ to stand for $\{x \mid p(x)\}, \{x \mid \sim p(x)\}$]

Proof ($\Longrightarrow$) Assume $\mathcal{A}'$ simulates $\mathcal{A}$ w.r.t. R_{in}, R_{ont}.
Take R_n $(0 \in n \in N)$ to be the set of d, d' such that for some $x \in D_{in}$, $\langle n, d\rangle$ and $\langle n, d'\rangle$ occur at corresponding positions in the pair of computations of $\mathcal{A}$ on x, $\mathcal{A}'$ on $R_{in}(x)$ - remembering that these computations have the same execution path.

To prove (2f) — (1) is similar — let $\langle d, e'\rangle \in R_n \circ f'_n$, n an f-node. Then $\exists d'. R_n(d,d') \,\&\, e' = f'_n(d')$ so $\langle n, d\rangle, \langle n, d'\rangle$ correspond in some computation pair (as above), and $\langle n^+, f_n(d)\rangle, \langle n^+, f'_n(d')\rangle$ also correspond, so $\langle f_n(d), e'\rangle = \langle f_n(d), f'_n(d')\rangle \in R_{n^+}$, i.e $\langle d, e'\rangle \in f_n \circ R_{n^+}$. Hence (2f).

To prove (2p) — let $\langle d, d'\rangle \in R_n \circ I_{p'_n}$, that is $R_n(d,d')$ and $p'_n(d')$. So $\langle n, d\rangle, \langle n, d'\rangle$ correspond as before in a computation pair with identical execution paths, and as the successor of $\langle n, d'\rangle$ is $\langle n^+, d'\rangle$ because of $p'_n(d')$, the successor of $\langle n, d\rangle$ is $\langle n^+, d\rangle$, which requires $p_n(d)$, and it follows that $R_{n^+}(d,d')$

So $\langle d,d'\rangle \in I_{fn} \circ R_{n+}$. The second of (2f) is similar.

For (3), suppose $\langle d, y'\rangle \in R_N \circ f'_{out}$, so that $\exists d'$. $R_N(d,d')$ & $y' = f'_{out}(d')$. So from the definition of R_N, $\exists x$. $f_{out}(d) = \hat{a}(x) = R_{out}(\hat{a}'(R_{in}(x)))$ (by covering)
$= R_{out}(f'_{out}(d')) = R_{out}(y')$. So $\langle d,y'\rangle \in f_{out} \circ R_{out}^{-1}$.

($\Leftarrow$) Assume (1)–(3). We first prove inductively that the computations of a under x, a' under $x' = R_{in}(x)$ have the same execution path, and that if $\langle n,d\rangle, \langle n,d'\rangle$ correspond in the two computations then $R_n(d,d')$.

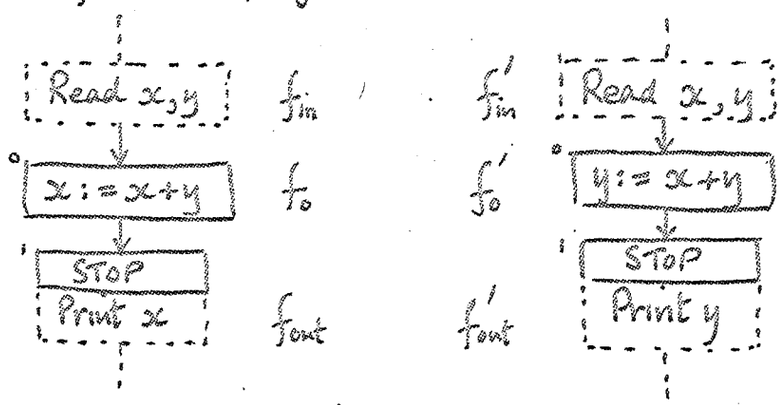
The computations start $x, \langle 0, f_{in}(x)\rangle, \ldots$ and $x', \langle 0, f'_{in}(x')\rangle, \ldots$ and (1) ensures $R_0(f_{in}(x), f'_{in}(x'))$.

