MATHEMATICAL PROBLEMS IN COMPUTATION THEORY
BANACH CENTER PUBLICATIONS, VOLUME 21
PWN - POLISH SCIENTIFIC PUBLISHERS
WARSAW 1988

FINITE AND INFINITE COMPUTATIONS OF LOGIC PROGRAMS

M. A. NAIT ABDALLAH

Department of Computer Science, University of Western Ontario, London, Ontario, Canada

In this paper, a tutorial introduction to the theory of finite and infinite computations of logic programs is given. The basic known results on finite computations and least fixpoint semantics are first outlined. The infinite computations are then introduced. We end up with some results on infinitary tree semantics of logic programs.

0. Introduction

In this paper we give a tutorial introduction to finite and infinite computations of logic programs. No prerequisite in logic programming is assumed. We are here more interested in the articulations of the ideas involved in logic programming computations, than in the strictly linear exposition of the theory. We shall introduce the concepts and definitions as we go, and as they happen to be needed for our exposition.

This paper is organized as follows. Section I gives an example of finite derivation of a logic program, with some of the intuition behind logic programming. Section II contains formal definitions of some fundamental notions: logic programs, Herbrand universe and Herbrand base, the subset transformation T. Section III describes the least fixpoint semantics of a logic program and its computational characterization through successful derivations. A canonical partition of the Herbrand base in the finitary case is given in Section IV. In Section V we give two examples of infinite logic program computations. This leads to the metric topology introduced on the Herbrand base and universe in Section VI. Section VII is devoted to the infinitary-tree semantics of logic programs described by the least fixpoint, the least closed fixpoint, and the greatest fixpoint of the transformation T. A canonical partition of the complete Herbrand base is given in Section VIII.

I. An example of finite derivation

Before going into the technical details, we first give an informal example of a finite derivation of a logic program and how it works. To make things simpler, we shall start *in medias res* in order to justify the formal definitions given later on. We consider the following logic program:

EXAMPLE 1. 1. $\mathrm{sum}\,(0, x, x) \leftarrow$.
2. $\mathrm{sum}(s(x), y, s(z)) \leftarrow \mathrm{sum}(x, y, z)$.

This program defines the sum of natural numbers. The first line says that, for every x, the sum of 0 and x is equal to x. The second line is the induction step of this definition. It says that if the sum of x and y is z, then the sum of the successor of x and y is equal to the successor of z. The general format of each one of these lines is that of a *definite clause*. A definite clause is a quantifier-free first-order formula of the form

$$A \leftarrow B_1 \& \ldots \& B_m, \qquad m \geqslant 0,$$

where A, $B_1, \ldots, B_m$ are all atomic formulae. This will be made more precise in the sequel. The general meaning of a definite clause such as the above is the following: *If B_1 and B_2 and ... and B_m are all true, then A is also true.* Notice that the right-hand side (i.e. the B-part) of the clause may be empty, i.e., may be an *empty conjunction*. Since an empty conjuction amounts to *true*, the clause in that case means that the A-part (i.e. the left-hand side) is true without any condition. This is the case in the first line of our sum program. Note finally that each definite clause is *implicitly* universally quantified. Now let us consider the following *query* asked from our sum program:

3. $\leftarrow \mathrm{sum}(s^2(0), u, v)$.

Here again we use the clausal format. In this case the left-hand side of the clause is empty. Since for general clauses the left-hand side is a disjunction, this amounts to saying that here the left-hand side amounts to *false*, i.e., using the implicit universal quantification, we are saying that *For any u, for any v, the fact that v is equal to the sum of $s^2(0)$ and u implies false*, or, equivalently, *there ėxist no u and no v such that v is the sum of $s^2(0)$ and u*. A *derivation* or *refutation* of this query is going to combine this statement with the information contained in the logic program defining sum in order to obtain a contradiction; this is the basic computation mechanism of logic programs. This is done step by step by using the logical rule of *Modus Tollens*. Modus Tollens works as follows: *If $\sim A$, and B implies A, then $\sim B$.* In clausal form this may be said as follows: *from $\leftarrow A$, and $A \leftarrow B$, derive $\leftarrow B$.* Of course since here we are using object variables (for example variables u and v in line number 3), an important preliminary step, before applying Modus Tollens, will be to make the A-*part* of $\leftarrow A$, and the A-part of $A \leftarrow B$ coincide exactly, if this is not already the case. This is done by renaming variables and/or binding them in some suitable way. This important intermediary step is called *unification* [13]. More precisely, we have in the present case:

1. $\text{sum}\ (0, x, x) \leftarrow$,		
2. $\text{sum}\ (s(x), y, s(z)) \leftarrow \text{sum}(x, y, z)$,		
3. $\leftarrow \text{sum}(s^2(0), u, v)$,		
4. $\leftarrow \text{sum}(s(0), u, v_1)$,	$v = s(v_1)$	2, 3, MT,
5. $\leftarrow \text{sum}(0, u, v_2)$,	$v_1 = s(v_2)$	3, 4 MT,
6. $\Box$,	$u = v_2 = x$	1, 5 MT.

