RAIRO
INFORMATIQUE THÉORIQUE

G. LÉVI
A. M. PEGNA

Top-down mathematical semantics and symbolic execution

RAIRO – Informatique théorique, tome 17, nº 1 (1983), p. 55-70.
<http://www.numdam.org/item?id=ITA_1983__17_1_55_0>

R.A.I.R.O. Informatique théorique/Theoretical Informatics
(vol. 17, n° 1, 1983, p. 55 à 70)

TOP-DOWN MATHEMATICAL SEMANTICS AND SYMBOLIC EXECUTION (*)

by G. LEVI and A. M. PEGNA [1]

Communicated by J.-C. PERROT

Abstract. — *We introduce a formal semantics which is equivalent to fixed-point semantics and which is very close to symbolic execution. Such a semantics can be considered the formal basis of several existing program analysis systems.*
The paper is based on a simple equational language and contains suggestions for possible generalizations to conventional programming languages.

Résumé. — *On introduit ici une sémantique formelle qui est équivalente à la sémantique du point fixe, et qui est très proche de l'exécution symbolique. Une telle sémantique peut être considérée la base formelle de nombreux systèmes pour l'analyse de programmes.*
L'article se base sur un simple langage équationnel et contient des suggestions pour de possibles généralisations à des langages de programmation conventionnelles.

1. SYMBOLIC EXECUTION, SEMANTICS AND PROGRAM ANALYSIS

Symbolic execution is a currently popular technique, which plays a major role in several program analysis systems. We will mention program verification systems [8, 13, 17], systems for proving theorems in recursive function theory [4, 5, 12], systems for program transformation and optimization [7], " sophisticated " testing and debugging systems [2, 3, 10, 11]. Although all of the above systems are concerned with program semantics, no formalization of language semantics is required.

The operational semantics defined by the symbolic interpreter seems to be " all you need " for program analysis. This is not true for the semantics defined by a conventional " numeric " interpreter. Such a difference could informally be explained as follows. A " numeric " interpreter can only give a meaning (output value) to a pair $\langle$ program, input values $\rangle$, while the ability to handle symbolic input values allows the symbolic interpreter to give a

(*) Received in June 1979, revised in April 1982.
([1]) Istituto di Scienze dell'Informazione, Università di Pisa, Corso Italia 40, I-56100 Pisa.

meaning to a program as a mapping from input values to output values. Hence, the denotation given by a symbolic operational semantics is similar to the one which could be obtained by formal methods (mathematical or denotational semantics).

In the paper we are concerned with the relationship between formal semantics and symbolic operational semantics, and more specifically, with a mathematical semantics, which is a close relative of the symbolic operational semantics. All our definitions and theorems are given for a simple programming language (TEL), that will be introduced in the next section. We will finally sketch a possible generalization of our results to conventional programming languages.

2. THE PROGRAMMING LANGUAGE TEL

TEL (Term Equations Language) is a simple applicative calculus, originally developed [1, 12] as a specification language to be used in an interactive system for proving properties of programs. Languages very close to TEL have been independently proposed by Burstall [6] and Goguen [9]. TEL has rather easy-to-define mathematical semantics and symbolic operational semantics, since the abstract TEL machine has no built-in data types, no operations with side effects (i. e. assignement) and control constructs are function composition and recursion only. Moreover, the language has a straightforward interpretation as a first order theory, which allows to define a model-theoretic tarskian semantics.

The language is based on the concept of *term*, which is defined, according to the syntax of first order logic, from constant symbols, variable symbols, n-adic data constructor symbols and n-adic function symbols. Formally, a term is either a constant symbol, or a variable symbol, or the application of an n-adic data constructor (or function) symbol to n terms.

Formulae in the calculus are term *equations* of the following form

$$f(t_1, \ldots, t_n)=t$$

where

i) f is an n-adic function symbol

ii) $t_1, \ldots, t_n$ are terms which do not contain any function symbol,

iii) t is a term which can only contain variable symbols occurring in some of the t_i's.

A TEL *procedure* of name f is a set of equations

$$\{ f(t_{11}, \ldots, t_{1m}) = t^1, \ldots, f(t_{n1}, \ldots, t_{nm}) = t^n \}$$

such that the left terms $f(t_{i1}, \ldots, t_{im})$ are pairwise *non unifiable*. That is, for each pair of left terms 1_i and 1_j there exists no instantiation of variable symbols to terms which makes 1_i and 1_j identical (such a constraint is a sufficient condition for the Church-Rosser property to hold).

