The Semantics of Predicate Logic as a Programming Language

M. H. VAN EMDEN AND R. A. KOWALSKI

University of Edinburgh, Edinburgh, Scotland

ABSTRACT Sentences in first-order predicate logic can be usefully interpreted as programs In this paper the operational and fixpoint semantics of predicate logic programs are defined, and the connections with the proof theory and model theory of logic are investigated It is concluded that operational semantics is a part of proof theory and that fixpoint semantics is a special case of model-theoretic semantics

KEY WORDS AND PHRASES predicate logic as a programming language, semantics of programming languages, resolution theorem proving, operational versus denotational semantics, SL-resolution, fixpoint characterization

CR CATEGORIES: 4 22, 5 21, 5 24

1. *Introduction*

Predicate logic plays an important role in many formal models of computer programs [3, 14, 17]. Here we are concerned with the interpretation of predicate logic as a programming language [5, 10]. The PROLOG system (for PROgramming in LOGic), based upon the procedural interpretation, has been used for several ambitious programming tasks (including French language question answering [5, 18], symbolic integration [9], plan formation [24], theorem proving, speech recognition, and picture interpretation). In this paper we ignore the practical aspects of programming in logic and investigate instead the semantics of predicate logic regarded as a programming language. We compare the resulting semantics with the classical semantics studied by logicians.

Two kinds of semantics [22], operational and fixpoint, have been defined for programming languages. Operational semantics defines the input-output relation computed by a program in terms of the individual operations evoked by the program inside a machine. The meaning of a program is the input-output relation obtained by executing the program on the machine. As a machine independent alternative to operational semantics, fixpoint semantics [1, 6, 17, 22] defines the meaning of a program to be the input-output relation which is the minimal fixpoint of a transformation associated with the program. Fixpoint semantics has been used [6, 7, 15, 17] to justify existing methods for proving properties of programs and to motivate and justify new methods of proof.

Logicians distinguish between the syntax and the semantics of formal languages. Syntax deals with the formal part of language in abstraction from its meaning. It deals not only with the definition of well-formed formulas, syntax in its narrow sense, but also with the broader study of axioms, rules of inference, and proofs, which constitutes proof theory. Semantics, on the other hand, deals with the interpretation of language and includes such notions as meaning, logical implication, and truth. Church's *Introduction to Mathematical Logic* [4] contains a thorough discussion of the respective roles of syntax and semantics.

This work was supported by the U K Science Research Council

Authors' present addresses M H van Emden, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1, R A Kowalski, Department of Computation & Control, Imperial College, 180 Queens Gate, London SW7, United Kingdom

We use the interpretation of predicate logic as a programming language in order to compare the notions of operational and fixpoint semantics of programming languages with the notions of syntax and semantics of predicate logic. We show that operational semantics is included in the part of syntax concerned with proof theory and that fixpoint semantics is a special case of model-theoretic semantics. With this interpretation of operational semantics as syntax and fixpoint semantics as semantics, the equivalence of operational and fixpoint semantics becomes a special case of Gödel's completeness theorem.

This paper is concerned with the analysis and comparison of some of the most basic notions of logic and computation As a by-product it is virtually self-contained and requires only a general knowledge of logic but no special familiarity with the operational and fixpoint semantics of programming languages.

2. *A Syntax of Well-Formed Formulas*

It is convenient to restrict attention to predicate logic programs written in clausal form. Such programs have an especially simple syntax but retain all the expressive power of the full predicate logic.

A *sentence* is a finite set of clauses.

A *clause* is a disjunction $L_1 \vee \cdots \vee L_n$ of *literals* L_i, which are *atomic formulas* $P(t_1, \ldots, t_m)$ or the negations of atomic formulas $P(t_1, \ldots, t_m)$, where P is a predicate symbol and t_i are terms. Atomic formulas are *positive* literals. Negations of atomic formulas are *negative* literals.

A *term* is either a *variable* or an expression $f(t_1, \ldots, t_m)$ where f is a *function symbol* and t_i are terms. *Constants* are 0-ary function symbols.

A set of clauses $\{C_1, \ldots, C_n\}$ is interpreted as the conjunction, C_1 and ... and C_n. A clause C containing just the variables $x_1, \ldots, x_m$ is regarded as universally quantified:

$$\text{for all } x_1, \ldots, x_m \; C$$

For every sentence S_1 in predicate logic there exists a sentence S_2 in clausal form which is satisfiable if and only if S_1 is. For this reason, all questions concerning the validity or satisfiability of sentences in predicate logic can be addressed to sentences in clausal form. Methods for transforming sentences into clausal form are described in [16].

We have defined that part of the syntax of predicate logic which is concerned with the specification of well-formed formulas. Aspects of syntax concerned with proof theory are dealt with in the next two sections.

