INFORMATION AND CONTROL 41, 9–28 (1979)

Programs for Instruction Machines

Z. PAWLAK

Computer Center of the Polish Academy of Sciences, Warsaw, Poland

G. ROZENBERG

Department of Mathematics, University of Antwerp, U.I.A., Wilrijk, Belgium

AND

W. J. SAVITCH*

Computer Science Division, Department of Applied Physics and Information Science, University of California at San Diego, La Jolla, California 92093

Formal models for a computer and for programs are introduced. These models are used to develop a theory for programs based on the underlying computational structure of the computer to be programmed. Several notions of "well-behaved" programs are introduced. Necessary and sufficient conditions for converting arbitrary programs to "well-behaved" programs are derived.

1. INTRODUCTION

One of the goals of theoretical computer science is to provide a systematic understanding of the basic phenomena of information processing. Among the many different approaches to this task, one can distinguish a machine-dependent approach which goes roughly as follows. Since a computer is a central unit in many information-processing systems, it seems reasonable to assume that some of the underlying principles of "how computers operate" are important to the understanding of information processing in general. One natural approach is to try to express these underlying principles of how computers operate through properties of the set of all computations possible on a computer. This philosophy can be expressed as follows: Once a computer is given to us, the set of all possible computations (runs) on it is fixed. The only thing we do when we execute a particular task on a computer (load a program) is to pick out a particular computation from the set of all possible computations.

* Part of this work was done while at the Mathematisch Centrum, Amsterdam. This research was supported, in part, by NSF Grant MCS-74-02338.

0019-9958/79/040009–20$02.00/0

One point of departure in building up a theory along these lines is to model the classical von Neumann concept of a stored program computer with indirect addressing. One such model, among others, was proposed in Pawlak (1969). The underlying idea was to provide a model which would help to explain how the various possible states of a computer relate to one another. In this context, a state is a function from the set of addresses to the set of possible contents of memory locations. One special memory location (the counter) contains the address of the location holding a coding of the current control step (statement) to be executed next. This statement is decoded by the control unit of the computer and executed, resulting in a new state. A computation is then a sequence of states. When observing a particular computation, one can distinguish a sequence of pairs ⟨address, contents⟩ activated in consecutive states. The second element of the pair is, approximately, the statement executed. The set of all such pairs is, in essense, the program determining this computation. To simplify the theory, let us consider only programs without self-modification. In this case, such a program is a subset of every state included in the computation sequence. Now, if we collect all the computation sequences that contain the given program, then we get the set of states associated with the given program. Similarly, with each "statement" of this program, we can associate a set of states; namely, those states which contain this statement in the activated location. In other words, each statement is associated with the set of states which represent the machine about to execute this statement. This leads us to a possible approach to investigating programs. Each program is a set of statements and, in turn, each statement is a set of states. The various statements will thus be pairwise disjoint sets of states.

Thus a natural approach to investigating programs is to forget about the burdensome, technical details of the definition of such address machines and to define a computer to be a set of states with several transition functions. Each transition function corresponds to an instruction (family of instructions). A program is then defined to be a collection of mutually disjoint subsets of the set of states, each subset lying within the domain of a single instruction. This approach is taken in this paper. In particular, we concentrate on the topics which are "well-haved" programs and, when it is possible, to construct such programs. We establish several results indicating that the possibility of writing "nice programs" is dependent on simple properties of the set of all computations available on the machine. This partially justifies our view that a theory for information processing can be based on simple, basic properties of the set of all computations available on a computer.

The order of topics presented is as follows. After introducing the necessary formal definitions, we present necessary and sufficient conditions for a machine to have the property that every program can be converted to an equivalent "well-behaved" program. We then go on to describe canonical forms for certain types of well-behaved programs. Finally, we consider quotient machines obtained by identifying machine states which are in some sense computationally equivalent. The relationship of the programming structure of a machine and its quotient machines is studied under this topic.

2. Basic Definitions

We now formalize our notations of machine and program.

2.1 Definition.

2.1.1. A *P machine* is a pair $M = (S, \pi)$, where S is a set of states and π is a partial function from S to S. Whenever $\pi(s_1) = s_2$, then we say that the machine M goes from states s_1 to s_2 in one step.

2.1.2. A state s is called a *halting state* if $\pi(s)$ is undefined. The set of all halting states of M is denoted HALT(M).

2.1.3. A *computation* of M is a finite or infinite sequence $s_0, s_1, s_2, \ldots$ of states such that $\pi(s_i) = s_{i+1}$ for all i, except the last s_i in case the sequence is finite. A *complete computation* is a computation that either is infinite or is finite and ends with a halting state.

2.1.4. An *instruction machine* (I machine) is a system $M = (S, I_1, I_2, \ldots, I_m)$ such that

(i) S is a set of states (possibly infinite)

(ii) each I_i, $1 \leqslant i \leqslant m$, is a partial function from S to S, and

(iii) for $1 \leqslant i < j \leqslant m$, DOM($I_i$) and DOM($I_j$) are disjoint, where DOM(I_i) denotes the domain of the partial function I_i.

The partial functions I_i are called the *instructions* of the machine M. On an intuitive level, we may think of the I_i as the instructions available on the machine M. So, for example, I_1 might be the plus instruction, I_2 might be the multiplication instruction, I_3 the store instruction, I_4 the fetch instruction and so on. Since we are modeling stored program machines which hold their program in storage, the program and "instruction counter" are part of the machine state. Hence, for any state of the machine there should be at most one applicable instruction. This is the reason for condition (iii) above.

It is worth noting that the I_i are not completely determined by the "hardware" of the machine M, but also represent our way of viewing the machine M. For example, instead of making each arithmetic operation a separate I_i, we might have a single instruction I_i for all arithmetic operations.