Suppose the execution paths agree up to and including the corresponding members $\langle n,d\rangle, \langle n,d'\rangle$ and that $R_n(d,d')$. Then for an f-node n, (2f) ensures that for their successors $\langle n^+, f_n(d)\rangle, \langle n^+, f'_n(d')\rangle$ $R_{n+}(f_n(d), f'_n(d'))$ also holds. For a p-node n, (2p) ensures that either their successors are $\langle n^+, d\rangle, \langle n^+, d'\rangle$ and $R_{n+}(d,d')$ or their successors are $\langle n^-, d\rangle, \langle n^-, d'\rangle$ and $R_{n-}(d,d')$.

So we conclude that the execution paths agree either as infinite sequences, or up to some occurrence of N, when $\langle N,d\rangle$ $\langle N,d'\rangle$ correspond in the computations for some $\langle d,d'\rangle \in R_N$. In the former case both $\hat{a}(x)$ and $\hat{a}'(x')$ are undefined: in the latter case (3) gives $R_{out}(f'_{out}(d')) = f_{out}(d)$, so $R_{out}(\hat{a}'(R_{in}(x))) = \hat{a}(x)$. Summarizing both cases, $R_{in} \circ \hat{a}' \circ R_{out} = \hat{a}$, i.e. a' covers a, and we have proved all of the simulation conditions (i)–(iii).

∎

The conditions (1) – (3) suggest the relation of weak homomorphism between finite automata (see Ginzsburg [8], p98). The main difference is in our allowing many relations R_n instead of a single one. Is this too weak for a definition of program simulation? It seems not; even for the following two programs



it is easy to prove that though they simulate one another (w.r.t identity functions R_{in}, R_{out}), R_0, R_1 cannot be identical. It is of course the slight difference in store use which spoils it; but they are remarkably similar programs!

Definition Call $\langle R_{in}, \{R_n\}, R_{out} \rangle$ a simulation of $\mathcal{A}$ by $\mathcal{A}'$, if they satisfy (1) – (3).

Theorem 2. If $\langle R_{in}, \{R_n\}, R_{out} \rangle$, $\langle R'_{in}, \{R'_n\}, R'_{out} \rangle$ are simulations of $\mathcal{A}$ by $\mathcal{A}'$, $\mathcal{A}'$ by $\mathcal{A}''$ respectively, then $\langle R_{in} \circ R'_{in}, \{R_n \circ R'_n\}, R'_{out} \circ R_{out} \rangle$ is a simulation of $\mathcal{A}$ by $\mathcal{A}''$.

Proof Immediate from (1) – (3) ∎

It follows immediately that there is a category with programs as objects and simulations (compounded as in the theorem) as morphisms. Moreover, there is an obvious functor taking this category onto the previous one mentioned

in this section, where the morphisms were $\langle R_{in}, R_{out} \rangle$ pairs.

We now relate simulation to verification of programs.

Definition. Let $S_{in} \subseteq D_{in}$, $S_{out} \subseteq D_{out}$.
A is partially correct w.r.t S_{in}, S_{out} if $\hat{A}(S_{in}) \subseteq S_{out}$
A is totally correct w.r.t. S_{in}, S_{out} if in addition $\hat{A}$ is total on S_{in}.

These are in effect the definitions of Manna [4,5].

The following result comes directly from the definition of simulation, and does not require the Simulation Theorem.

Theorem 3. If A' simulates A w.r.t. R_{in}, R_{out}, and A is partially (totally) correct w.r.t S_{in}, S_{out} then A' is partially (totally) correct w.r.t $R_{in}(S_{in})$, $R_{out}^{-1}(S_{out})$.

