TOP-DOWN SEMANTICS OF FAIR COMPUTATIONS OF LOGIC PROGRAMS

M.H. van Emden
Department of Computer Science
University of Waterloo
Waterloo, Ontario, N2L 3Gl, Canada
and
M.A. Nait Abdallah
Unit for Computer Science
Department of Mathematical Sciences
McMaster University
1280 Main Street West
Hamilton, Ontario, L8S 4L8, Canada

CS-84-27
Revised January 1985

TOP-DOWN SEMANTICS OF FAIR COMPUTATIONS OF LOGIC PROGRAMS

M.H. van Emden

and

M.A. Nait Abdallah †

Department of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1
Canada

1. Introduction

An advantage of logic programming is the justification of results in terms of logical implication, a much more intuitive notion than the fixpoints used in the semantics of programming languages other than pure Prolog. But, as a conceptually equivalent notion, fixpoints play a useful role in the semantics of logic programs [6,1].

So far, we know two ways in which results of logic programs are justified: by means of successful SLD-derivations and by means of finite and failed SLD-trees [5,1]. A successful SLD-derivation has as result a positive assertion which is a logical implication of the program regarded as a theory of first-order logic [6,5]. A finite and failed SLD-tree has as result a negation which is a logical implication of a certain theory which is a strengthened version of the program (called the "completion") [4].

In this paper we are concerned with a more general notion of result from logic programs, with a correspondingly weaker justification: from the results in this more general sense one does not conclude a logical implication. Our work relates results of computations of logic programs to greatest fixpoints of the associated transformation, and hence to greatest models rather than least models. The latter are (for logic programs) intersections of all models, hence contain assertions true in all models; hence the logical implication referred to above.

Let us consider the usual notion of output of a logic program, for example the program P below.

$$P = \{sum(0, y, y),\ sum(s(x), y, s(z)) \leftarrow sum(x, y, z)\}$$

† Current address: Unit for Computer Science, Department of Mathematical Sciences, McMaster University, 1280 Main Street West, Hamilton, Ontario, L8S 4L8, Canada.

Consider the SLD-derivation

$$\begin{aligned}
q &= \leftarrow sum(u,v,s(s(w))) \\
&\qquad \mid u = s(u_1) \\
q_1 &= \leftarrow sum(u_1, v, s(w)) \\
&\qquad \mid u_1 = s(u_2) \\
q_2 &= \leftarrow sum(u_2, v, w) \\
&\qquad \mid u_2 = 0, v = w \\
q_3 &= \qquad \square
\end{aligned}$$

This successful SLD-derivation produces the answer

$$sum(s(s(0)), v, s(s(v)))$$

which is the initial query with the composition of substitutions applied to it. This answer is justified by: (see [5])

$$P \models \forall\ v.\ sum(s(s(0)),v,s(s(v))).$$

Here we consider answers in a different, but related, sense: the set of all variable-free instances of the above conclusion restricted to the Herbrand universe of P and the initial question:

$$[sum(s(s(0)), v, s(s(v)))]$$

where $[e]$ is the ***ground set*** of e: the set of all variable-free instances of the expression e belonging to the Herbrand universe H_P of the program P. It may happen that the initial query contains a functor not occurring in P. Then the Herbrand universe H_P should also include this functor. To avoid this complication we assume without loss of generality that the initial query contains only functors from P.

In the above example the answer is obtained by the composition of *all* substitutions of the derivation. This composition is only available at the end of the computation. Using it suggests a *batch view* of computation. Let us see if it is possible to view computation of logic programs as a *continuing process*. This suggests that we look at the partial answer associated with each step. In the above example the partial results as accumulated in the first argument make up the following decreasing sequence of sets:

$$[u], [s(u_1)], [s(s(u_2))], [s(s(0))]$$

We view the sequence as one of *successive approximations* to the final result $[s(s(0))] = \{s(s(0))\}$, which is the intersection of all elements of the sequence. This view of computation, where an output argument initially denotes the set of all possible answers and where this denotation progressively narrows down as the computation proceeds, was proposed by L. Nolin [12]. Its mathematical treatment [9] goes under the name of *sort theory*.