3. *The Procedural Interpretation*

It is easiest to interpret procedurally sets of clauses which contain at most one positive literal per clause. Such sets of clauses are called *Horn sentences*. We distinguish three kinds of *Horn clauses*.

(1) $\square$ the *empty clause*, containing no literals and denoting the truth value *false*, is interpreted as a *halt statement*.

(2) $\bar{B}_1 \vee \cdots \vee \bar{B}_n$ a clause consisting of no positive literals and $n \geq 1$ negative literals is interpreted as a *goal statement*.

(3) $A \vee \bar{B}_1 \vee \cdots \vee \bar{B}_n$ a clause consisting of exactly one positive literal and $n \geq 0$ negative literals is interpreted as a *procedure declaration*. The positive literal A is the *procedure name* and the negative literals are the *procedure body*. Each negative literal $\bar{B}_i$ in the procedure body is interpreted as a *procedure call*. When $n = 0$ the procedure declaration has an empty body and is interpreted as an unqualified assertion of fact

In the procedural interpretation a set of procedure declarations is a program. Computation is initiated by an initial goal statement, proceeds by using procedure declarations

to derive new goal statements from old goal statements, and terminates with the derivation of the halt statement Such derivation of goal statements is accomplished by resolution [20], which is interpreted as *procedure invocation.* Given a selected procedure call $\bar{A}_i$ inside the body of a goal statement

$$\bar{A}_1 \vee \cdots \vee \bar{A}_{i-1} \vee \bar{A}_i \vee \bar{A}_{i+1} \vee \cdots \vee \bar{A}_n$$

and given a procedure declaration

$$A' \vee \bar{B}_1 \vee \cdots \vee \bar{B}_m, \quad m \geq 0$$

whose name matches the selected procedure call (in the sense that some most general substitution θ of terms for variables makes A_i and A' identical), resolution derives the new goal statement

$$(\bar{A}_1 \vee \cdots \vee \bar{A}_{i-1} \vee \bar{B}_1 \vee \cdots \vee \bar{B}_m \vee \bar{A}_{i+1} \vee \cdots \vee \bar{A}_n)\ \theta.$$

In general, any derivation can be regarded as a computation and any refutation (i.e. derivation of $\square$) can be regarded as a successfully terminating computation. However only goal oriented resolution derivations correspond to the standard notion of computation. Such a *goal-oriented derivation* from an initial set of Horn clauses **A** and from an initial goal statement C_1 in **A** is a sequence of goal statements $C_1, \ldots, C_n$ such that each C_i contains a single selected procedure call and C_{i+1} is obtained from C_i by procedure invocation relative to the selected procedure call in C_i using a procedure declaration in **A**

In model elimination [13], ordered linear resolution [19], and SL-resolution [12], the selection of procedure calls is governed by the *last in/first out rule*: A goal statement is treated as a stack of procedure calls. The selected procedure call must be at the top of the stack. The new procedure calls which by procedure invocation replace the selected procedure call are inserted at the top of the stack. The more general notion of goal oriented derivation defined above corresponds to computation with coroutines [10]. Computation with asynchronous parallel processes is obtained by using the splitting rule [2, 8, 23].

Predicate logic is a *nondeterministic* programming language: Given a single goal statement, several procedure declarations can have a name which matches the selected procedure call. Each declaration gives rise to a new goal statement. A *proof procedure* which sequences the generation of derivations in the search for a refutation behaves as an *interpreter* for the program incorporated in the initial set of clauses. These and other aspects of the procedural interpretation of Horn clauses are investigated in greater detail elsewhere [10]

The procedural interpretation has also been investigated for non-Horn clauses [11]. However, in this paper we restrict ourselves to Horn clauses.

Example The following two clauses constitute a program for appending two lists. The term cons(x,y) is interpreted as a list whose first element, the *head*, is x and whose *tail*, y, is the rest of the list. The constant nil denotes the empty list. The terms u, x, y, and z are variables. Append(x,y,z) denotes the relationship: z is obtained by appending y to x.

(1) Appen(nil,x,x).

(2) Append(cons(x,y),z,cons(x,u)) $\vee$ $\overline{\text{Append}}$(y,z,u)

To compute the result of appending the list cons(b,nil) to the list cons(a,nil), the program is activated by the goal statement

(3) $\overline{\text{Append}}$(cons(a,nil),cons(b,nil),v),

where v is a variable and a and b are constants, the "atoms" of the lists With this goal statement the program is deterministic. With a goal directed theorem prover as interpreter, the following computation ensues:

C_1 = $\overline{\text{Append}}$(cons(a,nil),cons(b,nil),v),
C_2 = $\overline{\text{Append}}$(nil,cons(b,nil),w) θ_1,
C_3 = $\square$ θ_2,

where θ_1 is the substitution $v := \text{cons}(a,w)$ and θ_2 is $w := \text{cons}(b,\text{nil})$. The result of the computation is the value of v in the substitution $\theta_1\theta_2$, which is $v := \text{cons}(a,\text{cons}(b,\text{nil}))$.

