A Portable Prolog Predicate for Printing Rational Terms

Theofrastos Mantadelis and Ricardo Rocha

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

Abstract. Rational terms or rational trees are terms with one or more infinite sub-terms but with a finite representation. Rational terms appeared as a side effect of omitting the *occurs check* in the unification of terms, but their support across Prolog systems varies and often fails to provide the expected functionality. A common problem is the lack of support for printing query bindings with rational terms. In this paper, we present a survey discussing the support of rational terms among different Prolog systems and we propose the integration of a Prolog predicate, that works in several existing Prolog systems, in order to overcome the technical problem of printing rational terms. Our rational term printing predicate could be easily adapted to work for the top query printouts, for user printing and for debugging purposes.

Keywords: Rational Terms, Implementation, Portability.

1 Introduction

From as early as [3, 8], 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*. Rational terms are infinite terms that can be finitely represented, i.e., they can include any finite sub-term but have at least one infinite sub-term. A simple example is `L=[1|L]`, where the variable `L` is instantiated to an infinite list of ones. Prolog implementers started omitting the occurs check in order to reduce the unification complexity from $O(Size_{Term1} + Size_{Term2})$ to $O(min(Size_{Term1}, Size_{Term2}))$.

While the introduction of cyclic terms in Prolog was a side effect of omitting the occurs check, soon after applications for cyclic terms emerged in fields such as definite clause grammars [3, 5], constraint programming [10, 2], coinduction [6, 1, 11, 12] or infinite automata [7]. But support for rational terms across Prolog systems varies and often fails to provide the functionality required by most applications. A common problem is the lack of support for printing query bindings with rational terms [11]. 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 address the problem of printing rational terms for a large number of Prolog systems. We thus propose a compliant with ISO Prolog predicate that can be used in several Prolog systems in order to print rational terms. The predicate functions properly in the Ciao, SICStus, SWI, XSB and YAP Prolog systems. The predicate was also tested with the BProlog, ECLiPSe, GNU Prolog, Jekejeke and Strawberry Prolog systems but for different reasons it failed to work (details about the Prolog versions tested are presented next).

The remainder of the paper is organized as follows. First, we discuss how rational terms are supported across a set of well-known Prolog systems. Next, we present our compliant with ISO Prolog predicate and discuss how it can be used to improve the printing of rational terms in several Prolog systems. We end by outlining some conclusions.

2 Rational Term Support in Prolog Systems

We tested several Prolog systems to figure out their available support for rational terms. Table 1 presents in brief our results. Initially, we performed ten different tests that we consider to be the very minimal required support for rational terms. First, we tested the ability of Prolog systems to create rational terms via the `=/2` operator (unification without occurs check). Second, and most critical test, was for the systems to be able to perform unification among two rational terms. Third, we checked whether the Prolog systems can infer the equality of two rational terms by using `==/2` operator. Our fourth test was to see whether a Prolog system can de-construct/construct rational terms through the `=../2` operator, we also investigated whether the Prolog system supports any form of build-in printing predicates for rational terms. The results of the above five tests are presented in Table 1(a).

Furthermore, we checked the support of the `acyclic_term/1` ISO predicate [4], we tested whether `assertz/1` supports asserting rational terms, checked if the `copy_term/2` and `ground/1` predicates work with rational terms and finally, we checked `recordz/3` and `recorded/3` functions with rational terms as an alternative for `assert/1`. The results of these tests appear in Table 1(b).

Finally, we performed a few more compatibility tests as we present in Table 1(c). We want to point out that the results of this table are expected and are sub covered by the test for `==/2` operator. We have the strong conviction that the same reason that forbids the `==/2` operator to function with rational terms in some Prolog systems is the same reason for the failure of the comparison support tests.

Currently, only three Prolog systems appear to be suitable for programming and handling rational terms, namely SICStus, SWI and YAP. The rest of the systems do not provide enough support for rational terms, which makes programming with rational terms in such systems challenging and cumbersome, if

[1] At the time of the publication the current stable version of Yap presented a problem with `recorded/3`, but the development version (6.3.4) already had the problem solved.