2.1.5. The P *machine* associated with the I machine $M = (S, I_1, I_2, ..., I_m)$ is the machine $M' = (S, \pi)$, where π is the union of all the I_i, $1 \leqslant i \leqslant m$. By a computation, complete computation, or halting state of M, we will mean a computation, complete computation, or halting state, respectively, of the associated P machine M'.

Having defined I machines, we now go on to define programs for I machines. Since we are modeling stored-program machines, the program and "instruction counter" are completely determined by the state of the machine. The approach that we will take is that, in this abstract context, a statement of a program is identified with the set of all states which represent this machine, holding this program in storage and about to execute this statement. So a program statement is, for us, just a set of states. Since a program statement is a refinement of a machine instruction (for example a refinement of the store instruction to store in location 20), each statement must lie within a single instruction domain. With this notion of program statement, we can define a program to be a set of statements. There is no need to order the statements of a program or to add any other mechanism for flow of control. The definition of a machine program is such that each state lie in at most one program statement and hence the state will determine which statement is executed next. We now make these ideas formal.

2.2 Definition. Let $M = (S, I_1, I_2, ..., I_m)$ be an I machine and let $M' = (S, \pi)$ be the P machine associated with M.

2.2.1. A *program* for M is a finite collection, $P = \{B_1, B_2, ..., B_n\}$, of non-empty sets of states such that

(i) the B_i are pairwise disjoint,

(ii) for each i, $1 \leqslant i \leqslant n$, either there is an instruction I_j such that B_i is a subset of $\mathrm{DOM}(I_j)$ or else B_i consists entirely of halting states.

The sets B_i are called the *statements* of the program P. If B_i consists entirely of halting states, then B_i is called a *halt statement*.

2.2.2. The *support* of the program P is defined and denoted by $\mathrm{SUP}(P) = \bigcup_{i=1}^{n} B_i$. P is said to be *closed* provided that, for all states S in $\mathrm{SUP}(P)$, either $\pi(s)$ is undefined or $\pi(s)$ is in $\mathrm{SUP}(P)$. Two programs P_1 and P_2 for M are said to be (*computationally*) *equivalent* if $\mathrm{SUP}(P_1) = \mathrm{SUP}(P_2)$. Notice that two programs are equivalent if and only if they realize exactly the same set of computations.

2.2.3. The program P is said to be a *universal program* for the machine M provided $\mathrm{SUP}(P) = S$. Clearly, every I machine has a universal program. For example, $P = \{\mathrm{DOM}(I_1), \mathrm{DOM}(I_2), ..., \mathrm{DOM}(I_m), \mathrm{HALT}(M)\}$ is a universal program for M. This particular universal program P will be called the *natural universal program* for M.

3. Tree Decompositions

In this section, we investigate conditions under which a program can be converted to an equivalent "well-structured" program. By "well-structured" we mean that the program statements can be arranged in a tree-like structure which exhibits the flow of control in a specific, simple and organized way. A formal definition of what we will take to be a "well-structured" program follows.

3.1 Definition. Let $P = \{B_1, B_2, ..., B_m\}$ be a program for an I machine $M = (S, I_1, I_2, ..., I_m)$ and let $M' = (S, \pi)$ be the P machine associated with M.

3.1.1. For any set B of states of M, the *closure*, respectively *exit set*, of B is defined and denoted by $\mathrm{CLOS}(B) = \{s \mid \text{for some } \ell \geqslant 0$, there is a computation $s_0, s_1, ..., s_\ell$ such that s_0 is in B and $s_\ell = s\}$, respectively $\mathrm{EXIT}(B) = \{s \mid s$ is not in B and there is a state s' in B such that $\pi(s') = s.\}$ In the definition of $\mathrm{CLOS}(B)$, the case $\ell = 0$ is to be interpreted to mean s is in B. So B is a subset of $\mathrm{CLOS}(B)$.

3.1.2. P is said to be *tree decomposable* provided that the statements of P can be arranged in a tree such that

(i) every node of the tree is a unique statement B_i and each B_i is some node of the tree,

(ii) for each B_i, $\mathrm{CLOS}(B_i)$ contains all B_j such that B_j is a descendant of B_i in the tree,

(iii) if B_j is a direct descendant of B_i, then $\mathrm{EXIT}(B_i) \cap B_j$ is nonempty, and

(iv) for each B_i, $\mathrm{EXIT}(B_i)$ is contained in the union of all B_j such that either B_j is a direct descendant of B_i or B_j is an ancestor of B_i.

3.1.3. P is said to be *forward tree decomposable* if the statements of P can be arranged in a tree such that this tree satisfies (i), (ii), (iii) above and

(iv′) for each B_i, $\mathrm{EXIT}(B_i)$ is contained in the union of all B_j such that B_j is a direct descendant of B_i.

Note that if a program has a tree decomposition, then it must be closed. For this reason we will confine our discussion to closed programs.

In a tree decomposition, we can think of the subtrees that hang below a node as subprograms of the statement at that node. With this intuitive view, a tree decomposition exhibits the program as a hierarchy of subprograms. Control may pass, from a given statement, down to a subprogram at the next level or up to any calling subprogram which is above the statement in the hierarchy. A forward tree decomposition has a particularly simple structure in that control can only pass down and never up in the hierarchy.

We next define some notions which will help to characterize when a closed program may be converted into an equivalent tree decomposable program.

3.2 Definition. Let $M = (S, I_1, I_2, ..., I_m)$ be an I machine and let $M' = (S, \pi)$ be its associated P machine.

3.2.1. An instruction I_j of M is said to be a *start instruction* provided that

(i) if s is in $\text{DOM}(I_j)$, then there is no state s' such that $\pi(s') = s$, and

(ii) if s is any state in S, then there is a computation $s_0, s_1, ..., s_\ell$ such that s_0 is in $\text{DOM}(I_j)$ and $s_\ell = s$.

The states in $\text{DOM}(I_j)$ are called *start states*. Clearly, if M has a start instruction, then it is unique. If M has a start instruction, we will denote it by $\text{START}(M)$.