The last line of this derivation has the form "$\leftarrow$", i.e., according to our conventions, states that "*true* implies *false*", which is a contradiction. Such a clause is called the *empty clause* and is denoted by $\Box$. Now the result of this computation is obtained by "gluing" together the bindings used in the derivation, and by restricting them to the sole variables occurring in the initial query. We obtain here the substitution $\vartheta = \langle v, s^2(u)\rangle$. This substitution is called the *answer substitution*. Indeed, the u's and v's which are solutions of the equation $s^2(0)+u = v$ are exactly those for which v is the second successor of u. Thus we have solved the original query.

Notice that the above answer substitution makes sense only because our derivation ended in the empty clause $\Box$. In particular, our derivation must be *finite*.

In this section we have given a general idea of how logic programs work. In the next section we shall give formal definitions for such programs, and study their theory.

II. Basic definitions

We now give formal definitions for some of the concepts used informally in the previous section.

A *logic program* is a finite set of definite clauses. A *definite clause* is an expression of the form

$$A \leftarrow B_1 \,\&\ldots\&\, B_m, \qquad m \geqslant 0,$$

where A, $B_1, \ldots, B_m$ are atoms. An *atom* is an expression of the form $p(t_1, \ldots, t_k)$, where p is a k-ary relation symbol, and $t_1, \ldots, t_k$ are terms. A *term* is either a variable, a constant, or an expression of the form $f(t_1, \ldots, t_n)$, where f is an n-ary function symbol, and $t_1, \ldots, t_n$ are terms.

An example of a logic program was given in the previous section; the clauses of that program were numbered.

1. $\text{sum}(0, x, x) \leftarrow$,
2. $\text{sum}(s(x), y, s(z)) \leftarrow \text{sum}(x, y, z)$.

A *goal clause*, or *query*, is an expression of the form

$$B_1 \,\&\ldots\&\, B_m, \qquad m \geqslant 0,$$

where $B_1, \ldots, B_m$ are atoms. An example of goal clause was line number 3 of the previous example:

3. $\leftarrow \mathrm{sum}(s^2(0), u, v)$.

Let P be a logic program. Let V be an infinite set of variables containing the variables of P. Let $F = F_0 \cup F_1 \cup \ldots$ be the set of function symbols occurring in P, where $f \in F$ belongs to F_i if and only if f is of arity i. Let $R = R_0 \cup R_1 \cup \ldots$ be the set of relation symbols occurring in P. We may notice that both F and R are finite, because P is finite. In the previous example $F = F_0 \cup F_1$, with $F_0 = \{0\}$ and $F_1 = \{s\}$, and $R = R_3 = \{\mathrm{sum}\}$.

We define the *Herbrand universe* H_u of the logic program P as being the set of all variable-free terms that can be constructed from the symbols of P. Thus H_u is the free F-algebra on the generating set $\emptyset$. In the sum example, the Herbrand base H_u is given by the set of all codings of natural numbers:

$$H_u = \{s^i(0) \colon i \in N\} = \{0, s(0), s^2(0), \ldots\}.$$

We also define the *Herbrand base* H_b of the logic program P as being the set of all atoms with relation symbols from R and entries from H_u. Thus H_b is exactly the set of all variable-free atoms that can be constructed with symbols from P. In the sum example, the Herbrand base consists of all the correct and incorrect "sums" of natural numbers:

$$H_b = \{\mathrm{sum}(a, b, c) \colon a, b, c \in H_u\}.$$

Finally, following van Emden and Kowalski [4], we define the following *one-step modus ponens transformation* T defined on the power set of the Herbrand base:

$$T\colon P(H_b) \to P(H_b),$$

$$S \to \{A\vartheta \colon A\vartheta \leftarrow B_1\vartheta \& \ldots \& B_m\vartheta \text{ is a variable-free instance of a clause of } P\text{, and } B_1\vartheta, \ldots, B_m\vartheta \in S\}.$$

Thus intuitively, what T does is apply, to all the (correct and incorrect) claims contained in the set of atoms S, a one-step modus ponens using the definite clauses of the program. The set $T(S)$ is exactly what can be deduced from S in one step, by using the rules which are in P. Notice that this definition uses substitutions. A *substitution* ϑ is a mapping:

$$\vartheta\colon V \to H_u(V)$$

which is equal almost everywhere to the identity. Thus a substitution ϑ may be represented by a finite list of pairs:

$$\vartheta = \{\langle x_1, t_1\rangle, \ldots, \langle x_n, t_n\rangle\}$$

such that $\vartheta(x_i) = t_i$ corresponding to the points where ϑ is not the identity. We define the result $t\vartheta$ of applying the substitution ϑ to the term t as follows.

(i) $x\vartheta = \vartheta(x)$ if $x \in V$ is a variable.

