The DRA Interpreter
User Manual

Feliks Kluźniak
Applied Logic, Programming Languages and Systems Lab
Department of Computer Science
University of Texas at Dallas

September 10, 2009

All comments, queries and suggestions about this manual or the software are welcome. The author's e-mail address is `feliks.kluzniak@utdallas.edu`.

Contents

Chapter 1

Introduction

1.1 Overview

This document is a user manual for *dra*, an interpreter for tabled logic programming with coinduction. The manual has been written for users of various versions of Unix: if you are running another system, various small details (such as the way to enter the end-of-file character) may be different.

The interpreter implements "top-down tabled programming" via so called "Dynamic Reordering of Alternatives" [1]. It also supports "co-logic programming", i.e., logic programs that contain coinductive predicates [4], [3].

Apart from support for coinduction, there are two significant changes with respect to the original description [1]:

1. A tabled goal will never produce the same result twice.[1]

   More precisely: a tabled goal will not succeed twice with instantiations that are variants of each other.

2. By default, new results for a tabled goal will be produced before old answers. The user can reverse the order by issuing the directive

   ```
   :- old_first.
   ```

[1] In this document *goal* means an instance of a procedure call (i.e., an invocation of a predicate). A *tabled goal* is a goal that invokes a tabled predicate. A *result* is the instantiation of a goal when it succeeds.

A "new result for a tabled goal" is a result that has not yet been tabled for this goal. (More precisely: a result such that the table does not yet contain a variant of this result associated with a variant of this goal.)

The default behaviour is intended to help computations converge more quickly. The user is given an option to change it, because some predicates may produce a very large (even infinite) set of answers on backtracking, and the application might not require those answers.

1.2 Tabling

By default, each predicate that is defined in the interpreted program is a "normal" Prolog predicate. If you want a predicate (*p/2*, say) to be tabled, you must put an appropriate declaring directive in your program:[2]

```
:- tabled p/2.
```

When you declare a predicate as tabled, there will be two consequences:

1. Every time a goal with this predicate symbol (i.e., an invocation of this predicate) succeeds, the result (i.e., the resulting instantiation of the goal) is stored in a special table called the *answer table*. Such a tabled result is commonly referred to as an *answer*.

   Once all the possible results have been so stored, subsequent invocations can draw upon the tabled answers, instead of recomputing them all over again. This is all done in a way that does not affect the semantics of the program, except perhaps for the order in which results are reported and the number of repetitions of the same result. (Of course, if your program is not sufficiently close to pure Prolog, e.g., if it relies on and/or produces side-effects, then all bets are off.)

2. When a goal that invokes a tabled predicate is found to be a variant of one of its ancestors on the recursion stack, the goal is not expanded.[3]

[2] In literature about logic programming with tabling one often encounters the form `:- table p/2`. Since the directive is almost universally called a declaration (and not a command), the word *tabled* is obviously more appropriate.

[3] Instead, it succeeds with results that have already been put into the table (or fails if there are none). Things are arranged so that no results are lost.

This allows you to write your predicates in a more declarative fashion, without worrying, e.g., about the danger of left recursion.

Tabling is not always as advantageous as one might think, for two reasons:

1. If the number of results is large, and goals are seldom repeated, the main effect might be a significantly increased demand for memory, which is not offset by a shorter processing time.

2. It so happens that in order to preserve the semantics of the program the tabled answers must be associated with the goal that produced them, and are accessible only to variant goals. Other goals that invoke the same predicate will require recomputation, and the computed results will increase the size of the table, even if they are the same as those that have already been stored.

A more extensive discussion of tabling is well outside the scope of this user manual.

1.3 Coinduction

The interpreter supports "old style" coinduction (see [4] and [3]), as well as "new style" coinduction [2].

If you want a predicate (*p/2*, say) to be treated as an "old style" coinductive predicate, you must put the following declaring directive in your program:

```
:- coinductive p/2.
```

Such a declaration has one effect:

- If a goal that invokes a coinductive predicate is found to be unifiable with an ancestor on the recursion stack, the goal is unified with the ancestor and succeeds. Upon backtracking it is unified—one by one[4]—with other such ancestors; it is expanded in the normal way (i.e., by using clauses) only when all the unifiable ancestors have been taken advantage of in this manner.