Proof First, it follows from simulation that

$$R_{in} \circ \hat{A}' \subseteq \hat{A} \circ R_{out}^{-1} \qquad ---(4)$$

For let $\langle x, d' \rangle \in R_{in} \circ \hat{A}'$. Then A' terminates on $R_{in}(x)$, so (by simulation) A terminates on x. Thus by covering $R_{out}(d') = R_{out}(\hat{A}'(R_{in}(x)))$ is defined and equal to $\hat{A}(x)$, i.e. $\langle x, d' \rangle \in \hat{A} \circ R_{out}^{-1}$. Hence (4).

For partial correctness of A' we need $\hat{A}'(R_{in}(S_{in})) \subseteq R_{out}^{-1}(S_{out})$.
But $\hat{A}'(R_{in}(S_{in})) = (R_{in} \circ \hat{A}')(S_{in}) \subseteq R_{out}^{-1}(\hat{A}(S_{in}))$ by (4)
$\subseteq R_{out}^{-1}(S_{out})$ by partial correctness of A.

In addition, if A is total on S_{in} then by simulation A' is total on $R_{in}(S_{in})$. ∎

The importance of this theorem is that the correctness proof for Q' – the real program – may be factored into two ; proof of correctness of Q – the ideal program – and proof of simulation of Q by Q'.

The next few results relate simulation still more closely with verification.

Definition If $S_{in} \subseteq D_{in}$, $S_n \subseteq D$ $(0 \le n \le N)$, $S_{out} \subseteq D_{out}$ call $\langle S_{in}, \{S_n\}, S_{out} \rangle$ a verification of Q if

$$f_{in}(S_{in}) \subseteq S_0 , \qquad \text{----(5)}$$

For $0 \le n < N$,
$$\begin{cases} f_n(S_n) \subseteq S_{n+} \text{ if } n \text{ is an } f\text{-node} & (6f) \\ I_{p_n}(S_n) \subseteq S_{n+} \text{ and } I_{\neg p_n}(S_n) \subseteq S_{n-} & (6p) \\ \qquad \text{if } n \text{ is a } p\text{-node} \end{cases}$$

and $$f_{out}(S_N) \subseteq S_{out} \qquad \text{-----(7)}$$

The following theorem of Floyd [2] and Manna [4,5] we state without proof (in fact the proof is a natural model for that of the Simulation Theorem) :

Theorem 4. Q is partially correct w.r.t. S_{in}, S_{out} if and only if for some $S_n \subseteq D$ $(0 \le n \le N)$ $\langle S_{in}, \{S_n\}, S_{out} \rangle$ is a verification of Q. ∎

The next three Theorems are natural results whose proofs we omit, because they only require a little algebra.

Theorem 5. If $\langle R_{in}, \{R_n\}, R_{out}\rangle$ is a simulation of $\mathcal{A}$ by $\mathcal{A}'$, then (a) $\langle \mathrm{dom}\, R_{in}\ (= I_{D_{in}}), \{\mathrm{dom}\, R_n\}, \mathrm{ran}\, R_{out}\rangle$ is a verification of $\mathcal{A}$

(b) $\langle \mathrm{ran}\, R_{in}, \{\mathrm{ran}\, R_n\}, \mathrm{dom}\, R_{out}\rangle$ is a verification of $\mathcal{A}'$. ∎

Normally however we would expect to prove simulation with an R_{out} too large for verification purposes.

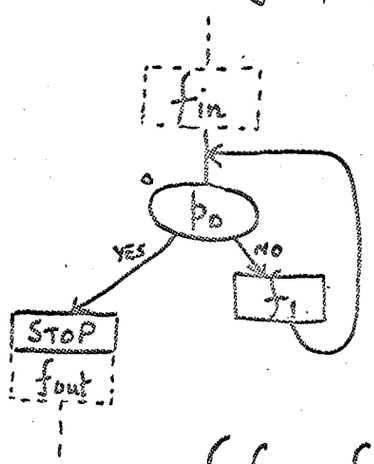
Theorem 6. If $\langle D_{in}, \{S_n\}, S_{out}\rangle$ is a verification of $\mathcal{A}$, then $\langle I_{D_{in}}, \{I_{S_n}\}, I_{S_{out}}\rangle$ is a simulation of $\mathcal{A}$ by $\mathcal{A}$. ∎