3.2.2. Assume M has a start instruction and $P = \{B_1, B_2, ..., B_n\}$ is a program for M. A statement B_i is said to be a *start statement* for P provided that

(i) B_i is a subset of $\text{DOM}(\text{START}(M))$ and

(ii) if s is any state in $\text{SUP}(P)$, then there is a partial computation $s_0, s_1, ..., s_\ell$ such that s_0 is in B_i, s_j is in $\text{SUP}(P)$ for $0 \leqslant j \leqslant \ell$, and $s_\ell = s$.

Note that if P has a start statement, then it is unique. In such a case we denote it by $\text{START}(P)$.

3.2.3. Let $s_0, s_1, s_2, ..., s_\ell$ be a computation of M. The *instruction history* of this computation is the sequence $I_{i_0}, I_{i_1}, I_{i_2}, ..., I_{i_\ell}$ of instructions such that s_j is in $\text{DOM}(I_{i_j})$. If s_ℓ is a halting state, then we end the sequence with H, where H is a new formal object used to denote "halting instruction." The *trace* of this partial computation is the subsequence of its instruction history obtained by deleting all $I_{i_{j+1}}$ such that $I_{i_{j+1}} = I_{i_j}$.

3.2.4. M is said to be *trace consistent* provided that

(i) M has a start instruction, and

(ii) any two partial computations which start with a start state and end with the same state have the same trace.

The proof of the next lemma is routine and hence omitted.

3.3 Lemma. *Let M be an I machine with a start instruction.*

(1) *If P is a program with a start statement and P has a tree decomposition, then this tree decomposition is unique.*

(2) *If P is a universal tree decomposable program for M, then this program has a start statement.*

(3) *If P is a universal program for M with a start statement, then* $\text{START}(P) = \text{DOM}(\text{START}(M))$.

When trying to decide if every closed program for a machine is equivalent to a (forward) tree decomposable program, it suffices to consider only universal programs. The next lemma makes this more precise. The proof is easy and hence omitted.

3.4 Lemma. *Let M be an instruction machine with a start instruction and let P be a universal program for M. If P has a tree decomposition, respectively forward tree decomposition, then every closed program for M, which has a start statement, is equivalent to a tree decomposable, respectively forward tree decomposable, program.*

3.5 Theorem. *If $M = (S, I_1, I_2, \ldots, I_m)$ is an I machine that is trace consistent and P is a closed program for M with a start statement, then P is equivalent to a tree decomposable program.*

Proof. We will describe a tree decomposable program, P', that is equivalent to P. The construction of P' is illustrated by Example 3.6. As an intermediate step, we first construct a tree, T, and a directed graph, G, that have each node labeled either by an instruction of M or by H. Without loss of generality, assume I_1 is the start instruction. Construct T to be of depth $m + 1$ as follows. Label the root node I_1. Give the root node m direct descendants labeled $I_2, I_3, \ldots, I_m$ and H. In general, give each node not labeled by H direct descendants labeled $I_{i_1}, I_{i_2}, \ldots, I_{i_\ell}$ and H; where these instruction labels are all labels such that neither the given node nor any ancestor of the given node is labeled by any of $I_{i_1}, I_{i_2}, \ldots, I_{i_\ell}$. The nodes labeled H have no descendants. Clearly, this process terminates and yields a tree, T, in which the node labels of paths from root to a leaf consist of all sequences of instructions such that the sequence starts with I_1, ends with H and has no instruction repeated. To get G from T: Change the arcs from parent to offspring nodes to directed arcs terminating at the offspring, and add directed arcs from each node, not labeled by H, to each of its ancestor nodes. The resulting directed graph is G. With each node of G, we associate a subset of $\text{SUP}(P)$ as follows. With the root node we associate the start statement of P. With each other node, N, we associate the set of all states s in $\text{SUP}(P)$ such that: if we first take the unique trace of computations from a start state to s and then, starting at the root node, follow the directed path which passes through nodes labeled by the elements of the trace (in order) then we end up at N. P' consists of all nonempty sets which are associated with some node.

Clearly, the tree T describes a tree decomposition of P', provided that P' is a program. That is, provided the sets making up P' are pairwise disjoint. But this follows easily from the fact that M is trace consistent. This completes the proof. ▮

3.6 Example. This example illustrates the construction given in the proof of Theorem 3.5 as well as a number of the concepts discussed previously. Let $M = (S, I_1, I_2, I_3)$, where $S = \{1, 2,..., 11\}$ and the functions I_1, I_2 and I_3 are described in Fig. 1. In Fig. 1 the states are given in four columns, one for the domain of each instruction and one column for the halting states: an arrow from one state to another means that M can make this state transition in one step; for example, $I_1(1) = 5$. Note that M has a start instruction, namely I_1. It is easy to see that M is trace consistent and hence, by Theorem 3.5, every closed program with a start statement is equivalent to a tree-decomposable program.

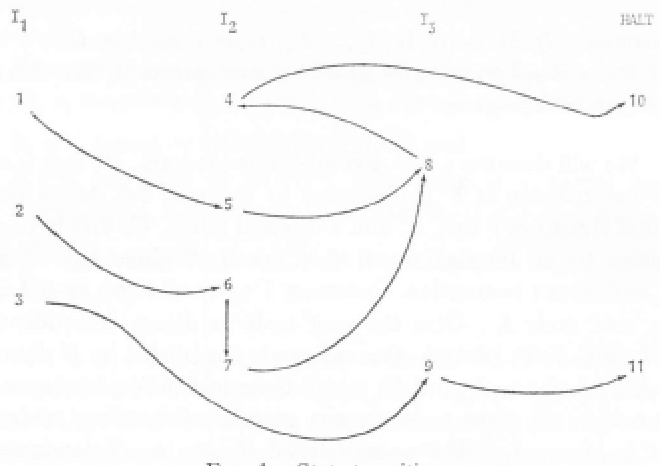

Fig. 1. State transitions.

