Regular corecursion in Prolog*

Davide Ancona
DISI - Università di Genova
Via Dodecaneso, 35
16146 Genova, Italy
davide@disi.unige.it

ABSTRACT

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

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

In this paper we propose two main alternative vanilla meta-interpreters to support regular co-recursion in Prolog as an interesting programming style in its own right, able to elegantly solve problems that would require more complex code if conventional recursion were used. In particular, the second meta-interpreters avoids non termination in several cases, by restricting the set of possible answers.

The semantics defined by these vanilla meta-interpreters are an interesting starting point to study new semantics able to support regular co-recursion for non logical languages.

Categories and Subject Descriptors

D.1.6 [**Programming Techniques**]: Logic Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*recursion*

General Terms

Languages

Keywords

Logic programming, coinduction and corecursion

1. INTRODUCTION

Corecursion [4] is the ability of defining a function that produces some infinite data in terms of the function and the data itself, and is typically supported by languages with lazy evaluation. As an example, the following Haskell code defines the infinite stream !0 : 1! : 2! : ... containing the factorial of all natural numbers.

```
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. Now let us use the predefined function **all** to check whether all elements in the stream are greater than 0.

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

Clearly these kinds of checks are only semi-decidable (termination is guaranteed only if the predicate does not hold for some element, as in `all (\x -> x<100) fact_stream`) because our stream represents an infinite non regular list of integers, that is, it unfolds into an infinite term that it is not regular.

A term is *regular* if it has a finite set of subterms (hence, trivially, every finite term is regular); infinite regular terms correspond to cyclic data that can be represented in a finite way.

```
ones = 1:ones
```

Variable `ones` as defined above contains the infinite regular stream 1 : 1 : ..., indeed the set of all subterms contains just two terms: 1 and the term itself. However, in Haskell the expression `all (\x -> x>0) ones` does not terminate, even though in this case the problem is trivially decidable; this happens because the logical conjunction `&&` is strict in its second argument (when the first argument evaluates to **True**), and because **all** is defined inductively.

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

SAC'12 March 25-29, 2012, Riva del Garda, Italy.

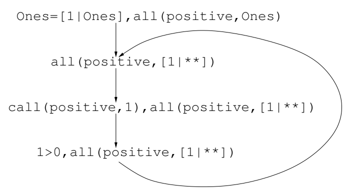

Figure 1: Regular derivation for the goal `Ones=[1|Ones],all(positive,Ones)`

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 above. Modern Prolog interpreters (as SWI-Prolog) support regular terms; the unification `Ones=[1|Ones]` succeeds, because occur-check is not performed, and `Ones` is substituted with the regular list containing infinite occurrences of `1` (represented by `[1|**]` in SWI-Prolog); in this way the Prolog interpreter tries to build an infinite derivation for the goal and, thus, does not terminate. The conventional interpreter is based on the inductive interpretation of Horn clauses (called the inductive Herbrand model), which 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. [11, 13, 12] 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 [13, 7].

Coinductive logic programming has proved to be useful for formal verification [8, 10], static analysis and symbolic evaluation of programs [2, 1, 3]. In this paper we propose two main alternative vanilla meta-interpreters to support regular corecursion in Prolog as an interesting programming style in its own right, able to elegantly solve problems that would require more complex code if conventional recursion were used.

CoSLD resolution is not computable in its general form, but it can be implemented if only regular terms and derivations are considered. Let `cosld` be a predicate, implemented by a Prolog meta-interpreter (see the next section), that coinductively resolves a goal; then the following goal succeeds:

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

The infinite regular derivation built by the meta-interpreter is depicted in Figure 1.

In Haskell a function with the same behavior cannot be implemented so simply, and specific datatypes must be expressly defined and used [14, 5]

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

Regular corecursion is a programming style that is implicitly adopted quite frequently when cyclic data structures are manipulated and termination becomes an issue; maybe the most evident examples are given by graph algorithms where vertices or edges must be marked to avoid infinite loops (in the next section we see a similar example involving automata). Direct support for regular coinduction allows elimination of all boilerplate code needed for manual bookkeeping of inspected data in a cyclic structure, thus making code simpler and more readable; furthermore, similarly as happens for recursion, regular corecursion supported by an interpreter or a compiler can be more reliable and efficient then a manual implementation; roughly, while recursion can always be eliminated in a program by using iteration with a stack, regular corecursion can be eliminated by using recursion with a set (that is, a data structure implementing the abstract data type set).

In the next section we will define different versions of a meta-interpreter supporting regular corecursion in Prolog, and see some concrete examples of regular corecursion in Prolog. In particular, we show that a rather drastic pruning of the search tree is needed to ensure termination in useful cases; even though such a pruning may limit the number of possible answers, this limitation does not affect the results in our examples where predicates are expected to be used with arguments that are either ground (input arguments) or variables (output argument). This seems a promising starting point for studying the design and the semantics of regular corecursion for programming languages not based on the logical paradigm.

2. META-INTERPRETERS

This section elaborates previous results [11] by defining two different versions of a vanilla meta-interpreter (where vanilla means based on built-in unification and predicate **clause**/2) implementing regular coSLD. Even though vanilla meta-interpreters are too inefficient to be suitable for practical uses, the meta-programming facilities offered by Prolog are an ideal tool to experiment implementations of coSLD adaptable to other programming language paradigms.[2]

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.

The basic meta-interpreter implementing regular coSLD is a straightforward extension of the conventional vanilla interpreter implementing standard SLD resolution for Prolog.