[4] In *dra* this is done in reverse chronological order, i.e., the more immediate ancestors are used first.

If you want *p/2* to be treated as a "new style" coinductive predicate, use the declaration:

```
:- coinductive1 p/2.
```

This causes the predicate to be treated as above, except that clauses are used only when there are no unifiable ancestors.[5]

The declaration `:- coinductive p/2.` subsumes `:- coinductive1 p/2.` if both are present in the same program. If a predicate is declared as both coinductive in the new style and tabled, then tabled answers are used even for goals that have unifiable ancestors.

Declaring a predicate as coinductive gives it a radically new meaning, which is often the appropriate one for operations on "infinite" terms (represented by cyclic terms). Again, further discussion of the concept is outside the scope of this document.

[5]The mnemonic value of our particular convention is that the "new style" coinductive predicates give you only one way to satisfy a goal.

Chapter 2

The interpreted programs

2.1 Limitations

The interpreter does not support full Prolog. Here are the main limitations of the interpreted language:

1. The interpreted program must not contain cuts (i.e., occurrences of *!/2*). Use of the conditional construct is permitted, as is the use of *once/1*.

2. The interpreted program must not contain variable literals. It may contain invocations of *call/1*, but if the argument of *call/1* is not properly instantiated at runtime, you will get an error message and the interpreter will quit.[1]

3. The repertoire of built-in predicates recognized by the interpreter is somewhat limited. This is done by design, mostly to facilitate porting to different Prolog systems.

   The recognized built-ins are declared in the file `dra_builtins.pl`, and new declarations can be added as the need arises. For most built-ins

[1] In some cases the interpreter can verify beforehand (i.e., at "compile-time") that the argument of an occurrence of *call/1* cannot be instantiated at run-time, and it will then raise a fatal error. The check is quite conservative, so the absence of such an error message does not mean that the program is safe in that respect.

just adding another line to the file will suffice, but a few might require special treatment by the interpreter.[2]

If these limitations seem too strict, you may in some cases get around them by separating your program into two layers: see Sec. 2.2.

2.2 The notion of "support"

The interpreter provides you with an opportunity to divide your program into two layers: an upper layer which makes use of the special facilities provided by the interpreter (i.e., tabling and/or coinduction), and a lower layer of "support" software that requires only standard Prolog. This can be useful for increasing efficiency: the support layer will be compiled just as all other "normal" Prolog programs. An additional advantage is that the support layer can use the full range of built-in predicates available in the host logic programming system, and in particular the cut.

The interface between the two layers consists of a handful of entry-point predicates, each of which is declared by a directive similar to the following one:

```
:- support check_consistency/1.
```

Please note that this directive cannot be entered interactively: it must be included in the text of the upper layer part of your program.

The support declaration means that the metainterpreter should treat the declared predicate as a built-in, i.e., just let Prolog execute it.

The support layer cannot invoke the upper layer, so there is no need to declare those predicates in the support layer that are not directly invoked by the upper layer.

Predicates that are declared as "support" (and those that are—directly or indirectly—called by them) must be defined in other files. To compile and load such a file, use the following directive in the text of your program:

[2] Having a file wherein you specify the names of built-in predicates you actually want to use does have its advantages. Some logic programming systems (e.g., ECL^{i}PS^e) support a very extensive set of libraries that define built-in predicates whose names are treated as reserved even if you don't use the libraries. As a result, many names that you might reasonably want to use in your programs are not available to you.

```
:- load_support( filename ).
```

In this context, the default extension of the *filename* will be the default extension used by the host logic programming system for names of files that contain Prolog code.

2.3 Declaring "entry points"

Before execution begins, the interpreted program is subjected to a number of sanity checks. One of these is a check whether every defined predicate is actually called from somewhere (i.e., whether there is no dead code).

Since it is not unusual for a program to contain a handful of such predicates on purpose (they are intended as "entry points" that are to be invoked from a query), the user can declare them by using a directive similar to the following:

```
:- top p/1, q/2.
```

The declaration is given only to suppress warnings. However, it is an error for an undefined predicate or a support predicate to be so declared.

