ON THE FIXED-POINT SEMANTICS OF HORN CLAUSES WITH INFINITE TERMS

M.Falaschi, G.Levi, C.Palamidessi
Dipartimento di Informatica
Universita' di Pisa, Italy

1. INTRODUCTION

Infinite terms (streams) have been introduced in several PROLOG-like languages [2,3,4,8,10] in order to define parallel communicating processes. The resulting operational semantics is quite similar to Kahn-McQueen's model [5], characterized by agents which communicate through channels. Most of the above mentioned languages are annotated versions of PROLOG. Hence some of the most relevant features of PROLOG, such as the ability to define relations, get lost.

If infinite terms are added to pure PROLOG (i.e. Horn clauses), the definition of a 'good' fixed-point semantics is still an open problem. In [1] a greatest fixed-point construction is proposed. Such solution, however, is not satisfactory, because:

i) the greatest fixed-point semantics gives a non-empty denotation not only to nonterminating procedures which compute infinite terms, but also to 'bad' standard nonterminating programs;

ii) the construction is not always effective, i.e. there exist programs whose greatest fixed-point cannot be computed.

In this paper we propose two semantics based on a least fixed point construction. In the first semantics we only consider all the finite approximations of an infinite term, while the second semantics allows to handle infinite terms. The language we will consider is a many sorted version of PROLOG. Its syntax will be defined in the next section. It is worth noting that the sorting mechanism will allow us to distinguish finite and infinite terms.

2. SYNTAX AND DERIVATION RULE

The language alphabet is composed by:

1) A set S of identifiers for the representation of the sorts. A sort s is:

a) simple if s belongs to S. The set of simple sorts is partitioned into two disjoint classes, canonical and non-canonical sorts, to cope with finite and infinite data structures respectively.
b) functional if s belongs to S*--> S . If s has the form: $s_1 \times \ldots \times s_n$ --> s', and at least one of the s_i's is non-canonical, then s' is non-canonical too.
c) relational if s belongs to S .

2) A family C of sets of constant symbols indexed by simple sorts. If s is a non-canonical sort, then the set of constants of sort s contains the special symbol ω_s, which denotes an undefined (not yet evaluated) data structure.
3) A family D of sets of data constructor symbols indexed by functional sorts.
4) A family V of numerable sets of variable symbols indexed by simple sorts.
5) A family R of sets of predicate symbols indexed by relational sorts.

The language data structures are obtained by applying data constructors to variables and constants of suitable sorts. More precisely, a term of sort s is:
i) a constant symbol of sort s.
ii) a variable symbol of sort s.
iii) a data constructor application $d(t_1,\ldots,t_n)$ such that $t_1,\ldots,t_n$ are data terms of sorts $s_1,\ldots,s_n$ and d belongs to D and has sort $s_1 \times \ldots \times s_n$ --> s.
A term which contains at least one occurrence of an undefined constant symbol is called suspension and denotes a not completely evaluated data structure.
Because of the condition in 1.b), if one of the t_i's has a non-canonical sort (briefly is non-canonical), then also the term is non-canonical. In fact, the result of the application of a data constructor to its components (arguments) is a suspension if some of its components are suspensions.

The language basic construct is the atomic formula. An atomic formula is a predicate application $P(t_1,\ldots,t_n)$ such that $t_1,\ldots,t_n$ are data terms of sort $s_1,\ldots,s_n$ respectively, and P is a predicate symbol of sort $s_1 \times \ldots \times s_n$.

A set of atomic formulas can be interpreted as a collection of processes or agents [2,7] connected by channels. Each atomic formula denotes a process. There exists a channel connecting processes p_h and p_k, if there exists a variable symbol which occurs in the atomic formulas denoting p_h and p_k. The basic activity is message passing through channels and reconfiguration of the collection of processes. Informations can pass through a channel in both directions. This is not the case of the SCA model [7], as well as of the Kahn-McQueen model [5].

The dynamic behaviour of the collection of processes is specified by a set of clauses, which are expressions of the language defined as follows:

1) A definite clause is a formula of the form:

   A <-- B_1, ..., B_n

   where A and the B_i's are atomic formulas. If n=0 the clause is called 'unit clause' and is denoted as follows:

   A <-- λ

   All the variables occurring in a clause are viewed as universally quantified.

2) A negative clause (goal statement) is a formula of the form:

   <-- A_1, ..., A_m

   where the A_i's are atomic formulas. If m=0 it is a null clause denoted by

   <-- λ (or □)

From a logical viewpoint, the symbol ',' denotes the logical connective AND, the symbol '<--' denotes the logical implication, and λ is the neutral element with respect to the operator ',', that is <-- A,λ = <-- A

The notion of derivation of a goal statement from a given goal statement and a program is essentially the same defined for PROLOG [6], and is based on resolution [9]. The only trivial difference has to do with sort checking.