An example of TEL procedure is the following definition of " append "

$$\{ \text{append}(\text{nil}, x) = x,\ \text{append}(\text{cons}(x, y), z) = \text{cons}(x, \text{append}(y, z)) \}$$

where nil is a constant symbol and cons is a diadic data constructor symbol.

TEL has a typing mechanism which gives each term a sort, by means of syntactic specifications, which in our example, could have the following form:

$$\begin{aligned} \text{nil}: &\Rightarrow \text{binary-tree} \\ \text{cons}: \text{binary-tree } X \text{ binary-tree} &\Rightarrow \text{binary-tree} \\ \text{append}: \text{binary-tree } X \text{ binary-tree} &\Rightarrow \text{binary-tree} \end{aligned}$$

Taking into account sorts would make our definitions of terms, equations and procedures more complex. For the sake of simplicity, in the sequel we will be concerned with a single sort. The extension of our definitions and propositions to the multiple sorts case is straightforward.

TEL equations are essentially definitions of recursive functions by disjoint cases and are very similar to algebraic data type specifications (*see*, for instance [9]). They can also be considered recursive program schemes, since no interpretation is given to the syntactic domains. Procedures are defined by cases on the structure of " abstract " data, which are trees (terms) built from constant and data constructor symbols. For example, " append " is defined by two equations. The equations are concerned with the cases in which the first parameter is the binary tree " nil " or a binary tree obtained by a " cons " operation.

The programming style in TEL is very close to the pure LISP programming style. The main differences are the following:

i) TEL has no built-in conditional, hence cases must be explicitely defined.

ii) TEL has no built-in data types. Any recursive data type can be defined by suitable constant and data constructor symbols.

iii) TEL is a first-order language, hence functional arguments are not allowed.

The interpreter of TEL is based on a call by name evaluation rule. Therefore, it is possible to define non-strict functions, including conditionals. See, for example, the following definition of if-then-else.

$$\{\text{if-then-else (true, } x, y) = x, \text{ if-then-else (false, } x, y) = y\}$$

The interpreter " evaluates " a term by applying equations as term rewriting rules. A subterm is rewritable if it is unifiable with an equation left term. The most general unifier binds the equation formal arguments (variable symbols occurring in the left term) and allows the instantiation of the right term (which does not contain any free variable). The evaluation of a term is the replacement of its outermost rewritable subterm with the instantiated right part of the equation. Note that, since equation left terms are pairwise non unifiable, at most one equation can be applied to rewrite a given subterm. Assume, for example, we have the following equations

1. append (nil, x) = x
2. append (cons (x, y), z) = cons (x, append (y, z))
3. reverse (nil) = nil
4. reverse (cons (x, y)) = append (reverse (y), cons (x, nil))

The evaluation of term reverse (append (cons (a, nil), cons (b, nil))) where a and b are constant symbols, is the following sequence of rewritings.

reverse (cons (a, append (nil, cons (b, nil)))),	by eq. 2
append (reverse (append (nil, cons (b, nil))), cons (a, nil))	by eq. 4
append (reverse (cons (b, nil)), cons (a, nil))	by eq. 1
append (append (reverse (nil), cons (b, nil)), cons (a, nil))	by eq. 4
append (append (nil, cons (b, nil)), cons (a, nil))	by eq. 3
append (cons (b, nil), cons (a, nil))	by eq. 1
cons (b, append (nil, cons (a, nil)))	by eq. 2
cons (b, cons (a, nil))	by eq. 1

In the next section we will consider sets of TEL equations as first-order theories. This will allow us to define a model-theoretic semantics. We will then introduce a more precise definition of the interpreter, which will be the basis of a formal operational semantics.

3. MODEL-THEORETIC SEMANTICS

A TEL equation can be considered the concrete syntactic representation of a well-formed-formula of a first-order theory, according to the following definitions.

A *data term* is

i) a variable symbol, or

ii) a constant symbol, or

iii) a n-adic data constructor symbol d^n applied to n data terms.

A *functional term* is the application of an n-adic function symbol f^n to n data terms.

An *atomic formula* is an equality

$1=r$, where 1 is a functional term and r is a data term.

A *well-formed-formula* is a clause

f *if* G, where
f is an atomic formula and
G is a (possibly empty) set of atomic formulas (*if-set*).

