Regular corecursion in Prolog☆

Davide Ancona

DIBRIS - Università di Genova,
Via Dodecaneso, 35
16146 Genova, Italy

Abstract

Corecursion is the ability of defining a function that produces some infinite data in terms of the function and the data itself, as supported by lazy evaluation. However, in languages such as Haskell strict operations fail to terminate even on infinite regular data, that is, cyclic data.

Regular corecursion is naturally supported by coinductive Prolog, an extension where predicates can be interpreted either inductively or coinductively, that has proved to be useful for formal verification, static analysis and symbolic evaluation of programs.

In this paper we use the meta-programming facilities offered by Prolog to propose extensions to coinductive Prolog aiming to make regular corecursion more expressive and easier to program with.

First, we propose an interpreter where the search tree is pruned to guarantee termination for certain kinds of predicate definition; then we introduce `finally` clauses, to provide a default value for all those cases where unification with a coinductive hypothesis is not correct. Finally, we propose a finer grain semantics where the user can specify only a subset of the arguments that have to be considered when coinductive hypotheses are unified.

The semantics defined by these vanilla meta-interpreters are an interesting starting point for a more mature design and implementation of coinductive Prolog.

Keywords: Logic programming, coinduction and corecursion

1. Introduction

Corecursion [6] has been used in some contexts to denote the ability, supported by lazy evaluation, of defining a function that produces some infinite data in terms of the function and the data itself.

☆This work has been partially supported by MIUR DISCO Distribution, Interaction, Specification, Composition for Object Systems.

As an example, let us consider the following Haskell code defining the infinite stream !0 : 1! : 2! : . . . of the factorials of all natural numbers.

```
fact_stream = 1:gen_fact 1 fact_stream
   where
      gen_fact n (m:l) = n*m:gen_fact (n+1) l
```

The stream `fact_stream` is defined in terms of itself and of the corecursive function `gen_fact` that takes the stream itself as one of its arguments.

An equivalent, but simpler definition is the following one:

```
fact_stream = 1:gen_fact 1 1
gen_fact  n m = let k = n*m in k:gen_fact k (m+1)
```

After having defined `fact_stream`, one can get the factorial of n by simply selecting the element at position n in `fact_stream`:

```
*Main> fact_stream !! 10
3628800
```

Though the stream is infinite, it is possible to access any arbitrary element because the list constructor '`:`' is non-strict and, hence, the call to function `gen_fact` is computed lazily. More abstractly, the data returned by `gen_fact` corresponds to a tree whose depth is infinite, and that is **not** *regular*; a regular tree can have infinite depth, but it is only allowed to have a finite set of subtrees. Trivially, finite trees are regular, whereas infinite regular trees can be effectively represented by finite cyclic data structures, without relying on lazily evaluated computations.

Now let us try to check whether all elements in the stream are greater than 0, with the predefined function **all**.

```
*Main> all (\x -> x>0) fact_stream
-- does not terminate
```

Checking that an arbitrary predicate holds on all the factorials of natural numbers is only semi-decidable: termination is guaranteed only if the predicate does not hold for some element, as in **all** `(\x -> x<100) fact_stream`.

Let us now consider this other stream declaration:

```
ones = 1:ones
```

Differently from `fact_stream`, stream `ones` is regular: it corresponds to a tree whose set of subterms contains just 1 and the stream itself. Such a stream is defined as a cyclic data structure, and no lazy evaluation is required: it is recursively defined by using just the list constructor.

Despite the regularity of `ones`, in Haskell the evaluation of the expression **all** `(\x -> x>0) ones` does not terminate; this happens because the logical conjunction `&&` is strict in its second argument (when the first argument evaluates to **True**), and **all** is defined in the standard inductive way.

Let us now consider the same problem in Prolog.[1] We can easily define predicate `all` s.t. `all(`p`,`l`)` succeeds iff predicate p is true for all elements of list l.

```
all(_,[]).
all(P,[X|L]) :- call(P,X),all(P,L).
positive(X) :- X>0.
```

The resolution of the goal `Ones=[1|Ones],all(positive,Ones)` does not terminate, for the same reason explained for Haskell. Modern Prolog interpreters support regular terms; the unification `Ones=[1|Ones]` succeeds, because occur-check is not performed, and `Ones` is substituted with the regular term corresponding to the cyclic list containing infinite occurrences of `1`. For instance, in SWI-Prolog the goal `?- Ones = [1|Ones]` succeeds with the answer `Ones = [1|Ones]`, that is, `Ones` is the unique regular tree which is solution of the equation `Ones = [1|Ones]`. Such a tree can be represented in infinite different ways; for instance, the following goal (where '`==`' is the built-in predicate corresponding to syntactic equality) succeeds as follows:

```
?- Ones=[1|Ones],OnesOnes=[1,1|OnesOnes],Ones==OnesOnes.
Ones=OnesOnes,OnesOnes=[1,1|OnesOnes].
```

If predicate `all` is interpreted in the standard inductive way, then the Prolog interpreter tries to build an infinite derivation for the goal and, thus, the computation does not terminate. The conventional inductive interpretation of a logic program is based on the *inductive Herbrand model*, that is, the least fixed point of the one-step inference operator defined by the clauses of the program. This can be proved equivalent to the set of all ground atoms for which there exists a finite SLD derivation.

Simon et al. [13, 15, 14] have proposed coinductive SLD resolution (abbreviated by coSLD) as an operational semantics for logic programs interpreted coinductively: the coinductive Herbrand model is the greatest fixed-point of the one-step inference operator. This can be proved equivalent to the set of all ground atoms for which there exists either a finite or an infinite SLD derivation [15, 9].

Coinductive logic programming has proved to be useful for formal verification [10, 12], static analysis and symbolic evaluation of programs [3, 2, 4].

CoSLD resolution is not computable in its general form, but it becomes implementable when restricted to the fragment where only regular terms and regular derivations are allowed. In SWI-Prolog the library `coinduction` has been recently introduced, to allow coinductive interpretation of predicates. For instance, we want predicate `all` to be interpreted coinductively:

```
:- use_module(library(coinduction)).
:- coinductive(all/2).
all(_,[]).
```

[1] All Prolog examples shown in the paper have been tested with SWI-Prolog, version 6.0.2.

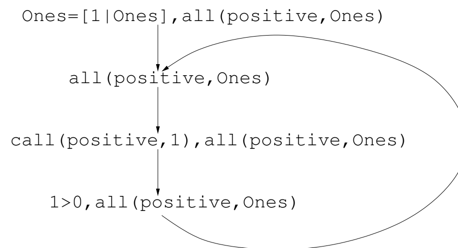

Figure 1: Regular derivation for the goal `Ones=[1|Ones],all(positive,Ones)`

```
all(P,[X|L]) :- call(P,X),all(P,L).
positive(X) :- X>0.
```

Now the resolution of our goal terminates succesfully:

```
?- cosld((Ones=[1|Ones],all(positive,Ones))).
Ones = [1|Ones].
```

This happens because the interpreter is able to build an infinite but regular derivation, as depicted in Figure 1, by unifying the atom `all(positive,Ones)`, that has to be solved, with the ancestor atom that have been already solved.