The relation

$$G \;|\overset{\theta}{\underset{W}{--}}>\; G'$$

denotes that the goal statement <-- G' is derivable from the goal statement <-- G and the program W, with the substitution θ, which is the composition of all the substitutions used in the elementary derivations.

If, for some θ, the relation

$$G \;|\overset{\theta}{\underset{W}{--}}>\; \lambda$$

holds, then <-- G is refutable in W.

Our interpretation of goal statements and clauses is exactly the same given by Kowalski [6] for PROLOG. However, we think of a goal statement as denoting a collection of processes. The derivation of a new goal statement corresponds to a reconfiguration of the collection. Each elementary variable binding in a unification can be seen as a message passing from a producer to a consumer. Our interpretation is motivated by the fact that we view processes as non terminating procedures which produce (or consume) infinite data structures. Such procedures have an empty denotation in PROLOG, both from the operational and the fixed-point semantics viewpoint.

3. OPERATIONAL SEMANTICS

In standard Horn Clause Logic the concept of computation of a goal statement is essentially based on the refutation of that goal statement, (i.e. the derivation of the null clause), and therefore on the concept of termination. In other words, the result of a computation of a goal statement (i.e. its operational semantics) is the relation established, for each predicate in the goal, by the substitutions determined in all the possible refutations [6].

This definition of operational semantics results inadequate to describe processes which handle infinite terms (streams). Consider, for example, the following program:

W = {list(x,x.L) <-- list(s(x),L)}

where the sort of x is 'naturals' (canonical sort), the sort of L is 'streams of natural' (non canonical sort), '.' denotes the stream of naturals constructor, and 's' denotes the successor constructor on naturals (for the sake of simplicity we will use 1 instead of s(0), 2 instead of s(s(0)), etc.).

Since the goal statement <-- list(0,L) has no refutations in W, the denotation of the predicate list given by the standard operational semantics is an empty relation. In spite of this, a derivation of list(0,L) produces, step by step, the substitutions:

L = 0.L
L = 0.1.L
L = 0.1.2.L etc...

It is easy to see that an infinite computation of this goal statement will lead L to be instanced to the infinite list of natural numbers. In general every process which produces infinite terms has the same problems with respect to its semantics definition, since its computation necessarily does not terminate.

The solution we propose is based on the introduction for each predicate symbol P which is non-canonical (i.e. which handles infinite terms), of a <u>terminal clause</u> (unit clause) defined as follows:
If P has sort $s_1 \times \ldots \times s_n$, then the terminal clause has the form $P(t_1,\ldots,t_n)$ <-- , where each t_i is:
- a variable of sort s_i, if s_i is canonical
- the undefined constant symbol ω_{s_i}, if s_i is non-canonical.

The terminal clause is added only if there exists no unit clause, in the program, for which there is a superposition. This condition is necessary because it must not be possible to introduce new solutions by adding a terminal clause. The new terminal clause must only allow termination.

Note that if there exists a terminal clause, for which there exists a superposition with the new one, then it contains some non-canonical terms that can be substituted with ω. For this reason the termination is guaranteed in this case.

In our example the terminal clause is

list(n,ω) <-- λ

This clause allows the goal statement <-- list(0,L) to have a refutation. The values that it computes for L are of the form: ω, 0.ω, 0.1.ω, 0.1.2.ω, etc...

The symbol ω, in this example, looks like the empty-list constant, and the values for L look like standard finite lists. Their pragmatics however is quite different, since the programmers can think in terms of infinite lists and not be worried about artificial terminal cases, which can be inserted systematically by the interpreter.
The introduction of the terminal clause is similar to the termination rule for infinite data productors proposed in [7]. In that case a process producing a (potentially) infinite data structure terminates when all the processes which consume that data structure have terminated (lazy evaluation). We obtain the same behaviour by exploiting the non-determinism of the language. A process which produces a (potentially) infinite stream, at each stream approximation can be reduced to λ. However, if there exist consuming processes, the process has an alternative reduction which produces a refinement of the stream.

The operational semantics is defined as follows:

If W is a set of clauses, and P is a predicate symbol of sort $s_1 \times \dots \times s_n$, then the operational semantics of P in W is:

$$\mathbf{D}_0(P,W)=\{(t_1,\dots,t_n) \mid t_i \text{ has sort } s_i, i=1,\dots,n \text{ and } P(t_1,\dots,t_n) \mid\!\frac{\theta}{W'}\, \lambda\}$$

where W′ is the union of the program W and of all of its terminal clauses, added accordingly to the rule above described.