The continuing-process view of computation is particularly appropriate in the case of infinite derivations, where a result in the usual sense does not exist. For example, for the same program P, there exists the nonterminating derivation

$$\begin{array}{l} \leftarrow sum\,(u,\, s(0),\, u) \\ \quad \mid\ u = s(u_1) \qquad \theta_1 \\ \leftarrow sum\,(u_1,\, s(0),\, u_1) \\ \quad \mid\ u_1 = s(u_2) \qquad \theta_2 \\ \qquad \cdot \\ \qquad \cdot \\ \qquad \cdot \end{array}$$

The partial results are the elements of the nested sequence

$$[u] \supseteq [u\theta_1] \supseteq [u\theta_1\theta_2] \supseteq \cdots$$

i.e.

$$[u] \supseteq [s(u_1)] \supseteq [s(s(u_2))] \supseteq \cdots$$

In the absence of a last element one is tempted to consider as result the limit

$$\cap\ \{[s^n(u_n)] \mid n \in \mathbf{N}\}$$

which is, however, empty. This is because the sets are included in the usual Herbrand universe. Its terms are finite objects; none of them is in all the sets of the sequence.

Yet infinite computations in Prolog are useful: see [2,10]. For this reason mainly we propose to include infinite terms. Mathematically speaking, we replace the Herbrand universe H by its *completion*, defined as follows [8]. To begin with we define a metric d on the terms of the Herbrand universe.

$$d(t,t') = 0 \quad \text{if} \quad t = t'$$
$$d(t,tprime) = 2^{-\inf\{n \mid \alpha_n(t) \neq \alpha_n(t')\}} \quad \text{if} \quad t \neq t'$$

where $\alpha_n(t)$ denotes the cut at height n of tree t. In other words, the exponent of the power of 2 is minus the smallest depth where t and t' differ.

This metric allows us to define the completion $\overline{H}$ of H in much the same way as the set of reals may be defined as a completion of the set of rationals. The metric on H makes it possible to define Cauchy sequences of elements of H, and to define the usual equivalence relation between Cauchy sequences. $\overline{H}$ is now the set of equivalence classes of Cauchy sequences.

Given a term t, we define the ***infinitary ground set*** of t as the set of all variable-free instances of t occurring in $\overline{H}$. Note that the infinitary ground set of any term t is the topological closure (in the topology introduced above) of the ground set of t. Hence the infinitary ground set is closed for any t. In the sequel we are only concerned with the infinitary case, so $[t]$ will henceforth mean the ***infinitary*** ground set of t.

Consider the mapping from rationals into reals where a rational q maps into the equivalence class containing the Cauchy sequence $q, q, q, \cdots$. The reals which are not images under this mapping are the irrational reals. The elements of $\overline{H}$ which are not images under the corresponding mapping from H can be regarded as infinite terms, as H already contains all finite terms constructible from the given vocabulary.

It has been observed by Mycielski and Taylor [8] that $\overline{H}$ is a compact metric space provided that finitely many symbols are used for the generation of H. This compactness property is what we are after: it guarantees that nested sequences of nonempty closed subsets have a nonempty intersection [3].

2. Tree derivations

We use the definitions usual in the syntax and semantics of logic programs [6,1]. In addition to these we also need those presented in this section. We also use the same terminology, except that nowadays we prefer to use "query" instead of "goal statement".

The Herbrand base B_P, which is usually defined as the set of all atomic formulas with terms in H_P, is here defined as the infinitary counterpart: the set of atomic formulas with terms in $\overline{H}_P$.

With each logic program P we associate a mapping T_P from subsets of B_P to subsets of B_P, and defined as follows.

Definition: $A \in T_P(I)$ iff there exists a variable-free instance (using terms from $\overline{H}_P$) $A \leftarrow B_1, \ldots, B_n$ of a clause in P such that $\{B_1, \ldots, B_n\} \subseteq I$.