(ii) $c\vartheta = c$ if $c \in F_0$ is a constant.

(iii) $f(t_1, \ldots, t_n)\vartheta = f(t_1\vartheta, \ldots, t_n\vartheta)$ if $f \in F_n$ is a function symbol and $t_1, \ldots, t_n$ are terms.

This is generalized to atoms by taking:

$$r(t_1, \ldots, t_n)\vartheta = r(t_1\vartheta, \ldots, t_n\vartheta)$$

if $r \in R_n$ is a relation symbol and $t_1, \ldots, t_n$ are terms.

III. The least fixpoint semantics of a logic program and its computational characterization

We have the following result [4]:

THEOREM 3.1 (van Emden, Kowalski). *The transformation T is Scott-continuous, i.e., for any ascending sequence of subsets (S_i), we have $T(\bigcup_i S_i) = \bigcup_i T(S_i)$.*

Proof. The transformation T is clearly monotone increasing, thus $\bigcup_i T(S_i) \subseteq T(\bigcup_i S_i)$. The inclusion in the other direction is obtained by remarking that each definite clause has only finitely many atoms as premisses. (Thus what we really have here is a compactness theorem.)

Since T is continuous, by the Knaster–Tarski *least fixpoint theorem*, it has a least fixpoint given by

$$\mathrm{lfp}(T) = \bigcup_n (T^n(\emptyset)).$$

We shall take this least fixpoint as the *mathematical definition of the meaning of the logic program P*. This corresponds to the usual least fixpoint semantics approach advocated by Scott and Strachey [14]. In the sum example, we have

$$T^n(\emptyset) = \{\mathrm{sum}(s^k(0), u, s^k(u)) \colon k \leqslant n,\ u \in H_u\},$$

$$\mathrm{lfp}(T) = \{\mathrm{sum}(s^n(0), u, s^n(u)) \colon u \in H_u,\ n \in N\}.$$

We now link this definition of the meaning of a logic program P first to Herbrand models, and then to computations. More precisely, we are going to show that lfp(T) is the smallest Herbrand model of P, and also give a computational characterization of this set.

We define *Herbrand interpretations* as being subsets of the Herbrand base. (Intuitively speaking, the atoms of a given subset are exactly those which are made true under the corresponding interpretation.) A *Herbrand model* of a logic program P is a interpretation $I \subseteq H_b$ of P such that the following condition holds: for any clause $A \leftarrow B_1 \& \ldots \& B_m$ of the program P, for any variable-free instance $A\vartheta \leftarrow B_1\vartheta \& \ldots \& B_m\vartheta$ of that clause, where

ϑ is some substitution, whenever $B_1\vartheta, \ldots, B_m\vartheta$ are all true under the interpretation I, then $A\vartheta$ is also true under the interpretation I.

THEOREM 3.2 (van Emden, Kowalski). $\operatorname{lfp}(T)$ *is the smallest Herbrand model of* P.

Thus by Gödel's completeness theorem for first-order predicate calculus, the meaning $\operatorname{lfp}(T)$ of a logic program P is exactly the set of all variable-free atoms which are logic consequences of P.

The next question to handle is whether there is a computational characterization of this least Herbrand model. This is done through the use of *SLD-derivations* (called here *derivations*).

We shall say that a substitution ϑ is a *unifier* for terms (resp. atoms) t_1 and t_2 if and only if $t_1\vartheta = t_2\vartheta$. A substitution ϑ is a *most general unifier* (*mgu*) for t_1 and t_2 if and only if ϑ is a unifier for t_1 and t_2, and $\forall\sigma \ \ t_1\sigma = t_2\sigma \Rightarrow \exists\gamma \ \ \sigma = \vartheta\gamma$. Robinson [13] gives a terminating algorithm which computes a most general unifier for any pair of unifiable terms.

We define a *derivation step* from a goal g as a triple $\delta = \langle\lambda, r, \vartheta\rangle$ such that:

(i) λ is an occurrence of an atom in g,

(ii) r is a rule variant of some clause of a logic program P, with no variables in common with g, $A \leftarrow B_1 \& \ldots \& B_m$,

(iii) ϑ is a most general unifier of the atom $(g\downarrow\lambda)$ of occurrence λ in g, and the left-hand side A of rule variant r.

The *yield* $g' = \delta(g)$ of performing the derivation step δ on the goal g is obtained from g by replacing the atom of occurrence λ in g by $B_1\vartheta \& \ldots \& B_m\vartheta$, and applying the substitution ϑ to the rest of the goal g.

A *finite derivation* from g is defined as a finite sequence

$$\Delta = g_0, \delta_1, g_1, \ldots, \delta_n, g_n$$

such that:

(i) $g_0 = g$,

(ii) δ_i is a derivation step $\langle\lambda_1, r_i, \vartheta_i\rangle$ from t_{i-1} for every $i = 1, \ldots, n$.

(iii) $t_i = \delta_i(t_{i-1})$ for every $i = 1, \ldots, n$.