Prolog System	Create =/2	Compare =/2	Compare ==/2	De-compose/ compose =../2	Build-in Printing
BProlog (8.1)	✓	✗	✗	✓	✗
Ciao (1.14.2)	✓	✓	✓	✓	✗
toplevel query	✓	✓	✗	✗	✗
ECLiPSe (6.1)	✓	✗	✗	✓	✓
GNU (1.4.4)	✓	✗	✗	✓	✗
Jekejeke (1.0.1)	✓	✗	✗	✓	✗
SICStus (4.2.3)	✓	✓	✓	✓	✓
Strawberry (1.6)	✗	✗	✗	✗	✗
SWI (6.4.1)	✓	✓	✓	✓	✓
XSB (3.4.0)	✓	✓	✗	✓	✗
YAP (6.2.3)	✓	✓	✓	✓	✓

(a) Operator Support

Prolog System	acyclic_term/1	assert/1	copy_term/2	ground/1	recordz/3
BProlog (8.1)	✓	✗	✗	✗	✗
Ciao (1.14.12)	✓	✗	✗	✗	✗
toplevel query	✓	✗	✗	✗	✗
ECLiPSe (6.1)	✓	✓	✗	✗	✓
GNU (1.4.4)	✓	✗	✗	✗	✗
Jekejeke (1.0.1)	✗	✗	✗	✗	✗
SICStus (4.2.3)	✓	✓	✓	✓	✓
Strawberry (1.6)	✗	✗	✗	✗	✗
SWI (6.4.1)	✓	✗	✓	✓	✓
XSB (3.4.0)	✓	✗	✗	✗	✗
YAP (6.2.3)	✓	✗	✓	✓	✓[1]

(b) Predicate Support

Prolog System	Compare @>/2	Compare @</2	Compare @>=/2	Compare @=</2	compare/3
BProlog (8.1)	✗	✗	✗	✗	✗
Ciao (1.14.2)	✓	✓	✓	✓	✓
toplevel query	✗	✗	✗	✗	✗
ECLiPSe (6.1)	✗	✗	✗	✗	✗
GNU (1.4.4)	✗	✗	✗	✗	✗
Jekejeke (1.0.1)	✗	✗	✗	✗	✗
SICStus (4.2.3)	✓	✓	✓	✓	✓
Strawberry (1.6)	✗	✗	✗	✗	✗
SWI (6.4.1)	✓	✓	✓	✓	✓
XSB (3.4.0)	✗	✗	✗	✗	✗
YAP (6.2.3)	✓	✓	✓	✓	✓

(c) Comparison Operator Support

Table 1. Rational term support by Prolog systems

not impossible. All Prolog systems we tested appear to support the creation through unification of rational terms. For Jekejeke and Strawberry Prolog, we where not able to verify the correctness of the created rational term but the system appeared to accept the instruction. SICStus, SWI and YAP Prolog systems also provide built-in predicate implementations capable of handling rational terms without falling into infinite computations making them the most suitable systems to work with rational terms.

For printing purposes, Table 1 shows us that only a few Prolog systems are able to print rational terms without problems. The best printing is offered by SWI as illustrated on the following examples:

```
?- A = [1|A].
A = [1|A].

?- B = [2|B], A = [1|B].
B = [2|B],
A = [1|B].

?- A = [1|B], B = [2|B].
A = [1|_S1], % where
    _S1 = [2|_S1],
B = [2|_S1].
```

YAP offers an alternative printing which is ambiguous:

```
?- A = [1|A].
A = [1|**].

?- B = [2|B], A = [1|B].
A = [1,2|**],
B = [2|**].

?- A = [1|B], B = [2|B].
A = [1,2|**],
B = [2|**].
```

ECLiPSe and SICStus print rational terms in the following way:

```
?- A = [1|A].
A = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...]

?- B = [2|B], A = [1|B].
B = [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, ...]
A = [1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, ...]

?- A = [1|B], B = [2|B].
A = [1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, ...]
B = [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, ...]
```

The printed ... from ECLiPSe and SICStus is a result of printing a higher depth term than what the system permits. Both ECLiPSe and SICStus have a depth limit option for their printing which terminates printing resulting to the

partially printed rational terms. Disabling the depth limit, traps those systems in infinite cycles[2].