In Haskell a function with the same behavior cannot be implemented so simply; for instance, specific datatypes have to be expressly defined and used [16, 7], or physical equality has to be exploited, at the cost of breaking referential transparency, and other nice properties.

Regular corecursion is expressly suited for cyclic data structures; among them, the most commonly used are certainly graphs, a kind of data structure that is heavily employed in many application domains; we will see some non trivial examples of corecursion with graphs in Section 3 and 4. Regular corecursion allows elimination of the boilerplate code needed for manual bookkeeping of inspected data in a cyclic structure, thus making code simpler, and more reliable, and favoring coinductive reasoning; furthermore, programming abstractions for regular corecursion promote code reuse, and optimization; roughly, recursion compares to iteration with a stack, as regular corecursion compares to ordinary recursion with a set (that is, a data structure implementing the abstract data type set).

In this paper we propose new semantics and programming abstractions to make regular corecursion more expressive, and easier to program with.

In the next section we will present three different semantics of regular corecursion in Prolog, by means of simple vanilla interpreters. Besides the semantics coinciding with the original definition of coSLD [15], two optimized versions are presented: the former is complete w.r.t. the original semantics, whereas the latter is not, but it ensures termination of coSLD under certain assumptions; furthermore, completeness is still guaranteed for some forms of goals.

In Section 4 we extend our interpreters by adding `finally` clauses, to enhance the expressivity of regular corecursion; such a feature allows a simpler

definition for some coinductive predicates (for instance, those that correspond to existentially quantified properties).

In Section 5 we introduce yet another feature aiming to make regular corecursion even more expressive and flexible to be used: the user can select the arguments that have to be considered in the coinductive definition of a predicate. Also in this case, some meaningful examples show the usefulness of such an extension.

Finally, Section 6 draws conclusions, and outlines further interesting research directions.

2. Meta-interpreters for coSLD

This section elaborates previous results [13] by defining two different versions of a vanilla meta-interpreter (where vanilla means based on built-in unification and predicate **clause/2**) supporting coSLD. Even though vanilla meta-interpreters are not efficient to be suitable for practical uses, the meta-programming facilities offered by Prolog are an ideal tool to experiment with new semantics and programming abstractions: vanilla meta-interpreters are concise and abstract enough to serve as a formal semantics, yet they provide prototype implementations to test new language features.

We first define a basic meta-interpreter, and then extend it to allow resolution of built-in and library predicates, mixing of coinductive and inductive predicates, and elimination of repeated answers.

2.1. Basic meta-iterpreters

The basic meta-interpreter corresponding to the original formulation of the coSLD operational semantics [13] is a straightforward extension of the conventional vanilla interpreter implementing standard SLD resolution for Prolog.

```
:- module(cosldmeta0,[cosld/1]).

:- use_module(library(ordsets)).

cosld(G) :- ord_empty(E),solve(E,G).
solve(H, (G1,G2)) :- !,solve(H, G1), solve(H,G2).
solve(_,true) :- !.
solve(H,A):- found(A, H).
solve(H,A):- !,clause(A,As),insert(H,A,NewH),solve(NewH,As).

insert(L1,A,L2) :- !,ord_add_element(L1,A,L2).

found(A,H) :- member(A,H).
```

Module `cosldmeta0` exports the main predicate `cosld/1` which takes a goal as argument, and solves it according to the coSLD operational semantics, which is implemented through the predicate `solve`.

The first argument of `solve` is the ordered set of ancestor atoms that have been already solved (hence the set is initially empty), which are called *coinductive hypotheses*, whereas the second argument is the goal that have to be solved. The set of coinductive hypotheses contains all atoms that the interpreter has been solved so far, and are needed for building infinite regular derivations (see below); such a set is implemented as a list with unique elements sorted to the standard order of terms, by using the library `ordsets`.

The first two clauses for `solve` deal with goals having more than one atoms and with the empty goal, respectively, while the remaining clauses manage the most interesting case when the goal contains just one atom. To solve an atom `A` the interpreter first tries to build an infinite regular derivation by searching for an atom in the coinductive hypotheses `H` that unifies with `A` (`found(A,H)`); for this simple version of the meta-interpreter predicate `found` coincides with the library predicate `member` that checks whether an element is a member of a list. If the search succeeds, then the atom is solved and removed from the goal, and the computed answer substitution is refined accordingly, since the atom `A` is unified with the coinductive hypothesis found in `H`.[2]

If no unifiable coinductive hypothesis can be found, then a clause in the program whose head unifies with the current atom is searched with the built-in predicate **clause**; if such a clause is found, then the unified body `As` of the clause is solved in the new set of coinductive hypotheses `NewH` where the atom `A` unified with the body of the clause has been added; for this simple version of the meta-interpreter predicate `insert` coincides with predicate `ord_add_element` defined in library `ordsets`, which inserts an element into an ordered set.

We show how the interpreter works with a very simple example program defining the predicate `is_nat`.

```
is_nat(z).
is_nat(s(N)) :- is_nat(N).
```

In this case, the only difference with inductive Prolog is that `is_nat` succeeds also when `N = s(N)`. The resolution of the goal `cosld(is_nat(N))` (corresponding to the coSLD resolution of the goal `is_nat(N)`) returns the following infinite sequence of answers (we will consider shortly the problem of avoiding some redundant answers):

```
N = z ;
N = s(N) ;
N = s(z) ;
N = s(_S1), % where
    _S1 = s(_S1) ;
N = s(s(N)) ;
N = s(s(z)) ;
...
```

[2]The atom `member(A,H)` succeeds iff there exists an atom in `H` unifying with `A`.

This very basic meta-interpreter has two serious restrictions: built-in predicates cannot be used, since the **clause** predicate does not work with them; all predicates are interpreted coinductively, whereas there are cases where we may want the standard inductive interpretation for predicates (consider, for instance, library predicates). To this aim, we introduce two predicates `inductive` and `coinductive` to partition predicates: predicates are inductive by default, those coinductive (and necessarily user-defined) have to be explicitly specified by the user. Therefore the inductive predicates are either built-in or imported from a standard library or they have not been declared coinductive.

```
:- module(cosldmeta0,[cosld/1]).

:- use_module(library(ordsets)).

cosld(G) :- ord_empty(E),solve(E,G).
solve(H, (G1,G2)) :- !,solve(H, G1), solve(H,G2).
solve(_,A) :- inductive(A),!,A.
solve(H,A):- found(A, H).
solve(H,A):- !,clause(A,As),insert(H,A,NewH),solve(NewH,As).

inductive(A) :- predicate_property(A,built_in),!.
inductive(A) :- predicate_property(A,file(AbsPath)),
                file_name_on_path(AbsPath,library(_)),!.
inductive(A) :- !,\+ coinductive(A).

insert(L1,A,L2) :- !,ord_add_element(L1,A,L2).

found(A,H) :- member(A,H).
```

If an atom is inductive, then it is directly solved by the Prolog interpreter; the cut allows the meta-interpreter to skip the clauses dealing with coinduction. Since **true** is a built-in predicate, the clause for the empty goal is no longer required. This solution enforces a stratification between coinductive and inductive predicates: while a coinductive predicate can be defined in terms of an inductive one, the opposite is not allowed; this restriction avoids contradictions due to naive mixing of coinduction and induction [14].