We have extended the usual definition of T_P by replacing H_P by $\overline{H}_P$. We may drop the subscript P when the program is obvious from the context.

The main component of an SLD-derivation in the sense of [1] is a sequence of negative clauses. We find it useful to consider a tree form for each of these clauses, called *and-trees.* Such trees have atomic formulas as nodes; some of the leaf nodes may be marked *closed.* An and-tree is called *closed* if all its leaves are marked closed. And-trees are defined in the course of the definition of a "tree derivation".

A *tree derivation* for a program P is a sequence of quadruples of the form $< at, p, ic, \theta >$ where at is an and-tree, p is a pointer, ic is a clause in P and θ is a substitution. The pointer p "points" to a non-closed leaf node of at ("closed leaves" are defined presently). We can think of it as an integer such that the leaf node pointed to is the p-th non-closed leaf node counted from the left. A quadruple $Q' = < at', p', ic', \theta' >$ is *P-derived* from a quadruple $Q = < at, p, ic, \theta >$ iff p points to G, ic is $A \leftarrow B_1, \ldots, B_k$ such that θ is a most general unifier of G and A and at' is at with $B_1, \ldots, B_k$ attached as descendants of G and then with θ applied to all nodes. In case $k = 0$ in the clause ic, no node is attached, the substitution is also applied and G is marked *closed.*

We are now in a position to define a tree derivation for a program P and a query $\leftarrow A$ with A an atomic formula (we can assume this without loss of generality). It is a sequence of quadruples where the first has A as and-tree containing one node and each next quadruple is P-derived from the previous one.

When we replace in a tree derivation in each quadruple the and-tree by a query consisting of the nonclosed leaf nodes of the and-tree in order from left to right, we have substantially the SLD-derivation from [1]. It is clear that, when an and-tree has closed leaves only, its quadruple can have no successor in a derivation. The corresponding SLD-derivation is called *successful*; we use this attribute also for the tree derivation. In the last quadruple of a successful tree derivation, only the first component is defined (and it is a closed and-tree).

We illustrate tree derivations using program P of section 1. The completed Herbrand universe $\overline{H}_P$ is $\{s^n(0) \mid n \in \mathrm{N}\} \cup \{s^\omega\}$. Let the query be $\leftarrow sum(u, s(0), u)$. The tree component of the n-th quadruple of the tree derivation is

$$\begin{array}{c} sum(s^n(u_n), s(0), s^n(u_n)) \\ | \\ \cdot \\ \cdot \\ \cdot \\ | \\ sum(u_n, s(0), u_n) \end{array}$$

Note that

$$[q] \cap T_P^n(B_P) = \{sum(s^n(a), s(0), s^n(a)) \mid a \in \overline{H}_P\}$$

where q is the atom in the query $\leftarrow sum(u, s(0), u)$. Apparently the right-hand side is the ground set of the root of the tree in the n-th quadruple of the tree derivation. Theorem 3.1 (see below) shows that this equality is not a coincidence.

Lemma 2.1.

Any variable-free instance from $\overline{H}_P$ of the root of a closed and-tree for a program P is in $T_P^{m+1}(\emptyset)$, where m is the length of a longest path from the root to a leaf of the and-tree.

Proof.

Let an and-tree and a variable-free substitution of its root be given. Apply this substitution throughout the and-tree. Apply an arbitrary variable-free substitution to any remaining variables in the and-tree. Every instance of a closed leaf node is in $T(\emptyset)$. Hence, of the resulting and-tree it is true that any node is in $T^{k+1}(\emptyset)$ where k is the length of a longest path to a leaf reachable from it. This follows from the definition of T and of and-trees □

Note that this lemma is a strengthened version of theorem 5.1 of [1] which states that variable-free instances of the root are in $T^n(\emptyset)$, where n is the length of the derivation corresponding to the given and-tree. If the clauses used in the derivation have at most one atom in their conditions, then $n = m$; otherwise $n > m$ □

An and-tree is defined to be k-fair if the shortest path from a nonclosed leaf to the root has length k. An and-tree is defined to be *fair* if there is no nonclosed leaf. Hence a successful and-tree is fair, and so are certain infinite and-trees.