4. *Operational Semantics*

To define an operational semantics [22] for a programming language is to define an implementation independent interpreter for it. For predicate logic the proof procedure behaves as such an interpreter.

We regard the terms containing no variables which can be constructed from the constants and other function symbols occurring in a set of clauses **A** as the *data structures* which the program, incorporated in **A**, manipulates. The set of all such terms is called the *Herbrand universe H* determined by **A**. Every n-ary predicate symbol P occurring in **A** denotes an n-ary relation over the Herbrand universe of **A**. We call the n-tuples which belong to such relations *input-output tuples* and the relations themselves *input-output relations*.

Given a specific inference system, the operational semantics determines a unique denotation for P: The n-tuple $(t_1, \ldots, t_m)$ belongs to the denotation of P in **A** iff $\mathbf{A} \vdash P(t_1, \ldots, t_n)$, where $X \vdash Y$ means that there exists a derivation of Y from X. For resolution systems we employ the convention that $X \vdash Y$ means that there exists a refutation of the sentence in clausal form corresponding to $X \,\&\, \bar{Y}$. We use the notation

$$\mathbf{D}_1(P) = \{(t_1, \ldots, t_n) : \mathbf{A} \vdash P(t_1, \ldots, t_n)\}$$

for the *denotation* of P in **A** as determined by *operational semantics*.

It needs to be emphasized that only goal oriented inference systems correspond to the standard notion of operational semantics, where procedure calls are replaced by procedure bodies. In theory, however, any inference system for predicate logic specifies, implicitly at least, an abstract machine which generates exactly those derivations which are determined by the given inference system.

Notice that in our treatment predicate logic programs compute relations. The relations computed are denoted by predicate symbols in the defining set of clauses **A**. Those special relations which are functions are also denoted by predicate symbols. The function symbols occurring in **A** do not denote functions computed by the program but construct the data structures which are the input and output objects of the relations (or functions) computed.

It is a significant application of the proof theory of resolution systems to the computation theory of predicate logic programs that if **A** is consistent and $\mathbf{A} \vdash P(t_1, \ldots, t_n)$ then there exists a resolution refutation of $\mathbf{A} \,\&\, \bar{P}(x_1, \ldots, x_n)$ in which the variables $x_1, \ldots, x_n$ are eventually instantiated to terms which have $t_1, \ldots, t_n$ as an instance. More generally, if $\mathbf{A} \vdash P(t_1, \ldots, t_n)$, then for any subset of the arguments $t_1, \ldots, t_n$ of P there exists a computation which accepts those arguments of P as input and computes the remaining arguments as output. A useful practical consequence of this fact is that a predicate logic program can first be written to test that a given relationship holds among the members of an n-tuple of objects but can later be used to generate, from some subset of objects in the n-tuple given as input, the remaining objects in the n-tuple as output. See, for example, the goal statement 3(a) below. Another important consequence is that variables occurring in input or output can be used to represent incompletely specified data. See, for example, the goal statement 3(b) below. It is these considerations which motivate the terminology "input-output relation" for the relation denoted by a predicate symbol in a set of clauses.

Given a consistent set of clauses **A** representing a program and given a goal statement C, the Herbrand universe for **A** can be different from the Herbrand universe for the set of clauses $\mathbf{A} \cup \{C\}$. Although this is an interesting case to consider, we assume for simplicity that it does not arise and that C contains only constant symbols and function symbols occurring in **A**. Similarly we assume that **A** always contains at least one constant symbol.

Example. The program for appending lists can be activated by the goal statement:
(3a) $\overline{\text{Append}}(x,\text{cons}(a,y),\text{cons}(a,\text{cons}(b,\text{cons}(a,\text{nil}))))$,
where a, b, and nil are constants, and x and y are variables. With this goal statement the program behaves nondeterministically: There are two computations, one ends with $x :=$ nil, $y :=$ cons(b,cons(a,nil)), and the other ends with $x :=$ cons(a,cons(b,nil)), $y :=$ nil. Activated by a goal statement with this pattern of constants and variables, the program checks whether a particular item occurs in the given list and gives a different computation for each different occurrence. For each occurrence of the item, it determines the list of items preceding the given occurrence as well as the list following it.

