JOURNAL OF COMPUTER AND SYSTEM SCIENCES 32, 97–104 (1986)

"During" Cannot Be Expressed by "After"

PAWEŁ URZYCZYN

Institute of Mathematics, University of Warsaw, PKiN, 00-901 Warsaw, Poland

Received October 10, 1984; revised May 11, 1985

We prove that the operator $\perp$ ("during") is not expressible in first-order logics of programs based on the operator $\langle\ \rangle$ ("after"), but it is expressible with the help of array assignments or rich tests. From this we deduce that array assignments add to the power of logics based on nondeterministic effective tree-schemes and that rich tests add to the power of logics based on flowcharts. The proof of the main theorem is based on a result of Furst, Saxe, and Sipser (*in* "Proc. 22nd Found. of Comput. Sci.," 1981). Then it shows an example of how the Boolean circuit complexity theory may be applied to logics of programs. © 1986 Academic Press, Inc.

1. INTRODUCTION

In the present paper we discuss the expressive power of three "non-standard" constructs used in first-order logics of programs. These constructs are: temporal operators, array assignments, and rich tests. Although the use of rich tests is treated, e.g., by Harel [2], as a basic property of dynamic logic, it is more natural, in the author's opinion, to consider programs with quantifier-free and program-free tests only. It follows from results of Meyer and Parikh [4] that rich tests enrich the power of logics based on recursively enumerable programs. We prove (Corollary 9) that the same holds for regular (flowchart) programs.

Logics with array assignments considered in the paper are defined in the spirit of [2], i.e., a name of an array occurring in a program scheme P may also occur in a formula α in the context $\langle P\rangle\alpha$. That is, some information about the behaviour of P may be transmitted "outside of P" in the formula. It was proved by Meyer and Tiuryn [5] that array assignments do not change the power of logics based on deterministic r.e. programs. However, as we prove below (Corollary 8), it is not true in the nondeterministic case.

Both the above mentioned results follow from Theorem 7, which states that a temporal operator $\perp$ ("during") adds to the power of logics of programs based on nondeterministic flowcharts, as well as nondeterministic tree-schemes. The operator $\perp$, introduced by Pratt [8], has the following meaning: we read $P \perp \alpha$ as "in every computation of P there is a state satisfying α." As a tool to prove Theorem 7 we use a theorem of Furst, Saxe, and Sipser who proved [1] that no sequence of circuits of constant depth and polynomially bounded sizes may define the language of all the words containing an even number of occurrences of "1." (We restate this result here as Lemma 5.) There are two other temporal operators considered below, namely $\sqcup\!\sqcup$ ("throughout") and $\lrcorner\!\ulcorner$ ("preserves"), but those turn out to be expressible in logics with input–output semantics (Theorems 1 and 2). All the three operators were proved to be expressible in algorithmic logic, but with the use of rich tests (Mirkowska and Stapp [7]).

The formalism of our consideration differs from that of [2]. The latter was oriented for the input–output semantics, while allowing temporal operators requires considering programs as objects defining computations. Especially, in case of r.e. programs, Harel's definition only allows those "infinite computations" whose initial segments are also initial segments of finite seq's. In addition, this definition leads to a quite unnatural effect: unbounded (even infinite) nondeterminism. Thus, we choose the notions of a flowchart scheme and an effective tree-scheme for our purposes. However, with respect to the input–output behaviour, our schemes are equivalent to regular and r.e. programs in [2].

