JOURNAL OF COMPUTER AND SYSTEM SCIENCES 29, 160–170 (1984)

Equivalences among Logics of Programs*

ALBERT R. MEYER

Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, Massachusetts 02139

AND

JERZY TIURYN

Warsaw University, Warsaw, Poland

Received December 9, 1981; revised February 8, 1984

Several different first-order formal logics of programs—Algorithmic Logic, Dynamic Logic, and Logic of Effective Definitions—are compared and shown to be equivalent to a fragment of constructive $L_{\omega_1\omega}$. When programs are modelled as effective flowcharts, the logics of deterministic and nondeterministic programs are equivalent.

1. INTRODUCTION

A number of systems of formal logics which extend predicate calculus have been proposed for reasoning about sequential and nondeterministic programs. These include in rough chronological order:

1. The infinitary logic $L_{\omega_1\omega}$—suggested in [6] as a logic for programming.
2. μ-Calculus—defined in [23]; extended in [11, 21, 12]. Further references can be found in [4].
3. Algorithmic Logic (AL)—defined and developed in [22]; extended in [1].
4. Dynamic Logic (DL)—[20,10].
5. Programming Logic (PL)—[3].
6. Logic if Effective Definitions (LED)—[26].

Each of these logical systems actually represents a family of formal logics, instances of the family being determined by the choice of a few parameters. The principal parameter is the class of programs allowed in formulas. For example, in the case of DL some variants which have been considered are

* This work was supported in part by National Science Foundation Grants MCS 7719754 and MCS 8010707, and by a grant to the M.I.T. Laboratory for Computer Science by the IBM Corporation.

0022-0000/84 $3.00

—*regular* DL, in which programs are taken essentially to be finite, possibly nondeterministic, flowchart schemes with atomic formulas as tests and with *simple assignment statements* of the form $x := \tau$ where τ is a term;

—*regular-array* DL in which *array assignments* of the form $\tau_1 := \tau_2$ may also occur (cf. [17]);

—*regular* DL$^+$ in which for every finite flowchart α, the predicate LOOPS$_\alpha$, which asserts that α has an infinite computation, is included as an extra atomic formula (cf. [17]);[1]

—*recursive-call* DL in which programs are taken to be flowchart schemes containing recursive calls with arguments (cf. [8, 9, 4]).

In general, such different choices of the parameters lead to logics which differ in expressive power [14]. For example, [27] and [7] have recently shown that there is a formula of recursive-call DL, as well as one of regular-array DL, which is not equivalent to any formula of regular DL. On the other hand, [17] has shown that regular DL and regular DL$^+$ are equivalent in expressive power. Reference [16] has also demonstrated distinctions among the expressive powers of several other versions of DL and $L_{\omega_1\omega}$.

Thus there are genuine distinctions in the expressive and also model-theoretic and undecidability properties among the various instances of DL. These distinctions complicate the problem of comparing the six systems of programming logics listed above. For example, the bulk of the literature on AL defined that system in the particular version where programs are deterministic **while** schemes.[2] Since the original DL allowed nondeterministic schemes, it appeared that DL and AL represented genuinely distinct conceptions of programming logic.

Nevertheless, we claim that with appropriately matched parameters, DL, AL, and LED are actually equivalent systems. We believe that PL can be incorporated into this common framework as well, although its numerous "practical" features make it harder to grasp theoretically.

These systems can be described in more classical terminology as fragments of the constructive portion of $L_{\omega_1\omega}$, with the differenct instances of the systems characterized by various simple syntactic conditions on infinitary formulas. Thus, we argue that there is a common intuition which leads to the DL–AL–LED–PL framework for programming logic. In what follows we focus on this framework.[3]

In order to compare the DL–AL–LED–PL frameworks, we restrict ourselves to instances of these systems using what we regard as the mathematically most natural and robust notion of *computability* over arbitrary structures, namely, computability

[1] However, LOOPS$_\alpha$ may not occur as a test in a program.

