TLP **14** (4-5): 429–443, 2014. © Cambridge University Press 2014
doi:10.1017/S147106841400012X

Tabling, Rational Terms, and Coinduction Finally Together!

THEOFRASTOS MANTADELIS, RICARDO ROCHA and PAULO MOURA

CRACS & INESC TEC, Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
(*e-mail:* {theo.mantadelis,ricroc}@dcc.fc.up.pt,pmoura@inescporto.pt)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Tabling is a commonly used technique in logic programming for avoiding cyclic behavior of logic programs and enabling more declarative program definitions. Furthermore, tabling often improves computational performance. Rational term are terms with one or more infinite subterms but with a finite representation. Rational terms can be generated in Prolog by omitting the occurs check when unifying two terms. Applications of rational terms include definite clause grammars, constraint handling systems, and coinduction. In this paper, we report our extension of YAP's Prolog tabling mechanism to support rational terms. We describe the internal representation of rational terms within the table space and prove its correctness. We then use this extension to implement a tabling based approach to coinduction. We compare our approach with current coinductive transformations and describe the implementation. In addition, we present an algorithm that ensures a canonical representation for rational terms.

KEYWORDS: Tabling, Rational Terms, Coinduction, Implementation.

1 Introduction

Tabling (Chen and Warren 1996) is a recognized and powerful implementation technique that solves some limitations of Prolog's operational semantics in dealing with left-recursion and redundant sub-computations. Tabling based models are able to reduce the search space, avoid looping, and always terminate for programs satisfying the *bounded term-size property*[1]. Tabling consists of saving and reusing the results of sub-computations during the execution of a program. This is accomplished by storing the calls and the answers to tabled subgoals in a proper data structure called the *table space*.

From as early as (Colmerauer 1982; Jaffar and Stuckey 1986), Prolog implementers have chosen to omit the *occurs check* in unification. This has resulted in generating cyclic terms known as *rational terms* or *rational trees* in Prolog. While the

Technical results can be found in the supplementary material online

[1] A logic program has the bounded term-size property if there is a function $f : N \rightarrow N$ such that whenever a query goal Q has no argument whose term size exceeds n, then no term in the derivation of Q has size greater than $f(n)$.

https://doi.org/10.1017/S147106841400012X Published online by Cambridge University Press

introduction of cyclic terms in Prolog was unintentional, soon after applications for cyclic terms emerged in fields such as definite clause grammars (Colmerauer 1982; Giannesini and Cohen 1984), constraint programming (Meister and Frühwirth 2006; Bagnara *et al.* 2001) and coinduction (Gupta *et al.* 2007). But support for rational terms across Prolog systems varies and often fails to provide the functionality required by most applications. Two common problems are the lack of support for printing query bindings with rational terms and the lack of a tabling mechanism that supports rational terms (Moura 2013). Furthermore, several Prolog features are not designed for compatibility with rational terms and can make programming using rational terms challenging and cumbersome.

In this paper, we first present an extension to the tabling mechanism of YAP Prolog (Santos Costa *et al.* 2012) to support rational terms. To the best of our knowledge, this is the first Prolog built-in tabling system that supports rational terms. The tabling system of XSB (Swift and Warren 2012) handles infinite terms by defining a limit on the term size stored within the table space. While this approach allows tabling to work with goals containing rational terms, it does not store a rational term within the table space and, as a result, the returned answers are not the expected ones. Our approach extents the table space to represent rational terms internally. Consequently, answers containing rational terms are properly stored and returned by the tabling mechanism.

Furthermore, by having extended tabling to support rational terms, we present a novel approach for calculating the *greatest fixed point* of goals by using the internal tabling mechanism. In this way, we propose a novel and efficient transformation to implement coinduction. Previous research presented different solutions to support coinduction (Simon *et al.* 2007; Gupta *et al.* 2007; Ancona 2013; Moura 2013) but, to the best of our knowledge, our approach is the first one that uses a native transformation, allowing a more elegant and efficient implementation. In addition, we present an algorithm that ensures a canonical representation for rational terms.

The remainder of the paper is organized as follows. First in Section 2, we introduce some background concepts about tabling, rational terms and coinduction. Next, Section 3 describes our extension to support rational terms within the tries of the table space. Then, we introduce our proposal to implement coinduction through the internal tabling mechanism in Section 4. Section 5 addresses the issue of rational terms in canonical form and finally, Section 6, concludes by discussing related and future work.

2 Background

In this section, we present some background information on tabling, rational terms, and coinduction.

2.1 Tabling and Tries

The basic idea behind tabling is straightforward: programs are evaluated by storing answers for tabled subgoals in an appropriate data space called the *table space*. Repeated calls to tabled subgoals are resolved whenever possible by consuming the

https://doi.org/10.1017/S147106841400012X Published online by Cambridge University Press

answers already stored in the table space instead of being re-evaluated against the program clauses. During this process, as new answers are found, they are stored in their tables and later returned to all repeated calls.

