Coinductive Logic Programming in Logtalk

Paulo Moura

Dep. of Computer Science, University of Beira Interior, Portugal
CRACS, INESC TEC (formerly INESC Porto), Portugal
pmoura@di.ubi.pt

Abstract. We describe the implementation of coinductive logic programming found in Logtalk, discussing its features and limitations. As Logtalk uses as a back-end compiler a compatible Prolog system, we discuss the status of key Prolog features for an efficient and usable implementation of coinduction.

Keywords: logic programming, coinduction, objects, implementation, portability.

1 Introduction

This paper describes the current implementation of coinductive logic programming found in Logtalk, discussing its features and limitations. As Logtalk uses as a back-end compiler a compatible Prolog system, we also discuss the status of key Prolog features for an efficient and usable implementation of coinduction. We assume that the reader is familiar with the theoretical work in coinduction (see e.g. [1–3]). Therefore, this paper is written from a practical, technical point-of-view.

The main motivation for implementing support for coinductive logic programming in Logtalk is to make it the preferred tool for solving problems that require coinductive reasoning. This is an ambitious and long term goal, but we believe that the core features of Logtalk, including its code encapsulation and code reuse mechanisms, provide a strong framework for solving complex problems where coinduction is one of the solution components. In addition, the inherent requirements on back-end Prolog compiler native features, for example, on support for rational terms, tabling, and constrain libraries, hopefully help drive future enhancements to Prolog implementations that will ultimately benefit the logic programming community at large.

The remainder of the paper is organized as follows. Section 2 provides an overview of Logtalk. Section 3 describes the coinductive predicate directives provided by Logtalk. Section 4 presents some examples of coinductive predicates. Section 5 describes in detail our implementation of coinduction, discussing its features and limitations. Section 6 shows Logtalk built-in support for debugging coinductive predicates. Section 7 compares our implementation with related work. Section 8 concludes and outlines future work.

2 Logtalk in a Nutshell

Logtalk [4, 5] is an open source object-oriented logic programming language that can use most Prolog implementations as back-end compilers. Logtalk focus in code encapsulation and code reuse features, providing a versatile alternative to Prolog module systems. As a multi-paradigm language, Logtalk supports classes, prototypes, parametric objects, categories (fine-grained units of code reuse), separation between interface and implementation using protocols, event-driven programming, and high-level multi-threading programming. Logtalk uses *object* as a generic term: an object can play the role of, e.g., an instance, a class, or a prototype. The relations between objects, protocols, and categories define different *patterns of code reuse*. Logtalk entities can be static, defined in source files, or dynamic, created at runtime. Computations are performed by sending *messages* (corresponding to predicates) to objects. Logtalk enforces predicate encapsulation (predicates can be declared public, protected, or private) and features a clear distinction between predicate declaration and predicate definition (using a closed-world assumption when a predicate is declared but not defined). Logtalk is developed with a strong emphasis in portability and reliability. It is used worldwide in academic and commercial projects. Its distribution includes extensive documentation, numerous examples, a library, and basic development tools (for debugging, unit testing, and documenting).

3 Coinductive Predicate Directives

Logtalk requires coinductive predicates to be explicitly declared, as the predicate clauses must be compiled with support for checking coinductive success and for keeping a stack of coinductive hypotheses. Coinductive predicates are declared using the `coinductive/1` predicate directive. The argument of this directive can be a predicate indicator when all the predicate arguments are relevant for coinductive success. For example:

Listing 1.1. Infinite lists with a repeating pattern of binary digits

```
:- object(binary).

    :- public(p/1).
    :- coinductive(p/1).

    p([0| T]) :- p(T).
    p([1| T]) :- p(T).

:- end_object.
```

When only some arguments should be considered when testing for coinductive success, the directive argument must be a predicate template. In this case, coinductive predicate arguments are represented by the atom `'+'`, while arguments

that should be disregarded are represented by the atom `'-'`. In the following example, we want to find the cyclic paths in a graph whose length (of the repeating pattern) is bound by a given value:

Listing 1.2. Length-limited cyclic paths in a graph

```
:- object(cyclic_paths).

    :- public(path/3).
    path(From, Path, MaxLength) :-
        path(From, Path, 0, MaxLength).

    :- private(path/4).
    :- coinductive(path(+, +, -, -)).
    path(From, [From| Path], Length, MaxLength) :-
        arc(From, Next),
        Length < MaxLength,
        Length1 is Length + 1,
        path(Next, Path, Length1, MaxLength).

    arc(a, b).
    arc(b, c).
    arc(c, a).  arc(c, d).
    arc(d, a).

:- end_object.
```

In this case, coinductive success depends only on the first two arguments of the `path/4` auxiliary predicate. The remaining two arguments are only used to limit the solutions found.

This representation of relevant arguments is the same representation used in predicate tabling directives in systems such as B-Prolog, where it is possible to indicate which arguments should be considered for variant checking, allowing selective tabling of answers. The use of a common representation for declaring relevant predicate arguments for coinductive success and for variant checking when tabling predicate answers may provide, however, benefits other than language consistency. Intuitively, we expect that the arguments that are relevant for coinductive success are the same that are relevant for variant checking. This would mean that the `coinductive/1` predicate directive would make writing tabling directives for the same predicates redundant, simplifying programming.

4 Implementation

A coinductive predicate is compiled by adding a *preflight predicate* that checks for coinductive success and, if not yet achieved, pushes the current goal to the stack of coinductive hypotheses (i.e., the ancestor goals for the coinductive predicate query). This preflight predicate calls the coinductive predicate defined by

the programmer. The clauses defined by the programmer are modified by replacing the recursive call to the coinductive predicate by a call to the preflight predicate. The per-object table of defined predicates ensures that a message corresponding to the coinductive predicate is translated to a call to the preflight predicate.

The stack of coinductive hypotheses is represented using a list and passed between predicate calls using a hidden extra argument that is used for representing the execution context. This extra argument is added by the Logtalk compiler to the compiled form of all predicates.[1] An alternative implementation of the coinductive hypotheses stack would be to use the destructive assignment built-in predicates that are found on some Prolog compilers. But these predicates are not standard and our goal is a portable implementation.

Checking for coinductive success is performed by attempting to unify the current goal with an elements of the coinductive hypotheses stack. This unification may succeed, on backtracking, for more than one hypothesis, thus leading to multiple solutions. On the other hand, the current goal is only pushed to the stack of coinductive hypotheses if it does not unify with any of its elements. This semantics is efficiently implemented using the *soft-cut* control construct found on several Prolog compilers, including all of those that provide the necessary minimal support for rational terms.[2]

The following example of the compilation of the coinductive predicate `p/1` in Listing 1.1 illustrates our current implementation (with all non-relevant details, including the internal names of the coinductive and preflight predicates, abstracted for clarity of presentation):

Listing 1.3. Compiled code for a coinductive predicate `p/1`

```
p_1_coinduction_preflight(A, Stack) :-
    (   member(p(A), Stack) *->
        true
    ;   p(A, [p(A)| Stack])
    ).

p([0| A], Stack) :-
    p_1_coinduction_preflight(A, Stack).
p([1| A], Stack) :-
    p_1_coinduction_preflight(A, Stack).
```

[1] Logtalk uses an extra predicate argument for passing execution context information, which includes the *sender* of a message and the object that received the message (*self*). This allowed a simple implementation of the stack of coinductive hypotheses as just an additional argument of the execution context term.

[2] In this paper, we use the usual definition of *rational term*: an infinite term with a finite representation.

In the code above, the predicate `member/2` has its usual definition and the `(*->)/2` operator denotes the soft-cut control construct, as found on several Prolog compilers such as ECLiPSe, GNU Prolog, SWI-Prolog, and YAP.[3]

4.1 Implementation Limitations

In the current Logtalk implementation, the stratification of programs mixing non-coinductive predicates and coinductive predicates is neither checked nor enforced. Thus, ensuring stratification is a responsibility left to the programmer.