2. Definitions

Throughout the paper, we assume that there is a fixed signature δ, i.e., a finite sequence of function and relation symbols (including $=$, always interpreted as equality). A *flowchart scheme* (over δ) is a finite directed graph with all nodes (except the only initial (input) node and the final (output) nodes) labelled by assignments, tests and "or" instructions. If not defined otherwise, all tests are assumed to be quantifier-free formulas of classical logic. Assignments take the form $x := t$, where x is a variable and t is a term over δ. The nodes labelled by "or" instructions, representing nondeterministic choices, are assumed to be of outdegree 2, as well as those labelled by tests. For any test-labelled node, the outgoing edges represent the two possible answers to the test: "yes" and "no." Assignment-labelled nodes are of outdegree 1. A generalization of the notion of a flowchart is the notion of an *effective tree scheme*, which is a (potentially) infinite tree of nodes labelled as described above, such that an effective procedure is able to determine the label of a given node, as well as all its neighbours. For simplicity, we assume that only a finite number of variables may occur in an effective tree scheme. (This does not cause any confusion, since arbitrary terms are allowed in assignments.) A program scheme is *deterministic* if it does not contain the instruction "or." The class of all flowcharts (all effective tree-schemes) we denote by FC (ET), while the subclasses of deterministic schemes are denoted by DFC (DET). Clearly, unwinding the loops of a given flowchart provides an effective tree, whence we may assume that $\mathrm{FC} \subseteq \mathrm{ET}$ and $\mathrm{DFC} \subseteq \mathrm{DET}$. It is a routine observation that effective trees correspond to "informal algorithms," defined in [3] by Kfoury, and thus are equivalent to flowcharts with counters and recursion (or stack).

A *computation* of a program scheme P on an input $\mathbf{a}$ in a δ-structure A is a sequence of *states*, i.e., valuations of variables, such that $\mathbf{a}$ is the initial state and the other are obtained by executing the instructions along a maximal path in P, the answers to tests of which are satisfied by the successive valuations. An output (if defined) is the last state of a computation. Thus, a scheme P defines in a δ-structure A a binary input-output relation on states, denoted P_A. It is easily seen that the class of all relations definable by FC- (ET-) schemes coincides with the class of relations definable by regular (r.e.) programs in the sense of [2], with the restriction that only quantifier-free tests are allowed.

If K is a class of program schemes then the *logic of programs over* K, denoted $L(K)$, is built up, in a standard way, from first-order connectives and program schemes in K, with help of the dynamic construct $\langle\ \rangle$, as in [2]. The semantics of $\langle\ \rangle$ is given by the following condition:

$$A, \mathbf{a} \models \langle P \rangle \alpha \quad \text{iff there is } \mathbf{b}\text{, with } (\mathbf{a}, \mathbf{b}) \in P_A \text{ and } A, \mathbf{b} \models \alpha.$$

For two logics, L_1 and L_2, we write $L_1 \leqslant L_2$ iff, for every formula of L_1, there is an equivalent formula in L_2. $L_1 \equiv L_2$ stands for "$L_1 \leqslant L_2$ and $L_1 \geqslant L_2$." Using the notation of [2, 5], we may write: $L(\mathrm{FC}) \equiv QDL^{(qf)}$, $L(\mathrm{DFC}) \equiv \mathbf{det}\text{-}QDL^{(qf)}$, $L(\mathrm{ET}) \equiv QDL_{\mathrm{re}}^{(qf)} \equiv DL$-w/o-array, $L(\mathrm{DET}) \equiv \mathbf{det}\text{-}QDL_{\mathrm{re}}^{(qf)} \equiv DDL$-w/o-array.

We are going to extend the notion of logic of programs in three ways. First, we may allow rich tests to occur in programs, i.e., define a class of rich-test program schemes and the corresponding logic by simultaneous induction. For a class K of program schemes, we put $K_0 = K$, $L_0 = L(K)$. Then K_{i+1} is obtained by allowing K-schemes to contain tests from L_i, and L_{i+1} is defined as $L(K_{i+1})$. Finally, $L_R(K) = \bigcup\{L_i : i \in N\}$. Clearly, we have e.g., $L_R(\mathrm{FC}) \equiv QDL$. We also consider program schemes with program-free, but not necessarily quantifier-free, formulas in tests. We use the notation $L_{FO}(K)$ to denote the appropriate subset of the formulas in $L_R(K)$.

Another way to enrich logics of programs is to consider *array assignments*, i.e., assignments of the form $F(\mathbf{x}) := t$, and $x := t$, for a finite number of function symbols F, not occurring in δ and terms containing these symbols. The meaning of $F(\mathbf{x}) := t$ is that the value of $F(\mathbf{x})$ is defined to be the value of t after the assignment. Similarly, we allow assignments $R(\mathbf{x}) := \mathit{true}$ (or false) for relation symbols R not in the signature. For a class K of schemes, let AK be obtained from K by allowing array assignments. Then $L(AK)$ is defined in the usual way, but in the construct $\langle P \rangle \alpha$, the formula α may contain symbols not in δ, provided they occur in the scheme P.