2.4 Declaring dynamic predicates

To declare a predicate whose clauses are asserted and/or retracted by the interpreted program, use

```
:-  dynamic p/k.
```

2.5 Hooks

The program may contain clauses that modify the definition of the interpreter's predicate *essence_hook/2* (the clauses will be asserted at the front of the predicate, and will thus override the default definition for some cases). The interpreter's default definition is

```
essence_hook( T, T ).
```

This predicate is invoked, in certain contexts, when:

– two terms are about to be compared (either for equality or to check whether they are variants of each other);

– an answer is tabled;

– an answer is retrieved from the table.

The primary intended use is to allow suppression of arguments that carry only administrative information and that may differ in two terms that are considered to be "semantically" equal or variants of each other.

For example, the presence of

```
essence_hook( p( A, B, _ ),  p( A, B ) ).
```

will result in `p( a, b, c )` and `p( a, b, d )` being treated as identical: each of them will be translated to `p( a, b )` before comparison.

> **Warning:** *This facility should be used with the utmost caution, as it may drastically affect the semantics of the interpreted program in a fashion that could be hard to understand for someone who is not familiar with the details of the interpreter.*

Chapter 3

Running a program

3.1 Loading the interpreter

The interpreter is written in Prolog. It is distributed in source form.

The interpreter is known to run on ECL^{i}PS^e 6.0, SICStus 4.0 and SWI Prolog 5 7. If you plan to run programs that take advantage of coinductive programming, you might prefer to avoid ECL^{i}PS^e, which has somewhat inadequate support for cyclic terms.

The simplest way to proceed is to:

1. start your logic programming system;
2. If you are using ECL^{i}PS^e or SICStus, type in the following directive:
   `:- [ 'Path/tabling/dra' ].`
   where *Path* is the path to the root of the distribution tree. If you are using SWI Prolog, the directive is:[1]
   `:- [ 'Path/tabling/drap' ].`

This will just load the interpreter, but you will still be interacting with the host logic programming system. Sec. 3.2 describes how to start the

[1] You can use also
`:- [ 'Path/tabling/drapf' ].`
if the interpreted program does not use cyclic terms (i.e., in particular if there are no coinductive predicates). This version could be significantly faster, even faster than SICStus.

interpreter.

The interpreter is encapsulated in its own module, called *dra* (or *drap* in the case of SWI Prolog). So if you are running ECL^{i}PS^e, you will probably find it more convenient to import the module by writing

```
:- import dra.
```

immediately after loading the interpreter.

It may well be that things have been installed differently on your site. This might be because the interpreter has been modified to run with a different Prolog system, or because an immediately-loadable version has been made available in some standard directory. The person responsible for the local installation of the interpreter will provide you with more details.

3.2 Loading a program

Once you have loaded the interpreter into your logic programing system, you may want to load and run a program in the interpreter. This is done by writing

`prog(` *`filename`* `).`[2]

filename should be the name of the file that contains your program. If the name is given with no extension, it will be automatically extended with `.tlp`. If the name should have a different extension, you must type in the entire name, enclosed in single quotes, e.g.,

```
prog( 'myfile.pl' ).
```

Quotes must also be used if the file is not in the current directory and you are providing an absolute or relative path.

As the file is being read and loaded, directives and queries are interpreted on-the-fly. Each query is evaluated to give all solutions (i.e., as if the user kept responding with a semicolon): to avoid that you can use the built-in predicate *once/1* in the queries.

You should be aware that loading a program obliterates all traces of previously loaded programs, including the contents of the answer table. If you are

[2] If you are running in ECL^{i}PS^e, and have not imported the module *dra* (as explained in Sec. 3.1), you must write `dra:prog` instead of `prog`.

interested in re-running your program from scratch (so that it does not take advantage of answers that were already tabled), you can just load it again.

3.3 Interacting with a loaded program

3.3.1 The interactive mode

After the file is loaded (and all the directives and queries it contains are executed), the interpreter enters interactive mode. This is very much like the usual top-level loop, except that it is the interpreter—and not the underlying logic programming system—that evaluates queries and executes directives.

