Infinite Derivations as Failures

Andrea Corradi and Federico Frassetto

DIBRIS, Università di Genova, Italy
name.*surname*`@dibris.unige.it`

Abstract. When operating on cyclic data, programmers have to take care in assuring that their programs will terminate; in our opinion, this is a task for the interpreter. We present a Prolog meta-interpreter that checks for the presence of cyclic computations at runtime and returns a failure if this is the case, thus allowing inductive predicates to properly deal with cyclic terms.

Keywords: Logic Programming, Non-Termination, Inductive Semantics

1 Introduction

Sometimes, non-termination is the desired behavior of a program; most of the time, it is not. Nevertheless, in a programming language in which all programs terminate, there are always-terminating programs that cannot be written in it [11]. Non-termination is a necessary evil, a powerful feature that we *need*, but we do not actually *want* most of the time.

No work has been done to apply inductive semantics to the complete Herbrand model of Coinductive Logic Programming [8]. With the usual operational semantics, structural-recursive inductive predicates diverge when operating on infinite terms. Since infinite derivations do not belong to the inductive semantics [3], we propose a new operational semantics where infinite regular derivations of inductive predicate fail. While this implies a performance overhead, sometimes it is what a programmer would have done by hand, but faster and less error-prone.

In section 2, we illustrate briefly the notions of (co-)inductive model and give the declarative and operational semantics of co-logic programming. In section 3, we present our contribution, both as operational semantics and as Prolog code, and show some examples of use. In section 4, we compare our meta-interpreter with a standard Prolog interpreter. In section 5, we compare our work with the state of the art, draw conclusions and state possible future work.

2 About Inductive and Coinductive Semantics

Let the Herbrand universe $\mathcal{H}$ be a set of terms. We denote a *clause* as $A \leftarrow B_1, \dots, B_n$ where $A \in \mathcal{H}$ and $\forall\, i \in \{1 \dots n\}\ .\ B_i \in \mathcal{H}$. A *logic program* P is a couple (F, C), where F is a set of predicate symbols and C is a set of clauses. The *Herbrand base of* P, written $\mathcal{H}_P$, is the set of all the ground terms of the form $p(\bar{t})$, where $p \in F$ and $\bar{t}$ is a tuple of terms. An interpretation of P is an element of $\mathcal{P}(\mathcal{H}_P)$, where $\mathcal{P}$ denotes the power-set constructor. A logic program P induces a *one-step-inference function* $\mathcal{I}_P$.

$$\begin{aligned} \mathcal{I}_P \colon \mathcal{P}(\mathcal{H}_P) &\to \mathcal{P}(\mathcal{H}_P) \\ S &\mapsto \{A \in \mathcal{H}_P \mid (A \leftarrow B_1, \ldots, B_n) \in P \land \forall i \in \{1 \ldots n\} \,.\, B_i \in S\} \end{aligned}$$

A *model* of a logic program P is a fix-point of $\mathcal{I}_P$, that is, an interpretation S such that $\mathcal{I}_P(S) = S$.

Consider the logic program $\{p(1,2) \leftarrow \texttt{true}\} \cup \{p(X,X) \leftarrow p(X,X) \mid X \in \mathcal{H}\}$: written below in Prolog syntax.

```
p(1,2).
p(X,X) :- p(X,X).
```

both $\{p(1,2)\}$ and $\{p(1,2)\} \cup \{p(X,X) \in \mathcal{H}_{\mathcal{P}} \mid X \in \mathcal{H}_{\mathcal{P}}\}$ are models. Due to the Knaster–Tarski theorem [10], any logic program has a smallest model, called *inductive*, and a biggest one, called *coinductive*.

The usual Logic Programming (LP) semantics is inductive, therefore $\mathcal{H}_P$ contains only atoms inferred by finite derivations [3]; using the coinductive interpretation, we obtain the complete Herbrand base that contains also atoms inferred by infinite derivations. Simon et al. [7,8] have extended LP to co-Logic Programming (co-LP), which allows the programmer to choose between inductive and coinductive semantics for each predicate. The Prolog interpreters SWI-Prolog and YAP support co-LP.

We show the co-LP operational semantics as given by Ancona and Dovier[1] [2]. An *hypothetical goal* G is a couple $\langle E \mathrel{\square} L \rangle$, where E is a set of equations and L is a list of pairs (A, S), where A is an atom and S is a set of atoms that represents the ancestors of A in the call stack. Given a program $\mathcal{P}$ and two hypothetical goals $G = \langle E \mathrel{\square} (p(s_1, \ldots, s_n), S_1), (A_2, S_2), \ldots, (A_u, S_u) \rangle$ and G', we define the rewriting rule $G \vdash_{\mathrm{co}} G'$ by cases as follows:

1. if p is a coinductive predicate and there is an atom $p(t_1, \ldots, t_n)$ in S_1 such that $E' = E \cup \{s_1 = t_1, \ldots, s_n = t_n\}$ is solvable, then $G' = \langle E' \mathrel{\square} (A_2, S_2), \ldots, (A_u, S_u) \rangle$;
2. if there is a clause $p(t_1, \ldots, t_n) \leftarrow B_1, \ldots, B_m$ that is a renaming of a clause in $\mathcal{P}$ with fresh variables and $E' = E \cup \{s_1 = t_1, \ldots, s_n = t_n\}$ is solvable, then $G' = \langle E' \mathrel{\square} (B_1, S'), \ldots, (B_m, S'), (A_2, S_2), \ldots, (A_u, S_u) \rangle$ where $S' = S_1 \cup \{p(s_1, \ldots, s_n)\}$.