Lemma 2.2.

Every variable-free instance of the root of a k-fair and-tree is in $T_P^k(B_P)$.

Proof.

As in the proof of lemma 2.1, consider a variable-free instance of the entire and-tree. Because leaf nodes are not necessarily closed, we only assume that they are in B_P. Now, the definition of T_P and of and-trees imply that every node n is in $T_P^k(B_P)$, where k is the length of a shortest path from n to a leaf node descendant □

An SLD-derivation is defined to be *k-fair* if the last and-tree in the corresponding tree derivation is *k-fair*. Two derivations (SLD-or tree-) are defined to be *homologous* if they refer to the same program, if they have the same length, and if, for each i, the pointer and the input clause components of the i-th quadruples are the same. Note that a derivation homologous to a successful one is successful.

In resolution theorem-proving "lifting lemmas" play an important role. For SLD-derivations the following one is useful.

Lemma 2.3.

Let there exist a derivation of $P \cup \{N\theta\}$, where P is a program and N a query. Then there exists a homologous derivation of $P \cup \{N\}$ $\Box$

3. Semantics of fair derivations

Our aim is to provide semantic justification for derivations independent of whether they are successful. In this section we give a characterization of the results of k-fair derivations. Our main result is the following.

Theorem 3.1
For all k = 0,1,..., for all atomic formulas q, and for all programs P,

$$[q] \cap T_P{}^k(B_P) = \bigcup \{[q\ \theta_1 \cdots \theta_n] \mid \exists \text{ a } k\text{-fair derivation using program } P, \text{ starting with } \leftarrow q, \text{ and having } \theta_1, \ldots, \theta_n \text{ as substitutions} \}$$

Proof.

Suppose that a k-fair derivation exists, starting with $\leftarrow q$. In the tree form of the derivation the last tree is one where no nonclosed leaf is closer to the root (which is $q\theta_1 \cdots \theta_n$) than k. For every $A \in [q\ \theta_1 \cdots \theta_n]$ there exists a variable-free instance of the entire and-tree having A as root. By lemma 2.2, $A \in T^k(B)$.

It remains to show that for every $A = [q] \cap T^k(B)$ there exists a k-fair derivation starting from A. We prove this by induction on k. It is obvious for $k = 0$. Let us assume it for $k - 1$. $A \in T^k(B)$ implies that there is a variable-free instance $R\theta \leftarrow Q_1\ \theta, \ldots, Q_q\theta$ of a clause $C = (R \leftarrow Q_1, \ldots, Q_q)$ in P such that $R\theta = A$ and $Q_1\ \theta, \ldots, Q_q\ \theta$ are in $T^{k-1}(B)$. By the induction assumption, $(k-1)$-fair derivations exist, starting from $Q_i\ \theta$, for $i = 1, \ldots, q$. Let $t_1, \ldots, t_q$ be the $(k-1)$-fair and-trees that exist by the induction assumption at the end of these derivations. Let t be the tree having $\leftarrow A$ as root and $t_1, \ldots, t_q$ as subtrees.

Note that t is not necessarily an and-tree. However, we can derive one from t. Let η be the most general unifier of A and R; hence $\theta = \eta\xi$ for some substitution ξ. Now t can be used to construct a k-fair and-tree t' with $\leftarrow A$ as root. The successors of the root in t' are $Q_1\ \eta, \ldots, Q_q\ \eta$. The induction assumption guarantees the existence of $(k-1)$-fair and-trees $t_1, \ldots, t_q$ with $Q_1\ \eta\ \xi, \ldots, Q_q\ \eta\ \xi$ as roots. By the lifting lemma 2.3 $(k-1)$-fair and-trees with $Q_1\ \eta, \ldots, Q_q\ \eta$ as roots exist, and these are the subtrees of the k-fair and-tree t with $\leftarrow A$ as root □