```
:- 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):- member(A, H).
solve(H,A):- clause(A,As),ord_add_element(H,A,NewH),
             solve(NewH,As).
```

The predicate `solve` takes two arguments where the first is an ordered set of atoms, called the coinductive hypotheses, and the second is the goal that have to be resolved. The set of coinductive hypotheses contains all atoms that the interpreter has been processed so far, and are needed for building infinite regular derivations (see below).

[2]We mainly think of functional languages, even though supporting regular coinduction for object-oriented languages could be interesting as well.

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 resolve 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` (`member(A,H)`); if the search succeeds, then the atom is resolved and removed from the goal, and the computed answer substitution is refined accordingly, since predicate `member` exploits unification.[3]

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.

Finally, the main predicate `cosld` tries to solve the goal starting from the empty set of coinductive hypotheses.

Let us see how the interpreter works with a very simple example program defining the predicate `is_nat`.

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

In this case, the only difference with inductive Prolog is that `is_nat` succeeds also for the infinite regular term `s(**)` solution of the unification problem `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 = s(z) ;
N = s(s(**)) ;
N = s(s(**)) ;
N = s(s(z)) ;
...
```

This very basic meta-interpreter has a serious restriction, since the **clause** predicate does not work with built-in predicates; furthermore, library predicates that have been defined for the standard inductive semantics should not be interpreted coinductively. To this aim, we introduce two predicates `inductive` and `coinductive` to partition predicates: the user has to explicitly specify all coinductive predicates (necessarily user-defined), whereas all other predicates are inductive: those that are built-in or imported from the Prolog library, and all user-defined predicates that have not been declared to be coinductive.

```
:- 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):- member(A, H).
solve(H,A):- clause(A,As),ord_add_element(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).
```

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.

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

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 [12].

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

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

We now propose two extensions to the meta-interpreter, the first allows elimination of repeated answers due to redundant coinductive hypotheses, while the second performs also a pruning of the search tree (therefore we call it "pruning meta-interpreter) to avoid some kinds of non terminating failures.

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 resolved, 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 the solution of the unification problem `is_nat(N0)=is_nat(s(N0))`, that is, `s(**)`. 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 resolved, 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, with the same answer `s(s(**))`.[4]

To avoid this problem, we modify the meta-interpreter: a new coinductive hypothesis is inserted only if it does not unify with any other element in the set. For brevity, we show only the modified clause for `solve`, and the clauses for predicate `is_in`.

```
solve(H,A):- clause(A,As),
             (is_in(A,H) -> NewH=H; %no insertion
                        ord_add_element(H,A,NewH)),
             solve(NewH,As).
is_in(E,[X|_]) :- unifiable(E,X,_),!.
is_in(E,[_|L]) :- is_in(E,L),!.
```

In this way the set of coinductive hypotheses is kept smaller, and some repeated answers are avoided. Now resolution of the goal `cosld(is_nat(N))` yields the following answers:

[4]In SWI-Prolog the goal `N1=s(N1),N2=s(s(N2)),N1==N2` succeeds, as expected; however, since terms are not simplified after unification, the interpreter displays `N1` and `N2` differently.

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

A pruning of the search trees can be performed by applying a clause only if the atom to be resolved does not unify with a coinductive hypothesis, after it has been unified with the head of the clause. Hence we derive from the previous meta-interpreter a pruning version by modifying the clause

```
solve(H,A):- clause(A,As),
             (is_in(A,H) -> NewH=H;
                            ord_add_element(H,A,NewH)),
             solve(NewH,As).
```

in the following way

```
solve(H,A):- clause(A,As),
          (\+ is_in(A,H) -> ord_add_element(H,A,NewH),
             solve(NewH,As)).
```

In this way the set of computed answers can be considerably restricted. For instance, the resolution of the goal `cosld(is_nat(N))` only yields three possible answers.

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

This version of the meta-interpreter does not work with programs based on generate-and-test methods. However, the resolution of all ground goals having shape is_nat(s^n(z)) still succeeds:

```
?- cosld(is_nat(s(s(s(z))))).
true.
```

Even though there are cases where resolution with pruning fails for ground goals which succeeds with the non pruning meta-interpreter, some interesting examples (see next section) require pruning of the search trees to avoid infinite failures.

3. REGULAR CORECURSION AT WORK

In this section we consider several examples where regular corecursion allows more succinct and elegant solutions; we also show how constraint logic programming can be usefully exploited in conjunction with regular corecursion. For two examples the pruning meta-interpreter is needed to avoid non terminating failures.