At last, consider the temporal operators $\sqcup\!\sqcup$ ("throughout"), $\perp$ ("during"), and $\int$ ("preserves"), defined as follows:

$A, \mathbf{a} \models P \sqcup\!\sqcup \alpha$	iff	α is true of each state in every computation of P on $\mathbf{a}$;
$A, \mathbf{a} \models P \perp \alpha$	iff	α is true of some state in every computation of P on $\mathbf{a}$;
$A, \mathbf{a} \models P \int \alpha$	iff	whenever α is true in some state in a computation then it is true in all following states.

We may allow one or more of these operators to occur in formulas of $L(K)$, obtaining new logics like, e.g., $L_{\perp}(K)$, $L_{\sqcup\!\sqcup\,\int}(K)$, or $L_{\sqcup\!\sqcup\,\perp\,\int}(K)$.

We end up with a notion needed for Lemma 5. A *Boolean circuit* is a finite net built up from input gates and gates for conjunction, disjunction, and negation. An assignment of Boolean values to input gates determines an output value assigned to a distinguished output gate in an obvious way. A circuit with n inputs $x_1,..., x_n$ *accepts* a word $w = a_1,..., a_n \in \{0, 1\}^n$ iff the output value is 1 provided the input values assigned to $x_1,..., x_n$ are $a_1,..., a_n$, respectively. The *size* of a circuit is the total number of its gates, and the *depth* of it is the length of the longest path from an input gate to the output gate. A sequence C_0, $C_1 C_2,...$, of circuits such that each C_i has i input gates *accepts* a language $L \subseteq \{0, 1\}^*$ iff, for each word $w \in \{0, 1\}^*$, $C_{|w|}$ accepts w iff $w \in L$. Such a sequence is said to be of constant depth and polynomial size iff there are a constant d and a polynomial $p(n)$ such that each C_n is of depth at most d and size at at most $p(n)$.

3. Results

We start with showing that, in the deterministic case, the temporal operators do not add to the power of basic logics of programs.

THEOREM 1. *$L_{\sqcup\!\sqcup\,\perp\,\int}(\mathrm{DFC}) \equiv \mathrm{L}(\mathrm{DFC})$ and $L_{\sqcup\!\sqcup\,\perp\,\int}(\mathrm{DET}) \equiv L(\mathrm{DET})$.*

Proof. For a given scheme $P(\mathbf{x})$, with the variables $\mathbf{x}$, consider new schemes $P_1(\mathbf{x}, \mathbf{y})$ and $P_2(\mathbf{x}, \mathbf{y}, \mathbf{z})$ such that:

—P_1 converges for the input $\mathbf{a}$, $\mathbf{b}$ iff $\mathbf{b}$ is a state in the computation of P on $\mathbf{a}$;

—P_2 converges for $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ iff $\mathbf{b}$ and $\mathbf{c}$ are states in the computation of P on $\mathbf{a}$ and some occurrence of $\mathbf{b}$ precedes some occurence of $\mathbf{c}$.

Then we have

$$\models P \sqcup\!\sqcup\, \alpha \leftrightarrow \forall \mathbf{y}\, (\langle P_1(\mathbf{x}, \mathbf{y})\rangle\, \mathit{true} \rightarrow \alpha(\mathbf{y}));$$
$$\models P \perp \alpha \leftrightarrow \exists \mathbf{y}\, (\langle P_1(\mathbf{x}, \mathbf{y})\rangle\, \mathit{true} \wedge \alpha(\mathbf{y}));$$
$$\models P \int \alpha \leftrightarrow \forall \mathbf{y}\, \mathbf{z}((\langle P_2(\mathbf{x}, \mathbf{y}, \mathbf{z})\rangle\, \mathit{true} \wedge \alpha(\mathbf{y})) \rightarrow \alpha(\mathbf{z})). \quad \blacksquare$$

In the nondeterministic case, the operators $\sqcup\!\sqcup$ and $\int$ may be still eliminated.