Theorem 7. If $\langle R_{in}, \{R_n\}, R_{out}\rangle$ is a simulation of $\mathcal{A}$ by $\mathcal{A}'$ and $\langle S_{in}, \{S_n\}, S_{out}\rangle$ is a verification of $\mathcal{A}$, then $\langle R_{in}(S_{in}), \{R_n(S_n)\}, R_{out}^{-1}(S_{out})\rangle$ is a verification of $\mathcal{A}'$. ∎

This says rather more than Theorem 3, at least as far as partial correctness is concerned; but Theorem 3 contains what is important in practice.

3.3. Practice. To make practical proof of simulation easier, we now show that it is sufficient to find R_n for all $n \in M$, where M is a cycle-breaking subset of $\{0, 1, \ldots, N\}$ – i.e. a subset such that every cycle in S contains a member of M. This is the content of Theorem 9; for completeness we give the analogous result for verification, which is well-known, as Theorem 8.

With each path π_{ij} from i to j $(i,j \in M \cup \{in,out\})$ containing no other member of M — call such paths <u>M-paths</u> — we associate partial functions f_{ij}, f'_{ij} in Q, Q'; for example in the following program



if $M = \{1\}$ we have $\left\{\begin{array}{ll} f_{in,1} = f_{in} \circ I_{\sim p_0} & f_{in,out} = f_{in} \circ I_{p_0} \circ f_{out} \\ f_{1,1} = f_1 \circ I_{\sim p_0} & f_{1,out} = f_1 \circ I_{p_0} \circ f_{out} \end{array}\right\}$

<u>Theorem 8</u> If M is a cycle breaking subset of $\{0,1,\cdots N\}$ and for every M-path π_{ij}, $i,j \in M \cup \{in,out\}$,

$$f_{ij}(S_i) \subseteq S_j \qquad ---(8)$$

where $S_{in} \subseteq D_{in}$, $S_n \subseteq D$ $(n \in M)$ and $S_{out} \subseteq D_{out}$, then we can find $S_n \subseteq D$ $(n \in \{0,1,\cdots N\} - M)$ so that $\langle S_{in}, \{S_n \mid 0 \le n \le N\}, S_{out} \rangle$ is a verification of Q.

<u>Proof</u> The proof is like that of Theorem 9, but simpler. ∎

<u>Theorem 9</u> If M is a cycle breaking subset of $\{0,1,\cdots N\}$ and we are given R_{in}, R_n $(n \in M)$ and R_{out} such that for every M-path π_{ij}, $i,j \in M \cup \{in,out\}$

$$R_i \circ f'_{ij} \subseteq f_{ij} \circ R_j \qquad ---(9)$$

— except that if $j = out$ we replace R_j by R_{out}^{-1} — then we can find $R_n \subseteq D \times D'$ $(n \in \{0,1,\cdots N\} - M)$ so that $\langle R_{in}, \{R_n \mid 0 \le n \le N\}, R_{out} \rangle$ is a simulation of Q by Q'.

Proof. It is enough to show that if $M \neq \{0,1 \cdots N\}$ then we can find an $n \in \{0,1 \cdots N\} - M$ and an R_n so that (9) holds for all $M \cup \{n\}$-paths π_{ij}. For then we can repeat the procedure until all the R_n have been defined, $0 \leq n \leq N$, and then (9) is equivalent to the simulation conditions (1)–(3).

Now if $M \neq \{0,1 \cdots N\}$, there must be a node $n \notin M$ which is either (a) an f-node with successor $n^+ \in M$ or (b) a p-node with successors $n^+, n^- \in M$.

