Extending Coinductive Logic Programming with Co-Facts

Davide Ancona Francesco Dagnino Elena Zucca
DIBRIS, University of Genova
{davide.ancona,elena.zucca}@unige.it, fra.dagn@gmail.com

We introduce a generalized logic programming paradigm where programs, consisting of facts and rules with the usual syntax, can be enriched by *co-facts*, which syntactically resemble facts but have a special meaning. As in coinductive logic programming, interpretations are subsets of the complete Herbrand basis, including infinite terms. However, the intended meaning (declarative semantics) of a program is a fixed point which is not necessarily the least, nor the greatest one, but is determined by co-facts. In this way, it is possible to express predicates on non well-founded structures, such as infinite lists and graphs, for which the coinductive interpretation would be not precise enough. Moreover, this paradigm nicely subsumes standard (inductive) and coinductive logic programming, since both can be expressed by a particular choice of co-facts, hence inductive and coinductive predicates can coexist in the same program. We illustrate the paradigm by examples, and provide declarative and operational semantics, proving the correctness of the latter. Finally, we describe a prototype meta-interpreter.

1 Introduction

Coinductive logic programming [11, 13, 12, 4] extends standard logic programming with the ability of reasoning about infinite objects and their properties. Whereas syntax of logic programs remains the same, semantics is different. To illustrate this, let us consider the following logic program which defines some predicates on lists of integers, constructed with the standard function symbols `[]` of arity 0 for the empty list and `[_|_]` of arity 2 for the list consisting of a head element and a tail. For simplicity, we will consider built-in integers, as they are in Prolog.

```
all_pos([]).
all_pos([N|L]) :- N>0, all_pos(L).

member(X,[X|_]).
member(X,[Y|L]):-X\=Y, member(X,L).

max([N],N).
max([N|L],M2) :- max(L,M), M2 is max(N,M).
```

The expected meaning is that `all_pos(`l`)` holds if all the elements of l are positive, `member(`x`,`l`)` if x is an element of l, `max(`l`,`n`)` if n is the greatest element of list l. As will be illustrated in detail in Sect. 2, in standard logic programming terms are inductively defined, that is, are finite, and predicates are inductively defined as well. In the example program, only finite lists are considered, such as, e.g., `[1|[2|[]]]`, and the three predicates are correctly defined on such lists.

In coinductive logic programming, instead, terms are coinductively defined, that is, can be infinite, and predicates are coinductively defined as well. In the example program, also infinite lists such as `[1|[2|[3|[4|...]]]]`, are considered, and the coinductive interpretation of the predicate `all_pos` gives the expected meaning on such lists. However, this is not the case for the other two predicates: for `member`

E. Komendantskaya and J. Power (Eds.):
Coalgebra, Horn Clause Logic Programming and Types
EPTCS 258, 2017, pp. 1–18, doi:10.4204/EPTCS.258.1

the correct interpretation is the inductive one, whereas for `max` neither the inductive nor the coinductive interpretation are correct: with the former the predicate is always false on infinite lists, with the latter `max(`l`,`n`)` is true whenever n is greater than all the elements of l.

The last example shows that the coinductive interpretation of predicates is sometimes *not precise enough*, in the sense that also wrong facts are included. This problem can be found, and has been studied, also in other programming paradigms, and some solutions have been proposed which allow the programmer to interpret corecursive definitions not in the standard coinductive way [8, 9, 2, 5, 6].

In this paper, we solve the problem in a more foundational way, by applying to the case of logic programs a notion recently introduced in the more general framework of inference systems [3] (indeed, a logic program can be seen as an inference system where judgments are atoms). That is, programs, consisting of facts and rules with the usual syntax, can be enriched by *co-facts* (corresponding to *coaxioms* in [3]), which syntactically resemble facts but have a special meaning: intuitively, they can only be applied "at infinite depth" in a proof tree, as will be formally defined in the following. By adding co-facts, the intended meaning (declarative semantics) of a program can be a fixed point which is not necessarily the least, nor the greatest one.

In this way, it is possible to express predicates on non well-founded structures, such as infinite lists and graphs, for which the coinductive interpretation would be not precise enough. Moreover, this paradigm nicely subsumes standard (inductive) and coinductive logic programming, since both can be expressed by a particular choice of co-facts, hence inductive and coinductive predicates can coexist in the same program.

For what concerns operational semantics, in coinductive logic programming standard SLD resolution is replaced by co-SLD resolution [13, 4], which, roughly speaking, keeps trace of the already encountered goals, called *(dynamic) coinductive hypotheses*, so that, when a goal is encountered the second time, it is considered successful. In this paper, we define an operational semantics of logic programs which is similar to co-SLD resolution, but takes co-facts into account, and prove its correctness with respect to the proposed declarative semantics. The proof is interesting since it is a non-trivial application of the *bounded coinduction principle* introduced in [3].

The operational semantics we define is in big-step style, as also proposed for co-SLD resolution [4], and, hence, is amenable for directly deriving an implementation; indeed, we have implemented in SWI-Prolog a prototype meta-interpreter, whose clauses are driven by our operational semantics, and with which we have been able to successfully test all examples shown in this paper, and many others.

The rest of the paper is organized as follows: in Sect. 2 we introduce logic programs with co-facts, their declarative semantics, and the bounded coinduction principle to reason on such programs, illustrating the notions with some examples. In Sect. 3 we formally define operational semantics, show a derivation example, and prove soundness with respect to declarative semantics. In Sect. 4 we describe the prototype meta-interpreter. In Sect. 5 we summarize our contribution, survey related work, and discuss further work.

2 Co-facts

We recall some notions about standard [10, 7] and coinductive [11, 13, 12] logic programming.

Assume a *signature* consisting of sets of *predicate symbols* p, *function symbols* f, and *variable symbols* X, each one with an associated *arity* ≥ 0, being 0 for variable symbols. A function of arity 0 is called a *constant*.

Terms are (possibly infinite) trees where nodes are labeled with function and variable symbols and

the number of children of a node corresponds to the symbol arity (for a more formal definition based on paths see, e.g., [4]). *Atoms* are (possibly infinite) trees where the root is labeled with a predicate symbol and other nodes are labeled with function and variable symbols, also accordingly with the arity. Terms and atoms are *ground* if they do not contain variable symbols, *finite* (or *syntactic*) if they are finite trees.

A *logic program* P is a set of *(definite) clauses* of shape $A :- B_1, \ldots, B_n$, where $A, B_1, \ldots, B_n$ are finite atoms. A clause where $n = 0$ is called a *fact*, otherwise it is called a *rule*.

A *substitution* θ is a mapping from a finite subset of variables into terms. We write $t\theta$ for the application of a substitution θ to a term t. We call $t\theta$ an *instance* of t. These notions can be analogously defined on atoms and clauses. A substitution is *ground* (or *grounding*) if it maps variables into ground terms, it is *syntactic* if it maps variables into finite (syntactic) terms. The *declarative semantics* of a logic program describes its meaning in an abstract way, as the set of ground atoms which are defined to be true by the program, in a sense to be made precise depending on the kind of declarative semantics we choose. In the following paragraphs, we briefly recall the standard declarative semantics of logic programs, then their coinductive declarative semantics, and finally we define the declarative semantics *generated by co-facts* and discuss its advantages.

Standard declarative semantics In the standard declarative semantics of logic programs, only finite terms and atoms are considered. The *Herbrand universe* HU is defined as the set of finite ground terms, and the *Herbrand base* HB as the set of finite ground atoms. Sets $I \subseteq \mathsf{HB}$ are called *interpretations*. Given a logic program P, the (one step) inference operator $T_P : \wp(\mathsf{HB}) \to \wp(\mathsf{HB})$ is defined as follows:

$$T_P(I) = \{A \mid (A :- B_1, \ldots, B_n) \in \mathsf{ground}(P), \{B_1, \ldots, B_n\} \subseteq I\}$$

where $\mathsf{ground}(P)$ is the set of instances of clauses in P obtained by a ground syntactic substitution.

An interpretation is a *model* of a program P (is *closed* with respect to P) if $T_P(I) \subseteq I$. The standard declarative semantics of P is the least interpretation which is a model taking as order set inclusion, that is, the intersection of all closed interpretations. Defining a *proof tree* for a ground atom A as a tree where the root is A, nodes are ground instances of rules, and leaves are ground instances of facts, the standard declarative semantics can be equivalently characterized as the set of finite ground atoms which have a finite proof tree.

It is easy to see that, with this definition, the predicates of the example program introduced in Sect. 1 have the expected meaning on finite lists. For instance, for the predicate `max` we obtain all atoms `max(`l`,`n`)` where l is a (ground term representing a) finite list and n is the greatest element of l.

Co-inductive declarative semantics A limit of the standard declarative semantics described above is that we cannot define predicates on non-well-founded structures, such as infinite lists or graphs. Considering our running example, we would like to define the predicates `all_pos`, `member`, and `max` on infinite lists as well. To obtain this, first of all infinite terms and atoms should be included. The *complete Herbrand Universe* $\mathsf{co\text{-}HU}$ [10] is the set of (finite and infinite) ground terms. The *complete Herbrand base* $\mathsf{co\text{-}HB}$ is the set of (finite and infinite) ground atoms. Sets $I \subseteq \mathsf{co\text{-}HB}$ are called *co-interpretations*.