Example. The program for appending can also be activated by the goal statement:
(3b) $\overline{\text{Append}}(\text{cons}(b,\text{nil}),y,z)$,
where b and nil are constants and y and z are variables. Starting from this goal statement there is one computation. It ends with $z :=$ cons(b,y), which can be interpreted as stating that z is the list whose head is b and whose tail is the unspecified input y.

5. *Model-Theoretic Semantics*

There is general agreement among logicians concerning the semantics of predicate logic. This semantics provides a simple method for determining the denotation of a predicate symbol P in a set of clauses **A**:

$$\mathbf{D}_2(P) = \{(t_1, \ldots, t_n) : \mathbf{A} \models P(t_1, \ldots, t_n)\},$$

where $X \models Y$ means that X logically implies Y. $\mathbf{D}_2(P)$ is the *denotation* of **P** as determined by *model-theoretic semantics*.

The completeness of first-order logic means that there exist inference systems such that derivability coincides with logical implication; i.e. for such inference systems $X \vdash Y$ iff $X \models Y$.

The equivalence of operational and model-theoretic semantics $\mathbf{D}_1(P) = \mathbf{D}_2(P)$ is an immediate consequence of the completeness of the inference system which determines $\mathbf{D}_1$.

In order to make a comparison of the fixpoint and model-theoretic semantics, we need a more detailed definition of $\mathbf{D}_2$. For this purpose we define the notions of Herbrand interpretation and Herbrand model.

An expression (term, literal, clause, set of clauses) is *ground* if it contains no variables. The set of all ground atomic formulas $P(t_1, \ldots, t_n)$, where P occurs in the set of clauses **A** and $t_1, \ldots, t_n$ belong to the Herbrand universe H of **A**, is called the *Herbrand base* $\hat{H}$ of **A**. A *Herbrand interpretation* I of **A** is any subset of the Herbrand base of **A**. A Herbrand interpretation simultaneously associates, with every n-ary predicate symbol in **A**, a unique n-ary relation over H. The relation $\{(t_1, \ldots, t_n) : P(t_1, \ldots, t_n) \in I\}$ is *associated* by I with the predicate symbol P in **A**.

(1) A ground atomic formula A is *true* in a Herbrand interpretation I iff $A \in I$.
(2) A ground negative literal $\bar{A}$ is *true* in I iff $A \notin I$.
(3) A ground clause $L_1 \vee \cdots \vee L_m$ is *true* in I iff at least one literal L_i is true in I.
(4) In general a clause C is *true* in I iff every ground instance $C\sigma$ of C is true in I. ($C\sigma$ is obtained by replacing every occurrence of a variable in C by a term in H. Different occurrences of the same variable are replaced by the same term.)
(5) A set of clauses **A** is *true* in I iff each clause in **A** is true in I.

A literal, clause, or set of clauses is *false* in I iff it is not true. If **A** is true in I, then we say that I is a *Herbrand model* of **A** and we write $\models_I \mathbf{A}$. It is a simple version of the Skolem-Löwenheim theorem that *a sentence* **A** *in clausal form has a model iff it has a Herbrand model.*

We can now formulate an explicit definition of the denotation determined by the model-theoretic semantics. Let **M(A)** be the set of all Herbrand models of **A**; then $\cap\mathbf{M}(\mathbf{A})$, the intersection of all Herbrand models of **A**, is itself a Herbrand interpretation

of **A**. If **A** contains the predicate symbol P, then the *denotation* $\mathbf{D}_2(P)$ is the relation associated with P by the Herbrand interpretation $\cap\mathbf{M}(\mathbf{A})$. In symbols,

$$\mathbf{D}_2(P) = \{(t_1, \ldots, t_n) : P(t_1, \ldots, t_n) \in \cap\mathbf{M}(\mathbf{A})\}$$

for any set of clauses **A**.

PROOF. $(t_1, \ldots, t_n) \in \mathbf{D}_2(P)$
iff $\mathbf{A} \models P(t_1, \ldots, t_n)$,
iff $\mathbf{A} \cup \{\bar{P}(t_1, \ldots, t_n)\}$ has no model,
iff $\mathbf{A} \cup \{\bar{P}(t_1, \ldots, t_n)\}$ has no Herbrand model,
iff $\bar{P}(t_1, \ldots, t_n)$ is false in all Herbrand models of **A**,
iff $P(t_1, \ldots, t_n)$ is true in all Herbrand models of **A**,
iff $P(t_1, \ldots, t_n) \in \cap\mathbf{M}(\mathbf{A})$.

Notice that the above equality holds for any set of clauses **A** even if **A** is inconsistent. If **A** is a consistent set of Horn clauses then $\cap\mathbf{M}(\mathbf{A})$ is itself a Herbrand model of **A**. More generally, Horn clauses have the *model intersection property*: If **L** is any nonempty set of Herbrand models of **A** then $\cap\mathbf{L}$ is also a model of **A**.