A second, more fundamental, limitation is partially a consequence of the lack of native Prolog support for tabling of rational terms (see Section 4.3). The practical consequence is that, while coinductive predicates can *recognize* any valid solution, they can only *generate* a (finite) subset of all possible solutions. For example, using the coinductive predicate `p/1` in Listing 1.1, we get the results illustrated in Listing 1.4.[4]

Listing 1.4. Solutions generated for the coinductive predicate `p/1` in Listing 1.1

```
?- binary::p(X).
X = [0|X] ;
X = [1|X] ;
false.

?- L = [0,1,0| L], binary::p(L).
L = [0, 1, 0|L] ;
false.
```

We describe the finite set of generated solutions as the set of *basic cycles*, where a basic cycle is a solution that cannot be expressed as a combination of other solutions. Ideally, any possible solution could be generated from a *combination* of these basic cycles. But we do not have yet a formal proof and our intuition can be wrong. With tabling support available, we could use an alternative compilation scheme where the current goal would be added to the stack of coinductive hypotheses, independently of the current goal unifying with any of the existing coinductive hypotheses. Without tabling, and for the example in Listing 1.1, this alternative compilation scheme repeatedly generates, as expected, and as long as memory is available, the first solution, as illustrated in Listing 1.5. With a suitable tabling implementation, we would not get stuck repeating the same solution, but we could still get an infinite number of solutions. In alternative, a breadth-first inference mechanism can also avoid repeatedly generating the same solution. In fact, this approach is used in one of the variations of the U.T.Dallas

[3] Some other Prolog compilers such as SICStus Prolog use a built-in meta-predicate, `if/3`, for implementing a soft-cut. Logtalk uses either the `(*->)/2` control construct or the `if/3` built-in meta-predicate depending on the used back-end Prolog compiler.

[4] Using SWI-Prolog as the Logtalk back-end compiler.

Prolog meta-interpreter for coinductive predicates. But a solution where we generate the finite set of basic cycles and use it to construct an *expression* representing all possible combinations of these basic cycles would be preferable as this expression could then be used to both generate and test solutions as necessary.

Listing 1.5. Solutions generated for the coinductive predicate `p/1` in Listing 1.1 using the alternative compilation scheme

```
?- binary::p(X).
?- binary::p(X).
X = [0|X] ;
X = [0|_S1], % where
    _S1 = [0|_S1] ;
X = [0, 0|X] ;
X = [0, 0|_S1], % where
    _S1 = [0|_S1] ;
X = [0|_S1], % where
    _S1 = [0, 0|_S1] ;
X = [0, 0, 0|X] ;
X = [0, 0, 0|_S1], % where
    _S1 = [0|_S1] ;
X = [0, 0|_S1], % where
    _S1 = [0, 0|_S1] ;
X = [0|_S1], % where
    _S1 = [0, 0, 0|_S1] ;
X = [0, 0, 0, 0|X] ;
...
```

4.2 Implementation Portability

The current coinduction implementation supports a subset of the Logtalk compatible back-end Prolog compilers. Namely, ECLiPSe, SICStus Prolog, SWI-Prolog, and YAP. The two main Prolog native features necessary for our implementation are (1) a soft-cut control construct or built-in predicate[5] and (2) minimal support for rational terms. The soft-cut control construct is already implemented or is being implemented on most Prolog compilers. The most problematic feature is the the support for rational terms, as we discuss next.

4.3 Rational Terms Support

Although an implementation of coinductive logic programming must be able to create, unify, and print bindings with rational terms, there is very limited standard support for this kind of terms. The latest official revision of the ISO

[5] Although it is possible to implement the preflight predicate without using a soft-cut, the resulting code would provide poor performance as it would require, in the worst case, traversing the list that implements the stack of coinductive hypotheses twice.