We can define the (one step) inference operator $T_P : \wp(\mathsf{co\text{-}HB}) \to \wp(\mathsf{co\text{-}HB})$ analogously to that above:

$$T_P(I) = \{A \mid (A :- B_1, \ldots, B_n) \in \mathsf{co\text{-}ground}(P), \{B_1, \ldots, B_n\} \subseteq I\}$$

where co-ground(P) is the set of instances of clauses in P obtained by a ground substitution.[1]

Again as above, a co-interpretation is a *model* of a program P (is *closed* with respect to P) if $T_P(I) \subseteq I$, and we can consider the the least co-interpretation which is a model, that is, the intersection of all closed co-interpretations. This co-interpretation, called in the following *inductive (declarative) semantics* of P and denoted $Ind(P)$, generalizes the standard declarative semantics of predicates to infinite terms. However, in order to prove predicates on such infinite terms, we would like to allow infinite proof trees as well, and to this end a different declarative semantics can be adopted, explained below.

A co-interpretation I is a *co-model* of P (is *consistent* with respect to P) if and only if $I \subseteq T_P(I)$. The *coinductive (declarative) semantics* of P, denoted $CoInd(P)$, is the greatest co-interpretation which is a co-model taking as order set inclusion, that is, the union of all consistent co-interpretations. Equivalently, $CoInd(P)$ can be characterized as the set of ground atoms which have a (finite or infinite) proof tree.

Let us analyze what happens in the running example. Consider first the predicate `all_pos`. It is easy to see that, with the inductive semantics $Ind(P)$, this predicate has the expected meaning only on finite lists (as in the standard semantics) and on infinite lists which have a non positive element. However, for l term representing an infinite list of positives, e.g., $l =$`[1|[2|[3|[4|...]]]]`, the atom `all_pos(`l`)` cannot be proved by a finite proof tree, hence the predicate turns out to be false. With the coinductive semantics, instead, we can prove `all_pos(`l`)` by the infinite proof tree shown below:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\texttt{all\_pos([3|[4|...]])}}}{\texttt{all\_pos([2|[3|[4|...]]])}}}{\texttt{all\_pos([1|[2|[3|[4|...]]]])}}}$$

Hence, this predicate is a typical example where coinductive semantics is necessary (when including infinite terms), and provides the expected meaning. However, this is not the case for the other two predicates. More precisely, for l (term representing an) infinite list:

- `member(`x`,`l`)` always holds. In this case, the coinductive semantics $CoInd(P)$ seems not to be the right choice, and the desired semantics is obtained by taking the inductive semantics (least model) on infinite lists as well, that is, $Ind(P)$. For this reason, in coinductive logic programming the programmer can specify by a special notation that some predicates should be interpreted inductively rather than coinductively.
- `max(`l`,`n`)` holds whenever n is greater than all the elements of l. In this case, the coinductive semantics *includes* the desired semantics (and it is necessary for this, since `max(`l`,`n`)` would never hold in the inductive semantics), but it is not precise enough. Indeed, for an infinite (regular) list $l =$`[1|[2|[1|[2|...]]]]`, we can prove `max(`l`,2)` as expected, but we can also prove, e.g., `max(`l`,4)`, as shown by the infinite proof trees (T1) and (T2), respectively, shown below:

$$\text{(T1)}\ \dfrac{\dfrac{\dfrac{\vdots}{\texttt{max([1|[2|[1|...]]],2)}}}{\texttt{max([2|[1|[2|...]]],2)}}}{\texttt{max([1|[2|[1|...]]],2)}} \qquad \text{(T2)}\ \dfrac{\dfrac{\dfrac{\vdots}{\texttt{max([1|[2|[1|...]]],4)}}}{\texttt{max([2|[1|[2|...]]],4)}}}{\texttt{max([1|[2|[1|...]]],4)}}$$

Declarative semantics generated by co-facts In order to define this semantics, first of all the syntax is slightly generalized allowing, besides facts and rules, *co-facts*, written `.A`. Co-facts are finite atoms,

[1] An instance of a clause in P obtained by mapping some variables into infinite ground terms belongs to co-ground(P), but does not belong to ground(P).

hence syntactically resemble facts, but have a special meaning. Below is the version equipped with co-facts of the running example.

```
all_pos([]).
all_pos([N|L]) :- N>0, all_pos(L).
.all_pos(_)

member(X,[X|_]).
member(X,[Y|L]) :- X\=Y, member(X,L).

max([N],N).
max([N|L],M2) :- max(L,M), M2 is max(N,M).
.max([N|_],N)
```

In the following, the metavariable C denotes a set of co-facts (finite atoms).

The *(declarative) semantics* of a program P *generated by co-facts* C, denoted $Gen(P,C)$, is defined as follows.

- First, we consider the program $P_{\sqcup C}$ obtained by enriching P by the co-facts in C considered as facts, and we take its inductive semantics $Ind(P_{\sqcup C})$.
- Then, we take the largest co-model which is included in $Ind(P_{\sqcup C})$. In other words, we take the coinductive interpretation of P where, however, clauses are instantiated only on elements of $Ind(P_{\sqcup C})$.

Note that $Gen(P,C)$ is different from $CoInd(P) \cap Ind(P_{\sqcup C})$. For instance, let P be the program

```
p(0) :- p(0), p(1)
p(1) :- p(0), p(1)
```

and C be the singleton set consisting of the co-fact

```
.(p(0))
```

Then, $CoInd(P) = \{\texttt{p(0)},\texttt{p(1)}\}$, and $Ind(P_{\sqcup C}) = \{\texttt{p(0)}\}$. Hence the intersection is $\{\texttt{p(0)}\}$, whereas $Gen(P,C) = \emptyset$.

As we have shown in [3] in the more general framework of inference systems, $Gen(P,C)$ corresponds to a fixed-point of the operator T_P which is neither the greatest, nor the least one.

In terms of proof trees, $Gen(P,C)$ is the set of ground atoms which have a (finite or infinite) proof tree in P whose nodes all have a finite proof tree in $P_{\sqcup C}$. Taking this semantics, all the predicates in the running example get the expected meaning. Indeed, for l (term representing an) infinite list:

- `all_pos(`l`)` holds for l infinite list of positives, since the atom `all_pos(`l`)` has the previously shown infinite proof tree, and all its nodes have a (trivial) finite proof tree in $P_{\sqcup C}$, consisting in an instantiation of the co-fact.
- `member(`x`,`l`)` only holds if x belongs to l. Otherwise, there would exist an infinite proof tree in P, but its nodes have no finite proof tree in $P_{\sqcup C}$, since there are no co-facts, and the only fact is not applicable.
- `max(`l`,`n`)` only holds when n is the greatest element of l. In this case, as shown at page 4, there exists an infinite proof tree in P whenever n is greater than all the elements of l. However, a finite proof tree in $P_{\sqcup C}$ for each node only exists when n belongs to the list, hence is actually the greatest element. For instance, the two nodes of the infinite proof tree (T1) for `max([1|[2|[1|[2|...]]]],2)` have the finite proof trees (FT1) and (FT2) shown below:

$$\text{(FT1)}\dfrac{\dfrac{}{\texttt{max([2|[1|[2|...]]],2)}}}{\texttt{max([1|[2|[1...]]],2)}} \qquad \text{(FT2)}\dfrac{}{\texttt{max([2|[1|[2|...]]],2)}}$$

whereas there is no finite proof tree for the nodes of the infinite proof tree (T2). In other words, co-facts allow the programmer to "filter out" atoms which should not be true, making the semantics precise.

Note that the condition to have a finite proof tree in $P_{\sqcup C}$ trivially holds for nodes which are roots of a finite subtree (a finite proof tree in P is a finite proof tree in $P_{\sqcup C}$ as well), hence is only significant for nodes which occur in an infinite path. Moreover, if the infinite tree is *rational*, that is, has a finite number of different subtrees, then an infinite path always consists of (possibly) a finite prefix and a period. Hence, if the condition holds for the first node of the period, then it holds for all the other nodes. In the example of (T1) above, another finite proof tree for the second node `max([2|[1|[2|...]]],2)` can be obtained from (FT1) for the first node:

$$\dfrac{\dfrac{\dfrac{}{\texttt{max([2|[1|[2|...]]],2)}}}{\texttt{max([1|[2|[1|...]]],2)}}}{\texttt{max([2|[1|[2|...]]],2)}}$$

Correspondingly, in the operational semantics which will be provided in Sect. 3, standard SLD resolution in $P_{\sqcup C}$ is triggered when an atom is encountered the second time.

Note also that, as the examples above clearly show, the inductive and coinductive semantics can be obtained as special cases of the semantics generated by co-facts of a program, notably:

- the inductive semantics when the set of co-facts is empty;
- the coinductive semantics when the set of (ground instances of) co-facts is co-HB.

That is, co-facts allow to mix together, without any need of a special notation, predicates for which the appropriate interpretation is either inductive or coinductive, and to express predicates for which the appropriate interpretation is neither of the two, as the `max` example shows.

Let G be a set of ground atoms, corresponding to the intended meaning of some predicates.

In order to prove that the the atoms in G are defined to be true by a program P enriched by co-facts C, that is, $G \subseteq \mathit{Gen}(P,C)$, we can use the following *bounded coinduction principle*, which is a generalization of the standard coinduction principle. This and other proof techniques are illustrated in the more general framework of inference systems in [3].

Theorem 2.1 (Bounded coinduction principle)**.** *If the following two conditions hold:*

Boundedness $G \subseteq \mathit{Ind}(P_{\sqcup C})$*, that is, each atom in* G *has a finite proof tree in* $P_{\sqcup C}$

Consistency $G \subseteq T_P(G)$*, that is, for each atom* $A \in G$*,*
$A :- B_1,\ldots,B_n \in \mathit{co\text{-}ground}(P)$ *for some* $B_1,\ldots,B_n \in G$

then $G \subseteq \mathit{Gen}(P,C)$.