Theorem 3.1 is useful for characterizing infinite computations of Prolog programs. These are not mere curiosities of interest only to theoreticians: in [3] it is shown that Prolog provides an elegant formalism for programming in the style typical of the LUCID language. A simple example of this style used in Prolog is the following:

$$\textit{list-succ}(u \cdot x, s(u) \cdot y) \leftarrow \textit{list-succ}(x,y) \tag{1}$$

where $\textit{list-succ}(\alpha, \beta)$ means that α and β are infinite sequences of natural numbers in successor notation $(0, s(0), s^2(0), \ldots)$ and that β is, element for element, the successor of α. The query

$$\leftarrow \textit{list-succ}(0 \cdot z, z) \tag{2}$$

asks whether a z exists such that $\textit{list-succ}(0 \cdot z, z)$. This indeed is the case: it is the infinite sequence $s(0) \cdot s^2(0) \cdot s^3(0) \cdots$. The query will cause Prolog to construct the sequence as far as resources of time and space allow.

We have defined the result of this infinite computation to be

$$\bigcap \{[\textit{list-succ}(0 \cdot z_0 \;\; z_0)\, \theta_1 \cdots \theta_n] \mid n \in \mathbf{N}\} \tag{3}$$

when θ_i substitutes $s^i(0) \cdot z_i$ for z_{i-1}. We saw that this intersection is nonempty when the completion $\overline{H}$ of the Herbrand universe is the underlying domain. The query (2) yields only one possible derivation with program (1). This derivation is k-fair for all $k \in \mathbf{N}$. Hence our theorem implies that the result (3) of this infinite derivation is

$$[\textit{list-succ}(0 \cdot z \;, z)] \cap \bigcap \{T^k(B) : k \in \mathbf{N}\}\,.$$

Another application of theorem 3.1 establishes a connection between k-fairness and depth of finitely failed SLD-trees. Suppose that an atom A begins a k-fair derivation. According to theorem 3.1, $A \in T^k(B)$. By theorem 7.1 in [1], A is not the root of any finitely failed SLD-tree of depth at most k.

Conversely, suppose that no k-fair derivation starts from A. According to theorem 3.1, A does not belong to $T^k(B)$. By the results of Lassez and Maher [7] it follows that a finitely failed SLD-tree exists with A as root. The depth of this tree must be at least k, and may exceed k by an arbitrary large number.

4. Least and greatest models and fixpoints

An interesting consequence of theorem 3.1 is

$$\begin{aligned}[q] \cap \bigcap \{T^k(B) : k \in \mathbf{N}\} = \bigcup \{[q\, \theta_1, \theta_2, \cdots] \\ &\mid \exists \text{ a fair derivation using program } P, \text{ starting with} \\ &\quad \leftarrow q, \text{ and having } \theta_1, \theta_2, \ldots \text{ as substitutions} \\ &\}\end{aligned} \tag{4}$$

The inclusion of the right-hand side in the left-hand side is obvious. Note that $A \in T^k(B)$; hence k-fair derivations start from $\leftarrow A$ for all k. The other inclusion now follows from the fact that among all k-fair derivations for a given k, at least one must be extendable to a fair derivation.

Note that in (4), $\cap \{T_P^k(B) : k \in N\}$ is the greatest fixpoint of T_P. This result is due to Tiuryn [14] and was independently proved by Nait Abdallah [10,11]. The greatest fixpoint is also the greatest model of the completion of the program P [1]. Thus our theorem enables infinite fair computations to be characterized by means of greatest models or fixpoints.

More familiar is the use of least fixpoints to characterize results, in the finitary case, of computations, as pioneered by Scott [13]. (Note that Theorem 3.1 also holds in the finitary case.) In the context of logic programming [6] introduces least fixpoints and shows that Herbrand models of logic programs are closely related to fixpoints of the transformations associated with the programs. Lemma 2.1. (and the similar theorem 5.1. in [1]) show that the result of a successful derivation is in $T^k(\varnothing)$ for some finite k. As, in the finitary case, $\cup \{T^k(\varnothing) : k \in N\}$ is the least fixpoint of T, successful derivations approximate the least fixpoint.