Prolog Core standard [6] added an `acyclic_term/1` built-in predicate but does not specify a comprehensive set of *operations* on rational terms that should be supported. In addition, for a long time, rational terms were regarded more as a problem than as a feature in Prolog compilers. Thus, the supported operations on rational terms depend on the Prolog compiler. Fortunately, implementing coinduction requires only three basic operations: (1) creation of rational terms, (2) unification of rational terms, and (3) a suitable printing of rational terms, such that bindings resulting from queries to coinductive predicates can be non-ambiguously interpreted. Creating and unifying rational terms are supported by all compatible back-end Prolog compilers. But non-ambiguous printing of rational terms is a problem for most compilers. To illustrate the problem, consider the `p/1` coinductive predicate in Listing 1.1 and the query `p(X)`. Our implementation provides two solutions for this query, the rational terms `X = [0|X]` and `X = [1|X]`. The solutions as printed by ECLiPSe, SICStus Prolog, SWI-Prolog, and YAP are presented in Table 1:

Prolog compiler	**First solution**	**Second solution**
ECLiPSe 6.1.115	`X = [0, 0, 0, 0, ...]`	`X = [1, 1, 1, 1, ...]`
SICStus Prolog 4.0.4	`X = [0, 0, 0, 0, ...]`	`X = [1, 1, 1, 1, ...]`
SWI-Prolog 6.1.11	`X = [0|X]`	`X = [1|X]`
YAP 6.3.2	`X = [0|**]`	`X = [1|**]`

Table 1. Printing of rational terms bindings

The only reason we do not get into trouble when using ECLiPSe and SICStus Prolog is that both limit, by default, the maximum length of a list when printing terms.[6] YAP prints an ambiguous mark, `**`, to alert the user that is printing a rational term. Only SWI-Prolog provides a non-ambiguous printing of rational terms bindings.

4.4 Tabling of Rational Terms

The set of coinductive problems that can be tackled by the current implementation is limited by the lack of a compatible back-end Prolog compiler that natively supports tabling of rational terms. A simple example where tabling is required is in the definition of the `comember/2` predicate. This predicate succeeds when an element occurs an infinite number of times in a list. It can be defined as illustrated in Listing 1.6.

Listing 1.6. Definition of the coinductive predicate `comember/2`

```
:- coinductive(comember/2).
comember(X, L) :-
    drop(X, L, L1),
    comember(X, L1).
```

[6] ECLiPSe uses, by default, a `depth(20)` output option. SICStus Prolog uses, by default, a `max_depth(10)` output option.

```
:- table(drop/3).
drop(H, [H| T], T).
drop(H, [_| T], T1) :-
    drop(H, T, T1).
```

The auxiliary predicate `drop/3` is used to drop elements from the input list non-deterministically. But, without tabling support for rational terms, the call to this predicate in the definition of the `comember/2` will unify the first element and will limit the coinductive predicate to return that solution repeatedly (on backtracking) without ever moving to the next solution.

Although it is always possible to implement a high-level tabling solution, the relatively poor performance of such solution makes it preferable to work with Prolog implementers that already provide a native tabling system in extending it to support rational terms.

4.5 Coroutining and CLP(R) Libraries

Some of the recent research on coinduction focuses on model checking and timed automata (see e.g. [7–9]). The implementation of solutions for these classes of problems require the use of coroutining and CLP(R) libraries. All the back-end Prolog compilers we support for coinduction provide built-in coroutining primitives and these constraint libraries, although the ECLiPSe versions differs in its interface from those found on SICStus Prolog, SWI-Prolog, and YAP. Logtalk can account for the differences using its conditional compilation directives. Not an ideal solution, however, as it still results in code duplication. But there are two other, more significant, potential issues: lack of activate maintenance of some of these libraries and semantic differences between the different implementations of coroutining and constraint libraries. In fact, there is currently no standardization effort for constraint programming in Prolog, despite the area being widely recognized as fundamental for the practical success of logic programming.

5 Examples

The current Logtalk distribution includes sixteen coinduction examples, most of them adapted from publications on coinductive logic programming or originating from discussions with researchers in this area. The examples are complemented by unit tests, thus providing a handy solution for testing our implementation across compatible back-end Prolog compilers. In this section, we present and briefly discuss some of the most interesting examples, mainly to familiarize the reader on how to define coinductive predicates. The example queries output are produced using Logtalk with SWI-Prolog as the back-end compiler.