The design of the table space data structures is critical for achieving an efficient tabling implementation. In YAP, as in most tabling systems, the table space representation is based on *tries* (Ramakrishnan *et al.* 1999). A trie is a tree structure where each different path through the tree nodes corresponds to a term described by the tokens labeling the nodes traversed. For example, the tokenized form of the term `path(X,1,f(Y))` is the sequence of 5 tokens `path/3`, `VAR0`, `1`, `f/1` and `VAR1`, where each variable is represented as a distinct `VARi` constant. Two terms with common prefixes will branch off from each other at the first distinguishing token. Consider, for example, a second term `path(Z,1,b)`. Since the main functor and the first two arguments, tokens `path/3`, `VAR0` and `1`, are common to both terms, only one additional node will be required to fully represent this second term in the trie, thus saving the space that would be taken by three nodes in this case.

YAP implements tables using two levels of tries (Ramakrishnan *et al.* 1999). The first level, the *subgoal trie*, stores the tabled subgoal calls. The second level, the *answer trie*, stores the answers for a given call. More specifically, each tabled predicate has a *table entry* data structure assigned to it, acting as the entry point for the predicate's subgoal trie. Each different subgoal call is then represented as a unique path in the subgoal trie, starting at the table entry and ending in a *subgoal frame* data structure, with the argument terms being stored within the path's nodes. The subgoal frame data structure acts as an entry point to the answer trie. Contrary to subgoal tries, answer trie paths hold just the substitution terms for the free variables that exist in the argument terms of the corresponding call.

An example for a tabled predicate `p/3` is shown in Fig. 1. Initially, the table entry for `p/3` points to an empty subgoal trie. When the subgoal `p(X,1,Y)` is called, three trie nodes are inserted to represent the arguments in the call: one for the variable `X` (`VAR0`), a second for the integer `1`, and third one for the variable `Y` (`VAR1`). Since the predicate's functor term is already represented by its table entry, we can avoid inserting an explicit node for `p/3` in the subgoal trie. Finally, the leaf node is set to point to a subgoal frame, from where the answers for the call will be stored. The example shows two answers for `p(X,1,Y)`: `{X=a, Y=f(VAR0)}` and `{X=a, Y=b}`. Since both answers have the same substitution term for argument `X`, they share the top node in the answer trie. For argument `Y`, each answer has a different substitution term and, thus, a different path is used to represent each.

2.2 Rational Terms

Rational terms, also known as rational trees or cyclic terms, are infinite terms that can be finitely represented. They can include any finite sub-term but have at least

https://doi.org/10.1017/S147106841400012X Published online by Cambridge University Press

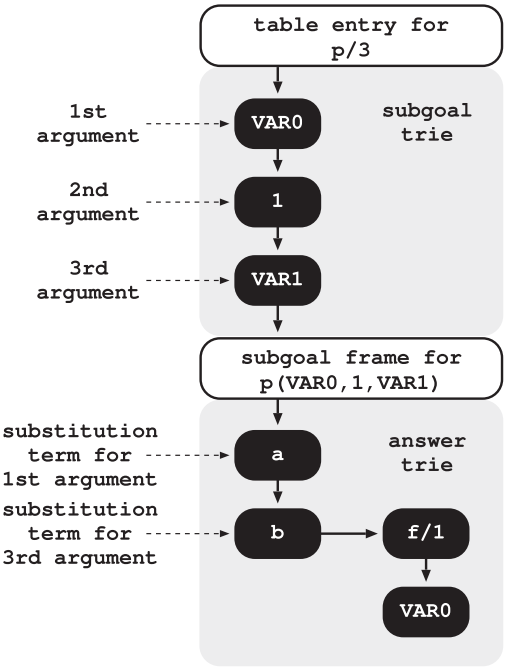


Fig. 1. YAP's table space organization

one infinite sub-term. Rational terms in logic programming appeared as a side effect of omitting the *occurs check* in unification. By omitting the occurs check, unification can return cyclic terms. A simple example is `L=[1|L]`, where the variable `L` is instantiated to an infinite list of ones. Prolog implementers omitted the occurs check in order to reduce the unification complexity from $O(Size_{Term1} + Size_{Term2})$ to $O(min(Size_{Term1}, Size_{Term2}))$, but since then these cyclic terms found application in areas such as coinduction, natural language processing or finite-state automata, among others.

Several Prolog systems support the creation and unification of rational terms. These systems also provide built-in predicate implementations capable of handling rational terms without falling into infinite computations. Still, several common predicates such as `member/2` cannot handle rational terms.

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

Consider the call `L=[1,2|L], member(E, L)`. The second clause of `member/2` spawns a new member call for every next suffix of the infinite list `L`, generating an infinite number of calls and answers for `member/2`. In order to handle the occurring cycles,

https://doi.org/10.1017/S147106841400012X Published online by Cambridge University Press

one could implement the following `member/2` predicate.

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

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

With `in_stack/2` predicate being defined as:

```
in_stack(E, [H|_]) :- E == H.
in_stack(E, [_|T]) :- in_stack(E, T).
```

The second clause of `member/3` checks whether the input list of the member call is repeated. If so, it terminates the execution by removing any left choice points and then fails. Using our example list `L=[1,2|L]`, it will permit only the two unique suffixes of the list `L`. Unfortunately, this is more of an ad-hoc solution instead of a clean scalable solution that deals with generic rational terms.