SICStus and SWI also provides the option `cycles(true)` for `write_term/2` in order to print terms using the finite `@/2` notation. This option returns similar printing output with SWI as the following examples illustrate:

```
?- _A = [1|_A], write_term(_A, [cycles(true)]).
@(_906,[_906=[1|_906]])

?- _A = [1|_B], _B = [2|_B], write_term(_A, [cycles(true)]).
@([1|_1055],[_1055=[2|_1055]])

?- _B = [2|_B], _A = [1|_B], write_term(_A, [cycles(true)]).
@([1|_1055],[_1055=[2|_1055]])
```

One can use this option in SICStus toplevel query printing, by setting appropriatly the Prolog flag `toplevel_print_options`.

GNU Prolog, identifies the term as a rational term and instead prints a message:

```
?- A = [1|A].
cannot display cyclic term for A
```

The rest of the systems get trapped in infinite calculation when printing rational terms. Specifically in the case of Ciao Prolog, we want to point out that the toplevel queries automatically print out unnamed variables making any query we tried to fall in infinite calculation. For that reason the Ciao toplevel is completely unsuitable for rational terms. On the other hand Ciao can run programs with a rather good support of rational terms making it the fourth in the row system to support rational terms.

3 Printing Rational Terms

The predicate `canonical_term/3` presented next at Algorithm 1 was originally designed to transform a rational term to its canonical form [9]. Here, we extended it in order to be able to compute a suitable to print term as its third argument. The predicate does not follow the optimal printing for rational terms but that was not our goal. We present a solution that can with minimal effort be used by several Prolog systems to print rational terms and for that we use the minimum amount of needed facilities.

Before explaining the `canonical_term/3` predicate, let's see some examples by using `canonical_term/3` with the XSB system:

```
?- _A = [a|_A], canonical_term(_A, _, Print).
Print = [a|cycle_at_depth(0)]
```

[2] We where unable to disable the depth limit for ECLiPSe toplevel query printing, but we could do it for non toplevel queries.

```
?- _A = [a|_B], _B = [b|_B], canonical_term(_A, _, Print).
Print = [a,b|cycle_at_depth(1)]

?- _A = [a|_B], _B = [b|_B], _F = f(foo, _A, _B, _F),
   canonical_term(_F, _, Print).
Print = f(foo, [a,b|cycle_at_depth(2)], [b|cycle_at_depth(1)],
        cycle_at_depth(0))
```

Notice that our rational term printing is similar with YAP's printing but instead of printing an ambiguous `**`, we print a special term `cycle_at_depth/1` that indicates at which tree depth of the specific tree branch the cyclic sub-term points at. Figure 1, illustrates the term `f(foo, [a,b|cycle_at_depth(2)], [b|cycle_at_depth(1)], cycle_at_depth(0))` using a tree notation. For illustrative purposes, we replaced `cycle_at_depth/1` with `'**'/1` and we use numbered superscripts to mark the respective tree node that each cyclic sub-term points at.

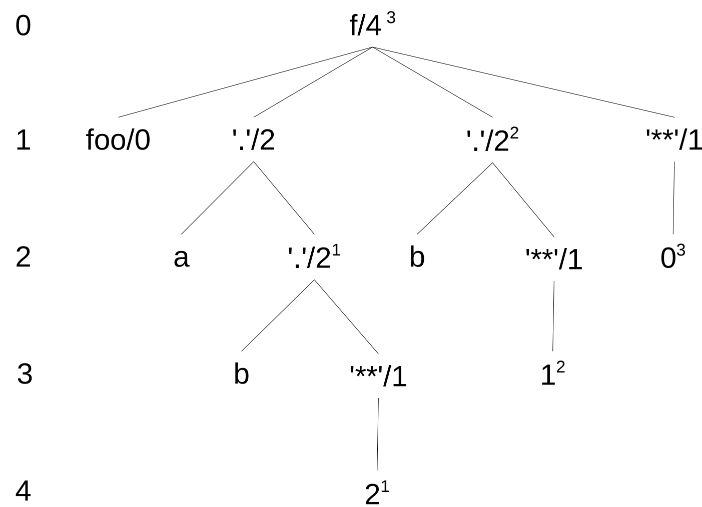

Fig. 1. Rational term: `f(foo, [a,b|cycle_at_depth(2)], [b|cycle_at_depth(1)], cycle_at_depth(0))` in tree notation

While our algorithm is not ambiguous when printing a rational term, it can become ambiguous if the term to be printed also contains `cycle_at_depth/1` terms and the reader of the printed term might falsely think that a cycle exists.