Let $P = \{\{1, 2, 3\}, \{4, 5, 6\}, \{7\}, \{8, 9\}, \{10, 11\}\}$. Then P is a universal closed program with a start statement, but P is not tree decomposable. We now illustrate the construction from the proof of Theorem 3.5 and thereby obtain a tree-decomposable program P' equivalent to P. The tree T and directed graph G for P are given in Figs. 2 and 3 respectively. Figure 4 shows the set of states associated with each node of G. ($\varnothing$ denotes the empty set.) From this we get the equivalent program $P' = \{\{1, 2, 3\}, \{4, 5, 6, 7\}, \{8\}, \{9\}, \{10\}, \{11\}\}$ and its tree decomposition. The tree decomposition of P' is given in Fig. 5. ▮

The converse to Theorem 3.5 does not hold as shown by the following example

3.7 EXAMPLE. Below we define a machine M such that M has a start instruction, every program for M which has a start instruction is equivalent to a tree decomposable program but M is not trace consistent. $M = (S, I_1, I_2, I_3)$, where $S = \{1, 2, 3\}$, $I_1(1) = 2$, $I_2(2) = 3$, and $I_3(3) = 2$. I_1 is a start instruction. Clearly, there is a universal tree decomposable program, $\{\{1\}, \{2\}, \{3\}\}$ and, hence, every program with a start instruction is equivalent to a tree decomposable program. However, the two computations 1, 2, and 1, 2, 3, 2 have different traces. ∎

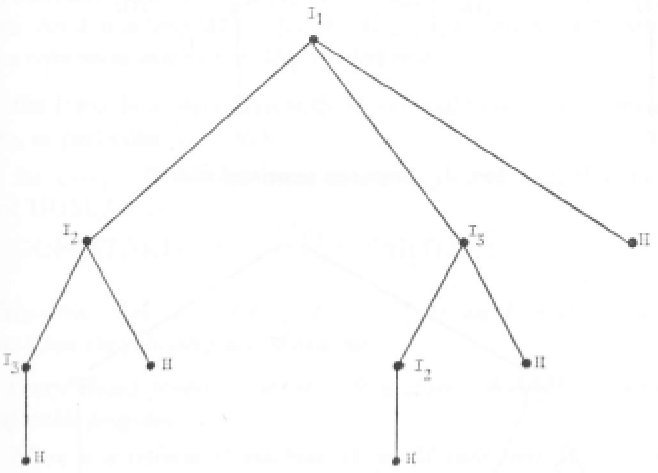

FIG. 2. Tree T.

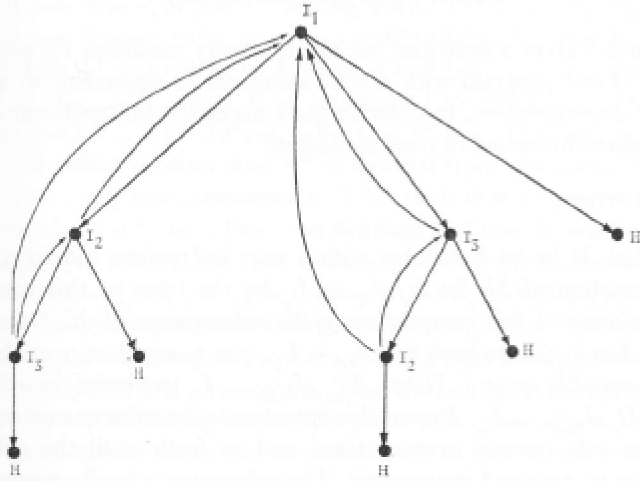

FIG. 3. Directed graph G.

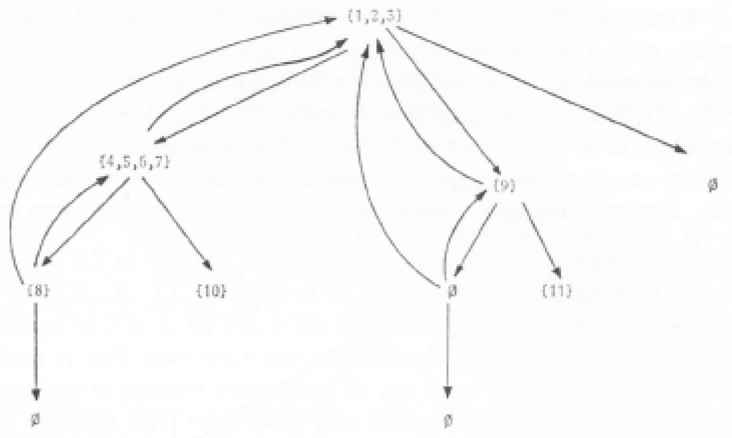

FIG. 4. State sets associated with G.

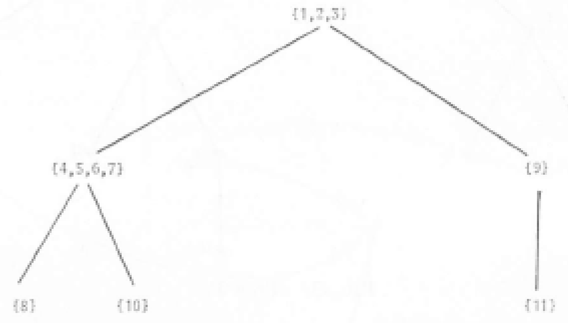

FIG. 5. Tree decomposition for P'.

Theorem 3.5 gives a sufficient but not necessary condition for guaranteeing that every closed program with a start statement is equivalent to a program with a tree decomposition. In order to get a necessary and sufficient condition, we will weaken the notion of trace consistent.