To allow regular coSLD resolution for predicate `all`, as defined in the previous section, we only need to declare it to be coinductive.

```
:- use_module(cosldmeta0).

coinductive(all(_,_)).

all(_,[]).
all(P,[X|L]) :- call(P,X),all(P,L).
positive(X) :- X>0.

test :- cosld((Ones=[1|Ones],all(positive,Ones))).
```

As expected, the goal `test` succeeds.

```
?- test.
true .
```

2.2. Avoiding redundant coinductive hypotheses

We extend the basic meta-interpreter to allow elimination of repeated answers due to redundant coinductive hypotheses.

The basic meta-interpreter computes set of redundant coinductive hypotheses, as shown by the resolution of the goal `cosld(is_nat(N))`: initially the set of coinductive hypotheses is empty, the first clause for `is_nat` is applicable, and the first computed answer is `N=z`; if backtracking is forced, then the second clause for `is_nat` is considered, the substitution `N=s(N0)` is computed, and the goal `is_nat(N0)` is solved, with the set of coinductive hypotheses `[is_nat(s(N0))]`.

Since `is_nat(N0)` unifies with the unique coinductive hypothesis, the meta-interpreter can build an infinite regular derivation whose answer is `N=s(N)`. Proceeding further, the meta-interpreter re-applies the first clause for `is_nat`, to get the answer `N=s(z)`, and then re-applies the second clause for `is_nat`; the substitution `N0=s(N1)` is computed, and the goal `is_nat(N1)` is solved, with the set of coinductive hypotheses `[is_nat(s(N1),is_nat(s(s(N1)))]`. At this point the insertion of atom `is_nat(s(N1)` in the set of coinductive hypotheses is redundant, since it unifies with the atom `is_nat(s(s(N1)))` already present in the set. However, predicate `ord_add_element` works by syntactic equality, therefore `is_nat(s(N1)` and `is_nat(s(s(N1)))` are considered different elements, hence the atom is inserted.

As a consequence of such a redundancy, the atom

```
member(is_nat(N1),[is_nat(s(N1)),is_nat(s(s(N1)))])
```

succeeds twice, the first time with the answer `N = s(_S1), _S1 = s(_S1)`, the second time with the equivalent answer `N = s(s(N))`; this happens because `member` allows backtracking. One may solve this problem by simply using `memberchk` which does not perform backtracking; however, inserting non unifiable atoms allows to keep the set of coinductive hypotheses smaller.

To avoid this problem, we modify the meta-interpreter: a new coinductive hypothesis is inserted in the set only if it does not unify with any other ancestor atoms in the set. Furthermore, predicate `find` is defined by `memberchk`.

```
:- module(cosldmeta1,[cosld/1]).

:- use_module(library(ordsets)).

cosld(G) :- ord_empty(E),solve(E,G).
solve(H, (G1,G2)) :- !,solve(H, G1), solve(H,G2).
solve(_,A) :- inductive(A),!,A.
solve(H,A):- found(A, H).
solve(H,A):- !,clause(A,As),insert(H,A,NewH),solve(NewH,As).

inductive(A) :- predicate_property(A,built_in),!.
inductive(A) :- predicate_property(A,file(AbsPath)),
```

```
            file_name_on_path(AbsPath,library(_)),!.
inductive(A) :- !,\+ coinductive(A).

insert(L1,X,L1) :- is_in(X,L1),!.
insert(L1,X,L2) :- !,ord_add_element(L1,X,L2).

is_in(E,[X|_]) :- unifiable(E,X,_),!.
is_in(E,[_|L]) :- is_in(E,L),!.

found(A,H) :- memberchk(A,H).
```

In this way the set of coinductive hypotheses is kept smaller, and some repeated answers are avoided. Now `found` is defined by `memberchk`, thus making the searching more efficient. With this new version of the meta-interpreter the goal `cosld(is_nat(N))` yields the following answers:

```
N = z ;
N = s(N) ;
N = s(z) ;
N = s(s(N)) ;
N = s(s(z)) ;
...
```

2.3. A pruning meta-interpreter

Let us consider the following logic program:

```
:- use_module(cosldmeta1).

coinductive(lth(_,_)).

lth(tree(N1,LT1,RT1),tree(N2,LT2,RT2)) :-
    N1<N2,lth(LT1,LT2),lth(RT1,RT2).
```

Atom `lth`(t_1,t_2) should succeed if and only if t_1 and t_2 are two infinite regular complete binary trees, where nodes are intenger numbers, such that each node in t_1 is less than the corresponding node in t_2. Although the coinductive definition of `lth` is correct, the program fails to terminate for some goals whose resolution should fail; for instance, this happens for the goal

```
?- T1=tree(4,T1,tree(5,T1,T1)),T2=tree(5,T2,tree(4,T2,T2)),
    cosld(lth(T1,T2)).
```

After one resolution step we get the goal

```
?- 4<5,lth(T1,T2),lth(tree(5,T1,T1),tree(4,T2,T2)).
```

Resolution of the first atom trivially succeeds, the second atom succeeds as well by coinductive hypothesis, whereas the third atom fails, therefore, by backtracking, resolution of `lth(T1,T2)` is tried again, thus entering an infinite loop with the following goal

```
?- 4<5,lth(T1,T2),lth(tree(5,T1,T1),tree(4,T2,T2)),
    lth(tree(5,T1,T1),tree(4,T2,T2)).
```

An ad hoc solution to this problem would consist in inserting a cut in the body of the clause, between the two atoms `lth(LT1,LT2)` and `lth(RT1,RT2)`; here, instead, we propose a pruning version of the meta-interpreter, to avoid this kind of non terminating failures.