A *well-formed-formula*

f *if* $(g_1, \ldots, g_n)$ must be read as the formula
$(g_1 \wedge \ldots \wedge g_n) \supset f$, where all the variable symbols are universally quantified.

It is rather easy to show that any TEL equation can be expressed as a well-formed-formula, by making explicit the relationship between inputs and outputs, which are implicit in the function composition construct.

Given an equation

$$f(t_1, \ldots, t_n)=t$$

the following algorithm, when applied to the right term t, transforms the equation into a well formed formula (*wff*).

EQUATION-TO-WFF-ALGORITHM: For each functional subterm t_i.

Step 1: t_i is replaced (inside-out) by a "new" variable symbol v_i.

Step 2: The atomic formula $t_i = v_i$ is inserted in the *if*-set.

For example, the set of equations

$$\{\text{append}(\text{cons}(x, y), z) = \text{cons}(x, \text{append}(y, z)),$$
$$\text{reverse}(\text{cons}(x, y)) = \text{append}(\text{reverse}(y), \text{cons}(x, \text{nil}))\}$$

is transformed to

$$\{\text{append}(\text{cons}(x, y), z) = \text{cons}(x, w) \textit{ if } (\text{append}(y, z) = w),$$
$$\text{reverse}(\text{cons}(x, y)) = z \textit{ if } (\text{reverse}(y) = w, \text{append}(w, \text{cons}(x, \text{nil})) = z)\}$$

A set E of TEL equations can thus be considered the set of axioms of a first-order-theory T_E. The semantics of E can then be defined as an interpretation which satisfies all the equations of E, i. e. a *model* of T_E. We will only be concerned with *free* (Herbrand) *interpretations*, over the abstract *data domain* D_E (Herbrand Universe, free magma, word algebra, etc.), defined as follows.

i) D_E contains all the constant symbols occurring in some equation of E and a distinct constant symbol ω (undefined).

ii) for each n-adic data constructor symbol d^n occurring in an equation of E, D_E contains all the terms $d^n(t_1, \ldots, t_n)$, such that $t_1, \ldots, t_n$ belong to D_E.

It is worth noting that D_E is exactly the set of TEL abstract data values, which may contain instances of the undefined symbol, since we are interested in a call by-name semantics.

A free interpretation is any set of *ground atomic formulas*, i. e. any subset of the *interpretation base* I_E (Herbrand base), which contains, for each n-adic function symbol f^n, all the atomic formulas $f^n(t_1, \ldots, t_n)=t$, such that $t_1, \ldots, t_n$, t belong to D_E, and t does not contain the undefined constant symbol ω. The following theorem holds for theories defined by TEL equations as well as for theories defined by Horn clauses [16, 18].

THEOREM 1: The intersection of all the free models of a theory is also a model (minimal model of the theory).

Proof: Let M_1, M_2 be free models of the theory T_E and let

$$e: f(t_1, \ldots, t_n)=t \quad \textit{if} \quad (g_1, \ldots, g_m)$$

be the well-formed-formula corresponding to an equation of E.

Since the *wff* e must be true in M_1, for each instantiation λ of variable symbols of e to terms of D_E

either $[f(t_1, \ldots, t_n)=t]_\lambda \in M_1$
or there exists a formula g_j, $1 \leq j \leq m$, such that $[g_j]_\lambda \notin M_1$.

In the last case, $[g_j]_\lambda \notin M_1 \cap M_2$. Hence, the corresponding equation instantiations are true also in $M_1 \cap M_2$.

This is the case for model M_2 also. Therefore we can restrict our analysis only to that set of instantiations such that for each λ

$$[f(t_1, \ldots, t_n)=t]_\lambda \in M_1 \quad \text{and}$$
$$[f(t_1, \ldots, t_n)=t]_\lambda \in M_2.$$

This implies $[f(t_1, \ldots, t_n)=t]_\lambda \in M_1 \cap M_2$. Hence, e is true in $M_1 \cap M_2$. Theorem 1 allows us to give the following definition.

DEFINITION 1: Given a set of equations E, the denotation $mt(E)$ defined by the *model-theoretic semantics* is the subset of I_E which is the minimal model of the theory T_E.