Because $\varnothing$ is an instance of "bottom" in Scott's partially ordered sets of data, we call such a characterization "bottom-up". Theorem 3.1 takes B as starting point, which plays the role of Scott's "top". Hence it provides a "top-down" characterization of results of logic programs.

Thus it seems that we have two competing models of computation: one using least fixpoints and one using greatest fixpoints. Some reflection shows that the two fixpoints play different, and, to a certain extent, complementary roles.

Theorem 3.1 is most widely applicable: it applies to *all* derivations (failed or successful, finite or infinite; they are k-fair for *some* k). And it provides approximations to the greatest fixpoint. Because it applies to all derivations, it applies to all initial segments of a given derivation. Hence the approximations are obtained as the derivation is being constructed, in a continuing process.

Given any atomic query $\leftarrow q$, a good picture of this process is obtained by considering a "microcosm" consisting only of the ground set $[q]$, neglecting its complement $B - [q]$. This means that any set now is only considered through its intersection with $[q]$. As a successful derivation is constructed, theorem 3.1 can be used to characterize the partial results as belonging to $T^k(B)$, for nondecreasing k, being approximations to the greatest fixpoint. As soon as success is reached at the end, the result is known to be in $T^n(\varnothing)$. Thus the partial results are bounded only by sets *above* the *greatest* fixpoints, whereas the final result is bounded by a set *below* the *least*, a drastic improvement usually.

The improvement is not only quantitative. The least fixpoint is the least Herbrand model, which is the intersection of all Herbrand models (as the program contains Horn clauses only). It can be shown [6] that an atom is the result of a successful derivation if and only if it is a logical consequence of the program. This is a much stronger conclusion than any that can be drawn from Theorem 3.1.

5. Acknowledgements

We gratefully acknowledge financial support for document preparation from the National Science and Engineering Research Council. MHvE thanks the Science and Engineering Research Council (UK) for support in the form of a Senior Visiting Fellowship. We thank Andre Trudel for his careful reading of the paper and for his suggestions for improvement.

6. References

[1] K.R. Apt and M.H. van Emden: Contributions to the theory of logic programming. Journal of the ACM, **29** (1982), pp. 841-862.

[2] D.R. Brough and M.H. van Emden: Dataflow, flowcharts, and "LUCID"-style programming in logic. Proc. IEEE Symposium on Logic Programming, Atlantic City, 1984, pp. 252-258.

[3] N. Bourbaki: General Topology, Addison-Wesley, 1966.

[4] K.L. Clark: Negation as failure; pp. 293-324 in: Logic and Data Bases, H. Gallaire and J. Minker (eds.), Plenum, 1978.

[5] K.L. Clark: Predicate logic as a computational formalism. Research Monograph 79/59, Department of Computing, Imperial College, London, 1979.

[6] M.H. van Emden and R.A. Kowalski: The semantics of logic as a programming language. Journal of the ACM, **23** (1976), 733-742.

[7] J-L. Lassez and M.J. Maher: Chaotic semantics of programming logic. Theoretical Computer Science, 1984, pp 167-184.

[8] J. Mycielski and W. Taylor: A compactification of the algebra of terms. Algebra Universalis, **6** (1976), pp. 159-163.

[9] M.A. Nait Abdallah: Sort theory, Research Report CS-82-19, Department of Computer Science, University of Waterloo, 1982.

[10] M.A. Nait Abdallah: Metric interpretation and greatest fixpoint semantics of logic programs, Research Report CS-83-29, Department of Computer Science, University of Waterloo, 1983.

[11] M.A. Nait Abdallah: On the interpretation of infinite computations in logic programming. ICALP 84, Springer LNCS 172, pp 358-370.

[12] L. Nolin: Algorithmes universels, RAIRO Theoretical Computer Science, **2** (1974), pp 5-18.

[13] D. Scott: Outline of a mathematical theory of computation. 4th Annual Princeton Conference on Information Science and Systems, 1970, pp. 169-176.

[14] J. Tiuryn: Unpublished letter dated 13 January 1979.