PROOF. Assume $\cap\mathbf{L}$ is not a model of **A**. Then $\cap\mathbf{L}$ falsifies some ground instance $C\sigma$ of a clause $C \in \mathbf{A}$.

If C is a procedure declaration, then

$$C\sigma = A \vee \bar{A}_1 \vee \cdots \vee \bar{A}_m,\ m \geq 0, \qquad A \notin \cap\mathbf{L}, \quad \text{and} \quad A_1, \ldots, A_m \in \cap\mathbf{L}.$$

Therefore for some $I \in \mathbf{L}$, $A \notin I$ and $A_1, \ldots, A_m \in I$. C is false in I, contrary to assumption that $I \in \mathbf{L}$.

If C is a goal statement, then

$$C\sigma = \bar{A}_1 \vee \cdots \vee \bar{A}_m,\ m > 0, \qquad A_1, \ldots, A_m \in \cap\mathbf{L}.$$

Therefore for all $I \in \mathbf{L}$, $A_1, \ldots, A_m \in I$. C is false in I, contrary to assumption that $I \in \mathbf{L}$.

$\{P(a) \vee P(b)\}$, where a and b are constants, is an example of a non-Horn sentence which does not have the model-intersection property: $\{\{P(a)\}, \{P(b)\}\}$ is a nonempty set of models, yet its intersection $\varnothing$ is a Herbrand interpretation which is not a model.

6. *Fixpoint Semantics*

In the fixpoint semantics, the denotation of a recursively defined procedure is defined to be the minimal fixpoint of a transformation associated with the procedure definition. Here we propose a similar definition of fixpoint semantics for predicate logic programs. In order to justify our definition we first describe the fixpoint semantics as it has been formulated for more conventionally defined recursive procedures. Our description follows the one given by de Bakker [6]

Let $P \Leftarrow \mathbf{B}(P)$ be a procedure declaration in an Algol-like language, where the first occurrence of P is the procedure name, where $\mathbf{B}(P)$ is the procedure body, and where the occurrence of P in $\mathbf{B}(P)$ distinguishes all calls to P in the body of the procedure. Associated with **B** is a transformation T which maps sets I of input-output tuples into other such sets $J = T(I)$. When the transformation T is monotonic (which means that $T(I_1) \subseteq T(I_2)$ whenever $I_1 \subseteq I_2$) the denotation of P is defined as

$$\cap\{I : T(I) \subseteq I\},$$

which is identical to the intersection of all fixpoints of T,

$$\cap\{I : T(I) = I\},$$

and which is itself a fixpoint (the least such) of T.

In a similar way a transformation T can be associated with a finite set of mutually recursive procedure declarations

$$P_1 \Leftarrow \mathbf{B}_1(P_1, \ldots, P_n)$$

$$\vdots$$

$$P_n \Leftarrow \mathbf{B}_n(P_1, \ldots, P_n).$$

The minimal fixpoint of T, which exists when T is monotonic, can be decomposed into components, the ith of which is the denotation of the procedure P_i.

By means of the procedural interpretation, the fixpoint semantics of predicate logic is defined similarly. A set of Horn clauses of the form $A \vee A_1 \vee \cdots \vee A_m$, where $m \geq 0$, is interpreted as a set of mutually recursive, possibly nondeterministic, procedure declarations We restrict the definition of the fixpoint semantics of predicate logic programs to sentences **A** which are sets of such procedure declarations. Associated with every such sentence **A** is a transformation T which maps Herbrand interpretations to Herbrand interpretations. Suppose that $P_1, \ldots, P_n$ are the predicate symbols occurring in **A**. The transformation T can be defined in terms of individual transformations T_i associated with the individual predicate symbols P_i. T_i maps Herbrand interpretations I to Herbrand interpretations $J_i = T_i(I)$ which contain only atomic formulas beginning with the predicate symbol P_i:

> $J_i = T_i(I)$ contains a ground atomic formula $A \in \hat{H}$ iff A begins with the predicate symbol P_i and, for some ground instance $C\sigma$ of a clause C in **A**, $C\sigma = A \vee \bar{A}_1 \vee \cdots \vee \bar{A}_m$ and $A_1, \ldots, A_m \in I$, $m \geq 0$.

The transformation T associated with **A** is defined by $T(I) = T_1(I) \cup \cdots \cup T_n(I)$.