Proof. The two conditions corresponds to require that G is a co-model of P (is *consistent* with respect to P) which is *bounded* by (included in) $\mathit{Ind}(P_{\sqcup C})$, and $\mathit{Gen}(P,C)$ is defined as the largest such co-model. □

The standard coinduction principle can be obtained as a specific instance of the principle above, when (ground instances of) C coincide with co-HB; for this particular case the first condition trivially holds.

We illustrate the proof technique by formally proving that, in the running example, we can derive all atoms in the set $G^{\texttt{max}} = \{\texttt{max}(n,l) \mid n \text{ greatest element of } l\}$.

- To prove boundedness, we have to show that, for each `max(`l`,`n`)` such that n is the greatest element of l, `max(`l`,`n`)` has a *finite* proof tree in $P_{\sqcup C}$. This can be easily shown. Indeed, if n is the greatest element of l, then $l = [n_1|[\ldots[n_k|l']]]$ with $n_k = n$, $n_i \leq n$ for $i \in [1..k-1]$. Hence, `max(`l`,`n`)` has

a finite proof tree in $P_{\sqcup C}$ which consists in $k-1$ nodes which are instances of the rule and a leaf which is an instance of the co-fact (a concrete example for `max([2|[1|[2|1|...]]],2)` has been shown before).

- To prove consistency, we have to show that, for each `max(`l`,`n`)` such that n greatest element of l, `max(`l`,`n`)` is the consequence of (an instance of) a clause with all atoms of the body in $G^{\texttt{max}}$. This can be easily shown, indeed, if n is the greatest element of l, then $l = [n'|l']$ with $n' \leq n$ and n greatest element of l'. Hence, `max(`l`,`n`)` is the consequence of the following instance of the rule: `max([`n'`|`l'`],`n`) :- max(`l'`,`n`),` n `is max(`n'`,`n`)` where the atom `max(`l'`,`n`)` belongs to $G^{\texttt{max}}$.

The same proof technique will be used in the next section to show that the operational semantics is sound with respect to the semantics generated by co-facts.

3 Big-step operational semantics

In this section we define an operational counterpart of the semantics generated by co-facts introduced in Sect. 2, and prove its correctness.

This operational semantics is a generalization of SLD [10, 7] and co-SLD [11, 13, 12] resolution. However, it is presented, rather than in the traditional small-step style, in big-step style, as introduced in [4]. This style turns out to be simpler since coinductive hypotheses (see below) can be kept local. Moreover, it naturally leads to an interpreter, and the proof of soundness with respect to declarative semantics is more direct since we compare two inference systems. For a proof of equivalence of big-step and small-step co-SLD resolution see [4].

We introduce some notations. First of all, in this section the metavariable A denotes finite (syntactic) atoms and the metavariables s,t denote finite (syntactic) terms. An *equation* has shape $s = t$ where s and t are finite terms. We write E for a set of equations. Equations allow a finite (syntactic) representation also of *rational* terms and atoms, that is, having a finite number of different subterms, see, e.g., [1, 4] for the details. Finally, a *goal* is a sequence of atoms, and the metavariable G denotes a syntactic goal, that is, a sequence of finite atoms. The empty sequence is denoted by ε.

The operational semantics is defined by a judgment $\vdash_{\langle P,C\rangle} G \Rightarrow E$, meaning that, given a program P and a set of co-facts C, resolution of the (syntactic) goal G succeeds and produces a set of equations E that describes a solution of the goal. For instance, assuming P and C from the previous section, which define, among others, the predicate `max` computing the greatest element of a list, it would be possible to derive the following judgment

$$\vdash_{\langle P,C\rangle} \texttt{max(L,2)} \Rightarrow \{\texttt{L=[1|L1], L1=[2|L]]}\}$$

meaning that a solution of the goal `max(L,2)` is the infinite term `l = [1|[2|[1|2| ... ]]]`.

This judgment is defined following a schema which is similar to co-SLD resolution, in the sense that resolution keeps track of the already encountered atoms, which are called *(dynamic) coinductive hypotheses* [13]. However, when the same atom A is encountered the second time, rather than just considering A successful as it happens in co-SLD resolution, standard SLD resolution of A is triggered in the program $P_{\sqcup C}$ obtained by enriching P by the co-facts in C.

Formally, two auxiliary judgments are introduced:

- $S \vdash_{\langle P,C\rangle} \langle G \square E\rangle \Rightarrow E'$, meaning that, given a program P and a set of co-facts C, resolution of the goal represented by G and E, under the *coinductive hypotheses* S, succeeds producing a set of equations E' which describes a solution of the goal.

$$\text{(empty)}\ \frac{}{\vdash_P \langle \varepsilon \,\square\, E\rangle \Rightarrow E}$$

$$\text{(step)}\ \frac{\begin{array}{l} p(t_1,\ldots,t_n) :- A_1,\ldots,A_m \text{ renaming of a clause in } P \text{ with fresh variables} \\ E_1 \cup \{s_1 = t_1,\ldots,s_n = t_n\} \text{ solvable} \\ \vdash_P \langle A_1,\ldots,A_m \,\square\, E_1 \cup \{s_1 = t_1,\ldots,s_n = t_n\}\rangle \Rightarrow E_2 \\ \vdash_P \langle G_1, G_2 \,\square\, E_2\rangle \Rightarrow E_3 \end{array}}{\vdash_P \langle G_1, p(s_1,\ldots,s_n), G_2 \,\square\, E_1\rangle \Rightarrow E_3}$$

$$\text{(co-empty)}\ \frac{}{S \vdash_{\langle P,C\rangle} \langle \varepsilon \,\square\, E\rangle \Rightarrow E}$$

$$\text{(co-step)}\ \frac{\begin{array}{l} p(t_1,\ldots,t_n) :- A_1,\ldots,A_m \text{ renaming of a clause in } P \text{ with fresh variables} \\ E_1 \cup \{s_1 = t_1,\ldots,s_n = t_n\} \text{ solvable} \\ S \cup \{p(s_1,\ldots,s_n)\} \vdash_{\langle P,C\rangle} \langle A_1,\ldots,A_m \,\square\, E_1 \cup \{s_1 = t_1,\ldots,s_n = t_n\}\rangle \Rightarrow E_2 \\ S \vdash_{\langle P,C\rangle} \langle G_1, G_2 \,\square\, E_2\rangle \Rightarrow E_3 \end{array}}{S \vdash_{\langle P,C\rangle} \langle G_1, p(s_1,\ldots,s_n), G_2 \,\square\, E_1\rangle \Rightarrow E_3}$$

$$\text{(co-hyp)}\ \frac{\begin{array}{l} p(t_1,\ldots,t_n) \in S \qquad E_1 \cup \{s_1 = t_1,\ldots,s_n = t_n\} \text{ solvable} \\ \vdash_{P_{\sqcup C}} \langle p(s_1,\ldots,s_n) \,\square\, E_1 \cup \{s_1 = t_1,\ldots,s_n = t_n\}\rangle \Rightarrow E_2 \\ S \vdash_{\langle P,C\rangle} \langle G_1, G_2 \,\square\, E_2\rangle \Rightarrow E_3 \end{array}}{S \vdash_{\langle P,C\rangle} \langle G_1, p(s_1,\ldots,s_n), G_2 \,\square\, E_1\rangle \Rightarrow E_3}$$

$$\text{(main)}\ \frac{\emptyset \vdash_{\langle P,C\rangle} \langle G \,\square\, \emptyset\rangle \Rightarrow E}{\vdash_{\langle P,C\rangle} G \Rightarrow E}$$

Figure 1: Big-step operational semantics

- $\vdash_P \langle G \,\square\, E\rangle \Rightarrow E'$ meaning that, given a program P, standard SLD resolution of the goal represented by G and E succeeds producing a set of equations E' which describes a solution of the goal.

Rules (inductively) defining these judgments are shown in Fig. 1.

The first two rules define the judgment $\vdash_P \langle G \,\square\, E\rangle \Rightarrow E'$ corresponding to standard SLD resolution. In rule (empty), resolution of the empty goal succeeds. In rule (step), an atom $p(s_1,\ldots,s_n)$ is selected from the goal to be resolved, and a clause of the program is chosen such that the selected atom unifies with the head of the clause, as expressed by the fact that adding the equations $s_1 = t_1,\ldots,s_n = t_n$ to the current set we get a solvable set of equations. Then, resolution of the original goal succeeds if resolution of both the body of the clause and the remaining goal succeed.

The following three rules define the judgment $S \vdash_{\langle P,C\rangle} \langle G \,\square\, E\rangle \Rightarrow E'$ corresponding to an intermediate step of resolution. Rules (co-empty) and (co-step) are analogous to rules (empty) and (step). The only difference is that, in rule (co-step), in the resolution of the body of the clause, the selected atom is added to the current set of coinductive hypotheses. In this way, rule (co-hyp) can handle the case when resolution encounters the same atom for the second time, that is, more formally, the selected atom unifies with a coinductive hypothesis, as expressed by the fact that adding the equations $s_1 = t_1,\ldots,s_n = t_n$ to the current set we get a solvable set of equations. In this case, standard SLD resolution of such atom is triggered in the program $P_{\sqcup C}$ obtained by enriching P by the co-facts in C, and resolution of the original goal succeeds if both such standard SLD resolution of the selected atom and resolution of the remaining goal succeed.