3.8 DEFINITION.

3.8.1. Let M be an I machine with a start instruction and let $s_0, s_1, ..., s_n$ be a computation of M. let $I_{i_0}, I_{i_1}, ..., I_{i_n}$ be the trace of this computation. The *reduced trace* of this computation is the subsequence of this trace obtained as follows. Let $j < k$ be such that $I_{i_j} = I_{i_k}$, j is as small as possible and k is as large as possible given j. Delete $I_{i_{j+1}}, I_{i_{j+2}}, ..., I_{i_k}$ to obtain the subsequence $I_{i_0}, I_{i_1}, ..., I_{i_j}, I_{i_{k+1}}, ..., I_{i_n}$. Repeat this operation to the subsequence so obtained, then to the subsequence next obtained and so forth until the subsequence obtained has no repeated instructions. The subsequence finally produced is the reduced trace.

3.8.2. An I machine, M, is said to be *reduced trace consistent* provided that

(i) M has a start instruction, and

(ii) any two computations that start with a start state and end with the same state have the same reduced trace.

Note that, if M is trace consistent, then it is reduced trace consistent. However, the converse is, in general, not true.

3.9 Definition. Let $M = (S, I_1, I_2, ..., I_m)$ be an I machine with a start instruction. An I machine $M' = (S', I'_1, I'_2, ..., I'_n)$ with a start instruction is said to be a *refinement machine* of M provided that

(i) the P machine associated with M' is equal to the P machine associated with M (so, in particular, $S' = S$),

(ii) for every $1 \leqslant i \leqslant n$, there is a $1 \leqslant j \leqslant m$ such that $\mathrm{DOM}(I'_i)$ is a subset of $\mathrm{DOM}(I_j)$, and

(iii) $\mathrm{DOM}(\mathrm{START}(M)) = \mathrm{DOM}(\mathrm{START}(M'))$.

3.10 Theorem. *Let $M = (S, I_1, I_2, ..., I_m)$ be an I machine with a start instruction. Then the following are equivalent.*

(1) *Every closed program for M with a start statement is equivalent to a tree decomposable program.*

(2) *There is a refinement machine M' of M such that M' is reduced trace consistent.*

Proof. We first show that (1) implies (2). Suppose (1) holds. First, consider the special case where M has no halting states. Let $P = \{B_1, B_2, ..., B_n\}$ be a universal tree decomposable program for M and let (S, π) be the P machine associated with M. Note that, by Lemma 3.3, P has a start statement. Define $M' = (S, I'_1, I'_2, ..., I'_n)$, where I'_i is π restricted to B_i, $1 \leqslant i \leqslant n$. With the held of Lemma 3.3, it is not difficult to see that M' is a refinement machine of M. Thus, is will suffice to show that M' is reduced trace consistent. To see this, suppose $s_0, s_1, ..., s_\ell$ is a computation of M' such that s_0 is a start state. For some unique i_0, s_ℓ is in B_{i_0}. From the definitions of tree decomposition and of reduced trace, it follows that the reduced trace of this computation is the sequence of labels $I'_{h_1}, I'_{h_2}, ..., I'_{h_r}$ such that $\mathrm{DOM}(I'_{h_1}), \mathrm{DOM}(I'_{h_2}), ..., \mathrm{DOM}(I'_{h_r})$ label the nodes on the path from the root node to B_{i_0} in the tree decomposition of P. So each such subcomputation has a reduced trace determined solely by s_ℓ. Thus M' is reduced trace consistent. This shows that (1) implies (2) in the case where M has no halting states. The proof in the case where M has halting states is basically the same but is notationally harder to express. Since the difference in proofs for the two cases is basically one of notational change, we will omit the proof for the latter case.

It remains to show that (2) implies (1). By Lemma 3.4 it suffices to produce a universal tree decomposable program for M. By the definition of a refinement machine, it suffices to produce a universal tree decomposable program for M'. However, inspecting the construction given in the proof of Theorem 3.5, one notices that under the assumption of reduced trace consistency for M', given the natural universal program for M', it produces an equivalent tree decomposable program. ∎

The remainder of this section is concerned with producing a necessary and sufficient condition for guaranteeing that every closed program with a start statement is equivalent to a program with a forward tree decomposition.

3.11 Definition. An I machine M is said to be *trace bounded* provided there is some N such that: for any computation of M, the trace of this computation has length at most N.

3.12 Lemma. *Let M be an I machine with a start instruction. If M has a universal, forward tree decomposable program, then M is trace consistent and trace bounded.*

Proof. Let P be a universal, forward tree decomposable program for M. If the tree in the tree decomposition of P has depth[1] N, then every trace of a computation of M will have length at most N. So M is trace bounded.

If s is any state of M, then every computation from a start state to s passes through the same path in the forward tree decomposition of P, namely the path from the root node to the unique node labeled by a statement containing s. From this it follows immediately that, any two such computations have the same trace. So M is also trace consistent. ∎

3.13 Theorem. *Let M be an I machine with a start instruction. The following are equivalent.*

(1) *Every closed program for M with a start statement has an equivalent forward tree decomposable program.*

(2) *M is trace consistent and trace bounded.*

Proof. By Lemma 3.12, it follows that (1) implies (2). Conversely, suppose (2) holds. In order to establish (1), it will suffice, by Lemma 3.4, to show that M has a universal, forward tree decomposable program. To accomplish this, we use a technique similar to that used in the proof of Theorem 3.5. Let N be such that the trace of every complete computation of M has length at most N and let m be the number of instruction of M. We first construct a tree T

[1] The depth of a tree equals the number of nodes on a maximal length path from the root to a leaf.

of depth N such that every node is labeled either by an instruction of M or by H, where, as before, H is a formal object used to denote "halting instruction". The tree is constructed from root to leaves down to N levels as follows. The root node is labeled by the start instruction. The root node has m offspring nodes labeled by H and the $m - 1$ instructions other than the start instruction. Each non-root node, not labeled by H, has $m - 1$ offspring nodes labeled by the $m - 1$ labels consisting of H and the $m - 2$ instructions other than the start instruction and the instruction labeling the parent node. Nodes labeled H have no offspring. So the set of paths from root to leaves of T are labeled by all possible sequences of length at most N such that:

(1) the first element of the sequence is the start instruction and no other element of the sequence is the start instruction,

(2) any two consecutive elements of the sequence are different, and

(3) H occurs only at the end of a sequence and all sequences of length less than N end with H.