The goal $g = g_0$ will be called the *root* of the derivation.

A finite derivation is *successful* iff it reaches the empty clause.

We now have the following characterization theorem for the least Herbrand model $\operatorname{lfp}(T)$ [1].

THEOREM 3.3 (Apt, van Emden).

$$\operatorname{lfp}(T) = \{a \in H_b\text{: } a \textit{ is the root of some successful derivation}\}.$$

At this stage the mathematical meaning of a logic program P has been characterized computationally, i.e., in terms of successful derivations. The question now is to make this characterization effective (i.e. constructive): how do we get these successful derivations? This question is answered by the following results, due to Apt and van Emden [1]. It basically says that *every SLD-tree is OK*. In other words, if we have a variable-free successful atom, then by picking up any SLD-tree rooted at a, if we are patient enough, by searching this SLD-tree we shall find a path in that tree which is a successful derivation from a. But first we need to define SLD-trees.

An SLD *search procedure* [1] is a procedure that constructs derivations and finds a successful derivation whenever one exists. To each SLD search procedure, given a goal g, we associate an *SLD-tree rooted at* g defined as follows: this tree is obtained by coalescing all initial segments of all derivations rooted at g that can be generated by using the search procedure.

THEOREM 3.4 (Apt, van Emden). *If* $a \in \mathrm{lfp}(T)$, *then every SLD-tree rooted at a contains a successful derivation.*

The above results characterize the elements of the least fixpoint of T in terms of finite *successful* derivations. There are also in the Herbrand base a class of elements which are not successful, but which can be characterized in a finitary manner; these are the *finitely failed elements*. An element $a \in H_b$ is finitely failed iff it is the root of some finite SLD-tree which contains no successful derivation. We have the following result [1]:

THEOREM 3.5 (Apt, van Emden). $FF(P) = \{a \in H_b\colon\ a \notin \bigcap_n T^n(H_b)\}$, *where* $FF(P)$ *designates the set of finitely failed elements of* H_b.

IV. Canonical partition of the Herbrand base

The results given in the previous section yield the following *canonical partition of the Herbrand base.*

finitely failed atoms $H_b - \bigcap\{T^q(H_b)\colon\ q \in N\}$
"infinite" atoms
results of successful derivations $\mathrm{lfp}(T) = \bigcup\{T^n(\emptyset)\colon\ n \in N\}$

Table 1. Canonical partition of the Herbrand base H_b

Notice that $\bigcap\{T^q(H_b)\colon\ q \in N\}$ is generally not the greatest fixpoint of T. For example for the program $P = \{M(0) \leftarrow N(x);\ N(s(x)) \leftarrow N(x)\}$ we have $\bigcap\{T^q(H_b)\colon\ q \in N\} = \{M(0)\}$ and $\mathrm{gfp}(T) = \emptyset$.

V. Some examples of infinite computations

We have given above an outline of the properties of *finite computations* of logic programs. The only "missing link" left was the "looping" part

$H_b - \mathrm{lfp}(T) \cup FF(P)$ of the Herbrand base, and some uncertainty concerning the greatest fixpoint of the transformation T.

What we are going to see now is that there are some logic program computations which are meaningful, at least intuitively, but which are ignored by the least fixpoint theory described above. The practical importance of these *infinite* computations will lead us into a revision of our semantic framework. The new tools we shall introduce will be another topology and a greatest fixpoint theorem (on the model-theoretic side), and the notion of fair derivation (on the computational side). The essential part of the least fixpoint semantics will be preserved by this revision, however.

Before going into the details of this revision, we shall first give two examples of such infinite computations. The first example will concern the Fibonacci numbers, and the second example the running square of an infinite stream of numbers.

V.1. Fibonacci numbers. Let us consider the infinite sequence $F = (F_n)$ of Fibonacci numbers: $F_0 = F_1 = 1$, and $F_{n+2} = F_{n+1} + F_n$. The infinite sequence F may be coded as an infinite list:

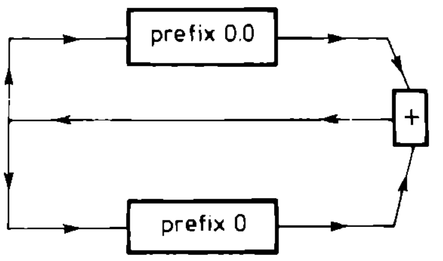
$$F = F_0 \cdot F_1 \cdot F_2 \cdot F_3 \cdot \ldots$$

which, by definition, satisfies the *fixpoint equation*

$$0 \cdot 0 \cdot F + 1 \cdot F = F$$

if addition of infinite lists is defined componentwise, and if $0 \cdot 0 \cdot F$ (resp. $1 \cdot F$) designates the infinite list F where the two atoms 0 and 0 (resp. the single atom 1) have been inserted in front of the list F.

Now the solution to the above fixpoint equation may be computed by using the following *data-flow program*:



This *data-flow program* may be coded into the following *logic program*:

EXAMPLE 2.