Finally, rule (main) defines the main judgment $\vdash_{\langle P,C\rangle} G \Rightarrow E$ corresponding to an initial step of resolution, where the goal is syntactic, that is, no equation has been produced yet.

In the following, to avoid confusion with proof trees considered in Sect. 2, trees obtained by instantiating the rules of big-step operational semantics are called *derivations*.

A derivation example Assuming P and C from the previous section, which define, among others, the predicate `max` computing the greatest element of a list, we show a derivation for the judgment $\emptyset \vdash_{\langle P,C\rangle} \langle \texttt{max(L,M)} \,\square\, E_0\rangle \Rightarrow E_4$, with[2] $E_0 = \{\texttt{L=[1,2|L]}\}$, $E_4 = \{\texttt{L=[1,2|L],M2=M,M3=2,M2=2,M1=2,M=2}\}$. For keeping the derivation simpler, in rules (co-step) and (step) we do not manifest some substitutions, but, instead, we implicitly apply them in the resulting goals. Furthermore, we consider the predicate `_ is max(_,_)` to be predefined, as it is the case in real Prolog systems; for it we have introduced a special semantic rule (labeled with p, for predefined) that deals with standard inductive predefined predicates (see Sect. 4). Finally, for space reasons we have abbreviated the names of the applied rules.

$$\text{(c-s)}\dfrac{\text{(c-h)}\dfrac{\text{(s)}\dfrac{\nabla_1 \quad \text{(e)}\dfrac{}{\vdash_{P_{\sqcup C}} \langle \varepsilon \,\square\, E_2\rangle \Rightarrow E_2}}{\vdash_{P_{\sqcup C}} \langle \texttt{max(L,M2)} \,\square\, E_0 \cup \{\texttt{M2=M}\}\rangle \Rightarrow E_2} \quad \nabla_2}{S_2 \vdash_{\langle P,C\rangle} \langle \texttt{max(L,M2),M1 is max(2,M2)} \,\square\, E_0\rangle \Rightarrow E_3} \quad \nabla_3}{S_1 \vdash_{\langle P,C\rangle} \langle \texttt{max([2|L],M1),M is max(1,M1)} \,\square\, E_0\rangle \Rightarrow E_4} \quad \text{(c-e)}\dfrac{}{\emptyset \vdash_{\langle P,C\rangle} \langle \varepsilon \,\square\, E_4\rangle \Rightarrow E_4}}{\emptyset \vdash_{\langle P,C\rangle} \langle \texttt{max(L,M)} \,\square\, E_0\rangle \Rightarrow E_4}$$

with $E_2 = \{\texttt{L=[1,2|L],M2=M,M3=2,M2=2}\}$, $E_3 = \{\texttt{L=[1,2|L],M2=M,M3=2,M2=2,M1=2}\}$, $S_1 = \{\texttt{max(L,M)}\}$, $S_2 = \{\texttt{max(L,M), max([2|L],M1)}\}$, and where ∇_1, ∇_2, and ∇_3 are the following derivations:

$$\nabla_1 = \text{(s*)}\dfrac{\text{(e)}\dfrac{}{\vdash_{P_{\sqcup C}} \langle \varepsilon \,\square\, E_1\rangle \Rightarrow E_1} \quad \text{(s)}\dfrac{\text{(e)}\dfrac{}{\vdash_{P_{\sqcup C}} \langle \varepsilon \,\square\, E_2\rangle \Rightarrow E_2} \quad \text{(e)}\dfrac{}{\vdash_{P_{\sqcup C}} \langle \varepsilon \,\square\, E_2\rangle \Rightarrow E_2}}{\vdash_{P_{\sqcup C}} \langle \texttt{M2 is max(1,M3)} \,\square\, E_1\rangle \Rightarrow E_2}}{\vdash_{P_{\sqcup C}} \langle \texttt{max([2|L],M3),M2 is max(1,M3)} \,\square\, E_0 \cup \{\texttt{M2=M}\}\rangle \Rightarrow E_2}$$

$$\nabla_2 = \text{(p)}\dfrac{\text{(e)}\dfrac{}{\vdash_P \langle \varepsilon \,\square\, E_3\rangle \Rightarrow E_3} \quad \text{(c-e)}\dfrac{}{S_2 \vdash_{\langle P,C\rangle} \langle \varepsilon \,\square\, E_3\rangle \Rightarrow E_3}}{S_2 \vdash_{\langle P,C\rangle} \langle \texttt{M1 is max(2,M2)} \,\square\, E_2\rangle \Rightarrow E_3}$$

$$\nabla_3 = \text{(p)}\dfrac{\text{(e)}\dfrac{}{\vdash_P \langle \varepsilon \,\square\, E_4\rangle \Rightarrow E_4} \quad \text{(c-e)}\dfrac{}{S_1 \vdash_{\langle P,C\rangle} \langle \varepsilon \,\square\, E_4\rangle \Rightarrow E_4}}{S_1 \vdash_{\langle P,C\rangle} \langle \texttt{M is max(1,M1)} \,\square\, E_3\rangle \Rightarrow E_4}$$

with $E_1 = \{\texttt{L=[1,2|L],M2=M,M3=2}\}$.

In the whole derivation the co-fact for `max` is used just once in rule (step) marked with (s*) in ∇_1 derivation; the co-fact could be employed also in rule (step) in derivation ∇_1, but without success, since the substitution $\{\texttt{M2=M,M2=1}\}$ cannot satisfy the goal `M1 is max(2,M2),M is max(1,M1)`.

Soundness proof The big-step operational semantics is sound with respect to the declarative semantics defined in Sect. 2. That is, if resolution of the goal G succeeds producing a set of equations E which describe a solution of the goal, then (any ground instance of) this solution is a set of atoms which are true in the declarative semantics. Formally, set $\mathsf{gsol}(E) = \{\theta \mid \theta \text{ ground}, t\theta = t'\theta \text{ for all } t = t' \in E\}$ the set of the *ground solutions* of E, necessarily defined on the set $\mathsf{V}(E)$ of the variables occurring in E. In the following, when we pick a substitution from $\mathsf{gsol}(E)$, we will implicitly assume that this substitution

[2] We use the abbreviated Prolog syntax for lists in this example.

has $\mathsf{V}(E)$ as domain. Note that the set of ground solutions is antitone with respect to set inclusion, that is, if $E_1 \subseteq E_2$ then $\mathsf{gsol}(E_2) \subseteq \mathsf{gsol}(E_1)$. Soundness can be stated as follows.

Theorem 3.1 (Soundness)**.** *If $\vdash_{\langle P,C\rangle} G \Rightarrow E$ holds then, for each $\theta \in gsol(E)$, $G\theta \subseteq Gen(P,C)$.*

The proof of soundness is an application of the bounded coinduction principle (Theorem 2.1) introduced in Sect. 2. For sake of clarity, we provide the proof in a top-down manner, that is, we first give the proof schema, and then state and prove the three needed subtheorems.

Set $OpSem(P,C)$ the set of atoms which are true in the operational semantics, that is, $OpSem(P,C) = \{A\theta \mid \vdash_{\langle P,C\rangle} A \Rightarrow E, \theta \in \mathsf{gsol}(E)\}$. We write $\theta|\theta'$ if θ and θ' agree on the common domain, that is, for each $X \in \mathsf{dom}(\theta) \cap \mathsf{dom}(\theta')$, $X\theta = X\theta'$. Note that, if $\theta|\theta'$, then $\theta \cup \theta'$ is a well-defined substitution; analogously, we write $E_1|E_2$ if each ground solution of E_1 agrees with a ground solution of E_2 on the common domain, that is, for each $\theta \in \mathsf{gsol}(E_1)$ there exists $\theta' \in \mathsf{gsol}(E_2)$ such that $\theta|\theta'$.

Proof of Theorem 3.1. Thanks to Lemma 3.2(4), the statement can be equivalently formulated as $OpSem(P,C) \subseteq Gen(P,C)$. By bounded coinduction, we have to show that:

Boundedness $OpSem(P,C) \subseteq Ind(P_{\sqcup C})$, that is, each $A \in OpSem(P,C)$ has a finite proof tree in $P_{\sqcup C}$.

Consistency $OpSem(P,C) \subseteq T_P(OpSem(P,C))$, that is, for all $A\theta \in OpSem(P,C)$, there are $B_1\theta,\ldots,B_n\theta \in OpSem(P,C)$ such that $A\theta :- B_1\theta,\ldots,B_n\theta \in \mathsf{co\text{-}ground}(P)$.

The two conditions can be proved as follows.