A pruning of the search trees can be performed by applying a clause only if the atom to be solved does not unify with a coinductive hypothesis, after it has been unified with the head of the clause.

```
:- module(cosldmeta2,[cosld/1]).

:- use_module(library(ordsets)).

cosld(G) :- ord_empty(E),solve(E,G).
solve(H, (G1,G2)) :- !,solve(H, G1), solve(H,G2).
solve(_,A) :- inductive(A),!,A.
solve(H,A):- found(A, H).
solve(H,A):- !,clause(A,As),insert(H,A,NewH),solve(NewH,As).

inductive(A) :- predicate_property(A,built_in),!.
inductive(A) :- predicate_property(A,file(AbsPath)),
                file_name_on_path(AbsPath,library(_)),!.
inductive(A) :- !,\+ coinductive(A).

insert(L1,A,L2) :- is_in(A,L1) -> fail;ord_add_element(L1,A,L2).

is_in(A1,[A2|_]) :- unifiable(A1,A2,_),!.
is_in(A,[_|L]) :- is_in(A,L),!.

found(A,H) :- memberchk(A,H).
```

With such an interpreter, the set of computed answers can be considerably restricted in some cases. For instance, the resolution of the goal `cosld(is_nat(N))` only yields three possible answers.

```
?- cosld(is_nat(N)).
N = z ;
N = s(N) ;
N = s(z) .
```

However, the resolution of all ground goals having shape `is_nat(`s^n`(z))` still succeeds.

Though this version of the meta-interpreter is not suitable for programs based on generate-and-test patterns, in the next section we will show several examples where pruning is very effective and useful.

3. Examples of use of regular corecursion

In this section we consider more significant examples of regular corecursion; some of them will show the usefulness of pruning, some others will be used to motivate the features that will be introduced in the next sections. We also show how constraint logic programming can be usefully exploited in conjunction with regular corecursion.

3.1. Membership for regular lists

In the previous section we have shown how predicate `all` can be easily defined corecursively in Prolog for regular lists; more generally, this is true whenever predicates corresponding to universally quantified properties have to be defined on regular terms. However, properties which are existentially quantified on cyclic data cannot be defined so easily; a classical example is given by the `member` predicate for regular lists [13].

```
coinductive(member(_,_)).
member(N,[N|_]).
member(N1,[N2|L]) :- member(N1,L).
```

At a first glance the definition above may seem correct, but a more thorough analysis reveals that this is not true. For instance, the resolution of goal `cosld(L=[1,2,3|L],member(5,L))` succeeds, whereas it should fail. Indeed, after three steps the initial goal is found again, and resolution succeeds by coinductive hypothesis; in fact, all ground atoms built with predicate `member` always succeeds.

To avoid this problem, one might define the complemented predicate `not_member`, which corresponds to a universally quantified property; unfortunately, this approach has the drawback that it requires coSLD negation [11].

An alternative solution is given by the following program.

```
coinductive(member(_,_)).
coinductive(aux_member(_,_,_)).

member(N,L) :- aux_member(N,L,_).
aux_member(N,[N|_],t).
aux_member(N1,[N2|L],R2) :-
    aux_member(N1,L,R1),R1==t,R2=t.
```

The coinductive auxiliary predicate `aux_member` has a third argument corresponding to the Boolean result of the membership test; such an argument is not used by the main predicate `member`, but it is necessary for ensuring a correct behavior. The body of `member` ensures that initially the last argument of `aux_member` is a logical variable; this corresponds to the fact that the truth value is initially unknown. If the element is found (first clause), then the variable is unified with the constant `t`, representing **true**. If the initial list is found again, then the goal is resolved by coinduction hypothesis, but the variable corresponding to the result is not substituted; hence, atom `aux_member(N1,L,R1)`

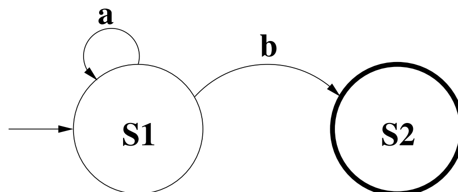

Figure 2: A deterministic finite automaton recognizing the language `a*b`

in the body of the second clause for `aux_member` always succeeds, but `R1` is substituted with `t` if and only if the element has been found. For this reason, the atom is followed by the syntactic equality test `R1==t`.

Such a solution is correct, providing that the pruning meta-interpreter is used: goals like `cosld(L=[1,2,3|L],member(5,L))` correctly fail only with pruning, otherwise the second clause of `aux_member` is selected infinitely many times.

In Section 4 a new feature, called `finally` clause, is expressly introduced to overcome ad hoc and rather involved definitions for predicates as `member`.

3.2. Finite automata and regular languages

We now consider a classical application from formal languages, by defining a predicate that succeeds iff a finite automaton (either deterministic or not) accepts all strings of a regular language (defined by an extended right linear grammar). In other words, the predicate succeeds iff the language defined by the grammar is a subset of the language defined by the automaton.

Regular terms allow a very compact representation of automata and regular grammars.[3] Let us consider the automaton depicted in Figure 2, where S1 (pointed by the arrow) is the initial state, and S2 (with a thicker circle) is final.

Such an automaton can be represented by the following cyclic Prolog term associated with the logical variable `S1`:

```
S1=state(notfinal,[(a,S1),(b,S2)]),S2=state(final,[])
```

Each state is represented by the term `state(`k`,`e`)`, where k can be one of the two constants `final` and `notfinal`, and e is the list of outgoing edges, represented by pairs (σ,S), where σ is a symbol of the alphabet of the automaton, and S is one of its states. Since an automaton has only an initial state, there is no need to explicitly represent initial states. If we consider the definition above, then `S1` is associated with the term corresponding to the automaton in Figure 2, whereas `S2` is associated with a term corresponding to another automaton, where S2 is

[3] In the example we consider extended right linear grammars since acceptance by a finite automaton can be defined more easily, and the standard Prolog constructors for lists can be suitably used for representing them.

both an initial and a final state (such an automaton accepts only the empty string).

Let us now consider the following right linear grammar: `A ::= b | aA`.

Using the Prolog constructors for list, and a binary operation **or** for expressing alternative productions, we can easily represent such a grammar by the following cyclic Prolog term: `A = or([b],[a|A])`.