With each node of T we associate the set of states s such that the unique trace from a start state to s is the sequence of labels encountered on the path from the root node to the given node. Let P be the set of all non-empty sets associated with the nodes of T. It is easy to see that P is a universal program for M and that the above construction exhibits a forward tree decomposition of P. ∎

4. Canonical Tree Decompositions

Given a program P which is forward tree decomposable, there are, in general, many programs P' which are equivalent to P and forward tree decomposable. There is, however, one program P' which is in some sense the smallest such program and which can be, in some intuitive sense, "effectively" constructed given P. This P' will be called the first canonical tree decomposition equivalent to P. There is also a second canonical tree decomposition equivalent to P. As with the first canonical decomposition, the second canonical decomposition is also unique. The difference between the two canonical forms is that the second canonical form displays more computational structure. If a machine has two complete computations such that the trace of one is a prefix of the trace of the other, then this can easily be detected by studying the structure of the second canonical form but this is not easily displayed using the first canonical form. The method of constructing canonical forms applies to tree decompositions as well as forward tree decompositions. However, if the tree decomposition is not forward, then the canonical forms may not be unique.

4.1 Definition. Let P be a program for an I machine $M = (S, I_1, I_2, \ldots, I_m)$.

4.1.1. A statement B in P is said to be *closed* if $\mathrm{CLOS}(B) = B$. B is said to be *open* provided that, for each state s in B, there is a computation starting with s and leading to a state which is not in in B. Note that there may be statements B which are neither open nor closed.

4.1.2. If B is in P, then the *instruction* of B is denoted $\mathrm{INST}(B)$ and is defined to be the unique I_i such that B is included in $\mathrm{DOM}(I_i)$. If B is a halt statement, then $\mathrm{INST}(B) = H$, where again H is a new formal object to stand for "halting instructions."

4.1.3. Suppose P is tree decomposable and consider a tree decomposition of P. The decomposition tree is said to be in *first canonical form* provided that the following holds: If A and B are in P and A, B are either siblings or one is the parent of the other in the tree decomposition, then $\mathrm{INST}(A) \neq \mathrm{INST}(B)$.

4.1.4 (Notation as in 4.1.3). The tree decomposition is said to be in *second canonical form* provided that

(i) if A is a statement of P which is not at the root node, then A is either open or closed,

(ii) if A and B are statements of P and A is the parent of B in the tree decomposition, then $\mathrm{INST}(A) \neq \mathrm{INST}(B)$, and

(iii) if A and B are statements of P, A, B are siblings and A, B are either both open or both closed, then $\mathrm{INST}(A) \neq \mathrm{INST}(B)$.

4.2. Theorem. *Let M be an I machine and let P be a tree decomposable program. Then there are equivalent tree decomposable programs P' and P'' such that the tree decomposition of P', respectively P'', is in first, respectively second, canonical form. Furthermore, if P is forward tree decomposable, then the tree decompositions of P' and P'' will be forward tree decompositions.*

Proof. Let us consider the first canonical form. To obtain P' and its tree decomposition from a tree decomposition of P, proceed as follows. If A is the parent of B in the given tree decomposition of P and $\mathrm{INST}(A) = \mathrm{INST}(B)$, then replace A by $A \cup B$ in the tree decomposition and hang both the subtrees that were below A and the subtrees that were below B, below $A \cup B$ in the tree decomposition. If A and B are siblings and $\mathrm{INST}(A) = \mathrm{INST}(B)$, then replace A by $A \cup B$ in the tree decomposition and hang both the subtrees that were below A and the subtrees that were below B, below $A \cup B$ in the tree decomposition. Repeat these two operations as often as possible. The resulting tree is a tree decomposition in first canonical form for a program P' equivalent to P. If the original tree decompositions were forward, then the tree decomposition produced in this way will also be forward.

The second canonical form P'' is obtained from the first canonical form P' as follows. For each non-root node statement A of the program in first canonical form, replace A by A_1 and add a sibling A_2 to A_1, where A_1 and A_2 are defined as follows. $A_1 = \{s \mid s$ is in A and there is a computation leading from s to a state not in $A\}$ and $A_2 = A - A_1$. Note that the subtrees that were below A are now below A_1 (as well as being changed themselves) and that A_2 has no offspring. If either A_1 or A_2 is empty, then it is omitted. Since what we have done is to factor each A into a closed statement A_2 and an open statement A_1, it follows that the resulting tree decomposition is in second canonical form. This second transformation also preserves the property of being a forward tree decomposition. ∎

4.3. THEOREM. *Let M be an I machine and let P be a forward tree decomposable program for M. Then there are unique programs P' and P'', and unique forward tree decompositions of P' and P'' such that: P, P' and P'' are equivalent, the forward tree decomposition of P' is in first canonical form and the forward tree decomposition of P'' is in second canonical form.*

Proof. The existence of the forward tree decompositions was proven by Theorem 4.2. So it remains to show that the first and second canonical forms are unique. Consider the first canonical form. Suppose P' is any tree decomposable program equivalent to P and such that P' has a tree decomposition T' in first canonical form. We proceed in two steps. First we show that a particular labeled tree T derived from T' can be characterized in terms of SUP(P) alone. Then we show that P' and T' can be characterized in terms of SUP(P) and T alone. From this it follows that P' and T' are uniquely determined by SUP(P).