- To prove boundedness, we have to show that, if $\vdash_{\langle P,C\rangle} A \Rightarrow E, \theta \in \mathsf{gsol}(E)$, then $A\theta \in Ind(P_{\sqcup C})$. This can be proved by two steps:
  - $\vdash_{\langle P,C\rangle} A \Rightarrow E$ implies $\vdash_{P_{\sqcup C}} \langle A \square \emptyset\rangle \Rightarrow E'$ with $\theta \in \mathsf{gsol}(E')$, that follows from Theorem 3.2, since $E' \subseteq E$ and so $\mathsf{gsol}(E) \subseteq \mathsf{gsol}(E')$.
  - $\vdash_{P_{\sqcup C}} \langle A \square \emptyset\rangle \Rightarrow E$ implies $A\theta \in Ind(P_{\sqcup C})$ (by Theorem 3.3)
- Consistency can be proved as follows:
  - $\vdash_{\langle P,C\rangle} A \Rightarrow E$ implies that there exist $B_1,\ldots,B_n$ such that $A\theta :- B_1\theta,\ldots,B_n\theta \in \mathsf{co\text{-}ground}(P)$ and $\vdash_{\langle P,C\rangle} B_1,\ldots,B_n \Rightarrow E'$ with $E|E'$ (by Theorem 3.4)
  - $\vdash_{\langle P,C\rangle} B_1,\ldots,B_n \Rightarrow E'$ implies $\vdash_{\langle P,C\rangle} B_i \Rightarrow E'_i$ with $E'_i \subseteq E'$ (by Lemma 3.2(4)), that implies $\mathsf{gsol}(E') \subseteq \mathsf{gsol}(E'_i)$.
  - $\vdash_{\langle P,C\rangle} B_i \Rightarrow E'_i$ implies $B_i\theta \in OpSem(P,C)$, that follows from the following facts:
    - there exists $\sigma \in \mathsf{gsol}(E')$ such that $\theta|\sigma$, since $E|E'$
    - $\sigma \in \mathsf{gsol}(E'_i)$, hence $\mathsf{V}(B_i) \subseteq \mathsf{dom}(\sigma)$
    - $B_i\sigma \in OpSem(P,C)$ by definition of $OpSem(P,C)$
    - $B_i\theta = B_i\sigma$ since $\theta|\sigma$ and $\mathsf{V}(B_i) \subseteq \mathsf{dom}(\theta) \cap \mathsf{dom}(\sigma)$.

□

Now we detail the proof, by stating and proving the three theorems and the lemma used above. Some proofs are given in Appendix A. We first introduce the following notations. If $A = p(t_1,\ldots,t_n)$, and $B = p(s_1,\ldots,s_n)$, then we write $E^{A=B}$ for the set of equations that represents the unification of A with B, that is, $E^{A=B} = \{s_1 = t_1,\ldots,s_n = t_n\}$.

The boundedness condition is obtained by Theorem 3.2 and Theorem 3.3. The former states that resolution under coinductive hypotheses implies standard SLD resolution in the program enriched by co-facts, the latter states that standard SLD resolution is sound with respect to the inductive semantics.

We start with an auxiliary lemma.

Lemma 3.1. *If* $\vdash_P \langle G \Box E_1 \rangle \Rightarrow E_2$ *holds, then*

1. *if* $E_1' \subseteq E_1$ *then* $\vdash_P \langle G \Box E_1' \rangle \Rightarrow E_2'$ *holds, with* $E_2' \subseteq E_2$
2. *if* $E \cup E_2$ *is solvable then* $\vdash_P \langle G \Box E_1 \cup E \rangle \Rightarrow E_2 \cup E$ *holds*

Proof. Straightforward induction on inference rules. □

Corollary 3.1. *If* $\vdash_P \langle G \Box E_1 \cup E \rangle \Rightarrow E_2$ *holds and* $E_2 \cup E'$ *is solvable then* $\vdash_P \langle G \Box E_1 \cup E' \rangle \Rightarrow E''$ *holds, with* $E' \subseteq E'' \subseteq E_2 \cup E'$.

The following theorem states that if resolution under coinductive hypotheses of a goal succeeds, then standard SLD resolution of the goal in the program enriched by co-facts succeeds as well.

Theorem 3.2. *If* $S \vdash_{\langle P,C \rangle} \langle G \Box E_1 \rangle \Rightarrow E_2$ *holds, then* $\vdash_{P_{\sqcup C}} \langle G \Box E_1 \rangle \Rightarrow E_2'$ *holds with* $E_2' \subseteq E_2$.

Proof. By induction on the inference rules which define $S \vdash_{\langle P,C \rangle} \langle G \Box E_1 \rangle \Rightarrow E_2$.

(co-empty) We trivially conclude by rule (empty).

(co-step) By inductive hypothesis we get that $\vdash_{P_{\sqcup C}} \langle A_1, \ldots, A_n \Box E_1 \cup E^{A=A'} \rangle \Rightarrow E_2'$ and $\vdash_{P_{\sqcup C}} \langle G_1, G_2 \Box E_2 \rangle \Rightarrow E_3'$ hold with $E_2' \subseteq E_2$, $E_3' \subseteq E_3$ and $A' :- A_1, \ldots, A_n$ a fresh renaming of a clause in $P_{\sqcup C}$. By Lemma 3.1(1) we get that $\vdash_{P_{\sqcup C}} \langle G_1, G_2 \Box E_2' \rangle \Rightarrow E_3''$ holds, with $E_3'' \subseteq E_3' \subseteq E_3$, thus by rule (step) we get the thesis.

(co-hyp) By hypothesis and by Lemma 3.1(1) we get that $\vdash_{P_{\sqcup C}} \langle A \Box E_1 \rangle \Rightarrow E_2'$ holds, with $E_2' \subseteq E_2$. From this, by rule (step), it follows that $\vdash_{P_{\sqcup C}} \langle A_1, \ldots, A_n \Box E_1 \cup E^{A=A'} \rangle \Rightarrow E_2'$ holds, with $A' :- A_1, \ldots, A_n$ a fresh renaming of a clause in $P_{\sqcup C}$. By inductive hypothesis we get $\vdash_{P_{\sqcup C}} \langle G_1, G_2 \Box E_2 \rangle \Rightarrow E_3'$ holds with $E_3' \subseteq E_3$ and by Lemma 3.1(1) we get that $\vdash_{P_{\sqcup C}} \langle G_1, G_2 \Box E_2' \rangle \Rightarrow E_3''$ holds with $E_3'' \subseteq E_3' \subseteq E_3$. Therefore by rule (step) we get the thesis.

□

The following theorem states that if standard SLD resolution of a goal succeeds, producing a set of equations E which describes a solution of the goal, then this solution is a set of atoms which are true in the inductive semantics. In other words, this theorem states the standard soundness property of SLD resolution. The theorem could be derived by showing equivalence of the $\vdash_P \langle A \Box \emptyset \rangle \Rightarrow E$ judgment with the traditional small-step definition of SLD resolution (as done in [4] for co-SLD resolution), and relying on the well-know soundness of the latter. A direct proof can be done by induction on the rules which define $\vdash_P \langle A \Box \emptyset \rangle \Rightarrow E$, as stated below

Theorem 3.3 (Soundness of standard SLD resolution)**.** *If* $\vdash_P \langle A \Box \emptyset \rangle \Rightarrow E$ *then, for each* $\theta \in gsol(E)$, $A\theta \in Ind(P)$.

We state now some lemmas needed to prove Theorem 3.4.

The following lemma states some properties of the judgment $S \vdash_{\langle P,C \rangle} \langle G \Box E \rangle \Rightarrow E'$. In particular, points 1 and 2 state a form of monotonicity with respect to the set of coinductive hypotheses and the set of input equations: that is, we can freely add coinductive hypotheses and remove input equations preserving the results of the evaluation. Point 3 states another monotonicity property: we can add input equations provided that this addition does not break the solvability of the output system of equations. Finally, point 4 states a decomposition property: the evaluation of a single atom in a goal produces a subset of the equations produced by the entire goal.

Lemma 3.2. *If* $S \vdash_{\langle P,C \rangle} \langle G \Box E_1 \rangle \Rightarrow E_2$ *holds, then:*

1. *if* $S \subseteq S'$ *then* $S' \vdash_{\langle P,C\rangle} \langle G \square E_1\rangle \Rightarrow E_2$ *holds*
2. *if* $E_1' \subseteq E_1$ *then* $S \vdash_{\langle P,C\rangle} \langle G \square E_1'\rangle \Rightarrow E_2'$ *holds for some* $E_2' \subseteq E_2$
3. *if* $E \cup E_2$ *is solvable then* $S \vdash_{\langle P,C\rangle} \langle G \square E_1 \cup E\rangle \Rightarrow E_2 \cup E$ *holds*
4. *if* $G = G_1, A, G_2$ *then* $S \vdash_{\langle P,C\rangle} \langle A \square E_1\rangle \Rightarrow E$ *holds for some* $E \subseteq E_2$

Proof. Straightforward induction on inference rules. □

As a consequence we get the following result.

Corollary 3.2. *If* $S \vdash_{\langle P,C\rangle} \langle G \square E_1 \cup E\rangle \Rightarrow E_2$ *holds and* $E_2 \cup E'$ *is solvable, then* $S \vdash_{\langle P,C\rangle} \langle G \square E_1 \cup E'\rangle \Rightarrow E''$ *holds, with* $E' \subseteq E'' \subseteq E_2 \cup E'$.

Proof. From point 2 and 3 of Lemma 3.2. □

Lemma 3.3. *If* $S \cup \{A\} \vdash_{\langle P,C\rangle} \langle G \square E_1\rangle \Rightarrow E_2$ *holds, and* $E_2 \cup E^{A=A'}$ *is solvable, then* $S \cup \{A'\} \vdash_{\langle P,C\rangle} \langle G \square E_1\rangle \Rightarrow E$ *holds and* $\mathit{gsol}(E_2 \cup E^{A=A'}) \subseteq \mathit{gsol}(E)$.

The proof can be found in the Appendix.

Lemma 3.4. *If* $\sigma_1 \in \mathit{gsol}(E_1)$, $\sigma_2 \in \mathit{gsol}(E_2)$, *and* $\sigma_1 | \sigma_2$, *then* $\sigma_1 \cup \sigma_2 \in \mathit{gsol}(E_1 \cup E_2)$.