2.3 Coinduction

Recently, we have seen an increase of interest in coinduction. Several new systematic approaches have appeared and important applications have been noted. co-SLD extends logic programming to allow reasoning over infinite and cyclic structures. For this paper, we follow similar semantics like the ones proposed by (Gupta *et al.* 2007) for co-SLD: the *coinductive hypothesis rule* states that *if while proving a goal G (ancestor goal) a subgoal G' (current goal) is found, such as goal G is a variant of G', then the subgoal G' has coinductive success.*

We have seen several approaches to implement co-SLD with the use of program transformation (Simon *et al.* 2007; Gupta *et al.* 2007; Moura 2013) or by using a meta caller approach (Ancona 2013). While SLD resolution proves a query and returns the answers of the least fixed points, co-SLD resolution succeeds and returns the answers of the greatest fixed points (Simon *et al.* 2007). As an example, consider the following logic program and the query `bin(X)`.

```
bin([]).
bin([0|T]) :- bin(T).
bin([1|T]) :- bin(T).
```

SLD resolution recognizes all finite lists containing only `0`s and `1`s. Now, if we remove the base case, the program does not have a least fixed point and, as a consequence, SLD resolution will not be able to solve it. As pointed out by (Gupta *et al.* 2007), there are two reasons why the program is not solved. First, because the Herbrand universe does not allow infinite terms and, second, because the Herbrand universe does not allow infinite proofs. co-SLD allows infinite terms and infinite proofs, as a result the query `bin(X)` succeeds and resolution returns two answers that represent the minimal greatest fixed points, `{X=[0|X], X=[1|X]}`. However, any

https://doi.org/10.1017/S147106841400012X Published online by Cambridge University Press

infinite pattern of 0s and 1s that can be expressed as a rational term is recognized by the bin/1 coinductive predicate, as exemplified below.

```
% generating solutions for the coinductive bin/1 predicate:
Query:   ?- bin(X).
Answers: X = [0|X] ? ;
         X = [1|X].

% recognizing solutions for the coinductive bin/1 predicate:
Query:   ?- X = [0,1,0,1,0,0,0|X], bin(X).
Answer:  X = [0,1,0,1,0,0,0|X].
```

3 Extending Tabling to Support Rational Terms

To the best of our knowledge, no current tabling system supports rational terms. A tabled subgoal call or tabled answer containing rational terms would result in an infinite computation. The reason for this behavior is that the table space data structures were not designed to support the finite storing of infinite terms. YAP is thus the first tabling system that handles rational terms by extending the table space to support the internal representation of infinite terms.

Consider, for example, the definition given in (Gupta *et al.* 2007) for a comember/2 coinductive predicate:

```
% comember(Element, List) is true iff Element appears infinite times in List
:- coinductive(comember/2).
comember(H, L) :- drop(H, L, L1), comember(H, L1).

% drop(Element, List, Rest) is true iff Element can be dropped from List resulting
% in the list Rest
:- table(drop/3).
drop(H, [H|T], T).
drop(H, [_|T], T1) :- drop(H, T, T1).
```

Without tabling with support for rational terms, the auxiliary predicate drop/3 can not be used by the comember/2 predicate to enumerate all the distinct elements of a (rational) list. Only the first element would be enumerated. To avoid tabling, one could write drop/3 as follows, by using the in_stack/2 auxiliary predicate, but certainly writing such a procedural predicate is not in the spirit of logic programming. Similarly, the solution presented by (Ancona 2013) that avoids tabling by using coinductive success hook predicates requires from the programmer a good understanding of the coinductive meta-interpreter or coinductive transformation mechanics. Ancona's coinductive success hook predicates are not illustrated here for brevity. The crucial concept behind them, is to enable the programmer to modify the behavior when a cycle is detected.

```
drop(E, L, NL) :- drop(E, L, NL, []).
drop(_, L, _, S) :- in_stack(L, S), !, fail.
drop(E, [E|T], T, _).
drop(E, [H|T], T1, S) :- drop(E, T, T1, [[H|T]|S]).
```

Motivated by the above example and the goal to elegantly and efficiently support coinduction for YAP at the low level engine, our first step was to extend YAP's

https://doi.org/10.1017/S147106841400012X Published online by Cambridge University Press