[2] Only recently has an AL with nondeterministic schemes been considered by [18].

[3] Technical results of [19] for μ-calculus, and [16] for the constructive fragment of $L_{\omega_1\omega}$, show that these latter logics are incomparable in expressive power, and both are strictly greater in expressive power than logics in the DL–L–LED–PL framework unless the notion of program scheme is stretched unreasonably.

by *effective flowcharts*. Effective flowcharts may be described informally as generally infinite, nondeterministic, uninterpreted flowchart schemes whose basic instructions are assignment statements and whose basic tests consist of atomic formulas (including equations). Moreover, given a box of the flowchart, one can effectively find the instruction in that box, the number of edges leaving the box, and the endpoints of those edges. For technical convenience we require that the *signature* (i.e., set of symbols occurring, including variables) of any flowchart is *finite*.

A *state* provides an interpretation for all function, predicate, and variable symbols. Given a state, a nondeterministic flowchart defines a set of executable instruction sequences. The set of states in which execution of these instruction sequences can finally terminate is the set of *output* states for the given input state. Thus, any flowchart α defines a binary *input–output* relation R_α on states where

$$R_\alpha = \{(s, t) \mid \text{starting in the state } s, \text{ there is an executable sequence of instructions in } \alpha \text{ which finishes in output state } t\}.$$

If there is an infinite executable sequence starting in state s, then α is said to *loop from state s*. Formal definitions are available in [17, 15, 16, 26].

Friedman (cf. [24]) proposed a notion of *effective definitional scheme* as the most general model of effective computability in arbitrary structures. These may be described as the special case of effective flowcharts which are of the form

if P_1 **then** ASSIGN_1 **else**

if P_2 **then** ASSIGN_2 **else**

if $P_3 \ldots$

where P_i is a finite conjunction of atomic formulas or their negations, and ASSIGN_i is a sequence of assignment statements of the form $x := \tau$ with distinct variables x on the left-hand side of each statement in the sequence.

We can generalize effective definitional schemes to be nondeterministic. These nondeterministic effective definitional schemes can be informally described as the infinite parallel **OR** of statements of the form

if P_i **then** ASSIGN_i **else** ABORT **fi**,

where ABORT is a program with empty input–output relation, e.g., **while** *true* **do** *anything* **od**. Equivalent notions of universal classes of effective procedures on arbitrary structures have been proposed by many other researchers. In particular, it is easy to show

LEMMA 1. *The following classes of program schemes define the same class of input–output relations*:

1. *(Non)Deterministic effective flowcharts without array assignments (i.e., simple assignments only)*;

2. (*Nondeterministic*) *Effective definitional schemes*;

3. (*Non*)*Deterministic* finite *flowcharts without array assingments but with stacks.*

Similar definitions and lemmas can be given for the case that array assignments are allowed. These results indicate the invariance of the class of *computable input–output relations* between states defined by effective flowcharts.

Our main observation is that when effective flowcharts are taken as the notion of program in the programming logics listed above, then all can be reduced to a simple fragment of constructive $L_{\omega_1\omega}$ which we define next.

DEFINITION 2. Let L_{re} be the class of infinitary first-order formulas defined inductively as follows:

(a) if $P_1, P_2, \ldots$ is a recursively enumerable sequence of quantifier-free formulas of predicate calculus among which there are only finitely many free variables, then $\vee\{P_i \mid i \geqslant 1\}$ is a basic formula of L_{re};

(b) if p, q are formulas of L_{re}, then so are $\neg p$, $p \wedge q$, $p \vee q$, $\exists x[p]$, $\forall x[p]$.