5.1 A Tangle of Coinductive Predicates

Our first example (Listing 1.7) illustrates a coinductive predicate with two starting points and no common solution prefix. This examples was originally written

by Feliks Kluźniak in the context of a discussion on how to combine coinductive predicate solutions to construct other valid solutions.

Listing 1.7. A coinductive predicate with two starting points and no common solution prefix, wrapped in a `tangle` object

```
:- object(tangle).

    :- public(p/1).
    :- coinductive(p/1).
    p([a| X]) :- q(X).
    p([c| X]) :- r(X).

    :- coinductive(q/1).
    q([b| X]) :- p(X).

    :- coinductive(r/1).
    r([d| X]) :- p(X).

:- end_object.
```

Listing 1.8 shows two queries for the `tangle::p/1` coinductive predicate. The first query works as a solution generator, while the second query tests a specific solution.

Listing 1.8. Sample queries for the `tangle::p/1` coinductive predicate

```
?- tangle::p(X).
X = [a, b|X] ;
X = [c, d|X] ;
false.

?- L = [a, b, c, d| L], tangle::p(L).
L = [a, b, c, d|L] ;
false.

?- L = [a, c| L], tangle::p(L).
false.
```

5.2 An Omega-Automaton

Our third example (Listing 1.9) is adapted from [2] and illustrates an omega-automaton, i.e. an automaton that accepts infinite strings. The commented out code shows how we can go from an automaton recognizing finite strings to an omega-automaton by simply dropping the base case in the recursive predicate definition.

Listing 1.9. A omega-automaton

```
:- object(automaton).

    :- public(automaton/2).
    :- coinductive(automaton/2).

    automaton(State, [Input| Inputs]) :-
        trans(State, Input, NewState),
        automaton(NewState, Inputs).
%   automata(State, []) :-  % we drop the base case in order
%       final(State).       % to get an omega-automaton

    trans(s0, a, s1).
    trans(s1, b, s2).
    trans(s2, c, s3).
    trans(s2, e, s0).
    trans(s3, d, s0).

    final(s2).

:- end_object.
```

Listing 1.10 shows generating and testing queries for the `automaton::automaton/2` coinductive predicate.

Listing 1.10. Sample queries for the `automaton::automaton/2` coinductive predicate

```
?- automaton::automaton(s0, X).
X = [a, b, c, d|X] ;
X = [a, b, e|X] ;
false.

?- L = [a, b, c, d, a, b, e| L], automaton::automaton(s0, L).
L = [a, b, c, d, a, b, e|L] ;
false.

?- L = [a, b, e, c, d| L], automaton::automaton(s0, L).
false.
```

5.3 A Sieve of Eratosthenes Coinductive Implementation

The second example (Listing 1.11) presents our coinductive implementation of the Sieve of Eratosthenes. An alternative solution, based on coroutining, is sketched in [2].

Listing 1.11. A Sieve of Eratosthenes coinductive implementation

```
:- object(sieve).

    :- public(primes/2).
    % computes a coinductive list with all the
    % primes in the 2..N interval
    primes(N, Primes) :-
        generate_infinite_list(N, List),
        sieve(List, Primes).

    % generate a coinductive list with a 2..N
    % repeating pattern
    generate_infinite_list(N, List) :-
        sequence(2, N, List, List).

    sequence(Sup, Sup, [Sup| List], List) :-
        !.
    sequence(Inf, Sup, [Inf| List], Tail) :-
        Next is Inf + 1,
        sequence(Next, Sup, List, Tail).

    :- coinductive(sieve/2).
    sieve([H| T], [H| R]) :-
        filter(H, T, F),
        sieve(F, R).

    :- coinductive(filter/3).
    filter(H, [K| T], L) :-
        (   K > H, K mod H =:= 0 ->
            % throw away the multiple we found
            L = T1
        ;   % we must not throw away the integer used for
            % filtering in order to return a filtered
            % coinductive list
            L = [K| T1]
        ),
        filter(H, T, T1).

:- end_object.
```

Listing 1.12 illustrates how to use our `sieve::primes/2` coinductive predicate to enumerate all the prime numbers in the `[1..20]` interval.