4. OPERATIONAL SEMANTICS

We will now define the inference rule f_E of the TEL interpreter.

f_E is a transformation which maps equations into equations. Let $g_i : g_l = g_r$ be a variable free equation, such that g_r contains at least one functional subterm, say g_{r_k}, and there exists in E an equation $e : t_l = t_r$ such that $[t_l]_\lambda = [g_{r_k}]_\lambda$ where λ is the most general unifier of t_l and g_{r_k}. The equation g_{i+1} which is obtained by applying the transformation f_E to g_i is the following:

$$g_{i+1} : [g_l]_\lambda = [[g_r]^{t_r}_{g_{r_k}}]_\lambda.$$

In other words, g_{i+1} is obtained from g_i by applying λ to the left-term, and to the term resulting from replacement of g_{r_k} by t_r in the right-term.

It is worth noting that, for a given (variable free) subterm g_{r_k} there is at most one equation whose left-term is unifiable with g_{r_k} (remember that the left-terms of the equations are pairwise non-unifiable). Hence, the only source of nondeterminism is the choice of the functional subterm to be rewritten. Such a nondeterminism is solved by letting f_E choose the leftmost outermost rewritable subterm.

The choice of the leftmost outermost rewritable subterm ('call-by-name' rule) corresponds to rewriting a subterm consisting in the application of a function symbol for which not all the arguments are needed to determine a 'value'.

The transformation f_E can iteratively be applied to evaluate a term t as follows:

Step 1: Start with the equation $g_0 : t = t$.

Step 2: If the right-term r_i of equation g_i has no rewritable subterms, then if r_i belongs to D_E stop with success (g_i is the result), otherwise stop with $g_i : l_i = \omega$, which does not belong to I_E.

Step 3: $g_{i+1} = f_E(g_i)$. Go to step 2.

Consider, for example, the evaluation of term reverse (cons (nil, nil)), with the set of equations

$$\begin{aligned} \{ \text{append}(\text{nil}, x) &= x \\ \text{append}(\text{cons}(x, y), z) &= \text{cons}(x, \text{append}(y, z)) \\ \text{reverse}(\text{nil}) &= \text{nil} \\ \text{reverse}(\text{cons}(x, y)) &= \text{append}(\text{reverse}(y), \text{cons}(x, \text{nil})) \} \end{aligned}$$

$$\begin{aligned} g_0&: \text{reverse}(\text{cons}(\text{nil}, \text{nil})) = \text{reverse}(\text{cons}(\text{nil}, \text{nil})) \\ g_1&: \text{reverse}(\text{cons}(\text{nil}, \text{nil})) = \text{append}(\text{reverse}(\text{nil}), \text{cons}(\text{nil}, \text{nil})) \\ g_2&: \text{reverse}(\text{cons}(\text{nil}, \text{nil})) = \text{append}(\text{nil}, \text{cons}(\text{nil}, \text{nil})) \\ g_3&: \text{reverse}(\text{cons}(\text{nil}, \text{nil})) = \text{cons}(\text{nil}, \text{nil}). \end{aligned}$$

DEFINITION 2: Given a set of equations E, the denotation $tdo(E)$ defined by the *top-down operational semantics* is the following:

$$tdo(E) = \{ f^n(t_1, \ldots, t_n) = t \in I_E \mid f^n(t_1, \ldots, t_n) = t$$

can be derived from $f^n(t_1, \ldots, t_n) = f^n(t_1, \ldots, t_n)$ by the transformation f_E }.

THEOREM 2: $tdo(E) = mt(E)$.

Proof: Transformation f_E is a top-down proof finding inference rule which can easily be shown equivalent to an extension of the resolution principle concerned with the call-by-name behaviour. Hence $tdo(E)$ is the set of all the ground atomic formulas which are theorems of T_E. On the other hand, $mt(E)$ is the set of all the ground atomic formulas which are true under all the interpretations. The theorem is then a straightforward consequence of the completeness theorem for first order theories.

Even if the operational semantics $tdo(E)$ is equivalent to $mt(E)$, the corresponding inference rule f_E is not adequate for reasoning about programs. In fact, it can only give a meaning to (i. e. evaluate) an application of a procedure to specific input terms. The transformation we need for program analysis, must be able to give a meaning to a program (in our case, to a *TEL* procedure) as a function from input to output values. This will be the case of the top-down mathematical semantics which will be introduced in the next section.