The input-output relation associated by $J_i = T_i(I)$ with P_i can be regarded as the relation obtained by "substituting," for the procedure calls in the declarations of P_i in **A**, the appropriate input-output relations associated by I. This interpretation of T_i is analogous to the corresponding definition for conventionally defined recursive procedures. A simpler definition of T, which is less directly analogous to the conventional definition, is the following:

> $T(I)$ contains a ground atomic formula $A \in \hat{H}$ iff for some ground instance $C\sigma$ of a clause C in **A**, $C\sigma = A \vee \bar{A}_1 \vee \cdots \vee \bar{A}_m$ and $A_1, \ldots, A_m \in I$, $m \geq 0$.

Notice that, independently of I, $T(I)$ always contains all ground instances $A\sigma$ of unqualified assertions A in **A** (corresponding to the case $m = 0$ in the definition of $T(I)$).

Let $\mathbf{C}(\mathbf{A})$ be the set of all Herbrand interpretations closed under the transformation T, i.e. $I \in \mathbf{C}(\mathbf{A})$ iff $T(I) \subseteq I$. The *denotation* of a predicate symbol P occurring in a set of procedure declarations **A**, as determined by the *fixpoint semantics,* is

$$\mathbf{D}_3(P) = \{(t_1, \ldots, t_n) : P(t_1, \ldots, t_n) \in \cap\mathbf{C}(\mathbf{A})\}.$$

As a corollary of the theorem below, $\cap\mathbf{C}(\mathbf{A})$ is itself closed under T and therefore $\mathbf{D}_3(P)$ is the smallest set of input-output tuples closed under T. In conventional fixpoint theory this fact is proved by using the monotonicity of T.

7. *Model-Theoretic and Fixpoint Semantics*

We shall show that for sets of procedure declarations **A**, model-theoretic and fixpoint semantics coincide: $\mathbf{D}_2 = \mathbf{D}_3$. It would be sufficient to show that $\cap\mathbf{M}(\mathbf{A}) = \cap\mathbf{C}(\mathbf{A})$, but it is easy to prove that even $\mathbf{M}(\mathbf{A}) = \mathbf{C}(\mathbf{A})$.

In other words, a Herbrand interpretation I of **A** is a model of **A** iff I is closed under the transformation T associated with **A**.

THEOREM. *If* **A** *is a set of procedure declarations, then* $\mathbf{M}(\mathbf{A}) = \mathbf{C}(\mathbf{A})$, i.e. $\models_I \mathbf{A}$ *iff* $T(I) \subseteq I$, *for all Herbrand interpretations* I *of* **A**

PROOF. ($\models_I \mathbf{A}$ implies $T(I) \subseteq I$.) Suppose that I is a model of **A**. We want to show $J = T(I) \subseteq I$, i.e. that if $A \in J$ then $A \in I$.

Assume that $A \in J$; then by the definition of T, for some $C \in \mathbf{A}$ and for some ground instance $C\sigma$ of C,

$$C\sigma = A \vee \bar{A}_1 \vee \cdots \vee \bar{A}_n \text{ and } A_1, \ldots, A_n \in I.$$

Because I is a model of $\mathbf{A}$, $C\sigma$ is true in I. But then A is true in I, because $\bar{A}_1, \ldots,$ and $\bar{A}_n$ are false in I. Therefore $A \in I$.

($T(I) \subseteq I$ implies $\models_I \mathbf{A}$). Suppose that I is not a model of $\mathbf{A}$. We want to show that $T(I) \not\subseteq \mathbf{I}$. But I falsifies some ground instance $C\sigma$ of a clause C in $\mathbf{A}$, where $C\sigma = A \vee \bar{A}_1 \vee \cdots \vee \bar{A}_m$, $m \geq 0$. Because I falsifies $C\sigma$, $A \notin I$ and $A_1, \ldots, A_m \in I$. But then, because $A_1, \ldots, A_m \in I$, it follows that $A \in T(I)$. Therefore $T(I) \not\subseteq \mathbf{I}$.

COROLLARY. *If* $\mathbf{A}$ *is a set of procedure declarations, then* $\cap\mathbf{C}(\mathbf{A})$ *is closed under* T.

PROOF. $\cap\mathbf{C}(\mathbf{A}) = \cap\mathbf{M}(\mathbf{A})$ by the model-intersection property is a model of $\mathbf{A}$ and by the theorem is therefore closed under T.

8. *Operational and Fixpoint Semantics, Hyperresolution*

The equivalence $\mathbf{D}_1 = \mathbf{D}_3$ between operational and fixpoint semantics, which follows from the equivalences $\mathbf{D}_1 = \mathbf{D}_2$ and $\mathbf{D}_2 = \mathbf{D}_3$, has different interpretations depending upon the inference system which determines $\mathbf{D}_1$. Here we investigate the interpretation associated with a particular inference system based upon hyperresolution [21].