It is not difficult to define two functions that are inspired by the two examples above, and that encode finite automata and extended right linear grammars into cyclic Prolog terms. We are now ready for defining the predicate `accept`.

```
coinductive(accept(_,_)).

accept(state(final,_),[]).
accept(state(_,E),[H|T]):-member((H,S),E),accept(S,T).
accept(S,or(L1,L2)) :- accept(S,L1),accept(S,L2).
```

The clauses show that using the list constructors for representing our grammars has an advantage: strings (that is, sequences of symbols), are considered as particular cases of grammars (defining just a single string), in the same way as an element can be identified with the singleton set containing it. The first two clauses define whether an automaton accepts a given string: any final state accepts the empty string, whereas the non empty string `[H|T]` is accepted from the state `state(_,E)` if there exists an outgoing edge labeled with `H` and pointing to a state `S` starting from which the tail `T` of the string can be accepted. If we consider just strings (that is, finite lists), then corecursion is not needed, since termination is guaranteed by the induction on strings. For instance, the following two goals can be resolved without the predicate `cosld` (obviously resolution for the former succeeds, whereas it fails for the second).

```
S1=state(notfinal,[(a,S1),(b,S2)]),S2=state(final,[]),
   accept(S1,[a,b]).
S1=state(notfinal,[(a,S1),(b,S2)]),S2=state(final,[]),
   accept(S1,[b,a]).
```

The third clause dealing with alternatives is self-explanatory: the union `or(L1,L2)` of the languages `L1` and `L2` is accepted if both languages are accepted starting from the state `S`. To verify that all strings generated by the grammar `A ::= b | aA` are accepted by our automaton we need regular corecursion, since the term representing the grammar is cyclic:

```
cosld((
   S1=state(notfinal,[(a,S1),(b,S2)]),
   S2=state(final,[]),A=or([b],[a|A]),
   accept(S1,A))).
```

To avoid infinite failure, we need to run the pruning version of the meta-interpreter. The resolution of the following goal terminates and fails, as expected, only if the pruning meta-interpreter is used.

```
cosld((
   S1=state(notfinal,[(a,S1),(b,S2)]),
```

```
   S2=state(final,[]),A=or([a|A],or([b|A],[b])),
   accept(S1,A))).
```

Clearly, the grammar `A ::= aA | bA | b` generates (among infinite others) the string `bb` which is not accepted by our automaton.

The careful reader may have noticed that the definition of `accept` is not completely correct, since it does not correctly manage the corner case when a grammar generates the empty set. Consider for instance the following two goals:

```
colsd((
   S1=state(notfinal,[(a,S1),(b,S2)]),
   S2=state(final,[]),A=[a|A],
   accept(S1,A))).
colsd((
   S1=state(notfinal,[(a,S1),(b,S2)]),
   S2=state(final,[]),A=[c|A],
   accept(S1,A))).
```

The former succeeds, while the second fails, even though both should succeed, since the two grammars `A ::= aA` and `A ::= cA` generate the empty set. To overcome this problem, we introduce the coinductive predicate `empty` checking whether a grammar generates the empty set, and add a clause for dealing with this corner case.

```
coinductive(accept(_,_)).
coinductive(empty(_)).
accept(_,L) :- empty(L).
accept(state(final,_),[]).
accept(state(_,E),[H|T]):-member((H,S),E),accept(S,T).
accept(S,or(L1,L2)) :- accept(S,L1),accept(S,L2).
empty([_|T]) :- empty(T).
empty(or(L1,L2)) :- empty(L1),empty(L2).
```

The definitions of the two predicates are extremely concise and simple to understand. The concatenation of a symbol with a set of strings `T` is empty iff `T` is empty, and the union **or**`(L1,L2)` of `L1` and `L2` is empty iff both `L1` and `L2` are empty. The definition works because `empty` is interpreted coinductively, and it fails (as expected) on the empty list (which represents the singleton set containing the empty string).

3.3. Repeating decimals

It is well-known that every rational number is either a terminating or repeating decimal, that is, all rational numbers can be represented by an infinite regular lists of digits.

In the sequel we only consider rational numbers in the interval $[0,1]$ represented with base 10; all clauses shown in this section can be generalized in a straightforward way to deal with the whole set of rational numbers, represented with any base (≥ 2).

For instance, the term associated with `N` in `N=[5|P],P=[7,2|P]` corresponds to the repeating decimal $0.5\overline{72}$ that equals the fraction $\frac{63}{110}$. Indeed, multiplying

a repeating decimal by 10^e (with $e > 0$) is equivalent to a left shift of e positions, therefore we have that the following equations hold:

```
100P=72+P
10N=5+P.
```

From the two equations above we can derive

$$\mathtt{P}=\frac{72}{99}=\frac{8}{11}$$

$$\mathtt{N}=\frac{1}{2}+\frac{4}{55}=\frac{55+8}{110}=\frac{63}{110}$$

For uniformity, we represent terminating decimals as infinite regular lists as well (by definition, a decimal is terminating if it has a repeating final 0). For instance, 0.5 is represented by the term associated with `N` in `N=[5|Z],Z=[0|Z]`.

We can now define a coinductive predicate to compute the addition between two repeating decimals represented as infinite regular lists of digits. Since the operands have infinite digits, we cannot simply mimic the conventional algorithm for addition, because the notion of least significant digit does not make sense in our case. We first consider a simple solution which consists in using an auxiliary predicate that computes digit-wise result and carry for all positions, and returns two corresponding regular lists.

```
coinductive(aux_add(_,_,_,_)).
aux_add([D1|N1],[D2|N2],[RD|R],[CD|C]) :-
   Sum is D1+D2, RD is Sum mod 10,
   CD is Sum // 10, aux_add(N1,N2,R,C).
```

The predicate takes two operands `[D1|N1]` and `[D2|N2]`, computes the addition `RD` and the carry `CD` for the two most significant digits `D1` and `D2`, and then continues corecursively for the rest of the digits `N1` and `N2`.

We can now define the main predicate `add`.

```
coinductive(add(_,_,_,_)).
add(O1,O2,R,CD) :-
   O2\=[0|O2],aux_add(O1,O2,PR,[CD1|C]),
   add(PR,C,R,CD2),CD is CD1 + CD2.
add(O1,Z,O1,0) :- Z=[0|Z].
```

If the second operand is zero (second clause), then the result is the first operand `O1`, and the carry digit for the next more significant position is 0.

Otherwise (first clause) the partial result `PR` and all carry digits `[CD1|C]` of the addition `O1`+`O2` are computed with `aux_add`; then we have to accommodate the carry digits: first they need to be left shifted of one position (thus we get `C`). Indeed, the carry digit generated at position i (corresponding to 10^i, with $i < -1$) must be added to the digit of the partial result `PR` at position $i+1$. Therefore the addition between `PR` and `C` is computed, to get the final result `R` and a carry digit `CD2` that has to be combined with the most significant digit `CD1` of the carry digits computed by `aux_add`, to get the carry digit `CD` corresponding to the next more significant position.

The computation terminates because of regularity, and because each position can yield a carry of 1 just once; actually, `add` (but not `aux_add`) is defined by induction, but since it depends from a coinductive predicate, stratification (recall Section 2) requires `add` to be interpreted coinductively as well.

In the next section we will see how `finally` clauses allow a simpler solution that does not require the use of an auxiliary predicate. A simpler and also more efficient solution can be obtained by exploiting constraints over finite domains; the following example shows how constraint logic programming fits well with regular corecursion.

```
:- use_module(library(clpfd)). % finite domain CLP
coinductive(add(_,_,_,_)).
add([D1|N1],[D2|N2],[RD|R],C) :-
    add(N1,N2,R,PC), PC in 0..1, Sum #= D1 + D2 + PC,
    RD #= Sum mod 10, C #= Sum / 10, label([RD]).
```

With constraints, propagation of the evaluation of integer expressions can proceed in both directions, therefore we can avoid using `aux_add` and define `add` coinductively with just one clause.