3 Finite failure

Let us consider the problem of checking whether an element appears in a list.

```
member(E,[E|_]).
member(E,[_|L]) :- member(E,L).
```

[1] The original semantics allows expanding any subgoal. For simplicity, our semantics always expands the first.

This naïve Prolog definition does not terminate if the second parameter is a cyclic list not containing `E`. This is not the desired behavior: we would like `member` to either succeed or fail in a finite amount of time. Simply applying coinduction to `member` leads to an erroneous semantics: for example, `L=[1|L], member(2,L)` succeeds even if `L` does not contain `2`, because the second call unifies with the first.

The definition to let it work on infinite lists leads to a more convoluted predicate `member2` that relies on the extra-logical cut operator `!`.

```
member2(E,L) :- member2([],E,L).
member2(H,E,L) :- member(L,H), !, fail.
member2(_,E,[E|_]).
member2(H,E,L) :- L=[_|T], member2([L|H],E,T).
```

The predicate we are trying to define is inherently inductive [7]. Our aim is to allow inductive predicates to work correctly on the complete Herbrand base. Since infinite derivations are not computable, the resolution procedure fails when it finds one.

3.1 Operational semantics

We give the operational semantics where the inductive predicates fail when the derivation diverges. We define the rewriting rule $\Vdash_{\overline{\text{co}}}$ in a similar way to rule $\vdash_{\overline{\text{co}}}$ in section 2, except for the second point:

2. if p is not inductive or if there is not an atom $p(t_1, \ldots, t_n)$ in S_1 such that $E' = E \cup \{s_1 = t_1, \ldots, s_n = t_n\}$ is solvable, use the second point of the definition of $\vdash_{\overline{\text{co}}}$.

In the second point if the p is coinductive we can behaves as $\vdash_{\overline{\text{co}}}$; if p is inductive we can proceed with the resolution only if it is not already in the call stack, $\Vdash_{\overline{\text{co}}}$ behaves as $\vdash_{\overline{\text{co}}}$. If it is in the call stack than atom could not be resolved and the resolution will fails.

3.2 Meta-interpreter

The implementation of a meta-interpreter follows the implementation of [1,7,8] but it returns a failure when it finds a cycle during the resolution inductive predicate. It is sound and complete with respect to $\Vdash_{\overline{\text{co}}}$.

```
cosld(G) :- solve([],G).
solve(H,(G1,G2)) :- !, solve(H,G1), solve(H,G2).
solve(_,A) :- built_in(A), !, A.
solve(H,A) :- member(A,H), !, coinductive(A).
solve(H,A) :- clause(A,As), solve([A|H],As).
coinductive(1).
```

The first argument of `solve` is the list of already processed atoms, used to avoid infinite computation; the second is the goal to resolve. The predicate `solve` has four clauses:

1. resolution distributes on conjunction as usual, with the same H in both calls, because it depends only on the ancestors;
2. resolution for built_in predicates is the default one;
3. if the atom A is in the hypotheses, the resolution succeeds if A is coinductive and fails otherwise;
4. if none of the above applies, the resolution proceeds normally.

Here we can avoid keeping the coinductive hypotheses for every atom in the goals, because we exploit the Prolog call stack to have the same result.

To mark a predicate p as coinductive, the source file must contain the fact coinductive(p(_,...,_)).. The declaration coinductive(1) assures that the coinductive predicate is defined even if there are no coinductive predicates.

This meta-interpreter keeps track of every non-built_in atom. We can improve efficiency by requiring to explicitly mark the inductive predicates and using the standard resolution procedure for the unmarked ones.

3.3 More Examples

Infinite trees. Let us represent a tree as t(E,Ts), where E is the element of the node and Ts is a finite list of sub-trees.

The predicate member_tree(E,T) searches a tree T for a node with element E.

```
member_tree(E,t(E, _)).
member_tree(E,t(_,Ts)) :- member(T,Ts), member_tree(E,T).
```

When using a standard interpreter, similarly to member and infinite lists, the predicate member_tree is not guaranteed to terminate with an infinite tree; for example, T1=t(1,[T1,T2]), T2=t(2,[T2,T3]), T3=t(3,[T3]), member_tree(3,T1) loops forever. Using our meta-interpreter, the goal succeeds correctly: during the resolution of the first recursive call member_tree(3,T), the call stack contains the atom member_tree(3,T1); T unifies with T1, so the goal fails; then, backtracking takes place, T unifies with T2 and the resolution continues in similar way until reaching member_tree(3,T3).