In case (a) there will be conditions (9) of form

$$R_i \circ f'_{in^+} \subseteq f_{in^+} \circ R_{n^+} \qquad \text{---} \quad (10)$$

i.e. $R_i \circ f'_{in} \circ f'_n \subseteq f_{in} \circ f_n \circ R_{n^+}$, whence, since $I_{D'} \subseteq f'_n \circ f'^{-1}_n$ $R_i \circ f'_{in} \subseteq f_{in} \circ (f_n \circ R_{n^+} \circ f'^{-1}_n)$. Now define $R_n = f_n \circ R_{n^+} \circ f'^{-1}_n$, giving

$$R_n \circ f'_n = f_n \circ R_{n^+}, \quad R_i \circ f'_{in} \subseteq f_{in} \circ R_n \qquad \text{---} \quad (11)$$

So by replacing (10) by (11) we obtain conditions (9) for all $M \cup \{n\}$ – paths.

In case (b) there will be pairs of conditions (9) of form

$$R_i \circ f'_{in^+} \subseteq f_{in^+} \circ R_{n^+}, \quad R_i \circ f'_{in^-} \subseteq f_{in^-} \circ R_{n^-} \qquad \text{---} \quad (12)$$

in which $f_{in^+} = f_{in} \circ I_{p_n}$, $f_{in^-} = f_{in} \circ I_{\sim p_n}$ and similarly for f'_{in^+}, f'_{in^-}. Now postmultiplying (12) by $I_{p'_n}, I_{\sim p'_n}$ respectively and taking unions both sides, since $I_{p'_n} \cup I_{\sim p'_n} = I_{D'}$

$$R_i \circ f'_{in} \subseteq f_{in} \circ (I_{p_n} \circ R_{n^+} \circ I_{p'_n} \cup I_{\sim p_n} \circ R_{n^-} \circ I_{\sim p'_n})$$

Now define R_n as the bracketed expression, giving immediately

$$\left.\begin{array}{l} R_n \circ I_{p'_n} \subseteq I_{p_n} \circ R_{n^+}, \; R_n \circ I_{\sim p'_n} \subseteq I_{\sim p_n} \circ R_{n^-} \\ \text{and} \quad R_i \circ f'_{in} \subseteq f_{in} \circ R_n \end{array}\right\} \qquad \text{--} \quad (13)$$

So by replacing (12) by (13) we again get conditions (9) for all $M \cup \{n\}$ – paths. □

4. An illustration of program simulation

We now return to the program of section 2, which reappears as program $\mathcal{A}'$ in figure 2. We show that it simulates program $\mathcal{A}$ shown beside it.

Let C be the character alphabet, C^* the set of strings over C. Then

— $D_{in} = D'_{in} = C^* \times C \times C^*$

— $D = \Sigma^L$ (functions $L \to \Sigma$) where L includes all the identifiers of $\mathcal{A}$ and $\Sigma = C^*$

— $D' = \Sigma'^{L'}$ where L' includes identifiers of $\mathcal{A}'$ and $\Sigma' = C \cup \{\text{arrays}\} \cup \{\text{integers}\}$

— $D_{out} = D'_{out} = C^* \times C \times C^* \times C^*$.

Now for $\sigma \in L$ we write σ_d for $d(\sigma)$ where $d \in D$ — and we omit the subscript when d is intended: similarly for $s \in L'$ we write for $d'(s)$, $s_{d'}$ or just s.

— f_{in}, f_{in}' satisfy:

if $d = f_{in}(\text{str1}, \text{char}, \text{str2})$ then $\sigma_0 = \text{str1}$, $\xi = \text{char}$, $\tau_0 = \text{str2}$

if $d' = f'_{in}(\text{str1}, \text{char}, \text{str2})$ then $\text{seq}_1(s,1,h1) = \text{str1}$, $x = \text{char}$, $\text{seq}_1(t,1,k1) = \text{str2}$.

