Spider vs. Prolog: Computation Control

Emilia Golemanova, Kostadin Kratchanov, Tzanko Golemanov

Abstract: This is the second in a series of two reports presenting a comparison between the Control Network Programming language Spider and the logic programming language Prolog. Here, the focus is on juxtaposing the tools for programmer's control of the computation (inference) offered by the two languages.

Key words: Control network programming, CNP, Spider, logic programming, Prolog, programming languages, programming paradigms, AI programming, nondeterministic programming, declarative programming, computation control, inference control

1. INTRODUCTION

This report continues the comparison of the Control Network Programming (CNP) language Spider and the logic programming language Prolog started in [6]. The focus here is on the comparative investigation of the means for programmer's control of the inference process supported in these languages. The need, advantages and drawbacks of introducing such a procedural semantics in the generally descriptive languages were discussed in [6, Sec. 4]. The terminology and exposition follow [6].

Prolog's rather controversial search control features include CUT, fail, true, not, once, repeat and the IF-THEN-ELSE structure [1, para. 7.5; 3, Ch. 1, 4]. Spider offers a powerful collection of control tools – system state STOP, control states (ORDER, SELECT, RANGE) and system options (SOLUTIONS, BACKTRACKING, ONEVISIT, LOOPS, RECURSION, ARROWCOST, MAXPATHCOST, NUMBEROFARROWS, PROXIMITY) [2; 3]. Some of these are used for static search control while others can be employed for implementation of “automatic” heuristic algorithms embodied within the CN interpreter.

The next sections contain a discussion of the control features in Prolog and how similar results can be achieved in Spider.

2. SIMULATING THE PROLOG'S CUT

The Prolog's CUT feature is by far the most well-known and widely used control tool in Prolog. In the following sections we discuss how it can be partially simulated in Spider, and how the major applications of CUT can be dealt with in Spider. We argue that Spider is adequately equipped with related features, and in some cases it is actually more efficient than Prolog in similar situations. We also discuss how Prolog influenced us to realize a better version of our system option BACKTRACKING.

2.1 The re-designed Spider system option BACKTRACKING

The system option BACKTRACKING implemented until recently in Spider is specified in [9, para. 6] where examples of its usage can be also found. In the process of the research reported here (a comparative study of Spider vs. Prolog), it was realized that a modified version of the interpretation of this option would not only allow us to more closely model the behaviour of CUT in Prolog but, more importantly, will be more natural and will have certain advantages. This new interpretation of BACKTRACKING was implemented in the latest version of Spider. This modified system option will be described below and will be used in the rest of the paper. As a side effect of this research, we were able to project ideas and solutions from Prolog into the domain of CNP.

The scope of the system option BACKTRACKING can be the entire CN, a subnet, or a state. It is used when the programmer wants to “switch off” backtracking. In the older version, when the value of the system option is NO, if all arrows outgoing from a state have been attempted unsuccessfully, the control does not move back through the arrow along which the state was entered – instead, the computation halts unsuccessfully. In the new implementation, the very concept for BACKTRACKING=NO has been changed. Now,
in the case of a dead end the control moves backwards but only along the existing path. In other words, backtracking is switched off in the sense that no more alternatives are attempted; however the control moves back restoring the previous status.

Let us consider a simple example (Figure 1). Assume that in a given state A the value of the option BACKTRACKING is NO. Assume three arrows emanate from state A. Assume the execution [6] of the first arrow was unsuccessful. The second arrow is now tried, and assume this attempt is successful and the control moves to state B. Assume the attempts to continue from B failed, and therefore the control moves back to A after a backward execution of the second arrow. The effect of BACKTRACKING = NO in state A is that the third arrow is cut off – it is not attempted.

To define the effect of BACKTRACKING = NO: in a given state, after the first successful arrow, all remaining arrows are pruned.

Note that if the first successful arrow (such as the second arrow in our example) contains subnet calls, during the backward movement (from B back to A) new paths in the called subnets will be sought, as usual. We will see soon that because of this particular aspect the simulation of CUT in Spider will differ from the behaviour of Prolog.

Let us conclude the discussion of the new BACKTRACKING system option with a final note. If a subnet, S1 with backtracking on calls a subnet, S2 where backtracking is off, and if S2 is not successful then the control will return to S1 where the search for new paths will continue. (In the old version, the computation will stop in S2 with HALT.)