The idea behind the original algorithm as presented at Algorithm 1 is to first fragment the term to its cyclic sub-terms, continue by reconstructing each cyclic sub-term (now acyclic) and, finally, reintroduce the cycle to the reconstructed

sub-terms. To reconstruct each cyclic sub-term 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 sub-term, recreating the cycle.

Take for example the rational term `L=[1,2,1,2|L]`. Term `L` is being fragmented in the following sub-terms: `L0=[1|L1]`, `L1=[2|L3]` and `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 sub-terms: `L0=[1|L1]` and `L1=[2|OpenEnd]`. Binding `OpenEnd=L0` results to the canonical rational term `L0=[1,2|L0]`. One might notice that instead of recreating the cycles, if we bind the `OpenEnd` variables with the special term `cycle_at_depth/1` we get the desirable printout. Furthermore, we keep a counter for each decomposition we do in order to keep track of the tree depth of the term.

The bulk of the algorithm is at the fourth clause of `decompose_cyclic_term/7`. At that part we have detected a cyclic sub-term that we have to treat recursively. In particular, lines 31–37 implement an important step. Returning to our example when the cycle is detected, the algorithm returns the unbound variable to each fragmented sub-term. First, the sub-term `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 sub-term. In order to verify that, lines 31–37 of the algorithm unify the sub-term 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 sub-term `L0=[1|L1]`.

The fifth clause of `decompose_cyclic_term/7` is the location where a cycle is actually found. At that point we can drop the original cyclic sub-term and place an unbound variable within the newly constructed term. The third clause of `decompose_cyclic_term/7` 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 clause of `decompose_cyclic_term/7` is the termination condition and the second clause copies the non-rational parts of the term to the new term.

Our algorithm ensures termination by reaching an empty list on the second clause of `decompose_cyclic_term/7`. This happens as at each iteration of the algorithm the second argument list will be reduced by one element. Cyclic elements are detected and removed and while the list might contain cyclic elements it is not cyclic as it is the decomposed list derived by the `=../2` operator that constructs the originally cyclic term. Finally, the call of `in_stack/2` at line 24 ensures that a cyclic term is not been processed more than once.

Complexity wise, our algorithm behaves linearly to the size of the term in all cases but one. Terms of the form `L = [1,2,3,...|L]` cause the algorithm to have a quadratic complexity ($O(N^2)$). The cause of the worst case complexity is the fourth clause of `decompose_cyclic_term/7`. We are currently considering an