— $f_{out} = \lambda d.\, \langle \sigma_0, \xi, \tau_0, \sigma\sigma \rangle$

— $f_{out}' = \lambda d'.\, \langle \text{seq}(s,1,h1), x, \text{seq}(t,1,k1), \text{seq}(ss,1,hh) \rangle$

The remaining formal details of $\mathcal{A}, \mathcal{A}'$ — e.g. the scheme $S = S'$ — should be apparent from figure 2.

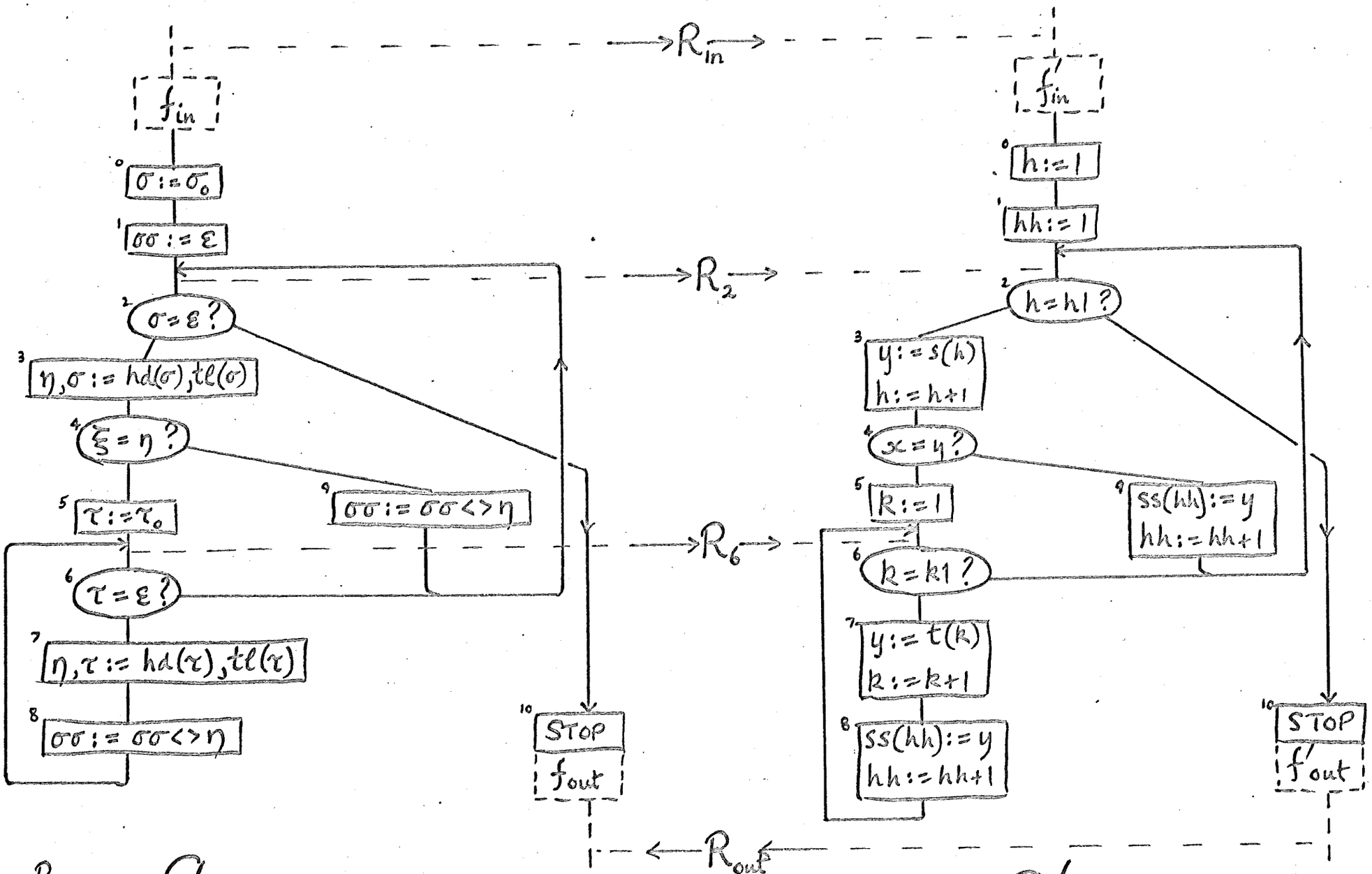

R_{in}
f_{in}
0 $\sigma := \sigma_0$
1 $\sigma\sigma := \varepsilon$
R_2
2 $\sigma = \varepsilon$?
3 $\eta, \sigma := hd(\sigma), tl(\sigma)$
4 $\xi = \eta$?
9 $\sigma\sigma := \sigma\sigma <> \eta$
5 $\tau := \tau_0$
R_6
6 $\tau = \varepsilon$?
7 $\eta, \tau := hd(\tau), tl(\tau)$
8 $\sigma\sigma := \sigma\sigma <> \eta$
10 STOP
f_{out}
R_{out}
Program A
f_{in}
0 $h := 1$
1 $hh := 1$
2 $h = h1$?
3 $y := s(h)$
$h := h+1$
4 $x = y$?
5 $k := 1$
6 $k = k1$?
7 $y := t(k)$
$k := k+1$
8 $ss(hh) := y$
$hh := hh+1$
9 $ss(hh) := y$
$hh := hh+1$
10 STOP
f_{out}
Program A'

Figure 2

To show now that A' simulates A w.r.t $R_{in} = I_{D_{in}}$, $R_{out} = I_{D_{out}}$ we need only supply R_2, R_6 for the cycle-breaking set $\{2,6\}$ and verify conditions (9) of Theorem 9. We give R_2, R_6 as formulae in which the identifiers occur free (we conveniently confuse R_2 and $\{\langle d,d'\rangle \mid R_2\}$):