2.2 Partial simulation in Spider of the CUT behaviour

By definition [4], the CUT goal succeeds whenever it is the current goal; in addition the derivation tree is trimmed of all other choices on the way back to and including the point in the derivation tree where the cut was introduced. The node where the CUT was introduces is referred to as the “parent” node [1, para. 5.1].

In a Spider CN simulating a Prolog program [6, para. 5], a CUT in the Prolog program can be modelled as follows: CUT is replaced by an auxiliary state (let us name it CUT), and the subnet (or its initial state only) is given option value [BACKTRACKING=NO]. This subnet represents the predicate - parent goal.

To illustrate the rule let us model the following fragment of a Prolog program from [1, para. 5.1]

\[
\begin{align*}
C &: \neg P, Q, R, !, S, T, U. \\
C &: \neg V.
\end{align*}
\]

where P, Q, ..., V denote given predicates. The corresponding fragment of the Spider CN is shown in Figure 2.

The only difference between the the behaviour of CUT in the original Prolog program and the Spider model is that in Spider new alternatives will be sought within subnets P, Q, and R which is not the case in Prolog. Therefore, we say that in Spider the simulation of CUT is partial. To achieve complete simulation we will need to introduce a special type of system state or primitive CUT (in the simulation above we used an ordinary state called CUT). This is in principle possible but
has not been done. We argue below that the control tools currently offered in Spider are enough and a Spider programmer can easily and conveniently deal with most situations where user’s control is necessary.

2.3 The Spider simulation actually corresponds to a better-style Prolog program

Let us consider the Prolog program shown in Figure 3. It is almost identical to the program of [6, Figure 1] used there as the main example illustrating how core Prolog is modelled in Spider. The only difference is that the program of Figure 3 has a CUT in the first clause.

Without the CUT, the Spider model was shown in [6, Figure 2]. With the CUT, the only difference will be in subset lucky. The new version of this subset is given in Figure 4.

The Spider simulation will behave differently from the Prolog program of Figure 3, because in Spider generous can be “re-satisfied”. The Spider version will produce the same results as the following modification of the Prolog predicate lucky:

lucky(Y):-generous(X), likes (X,Y).
lucky(Y):-not generous(X), content(Y).

Note that the latter Prolog version corresponding to the Spider model is more readable than the previous (but not entirely equivalent) version shown in Figure 3. In addition, the Spider variant is more efficient as it avoids the repeated satisfying of generous in first rule and then, after backtracking, in the second rule of lucky.

3. SPIDER MODELLING OF THE COMMON USES OF THE CUT

As emphasized in Section 2.2 above, the simulation of the CUT in Spider is only partial. However, Spider is perfectly equipped to allow convenient handling of the typical situations when the CUT is usually used.

The following main common uses of the CUT are identified in [2, para. 4.3]:

- Confirming the choice of a rule: “if you get this far, you have picked the correct rule for this goal”.
- The CUT-FAIL combination: “if you get to here, you should stop trying to satisfy this goal.”
- Terminating generate-and-test: “if you get to here, you have found the only solution to this problem, and there is no point in ever looking for alternatives.”

We discuss the Spider solutions for these situations in the sections that follow.

3.1 Confirming the choice of a rule

It describes a situation where mutually exclusive clauses are present. A good example illustrating this situation is described in [1, para. 5.1]. It relates to the so called double-step function specified by the below pseudo-code:

```prolog
lucky(Y):-generous(X), likes (X,Y).
lucky(Y):-not generous(X), content(Y).
```

Figure 3 Main example with CUT

Sub lucky(var Y:string);
var X:string;
```prolog
[cut|X(0,0)|] generous(X,Y)
```
if \( X<3 \) then \( Y = 0 \)
else if \( X<6 \) then \( Y = 2 \)
else \( Y = 4 \)

The function can be modelled by the following three Prolog programs of which the first one is less efficient. Only the first two are considered in [1]. Unlike the second program, the third one will also correctly work for queries such as \(?- f(2,4)\).