For ground procedure declarations the definition of hyperresolution is very simple:

> An atomic formula A is the *hyperresolvent* of ground clauses $A \vee \bar{A}1 \vee \cdots \vee \bar{A}_m$ and $A_1, \ldots, A_m$. A is said to be obtained from $A \vee \bar{A}_1 \vee \cdots \vee \bar{A}_m$ and $A_1, \ldots, A_m$ by *hyperresolution*.

The connection with fixpoint semantics is obvious: If T is the transformation associated with the set of procedure declarations $\mathbf{A}$ and if I is a Herbrand interpretation of $\mathbf{A}$, then $T(I)$ is the set of all ground instances of assertions in $\mathbf{A}$ together with all hyperresolvents derivable in one step from ground instances of clauses in $\mathbf{A}$ and from assertions in I. It follows that

> A is derivable by means of a hyperresolution derivation from ground instances of clauses in $\mathbf{A}$ iff $A \in \bigcup_{m=0}^{\infty} T^m(\varnothing)$ where $T^0(\varnothing) = \varnothing$ and $T^{m+1}(\varnothing) = T(T^m(\varnothing))$.

Let $\mathbf{D}_1^{\mathbf{H}}$ be the operational semantics associated with the two inferences rules of ground instantiation of clauses in $\mathbf{A}$ and ground hyperresolution, i.e define

$$(t_1, \ldots, t_n) \in \mathbf{D}_1^{\mathbf{H}}(P) \text{ iff } P(t_1, \ldots, t_n) \in \bigcup_{m=0}^{\infty} T^m(\varnothing).$$

The equivalence of $\mathbf{D}_1^{\mathbf{H}}$ and the model-theoretic semantics $\mathbf{D}_2$ is the completeness, for Horn clauses, of the inference system whose inference rules are ground instantiation of input clauses and ground hyperresolution. Completeness can be proved using standard resolution-theoretic arguments. Here we present an alternative direct proof that for any set of declarations $\mathbf{A}$ with associated transformation T, $\bigcup_{m=0}^{\infty} T^m(\varnothing) = \cap\mathbf{M}(\mathbf{A})$.

PROOF. Let $\mathbf{U}$ abbreviate $\bigcup_{m=0}^{\infty} T^m(\varnothing)$.

($\mathbf{U} \subseteq \cap\mathbf{M}(\mathbf{A})$). Suppose that $A \in \mathbf{U}$. Then A is derivable by means of a hyperresolution derivation from ground instances of clauses in $\mathbf{A}$. By the correctness of hyperresolution and instantiation, $\mathbf{A} \models A$ and therefore $A \in \cap\mathbf{M}(\mathbf{A})$.

($\cap\mathbf{M}(\mathbf{A}) \subseteq \mathbf{U}$). We show that $\mathbf{U}$ is closed under T, because then $\mathbf{U} \in \mathbf{M}(\mathbf{A})$, and therefore $\cap\mathbf{M}(\mathbf{A}) \subseteq \mathbf{U}$. Suppose that $A \in T(\mathbf{U})$. By the definition of T, either A is an instance of an unqualified assertion in $\mathbf{A}$ or some clause $A \vee \bar{A}_1 \vee \cdots \vee \bar{A}_n$ is an instance of a clause in $\mathbf{A}$ and $A1, \ldots, A_n \in \mathbf{U}$. In the first case $A \in \mathbf{U}$, because $A \in T^m(\varnothing)$, $m > 0$. In the second case $A_1, \ldots, A_n \in T^N(\varnothing)$ for some $N \geq 0$, and therefore $A \in T^{N+1}(\varnothing)$ and $A \in \mathbf{U}$. Therefore $\mathbf{U}$ is closed under T.

Therefore for sets of declarations, $\mathbf{D}_1^{\mathbf{H}} = \mathbf{D}_2$.

Because of the equivalence between model-theoretic and fixpoint semantics, we also have that $\mathbf{D}_1^{\mathbf{H}} = \mathbf{D}_3$, i.e. $\bigcup_{m=0}^{\infty} T^m(\varnothing) = \cap\{I : T(I) \subseteq I\}$.

This last fact is usually proved in the fixpoint theory by demonstrating the continuity of the transformation T.

9. *Conclusion*

For arbitrary sentences X and Y of first-order predicate logic, proof theory determines when $X \vdash Y$ and model theory determines when $X \models Y$. We have argued that in the procedural interpretation, operational semantics is proof theory and fixpoint semantics is model theory. On the other hand, operational and fixpoint semantics only deal with the case where Y is a set of ground atomic formulas. Moreover, fixpoint semantics only deals with X, a set of procedure declarations. We believe that the added generality of proof theory and model theory has useful consequences.