THEOREM 3. *There is an effective procedure to translate a formula of any one of the following formal logics into an equivalent formula of any of the others*:

1. L_{re};

2. DL *of deterministic effective flowcharts without array assignments* (*i.e., only simple assignments occur*), *henceforth called* DDL-*w/o*-array;

3. DL *of deterministic effective flowcharts* (*i.e., array assignments may occur*), *henceforth called* DDL;

4. DL^+ *of nondeterministic effective flowcharts without array assignments, henceforth called* DL^+-*w/o*-array;

5. LED;

6. *Logic of nondeterministic effective definitional schemes* (*without array assignments*);

7. AL *of deterministic effective flowcharts without array assignments*;

8. AL *of deterministic effective flowcharts without array assignments and without the iteration quantifier* $\bigcap$.

We would like to emphasize that according to Theorem 3, DDL-w/o-array and DL^+-w/o-array are equivalent, viz., *adding nondeterminism to effective flowcharts does not increase the expressive power of the dynamic logic.*

Although in many programming situations nondeterminism is a significant addition, we can explain informally why it adds nothing to the logic of deterministic effective schemes: the rich control structure provided by arbitrary effective flowcharts enables a deterministic scheme α_d to "check the results" of any nondeterministic scheme α by carrying out a backtracking search. In particular, suppose α is a

nondeterministic effective flowchart without array assignments whose registers, i.e., free variables, are $\mathbf{x} = x_1, \ldots, x_n$. Then there is a deterministic effective flowchart α_d such that $\alpha_d(\mathbf{x}, \mathbf{y})$ halts iff $\alpha(\mathbf{x})$ can halt with the final contents of registers $\mathbf{x}$ set to $\mathbf{y}$. Thus the assertion that after $\alpha(\mathbf{x})$ halts, it is possible that some property $p(\mathbf{x})$ holds, is equivalent to the assertion that *there exist* $\mathbf{y}$ such that $\alpha_d(\mathbf{x}, \mathbf{y})$ halts and $p(\mathbf{y})$ holds. In this way, an existentially quantified assertion about a deterministic flowchart has the same expressive power as an assertion about a nondeterministic flowchart.

For more restricted control structures which cannot carry out the backtrack search, nondeterminism indeed makes a difference: [2] and independently [25] have recently shown that for *regular* programs, DDL is strictly less expressive than DL.

In the case that array assignments do occur in nondeterministic programs, our proof of Theorem 3 breaks down. The nondeterministic flowchart α may have registers $\mathbf{x}$ and also assignable arrays i.e., function symbols $\mathbf{f}$. Again, there is a deterministic "checking" flowchart α_d such that $\alpha_d(\mathbf{x}, \mathbf{f}, \mathbf{y}, \mathbf{g})$ halts iff $\alpha(\mathbf{x}, \mathbf{f})$ can halt with the final values of registers $\mathbf{x}$ and arrays $\mathbf{f}$ equal to $\mathbf{y}, \mathbf{g}$. Now, however, in order to reduce an assertion about α to one about α_d as above, it is necessary to existentially bind not only the $\mathbf{y}$ variables by also the function symbols $\mathbf{g}$. This second-order quantification exceeds the power of DL. But because the values of the arrays $\mathbf{g}$ differ only finitely from the values of the $\mathbf{f}$, the full power of second-order quantification is not necessary. If there are elements in the domain of interpretation which can serve to represent finite sets, it is possible to simulate this weak second-order quantification by first-order quantifiers. Any infinite set of finitely generated elements will serve to represent finite sets, so, aside from the pathological case of (essentially) finite domains, we can extend the theorem to nondeterministic effective flowcharts even with array assignements.

Namely, let Σ be some finite set of function symbols. A state is *n, Σ-infinite* iff there are n elements of the domain of the state such that the set of elements generated by applying the functions (which are the interpretations in the state of the symbols) in Σ to these n elements is infinite.

THEOREM 4. *For any $n > 0$ and finite set Σ of function symbols, there is an effective procedure to translate any formula p of the logics* 9–11 *below into a formula p' of L_{re} such that for every n, Σ-infinite sate s,*

$$s \models p \quad \text{iff} \quad s \models p'.$$

9. DL^+ *of nondeterministic effective flowcharts*;

10. *Logic of nondeterministic effective definitional schemes (with array assignments)*;

11. AL *of nondeterministic effective flowcharts without the iteration quantifier* $\bigcap$.

It remains open whether the unattractive n, Σ-infinity condition is eliminable from Theorem 4; it is also open whether the iteration quantifier $\bigcap$ makes a difference in the presence of nondeterministic programs. (The iteration quantifier is definable in

terms of the other operations of AL for deterministic programs, and has no particular significance for nondeterministic programs. Indeed, [18] omits it in that definition of nondeterministic AL.)