$$\begin{aligned} R_2: \quad & \sigma_0 = seq(s,1,h1) \ \wedge \\ & \sigma = seq(s,h,h1) \ \wedge \\ & \tau_0 = seq(t,1,k1) \ \wedge \\ & \sigma\sigma = seq(ss,1,hh) \ \wedge \\ & \xi = x \ \wedge \\ & 1 \le h \le h1 \ \wedge \ 1 \le k1 \ \wedge \ 1 \le hh \end{aligned}$$

$$\begin{aligned} R_6: \quad & R_2 \ \wedge \\ & \tau = seq(t,1,k) \ \wedge \\ & \eta = y \ \wedge \\ & 1 \le k \le k1 \end{aligned}$$

Now to prove conditions (9), e.g. $R_2 \circ f'_{26} \subseteq f_{26} \circ R_6$, or

$$R_2 \circ I_{\sim p'_2} \circ f'_3 \circ I_{p'_4} \circ f'_5 \subseteq I_{\sim p_2} \circ f_3 \circ I_{p_4} \circ f_5$$

amounts to proving that

$$R_2 \wedge h \neq h1 \wedge \xi = s(h) \Rightarrow \sigma \neq \varepsilon \wedge \xi = hd(\sigma) \wedge R_6\left[hd(\sigma)/\eta, tl(\sigma)/\sigma, \tau_0/\tau, s(h)/y, h+1/h, 1/k\right]$$

(compare (3) of section 2)

which follows easily from assumptions (2) of section 2 and

$$i < j \Rightarrow hd(seq(a,i,j)) = a(i) \wedge tl(seq(a,i,j)) = seq(a,i+1,j)$$

The remaining conditions (9) are similarly straightforward.