5. TOP-DOWN MATHEMATICAL SEMANTICS

TEL symbolic execution uses an inference rule which is essentially the same rule (f_E) used for standard evaluation. Differences arise because *sym-*

bolic constants (skolem constants) may appear in the term to be evaluated. A symbolic constant is a constant symbol which stands for any element of the domain D_E. Symbolic constants can be handled as variable symbols. When unification of a term containing symbolic constants with the left-term of an equation of E is attempted, symbolic constants can be instantiated so as to make unification successful. More precisely, symbolic constants can be bound either to terms belonging to D_E, or to terms containing newly created symbolic constants.

For example, if term reverse(x), where x denotes a symbolic constant, is unified with left-terms reverse(nil) and reverse (cons (x, y)), x will be bound to nil and cons $(x1, x2)$ respectively.

The unification behaviour of symbolic constants makes the transformation nondeterministic. A given term containing symbolic constants can generally be unified with more than one equation left-term. Hence the application of the transformation to a single equation may generate more than one equation.

We will then define a new transformation s_E (inference rule of the symbolic interpreter) which maps sets of equations onto sets of equations. Consider a set of equations G_i, such that all the equations of G_i satisfy the following constraints:

i) each equation left-term is a functional term.

ii) there are no equations whose left terms are unifiable. (Note that such constraints are exactly the constraints given in section 1 for procedure defining sets of equations, if constant symbols are handled as variable symbols).

The set $G_{i+1} = s_E(G_i)$ is obtained as follows:

i) each equation of G_i that cannot be rewritten by g_E is in G_{i+1}.

ii) all the equations which are obtained by applying f_E to an equation in G_i are in G_{i+1} (the application of f_E to a single equation in G_i may cause more than one equation to be inserted in G_{i+1}).

A partial example of symbolic execution for the term reverse (cons (x, y)), with the set of equations defined in section 2, is given below.

$G_0 = \text{reverse}(\text{cons}(x, y)) = \text{reverse}(\text{cons}(x, y))$

$G_1 = \text{reverse}(\text{cons}(x, y)) = \text{append}(\text{reverse}(y), \text{cons}(x, \text{nil}))$

$G_2 = \text{reverse}(\text{cons}(x, \text{nil})) = \text{append}(\text{nil}, \text{cons}(x, \text{nil}))$,
$\quad \text{reverse}(\text{cons}(x, \text{cons}(y1, y2) =$
$\qquad \text{append}(\text{append}(\text{reverse}(y2), \text{cons}(y1, \text{nil})), \text{cons}(x, \text{nil}))$

$$G_3 = \text{reverse}(\text{cons}(x, \text{nil})) = \text{cons}(x, \text{nil}),$$
$$\text{reverse}(\text{cons}(x, \text{cons}(y1, \text{nil}))) = \text{append}(\text{append}(\text{nil}, \text{cons}(y1, \text{nil})), \text{cons}(x, \text{nil}),$$
$$\text{reverse}(\text{cons}(x, \text{cons}(y1, \text{cons}(y21, y22)))) = \text{append}(\text{append}(\text{append}(\text{reverse}(y22), \text{cons}(y21, \text{nil})), \text{cons}(y1, \text{nil})), \text{cons}(x, \text{nil}))$$

Symbolic execution is generally non terminating and provides an enumeration of all the possible computation paths of a given procedure. Roughly speaking, it gives a meaning to the procedure, as opposed to standard evaluation which gives a meaning to a specific procedure application.

DEFINITION 3: Given a set of equations E, the denotation $tdso(E)$ defined by the *top-down symbolic operational semantics* of the procedure of name f^n is the set

$tdso(E, f^n) = \{f^n(t_1, \ldots, t_n) = t \in I_E$ such that $f^n(t_1, \ldots, t_n) = t$ is a (possibly instantiated) atomic formula derived from $\{f^n(x_1, \ldots, x_n) = f^n(x_1, \ldots, x_n)\}$ by the inference rule s_E, where $x_1, \ldots, x_n$ are symbolic constants $\}$.

A mathematical semantics based on the inference rule s_E can be defined, following Nivat's construction of language semantics [15]. A set of equations E may be seen as a recursive program scheme [15], i. e. a rewriting system E:

$$E = \{ f_1(t_1^1, \ldots, t_{n_1}^1) = t^1$$

$$f_m(t_1^m, \ldots, t_{n_m}^m) = t^m \}$$

on a free magma $M(F, V, C)$, where V is a set of symbolic constant symbols, F is the set of data constructor symbols and C is the set of constant symbols.