In the next section we present the main definitions among the logics 1–11, and prove Theorems 3 and 4.

2. Definitions and Proofs

All of the logics 1–11 are subsets of the following class L_{univ} of formulas which is obtained by combining the features of all the languages.

Definition 5. L_{univ} is defined inductively as follows:

(a) any atomic formula of predicate calculus with equality is a formula of L_{univ};

(b) if α is an effective flowchart, then $LOOPS_\alpha$ is a formula of L_{univ};

(c) if p, q are formulas of L_{univ}, then so are $\neg p$, $p \wedge q$, $p \vee q$, $\exists x[p]$, $\forall x[p]$;

(d) if $P_1, P_2, \ldots,$ is an r.e. sequence of formulas of L_{univ}, then so are $\vee\{P_i \mid i \geqslant 1\}$ and $\wedge\{P_i \mid i \geqslant 1\}$;

(e) if α is an effective flowchart and p is a formula of L_{univ}, then so are $\langle\alpha\rangle p$ and $[\alpha]p$;

(f) if α is an effective flowchart and p is a formula of L_{univ} then so are $(\bigcap \alpha)p$ and $(\bigcup \alpha)p$.

We remark that rigorously formulating the syntax of L_{univ} requires introducing an *effective system of notations*, i.e., Gödel numbers, for constructive infinitary formulas as in [13]. Methods for doing this are well known, and we leave it to the concerned reader to supply the details.

Whether a state s *satisfies* a formula p of L_{univ}, denoted $s \models p$, is defined in the usual way for p of the form (a), (c), or (d) above.

For case (b), $s \models LOOPS_\alpha$ iff α loops from state s.

For case (e), $s \models \langle\alpha\rangle p$ iff $t \models p$ for *some* state t such that $(s, t) \in R_\alpha$; $s \models [\alpha]p$ iff $t \models p$ for *all* states t such that $(s, t) \in R_\alpha$.

Case (f) covers the *iteration quantifiers* of AL. $s \models (\bigcap \alpha)p$ iff $s \models \langle\alpha^n\rangle p$ for *all* $n \geqslant 0$, and $s \models (\bigcup \alpha)p$ iff $s \models \langle\alpha^n\rangle p$ for *some* $n \geqslant 0$, where α^n is the effective flowchart specifying n consecutive executions of α. Thus $(\bigcup \alpha)p$ is equivalent in DL formalism to $\langle\alpha^*\rangle p$, where α^* is an effective flowchart such that R_{α^*} is the reflexive transitive closure of R_α. Algorithmic Logic, however, does not explicitly use the *-construct.

This defines the *semantics* of L_{univ}.

L_{re} is easily embeddable in all of the logics of Theorem 3, and all are obviously embeddable into one of DDL or DL^+-w/o-array, so we give precise definitions and proofs only for these latter two logics.

DEFINITION 6. DDL is the class of formulas defined by rules (a), (c), (e) of Definition 5 such that the flowcharts α of rule (e) are deterministic. DL^+-w/o-array is the class of formulas defined by rules (a), (b), (c), (e) such that the flowcharts α of rule (e) do not contain array assignments.

To prove Theorem 3, we discribe translations between L_{re} and DDL, and between L_{re} and DL^+-w/o-array.

The translation from L_{re} actually takes formulas of L_{re} into the intersection of DDL and DL^+-w/o-array. It is obtained trivially from the observation that the atomic formula $\vee\{P_i \mid i \geqslant 1\}$ of L_{re} is equivalent to $\langle\alpha\rangle$*true* where α is the effective flowchart

if P_1 **then** $x := x$ **else**

if P_2 **then** $x := x$ **else**

if P_3 **then** $x := x$ **else**....