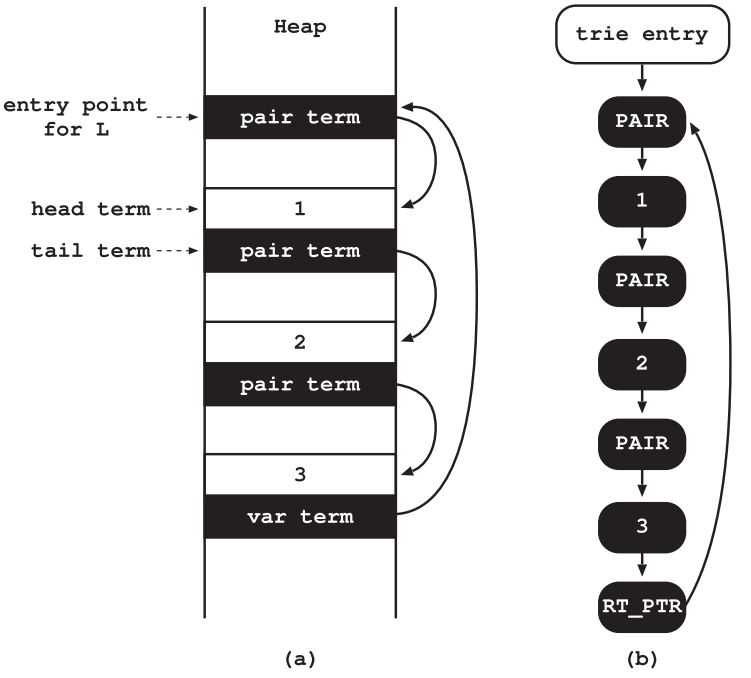


Fig. 2. YAP's representation for the cyclic term `L=[1,2,3|L]` in the **(a)** heap and **(b)** the trie

internal tries representation to support rational terms. So far, tries do not support cyclic terms and inserting one such term in a trie results in trapping the program in a cycle. In order for tries to support cyclic terms, we first defined a representation of cyclic tokens within the tries; we then extended the trie check/insert algorithm to detect the tokens where a cycle occurs and at which upper trie token it should point to; last, we adjusted the mechanism that allows to reconstruct a term from the stored tokens within the trie to also handle cyclic terms.

To better explain our trie representation for cyclic terms let's have a look at the heap representation. At the low level engine, a cycle in YAP appears as a *variable term* cell unified with the term that is repeated. Figure 2(a) shows YAP's internal representation for the cyclic term `L=[1,2,3|L]` as it is stored in the heap.

Current trie definitions do not store pointer tokens by default. We extend the trie definition so that it can store a pointer token that points to a trie node. As no other trie nodes are storing pointers, we can safely assume that such a pointer token represents a rational term cycle. In order to represent the cycle within the trie, we store in the node of the cyclic token a pointer to the trie node of the pointed token. Figure 2(b) illustrates the cyclic term `L=[1,2,3|L]` as it is stored in the trie representation[2] with `RT_PTR` illustrating the rational term pointer token. In order for our approach to be safe we consider Proposition 1.

[2] For the sake of simplicity, we describe YAP's original trie implementation for lists. In fact, our current implementation extends the optimized *compact lists* representation for tries as described in (Raimundo and Rocha 2010).

https://doi.org/10.1017/S147106841400012X Published online by Cambridge University Press

Proposition 1
Cycles in a Prolog term can only point at a list (pair) term or a functor term.

Proof
If a variable term cell points to an atomic term or to an unbound variable then it would be dereferenced to that atomic term or unbound variable. □

We detect the cycles upon the insertion of a term in the trie. To do so, we first memorize the heap address of every list or functor term that we encounter *tuppled* with the address of the trie node that represents the list or functor token in the trie. Before inserting a new list or functor token in the trie, we check whether we have already inserted a node for it. If so, we return the address of the corresponding trie node and create a new trie node containing a pointer token with the address of the returned trie node.

Besides storing a cyclic term, the tabling system must also be able to return the same cyclic term when asked. In YAP Prolog, to reconstruct a term from the trie, the tabling mechanism reads the trie branch bottom up and constructs the subterms stored in each token incrementally directly in the Prolog heap. The original term reconstruction mechanism assumes that no reference pointers are stored within the trie nodes. As the construction of the term is bottom up our system does not know the reference of a cycle before fully constructing the term. In order to tackle this issue when a reference pointer is found, we create a new unbound variable in the heap and memorize the heap address and the trie node that has the token which will be the entry point of the term that this variable must be unified. When the term has been fully constructed, we unify that unbound variable with the term generated from the corresponding trie node that the pointer token is pointing at. In order for our approach to be safe we considered Proposition 2.

Proposition 2
When traversing a Prolog term from the heap entry point, cyclic references point at preceding heap locations.

Proof
Indeed, if while traversing a Prolog term, starting from the entry point, a variable term cell points to a following heap location it does not introduce a cycle. □

Returning to our `member/2` example from Section 2.2, we now present an alternative more elegant solution to handle lists with rational terms.

```
:- table(member/2).
member(E, [E|_]).
member(E, [_|T]) :- member(E, T).
```

Please note that tabling not only remembers answers but also handles introduced cycles. Similarly, any predicate that needs to process a list which contains rational terms can use tabling to avoid using a similar mechanism like the one presented at Section 2.2.

https://doi.org/10.1017/S147106841400012X Published online by Cambridge University Press

4 A Tabling Based Approach to Coinduction

In this section, we describe a modified tabling strategy that instead of computing the least fixed point of a logic program, computes the greatest fixed point. In order to compute the greatest fixed point of a logic program, it is customary to either define a program transformation (Simon *et al.* 2007; Gupta *et al.* 2007; Moura 2013) or implement a meta-interpreter (Ancona 2013) that solves the logic program. Both approaches have been presented in the past and, to the best of our knowledge, there is no previous implementation that computes the greatest fix point within the WAM.