Let $t_1, \ldots, t_{n_i}$ be symbolic constant symbols and let $\underset{E}{\rightarrow}$ denote the inference rule s_E (reduction). $\overset{*}{\underset{E}{\rightarrow}}$ is the reflexive and transitive closure of $\underset{E}{\rightarrow}$. The semantics of E is the language [15]

$$L^* = \{ L(E, f_1), \ldots, L(E, f_m) \}$$

where

$$L(E, f_i) = \{ f_i(t_1^i, \ldots, t_{n_i}^i) = t^i \mid t_1^i, \ldots, t_{n_i}^i, t^i \in M(F, V, C)$$

and

$$f_i(t_1, \ldots, t_{n_i}) = f_i(t_1, \ldots, t_{n_i}) \overset{*}{\underset{E}{\rightarrow}} f_i(t_1^i, \ldots, t_{n_i}^i) = t^i \}$$

DEFINITION 4: Given a set of equations E, the *top-down mathematical semantics* of a procedure of name f_i is the set

$$tdm(E, f_i) = \{ [e]_\lambda \in I_E \mid e \in L(E, f_i) \},$$

which is obtained from $L(E, f_i)$ by instantiating all the symbolic constants to terms belonging to D_E.

It is worth noting that the top-down symbolic transformation (whose inference rules are reduction and instantiation) gives a meaning to procedure definitions rather than procedure applications. The semantics defined by such a transformation is therefore a denotational semantics, as well as the fixed-point semantics we will describe in the next-section.

6. FIXED-POINT SEMANTICS

Our definition of fixed-point semantics is very similar to Horn clauses mathematical semantics [18] and can more easily be defined if equations are transformed to well-formed-formulas, according to the definition given in section 3.

Let I be an interpretation, i. e. a subset of the interpretation base I_E for a given set of equations E and let

$$e_i : f(t_1, \ldots, t_n) = t \quad \textit{if} \quad (h_1 = v_1, \ldots, h_m = v_m)$$

be the well-formed-formula corresponding to an equation of E. The *wff* e_i defines a transformation F_E^i which maps I onto the interpretation

$$I_i = F_E^i(I) \quad \text{such that}$$

i) all the atomic formulas in I are in I_i.

ii) for each instantiation λ of variables to terms, such that, for each $1 \leqslant j \leqslant m$ either $[h_j = v_j]_\lambda$ is in I or $[v_j]_\lambda$ contains ω and t does not contain ω, the atomic formula $[f(t_1, \ldots, t_n) = t]_\lambda$ is in I_i.

It is worth noting that if the *if*-set is empty condition *ii*) is always satisfied. Condition *ii*) simply asserts that if for some instantiation λ, all the atomic formulas in the *if*-set are true in I (i. e. they belong to I) then the atomic formula $[f(t_1, \ldots, t_n) = t]_\lambda$ is also true. Because of our definition of interpretations, if $[t]_\lambda$ contains the undefined constant symbol ω, the atomic formula cannot belong to an interpretation. For this same reason, one possibility for an atomic formula of the *if*-set to be true is that its dataterm $[v_j]_\lambda$ contains ω.

Of course, our treatment of ω corresponds to a call-by-name semantics, i. e. a new atomic formula can be computed (provided its right term is not undefined) even if some of its subterms are undefined.

Consider the following example, concerned with the equation

$$\mathrm{f}(x, y)=\text{if-then-else}\,(\mathrm{gt}(x, y), -(x, y), -(y, x)),$$

whose corresponding well-formed-formula is

$$\mathrm{f}(x, y)=z \quad \textit{if} \quad (\mathrm{gt}(x, y)=w_1, \quad -(x, y)=w_2, \quad -(y, x)=w_3, \quad \text{if-then-else} \quad (w_1, w_2, w_3)=z)$$

and let the interpretation I be the following

$$I=\{\,\mathrm{gt}(1, 2)=\text{false}, \quad -(2, 1)=1, \quad \text{if-then-else} \quad (\text{false}, \omega, 1)=1\,\}$$

An instantiation λ satisfying condition *ii*) is

$$\lambda=\{\,(x, 1), (y, 2), (z, 1)(w_1, \text{false}), (w_2, \omega), (w_3, 1)\,\},$$

which allows to derive the new atomic formula $\mathrm{f}(1, 2)=1$.

The transformation F_E associated with E is the transformation defined by all the equations of E according to the above definition, i. e. $F_E(I)=\bigcup_{e_i \in E} F_E^i(I)$.