To compute `[D1|N1]`+`[D2|N2]`, `N1`+`N2` is first computed, yielding the result `R`, and the carry `PC` that must be added to the most significant digits `D1` and `D2` to compute the most significant digit `RD` of the result, and the carry `C` for the next more significant position.

Since the predicate is coinductive, the atom `add(N1,N2,R,PC)` can be placed indifferently before or after all constraints; the atom `label([RD])` is required for obtaining ground solutions (all values for the finite domain variable `RD` are systematically tried out), since in some cases there exist two different solutions. For instance, let us consider the addition $0.0\overline{8} + 0.0\overline{1}$:

```
?- cosld((N1=[0|Eights],N2=[0|Ones],Eights=[8|Eights],
    Ones=[1|Ones],add(N1,N2,R,O))).
R = [1|_S1], % where
     _S1 = [0|_S1],
O = 0 ;
R = [0|_S1], % where
     _S1 = [9|_S1],
O = 0 ;
false.
```

The meta-interpreter finds two different solutions (clearly equivalent): $0.1\overline{0}$ with carry 0, or $0.0\overline{9}$ with carry 0.

As a final remark, both definitions (with or without constraints) work with both versions of the meta-interpreter, with the only difference that the pruning version does not return redundant answers.

4. Extending regular corecursion with the `finally` clause

In Section 3 we have seen that properties which are existentially quantified on cyclic data (as membership for regular lists) cannot be defined easily in a coinductive way, and a rather involved and ad hoc solution has been proposed.

Here we propose a new feature aiming to solve this problem, by allowing the user to define the specific behavior of a predicate when an atom is solved by coinductive hypothesis, by means of `finally` clauses.

While facts are used in Prolog for defining the base cases for induction, `finally` clauses specify the behavior in case of application of the coinductive hypothesis in regular coinduction.

Let us first introduce `finally` clauses with the definition of predicate `member`.

```
:- use_module(cosldmeta2finally).

coinductive(member(_,_)).

member(N,[N|_]).
member(N,[_|L]) :- member(N,L).
finally(member(_,_)) :- fail.
```

In the case of `member` the coinductive hypothesis is applied when all the elements of the cyclic list has been already inspected; this means that none of them was found equal to the first argument, therefore in this case the goal must fail. The last clause with `finally` is used for specifying such a behavior: when a coinductive hypothesis can be applied for `member` (independently of the arguments), then the goal must fail.

The semantics of `finally` clauses is specified by the following meta-interpreter, which is an extension of the pruning meta-interpreter presented in Section 2; the non pruning version can be extended in a very similar way.

```
:- module(cosldmeta2finally,[cosld/1]).

:- use_module(library(ordsets)).

cosld(G) :- ord_empty(E),solve(E,G).
solve(H, (G1,G2)) :- !,solve(H, G1), solve(H,G2).
solve(_,A) :- inductive(A),!,A.
solve(H,A):- found(A, H),(clause(finally(A),As) *-> solve(H,
    As);true).
solve(H,A):- !,clause(A,As),insert(H,A,NewH),solve(NewH,As).

inductive(A) :- predicate_property(A,built_in),!.
inductive(A) :- predicate_property(A,file(AbsPath)),
                file_name_on_path(AbsPath,library(_)),!.
inductive(A) :- !,\+ coinductive(A).

insert(L1,A,L2) :- is_in(A,L1) ->
    fail;ord_add_element(L1,A,L2).

is_in(A1,[A2|_]) :- unifiable(A1,A2,_),!.
is_in(A,[_|L]) :- is_in(A,L),!.

found(A,H) :- memberchk(A,H).
```

For keeping the treatment simple, `finally` is managed as a predicate symbol, hence in this approach `finally` cannot be chosen for naming user-defined predicates; in a real implementation this limitation can be easily avoided with a suitable syntax.

In comparison with the pruning meta-interpreter defined in Section 2, the only difference is the definition of the third clause for `solve/2`, which deals with the application of the coinductive hypothesis. If the current atom `A` unifies with some coinductive hypothesis in `H` (that is, `found(A,H)` succeeds), then the meta-interpreter checks whether there exists a `finally` clause applicable for `A`, with the atom **clause**`(finally(A),As)`; if it is the case, then the body of the corresponding `finally` clause is solved; if no `finally` clause is found, then the default behavior is implemented: the atom `A` succeeds.

The built-in predicate `*->` has been used instead of `->`, to allow backtracking for the resolution of **clause**`(finally(A),As)`; we will see in the sequel why backtracking for the `finally` clause may be useful.

We conclude this section with two examples showing more advanced uses of the `finally` clause.

Maximum of a regular list

We define the predicate `max/2` to compute the greatest element of a regular (hence, possibly cyclic) list of integers.

```
:- use_module(cosldmeta2finally).

coinductive(max(_,_)).
coinductive(aux_max(_,_,_)).

max([N|L],M) :- aux_max(L,N,M).

aux_max([],N,N).
aux_max([N1|L],N2,M) :- (N1 > N2 -> N3 = N1; N3 = N2),
    aux_max(L,N3,M).
finally(aux_max(_,N,N)).
```

The main predicate `max/2` is defined in terms of the auxiliary predicate `aux_max/3` where the second argument is used as an accumulator to store the maximum value computed so far.

As usual, the predicate correctly works for both cyclic (that is, coinductive) and non cyclic (that is, inductive) lists.

If the list is not cyclic, then the empty list is eventually reached, and then the returned value is the current value of the accumulator (fact `aux_max([],N,N)`).

If the list is cyclic, then the atom eventually unifies with a coinductive hypothesis, and then the returned value is the current value of the accumulator (`finally` clause).

For instance, the following atoms succeed:

```
?- L=[1,2,3,2,1|L],cosld(max(L,M)).
L = [1, 2, 3, 2, 1|L],
```

```
M = 3 .

?- L=[1,2,3,2,1],cosld(max(L,M)).
L = [1, 2, 3, 2, 1],
M = 3.
```

Note that, in the worst case, if the cyclic list has period n, then $2n$ elements have to be inspected to return the correct value; indeed, to be able to apply the coinductive hypothesis the second argument to `aux_max` must first become the maximum of the whole list. This limitation will be overcome in the next section, with a more expressive semantics allowing the user to have more control on the application of coinductive hypotheses.

Finally, we point out that the following direct definition for `max` is not correct.

```
:- use_module(cosldmeta2finally).

coinductive(max(_,_)).

max([N],N).
max([N|L],M) :- max(L,M1),(N > M1 -> M = N; M = M1).
finally(max([N|_],N)).
```

For instance, the following goal fails:

```
?- L=[1,2,3,2,1|L],cosld(max(L,M)).
false.
```

Indeed, as happens in this case, the first element of the repeated pattern of a cyclic list is not guaranteed to be the maximum of the whole list. Again, the more expressive semantics proposed in the next section allows us to make the definition above correct, with minimal changes.