The translation from DDL to L_{re} is based on

LEMMA 7. *The following formulas are valid for any flowchart α and formula p of* L_{univ}:

1. $\langle\alpha\rangle(p \vee q) \equiv (\langle\alpha\rangle p \vee \langle\alpha\rangle q)$;

2. $\langle\alpha\rangle \exists x[p] \equiv \exists z[\langle\alpha\rangle(p[z/x])]$, *where z does not occur in α or p, and $p[z/x]$ is the result of substituting z for x in p.*

In addition, the following formula is valid for any deterministic *flowchart α and formula p of* L_{univ}:

3. $\langle\alpha\rangle \neg p \equiv (\langle\alpha\rangle \mathit{true} \wedge \neg\langle\alpha\rangle p)$.

The equivalences of Lemma 7 allow one to "move the $\langle\ \rangle$'s in" thereby converting an arbitrary formula of DDL into an equivalent formula built solely by first-order constructs, i.e., the rules of Definition 5(c), starting from formulas of the form $\langle\beta_1\rangle \cdots \langle\beta_n\rangle P$ where P is an atomic formula of predicate calculus. But a formula of the form $\langle\beta_1\rangle \cdots \langle\beta_n\rangle P$ is equivalent to an r.e. disjunction of formulas $\langle\alpha_i\rangle P$ where α_i ranges over the terminating instruction sequences of the program $\beta_1;\ldots;\beta_n$. Each formula $\langle\alpha_i\rangle P$, where α_i is a *finite* sequence of assignments and atomic tests and P is quantifier-free, can be effectively translated into an equivalent quantifier-free formula of predicate calculus (cf. [20, 16]). In this way DDL translates into L_{re}.

The translation from DL^+-*w*/o-array into L_{re} proceeds by induction on the definition of DL^+. The only interesting case in the basis of the induction is for formulas of the form LOOPS_α. But LOOPS_α is obviously equivalent to the r.e. conjunction of the quantifier-free first-order formulas which assert that there is an executable instruction sequence of length n in α, for each $n > 0$.

The only non-trivial case in the rest of the inductive definition of the translation is $\langle\ \rangle$-elimination. Let α be a nondeterministic effective flowchart without array

assignments, and let p be a formula of DL$^+$-w/o-array. We must translate $\langle\alpha\rangle p$ into a formula of L_{re}. By induction, we may assume there is a formula q of L_{re} equivalent to p.

Now the equivalence between flowcharts and effective definitional schemes stated in Lemma 1 is an effective one, so we may assume α is an infinite **OR** of statements

$$\textbf{if } P_l \textbf{ then } x_1 := \tau'_{l,1}; \ldots; x_n := \tau'_{l,n} \textbf{ else ABORT fi}$$

where P_l is quantifier-free and $\tau_{i,j}$ are terms with free variables among $x_1, \ldots, x_n$. Define $\tau_{i,1} = \tau'_{i,1}$ and $\tau_{i,j+1} = [\tau_{i,1} \cdots \tau_{i,j}/x_1, \ldots, x_j]\, \tau'_{i,j+1}$.

Then α started in s can terminate with the values of $x_1, \ldots, x_n$ equal to elements $a_1, \ldots, a_n$ iff there is an i such $s \models P_i$ and the value of $\tau_{i,j}$ in s is a_j.

Let $y_1, \ldots, y_n$ be new variables which occur neither in α nor in q. The reader can easily check that $\langle\alpha\rangle p$ is equivalent in all states to

$$\exists y_1 \cdots \exists y_n [\vee_i \{P_i \wedge (\wedge_{j<n}\, y_j = \tau_{i,j})\} \wedge q[y_1, \ldots, y_n/x_1, \ldots, x_n]]. \tag{1}$$

We remark that introducing quantifiers in formula (1), or indeed any such formula which accomplishes $\langle\ \rangle$-elimination, is unavoidable. This follows from the fact that the quantifier-free fragment of Deterministic DL$^+$-w/o-array, which is equivalent to quantifier-free L_{re}, is strictly weaker than the quantifier-free fragment of DL$^+$-w/o-array (cf. [17]).