In order to better describe our tabling based approach, we first present in Alg. 1 the program transformation described in (Moura 2013). There are three key features in the transformation: (i) it uses a stack to keep track the ancestor calls (also called coinductive hypothesis) of the goal that the greatest fix point is under computation; (ii) the stored ancestor calls in the stack are then used for detecting cycles (for coinduction purposes a cycle occurs when an ancestor call unifies with the current call) in the execution of the goal; and (iii) when a cycle is detected, it is handled as a success and unifies the called goal with the repeated ancestor goal.

Input: a coinductive logic predicate `p/1`

```
:- coinductive(p/1).
p(Args) :-
    % the body/2 predicate abstracts the processing of the predicate arguments
    body(Args, NewArgs),
    p(NewArgs).
```

Output: `p/1` transformed to call an auxiliary predicate that implements coinduction

```
p(Args) :-
    % start with an empty stack of coinductive hypothesis
    p(Args, []).
p(Args, Stack) :-
    body(Args, NewArgs),
    p_coinduction_perflight(NewArgs, Stack).
p_coinduction_perflight(Args, Stack) :-
    (   member(p(Args), Stack) *->
        % coinductive success;
        % the (*->)/2 soft-cut construct supports finding of multiple solutions
        true
    ;   % add current goal to the stack of coinductive hypothesis and continue
        p(Args, [p(Args)|Stack])
    ).
```

Alg. 1: The coinductive program transformation of (Moura 2013).

Algorithm 1 is akin both to the cycle handling used in the PTTP (Stickel 1988) and in tabling systems. Tabling memoizes subgoals in a table and checks whether there is a repeated call in order to break the cycle, thus allowing cyclic programs to terminate. Conceptually, the subgoal memoization mechanism of tabling used to perform cycle detection is similar to the collection of calls in the stack used by Alg. 1. One slight difference is that Alg. 1 is using unification to detect repeating goals and YAP's tabling mechanism uses *variant checking*[3], which may alter the

[3] Two goals are said to be variants if they are the same up to variable renaming.

https://doi.org/10.1017/S147106841400012X Published online by Cambridge University Press

```
:- coinductive(bin/1).
bin([0|T]) :- bin(T).
bin([1|T]) :- bin(T).

Query:   ?- bin(L).
Answers: L = [0|L] ? ;
         L = [1|L].
Query:   ?- X=[0,1,0,0|X], bin(X).
Answer:  X = [0,1,0,0|X].
```

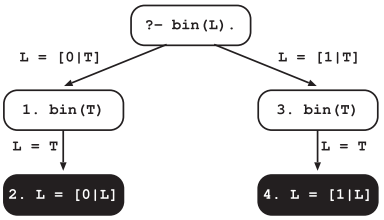


Fig. 3. The coinductive predicate `bin/1` and its coinductive SLG tree

depth of derivations needed to detect the coinductive cycle. To compute the greatest fixed point of a logic program by reusing the mechanism of tabling, we thus noticed that we need to do two main modifications to YAP's tabling mechanism. First, we need to modify the behavior of the cycle detection mechanism in order to succeed the subgoal when a cycle was detected and, second, we need to unify the arguments of the original call of the goal with the arguments of the repeated subgoal where the cycle was detected. While the first modification is easy to motivate, the second modification is not so obvious. To clarify the function and necessity of the second modification, we use the example shown in Fig. 3.

In this example, the unification of the query goal `bin(L)` with the head of both clauses for `bin/1` leaves the tail `T` of the list unbound (`L=[0|T]` and `L=[1|T]` for the first and second clause, respectively). The cycle is then detected when the subgoal `bin(T)` is called next (steps 1 and 3 in Fig. 3). Unifying the arguments for the initial query goal `bin(L)` with the subgoal `bin(T)` leads to unifying `L` with `T`, which results in the answers `L=[0|L]` and `L=[1|L]` for the query goal (steps 2 and 4 in Fig. 3).

Now that we have identified the necessary behavior for computing the greatest fixed point with co-SLG, we present how it is actually implemented. As explained in section 2.1, the tabling mechanism retains a state called subgoal frame for each subgoal call that is proving. In particular, this includes the *substitution factor* (Ramakrishnan *et al.* 1999) for the subgoal at hand, i.e., the set of free variables which exist within the term arguments of the subgoal call. For example, the substitution factor for the subgoal call `p(X,1,f(Y))` is the triplet `<2,X,Y>`, where the first entry indicates the number of variables. The substitution factor is stored on the choice point stack during the process of checking/inserting the call in the subgoal trie for the predicate at hand. The substitution factor is then used to reduce the number of (substitution) terms to copy into and out of the answer tries.