EXAMPLE 1)

list(n,n.L) <-- list(s(n),L)

P(s(n),k.L,y) <-- P(n,L,m) , prod(k,m,y)
P(0,L,1) <-- λ

Assume <-- prod(k,m,y) be refutable iff y results to be the product of m and k.
list(n,L) is the process which produces the stream L of all the natural numbers starting from n.

P(n,L,m) defines the relation 'm is the product of the first n numbers in the stream L'.
Then, consider the program:

$$W' = \begin{cases} 1)\ fact(n,m) \leftarrow list(1,L)\ ,\ P(n,L,m) \\ 2)\ P(s(n),k.L,y) \leftarrow P(n,L,m)\ ,\ prod(k,m,y) \\ 3)\ P(0,L,1) \leftarrow \lambda \\ 4)\ list(n,n.L) \leftarrow list(s(n),L) \\ 5)\ list(n,\omega) \leftarrow \lambda \qquad \text{(terminal clause)} \end{cases}$$

Note that 5 is the only terminal clause, since the clause $P(x,\omega,y) \leftarrow \lambda$ will not satisfy our condition.

fact(n,m) defines the relation 'm is the factorial of n'.
We will now give an example of computation. For the sake of simplicity, the second clause will be rewritten in the form:

$$P(s(n),k.L,k*m) \leftarrow P(n,L,m)$$

where the symbol '*' is interpreted as the product operator on natural numbers.

Initial goal statement:
$\leftarrow fact(2,x)$

by clause 1),and the substitution x=m:
$\leftarrow list(1,L)\ ,\ P(2,L,m)$

by clause 2, and the substitution $L=k.L_1$, $m=k*m_1$:
$\leftarrow list(1,k.L_1)\ ,\ P(1,L_1,m_1)$

by clause 2, and the substitution $L_1=k_1.L_2$, $m_1=k_1*m_2$:
$\leftarrow list(1,k.k_1.L_2)\ ,\ P(0,L_2,m_2)$

by clause 3, and the substitution $m_2=1$:
$\leftarrow list(1,k.k_1.L_2)$

by clause 4, and the substitution k=1:
$\leftarrow list(2,k_1.L_2)$

by clause 4, and the substitution $k_1=2$:
$\leftarrow list(3,L_2)$

by clause 5, and the substitution $L_2=\omega$:
$\leftarrow \lambda$

The resulting substitution for x is:
$x=m=k*m_1=k*k_1*m_2=k*k_1=k_1=2$

The resulting substitution for L is:
$L=k.L_1=k.L_1.L_2=1.2.\omega$

Note that, to have a refutation, at least two elements of the list L have to be computed.

4. FIXED POINT SEMANTICS: FINITE APPROXIMATIONS

The fixed point semantics for a program W is defined as a model of the set of clauses W U {terminal clauses}, obtained as the least fixed point of a transformation which is defined on the set of the interpretations of W [1,10,11].

The interpretations of W are defined over an abstract domain U, which is a family of sets U_s, each set being indexed by a sort s occurring in W.
Each U_s is defined as follows:

1) All the constant symbols of sort s, occurring in W, are in U_s (note that if s is a non-canonical sort, also ω_s is a constant symbol of sort s and then also ω_s belongs to U_s).
2) For each data constructor symbol of sort $s_1 \times \ldots \times s_n \rightarrow s$, U_s contains all the terms $d(t_1,\ldots,t_n)$ such that $t_1,\ldots,t_n$ belongs to $U_{s_1},\ldots,U_{s_n}$, respectively.

Note that U contains the standard many sorted Herbrand Universe as a proper subset, i.e. the set of all the ground terms in which none of the ω_s occurs. In addition U contains suspensions, i.e. non completely evaluated data, where both undefined and standard constant symbols occur. Finally, U contains also the fully undefined terms, i.e. the terms ω_s.

The Herbrand Base $\mathcal{B}$ of W is the set of all the ground atomic formulas: for each predicate P occurring in W, of sort $s_1 \times \ldots \times s_n$, and for each n-tuple of terms $t_1,\ldots,t_n$ belonging to $U_{s_1},\ldots,U_{s_n}$ respectively, $P(t_1,\ldots,t_n)$ belongs to $\mathcal{B}$.

A Herbrand Interpretation I of W is any subset of $\mathcal{B}$ containing λ.