It is straightforward to show that the transformation F_E on the set of interpretations partially ordered by set inclusion is monotonic and continuous. Hence there exists the least fixed-point interpretation I^* such that $I^*=F_E(I^*)$, which can be obtained by iteratively applying F_E, starting with the empty subset of I_E, which is the bottom element of the partially ordered set of interpretations.

The transformation F_E is a bottom-up consequence-finding inference rule which builds up the theory from the axioms. The semantics based on such a transformation will then be called *bottom-up fixed-point-semantics.*

DEFINITION 5: Given a set of equations E, the denotation $bufp(E)$ defined by the bottom-up fixed-point semantics is the subset of I_E which is the least fixed-point of F_E.

THEOREM 3: $bufp(E)=mt(E)=tdo(E)$.

Proof: The proof is similar to the proof of theorem 2, since transformation F_E is a consequence-finding inference rule (an extension of hyperresolution). $Bufp(E)$ is then the set of all the ground atomic formulas which are true under all the interpretations and therefore it is the same as $mt(E)$ and $tdo(E)$.

We want finally to compare the bottom-up fixed-point semantics and the top-down mathematical semantics. The top-down transformation s_E was the basis of the semantics of a single procedure. On the contrary, the bottom-up transformation gives the semantics to all the procedures in E. If we define

$$bufp(E, f_i) = \{ f_i(t_1, \ldots, t_n) = t \in bufp(E) \},$$

we are able to prove the following theorem.

THEOREM 4: $tdm(E, f_i) = bufp(E, f_i)$.

Proof: The proof is similar to the equivalence proof between top-down and bottom-up derivation of the language defined by a context-free-grammar. In our case, we show that, at each step, the set of ground atomic formulas $f_i(t_1, \ldots, t_n) = t$ derived by the bottom-up transformation (inference rule F_E) is the same as the set of ground atomic formulas generated by the top-down transformation (inference rule s_E and instantiation of symbolic constants).

We have thus defined two equivalent formal semantics. The bottom-up fixed-point semantics is based on a bottom-up inference rule and is defined by a fixpoint transformation. The top-down mathematical semantics is based on a top-down inference rule and is defined by the closure of a reduction transformation. Each semantics has its own induction technique. Thus, while the bottom-up proof rule is based on F_E and fixpoint induction, top-down proofs could be based on s_E (symbolic execution) and subgoal induction [14].

The above results can be considered a step towards a formal understanding of why symbolic execution works in program analysis. Actually, all the program verification systems mentioned in section [2-5, 8, 10-13, 17] are based on top-down proofs, subgoal induction and symbolic execution.

7. TOP-DOWN MATHEMATICAL SEMANTICS OF HIGH LEVEL PROGRAMMING LANGUAGES

In this section we will informally try to extend our results about symbolic execution to high level programming languages.

The only difference between transformation f_E (standard interpretation) and transformation s_E (symbolic interpretation) is nondeterminism. In fact, s_E is a mapping from sets of equations to sets of equations, while f_E is a mapping from equations to equations. We would like to keep symbolic interpretation as close as possible to standard interpretation even for programming languages more complex than *TEL*. The semantics of a programming language construct, such that its standard and symbolic interpretation are the same, will be completely defined by its standard operational semantics.

In the sequel, we will consider those constructs which are present in most high level programming languages and are either absent or very simple in *TEL*.

i) Primitive data types. The operational semantics of primitive operations does not allow to provide a denotation to the application of an operation to symbolic operands. For example, it is not defined the application of the primitive operation + to the symbolic constants a and b. In such a situation, the symbolic interpreter simply builds the symbolic expression $+(a,b)$. The semantics of symbolic expressions must be defined through a formal specification of the semantics of primitive data types.

ii) Variables, assignment, storage, pointers and side effects. In a symbolic interpretation, the assignment can always be executed numerically, provided that its operand of type location (variable or pointer) does not have a symbolic value. This constraint is always satisfied if the language does not possess primitive data types (with side effects) with operations which return locations. In standard high level programming languages, this implies reasonable constraints on array-like structured data types and rather heavy constraints on pointers. If such constraints are satisfied, we need no formalization of storage.

iii) Higher order types. If we want to be able to cope with higher order types, i. e. functional arguments, we have to define higher order domains and to provide symbolic constants and axioms for higher order types. Symbolic interpretation does not seem to cope naturally with such features.

iv) Environment. Basic environment operations (referencing, binding, etc.) are identical in symbolic interpretation and standard interpretation.

v) Sequence control. If the language does not allow to handle labels as data types, standard interpretation provides a meaning even to those sequence control operations like goto, for which it is rather complex to define a denotational semantics. One aspect which is worth further investigation is related to the semantics of conditionals. In fact, symbolic execution of conditionals generally leads to the so-called path condition, which is a conjunction of formulae stating the conditions under which a specific program path is executed. In our description of *TEL* top-down mathematical semantics, we have only considered the situation in which the path condition is a conjunction of bindings for symbolic constants.