During evaluation, when a tabled subgoal is first called, it is marked as a *generator call* and tagged as *evaluating*. At the low level engine, that corresponds to storing a generator choice point on top of the substitution factor for the subgoal. Later, if a subgoal tagged as evaluating is encountered again then tabling treats it as a cycle. Exactly at that part of the tabling mechanism we intervene. Please note that at that point, we already have the substitution factor for the repeated call also on the choice point stack. Our task is then to unify both substitution factors in order to perform the needed unification for the coinductive strategy. Figure 4 illustrates

https://doi.org/10.1017/S147106841400012X Published online by Cambridge University Press

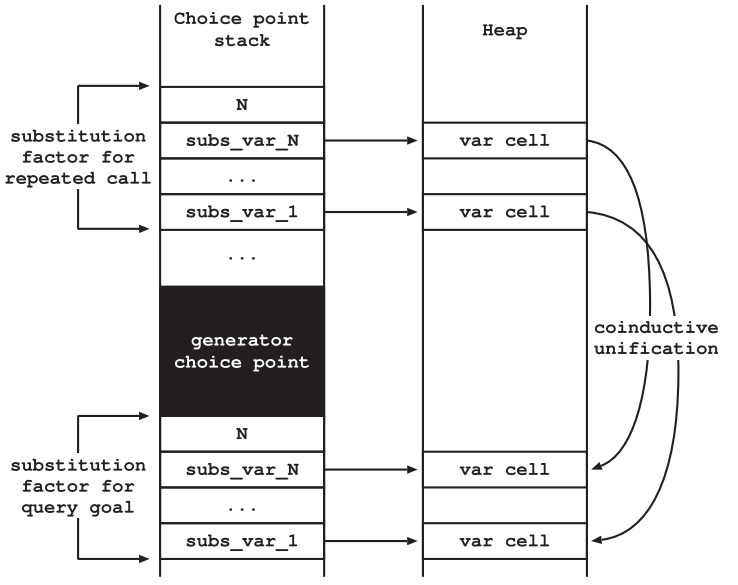


Fig. 4. Using the substitution factors to implement coinductive unification

how this mechanism works and how we use the substitution factors to implement coinductive unification. It presents the choice point and heap stacks at the moment a cycle is found and after performing coinductive unification. This unification will always succeed as YAP uses variant checking in order to identify cycles. The next step is to follow the substitution terms for the generator choice point and insert the resulting unification as an answer in the table space for the generator call. Finally, the execution succeeds and returns the answer.

5 Canonical Representation for Rational Terms

In this section, we address the known issue of rational terms and their canonical or minimal representation (Colmerauer 1982). For example, the rational terms `A=[1|A]`, `B=[1,1|B]`, `C=[1|A]` are all both unifiable and equal:

```
?- A=[1|A], B=[1,1|B], C=[1|A], A=B, B=C, A==B, B==C.
% SWI Prolog Result:
A = B, B = C, C = [1|_S1], % where
    _S1 = [1|_S1].
% YAP Prolog Result:
Stable version:      A = B = C = [1|**].
Development version: A=[1|A], B=[1,1|B], C=[1|A].
```

While all three lists represent the same rational term, their underlying WAM representation in the heap may not be the same[4]. Figure 5 presents YAP's

[4] We have verified that the terms have different heap representation in SWI and YAP Prolog systems.

https://doi.org/10.1017/S147106841400012X Published online by Cambridge University Press

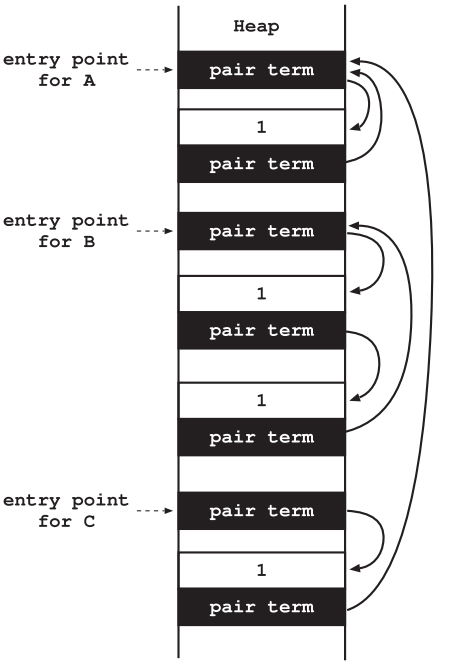


Fig. 5. Heap representation of rational terms `A=[1|A]`, `B=[1,1|B]`, `C=[1|A]`

representation of each of the lists in the heap. Notice that all three lists have a different representation. This particularity of rational terms, which does not appear in normal terms, imposes a limitation for tabling. All these representations appear as different terms for tabling and create different subgoals and answers within the table space. As a result a tabled predicate may table more subgoals than needed or succeed multiple times for an answer that should succeed only once.

In order to address the problem of the extra answers, a possible solution would be to perform a kind of answer subsumption. But a more elegant solution would be to implement an algorithm that converts a rational term to its canonical form. In this section, we present such an algorithm. Predicate `canonical_term/2` presented in Alg. 2 takes any term and converts its rational parts to their canonical form.