Proof. Trivial. □

Theorem 3.4 (Consistency)**.** *If* $\vdash_{\langle P,C\rangle} A \Rightarrow E$ *holds and* $\theta \in \mathit{gsol}(E)$, *then there exist atoms* $B_1, \ldots, B_n$ *such that* $A\theta :- B_1\theta, \ldots, B_n\theta \in \mathit{co\text{-}ground}(P)$ *and* $\vdash_{\langle P,C\rangle} B_1, \ldots, B_n \Rightarrow E'$ *holds with* $E | E'$.

Proof. Since by hypothesis $\vdash_{\langle P,C\rangle} A \Rightarrow E$ holds, we have necessarily applied rule (main), hence $\emptyset \vdash_{\langle P,C\rangle} \langle A \square \emptyset\rangle \Rightarrow E$ holds. We have also necessarily applied rule (co-step), hence $\{A\} \vdash_{\langle P,C\rangle} \langle B_1, \ldots, B_n \square E^{A=A'}\rangle \Rightarrow E$ holds, for some fresh renamed clause $A' :- B_1, \ldots, B_n$ in P such that $E^{A=A'}$ is solvable. Therefore for all $\theta \in \mathsf{gsol}(E)$ we have $A'\theta :- B_1\theta, \ldots, B_n\theta \in \mathsf{co\text{-}ground}(P)$, and since $E^{A=A'} \subseteq E$, that implies $A\theta = A'\theta$, we get $A\theta :- B_1\theta, \ldots, B_n\theta \in \mathsf{co\text{-}ground}(P)$.
Now we have to show that $\emptyset \vdash_{\langle P,C\rangle} \langle B_1, \ldots, B_n \square \emptyset\rangle \Rightarrow E'$ holds for some E' such that $E | E'$. We prove this statement in two steps.

1. First we prove that $\emptyset \vdash_{\langle P,C\rangle} \langle B_1, \ldots, B_n \square E^{A=A'}\rangle \Rightarrow E''$ for some E'' with $E | E''$
2. Then we remove the equations in $E^{A=A'}$.

The first part can be proved by showing that for each judgement $S \cup \{A\} \vdash_{\langle P,C\rangle} \langle G \square E_1\rangle \Rightarrow E_2$ in the derivation of $\{A\} \vdash_{\langle P,C\rangle} \langle B_1, \ldots, B_n \square E^{A=A'}\rangle \Rightarrow E$, also the judgement $S \vdash_{\langle P,C\rangle} \langle G \square E_1\rangle \Rightarrow E_2'$ holds for some E_2' such that $E | E_2'$. This statement can be proved by induction on the derivation as follows.

(co-empty) Trivial.

(co-step) The rule has the following shape:

$$\text{(co-step)}\ \frac{\begin{array}{c} S \cup \{A\} \cup \{B\} \vdash_{\langle P,C\rangle} \langle C_1, \ldots, C_m \square E_1 \cup E^{B=C}\rangle \Rightarrow E_2 \\ S \cup \{A\} \vdash_{\langle P,C\rangle} \langle G_1, G_2 \square E_2\rangle \Rightarrow E_3 \end{array}}{S \cup \{A\} \vdash_{\langle P,C\rangle} \langle G_1, B, G_2 \square E_1\rangle \Rightarrow E_3} \quad \begin{array}{l} C :- C_1, \ldots, C_m \text{ in } P \\ \mathsf{V}(C, C_1, \ldots, C_m) \text{ fresh} \\ E_1 \cup E^{B=C} \text{ solvable} \end{array}$$

By inductive hypothesis we have $S \cup \{B\} \vdash_{\langle P,C\rangle} \langle C_1, \ldots, C_m \square E_1 \cup E^{B=C}\rangle \Rightarrow E_2'$ and $S \vdash_{\langle P,C\rangle} \langle G_1, G_2 \square E_2\rangle \Rightarrow E_3'$, with $E | E_2'$ and $E | E_3'$. Consider $\theta \in \mathsf{gsol}(E)$, since $E_2 \subseteq E_3 \subseteq E$, we get $\mathsf{gsol}(E) \subseteq \mathsf{gsol}(E_3) \subseteq \mathsf{gsol}(E_2)$, therefore $\theta \in \mathsf{gsol}(E_2)$ that is, $\mathsf{V}(E_2) \subseteq \mathsf{dom}(\theta)$; moreover there

exist $\sigma_1 \in \mathsf{gsol}(E'_2)$ and $\sigma_2 \in \mathsf{gsol}(E'_3)$ such that $\theta|\sigma_1$ and $\theta|\sigma_2$, because $E|E'_2$ and $E|E'_3$. Since all introduced variables are fresh, we have that $\mathsf{V}(E'_2) \cap \mathsf{V}(E'_3) \subseteq \mathsf{V}(E)$, that is, $\mathsf{dom}(\sigma_1) \cap \mathsf{dom}(\sigma_2) \subseteq \mathsf{V}(E) = \mathsf{dom}(\theta)$, thus, from what we know above we get that for each $X \in \mathsf{dom}(\sigma_1) \cap \mathsf{dom}(\sigma_2)$, $X\sigma_1 = X\theta = X\sigma_2$, that is, $\sigma_1|\sigma_2$ and $\theta|\sigma_1 \cup \sigma_2$. Therefore by Lemma 3.4 we get that $\sigma_1 \cup \sigma_2 \in \mathsf{gsol}(E'_2 \cup E'_3)$, that is, $E'_2 \cup E'_3$ is solvable and $E|E'_2 \cup E'_3$.
Therefore by Corollary 3.2 we get $S \vdash_{\langle P,C \rangle} \langle G_1, G_2 \square E'_2 \rangle \Rightarrow E'$ with $\mathsf{gsol}(E'_2 \cup E'_3) \subseteq \mathsf{gsol}(E')$, thus $E|E'$ and by rule (co-step) we get the thesis.

(co-hyp) We consider the case where the rule has the following shape, if B unifies with an atom in S the thesis follows from inductive hypothesis by applying rule (co-hyp).

$$\text{(co-hyp)}\frac{\vdash_{P_\sqcup C} \langle B \square E_1 \cup E^{A=B} \rangle \Rightarrow E_2 \quad S \cup \{A\} \vdash_{\langle P,C \rangle} \langle G_1, G_2 \square E_2 \rangle \Rightarrow E_3}{S \cup \{A\} \vdash_{\langle P,C \rangle} \langle G_1, B, G_2 \square E_1 \rangle \Rightarrow E_3} \; E_1 \cup E^{A=B} \text{ solvable}$$

By hypothesis $\{A\} \vdash_{\langle P,C \rangle} \langle B_1, \ldots, B_n \square E^{A=A'} \rangle \Rightarrow E$ holds with $A' :- B_1, \ldots, B_n$ a fresh renaming of a clause in P. Let ρ be a renaming of variables in $\mathsf{V}(E)$ which maps variables in $\mathsf{V}(A)$ in themselves, and other variables to fresh variables, then $\{A\} \vdash_{\langle P,C \rangle} \langle B'_1, \ldots, B'_n \square E^{A=A'\rho} \rangle \Rightarrow E\rho$ holds with $B'_i = B_i\rho$ and $\theta \in \mathsf{gsol}(E)$ iff $\rho^{-1}\theta \in \mathsf{gsol}(E\rho)$ and $\theta|\rho^{-1}\theta$. Now we have to show that $E\rho \cup E_1 \cup E^{B=A'\rho}$ is solvable; to this aim we consider a substitution $\theta \in \mathsf{gsol}(E)$, and note that, since $E_1 \subseteq E$, $\mathsf{gsol}(E) \subseteq \mathsf{gsol}(E_1)$, this implies $\theta \in \mathsf{gsol}(E_1)$, therefore by Lemma 3.4, $\theta \cup \rho^{-1}\theta \in \mathsf{gsol}(E\rho \cup E_1)$. Moreover, since $E^{B=A} \subseteq E$ and $E^{A=A'\rho} \subseteq E\rho$, we have $B\theta = A\theta = A(\rho^{-1}\theta) = A'\rho(\rho^{-1}\theta)$; therefore, setting $\sigma = \theta \cup \rho^{-1}\theta$ and $E' = E\rho \cup E_1 \cup E^{B=A'\rho}$, we get that $\sigma \in \mathsf{gsol}(E')$. Thus by Lemma 3.2(1), Corollary 3.2 and Lemma 3.3 we get $S \cup \{B\} \vdash_{\langle P,C \rangle} \langle B'_1, \ldots, B'_n \square E_1 \cup E^{B=A'\rho} \rangle \Rightarrow E'_2$ holds, with $\mathsf{gsol}(E') \subseteq \mathsf{gsol}(E'_2)$.
By inductive hypothesis we get $S \vdash_{\langle P,C \rangle} \langle G_1, G_2 \square E_2 \rangle \Rightarrow E'_3$ with $E|E'_3$. If $\theta \in \mathsf{gsol}(E)$ there exist $\sigma_1 \in \mathsf{gsol}(E'_2)$ and $\sigma_2 \in \mathsf{gsol}(E'_3)$ such that $\theta|\sigma_1$ and $\theta|\sigma_2$ and, since all introduced variables are fresh, $\mathsf{dom}(\sigma_1) \cap \mathsf{dom}(\sigma_2) \subseteq \mathsf{V}(E) = \mathsf{dom}(\theta)$. This implies that, for all $X \in \mathsf{dom}(\sigma_1) \cap \mathsf{dom}(\sigma_2)$, $X\sigma_1 = X\theta = X\sigma_2$, that is, $\sigma_1|\sigma_2$. Therefore by Lemma 3.4 we get $\sigma_1 \cup \sigma_2 \in \mathsf{gsol}(E'_2 \cup E'_3)$ and $\theta|\sigma_1 \cup \sigma_2$.
Finally, by Corollary 3.2 we get that $S \vdash_{\langle P,C \rangle} \langle G_1, G_2 \square E'_2 \rangle \Rightarrow E''$ with $\mathsf{gsol}(E'_2 \cup E'_3) \subseteq \mathsf{gsol}(E'')$, so the thesis follows by application of rule (co-step).