This completes the proof of Theorem 3.

In proving Theorem 4, we note that all of the logics 9–11 are no more expressive than DL$^+$. We therefore only describe the translation of DL$^+$ into L_{re}.

As in the proof of Theorem 3, the translation is given inductively. The only interesting case is $\langle\ \rangle$-elimination for a formula of the form $\langle\alpha\rangle p$.

By induction, let q be an L_{re} formula equivalent over all n, Σ-infinite states to the DL$^+$ formula p. As in the previous proof, we may assume α is an infinite parallel **OR** of finite deterministic programs α_i of the form

$$\textbf{if } P_i \textbf{ then ASSIGN}_i \textbf{ else ABORT fi}$$

where ASSIGN$_i$ is a finite sequence of array assignments.

So $\langle\alpha\rangle p$ is equivalent to the L_{univ} formula $\langle\alpha\rangle\ q$ which is equivalent to

$$\vee_i \langle\alpha_i\rangle\, q. \tag{2}$$

It is not hard to show that any formula $\langle\alpha_i\rangle\, q$ is equivalent to a formula of L_{re}, but this still leaves the difficulty that (2) is an infinite disjuction of L_{re}, not first-order, formulas, and L_{re} is not closed under infinite disjunctions. We could eliminate this difficulty if the integer variable i in $\langle\alpha_i\rangle\, q$ could somehow be taken as a variable of DL, for then the infinite disjunction over i in 2 could simply be replaced by an existential quantification of i. With the aid of the hypothesis of n, Σ-infinity, we will essentially accomplish this.

Let $\mathbf{y} = y_1, \ldots, y_n$, and z be $n+1$ individual variables which occur neither in α nor in q_i, and choose some effective enumeration $\tau_1(\mathbf{y}), \tau_2(\mathbf{y}), \ldots$ of all the terms over $\mathbf{y} \cup$ (signature(α)-variables(α)), i.e., the terms with function symbols from α whose only variables are from $\mathbf{y}$.

For $k, i \geqslant 1$, let $D_{i,k}(\mathbf{y}, z)$ be a quantifier-free formula of predicate calculus which expresses the following property: "z is the value of the kth term (in the above enumeration), there are exactly i distinct values among those first k terms, and k is the least integer with the above two properties."

By definition of $D_{i,k}$, for every state s there is *at most* one pair of integers $i, k \geqslant 1$ such that $s \models D_{i,k}(\mathbf{y}, z)$. Moreover, for every n, Σ-infinite state s and for arbitrary $i \geqslant 1$ there exists $k \geqslant 1$, such that $s \models \exists \mathbf{y}\, \exists z [D_{i,k}(\mathbf{y}, z)]$.

Let α' be the infinite **OR** of statements $\alpha'_{i,k}$ where

$$\alpha'_{i,k} = \textbf{if } P_i \wedge D_{i,k}(\mathbf{y}, z) \textbf{ then } \mathrm{ASSIGN}_i \textbf{ else } \mathrm{ABORT} \textbf{ fi}$$

where $\mathbf{y}, z$ do not occur free in α or q.

Since at most one test $D_{i,k}(\mathbf{y}, z)$ is true in any state, α' is equivalent to a *deterministic* program, and therefore $\exists \mathbf{y} z \langle \alpha' \rangle q$ is translatable to a formula q' of L_{re} by Theorem 3.

We claim that

$$s \models \langle \alpha \rangle q \qquad \text{iff } s \models \exists \mathbf{y}, z \langle \alpha' \rangle q \tag{3}$$

for every $n, \sum$-infinite state s. This verifies that q' is the desired translation of $\langle \alpha \rangle q$ into an L_{re} formula.