Listing 1.12. Enumerating prime numbers using coinduction

```
?- sieve::primes(20, P).
P = [2, 3|_S1], % where
    _S1 = [5, 7, 11, 13, 17, 19, 2, 3|_S1] .
```

6 Debugging Coinductive Predicates

As most extensions to existing logic programming languages, the practical use of coinduction depends not only on robust implementations with good performance but also on development tools support, specially, for debugging. Logtalk provides specific support for debugging coinductive predicates by allowing (1) tracing of coinductive success checks, (2) tracing of pushing the current goal to the stack of coinductive hypotheses, and (3) printing of the stack of coinductive hypotheses at any time. Operations (1) and (2) are collectively described as a *coinduction preflight* step, which takes place at every coinductive predicate call before proceeding to the clauses defined by the programmer. The example in Listing 1.13 shows a debugging section (with internal variable names renamed for clarity and using SWI-Prolog as the Logtalk back-end compiler):

Listing 1.13. Debugging a coinductive predicate call

```
?- binary::p(X).
  Call: (1) binary::p(X) ?
  Rule: p_1_coinduction_preflight(X) ?
  Call: (2) check_coinductive_success(p(X),[]) ?
  Fail: (2) check_coinductive_success(p(X),[]) ?
  Call: (3) push_coinductive_hypothesis(p(X),[],S) ?
  Exit: (3) push_coinductive_hypothesis(p(X),[],[p(X)]) ?
  Call: (4) p(X) ?
  Rule: (clause #1) p([0|L]) ?
  Call: (5) p_1_coinduction_preflight(L) ? x
  Sender:            user
  This:              binary
  Self:              binary
  Meta-call context: []
  Coinduction stack: [p([0|L])]
  Call: (5) p_1_coinduction_preflight(L) ?
  Rule: p_1_coinduction_preflight(L) ?
  Call: (6) check_coinductive_success(p(L),[p([0|L])]) ?
  Exit: (6) @(check_coinductive_success(p(S_1),[p(S_1)]),
                                        [S_1=[0|S_1]]) ?
  Call: (7) true ?
  Exit: (7) true ?
  Exit: (5) @(p_1_coinduction_preflight(S_1),[S_1=[0|S_1]]) ?
  Exit: (4) @(p(S_1),[S_1=[0|S_1]]) ?
  Exit: (1) @(binary::p(S_1),[S_1=[0|S_1]]) ?
X = [0|X] ;
...
```

7 Related Work

The U.T.Dallas research group on coinduction makes available a Prolog meta-interpreter, implemented by Feliks Kluźniak in 2009, that supports both tabling

and coinduction [10]. The meta-interpreter distribution includes example applications for the model checker. Although the meta-interpreter suffers from a comparatively slower performance compared with the Logtalk implementation, the high-level implementation of tabling allows it to solve a wider class of problems, without being dependent on native Prolog tabling support. For problems that do not require tabling, the U.T.Dallas implementation provides a simple program transformer that adds an extra argument (with the stack) to coinductive predicates, thus enabling them to be executed without the overhead of interpretation.

Two Prolog compilers, SWI-Prolog and YAP, include limited support for coinduction, implemented by a library module. The YAP implementation takes advantage of non-portable primitives for destructive assignment for representing the coinduction stack when constructing a proof for a coinductive predicate. The SWI-Prolog implementation uses proprietary hook predicates to access a goal and its parent goal during a proof. Although these choices render the implementations non-portable, they also make them potentially more efficient than a portable implementation such as the one found in Logtalk. Both the SWI-Prolog and YAP implementations only support the most simple form of the coinductive directive where only a predicate indicator can be specified. As in the current Logtalk implementation, stratification of programs mixing non-coinductive predicates and coinductive predicates is neither checked nor enforced.