THEOREM 2. *$L_{\sqcup\!\sqcup\,\int}(\mathrm{FC}) \equiv L(\mathrm{FC})$ and $L_{\sqcup\!\sqcup\,\int}(\mathrm{ET}) \equiv L(\mathrm{ET})$.*

Proof. Let $P(\mathbf{x})$ be given, and let $P_1(\mathbf{x})$ denote a new scheme, obtained from $P(\mathbf{x})$ by adding a possibility to stop in an arbitrary (nondeterministically chosen) state of any computation. Then

$$\models P \sqcup\!\sqcup\, \alpha \leftrightarrow [P_1]\alpha,$$

where $[\ \ ]$ abbreviates $\neg\langle\ \ \rangle\neg$.

The operator $\int$ is expressed the same way as in the deterministic case. The only difference is that one must replace "the computation" by "a computation" in the description of P_2. ▌

Before we discuss the power of $\perp$, we observe that array assignments and rich tests are at least as expressive as $\perp$.

THEOREM 3. *$L_\perp(K) \leqslant L(AK)$ and $L_\perp(K) \leqslant L_R(K)$ for $K \in \{\mathrm{FC}, \mathrm{ET}\}$.*

Proof. Let $P(\mathbf{x}) \in \mathrm{ET}$ and let $R(\mathbf{x})$ be a relation symbol not in δ. Construct a program scheme $P_1(\mathbf{x}) \in \mathrm{AET}$ to compute the following algorithm. First, P_1 guesses a natural number n, then it successively simulates all the initial fragments of computations of P (on a given input $\mathbf{a}$) of length not exceeding n. Each simulation is interrupted in a nondeterministically chosen state and then $R(\mathbf{x}) := \mathit{true}$ is executed. Now, consider the tree built up from all the computations of P on $\mathbf{a}$. If $\mathbf{a} \models P \perp \alpha$ holds then, by König's lemma, there is n such that a state satisfying α can be found in the first n steps of any computation, since the tree is finitely branching. Thus we have

$$\models P \perp \alpha \leftrightarrow \langle P_1 \rangle \forall \mathbf{y}(R(\mathbf{y}) \rightarrow \alpha(\mathbf{y})).$$

It remains to explain the use of natural numbers in P_1, since ET-schemes do not explicitly perform arithmetical operations. However, any arithmetical operation may be encoded into the recursively enumerable control of an effective tree, and we may assume that the algorithm of it is able to simulate the use of counters. Another possibility is to consider P_1 as a flowchart with counters and recursion, and then translate it into an informal algorithm (see [3]) presented as an effective tree.

The case of $P \in \mathrm{FC}$ is a bit more difficult, since AFC-schemes cannot in general simulate operations on integers. However, we may distinguish between two situations: when the input generates infinitely or finitely many elements, and apply a technique similar to that used in [9]. In the former case, an AFC-scheme P_1' may behave exactly like P_1 above, since it is able to simulate counters with help of an infinite chain of elements generated by the input. Otherwise, P_1' is able to recognize that the number of elements accessible from the input is finite, say m. Thus, for some k, each computation of P must stop or loop (by repeating the same states) after at most m^k steps. (To be more precise, it may happen that a computation needs more than m^k steps to generate a value, but this is possible only if the computation visits twice the same state, executing a number of "superfluous" instructions. Hence, there is another computation, producing the same values within m^k steps, and it suffices to consider the shorter one.) The scheme P_1' can simulate all initial segments of computations of P of length m^k, using an additional k-ary array as an auxiliary memory composed from m^k cells, each containing one among m elements. It is a routine to verify that this amount of memory is sufficient to control the simulation.

To prove $L_\perp(K) \leqslant L_R(K)$, consider a new rich test scheme P_2 which simulates P and stops when it reaches a state satisfying α. We have

$P \perp \alpha$ iff there is no infinite computation of P_2.

Since looping is expressible in $L_R(K)$ (see Meyer and Winklmann, [6]), we are done. ▌

Now we observe that "during" can express first-order tests in deterministic flowcharts.

LEMMA 4. $L_{FO}(\mathrm{DFC}) \leqslant L_\perp(\mathrm{FC})$.