In the interactive mode the interpreter will read your input and act on it. Input consists of a term, terminated by a fullstop (i.e., the period character) and immediately followed by a newline (i.e., you must press the `ENTER` key).[3] If a query succeeds, you will get a printout that looks like this:

```
Yes  (more?)
```

You should then type in a semicolon immediately followed by a newline (if you want more answers), or just a newline (if you don't).

When you type in a term of the form "`:- ...`", it will be treated as a directive; when you type in a term of the form "`?- ...`", it will be treated as a query; when you type in a term that does not begin with `:-` or `?-`, it will also be treated as a query.

The difference between directives and queries is quite crucial, because the names of the directives do not occupy the same name space as the names of predicates. If you type in, say,

```
answers( _, _ ).
```

this will have nothing to do with the directive

```
:- answers( _, _ ).
```

and the interpreter will try to invoke the predicate *answers/2* in your pro-

[3] You cannot input more than one term per line. On SICStus all characters between the fullstop and the newline will be ignored. On ECL^{i}PS^e, if the answer to your query is "`Yes  (more?) `", the remainder of the previous line will be treated as your input and the interpreter will seem to cease responding. To get out of this state type in a fullstop followed by a newline.

gram. This may be a little confusing, but the good news is that you don't have to worry about potential conflicts between the names in your program and the names of the interpreter's directives.

Neither do you have to worry about conflicts between your program and the interpreter itself. The interpreted program is loaded into a separate module called `interpreted`. If there is a support layer, it is loaded into the module `support`. I mention these names, because the host system may show them in error messages if something goes horribly wrong.

3.3.2 Resuming the interactive mode

To just enter interactive mode (without loading a new program) invoke[4]

```
top.
```

The interpreter does not allow you to input clauses directly from your terminal, but it's good to have recourse to this call if you have exited interactive mode (see below) or if the execution of the interpreter was interrupted (either because of a fatal error, or because you pressed Ctrl-C on your keyboard). The program that was most recently loaded is still there, the answer table might have been populated, so you might want to resume interactive mode.

3.3.3 Exiting the interactive mode

To exit the interactive mode enter the end of file character (*Ctrl-D*),[5] or just write

```
quit.
```

3.3.4 Statistics

Just before the result of a query is reported, the interpreter produces a printout with statistics accumulated since the previous such printout (or since the beginning, if this is the first printout during the current session

[4] Again, `dra:top` in ECL^{i}PS^e, if you have not imported *dra*.
[5] *Ctrl-D* appears not to work with tkeclipse.

with the interpreted program). The printout looks like this:

```
[K steps, M new answers tabled (N in all)]
```

K,M and N are natural numbers. K is the number of evaluated goals, M is the number of new additions to the answer table, and N is the current size of the answer table.

Please note that you might sometimes see new answers tabled in 0 steps: this may happen when you ask for more results (by typing a semicolon) and the last goal to be activated has still not completed its task. You might also see that new answers were added even though the final response is `No`: this only means that some auxiliary goals were successful, while the main one was not.

3.3.5 Print depth

When a query succeeds, the instantiations of its variables should be printed upto a certain maximum depth. The default value in the distributed version of the interpreter is 10. The maximum depth can be changed from the interpreted program (or interactively from the top-level) by invoking

```
set_print_depth( N )
```

where N is a positive integer.

Please note that with some Prolog implementations this might not prevent a loop if the printed term is cyclic (as will often happen for coinductive programs).

Note also that the foregoing does not apply to invocations of built-in predicates in the interpreted program. It is up to the user to apply the built-in that is appropriate for the host logic programming system. For example, in the case of SICStus, use `write_term( T, [ max_depth( 10 ) ] )`, rather than just `write( T )`, if you expect the instantiation of `T` to be cyclic.

3.4 Including other files

To include files (interactively or from other files) you can use the usual Prolog syntax:

```
:- [ filename1, filename2, ... ].
```

The default extension is `.tlp`.