Membership for regular lists. In the previous section we have shown how predicate `all` can be easily defined corecursively in Prolog; more generally, this is true whenever universally quantified predicates have to be checked on regular terms. Checking existentially quantified predicates is less simple. A classical example is membership test on regular lists. The following definition is not correct.

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

For instance, the goal `cosld(L=[1,2,3|L],member(5,L))` succeed, instead of failing. Indeed, with coinductive recursion the meta-interpreter ends up resolving the initial goal, and, hence, always succeeds. A possible solution consists in defining the predicate `not_member` that checks that the negated property holds universally, but then one has to rely on coSLD negation [9].

An alternative solution is given by the following clauses.

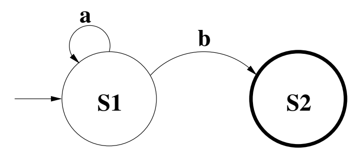

Figure 2: A deterministic finite automaton recognizing the language `a*b`

```
coinductive(member(_,_)).
coinductive(aux_member(_,_,_)).
member(N,L) :- aux_member(N,L,_).
aux_member(N,[N|_],t).
aux_member(N1,[N2|L],R2) :-
   N1\=N2,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 is necessary for ensuring correctness. To be used correctly, the third argument of `aux_member` must be a variable; the search of the element in the list succeeds iff such a variable is instantiated with the constant `t`; for this reason the syntactic equality test `R1==t` is used.

The resolution of the goal `cosld(L=[1,2,3|L],member(5,L))` correctly fails with this new definition, but only when the pruning meta-interpreter is used; resolution without pruning does not terminate since the second clause of `aux_member` is selected infinitely many times. Note that the clauses defined above work with both finite and infinite regular lists.

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 (generated 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.[5]

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 infinite regular Prolog term associated with the logical variable `S1` after the resolution of the following unification problem:

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

Let us now consider the following right linear grammar:

[5]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.

```
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 infinite regular Prolog term associated with the logical variable `A`, after the resolution of the following unification problem:

```
A = or([b],[a|A])
```

Even though we have omitted the formal definitions for space limitations, from the two examples above it should be clear how any finite automaton and extended right linear grammar can be represented by a regular Prolog term. 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 first two clauses for `accept` 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 not inductive:

```
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:

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

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` after the resolution of the unification problem `N=[5|P],P=[7,2|P]` corresponds to the repeating decimal $0.5\overline{72}$ that equals the fraction $\frac{62}{110}$. Since multiplying a repeating decimal by 10^e (with $e > 0$) is equivalent to a left shift of e positions, we have that the following equations hold: $100\texttt{P}= 72+\texttt{P}, 10\texttt{N}= 5+\texttt{P}$. Therefore $\texttt{P}= \frac{72}{99}, \texttt{N}= \frac{5}{10} + \frac{72}{99} = \frac{62}{110}$. 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` after the resolution of the unification problem `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 all result and carry digits for all infinite 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 the power 10^{-i}) 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.

We consider now a more advanced solution exploiting constraints over finite domains, to show also 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 all 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 $0.\overline{9}+0.1$:

```
?- cosld((_M=[9|_M],_Z=[0|_Z],_N=[1|_Z],
   add(_M,_N,R,C))).
R = [1, 0|**],
C = 1 ;
R = [0, 9|**],
C = 1 ;
false.
```

The meta-interpreter finds two different solutions (clearly equivalent): $0.1\overline{0}$ with carry 1, or $0.0\overline{9}$ with carry 1.

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. CONCLUSION

We have proposed two alternative meta-interpreters to support regular corecursion in Prolog as an interesting programming style in its own right, able to elegantly solve problems that would require more complex code if conventional recursion were used. To avoid infinite failure, one of the meta-interpreters uses a simple but effective heuristic for pruning search trees.

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 [6] 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, A. Corradi, G. Lagorio, and F. Damiani. Abstract compilation of object-oriented languages into coinductive CLP(X): can type inference meet verification? In *FoVeOOS 2010*, *LNCS*. Springer, 2011.

[2] D. Ancona and G. Lagorio. Coinductive type systems for object-oriented languages. In *ECOOP 2009*, *LNCS*, Springer, 2009. Best paper prize.

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

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

[5] N. Ghani, M. Hamana, T. Uustalu, and V. Vene. Representing cyclic structures as nested datatypes. In *TFP*, 2006.

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

[7] X. Leroy and H. Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207(2), 2009.

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

[9] R. Min and G. Gupta. Coinductive logic programming with negation. In *LOPSTR*, 2009.

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

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

[12] L. Simon, A. Bansal, A. Mallya, and G. Gupta. Co-logic programming: Extending logic programming with coinduction. In *ICALP 2007*, 2007.

[13] L. Simon, A. Mallya, A. Bansal, and G. Gupta. Coinductive logic programming. In *ICLP 2006*, 2006.

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