Proof. It suffices to prove that, for any $P \in \mathrm{DFC}$, there is a formula in $L_\perp(\mathrm{FC})$, equivalent to $\langle P \rangle$*true* (see [9]). Let $P(\mathbf{x}) \in \mathrm{DFC}$. We construct a new scheme $P_1 \in \mathrm{FC}$ to simulate P as follows. For each test $\alpha(\mathbf{x})$ occurring in P (note that there is only a finite number of tests in P), whenever P tests $\alpha(\mathbf{x})$, P_1 nondeterministically guesses an answer. A guess "yes" ("no") is announced by assigning equal values to auxiliary variables z_1^α and z_2^α (v_1^α and v_2^α) which are initially assigned different values. The variables are immediately (before proceeding to the next step of the simulation) assigned different values again. Thus, for each state of the computation tested for α, exactly one of the equations $z_1^\alpha = z_2^\alpha$, $v_1^\alpha = v_2^\alpha$ is true in exactly one state of the simulation. A simulation will be thus faithful if all its states satisfy

$$\alpha_1 : z_1^\alpha = z_2^\alpha \rightarrow \alpha(\mathbf{x}),$$

and

$$\alpha_2 : v_1^\alpha = v_2^\alpha \rightarrow \neg\alpha(\mathbf{x}).$$

P_1 uses two special variables y_1 and y_2 to indicate the end of simulation. They are assigned different values at the beginning and $y_1 := y_2$ is executed when P reaches its final state. Now, let φ be a conjunction of all formulas of the form $\alpha_1 \wedge \alpha_2$ for α being a test in P. The reader may verify that $\langle P \rangle$*true* is equivalent to $P_1 \perp \neg (\varphi \wedge y_1 = y_2)$.

Note that the above holds under the assumption that there are at least two elements accessible from the input to P. Let *two*$(\mathbf{x})$ be a first-order formula expressing this property, and let γ be equivalent to $\langle P \rangle$*true* over all one-element structures. Then, for all $\mathbf{x}$, $\langle P \rangle$*true* is equivalent to

$$(\mathit{two}(\mathbf{x}) \wedge P_1 \perp \neg(\varphi \wedge y_1 = y_2)) \vee (\neg \mathit{two}(\mathbf{x}) \wedge \gamma). \quad ▌$$

For the proof of our main negative results we need the following fact. Let *Parity* $= \{w \in \{0, 1\}^*$: the number of 1's in w is even$\}$.

LEMMA 5 (Furst, Saxe, and Sipser, [1]). No sequence of Boolean circuits of constant depth and polynomial size can accept *Parity*.

The next theorem demonstrates the power of first-order tests.

THEOREM 6. $L_{FO}(\mathrm{DFC}) \not\leq L(\mathrm{ET})$.

Proof. For $w \in \{0, 1\}^*$, $|w| = n$, let $A_w = \langle D_n, f, r\rangle$ be the structure with the domain $D_n = \{(i, j)\colon i = 0, 1; j = 1, ..., n\}$ and such that

$$\begin{aligned} f(1, j) &= (1, j) && \text{for all } j; \\ f(0, j) &= (0, j+1) && \text{for } j < n; \\ &= (0, n) && \text{for } j = n; \end{aligned}$$

$$r((i, j), (k, 1)) = \mathit{true} \text{ iff } i = 0, k = 1, j = 1,$$

and the jth symbol in w is 1.

It is an easy exercise to write a deterministic rich-test flowchart scheme P, such that if the input $\mathbf{a}$ assigns $(0, 1)$ to all variables then $A_w, \mathbf{a} \models \langle P\rangle$ *true* iff the number of 1's in w is even, i.e., iff $w \in \mathit{Parity}$. For this, P tests the formula $\exists y(r(x, y))$, once for each $x = (0, i)$, $i = 1, ..., n$. It remains to prove that no formula φ of $L(\mathrm{ET})$ has the property. Suppose the contrary and assume that φ is in a prenex normal form

$$Q_1 x_1 \cdots Q_k x_k \psi(\mathbf{x}, x_1, ..., x_k),$$

where ψ is an open formula of $L(\mathrm{ET})$.