Please note that including a file with `:- [ filename ].` and loading a program with `prog( filename ).` are very different actions. When the interpreter includes a file, the contents are just added to its memory. When it loads a program, it first (re)initializes itself, wiping out the previously loaded program, all included files and the answer table.

3.5 Inspecting the answer table

In principle, the answer table is an auxiliary data structure that is, in effect, accessed by normal queries.

However, the interpreter gives you the possibility of looking "under the hood" by accessing the table directly. This might be useful for assessing the efficacy of your tabling declarations, or simply for satisfying your curiosity.

To print out subsets of the current answer table, use

```
:- answers( Goal, Pattern ).
```

where *Goal* and *Pattern* are terms. This will print all those tabled answers that are associated with a variant of the goal and unifiable with the pattern. If the first argument is a variable, the pattern will be used as a filter for all the answers in the table.

To produce a dump of the entire table, just use

```
:- answers( _, _ ).
```

3.6 The "wallpaper" trace

The interpreter does not incorporate an interactive debugger, but it can produce a long trace of what happens during the execution of an interpreted program. This facility is useful mainly for helping to diagnose problems with the interpreter: some of the information in the trace will not be easy to understand for someone who does not know the details of the DRA method [1], and I will not try to explain it all here. Still, you might sometimes be able to get some useful information from the trace, e.g, about how new answers are added to the table.

To produce a wallpaper trace of what happens to some chosen predicates, use a directive similar to the following:

```
:- trace p/3, q/0, r/1.
```

If you want to trace all predicates, use

```
:- trace all.
```

These directives are cumulative.

Summary of directives

An argument specified as *PredSpec* can take three forms:

(a) A predicate specification written as *name/arity*: for example `foo/3` (in the short descriptions below we will assume this is the form that is used);

(b) A sequence of such specifications, separated by commas: for example `p/2, q/1, r/3`;

(c) The word `all`, which specifies all predicates. (This cannot be used for `support` and `dynamic`!)

If the same kind of directive occurs a number of times, specifying different predicates, the results are cumulative. In particular, `all` subsumes all other predicate specifications.

Directive:	*Short description:*
`:- [` *filename* `].`	load a part of the program (p. 15)
`:- answers(` *Goal*`,` *Pattern* `).`	inspect the answer table (p. 16)
`:- coinductive` *PredSpec*`.`	predicate is coinductive (old style) (p. 5)
`:- coinductive1` *PredSpec*`.`	predicate is coinductive (new style) (p. 6)
`:- dynamic` *PredSpec*`.`	predicate is dynamic (p. 9)
`:- load_support(` *filename* `).`	load (a part of) the support layer (p. 9)
`:- old_first.`	change the order in which results are produced (p. 3)
`:- support` *PredSpec*	predicate is an entry point to the support layer (p. 8)
`:- tabled` *PredSpec*`.`	predicate is tabled (p. 4)
`:- top` *PredSpec*	predicate is an entry point (p. 9)
`:- trace` *PredSpec*`.`	trace the predicate (p. 17)

Bibliography

[1] Hai-Feng Guo and Gopal Gupta. Tabled logic programming with dynamic ordering of alternatives. In Philippe Codognet, editor, *Logic Programming, 17th International Conference, ICLP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*, volume 2237 of *Lecture Notes in Computer Science*. Springer, 2001.

[2] Richard Min and Gopal Gupta. There is often no need to use clauses if coinductive hypotheses match the goal. Personal communication, 7 May 2009.

[3] Luke Simon, Ajay Bansal, Ajay Mallya, and Gopal Gupta. Co-logic programming. In Lars Arge, Christian Cachin, Tomasz Jurdziński, and Andrzej Tarlecki, editors, *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wrocław, Poland, July 9-13, 2007, Proceedings*, volume 4596 of *Lecture Notes in Computer Science*. Springer, 2007.

[4] Luke Simon, Ajay Mallya, Ajay Bansal, and Gopal Gupta. Coinductive logic programming. In Sandro Etalle and Mirosław Truszczyński, editors, *Logic Programming, 22nd International Conference, ICLP 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4079 of *Lecture Notes in Computer Science*. Springer, 2006.

Index