Addition between repeating decimals

With `finally` clauses, the definition of predicate `add/4`, as given in Section 3 (without constraints), can be significantly simplified.

```
:- use_module(cosldmeta2finally).

coinductive(add(_,_,_,_)).

add([D1|R1],[D2|R2],[Sd|S],O) :- add(R1,R2,S,C),Sum is D1 +
    D2 + C,Sd is Sum mod 10,O is Sum // 10.
finally(add(_,_,_,0)).
finally(add(_,_,_,1)).
```

Such a definition works correctly because the meta-interpreter allows backtracking for `finally` clauses. When the coinductive hypothesis is eventually applied, the only unknown value is the final carry, which, however, must be either 0 or 1. As already shown in Section 3, there are cases where both values of the carry are correct as happens in the following example of goal:

```
?- cosld((N1=[0|Eights],N2=[0|Ones],Eights=[8|Eights],Ones
    =[1|Ones],add(N1,N2,R,O))).
R = [0|_S1], % where
     _S1 = [9|_S1],
O = 0 ;
R = [1|_S1], % where
     _S1 = [0|_S1],
O = 0 ;
false.
```

5. Argument annotations for corecursion

In this section we further extend the semantics defined in Section 4 to make corecursion even more expressive. More precisely, the meta-interpreter we present here allows the user to select only a part of the arguments of a coinductive predicate on which regular corecursion is defined.

Let us consider again the definition of predicate `max` given in the previous section.

```
:- use_module(cosldmeta2finally).

coinductive(max(_,_)).
coinductive(aux_max(_,_,_)).

max([N|L],M) :- aux_max(L,N,M).

aux_max([],N,N).
aux_max([N1|L],N2,M) :- (N1 > N2 -> N3 = N1; N3 = N2),
    aux_max(L,N3,M).
finally(aux_max(_,N,N)).
```

As already noted, in the worst case, if a list is cyclic with period n, then $2n$ elements have to be inspected to return the maximum of the list; indeed, to be able to apply the coinductive hypothesis, the second argument to `aux_max` must first become the maximum of the whole list.

This problem is due to the fact that all arguments of `aux_max` are involved when a coinductive hypothesis is applied, whereas coinduction could be confined to the first argument only.

To this aim, we allow the user to annotate the argument of a coinductive predicate with `n`, if regular corecursion does not depend on such an argument; for brevity, the arguments which are not annotated with `n` are called coinductive.

```
:- use_module(cosldmeta2annotated).

coinductive(max(n,n)).
coinductive(aux_max(_,n,n)).

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

```
aux_max([],N,N).
aux_max([N1|L],N2,M) :- (N1 > N2 -> N3 = N1; N3 = N2),
    aux_max(L,N3,M).
finally(aux_max(_,N,N),_).
```

Now, it is made explicit that the first argument of `aux_max` is the only one on which regular corecursion depends on; all other arguments are annotated with `n`. This allows the program to compute the maximum of a cyclic list in n steps, where n is the period of the list.

Note that, differently from the examples in the previous section, here `finally` has arity 2 instead of 1, even though in this example the second argument is unused. We will explain its use in the last example of this section.

With annotations predicate `max` can be easily defined directly, without introducing an auxiliary predicate that uses an accumulator.

```
:- use_module(cosldmeta2annotated).

coinductive(max(_,n)).

max([N],N).
max([N|L],M) :- max(L,M1),(N > M1 -> M = N; M = M1).
finally(max([N|_],N),_).
```

Now that corecursion involves only the first argument of the predicate (recall the same example without annotations in the previous section), the definition of `max` works correctly:

```
?- L=[1,2,3,2,1|L],cosld(max(L,M)).
L = [1, 2, 3, 2, 1|L],
M = 3 .
```

The pruning meta-interpreter for dealing with argument annotations is defined as follows (the non pruning version can be defined in a very similar way):

```
:- module(cosldmeta2annotated,[cosld/1]).

:- use_module(library(ordsets)).

cosld(G) :- ord_empty(E),solve(E,G).

solve(H, (G1,G2)) :- !,solve(H, G1), solve(H,G2).
solve(_,A) :- inductive(A),!,A.
solve(H,A):- found(A, H, MA),(clause(finally(A,MA),As) *->
    solve(H,As);true).
solve(H,A):- !,clause(A,As),insert(H,A,NewH),solve(NewH,As).

inductive(A) :- predicate_property(A,built_in),!.
inductive(A) :- predicate_property(A,file(AbsPath)),
                file_name_on_path(AbsPath,library(_)),!.
inductive(A) :- !,\+ coinductive(A,_).
```

```
insert(L1,A,L2) :- filtered(A,FA),is_in(FA,L1) -> fail;
    ord_add_element(L1,A,L2).

is_in(A1,[A2|_]) :- filtered(A2,FA2),unifiable(A1,FA2,_),!.
is_in(A,[_|L]) :- is_in(A,L),!.

coinductive(A1,A2) :- !,functor(A1,N,A), functor(A2,N,A),
    coinductive(A2).

filtered(T1,T2) :- !,coinductive(T1,AnnT1), T1 =.. L1, AnnT1
     =.. AnnL1, combine(L1, AnnL1, L2), T2 =.. L2.

combine([F|Args1],[F|Anns],[F,N|Args2]) :- combine_args(
    Args1,Anns,Args2), length(Args1,N).

combine_args([],[],[]) :- !.
combine_args([Arg1|Args1],[Ann|Anns],Args) :- (Ann == n ->
    Args = Args2; Args = [Arg1|Args2]), combine_args(Args1,
    Anns,Args2).

found(A,L,MA) :- filtered(A,FA),is_in(FA,L,MA).

is_in(A1,[A2|_],A2) :- filtered(A2,FA2),A1=FA2,!.
is_in(A,[_|L],MA) :- is_in(A,L,MA),!.
```

The predicate `filtered` has been implemented to filter out the arguments that are not coinductive; furthermore, a first argument corresponding to the arity of the predicate is added, to correctly deal with overloaded predicate symbols: one might declare two coinductive predicates with the same name, different arity, but the same number of coinductive arguments.

For instance, given the annotation `aux_max(_,n,n)` in the previous example, the goal `?- filtered(aux_max([], 1, X), A)` succeeds with `A=aux_max(3, [])`, since only the first argument is coinductive.

Predicate `filtered` is used for correctly managing the list of coinductive hypotheses (predicates `insert/3`, `is_in/2`, `found/3`, and `is_in/3`); note that the list of coinductive hypotheses still contains the unfiltered atoms; filtering is needed for finding a coinductive hypothesis which unifies (modulo filtering of non coinductive arguments) with a given atom.

In comparison with all previous versions of the meta-interpreter, the arity of predicates `found` and `finally` has been incremented by 1. If the current atom $p(t_1, \ldots, t_n)$ to be solved unifies with a coinductive hypothesis, and all arguments of p are coinductive, then they are all involved in the unification process. However, if some argument is not coinductive, then it is possible to keep distinct the corresponding term in the atom to be solved, and the corresponding term in the coinductive hypothesis, since they will not be unified.

For instance, consider the following goal resolution:

```
?- L1=[1,2|L1],L2=[2|L1],found(aux_max(L2,2,R1),[aux_max(L2
    ,1,R2),aux_max(L1,2,R3)],A).
```

```
L1 = [1,2|L1],
L2 = [2|L1],
A = aux_max(L2, 1, R2).
```

The first argument of `found` (`aux_max(L2,2,R1)`) is the atom to be found in the list of coinductive hypotheses, whereas the third one (`aux_max(L2, 1, R2)`) is the found coinductive hypothesis (after unification): the only argument which is the same in both atoms is the coinductive one.

In the third clause defining `solve/2`, if a coinductive hypothesis is found that can be unified (modulo filtering of non coinductive arguments) with the current atom `A`, then such a coinductive hypothesis `MA` is returned (containing also its non coinductive arguments); then, the `finally` clause can mention both the atom `A` and the coinductive hypothesis `MA`; that is why predicate `finally` has arity 2.

To show the usefulness of the second argument of `finally`, we implement a predicate for testing bipartiness of a graph. We recall that a bipartite undirected graph is a graph whose set of vertices can be partitioned in two sets such that the two vertices of every edge belong to different sets.

In a similar way shown for automata, a graph can be represented by one of its vertices, together with its adjacency list.

```
:- use_module(cosldmeta2annotated).

coinductive(bipartite(n)).
coinductive(no_odd_cyc(_,n)).

bipartite(V) :- no_odd_cyc(V,0).

no_odd_cyc(vertex(_,L),N1) :- N2 is (N1 + 1) mod 2,
    no_odd_cyc(L,N2).
no_odd_cyc([],_).
no_odd_cyc([V|L],N) :- no_odd_cyc(V,N),no_odd_cyc(L,N).
finally(no_odd_cyc(_,N1),no_odd_cyc(_,N2)) :- N1 == N2.
```

The implementation of predicate `bipartite` is based on the well-known property that states that a graph is bipartite if and only if it does not contain an odd cycle (that is, a cycle containing an odd number of vertices). The predicate uses the auxiliary coinductive predicate `no_odd_cyc` which takes the graph and a parity bit (initially set to 0), and succeeds only if the graph does not contain an odd cycle.

If the graph does not contain a cycle, then predicate `no_odd_cyc` trivially succeeds. For every cycle, the `finally` clause is applied: the parity bit `N1` of the current atom `no_odd_cyc(_,N1)` must be equal to the parity bit `N2` of the corresponding coinductive hypothesis `no_odd_cyc(_,N2)`, otherwise an odd cycle has been detected, and the predicate fails.

6. Conclusion

We have proposed several meta-interpreters to study better semantics and program abstractions supporting regular corecursion in Prolog.

This paper is an extended version of a previous work by the same author [1]; in particular, both Section 4 and 5 are new, and contain original contributions.

In Section 4, `finally` clauses have been introduced, to enhance the expressive power of regular corecursion; such a feature allows a simpler definition for some coinductive predicates (for instance, those that correspond to existentially quantified properties).

In Section 5 we have introduced another program abstraction with the aim of making regular corecursion even more expressive and flexible to be used: the user can select the arguments that have to be considered in the coinductive definition of a predicate.

For future developments we envisage at least two different interesting directions.

The first direction is to investigate on efficient implementations of coinductive Prolog able to avoid non terminating failures as the pruning vanilla meta-interpreter presented here; for instance, one could use DRA tabling [8] to implement efficient meta-interpreters supporting effective regular corecursion in Prolog. Another interesting research direction consists in studying regular corecursion for non logical programming languages [5].

References

[1] D. Ancona. Regular corecursion in Prolog. In *SAC 2012*, pages 1897–1902, 2012.

[2] D. Ancona, A. Corradi, G. Lagorio, and F. Damiani. Abstract compilation of object-oriented languages into coinductive CLP(X): can type inference meet verification? In B. Beckert and C. Marché, editors, *Formal Verification of Object-Oriented Software International Conference, FoVeOOS 2010, Paris, France, June 28-30, 2010, Revised Selected Papers*, volume 6528 of *LNCS*. Springer, 2011.

[3] D. Ancona and G. Lagorio. Coinductive type systems for object-oriented languages. In S. Drossopoulou, editor, *ECOOP 2009*, volume 5653 of *LNCS*, pages 2–26. Springer, 2009. Best paper prize.

[4] D. Ancona and G. Lagorio. Idealized coinductive type systems for imperative object-oriented programs. *RAIRO - Theoretical Informatics and Applications*, 45(1):3–33, 2011.

[5] D. Ancona and E. Zucca. Corecursive Featherweight Java. In *FTfJP '12*. ACM, 2012. To appear.

[6] J. Barwise and L. Moss. Vicious circles: On the mathematics of non-wellfounded phnenomena. *J. of Logic, Lang. and Inf.*, 6:460–464, 1997.

[7] N. Ghani, M. Hamana, T. Uustalu, and V. Vene. Representing cyclic structures as nested datatypes. In *TFP*, pages 173–188, 2006.

[8] H.-F. Guo and G. Gupta. A simple scheme for implementing tabled logic programming systems based on dynamic reordering of alternatives. In *ICLP*, pages 181–196, 2001.

[9] X. Leroy and H. Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207(2):284–304, 2009.

[10] R. Min and G. Gupta. Coinductive logic programming and its application to boolean sat. In *FLAIRS Conference*, 2009.

[11] R. Min and G. Gupta. Coinductive logic programming with negation. In *LOPSTR*, pages 97–112, 2009.

[12] N.Saeedloei and G. Gupta. Verifying complex continuous real-time systems with coinductive CLP(R). In *LATA 2010*, LNCS. Springer, 2010.

[13] L. Simon. *Extending logic programming with coinduction.* PhD thesis, University of Texas at Dallas, 2006.

[14] L. Simon, A. Bansal, A. Mallya, and G. Gupta. Co-logic programming: Extending logic programming with coinduction. In *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007*, pages 472–483, 2007.

[15] L. Simon, A. Mallya, A. Bansal, and G. Gupta. Coinductive logic programming. In *Logic Programming, 22nd International Conference, ICLP 2006*, pages 330–345, 2006.

[16] F. A. Turbak and J. B. Wells. Cycle therapy: A prescription for fold and unfold on regular trees. In *PPDP*, pages 137–149, 2001.