The idea behind the algorithm is to first fragment the term to its cyclic subterms, continue by reconstructing each cyclic subterm (now acyclic) and, finally, reintroduce the cycle to the reconstructed subterms. To reconstruct each cyclic subterm as acyclic, the algorithm copies the unique parts of the term and introduces an unbound variable instead of the cyclic references. Then, the algorithm binds the unbound variable to the reconstructed subterm, recreating the cycle. Take for example the rational term `L=[1,2,1,2|L]`, term `L` is being fragmented in the following subterms `L0=[1|L1]`, `L1=[2|L3]`, `L3=[1,2|L0]`. We do not need to fragment the term `L3` as, at that point, our algorithm detects a cycle and replaces term `L3` with an unbound variable `OpenEnd`. Thus we get the following subterms `L0=[1|L1]` and `L1=[2|OpenEnd]`. Binding `OpenEnd=L0` results to the canonical rational term `L0=[1,2|L0]`.

https://doi.org/10.1017/S147106841400012X Published online by Cambridge University Press

Input: a rational term `Term`
Output: a rational term `Canonical` in canonical representation

```
1  canonical_term(Term, Canonical) :-
2      Term =.. InList,
3      decompose_cyclic_term(Term, InList, OutList, OpenEnd, [Term]),
4      Canonical =.. OutList,
5      Canonical = OpenEnd.
6
7  decompose_cyclic_term(_CyclicTerm, [], [], _OpenEnd, _Stack).
8  decompose_cyclic_term(CyclicTerm, [Term|Tail], [Term|NewTail], OpenEnd, Stack) :-
9      acyclic_term(Term), !,
10     decompose_cyclic_term(CyclicTerm, Tail, NewTail, OpenEnd, Stack).
11 decompose_cyclic_term(CyclicTerm,[Term|Tail],[OpenEnd|NewTail],OpenEnd,Stack) :-
12     CyclicTerm == Term, !,
13     decompose_cyclic_term(CyclicTerm, Tail, NewTail, OpenEnd, Stack).
14 decompose_cyclic_term(CyclicTerm,[Term|Tail],[Canonical|NewTail],OpenEnd,Stack) :-
15     \+ in_stack(Term, Stack), !,
16     Term =.. InList,
17     decompose_cyclic_term(Term, InList, OutList, OpenEnd2, [Term|Stack]),
18     Canonical =.. OutList,
19     (   Canonical = OpenEnd2,
20         Canonical == Term,
21         !
22     ;   OpenEnd2 = OpenEnd
23     ),
24     decompose_cyclic_term(CyclicTerm, Tail, NewTail, OpenEnd, Stack).
25 decompose_cyclic_term(CyclicTerm,[_Term|Tail],[OpenEnd|NewTail],OpenEnd,Stack) :-
26     decompose_cyclic_term(CyclicTerm, Tail, NewTail, OpenEnd, Stack).
```

Alg. 2: Predicate `canonical_term/2`

The bulk of the algorithm is found at the fourth clause of `decompose_cyclic_term/5`. At that part we have detected a cyclic subterm that we have to treat recursively. In particular, lines 19–23 implement an important step. Returning to our example when the cycle is detected, the algorithm returns the unbound variable to each fragmented subterm. First, the subterm `L1=[2|OpenEnd]` appears and the algorithm needs to resolve whether it must unify `OpenEnd` with `L1` or whether `OpenEnd` must be unified with a parent subterm. In order to verify that, lines 19–23 of the algorithm unify the subterm with the unbound variable and after attempt to unify the created rational term with the original rational term. For our example the algorithm generates `L1=[2|L1]` and attempt to unify with `L=[1,2,1,2|L]`, as the unification fails the algorithm propagates the unbound variable to be unified with the parent subterm `L0=[1|L1]`.

The fifth clause of `decompose_cyclic_term/5` is the location where a cycle is actually found. At that point we can drop the original cyclic subterm and place an unbound variable within the newly constructed term. The third clause of `decompose_cyclic_term/5` could be omitted; it operates as a shortcut for simplifying rational terms of the form `F=f(a,f(a,F,b),b)`. The rest of the algorithm is pretty much straightforward, the first and second clause of `decompose_cyclic_term/5` are the termination condition and a copy of the non-rational parts of the term to the new term respectively.

6 Conclusions, Related and Future Work

In this paper, we presented a novel approach that permits the insertion of rational terms in tries and we extended tabling over goals with rational terms. Rational terms are used in several fields such as coinduction, definite clause grammars,

https://doi.org/10.1017/S147106841400012X Published online by Cambridge University Press

constrain programming, lazy evaluation, finite state automata, etc. Our second novel contribution is a native-to-WAM coinductive transformation. We introduced co-SLG by presenting a tabling transformation that computes the greatest fixed point that provides an efficient elegant solution to coinduction. Finally, we provided an ISO Prolog approach that converts any rational term to its canonical form.

As this work spans over several different fields of logic programming, one finds related work from a variety of sources. In tabling, XSB Prolog (Swift and Warren 2012) has limited support for infinite terms within XSB's tabling mechanism. The user can set a limit for the length of a term and define an action among *error*, *warning* or *abstract*. When the limit is reached the system performs the chosen action. This mechanism, while it can ensure a cyclic safe program, is not suitable to handle rational terms as entities neither it is suitable to be used for coinduction purposes.

In the area of coinduction, we find similarities in the work of (Simon *et al.* 2007; Gupta *et al.* 2007; Moura 2013) that makes clear the importance of tabling for coinduction. (Ancona 2013) presents an alternative approach to coinduction where tabling is not required, still this approach may hurt readability of the coinductive predicate definitions. Coinduction for functional languages is being a topic of research for some time now (Gordon 1994) and it is been used to prove properties of lazy streams.