To prove the claim from left to right, suppose $s \models \langle \alpha \rangle q$ and s is n, Σ-infinite. Then $s \models \langle \alpha_i \rangle q$ for some i. So $s \models P_i \wedge \langle \mathrm{ASSIGN}_i \rangle q$ by definition of α_i. But since s is n, Σ-infinite, there is a $k \geqslant 1$ such that $s \models \exists \mathbf{y}, z \cdot D_{i,k}(\mathbf{y}, z)$. Moreover, $\mathbf{y}, z$ do not occur in P_i or $\langle \mathrm{ASSIGN}_i \rangle q$, so

$$s \models \exists \mathbf{y}, z [P_i \wedge D_{i,k}(\mathbf{y}, z) \wedge \langle \mathrm{ASSIGN}_i \rangle q].$$

But this is equivalent to

$$s \models \exists \mathbf{y}, z [\langle \alpha'_i \rangle q],$$

which immediately implies that

$$s \models \exists \mathbf{y}, z \langle \alpha' \rangle q$$

as required. The proof of (3) from right to left follows similarly, without even requiring the n, Σ-infinite hypothesis.

3. Conclusion

Having reduced essentially all the various programing logics to the L_{re} fragment of infinitary logic, it is easy to deduce a body of model-theoretic and undecidability results about programming logic from known results for infinitary logic. Moreover, the reduction to L_{re} is sufficiently straightforward that various infinitary proof-theoretic results can also be carried over directly to programming logic.

We interpret these results as evidence that no very new model-theoretic or recursion-theoretic issues arise from logics of effective flowcharts on first-order structures.

Nevertheless, we believe that the problem of developing formal systems for reasoning about programs offers significant challenges in at least two directions. First, to be true to the purpose for which high-level programming languages were originally developed and continue to be developed—namely, for economy and ease in the expression of algorithms—it is important to develop proof methods for dealing with high-level programs as textual objects. This has in fact been the focus of the bulk of the literature on program correctness, although many of the complex features of modern programming languages have yet to be adequately addressed (cf. [28]). In our treatment we assumed in effect that the high-level programs had already been transformed into effective flowcharts, and thereby we avoided the challenge of developing a proof theory.

Acknowledgments

We are grateful to Piotr Berman and Rohit Parikh for their detailed comments.

References

1. L. Banachowski, *et al.*, An introduction to algorithmic logic; Metamathematical investigations in the theory of programs, *in* "Mathematical Foundations of Computer Science," (A. Mazurkiewicz and Z. Pawlak, Eds.), Banach Center Publications, Vol. 2, pp. 7–100, Polish Scientific Publishers, Warsaw, 1977.
2. P. Berman, J. Halpern, and J. Tiuryn, On the power of nondeterminism in dynamic logic, *in* "Proceedings, Ninth Colloquium on Automata, Languages and Programming," Lecture Notes in Computer Science, pp. 48–60, Springer-Verlag, New York/Berlin, 1982.
3. R. L. Constable and M. J. O'Donnell, "A Programming Logic," Winthrop, 1978.
4. J. De Bakker, "Mathematical Theory of Program Correctness," Prentice–Hall, Englewood Cliffs, N.J., 1980.
5. E. Engeler, Algorithmic properties of structures, *Math. Systems Theory* **1** (1967), 183–195.
6. E. Engeler, Algorithmic logic, *in* "Mathematical Centre Tracts (63)" (J. De Bakker, Ed.), pp. 57–85, Amsterdam, 1975.
7. M. M. Erimbetov, On the expressive power of programming logics (Russian), *in* "Proceedings, Conference on Research in Theoretical Programming," pp. 49–68, ALMA-ATA, 1981.
8. J. H. Gallier, Nondeterministic flowchart programs with recursive procedures: Semantics and correctness, *Theoret. Comput. Sci.* **13** (2) (1981), 193–224.