Let $|w_1| = |w_2| = n$ and let $a_1, ..., a_k \in D_n$ be an arbitrary assignment to $x_1, ..., x_k$. Assume that, for all i, j, if $a_i = (1, j)$ then the jth symbols in w_1 and w_2 coincide. The reader may easily prove that in this case,

$$A_{w_1}, a_1, ..., a_k \models \psi \qquad \text{iff} \qquad A_{w_2}, a_1, ..., a_k \models \psi.$$

In other words, the Boolean value of $\psi(\mathbf{a}, a_1, ..., a_k)$, for a fixed $n = |w|$ and fixed $a_1, ..., a_k \in D_n$ is fully determined by the truth of $\exists x(r(x, a_i))$ in cases when the left coordinate of a_i is 1. Thus, $\psi(\mathbf{a}, a_1, ..., a_k)$ is a Boolean function on w, which depends on at most k of its symbols, and hence can be computed by a Boolean circuit of size at most 2^k. In particular, the size and also the depth of such a circuit does not depend on n.

Now, for any n, we may easily build up a Boolean circuit C_n computing the value of $\varphi(\mathbf{a})$ as a function in w. Each quantifier correspond to a level of $\wedge$ - or $\vee$ -gates, whence the depth of our circuit depends only on k, but not on n. In addition, the number of gates will be of order n^k—thus, polynomial in n. Clearly, the sequence $C_0, C_1, C_2, ...,$ accepts *Parity*, what contradicts Lemma 5. ∎

The result announced in the title of the paper is an immediate consequence of Lemma 4 and Theorem 6.

THEOREM 7. $L(\mathrm{FC}) < L_{\perp}(\mathrm{FC})$ *and* $L(\mathrm{ET}) < L_{\perp}(\mathrm{ET})$.

The following two facts follow from Lemma 4 and Theorems 3 and 6.

COROLLARY 8. *$L(\mathrm{ET}) < L(\mathrm{AET})$ and $L(\mathrm{FC}) < L(\mathrm{AFC})$.*

COROLLARY 9. *$L(K) < L_{FO}(K)$, $L_R(K)$ for all K in $\{\mathrm{FC}, \mathrm{DFC}, \mathrm{ET}, \mathrm{DET}\}$.*

Remark 1. It was known that $L(\mathrm{ET}) < L_R(\mathrm{ET})$ (see [4]), as well as that $L((D)\mathrm{FC}) < L(A(D)\mathrm{FC})$ (see [9]). However, by [5], $L(\mathrm{DET}) \equiv L(\mathrm{ADET})$.

Remark 2. Consider the operator $\perp_s$, defined by

$A, \mathbf{a} \models P \perp_s \alpha$ iff α is true of some state in every *finite* computation of P on $\mathbf{a}$.

It was observed by Stolboushkin [10] that

$$L\perp_s(\mathrm{FC}) \equiv L_R(\mathrm{FC}).$$

It is not known whether the above holds for $\perp$ or not.

ACKNOWLEDGMENTS

Thanks are due to Damian Niwiński, for turning my attention to the paper [1], also for A. J. Kfoury and two anonymous referees for their critical comments.

REFERENCES

1. M. FURST, J. B. SAXE, AND M. SIPSER, Parity, circuits and the polynomial time hierarchy, *Math. Systems Theory* **17** (1984), 13–27.
2. D. HAREL, Dynamic logic, *in* "Handbook of Philosophical Logic," Reidel, Dordrecht, 1983.
3. A. J. KFOURY, Definability by programs in first-order structures, *Theoret. Comput. Sci.* **25** (1983), 1–66.
4. A. R. MEYER AND R. PARIKH, Definability in dynamic logic, *J. Comput. System Sci.* **23** (1981), 279–298.
5. A. R. MEYER AND J. TIURYN, Equivalences among logics of programs, *J. Comput. System Sci.* **29**, No. 2 (1984), 160–170.
6. A. R. MEYER AND K. WINKLMANN, Expressing program looping in regular dynamic logic, *Theoret. Comput. Sci.* **18** (1982), 301–323.
7. G. MIRKOWSKA AND L. STAPP, AL can express progressive behaviour of programs, manuscript, 1982.
8. V. R. PRATT, Process logic, *in* "Proc. 6th ACM Sympos. on Principles of Programming Languages," 1979.
9. J. TIURYN AND P. URZYCZYN, Some relationships between logics of programs and complexity theory, *in* "Proc. 24th Found. of Comput. Sci.," 1983.
10. A. P. STOLBOUSHKIN, private letter, 1984.