The completeness theorem of first-order logic states that the relations $\vdash$ of derivability and $\models$ of logical implication are equivalent. For goal oriented inference systems this equivalence establishes that various computation rules compute the relation determined by the fixpoint semantics. More generally, this equivalence can be used to justify various rules (such as Scott's induction rule [6]) for proving properties of programs.

We have argued that various notions of the conventional theory of computing can be understood in terms of the classical theory of predicate logic. We believe moreover that the predicate logic theory has further contributions to make both to the theory and to the practice of computing.

ACKNOWLEDGMENTS. We are indebted to Michael Gordon for his interest and useful criticism of work leading to this paper. Thanks are due also to Keith Clark, Alain Colmerauer, Gerard Huet, David Park, and Willem-Paul de Roever for their helpful comments on earlier versions of the paper. Suggestions from the referees have also been incorporated in the paper.

REFERENCES

1 BEKIĆ, H Definable operations in general algebra, and the theory of automata and flow charts IBM Res Rep , Vienna, 1971
2 BLEDSOE, W W Splitting and reduction heuristics in automatic theorem proving *Artif Intel 2* (1971), 55-77
3 BURSTALL, R M Formal description of program structure and semantics in first order logic. In *Machine Intelligence 5*, B Meltzer and D Michie, Eds , Edinburgh U Press, Edinburgh, 1969, pp 79-98
4 CHURCH, A *Introduction to Mathematical Logic, Vol 1* Princeton U Press, Princeton, N J , 1956
5 COLMERAUER, A , KANOUI, H , PASERO, R., AND ROUSSEL, P. Un système de communication homme-machine en français. Groupe d'Intelligence Artificielle, U E R de Luminy, Université d'Aix-Marseille, Luminy, 1972
6 DE BAKKER, J W Recursive procedures Tract No 24, Mathematical Centre, Amsterdam, 1971
7 DE BAKKER, J W , AND DE ROEVER, W P A calculus of recursive program schemes. In *Automata, Languages and Programming*, M Nivat, Ed , North-Holland Pub Co , Amsterdam, 1973, pp 167-196
8 ERNST, G W The utility of independent subgoals in theorem proving *Inform Contr 18*, 3 (April 1971), 237-252
9 KANOUI, H Application de la démonstration automatique aux manipulations algebriques et à l'intégration formelle sur ordinateur Groupe d'Intelligence Artificielle, U E R de Luminy, Université d'Aix-Marseille, Luminy, 1973.
10 KOWALSKI, R Predicate logic as programming language Proc IFIP Cong 1974, North-Holland Pub Co , Amsterdam, 1974, pp 569-574
11 KOWALSKI, R Logic for problem-solving DCL Memo 75, Dep Artificial Intelligence, U. of Edinburgh, Edinburgh, 1974
12 KOWALSKI, R , AND KUEHNER, D Linear resolution with selection function *Artif Intel 2* (1971), 227-260
13. LOVELAND, D.W A simplified format for the model elimination theorem-proving procedure. *J. ACM 16*, 3 (July 1969), 349-363
14 MANNA, Z Properties of programs and the first-order predicate calculus *J ACM 16*, 2 (April 1969), 244-255
15 MILNER, R Implementation and applications of Scott's logic for computable functions. Proc. ACM Conf on Proving Assertions About Programs, Jan 1972, pp 1-6

16. NILSSON, N J *Problem Solving Methods in Artificial Intelligence.* McGraw-Hill, New York, 1971
17 PARK, D Fixpoint induction and proofs of program properties In *Machine Intelligence 5,* B Meltzer and D Michie, Eds , Edinburgh U Press, Edinburgh, 1969, pp 59–78
18 PASÉRO, R. Représentation du français en logique du premier ordre en vue de dialoguer avec un ordinateur Group d'Intelligence Artificielle, U.E R de Luminy, Université d'Aix-Marseille, Luminy, 1973.
19 REITER, R Two results on ordering for resolution with merging and linear format *J ACM 18,* 4 (Oct 1971), 630–646
20. ROBINSON, J A A machine-oriented logic based on the resolution principle. *J ACM 12*, 1 (Jan. 1965), 23–41.
21 ROBINSON, J A Automatic deduction with hyper-resolution *Int J Comptr Math 1* (1965), 227–234
22 SCOTT, D Outline of a mathematical theory of computation Tech Monog PRG-2, Comptg Lab , Oxford U , Oxford, England
23 SLAGLE, J R , AND KONIVER, P Finding resolution graphs and using duplicate goals in AND/OR trees *Inform Sci 3* (1971), 315–342
24 WARREN, D H D WARPLAN A system for generating plans DCL Memo 76, Dep. of Artificial Intelligence, U of Edinburgh, Edinburgh, 1974

RECEIVED MARCH 1974, REVISED APRIL 1976