The set $\mathfrak{I}$ of all the Herbrand Interpretations of W is partially ordered by the relation $\subseteq$ (set inclusion). As is the case for standard Horn clauses, $(\mathfrak{I},\subseteq)$ is a complete lattice, i.e. for every possibly non finite subset $\mathcal{L}$ of $\mathfrak{I}$, there exists lub($\mathcal{L}$) and glb($\mathcal{L}$).

It is possible to associate, to any program W, a transformation T on the domain of interpretations, defined in the following way:

$$T(I)=\{A \mid A \leftarrow B_1,\ldots,B_n \text{ is a ground instance of a clause of } W' \text{ and } B_1,\ldots,B_n \in I\} \cup \{\lambda\}$$

where W' is the union of the set W and of the terminal clauses for W.

It is well-known that the transformation T is monotonic and continuous [6].

Since T is monotonic, there exists:

$$I_F = \min\{I \mid I=T(I)\}$$

Moreover, since T is continuous:

$$I_F = \bigcup_{k\geq 0} T^k(\{\lambda\})$$

The fixed point semantics of a predicate P, of sort $s \times \ldots \times s_n$, in a program W is defined as follows:

$$D_F(P,W) = \{(t_1,\ldots,t_n) \mid t_1 \in U_{S_1},\ldots, t_n \in U_{S_n}, P(t_1,\ldots,t_n) \in I\}$$

The equivalence of the operational and fixed-point semantics comes directly from the similar result for PROLOG.

5. FIXED-POINT SEMANTICS: INFINITE TERMS.

Now we want to define an alternative fixed-point semantics, which reflects the idea that non-canonical data, containing the symbols ω_S, are suspensions, that is partial approximations of infinite terms.

A term containing occurrences of the symbol ω_S cannot be transformed into an infinite term containing no occurrences of ω_S, because it would be necessary an infinite number of derivations. However it is possible to compare two suspensions to establish which is a better approximation.

Consider, for example, the process P(n,L) which produces the stream of all the odd numbers starting from n, if n is odd, and the stream of the even numbers starting from n, if n is even. Such process is defined by the clause:

1. P(n,n.L) <-- P(s(s(n)),L)

while the terminal clause is:

2. $P(n,\omega)$ <-- λ

One of the streams produced by the process P, starting from 0, is $L_1 = 0.2.\omega$, obtained by applying clause 1 twice and clause 2 once.
Another stream is $L_2 = 0.2.4.\omega$, obtained by applying clause 1 three times, and clause 2 once.
L_1 is a better approximation than L_2 of the stream which could be obtained starting from 0 and applying clause 1 forever:

0.2.4.6. ...

Clearly L_1 cannot be compared to any of the streams obtainable, for example, starting from 1 ($1.\omega$,$1.3.\omega$, etc.).

It is then necessary to define a partial ordering $\leq$ on the elements of A (ground terms), which corresponds to the concept of 'better approximation'.

i) For any constant symbol c of sort s, $c_s \leq c_s$ and, if s is non-canonical, $\omega_s \leq c_s$.
ii) For any constructor symbol of sort $s_1 \times \ldots \times s_n \rightarrow s$:
   a) if $t_i = \omega_{s_i}$, i=1,...,m, then $d(t_1,\ldots,t_n) = \omega_s$
   b) if $t_i \leq t_i'$, i=1,...,m, then $d(t_1,\ldots,t_n) \leq d(t_1',\ldots,t_n')$

A similar partial ordering is defined on the Herbrand Base $\mathcal{B}$, as follows:

For any predicate symbol P of sort $s_1 \times \ldots \times s_m$, and for any $t_1,\ldots, t_m, t_1',\ldots,t_m'$ of sorts $s_1,\ldots,s_m$:
if $t_i \leq t_i'$ i=1,...,m, then $P(t_1,\ldots,t_m) \leq P(t_1',\ldots,t_m')$.

Furthermore, it is necessary to introduce in the universe U all the infinite terms which are limits of monotonic sequences of terms. Similarly, it is necessary to introduce in the base $\mathcal{B}$ all the atomic formulas which contain infinite terms and which are limits of monotonic sequences of atomic formulas.

An interpretation of W is any subset of $\mathcal{B}$ which contains λ and which does not contain any pair formulas A and A', such that $A < A'$.

Obviously, the interpretation containing atomic formulas in which there occur infinite terms can be regarded as limits of monotonic sequences of interpretations without infinite terms.

Let ρ be a function which transforms subsets of B (containing λ) into interpretations. It is defined as follows: if s is a subset of B then

$$\rho(S) = S - \{A \mid A \in S,\ \exists A' \in S\ \ A < A'\}$$

In other words ρ eliminates all those atomic formulas for which there exists in S a better approximation.