1. $\mathrm{sum}(0, x, x) \leftarrow$,
2. $\mathrm{sum}(s(x), y, s(z)) \leftarrow \mathrm{sum}(x, y, z)$,
3. $\mathrm{lsum}(a \cdot x, b \cdot y, c \cdot z) \leftarrow \mathrm{sum}(a, b, c)\ \&\ \mathrm{lsum}(x, y, z)$,

together with the goal clause:

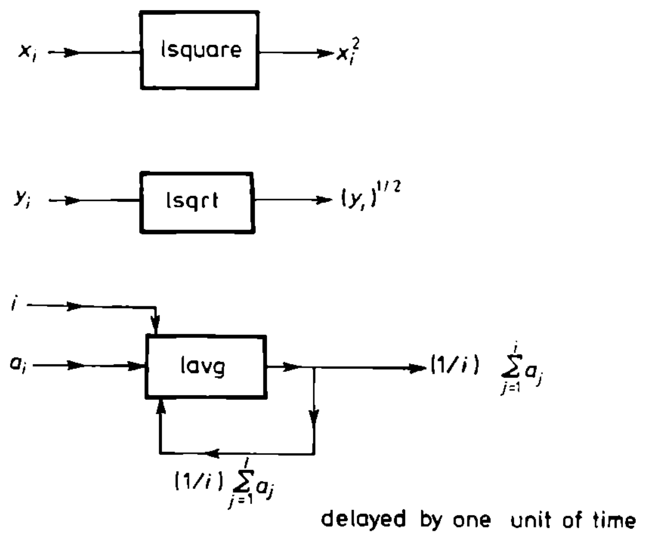
4. $\leftarrow \mathrm{lsum}(0 \cdot 0 \cdot f, 1 \cdot f, f)$.

Lines 1 and 2 of this program have already occurred above, they define the sum operation over natural numbers. Line 3 extends this sum operation, componentwise, to infinite lists. (To get the extension to finite lists, the clause lsum (nil, nil, nil) $\leftarrow$ must be added.) Finally, the query in line 4 simply negates the existence of a solution to the fixpoint equation satisfied by the infinite list of Fibonacci numbers.

Now if this logic program is executed, the reader may check that although the empty clause is never reached, it produces successive approximations to the list of all Fibonacci numbers: nil, $1 \cdot \text{nil}$, $1 \cdot 1 \cdot \text{nil}$, $1 \cdot 1 \cdot 2 \cdot \text{nil}$, etc. This may be visualized on the data-flow program as follows. Once the data circulating along this middle wire has been initialized to nil, and the data flow program started executing, an observer placed on the middle "wire" of this data-flow program may see these successive approximations flow by as time goes.

Thus we have here an example of an infinite logic program computation which intuitively makes sense, and computes a significant object (the Fibonacci numbers). It is, however, completely ignored by the least fixpoint semantics of the logic program 1–4 above. Indeed, for the program in Example 2, lfp(T) contains only the "graph" of the sum operation, and completely ignores infinite lists. Therefore, as far as the least fixpoint semantics is concerned, the programs in Examples 1 and 2 have exactly the same meaning.

V.2. Running square root of an infinite stream of numbers. This example is borrowed from [17]. Let (x_i) be an infinite input sequence of numbers, and suppose that we want to compute the infinite output sequence (y_i) defined as follows: y_i is the square root of the average of the squares of the input numbers $x_0, x_1, \ldots, x_i$ seen so far. This problem may be solved by using a data-flow program as follows. We first defined the three *elementary boxes*:


x_i
lsquare
x_i^2
y_i
lsqrt
$(y_i)^{1/2}$
i
a_i
lavg
$(1/i)\sum_{j=1}^{i} a_j$
$(1/i)\sum_{j=1}^{i} a_j$
delayed by one unit of time

These elementary boxes may be wired together in a sequential manner in order to get the circuit coding the data-flow program we are after.

The associated *logic program* is as follows:

EXAMPLE 3.

1. lsquare$(u\cdot x, v\cdot y) \leftarrow$ square(u, v)& lsquare(x, y),
2. lsqrt$(u\cdot x, v\cdot y) \leftarrow$ sqrt(u, v)& lsqrt(x, y),
3. lavg $(i, w\cdot y, u\cdot v\cdot x) \leftarrow$ newavg (i, u, v, w)& lavg $(s(i), y, v\cdot x)$ where newavg $(i, u, v, w) \Leftrightarrow v = u+(w-u)/(i+1)$,
4. result $(x, y) \leftarrow$ lsquare $(x, z_0\cdot z)$& lavg $(1, z_0\cdot z, z_0\cdot u)$& lsqrt $(z_0\cdot u, y)$.

We do not give details here as how squares (*square*) and square roots (*sqrt*) are computed. As the reader may check, the attempt to solve the goal clause $\leftarrow$ result (x, y), where x is a given infinite input sequence, will result in computing the "running" square root y of its input x, i.e., the nth number of the output will be the square root of the average of the squares of the first n input numbers $x_1, \ldots, x_n$.