As for verifying program Q, we may easily show that the following S_i satisfy the conditions of Theorem 8, and so may be extended to a verification

$$S_{in} = D_{in}\ ,\ S_2 = \{d \mid \sigma\sigma\langle\rangle\, subt(\sigma,\xi,\tau_0) = subt(\sigma_0,\xi,\tau_0)\}\ ,$$

$$S_6 = \{d \mid \sigma\sigma\langle\rangle\tau\langle\rangle\, subt(\sigma,\xi,\tau_0) = subt(\sigma_0,\xi,\tau_0)\}\ ,$$

$$S_{out} = \{\langle\rho_1,\zeta,\rho_2,\rho_3\rangle \mid \rho_3 = subt(\rho_1,\zeta,\rho_2)\}$$

so Q is partially correct w.r.t S_{in}, S_{out}, and by Theorem 3 Q' is partially correct w.r.t $R_{in}(S_{in}) = S_{in}$, $R_{out}^{-1}(S_{out}) = S_{out}$. Also because simulation is proved, total correctness of Q implies the same for Q'.

5. DISCUSSION AND CONCLUSION

Although the example of section 4 is simple, it illustrates some points of interest. First, we have $R_6 \subseteq R_2$. But we cannot make $R_6 = R_2$: R_6 requires restrictions on σ, t, k, η, y which R_2 does not, and which in general will not hold at node 2 — e.g. on the first iteration. R_6^{-1} is single-valued, since all variables in Q^* are uniquely expressed in terms of those of Q', but it is not total on D because of the restrictions on $h, h1$ etc. It is this roughly many-one relationship, which appears in this example and also between compiled and source programs, which might make one hope for a homomorphism from the real to the ideal program, but we have seen that this is not always possible or necessary.

Another important point is that the R_n need only respect the particular functions and predicates assigned to the nodes, and these may be complex — in the example they are sometimes more than one assignment statement. The simulation is not faithful to the level of the primitive operations (e.g. +, hd, tl) on the data types of the programs in question; it is not even faithful w.r.t. all possible assignment statements built from these primitives.

Some of the assigned $f \in \mathcal{F}$ are suggestive of Naur's [9] action clusters. The pair of instructions ss(hh):=y, hh:=hh+1 in program A' are an example of an action cluster, on the grounds that array ss and integer hh are updated only in this context, apart from the initial hh:=1; the pair of instructions has more meaning for the program than they do taken separately — i.e. together they are simulating the concatenation of a character to a string.

It is doubtful whether a programmer will be persuaded to go to the length of writing his ideal program, verifying it and proving that this already written real program simulates it, in order to verify the real program. Given that he has written the latter, the half-way technique of section 2 (which makes semi-explicit the ideal program and the simulation relations) appears important for verification purposes, but I believe the advance should be towards better high-level languages in which to write the ideal program and specify the simulation relations, from which the compiler will generate the real program. Is it possible, for example, for a programmer to define a data type $\{\begin{matrix}\text{string}\\\text{graph}\end{matrix}\}$, and request

that a variable of this type be represented in one part of the program by an {array segment / incidence matrix} and in another part by a {chained list / linkage of records} ? The work of Standish [10] seems to be a valuable step in this direction.

REFERENCES

1. R. Milner. "The difficulty of verifying a program with unnatural data representation" Memo 3 (1969) Computers and Logic Research Group, Swansea.
2. R.W. Floyd. "Assigning meanings to programs" Proc. Amer. Math. Soc, Symposium in Applied Maths Vol 19 (1967) 19–32
3. D.C. Cooper. "Program Scheme Equivalences and Second Order Logic" Machine Intelligence 4 (1969) 3–15
4. Z. Manna. "The correctness of programs" J. Comp & Sys. Sci 3 (1969) 119–127
5. Z. Manna. "Properties of programs and the first order predicate calculus" JACM 16 (1969) 244–255
6. R.M. Burstall. "Formal Description of program structure and semantics in first order logic" Machine Intelligence 5 (1969) 79–98
7. J.W. de Bakker and D. Scott. "A Theory of Programs" unpublished (1969)
8. A. Ginzburg. "Algebraic Theory of Automata", Academic Press (1968)
9. P. Naur. "Programming by Action Clusters" BIT 9 (1969) 250–258
10. T.A. Standish. "A data definition facility for programming languages" Ph.D. Thesis, Dept. of Computer Science, Carnegie Inst. Tech. (1967)