At this point we have proved that $\emptyset \vdash_{\langle P,C \rangle} \langle B_1, \ldots, B_n \square E^{A=A'} \rangle \Rightarrow E'$ holds with $E|E'$. Now applying Lemma 3.2(2) we get that $\emptyset \vdash_{\langle P,C \rangle} \langle B_1, \ldots, B_n \square \emptyset \rangle \Rightarrow E''$ with $E'' \subseteq E'$. Thus $\mathsf{gsol}(E') \subseteq \mathsf{gsol}(E'')$, hence $E|E''$. Therefore by rule (main) we get $\vdash_{\langle P,C \rangle} B_1, \ldots, B_n \Rightarrow E''$. □

4 Implementation

We have implemented a prototype meta-interpreter in SWI-Prolog, driven by the rules of the big-step operational semantics defined in Sect. 3; the complete source code can be found in Appendix B, and it is publicly available on the Web[3] together with a unit test-suite.

Tests include the predicates on lists considered in the examples provided in Sect. 2, but also other predicates on lists, as well as predicates defined on repeating decimals, grammars, graphs, and infinite regular trees [3].

[3] At `http://www.disi.unige.it/person/AnconaD/Software`.

The three main predicates `solve/1`, `solve_gfp/2`, and `solve_lfp/2` implement the semantic judgments $\vdash_{\langle P,C\rangle} G \Rightarrow E$, $S \vdash_{\langle P,C\rangle} \langle G \square E\rangle \Rightarrow E'$, and $\vdash_P \langle G \square E\rangle \Rightarrow E'$, respectively. As usual, sets of equations do not need to be explicitly manipulated by the meta-interpreter, which relies on the built-in unification mechanism offered by the Prolog interpreter, therefore the above mentioned predicates take less arguments than their corresponding judgments. Despite this fact, `solve_gfp`, and `solve_lfp` have the same arity, because `solve_lfp` needs an extra argument (in comparison to $\vdash_P \langle G \square E\rangle \Rightarrow E'$) to guarantee termination for some queries (see more explanations below).

Differently from the operational semantics, the meta-interpreter is fully deterministic and follows the standard selection rules for goal atoms (leftmost) and clauses (topmost/leftmost, as supported by the system predicate `clause`).

Despite the meta-interpreter is driven by the semantic rules, there is no one-to-one correspondence between the rules and the meta-interpreter clauses, for several reasons explained below.

The meta-interpreter allows a correct management of predefined predicates which can be identified thanks to the SWI-Prolog system predicates; for instance, the examples shown in Sect. 2 uses the built-in predicates `>`, `\=`, and `max`. To handle such predicates, we have added a specific clause which does not have a counterpart in the operational semantics: if an atom uses a predefined and necessarily inductive (either built-in, or defined in the standard library) predicate, then the meta-interpreter delegates its resolution to the Prolog interpreter.

Since the meta-programming facilities manage goals as non-empty sequences of atoms (the empty goal is represented by the single atom `true`), there are no clauses corresponding to the semantic rules (empty) and (co-empty); rather, there are two clauses (named `seq` and `co-seq`) which simply decompose non-singleton goals in their leftmost atom and rest of atoms, while the clauses dealing with singleton goals correspond to (step), (co-step), and (co-hyp) rules, with the difference that in their bodies there is no atom for resolving the remaining goal, as happens for the corresponding judgment with G_1, G_2.

From the complexity results concerning coinductive programming [4] we know that determining whether a goal succeeds w.r.t. the coinductive semantics is not even semi-decidable, and, hence, the same applies also to our extension with co-facts which includes the standard coinductive semantics.

The interpreter limits non termination in two ways:

- a cut is inserted right after atom `co_find(Atom, AtomList)` in the body of the clause for `solve_gfp` corresponding to rule (co-hyp); in this way, the clause corresponding to (co-step) is never applied for an atom which unifies with an element in the list `AtomList` of the coinductive hypotheses (that is, rule (co-hyp) is applicable); this ensures that non termination is avoided when a "loop" in the proof tree is detected. Of course, with the insertion of such a cut we may miss some correct answer [2].
- we have inserted an additional clause (called `cut`) for the `solve_lfp` predicate which uses an association to map encountered atoms (modulo unification) to the number of times they have been already processed by the meta-interpreter; clause `step` updates such an association, by checking whether the currently processed atom unifies with an atom already present in the association; since `solve_lfp` has to build a finite proof tree, no substitution is applied when unification succeeds. Clause `cut`, which precedes clause `step`, fails with no backtracking if, according to the association, the current atom has been already processed twice. Also in this case, we ensure termination at the cost of missing some correct answer.

Clause `cut` is also responsible for co-facts: it attempts to apply co-facts only when the corresponding atom is found in the association (but only for the first time); in this way termination is guaranteed in more cases. Since co-facts are managed as facts with the system predicate `clause`, it has been pretty easy to

extend the meta-interpreter to take into account also the possibility of defining *co-clauses* (see further comments in Sect. 5).

5 Conclusion

We have proposed a generalized logic programming paradigm inspired by the notion of inference system with coaxioms [3], where it is possible to consider interpretations which are between the inductive and the coinductive one. At the model-theoretic level (declarative semantics) this has been achieved by taking as semantics of a program the largest co-model of the program included in the least model of the program enriched by co-facts; at the operational level we have defined a big-step operational semantics which is a refinement of co-SLD resolution: when the same goal is encountered the second time, its standard SLD resolution is triggered in the program enriched by co-facts.

We have proved that such an operational semantics is sound with respect to the declarative semantics; furthermore, the big-step semantics rules have driven the implementation of a prototype meta-interpreter that has allowed us to successfully experiment our generalized logic programming paradigm.

Coinductive logic programming has been initially investigated and implemented by Simon et al. [13, 12]; since the earlier stages, the problem has been recognized that not all predicate definitions require a coinductive interpretation; for instance, Simon et al. have pointed out this issue for the `member` predicate, whose semantics is inductive even when infinite lists are considered. Anyway, to our knowledge, no current implementation of coinductive logic programming supports the extension of inductive predicates for the complete Herbrand base, neither the ability of defining programs where coinductive and inductive predicates can be freely mixed together.

Similar issues have been investigated in the context of coinductive functional [8, 9], and object-oriented [5, 6] paradigms, but the proposed solutions lack proof principles useful for proving correctness of programs written in these extended paradigms.

We have commented in Sect. 4 that the prototype meta-interpreter naturally supports not only co-facts, but also co-clauses, and the big-step semantics can be trivially extended to take into account this possibility, but it would be interesting to investigate whether this generalization has a natural counterpart at the level of the declarative semantics.

For what concerns the implementation, much more work is required to guarantee that the extension of logic programming with co-facts can be effectively used in practice. The meta-interpreter offers a simple solution to rapid prototyping of an implementation of the operational semantics by exploiting the reflection facilities of Prolog, but is far from being an efficient solution. Furthermore, the combination of the two predicates `solve_gfp` and `solve_lfp` for coinductive and inductive reasoning, respectively, has a bad impact on the performance because in fact a proof tree needs to be built twice. It would be interesting to investigate more clever algorithms to avoid such a duplication, or, at least, identify conditions on logic programs which are sufficient to guarantee more efficient implementations.

Another direction for further work consists in studying restricted classes of logic programs for which the operational semantics of co-facts presented here turns out to be sound and complete.

References

[1] Jirí Adámek, Stefan Milius & Jiri Velebil (2006): *Iterative algebras at work*. *Mathematical Structures in Computer Science* 16(6), pp. 1085–1131, doi:10.1017/S0960129506005706.

[2] Davide Ancona (2013): *Regular corecursion in Prolog*. *Computer Languages, Systems & Structures* 39(4), pp. 142–162, doi:10.1016/j.cl.2013.05.001.

[3] Davide Ancona, Francesco Dagnino & Elena Zucca (2017): *Generalizing inference systems by coaxioms*. In Hongseok Yang, editor: *ESOP 2017 - European Symposium on Programming*, *Lecture Notes in Computer Science* 10201, Springer, pp. 29–55, doi:10.1007/978-3-662-54434-1_2.

[4] Davide Ancona & Agostino Dovier (2015): *A Theoretical Perspective of Coinductive Logic Programming*. *Fundamenta Informaticae* 140(3-4), pp. 221–246, doi:10.3233/FI-2015-1252.

[5] Davide Ancona & Elena Zucca (2012): *Corecursive Featherweight Java*. In Wei-Ngan Chin & Aquinas Hobor, editors: *FTfJP'12 - Formal Techniques for Java-like Programs*, ACM Press, pp. 3–10, doi:10.1145/2318202.2318205.

[6] Davide Ancona & Elena Zucca (2013): *Safe Corecursion in coFJ*. In Werner Dietl, editor: *FTfJP'13 - Formal Techniques for Java-like Programs*, ACM Press, pp. 2:1–2:7, doi:10.1145/2489804.2489807.

[7] Krzysztof R. Apt (1997): *From logic programming to Prolog*. Prentice Hall International series in computer science, Prentice Hall.