T is the labeled tree obtained from T' by replacing each statement A, which labels a node of T', by the lable INST(A). We wish to describe T in terms of SUP(P). First note that the lable of the root node of T can be derived from SUP(P), This is because P has a forward tree decomposition and, hence, there is a unique instruction I, namely the instruction of the statement labeling the root node of a forward tree decomposition of P, such that every state in SUP(P) can be reached by a computation from some state in DOM(I). Hence the root node of T must be labeled by this instruction I. To see that T is uniquely determinded by SUP(P), note the following two points. The set of all sequences of labels from root node to some node of T must equal the set of traces of computations in SUP(P) that start with the instruction I. Also, T has the property that no two nodes N_1, N_2 such that N_1 is the parent of N_2 or N_1 and N_2 are siblings, can be labeled by the same instruction. There is only one labeled tree with these two properties. Hence T is determined by SUP(P).

We now describe T', the forward tree decomposition in first canonical form in terms of T and SUP(P). If A labels the root node of T', then A consists of all states s in DOM(I) such that there is no computation $s_0, s_1, \dots, s_\ell$ with $s_\ell = s$

and s_0 in SUP(P) — DOM(I). (Recall that I is the instruction that labels the root node of T.) Let A_0 denote the statement that labels the root node in T'. If statement A labels a non-root node of T', then A is the set of all states s such that the trace of all computations in SUP(P) which start in A_0 and end with s is equal to the instructions encountered along the path in T from the root node to the node corresponding to A. Since we have completely described T' in terms of SUP(P) alone, it follows that T' and P' are unique.

The uniqueness of the second canonical form follows from the uniqueness of the first canonical form as follows. Let T be a forward tree decomposition. Let F_1 and F_2 be the transformations defined in the proof of Theorem 4.2 such that $F_1(T)$ and $F_2(T)$ are equivalent forward tree decompositions in first, respectively second canonical form. Note that if T is in second canonical form, then $F_2(F_1(T)) = T$. Now let T_1 and T_2 be equivalence forward tree decompositions in second canonical form. We wish to show that $T_1 = T_2$. But we have already shown that $F_1(T_1) = F_1(T_2)$. So $T_1 = F_2(F_1(T_1)) = F_2(F_1(T_2)) = T_2$. ▌

The analog of Theorem 4.3 for tree decompositions, as apposed to forward tree decompositions, is not valid. To see this note that a program P may have a canonical tree decomposition such that every computation is a cycle and hence any node may be taken as the root node. There are also other situations which can produce non-unique canonical tree decompositions. This is true for both the first and the second canonical forms.

Our next result shows that the canonical forms are, in some sense, the smallest forward tree decompositions equivalent to a given program P. The proof is easy, given the techniques already developed, and hence is omitted.

4.4 THEOREM. *Let M be an I machine with a start instruction. Let P and P' be two equivalent, forward tree decomposable programs for M with start statements.*

(1) *If the forward tree decomposition of P' is in first canonical form, then every statement of P' is equal to the union of some statements of P.*

(2) *If the forward tree decomposition of P' is in second canonical form and every statement of P is either open or closed, then every statement of P' is equal to the union statements of P.*

5. QUOTIENT MACHINES

In previous sections we saw that the extent to which we can construct "well structured" programs for a given machine, M, depends not so much on the actual computations of M as it does on the traces of computations. In this section we consider "quotient machines" obtained by identifying states which produce computations having the same trace. In this way we can, in some sense, factor out properties of the machine M which are irrelevant to our current study. In addition to considering equivalence relations that identify two states which lead to the same computation trace, we also consider weaker equivalence relations that simply require that the traces are equal for some initial segment. Also, we consider equivalence relations that identify two states, provided that they are both the last state in computations with equal traces.

5.1 DEFINITION. Let $M = (S, I_1, I_2, ..., I_m)$ be an I machine.

5.1.1. Define the equivalence relation $\sim_f$ on S by $s_1 \sim_f s_2$ provided that the traces of the complete computations starting with s_1 and s_2 are equal.

5.1.2. Define equivalence relations $\sim_f^k$ $(k = 1, 2, 3, ...)$ on S by $s_1 \sim_f^k s_2$ provided that either

(i) the traces of the complete computations starting with s_1 and s_2 both have length at least k and agree on their first k entries,

(ii) the traces of the complete computations starting with s_1 and s_2 both have length less than k and are equal.

5.1.3. Let S_f and $S_f{}^k$ denote the set of equivalence classes of elements of S induced by $\sim_f$ and $\sim_f^k$ $(k = 1, 2, 3, ...)$ respectively. Let M_f denote the I machine $(S_f, \bar{I}_1, \bar{I}_2, ..., \bar{I}_m)$ where the instructions $\bar{I}_i$ are interpreted as follows. Let $[s]$ be the equivalence class of s with respect to $\sim_f$.

(i) If s is in DOM(I_i) and the trace of the complete computation starting with s is the one element sequence I_i, then $\bar{I}_i([s]) = [s]$.

(ii) If s is in DOM(I_k), $s = s_1, s_2, s_3, ...$ is the complete computation in M starting with s and j is the least j such that s_j is not in DOM(I_i), then $I_i([s]) = [s_j]$.

(iii) If s is a halting state of M, then $[s]$ is a halting state of M_f.

5.1.4. Let k be a positive integer. The machine M is said to be *initially k determined* provided that, for any states s_1 and s_2 of M, $s_1 \sim_f^k s_2$ implies $s_1 \sim_f s_2$.

We shall see that there are a number of things we can say about the programming structure of M in terms of the programming structure of M_f.