Input: a rational term `Term`
Output: a rational term `Canonical` in canonical representation and `Print` an acyclic term that can be used for printing.

```
1  canonical_term(Term, Canonical, Print) :-
2    Term =.. InList,
3    decompose_cyclic_term(Term, InList, OutList, OpenEnd, [Term],
4                                                PrintList-Cycle_mark, 0),
5    Canonical =.. OutList,
6    Canonical = OpenEnd,
7    Print =.. PrintList,
8    Cycle_mark = cycle_at_depth(0).
9
10 decompose_cyclic_term(_CyclicTerm, [], [], _OpenEnd, _Stack, []-_, _).
11 decompose_cyclic_term(CyclicTerm, [Term|Tail], [Term|NewTail], OpenEnd,
12                      Stack, [Term|NewPrintTail]-Cycle_mark, DepthCount) :-
13   acyclic_term(Term), !,
14   decompose_cyclic_term(CyclicTerm, Tail, NewTail, OpenEnd, Stack,
15                                     NewPrintTail-Cycle_mark, DepthCount).
16 decompose_cyclic_term(CyclicTerm, [Term|Tail], [OpenEnd|NewTail], OpenEnd,
17                Stack, [Cycle_mark|NewPrintTail]-Cycle_mark, DepthCount) :-
18   CyclicTerm == Term, !,
19   decompose_cyclic_term(CyclicTerm, Tail, NewTail, OpenEnd, Stack,
20                                      NewPrintTail-Cycle_mark, DepthCount).
21
22 decompose_cyclic_term(CyclicTerm, [Term|Tail], [Canonical|NewTail],
23             OpenEnd, Stack, [Print|NewPrintTail]-Cycle_mark, DepthCount) :-
24   \+ instack(Term, Stack), !,
25   Term =.. InList,
26   NewDepthCount is DepthCount + 1,
27   decompose_cyclic_term(Term, InList, OutList, OpenEnd2, [Term|Stack],
28                                     PrintList-Cycle_mark_2, NewDepthCount),
29   Canonical =.. OutList,
30   Print =.. PrintList,
31   ( Canonical = OpenEnd2,
32     Canonical == Term,
33     Cycle_mark_2 = cycle_at_depth(NewDepthCount),
34     !
35   ; OpenEnd2 = OpenEnd,
36     Cycle_mark_2 = Cycle_mark
37   ),
38   decompose_cyclic_term(CyclicTerm, Tail, NewTail, OpenEnd, Stack,
39                                      NewPrintTail-Cycle_mark, DepthCount).
40
41 decompose_cyclic_term(CyclicTerm, [_Term|Tail], [OpenEnd|NewTail], OpenEnd,
42                Stack, [Cycle_mark|NewPrintTail]-Cycle_mark, DepthCount) :-
43   decompose_cyclic_term(CyclicTerm, Tail, NewTail, OpenEnd, Stack,
44                                      NewPrintTail-Cycle_mark, DepthCount).
45
46 instack(E, [H|_T]) :- E == H, !.
47 instack(E, [_H|T]) :- instack(E, T).
```

Alg. 1: Predicate `canonical_term/3`

improvement for this, one improvement would be to use a sorted binary tree instead of a list to store and recall the seen cyclic subterms. This improvement would improve the complexity to $(O(N \cdot log(N)))$ but would increase the required build-in support from the Prolog System.

As our target is to print out rational terms at different systems, we had to do a few modifications in order for the predicate to work in other systems. For SWI and YAP, the predicate `canonical_term/3` works as is. For Ciao and SICStus, we only needed to import the appropriate library that contains `acyclic_term/1` and, for XSB, we needed to bypass the lack of support for rational terms of the `==/2` operator by introducing the `compare_rational_terms/2` predicate and replacing the `==/2` operator at lines 1, 21 and 35.

```
% Needed in Ciao to import acyclic_term/1
:- use_module(library(cyclic_terms)).

% Needed in SICStus to import acyclic_term/1
:- use_module(library(terms)).

% Needed in XSB in order to replace ==/2 operator
compare_rational_terms(A, B) :-
  acyclic_term(A),
  acyclic_term(B), !,
  A == B.
compare_rational_terms(A, B) :-
  \+ var(A), \+ var(B),
  \+ acyclic_term(A),
  \+ acyclic_term(B),
  A = B.
```

We want to point out that `compare_rational_terms/2` predicate is not the same with `==/2` predicate and comparisons among terms like: `A = [1,_,2|A]`, `B = [1,a,2|B]` would give wrong results. But for our purpose, where the terms being compared are sub-terms, this problem does not appear as it compares the sub-terms after decomposing them to their smallest units.

4 Towards Optimal Printing of Rational Terms

4.1 SWI

As we earlier pointed out, SWI is the closest to the desirable printing system. However, SWI printing of rational terms suffers from two problems. First, SWI does not print the canonical form of rational terms, as the following example illustrates:

```
?-  A = [1,2|B], B = [1,2,1,2|B].
A = B, B = [1, 2, 1, 2|B].
```

This could be easily corrected by using our predicate to process the rational term before printing it.