[8] J. Jeannin, D. Kozen & A. Silva (2012): *CoCaml: Programming with Coinductive Types*. Technical Report, Computing and Information Science, Cornell University. Available at http://hdl.handle.net/1813/30798.

[9] J. Jeannin, D. Kozen & A. Silva (2013): *Language Constructs for Non-Well-Founded Computation*. In Matthias Felleisen & Philippa Gardner, editors: *ESOP 2013 - European Symposium on Programming*, *Lecture Notes in Computer Science* 7792, Springer, pp. 61–80, doi:10.1007/978-3-642-37036-6_4.

[10] John W. Lloyd (1987): *Foundations of Logic Programming, 2nd Edition*. Springer, doi:10.1007/978-3-642-83189-8.

[11] Luke Simon (2006): *Extending logic programming with coinduction*. Ph.D. thesis, University of Texas at Dallas.

[12] Luke Simon, Ajay Bansal, Ajay Mallya & Gopal Gupta (2007): *Co-Logic Programming: Extending Logic Programming with Coinduction*. In Lars Arge, Christian Cachin, Tomasz Jurdzinski & Andrzej Tarlecki, editors: *ICALP'07 - International Colloquium on Automata, Languages and Programming 2003*, *Lecture Notes in Computer Science* 4596, Springer, pp. 472–483, doi:10.1007/978-3-540-73420-8_42.

[13] Luke Simon, Ajay Mallya, Ajay Bansal & Gopal Gupta (2006): *Coinductive Logic Programming*. In Sandro Etalle & Miroslaw Truszczynski, editors: *International Conference on Logic Programming*, *Lecture Notes in Computer Science* 4079, Springer, pp. 330–345, doi:10.1007/11799573_25.

A Proofs

Proof of Lemma 3.3. By induction on inference rules.

(co-empty) We trivially conclude by rule (empty).

(co-step) By hypothesis $E_3 \cup E^{A=A'}$ is solvable, so by inductive hypothesis we get that $S \cup \{A'\} \vdash_{\langle P,C\rangle} \langle G_1, G_2 \,\square\, E_2\rangle \Rightarrow E_3'$ holds with $\mathsf{gsol}(E_3 \cup E^{A=A'}) \subseteq \mathsf{gsol}(E_3')$. Since $E_2 \subseteq E_3$, $E_2 \cup E^{A=A'}$ is solvable by hypothesis, by inductive hypothesis we get $S \cup \{A'\} \cup \{B'\} \vdash_{\langle P,C\rangle} \langle B_1, \ldots B_n \,\square\, E_1 \cup E^{B=B'}\rangle \Rightarrow E_2'$ with $\mathsf{gsol}(E_2 \cup E^{A=A'}) \subseteq \mathsf{gsol}(E_2')$. Note that $\mathsf{gsol}(E_3 \cup E^{A=A'}) \subseteq \mathsf{gsol}(E_2 \cup E^{A=A'}) \subseteq \mathsf{gsol}(E_2')$, therefore $\mathsf{gsol}(E_3 \cup E^{A=A'}) \subseteq \mathsf{gsol}(E_2') \cap \mathsf{gsol}(E_3')$, that implies $\mathsf{gsol}(E_3 \cup E^{A=A'}) \subseteq \mathsf{gsol}(E_2' \cup E_3')$. Therefore by Corollary 3.2 we get that $S \cup \{A'\} \vdash_{\langle P,C\rangle} \langle G_1, G_2 \,\square\, E_2'\rangle \Rightarrow E_3''$ holds with $E_3'' \subseteq E_2' \cup E_3'$, thus $\mathsf{gsol}(E_3 \cup E^{A=A'}) \subseteq \mathsf{gsol}(E_3'')$. Finally we get the thesis by applying rule (co-step).

(co-hyp) If the atom B unifies with an atom in S the the thesis follow by the inductive hypothesis applying rule (co-hyp).

Suppose that B unifies with A, that is $E_1 \cup E^{B=A}$ is solvable. By hypothesis $E_3 \cup E^{A=A'}$ is solvable, thus by inductive hypothesis we get $S \vdash_{\langle P,C \rangle} \langle G_1, G_2 \Box E_2 \rangle \Rightarrow E_3'$ with $\mathsf{gsol}(E_3 \cup E^{A=A'}) \subseteq \mathsf{gsol}(E_3')$. Since $E_2 \subseteq E_3$ we know that $\mathsf{gsol}(E_3 \cup E^{A=A'}) \subseteq \mathsf{gsol}(E_2 \cup E^{A=A'})$; moreover since $E^{B=A} \subseteq E_2$, we get that $\mathsf{gsol}(E_2 \cup E^{A=A'} \cup UnifyEqBA') = \mathsf{gsol}(E_2 \cup E^{A=A'})$, thus, by Corollary 3.1, we get that $\vdash_{P_{\sqcup C}} \langle B \Box E_1 \cup E^{B=A'} \rangle \Rightarrow E_2'$ with $\mathsf{gsol}(E_2 \cup E^{A=A'}) \subseteq \mathsf{gsol}(E_2')$. Also $\mathsf{gsol}(E_3 \cup E^{A=A'}) \subseteq \mathsf{gsol}(E_2 \cup E^{A=A'}) \subseteq \mathsf{gsol}(E_2')$, that implies $\mathsf{gsol}(E_3 \cup E^{A=A'}) \subseteq \mathsf{gsol}(E_2') \cap \mathsf{gsol}(E_3') \subseteq \mathsf{gsol}(E_2' \cup E_3')$. Hence by Corollary 3.2 we get that $S \cup \{A'\} \vdash_{\langle P,C \rangle} \langle G_1, G_2 \Box E_2' \rangle \Rightarrow E_3''$ holds with $\mathsf{gsol}(E_2' \cup E_3') \subseteq \mathsf{gsol}(E_3'')$. Finally we get the thesis by applying rule (co-hyp).

□

B Source code

```
:- module(meta_interpreter, [solve/1]).

:- use_module(library(assoc)). %%% needed to keep track of atom occurrences for finite
    failure

solve(Goal) :- solve_gfp([], Goal). %%% (main)

%%% solver for the inductive system with cofacts

solve_lfp(AtomAssoc, (Goal1, Goal2)) :- !, solve_lfp(AtomAssoc, Goal1), solve_lfp(
    AtomAssoc, Goal2). %%% seq
solve_lfp(_, Atom) :- predefined(Atom), !, Atom. %%% predef
solve_lfp(AtomAssoc, Atom) :- find(Atom, AtomAssoc, Count), (Count<2 -> clause(cofact(
    Atom), Body), Body; !, fail). %%% cut
solve_lfp(AtomAssoc, Atom) :- clause(Atom, Body), insert(Atom, AtomAssoc, NewAtomAssoc),
    solve_lfp(NewAtomAssoc, Body). %%% step

%%% solver for the coinductive system with no cofacts

solve_gfp(AtomList, (Goal1, Goal2)) :- !, solve_gfp(AtomList, Goal1), solve_gfp(AtomList,
     Goal2). %%% co-seq
solve_gfp(_, Atom) :- predefined(Atom), !, Atom. %%% co-predef
solve_gfp(AtomList, Atom) :- co_find(Atom, AtomList), !, empty_assoc(EmptyAssoc),
    solve_lfp(EmptyAssoc, Atom). %%% co-hyp
solve_gfp(AtomList, Atom) :- clause(Atom, Body), co_insert(Atom, AtomList, NewAtomList),
    solve_gfp(NewAtomList, Body). %%% co-step

%%% predefined predicates are interpreted in the standard way

predefined(Atom) :- predicate_property(Atom, built_in), !.
predefined(Atom) :- predicate_property(Atom, file(AbsPath)), file_name_on_path(AbsPath,
    library(_)), !.

%%% auxiliary predicates for the coinductive solver

co_find(Atom, AtomList) :- member(Atom, AtomList).

co_insert(Atom, AtomList, [Atom|AtomList]).

%%% auxiliary predicates for the inductive solver

%%% finds if AtomAssoc contains an AtomKey unifiable with Atom, but does not perform
    unification; if so, returns its corresponding hit count
%%% used by the inductive solver

find(Atom, AtomAssoc, Count) :- retrieve(Atom, AtomAssoc, AtomKey), get_assoc(AtomKey,
    AtomAssoc, Count).
```

```
%%% retrieve the AtomKey in AtomAssoc which is unifiable with Atom; no unification is
    performed
%%% used by found

retrieve(Atom, AtomAssoc, AtomKey) :- assoc_to_keys(AtomAssoc, AtomKeyList), get(Atom,
    AtomKeyList, AtomKey).

%%% checks if Atom is unifiable with an atom in AtomList; if so, returns such an atom. No
     unification is performed
%%% used by retrieve

get(Atom, [UnifiableAtom|_], UnifiableAtom) :- unifiable(Atom, UnifiableAtom, _), !.
get(Atom, [_|AtomList], UnifiableAtom) :- get(Atom, AtomList, UnifiableAtom).

%%% insertion in the list of visited atoms when implementing the inductive solver
%%% keeps track of how many times an atom has been hit

insert(Atom, AtomAssoc, NewAtomAssoc) :-
    retrieve(Atom, AtomAssoc, AtomKey) ->
        get_assoc(AtomKey, AtomAssoc, Counter), IncCounter is Counter+1, put_assoc(
            AtomKey, AtomAssoc, IncCounter, NewAtomAssoc);
        copy_term(Atom, CopiedAtom), put_assoc(CopiedAtom, AtomAssoc, 1, NewAtomAssoc).
```