Again, this infinite computations is completely invisible from the least fixpoint semantics point of view. Indeed, the least fixpoint in this case is equal to the least fixpoint of the subset of the program consisting only of the definitions of the predicates "sqrt", "square" and "newavg".

As a conclusion to this section, we see that there is an inadequacy between the mathematical definition of the meaning of a logic program, given by the least fixpoint of T, and the existence of these infinite logic program computations which, although useful, are completely ignored by this mathematical definition. Thus in order to get a better description of this computational reality, the theoretical definition of the meaning of a logic program has to be changed. This will be done in the next sections.

VI. Basic definitions: metric completion of H_u and H_b

A close examination of the situation described in the last two examples shows that in both cases we are computing with infinite objects (namely infinite lists), which by definition do not belong to the respective Herbrand universes. Thus it seems that we do not have enough elements in our Herbrand universes, and these infinite objects just "slip" through the "gaps" present in the universes. This situation is similar to the one we have with rational numbers: we cannot solve in Q the equation $x^2 = 2$ because there is a "gap" where the square root of 2 "should have been". This explains what we are going to do now: "fill in" these "gaps" by using some "glue". This is done by using the *metric completion* technique which allows us to go from the rational numbers to the real numbers, and allows us to solve equations like $x^2 = 2$. We recall that the *metric completion* of a given metric space X is just the set of all equivalence classes of *Cauchy sequences* of elements of X, where two Cauchy sequences are said to be *equivalent* iff the sequence obtained by merging them is also Cauchy [3].

To this end we define the following *distance between trees*:

$$d(t, t') = \begin{cases} 0 & \text{if } t = t', \\ 2^{-p}, & \text{where } p = \inf\{n: \alpha_n(t) \neq \alpha_n(t')\}, \text{ otherwise,} \end{cases}$$

where each element of H_u (H_b) is identified with the (expression) tree it defines. It can be shown that this distance is an ultrametric [8]. We now define $\boldsymbol{H}_u$ (resp. $\boldsymbol{H}_b$) as being the metric completion of H_u (resp. H_b). It turns out that, because the logic program P is *finite*, both $\boldsymbol{H}_u$ and $\boldsymbol{H}_b$ are *compact*. This is due to the fact that there are only finitely many open balls of any given radius, because of the definition of the distance in terms of tree depth, and of the finiteness of the sets F and R.

The subset transformation T defined in Section 3 of this paper may be canonically extended to a mapping $\boldsymbol{T}$ defined on the complete Herbrand base:

$$\boldsymbol{T}\colon P(\boldsymbol{H}_b) \to P(\boldsymbol{H}_b),$$

$$S \to \{A\vartheta\colon A\vartheta \leftarrow B_1\vartheta \& \ldots \& B_m\vartheta \text{ is a variable-free instance of clause of } P \text{ and } B_1\vartheta, \ldots, B_m\vartheta \in S\}.$$

This extended mapping $\boldsymbol{T}$ is also Scott-continuous, i.e., for any ascending sequence of subsets (S_i), we have $\boldsymbol{T}(\bigcup_i S_i) = \bigcup_i \boldsymbol{T}(S_i)$; this is because the reasons for Scott-continuity have not changed under the metric completion process. Thus $\boldsymbol{T}$ has a least fixpoint $\operatorname{lfp}(\boldsymbol{T}) = \bigcup_n \boldsymbol{T}^n(\emptyset)$.

Another property of $\boldsymbol{T}$ is that it maps closed subsets onto closed subsets. Also, if we take $C(\boldsymbol{H}_b) = \{S \subseteq \boldsymbol{H}_b\colon S \neq \emptyset \text{ closed}\}$, supplied with the *Hausdorff distance*, then as far as $\boldsymbol{T}$ is concerned, $C(\boldsymbol{H}_b)$ is a metric space with a "hole", i.e., $\boldsymbol{T}(C(\boldsymbol{H}_b)) = C(\boldsymbol{H}_b) \cup \{\emptyset\}$. An illustration of this situation is given by the following example. Let $S_n = \{p(s^n(0), s^{n+1}(0))\}$, and let $P = \{p(s(x), s(x)) \leftarrow p(x, x)\}$. Then, for every n, $\boldsymbol{T}(S_n) = \emptyset$, but $\boldsymbol{T}(\lim_n S_n) = \boldsymbol{T}(\{p(s^\omega, s^\omega)\}) = \{p(s^\omega, s^\omega)\}$.