8. CONCLUSIONS

We do not consider the top-down mathematical semantics an alternative to denotational semantics, which is, in our opinion, the best formal definition tool. Rather we believe that top-down mathematical semantics (which is a little more than a standard programming language implementation) can be very useful in program analysis (testing, verification and optimization). In fact, a program analysis system based on denotational semantics will act upon a complete formal definition of the programming language. If the system is interactive, the user will interact with a rather complex formal theory.

On the contrary, if some reasonable constraints (no expressions of type location, procedure and label) are imposed on the language, we can perform top-down program analysis using a symbolic interpreter and providing only a formal specification of primitive data types.

ACKNOWLEDGMENTS

The authors are indebted to J. F. Perrot for his useful comments and suggestions.

REFERENCES

1. V. Ambriola and G. Levi, *The equational language TEL: formal semantics and implementation*, IEI Internal Report *(in preparation)*.
2. P. Asirelli, P. Degano, G. Levi, A. Martelli, U. Montanari, G. Pacini, F. Sirovich and F. Turini, *A flexible environment for program development based on a symbolic interpreter*, Proc. 4th Int'l Conf. on Software Engineering, 1979, p. 251-263.
3. R. S. Boyer, B. Elspas and K. N. Levitt, SELECT. *A formal system for testing and debugging programs by symbolic execution*. Proc. Int'l Conf. on Reliable Software, 1975, p. 234-245.
4. R. S. Boyer and J. S. Moore, *Proving theorems about LISP functions*, J. ACM 22, 1975, p. 129-144.
5. R. S. Boyer and J. S. Moore, *A lemma driven automatic theorem prover for recursive function theory*, Proc. 5th Int'l Joint Conf. on Artificial Intelligence, 1977, p. 511-519.
6. R. M. Burstall, *Program proof, program transformation, program synthesis for recursive programs*. Rivista di Informatica, vol. 7, Suppl. 1, 1977, p. 25-43.
7. R. M. Burstall and J. Darlington, *A transformation system for developing recursive programs*. J. ACM 24, 1977, p. 44-67.
8. L. P. Deutsch, *An interactive program verifier* — Ph. D. — dissertation, Dept. of Comp. Sci., Univ. of California, Berkeley (May 1973).
9. J. A. Goguen, *Abstract errors for abstract data types*. Formal Description of Programming Concepts, E. J. Neuhold Ed., North-Holland, 1978, p. 491-522.

10. J. C. KING, *A new approach to program testing*. Proc. Int'l Conf. on Reliable Software, 1975, p. 228-233.
11. J. C. KING, *Symbolic execution and program testing*. Comm. ACM 19, 1976, p. 385-395.
12. G. LEVI and F. SIROVICH, *Proving program properties, symbolic evaluation and logical procedural semantics*. Mathematical Foundations of Computer Science 1975. Lecture Notes in Computer Science, Springer Verlag, 1975, p. 294-301.
13. R. L. LONDON and D. R. MUSSER, *The application of a symbolic mathematical system to program verification*, Proc. ACM Annual Conf., 1974, p. 265-273.
14. J. A. MORRIS and B. WEGBREIT, *Subgoal induction*, Comm. ACM 20, 1977, p. 209-222.
15. M. NIVAT, *On the interpretation of recursive polyadic program schemes*, Symposia Mathematica, vol. 15, 1975, p. 255-281.
16. A. M. PEGNA, *Una caratterizzazione della semantica dei linguaggi programmativi basata sulla valuatazione simbolica*, Proc. AICA, 77, 3, 1977, p. 93-99.
17. R. W. TOPOR, *Interactive program verification using virtual programs*, Ph. D. dissertation, Dept. of Artificial Intelligence, Univ. of Edinburgh (February, 1975).
18. M. H. VAN EMDEN and R. A. KOWALSKI, *The semantics of predicate logic as a programming language*, J. ACM 23, 1976, p. 733-742.