The second problem is that SWI can insert auxiliary terms that are not always necessary. For example:

```
?- A = [1|B], B = [2|B].
A = [1|_S1], % where
    _S1 = [2|_S1],
B = [2|_S1].
```

This problem could be addressed with SWI's built-in `term_factorized/3` predicate. Using the same example:

```
?- A = [1|B], B = [2|B], term_factorized((A, B), Y, S).
A = [1|_S1], % where
    _S1 = [2|_S1],
B = [2|_S1],
Y = ([1|_G34], _G34),
S = [_G34=[2|_G34]].
```

Notice that `Y` and `S` contain the desirable printouts. We also want to point out that `term_factorized/3` appears to compute also the canonical form of rational terms which would solve both printing issues. Using again the initial example:

```
?- A = [1,2|B], B = [1,2,1,2|B], term_factorized((A, B), Y, S).
A = B, B = [1, 2, 1, 2|B],
Y = (_G46, _G46),
S = [_G46=[1, 2|_G46]].
```

4.2 YAP

Similarly with SWI, YAP's development version implements a `term_factorized/3` predicate. Future printing of the Yap Prolog system should take advantage of the predicate in order to printout rational terms better.

```
?- A = [1|B], B = [2|B], term_factorized((A, B), Y, S).
A = [1,2|**],
B = [2|**],
S = [_A=[2|_A]],
Y = ([1|_A],_A).

?- A = [1,2|B], B = [1,2,1,2|B], term_factorized((A, B), Y, S).
A = B = [1,2,1,2,1,2,1,2,1,2|**],
S = [_A=[1,2,1,2|_A]],
Y = ([1,2|_A],_A).
```

Notice that YAP's current `term_factorized/3` predicate does not work exactly like SWI's and, currently, it still does not ensure canonical form for rational terms.

4.3 SICStus

SICStus should use the build-in `write_term/2` predicate in order to improve the printing of rational terms. The `write_term/2` predicate appears to both compute the canonical form of the rational term and to generate the minimal needed sub-terms for printing, as the following examples illustrate:

```
?- A = [1|B], B = [2|B], write_term((A, B), [cycles(true)]).
@(([1|_1092],_1092),[_1092=[2|_1092]])
A = [1,2,2,2,2,2,2,2,2,2,2|...],
B = [2,2,2,2,2,2,2,2,2,2|...] ?

?- A = [1,2|B], B = [1,2,1,2|B], write_term((A, B), [cycles(true)]).
@((_1171,_1171),[_1171=[1,2|_1171]])
A = [1,2,1,2,1,2,1,2,1,2|...],
B = [1,2,1,2,1,2,1,2,1,2|...] ?
```

4.4 Ciao

Ciao provides a rather good support of rational terms in comparison with other Prolog systems. However, it has the most problematic toplevel query interface. All queries that would contain rational terms are trapped on an infinite computation and using unnamed variables does not override the problem. The authors believe that this problem is directly related with the printing of rational terms and if Ciao would use a different printing strategy the problem would be solved. Our proposed solution would be an easy way for Ciao to support printing for rational terms. Similarly, printing should be improved also for debugging purposes.

4.5 XSB

XSB imposes several challenges to the programmer to use rational terms. Further than being trapped on infinite computations when trying to print rational terms, it also does not support comparison operators like `==/2`. Regardless of the limitations of the system, we believe that XSB would significantly benefit by using a better printing strategy for rational terms. Similarly, printing should be improved also for debugging purposes.

4.6 Other Prolog Systems

The other Prolog systems that we tried are further away from achieving even the basic rational term support. Even if we were able to print simple rational terms in BProlog, ECLiPSe and GNU Prolog, the lack of support for unification among two rational terms makes it impossible to work with. These systems still treat rational terms as a known bug of unification rather than a usable feature. GNU Prolog in that respect behaved rather well as it identifies rational terms and gives warning messages both when compiling and at runtime. Also, ECLiPSe is not caught in infinite computation and is able to print a representation of rational terms even if the programmer is unable to work with them.

4.7 About `term_factorized/3`

The predicate `term_factorized(+Term, -Skeleton, -Substitution)` is true when: (i) `Skeleton` is the *skeleton* term of `Term`, this means that all subterms of `Term` that appear multiple times are replaced by variables; and (ii) `Substitution` is a list of terms in the form of `VAR = SubTerm` that provides the necessary substitutions to recreate `Term` from `Skeleton` by unifying `VAR = SubTerm`.