Thus here we have a drastic departure from the properties of recursive program schemes [2], even though the theory may look the same. The deep reason for this situation is given by the fact that rewriting, used in recursive program schemes, is a continuous operation, whereas unification, used in logic programming, is not a continuous operation [11]. This is due to the fact that the *unifiability* function which takes two terms as arguments and answers *true* if they are unifiable, and *false* otherwise, is *discontinuous* if $\{$*true*, *false*$\}$ is supplied with the *discrete topology*. The unifiability function, however, is *continuous* if $\{true, false\}$ is considered as a Sierpiński space equipped with the lower Scott topology [11]. The *Sierpiński space* is defined as the two-element ordered set $\{true, false\}$ with $false \leqslant true$. Its *lower* Scott topology is the topology whose open sets are $\emptyset$, $\{false\}$, and $\{true, false\}$. As a side-remark, the Sierpiński space is the simplest (nontrivial) *continuous* lattice, and every continuous lattice can be obtained from it by retractions and cartesian products [15].

VII. Infinitary-tree semantics of logic programs, and their computational characterization

In the infinitary case, three fixpoints of T may be canonically associated with a logic program P: the least fixpoint, the least closed fixpoint, and the greatest fixpoint. Each one of them defined a possible meaning of the program. We shall link each such definition of the meaning of a logic program P first to Herbrand models, and then to computations. More precisely, we are going to show that in each case the fixpoint of T is some specific Herbrand model, which can be characterized computationally.

VII.1. Least fixpoint semantics. As we have noted earlier, the transformation T is Scott-continuous, thus by the Knaster-Tarski fixpoint theorem, it has $\bigcup_n T^n(\emptyset)$ as a least fixpoint. In the sum example we have $\mathrm{lfp}(T) = \mathrm{lfp}(T) \cup \{\mathrm{sum}(0, s^\omega, s^\omega)\}$.

Derivations may be generalized to the infinitary case in a straightforward manner. The only difficulty comes from the *existence* and *construction* of *most general unifiers*. Indeed, Robinson's algorithm applies only to *finitary* terms. A terminating unification algorithm for *rational terms* (i.e., terms with only finitely many distinct subterms) is given in [6], [7]. The determination of the largest class (or maximal classes) of infinitary terms for which there exists a terminating unification algorithm is still an open problem.

THEOREM 7.1. $\mathrm{lfp}(T) = \{a \in \boldsymbol{H}_b$: *a is the root of some successful derivation*$\}$.

VII.2. Least closed fixpoint semantics. When restricted to *closed* subsets of $\boldsymbol{H}_b$, the transformation T is generally *not* Scott-continuous, because the union of an increasing chain of closed sets may or may not be closed. However, T is monotone, and the Tarski theorem implies it has a least *closed* fixpoint we denote by $\mathrm{lcfp}(T)$. Let us define *continuous Herbrand interpretations* as being *closed* subsets of $\boldsymbol{H}_b$; then we have the following result [11]:

THEOREM 7.2 (Nait Abdallah). $\mathrm{lcfp}(T)$ *is the smallest continuous Herbrand model of P.*

A computational characterization of the least closed fixpoint of T is given by means of *infinitary proofs* using the following inference rule we call the *Cauchy rule* [11]:

$$\frac{\forall n\, P \vdash a_n,\ (a_n)\ \text{Cauchy sequence}}{P \vdash \lim_n a_n},$$

where a_n stands for any Cauchy sequence and $\lim_n a_n$ is its metric limit. We designate by $\boldsymbol{P}$ the set of definite clause of a logic program P augmented by the Cauchy rule.

THEOREM 7.3 (Nait Abdallah). $\mathrm{lcfp}(\boldsymbol{T}) = \{a \in \boldsymbol{H}_b\colon\ a \textit{ is } \boldsymbol{P}\text{-provable}\}$.

In the sum example, we have $\mathrm{lcfp}(\boldsymbol{T}) = \mathrm{lfp}(T) \cup \{\mathrm{sum}(s^\omega, u, s^\omega), \mathrm{sum}(u, s^\omega, s^\omega)\colon\ u \in \boldsymbol{H}_b\}$.

VII.3. Greatest fixpoint semantics. We may remark that each of the two infinitary-term semantics above has failed to give us the meaning of the infinite computations we examined in Section V of this paper. Thus the model-theoretic approach has not, at this point of our exposition, given us the meaning of these computations. We are now going back to the computational approach. In fact, we are going to see that each one of the infinite computations shown above is adequately described by the *greatest fixpoint semantics.*

An *infinite derivation* Δ is an infinite sequence

$$t_0, \delta_1, t_1, \delta_2, t_2, \ldots, \delta_n, t_n, \ldots$$

where each t_i is a goal, $\delta_i = \langle r_i, \lambda_i, \vartheta_i \rangle$ is a derivation step, and $t_{i+1} = \delta_{i+1}(t_i)$ for every $i \in N$. To each infinite derivation Δ from an *atom* t, we associate a *computed subset* (included in the complete Herbrand base $\boldsymbol{H}_b$):

$$[\Delta(t)] = \bigcap \{[\Delta|_j(t)]\colon\ j \in N\} = \bigcap \{[t\vartheta_1 \vartheta_2 \ldots \vartheta_j]\colon\ j \in N\},$$

where $\Delta|_j$ designates the finite derivation obtained from Δ by taking only the first j steps, and the notation $[t']$, where t' is a tree, stands for the set of all variable-free instance of t'. Since $\boldsymbol{H}_b$ is compact, this set is never empty.