9. S. GREIBACH, "Theory of Program Structures: Schemes, Semantics, Verification," Lecture Notes in Computer Science No. 36, Springer-Verlag, New York/Berlin, 1975.
10. D. HAREL, "First-Order Dynamic Logic," Lecture Notes in Computer Science No. 68, Springer-Verlag, New York/Berlin, 1979.
11. P. HITCHCOCK AND D. PARK, Induction rules and termination proofs, *in* "Automata, Languages and Programming" (M. Nivat, Ed.)., pp. 225–251, Amer. Elsevier, New York, 1973.
12. D. KOZEN, Results on the propositional μ-calculus, *in* "Proceedings, 9th International Colloquium on Automata, Languages, and Programming," pp. 348–359, Springer-Verlag, New York/Berlin, 1982.
13. E. G. K. LOPEZ-ESCOBAR, Remarks on an infinitary language with constructive formulas. *J. Symbolic Logic* **32** (3) (1967), 305–317.
14. A. R. MEYER, Ten thousand and one logics of programming. *EATCS Bull.* 11–29; M.I.T. LCS TM 150, MIT Laboratory for Computer Science, Cambridge, Mass., February 1980.
15. A. R. MEYER AND J. Y. HALPERN, Axiomatic definitions of programming languages: A theoretical assessment, *J. Assoc. Comput. Mach.* **29** (2) (1982), 555–576.
16. A. R. MEYER AND R. PARIKH, Definability in dynamic logic, *J. Comput. System Sci.* **23** (1981), 279–298.
17. A. R. MEYER, AND K. WINKLMANN, Expressing program looping in regular dynamic logic, *Theoret. Comput. Sci.* **18** (3) (1982), 301–323.
18. G. MIRKOWSKA, Complete axiomatization of algorithmic properties of program schemes with bounded nondeterministic interpretations, *in* "Proceedings, 12th Annual ACM Symposium on Theory of Computing (1980)," pp. 14–21.
19. D. PARK, Finiteness is mu-ineffable, *Theoret. Comput. Sci.* **3** (1976), 173–181.
20. V. PRATT, Semantical considerations on Floyd–Hoare logic, *in* "Proceedings, 17th Symposium on Foundations of Computer Science, Houston, Texas, October 1976," pp. 109–121.
21. V. PRATT, A decidable μ-calculus (preliminary report), *in* "Proceedings, 22nd IEEE Symposium on the Foundations of Computer Science 1981," pp. 421–427.
22. A. SALWICKI, Formalized algorithmic languages, *Bull. Acad. Polon. Sci. Sér. Math. Astr. Phys.* **18** (1970), 227–232.
23. D. S. SCOTT AND J. DE BAKKER, A theory of programs, unpublished seminar notes, IBM, Vienna, 1969.
24. J. C. SHEPHERDSON, Computing over abstract structures: Serial and parallel procedures and Friedman's effective definitional schemes, *in* "Logic Colloquium 73" (Shepherdson and Rose, Eds.), pp. 445–513, North-Holland, Amsterdam, 1973.
25. A. P. STOLBOUSHKIN AND M. A. TAITSLIN, Deterministic dynamic logic is strictly weaker than dynamic logic, *Inform. and Control* **57** (1983), 48–55.
26. J. TIURYN, A survey of the logic of effective definitions, *in* "Proceedings, Workshop on Logics of Programs" E. Engeler, Ed.), pp. 198–245, Lecture Notes in Computer Science No. 125, Springer-Verlag, New York/Berlin, 1981.
27. J. TIURYN, Unbounded program memory adds to expressive power of first-order dynamic logic, *in* "Proceedings 22nd IEEE Symposium on Foundations of Computer Science, 1981," pp. 335–339.
28. B. A. TRAKHTENBROT, J. Y. HALPERN, AND A. R. MEYER, From denotational to operational and axiomatic semantics for ALGOL-like languages: An overview, *in* "Logics of Programs, Proceedings" (Clarke and Kozen, Eds.), Lecture Notes in Computer Science, Springer-Verlag, New York/Berlin, in press.