The set of the interpretations of W is partially ordered by the relation $<$ defined as follows: if I,J belongs to :

$$I < J \text{ iff } \forall A \in I\ \exists A' \in J\ \ A < A'$$

or, equivalently:

$$I < J \text{ iff } I \in \sigma(J)$$

where σ is defined as follows:

$$\sigma(I) = \{A \mid \exists A' \in I \quad A < A'\} \cup \{\lambda\}$$

Note that, if I is an interpretation: $\rho(\sigma(I))=I$

The set $\mathcal{J}$ of the interpretations is a complete lattice with respect to $<$, and it holds, if $\mathcal{L}$ is a subset of $\mathcal{J}$:

$$glb(\mathcal{L}) = \rho(\cup\, \sigma(\mathcal{L}))$$
$$lub(\mathcal{L}) = glb(\mathcal{L}')$$

where $\mathcal{L}' = \{I' \mid \forall I \in \mathcal{L} \quad I < I'\}$

Note that $\mathcal{L}'$ is never empty, because it contains at least $\rho(B)$. In particular, if $\mathcal{L}$ is finite:

$$lub(\mathcal{L}) = \rho(\cup\, \sigma(\mathcal{L}))$$

The transformation T' associated to a program W is defined in the following way:

$$T'(I) = \rho(\{A \mid A \leftarrow B_1, \ldots, B_n \text{ is a ground instance of a clause of } W', \text{ and } B_1, \ldots, B_n \in \sigma(I)\} \cup \{\lambda\})$$

where W' is the union of W and of the terminal clauses of W.

$\sigma(I)$ occurs in the definition of I because, if a certain approximation of a data structure is computed, then also any less defined approximation of such a data structure must be considered as computed.

It can easily be proved that T' is monotonic and continuous, hence there exists the least fixed-point I_f' of T' and:

$$I_f' = \bigcup_{k\geq 0} T'^k(\{\lambda\})$$

The second fixed-point semantics is defined analogously to the first:

$$D_{f'}(P,W)=\{(t_1, \ldots, t_n) \mid t_1 \in U_{S_1}, \ldots, t_n \in U_{S_n}, P(t_1, \ldots, t_n) \in \sigma(I_f')\}$$

It is worth noting that in the previous semantics, the lub of the chain $T^k(\{\lambda\})$ contains only finite approximations (suspensions), while, for this semantics, the lub of $T'^k(\{\lambda\})$ can contain also infinite terms.

BIBLIOGRAPHY

1. Apt, K.R. and M.H. van Emden. "Contributions to the theory of logic programming". J. ACM 29 (1982).

2. Bellia,M., Dameri,E., Degano,P., Levi,G. and M.Martelli. 'Applicative Communicating Processes in First Order Logic'. Symposium on Programming. Lecture Notes in Computer Science 137 (Springer Verlag, 1982) 1-14.

3. Clark,K.L. and S.Gregory. 'A relational language for parallel programming'. Proc. of Functional Programming Languages and Computer Architecture Conf. (1981) 171-178.

4. Hannson,A., Haridi,S. and S.A.Tärnlund. 'Properties of a Logic Programming Language'. Logic Programming, Clark and Tarnlund Eds. (Academic Press, 1982) 267-280.

5. Kahn,G. and D.B.MacQueen. 'Coroutines and networks of parallel processes'. Information Processing 77, North Holland (1977), 993-998.

6. Kowalski,R. 'Predicate logic as a programming language'. Proc.IFIP Cong. 1974, North-Holland Pub. Co., Amsterdam, 1974, pp.569-574.

7. Monteiro,L. 'An extension to Horn Clause Logic allowing the definition of concurrent processes'. Proc.I.C.F.P.C. (Eds: J.Diaz, I.Ramos), LNCS 107, Springer-Verlag 1981.

8. Pereira,L.M. 'A PROLOG demand-driven computation interpreter'. Logic Programming Newsletter 4 (1982), 6-7.

9. Robinson,J.A. 'A machine-oriented logic based on the resolution principle'. J.ACM 12 (1965), pp.23-41.

10. Van Emden,M.H. and G.J. de Lucena. 'Predicate logic as a language for parallel programming'. Logic Programming, Clark and Tärnlund Eds. (Academic Press, 1982) 189-198.

11. Van Emden,M.H., Kowalski,R. 'The semantics of predicate logic as a programming language'. J.ACM vol.23 (1976) n.4, pp.733-742.