Finally, we say that a derivation Δ from t is *k-fair*, for $k \in N$, if and only if intuitively each atom occurring in the root of the derivation is eventually replaced by some step δ of the derivation, and for each atom v introduced by δ, the subderivation of Δ starting from v is $(k-1)$-fair. Every derivation is 0-fair. Successful derivations are trivially fair.

THEOREM 7.4 (van Emden, Nait Abdallah [5]). *For every atom t and for every $k \in N$, we have*

$$[t] \cap \boldsymbol{T}^k(\boldsymbol{H}_b) = \bigcup \{[\Delta(t)]\colon\ \Delta\ \textit{k-fair derivation from } t\}.$$

This theorem implies the following two results. These two results basically characterize the greatest fixpoint of T in terms of fair derivations.

THEOREM 7.5 [10]. *$\bigcap\{T^n(H_b)$: $n \in N\} = \{a \in H_b$: a is the root of a fair derivation$\}$.*

THEOREM 7.6 (Greatest fixpoint theorem; Tiuryn, Nait Abdallah [9], [16]). *$\bigcap\{T^n(H_b)$: $n \in N\}$ is the greatest fixpoint of T.*

This last statement looks very much like the least fixpoint theorem we had in the finitary case, where $\bigcup\{T^n(H_b)$: $n \in N\}$ was the least fixpoint of T. This seems to indicate that there is a hidden Scott-continuity property. Indeed, this is the case [9], [16]:

THEOREM 7.7 (Tiuryn, Nait Abdallah). *For any sequence (S_n) of closed subsets, we have $T(\bigcap_n S_n) = \bigcap_n T(S_n)$.*

IX. Canonical partition of the complete Herbrand base

The above results may be summarized by the following diagram [10], [12]:

finitely failed atoms $H_b - \bigcap\{T^q(H_b)$: $q \in N\}$
results of fair derivations $\mathrm{gfp}(T) = \bigcap\{T^q(H_b)$: $q \in N\}$
atoms having an (infinitary) P-proof least closed fixpoint of T $\mathrm{lcfp}(T)$
results of successful derivations $\mathrm{lfp}(T) = \bigcup\{T^n(\emptyset)$: $n \in N\}$

Table 2. Canonical partition of the complete Herbrand base H_b

References

[1] K. R. Apt and M. H. van Emden, *Contributions to the theory of logic programming*, J. Assoc. Comput. Mach. 29 (3) (1982), 811–862.

[2] A. Arnold and M. Nivat, *The metric space of infinite trees. Algebraic and topological properties*, Fund. Inform. 3 (4) (1980), 445–476.

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

[4] M. H. van Emden and R. Kowalski, *The semantics of predicate logic as a programming language*, J. Assoc. Comput. Mach. 23 (4) (1976), 733–743.

[5] M. H. van Emden and M. A. Nait Abdallah, *Top-down semantics of fair derivations in logic programs*, J. Logic Programming 1 (1985), 67–75.

[6] Hopcroft and Karp, *An algorithm for testing the equivalence of finite automata*, TR 71-114, CS Department, Cornell University (1971).

[7] G. Huet, *Résolution d'équations dans des languages d'ordre* $1, 2, \ldots, \omega$, Thèse d'État, Université de Paris VII, 1976.

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

[9] M. A. Nait Abdallah, *Metric interpretation and greatest fixpoint semantics of logic programs*, University of Waterloo Report CS-82-19 (September 1983, revised November 1983); shorter version published in C. R. Acad. Sci. Paris, Ser. I, 300, 145–147.

[10] –, *On the interpretation of infinite computations in logic programming*, Lecture Notes in Comput. Sci. 172, Springer, 1984, 358–370.

[11] –, *On some topological properties of logic programs*, Lecture Notes in Comput Sci. 199, Springer, 1985, 310–319.

[12] –, *Fair derivations in logic programming: operational and greatest fixpoint semantics* Fund. Inform. 10 (1987), 247–308.

[13] J. A. Robinson, *A machine-oriented logic based on the resolution principle*, J. Assoc. Comput. Mach. 12 (1) (1965), 23–41.

[14] D. S. Scott, *Outline of a mathematical theory of computation*, 4th Annual Princeton Conference on Information Sciences and Systems, 1970, 169–176.

[15] –, *Continuous lattices*, Lecture Notes in Math. 274, Springer, 1972, 97–136.

[16] J. Tiuryn, Unpublished letter to M. H. van Emden, 1979.

[17] W. Wadge and E. Ashcroft, *LUCID, the dataflow programming language*, Academic Press, 1985.

Presented to the semester
Mathematical Problems in Computation Theory
September 16–December 14, 1985