Graphs. Finding a path between two nodes is a common operation on graphs. We encode the previous tree as a graph represented by an adjacency list, obtaining [1-[1,2],2-[2,3],3-[1,3]].

The predicate path(N1,N2,G,P) searches for a path P from the node N1 to the node N2 in the graph G.

```
path(N1,N1,G,[N1]) :- neighbors(N1,G,_).
path(N1,N2,G,[N1|P]) :- neighbors(N1,G,Ns), member(N3,Ns),
                        path(N3,N2,G,P).
neighbors(N, [N-Ns|_], Ns).
neighbors(N1, [N2-_|G], Ns) :- N1 \= N2, neighbors(N1, G, Ns).
```

4 Comparison with ISO Prolog

Our meta-interpreter and a standard Prolog interpreter are not complete with respect to the inductive interpretation and not comparable to each other. Let us look at the following predicate.

```
p(3).
p(A) :- p(B).
```

In the inductive interpretation of the predicate `p`, `p(`x`)` holds for any x. This is the behavior of the standard interpreter; but, in our meta-interpreter, `p(`x`)` fails for any $x \neq 3$, because `A` and `B` always unify.

Consider the predicate `member`, but with the order of the clauses reversed.

```
member(E,[_|L]) :- member(L).
member(E,[E|_]).
```

In this case, the query `L=[1|L], member(1,L)` should succeed, because $(\mathtt{1},\mathtt{L})$ is in the inductive interpretation. The query succeeds with our meta-interpreter, but does not terminate with a standard Prolog interpreter.

5 Conclusion

In this work we show an operational semantics that allow inductive predicates to behave correctly on the complete Herbrand base; moreover we show how it is possible with a simple meta-interpreter to have inductive predicates that fail when they have a infinite, but rational, proof.

A meta-interpreter by Ancona [1] executes a user-specified predicate when it finds a cycle: it can simulate ours by executing `fail`. Moura obtained similar results [5]. This system is more expressive, but the relation with the standard Prolog semantics is not clear.

Tabling [6,9,4] uses a table to store the sub-goals encountered during the evaluation and their answers. When it finds the same sub-goal again, it uses the information from the table; this allows to increase performance and ensure termination. In tabling, when a goal is found in the table but its answer is not yet available, instead of failing as we do, it *suspends* the current evaluation and try another clause. When the resolution of this second clause finish providing an answer it *resume* the first one and continue its evaluations using the the answer. We do not have such behavior, and in this respect, our semantics is similar to the classical Prolog one.

As future work, we plan to prove soundness with respect to the inductive semantics and to investigate whether completeness is attainable. Unification is not ideal for checking whether the resolution procedure is in a loop, because it succeeds too often. We are looking for a relation that better suits our needs.

References

1. Ancona, D.: Regular corecursion in Prolog. Computer Languages, Systems and Structures 39(4), 142–162 (Dec 2013)

2. Ancona, D., Dovier, A.: A theoretical perspective of coinductive logic programming. Tech. rep., University of Genova and University of Udine (2015)
3. Leroy, X., Grall, H.: Coinductive big-step operational semantics. Information and Computation 207(2), 284–304 (Feb 2009)
4. Mantadelis, T., Rocha, R., Moura, P.: Tabling, rational terms, and coinduction finally together! Theory and Practice of Logic Programming 14(Special Issue 4-5), 429–443 (Jul 2014)
5. Moura, P.: A portable and efficient implementation of coinductive logic programming. In: Sagonas, K. (ed.) Pratical Aspects of Declarative Languages. Lecture Notes in Computer Science, vol. 7752, pp. 77–92. Springer (Jan 2013)
6. Rocha, R., Silva, F., Santos Costa, V.: On applying or-parallelism and tabling to logic programs. Theory and Practice of Logic Programming 5(1-2), 161–205 (Mar 2005)
7. Simon, L., Mallya, A., Bansal, A., Gupta, G.: Co-logic programming: Extending logic programming with coinduction. Lecture Notes in Computer Science, vol. 4596, pp. 472–483. Springer (2007)
8. Simon, L.E.: Extending logic programming with coinduction. Ph.D. thesis, University of Texas at Dallas (Aug 2006)
9. Swift, T., Warren, D.S.: XSB: Extending Prolog with tabled logic programming. Theory and Practice of Logic Programming 12(Special Issue 1-2), 157–187 (Jan 2012)
10. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics 5(2), 285–309 (1955)
11. Turner, D.A.: Total functional programming. Journal of Universal Computer Science 10(7), 751–768 (Jul 2004)