The problem of equivalence for finite automata (J. E. Hopcroft and Karp 1971) is similar with the problem of reducing a rational term to its canonical form. Recently, (Schrijvers and Oliveira 2012) presented an approach to check equality of rational terms in functional programming languages. Finally, SWI Prolog (Wielemaker *et al.* 2012) implements a `term_factorized/3` predicate that collects all subterms that are used multiple times in a term and substitutes them by variables. While `term_factorized/3` serves a different purpose, the generated subterms are in canonical form and one could use this predicate to create the canonical form of a rational term.

As this is the first tabling mechanism that supports rational terms there is room for improvements and future work. On the tabling side, we should migrate our extension to work with more tabling strategies and options. On the side of rational terms, we intent to further investigate transforming rational terms to their canonical form.

Acknowledgments

The authors want to thank Vítor Santos Costa for his suggestions and technical support. We also want to thank the anonymous reviewers for their comments and help to improve our paper. This work is partially funded by the ERDF (European Regional Development Fund) through the COMPETE Programme and by FCT (Portuguese Foundation for Science and Technology) within projects SIBILA (NORTE-07-0124-FEDER-000059) and PEst (FCOMP-01-0124-FEDER-037281).

https://doi.org/10.1017/S147106841400012X Published online by Cambridge University Press

References

Ancona, D. 2013. Regular Corecursion in Prolog. *Computer Languages, Systems & Structures 39,* 4, 142–162. Special issue on the Programming Languages track at the 27th ACM Symposium on Applied Computing.

Bagnara, R., Gori, R., Hill, P. M., and Zaffanella, E. 2001. Finite-Tree Analysis for Constraint Logic-Based Languages: The Complete Unabridged Version.

Chen, W. and Warren, D. S. 1996. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM 43,* 1, 20–74.

Colmerauer, A. 1982. Prolog and Infinite Trees. In *Logic Programming*, K. L. Clark and S.-A. Tärnlund, Eds. Academic Press, 231–251.

Giannesini, F. and Cohen, J. 1984. Parser generation and grammar manipulation using prolog's infinite trees. *The Journal of Logic Programming 1,* 3, 253 – 265.

Gordon, A. D. 1994. A Tutorial on Co-induction and Functional Programming. In *Glasgow Functional Programming Workshop*. Springer-Verlag, 78–95.

Gupta, G., Bansal, A., Min, R., Simon, L., and Mallya, A. 2007. Coinductive logic programming and its applications. In *Logic Programming*. LNCS, vol. 4670. Springer-Verlag, 27–44.

J. E. Hopcroft, J. E. and Karp, R. M. 1971. A linear algorithm for testing equivalence of finite automata. Tech. rep., Cornell University.

Jaffar, J. and Stuckey, P. J. 1986. Semantics of Infinite Tree Logic Programming. *Theoretical Computer Science 46,* 0, 141–158.

Meister, M. and Frühwirth, T. 2006. Complexity of the CHR rational tree equation solver. In *Constraint Handling Rules*. Vol. 452. 77–92.

Moura, P. 2013. A Portable and Efficient Implementation of Coinductive Logic Programming. In *International Symposium on Practical Aspects of Declarative Languages*. LNCS, vol. 7752. Springer-Verlag, 77–92.

Raimundo, J. and Rocha, R. 2010. Compact Lists for Tabled Evaluation. In *International Symposium on Practical Aspects of Declarative Languages*. Number 5937 in LNCS. Springer-Verlag, 249–263.

Ramakrishnan, I. V., Rao, P., Sagonas, K., Swift, T., and Warren, D. S. 1999. Efficient Access Mechanisms for Tabled Logic Programs. *Journal of Logic Programming 38,* 1, 31–54.

Santos Costa, V., Rocha, R., and Damas, L. 2012. The YAP Prolog System. *Journal of Theory and Practice of Logic Programming 12,* 1 & 2, 5–34.

Schrijvers, T. and Oliveira, B. C. 2012. Rational Term Equality, Functionally. In *Implementation and Application of Functional Languages, 24th Symposium, Pre-Proceedings*, R. Hinze, Ed.

Simon, L., Bansal, A., Mallya, A., and Gupta, G. 2007. Co-Logic Programming: Extending Logic Programming with Coinduction. In *Automata, Languages and Programming*. Lecture Notes in Computer Science, vol. 4596. Springer-Verlag, 472–483.

Stickel, M. E. 1988. A prolog technology theorem prover: Implementation by an extended prolog compiler. *Journal of Automated Reasoning 4,* 4, 353–380.

Swift, T. and Warren, D. S. 2012. XSB: Extending Prolog with Tabled Logic Programming. *Theory and Practice of Logic Programming 12,* 1 & 2, 157–187.

Wielemaker, J., Schrijvers, T., Triska, M., and Lager, T. 2012. SWI-Prolog. *Journal of Theory and Practice of Logic Programming 12,* 1 & 2, 67–96.

https://doi.org/10.1017/S147106841400012X Published online by Cambridge University Press