5.2 THEOREM. *If M is an I machine, then the following statements are equivalent.*

(1) *S_f is finite.*

(2) *$S^f = S_f{}^k$, for some k.*

(3) *M is initially k determined, for some k.*

Proof. Clearly (2) implies (3) which, in turn, implies (1). So it will suffice to show (1) implies (2). To see this, note that $S_f{}^1, S_f{}^2, S_f{}^3, ...$ is a sequence of finer and finer partitions of S and that S_f is the common refinement of all these partitions. Thus, if S_f is finite, then there must be a k such that, after S_f^k, the refinements in the sequence are no longer proper refinements and hence, $S_f = S_f^k$ for this k. ∎

5.3 THEOREM. *Let M be an I machine with a start instruction. If M_f has a universal tree decomposable, respectively universal forward tree decomposable, program, then every closed program for M with a start statement has an equivalent tree decomposable, respectively forward tree decomposable, program.*

Proof. Suppose M_f has a tree decomposable, respectively forward tree decomposable, universal program. Take the tree decomposable universal program for M_f and replace each statement A of this program by the union of all equivalence classes in A. The result is a universal tree decomposable, respectively universal forward tree decomposable, program for M. The Theorem now follows directly from Lemma 3.4. ∎

The converse to Theorem 5.3 does not hold as shown by the next example.

5.4 EXAMPLE. Let $M = (S, I_1, I_2)$ where $S = \{1, 2, 3, 4, 5\}$, $I_1(1) = 3$, $I_1(2) = 4$, $I_2(3) = 5$ and 4, 5 are halting states. M has I_1 as a start instruction, M has a universal forward tree decomposable prgram and, hence, every closed program for M has an equivalent forward tree decomposable program. However, M_f does not have a universal tree decomposable program. ∎

If M is an I machine and s is a state of M, then there is a unique complete computation of M starting with state s and, hence, it trivially follows that there is a unique trace for complete computations starting with s. If M is trace consistent, then there is also a unique trace for computations that start with a start state and end with s. So, if M is trace consistent, then each state has a unique "forward" trace and a unique "backward" trace. We have already considered the equivalence relation induced by "forward" traces. We now consider the equivalence relation induced by "backward" traces.

5.5 DEFINITION. Let $M = (S, I_1, I_2, \ldots, I_m)$ be an I machine that is trace consistent.

5.5.1. Let s be a state of M. Since M is trace consistent, there is a unique trace such that every computation from a start state to s has this trace. Call this trace the *backward trace* of s.

5.5.2. Define the equivalence relation $\sim_b$ on S by $s_1 \sim_b s_2$ provided s_1 and s_2 have the same backward trace.

5.5.3. Define equivalence relations $\sim_b^k$ $(k = 1, 2, 3, \ldots)$ on S by $s_1 \sim_b^k s_2$ provided that either

(i) the backward traces of s_1 and s_2 both have length at least k and agree on their last k entries, or

(ii) the backward traces of s_1 and s_2 both have length less than k and are equal.

5.5.4. M is said to be ***terminally k determined*** provided $s_1 \sim_b^k s_2$ implies $s_1 \sim_b s_2$.

5.5.5. Define the equivalence relation $\sim$ on S by $s_1 \sim s_2$ provided both $s_1 \sim_f s_2$ and $s_1 \sim_b s_2$. Let $S_b{}^k$, S_b and $S_\sim$ denote the set of equivalence classes of S induced by $\sim_b^k$, $\sim_b$ and $\sim$ respectively ($k = 1, 2, \ldots$).

5.6 LEMMA. *Let M be an I machine that is trace consistent. If S_b is finite, then S_f is finite.*

Proof. Suppose S_f is infinite. It will suffice to show that S_b is infinite. If there is an infinite trace for some complete computation in M, then this trace can be extended backwards to obtain an infinite trace of a complete computation starting with a start state. From this it follows that S_b is infinite. So suppose all complete computations have finite trace and S_f is infinite. Then, since each trace can be extended backwards to obtain a finite treace starting from the start Instruction, it follows that there are infinitely many finite traces which start with a start instruction. Hence S_b is infinite. ▮

5.7 THEOREM. *Let $M = (S, I_1, I_2, \ldots, I_m)$ be an I machine that is trace consistent. The following are equivalent.*

(1) *S_b is finite.*

(2) *$S_b = S_b{}^k$ for some k.*

(3) *M is terminally k determined.*

(4) *$S_\sim$ is finite.*

(5) *$S_\sim$ is the common refinement of $S_b{}^k$ and $S_f{}^k$, for some k.*

Proof. The equivalence of (1), (2) and (3) and the equivalence of (4) and (5) are proven in the same way as Theorem 5.2. Since $S_\sim$ is the common refinement of S_b and S_f, the equivalence of (1) and (4) follows directly from Lemma 5.6. So all statements are equivalent. ▮

5.8 THEOREM. *Let M be an I machine that is trace consistent. The following are equivalent.*

(1) *M has a universal forward tree decomposable program.*

(2) *Every closed program for M with a start statement is equivalent to forward tree decomposable program.*

(3) *S_b is finite.*

(4) *$S_\sim$ is finite.*

Proof. The equivalence of (1) and (2) and the equivalence of (3) and (4) follow from Lemma 3.3 and Theorem 5.7 respectively. The equivalence of (1) and (3) follows from the proof of Theorem 4.3. So all statements are equivalent. ∎

5.9 Corollary. *Let $M = (S, I_1, I_2, \ldots, I_m)$ be an I machine with a start instruction. If M has a universal, forward tree decomposable program, then M is trace consistent and S_b is the unique universal, forward tree decomposable program for M in first canonical form.*

Proof. The result follows from Lemma 3.4, Theorem 3.13 and the proof of Theorem 4.3. ∎

5.10 Corollary. *Let $M = (S, I_1, I_2, \ldots, I_m)$ be an I machine with a start instruction. The following are equivalent.*

(1) *M has a universal, forward tree decomposable program.*

(2) *M is trace consistent and S_b is a forward tree decomposable program.*

(3) *M is trace consistent and S_b is finite.*

(4) *M is trace consistent and $S_\sim$ is finite.*

Received: November 4, 1977; revised: October 26, 1978

Reference

Pawlak, Z. (1969), Programmed machines [in Polish], *Algorytmy* **5**, No. 10.