The `term_factorized/3` predicate in SWI Prolog is implemented using the red-black tree Prolog library by Vítor Santos Costa. The red-black tree library originally appears in Yap Prolog and is an easy to port in other Prolog systems library. Using `term_factorized/3` for printing rational terms would increase the operators that require to support rational terms to at least: `==/2`, `@</2`, `@>/2`, `compare/3`. For these reasons migrating `term_factorized/3` would be more work than using our `canonical_term/3` predicate.

5 Conclusions

Rational terms, while not being very popular, they have found applications in fields such as definite clause grammars, constraint programming, coinduction or infinite automata. With this paper, we try to motivate Prolog developers to support rational terms better and to provide a minimal support for researchers and programmers to work with. We have presented a short survey of the existing support for rational terms in several well-know Prolog systems and we proposed a printing predicate that Prolog systems could use in order to improve their printing of rational terms.

In particular, Ciao and XSB Prolog systems would benefit the most from our predicate. As we explained, Ciao and XSB fall on infinite computations when they need to print a rational term. Our predicate gives them an easy to integrate solution that will allow printing of rational terms and debugging of code that contains rational terms. Our `canonical_term/3` predicate could also be used in YAP to improve the current ambiguous printing format of rational terms and to present rational terms in their canonical form. SWI could also use our predicate in order to benefit by printing rational terms in canonical form. Still, we believe that both YAP and SWI should do an integration of their `term_factorized/3` predicate with their printing of rational terms. Finally, SICStus can use our predicate to provide an alternative printing, but integrating `write_term/2` predicate on the default printing of terms would be more beneficial.

Acknowledgments

The authors want to thank Paulo Moura and Vítor Santos Costa for their suggestions and technical support. We also want to thank the anonymous reviewers for their constructive and supportive comments that helped us improve this 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 project SIBILA (NORTE-07-0124-FEDER-000059) and PEst (FCOMP-01-0124-FEDER-037281).

References

1. Ancona, D.: Regular Corecursion in Prolog. Computer Languages, Systems & Structures 39(4), 142–162 (2013), Special issue on the Programming Languages track at the 27th ACM Symposium on Applied Computing
2. Bagnara, R., Gori, R., Hill, P.M., Zaffanella, E.: Finite-Tree Analysis for Constraint Logic-Based Languages: The Complete Unabridged Version (2001)
3. Colmerauer, A.: Prolog and Infinite Trees. In: Clark, K.L., Tärnlund, S.A. (eds.) Logic Programming, pp. 231–251. Academic Press (1982)
4. Committee ISO/IEC JTC 1/SC 22: ISO/IEC 13211-1:1995/Cor.2:2012(en): Information technology — Programming languages — Prolog — Part 1: General core TECHNICAL CORRIGENDUM 2 (2012)
5. Giannesini, F., Cohen, J.: Parser generation and grammar manipulation using prolog's infinite trees. The Journal of Logic Programming 1(3), 253 – 265 (1984)
6. Gupta, G., Bansal, A., Min, R., Simon, L., Mallya, A.: Coinductive logic programming and its applications. In: Logic Programming, LNCS, vol. 4670, pp. 27–44. Springer-Verlag (2007)
7. J. E. Hopcroft, J.E., Karp, R.M.: A linear algorithm for testing equivalence of finite automata. Tech. rep., Cornell University (1971)
8. Jaffar, J., Stuckey, P.J.: Semantics of Infinite Tree Logic Programming. Theoretical Computer Science 46(0), 141–158 (1986)
9. Mantadelis, T., Rocha, R., Moura, P.: Tabling, Rational Terms, and Coinduction Finally Together! Journal of Theory and Practice of Logic Programming, International Conference on Logic Programming, Special Issue (2014 to appear)
10. Meister, M., Frühwirth, T.: Complexity of the CHR rational tree equation solver. In: Constraint Handling Rules. vol. 452, pp. 77–92 (2006)
11. Moura, P.: A Portable and Efficient Implementation of Coinductive Logic Programming. In: International Symposium on Practical Aspects of Declarative Languages, LNCS, vol. 7752, pp. 77–92. Springer-Verlag (2013)
12. Simon, L., Bansal, A., Mallya, A., Gupta, G.: Co-Logic Programming: Extending Logic Programming with Coinduction. In: Automata, Languages and Programming, Lecture Notes in Computer Science, vol. 4596, pp. 472–483. Springer-Verlag (2007)