8 Conclusions and Future Work

Logtalk provides a widely available and portable implementation of coinductive logic programming. It features basic debugging support and includes several examples that are complemented by unit tests. It can be easily used for demoing the basic ideas of coinductive logic programming in the classroom and for solving actual problems. Its implementation avoids meta-interpretation and thus provide good performance for coinduction applications that do not require tabling support.

The current implementation is designed with the intuition is that it can generate, by backtracking, all *basic cycles*, whose combinations account for all possible solutions. If our intuition is correct, it should be possible to derive an *expression* that represents that combination and that can be used for checking or generating any solution. Assuming that deriving such an expression can be soundly accomplished in practice and for any problem, this would provide a potential alternative to all current implementations, which all suffer from the fact that an infinite set of solutions cannot be enumerated in a finite time. Thus, the problem of how to discover all basic cycles and how to combine them in an expression appears to be the most interesting open problems and thus a promising line for future work.

Our plans for better coinduction support in Logtalk, while maintaining or improving portability, are partially dependent on the evolution of the compatible Prolog systems. There are two main issues. First, printing of rational terms,

which is used when printing bindings for solutions to coinductive queries, only works acceptably on SWI-Prolog. For all the other supported back-end Prolog compilers, the bindings printed are often ambiguous. Second, tabling of rational terms. This will enable Logtalk to tackle problems, including some simple ones, that cannot currently be solved. We plan to work closely with Prolog implementers on solving both issues.

We are also following progress on the theoretical aspects of coinduction, specially when combined with constraint programming, and hope to be able to implement new, proven, ideas when feasible and in a timely manner.

Acknowledgements. We thank Gopal Gupta, Feliks Kluźniak, Neda Saeedloei, Brian DeVries, and Kyle Marple for helpful discussions on coinduction during a sabbatical visit to U.T.Dallas and for most of the coinduction examples that are currently adapted and distributed with Logtalk. We also thank Jan Wielemaker and Vitor Santos Costa for feedback on using destructive assignment primitives for representing coinduction stacks and on representing and printing rational terms. This work is partially supported by the LEAP project (PTDC/EIA-CCO/112158/2009), the ERDF/COMPETE Program and by FCT project FCOMP-01-0124-FEDER-022701.

References

1. Gupta, G., Saeedloei, N., DeVries, B.W., Min, R., Marple, K., Kluźniak, F.: Infinite Computation, Co-induction and Computational Logic. In: CALCO. (2011) 40–54
2. Gupta, G., Bansal, A., Min, R., Simon, L., Mallya, A.: Coinductive Logic Programming and Its Applications. In: ICLP, Springer (2007) 27–44
3. Simon, L.: Coinductive Logic Programming. PhD thesis, University of Texas at Dallas, Richardson, Texas (2006)
4. Moura, P.: From Plain Prolog to Logtalk Objects: Effective Code Encapsulation and Reuse (Invited Talk). In Hill, P.M., Warren, D.S., eds.: Proceedings of the 25th International Conference on Logic Programming. Volume 5649 of Lecture Notes in Computer Science., Berlin Heidelberg, Springer-Verlag (July 2009) 23
5. Moura, P.: Logtalk - Design of an Object-Oriented Logic Programming Language. PhD thesis, Department of Computer Science, University of Beira Interior, Portugal (September 2003)
6. ISO/IEC: International Standard ISO/IEC 13211-1 Information Technology — Programming Languages — Prolog — Part I: General core, Corrigenda 2. ISO/IEC (2012)
7. Saeedloei, N., Gupta, G.: Coinductive Constraint Logic Programming. In Schrijvers, T., Thiemann, P., eds.: FLOPS. Volume 7294 of Lecture Notes in Computer Science., Springer (2012) 243–259
8. Saeedloei, N.: Modeling and Verification of Real-Time and Cyber-Physical Systems. PhD thesis, University of Texas at Dallas, Richardson, Texas (2011)
9. Saeedloei, N., Gupta, G.: Verifying complex continuous real-time systems with coinductive CLP(R). In: LATA. (2010) 536–548
10. Kluźniak, F.: Metainterpreter supporting tabling (DRA) and coinduction with applications to LTL model checking. `http://www.utdallas.edu/~gupta/meta.html`