\[
\begin{align*}
  f(X,0) & : - X<3. \\
  f(X,2) & : - 3=<X,X<6. \\
  f(X,4) & : - 6=<X. \\
\end{align*}
\]

\[
\begin{align*}
  f(X,0) & : - X<3, !. \\
  f(X,2) & : - X<6, !. \\
  f(X,4) & : - Y=4. \\
\end{align*}
\]

\[
\begin{align*}
  f(X,Y) & : - X<3, !, Y=0. \\
  f(X,Y) & : - X<6, !, Y=2. \\
\end{align*}
\]

The usual Spider simulation with a CUT state described in Section 2.2 above will yield the results expected in Prolog. A Spider model of the third improved program is illustrated in Figure 5. Deterministic predicates such as those for comparison like \(< \) and \(>\) can be modelled by primitives rather than using subnets.

As suggested in [2, para. 4.3], it is a good programming style to replace cuts by the use of \(^*\) (not). This is because programs containing cuts are in general harder to read. For example the first fragment should be replaced by the second, as shown below:

\[
\begin{align*}
  A & : - B, !, C. \\
  A & : - C. \\
\end{align*}
\]

The second program, although more readable, is less efficient because if it ever backtracks and considers the second rule, it will have to try to satisfy \( B \) again to see if not \( B \) can be satisfied. In addition, the two programs will be actually equivalent only for "deterministic" predicate \( B \). In Spider, the two versions will be always equivalent, and no duplication will be present.

We can draw the conclusion that the usage of BACKTRACKING=NO in Spider similarly to the example of Figure 2, is an efficient and convenient way to handle mutually exclusive alternatives.

3.2 The CUT-FAIL combination

A well-known application of the CUT facility is the so called CUT-FAIL combination [1, para. 5.3; 2, para. 4.3.2]. It is equivalent to a not predicate:

\[
\begin{align*}
  \text{not}(P) & : - P, !, \text{fail} ; \\
  \text{true}. \\
\end{align*}
\]

This predicate will be simulated in Spider by the subnet shown in Figure 6.

The following example (in two versions) is from [1, para. 5.3]:

\[
\begin{align*}
  \text{likes}(\text{mary}, X) : - \\
  & \quad \text{animal}(X), \\
  & \quad \text{not snake}(X). \\
\end{align*}
\]

\[
\begin{align*}
  \text{likes}(\text{mary}, X) : - \\
  & \quad \text{not snake}(X), \\
  & \quad \text{animal}(X). \\
\end{align*}
\]

The corresponding two versions of Spider subnets are shown in Figure 7.
3.3 Terminating a “generate and test”

This application of the CUT cannot be directly modelled in Spider (unless a special system state CUT is introduced). Alternative approaches must be used.

4. DANGER OF INFINITE LOOPING

The danger of indefinite looping is well recognized in Prolog [1, para. 2.6.1; 3, para. 1.14]. Similar effects are possible in CNP. All of the system options BACKTRACKING, ONEVISIT, LOOPS, and RECURSION can be used in different situations to fight looping [9]. However, a systematic investigation of this topic has not been presented. In relation to looping, In Prolog, significant factors are the order of the clauses in the program, and the order of goals in the bodies of clauses [1, para. 2.6]. In Spider, significant factors are the order of the arrows emanating from a given state, and the order of the primitives on an arrow.

Using the general approach to simulate a Prolog program in Spider presented in [6, Sec. 5], one obtains Spider CNs that behave analogously to the original Prolog programs. For example, we have performed experiments with Spider models of the four variations of the predecessor predicate given in [1, para. 2.6.2], and the results obtained fully correspond to the Prolog ones described.

Fortunately, unlike Prolog, Spider offers a very easy tool for avoiding (more precisely, stopping) an infinite loop in such a case. This can be done by simply defining [RECURSION=n] for some specific value n>1. This will result in depth-limited search.

5. CONCLUSIONS

This report is a detailed comparative study of the CUT feature as the major tool for user’s procedural control in Prolog, with the methods for its modelling in Spider. It has been shown that the effect of CUT can be partially simulated in Spider, and most of the typical usages of CUT can be easily handled.

Some of the major points in the comparison of the computation control features of Spider and Prolog are summarized below:

- Spider offers a much more developed collection of control tools that can be used by an advanced CNP programmer.
- The levels of recursion can be restricted.
- The system options can have various ranges – the whole CN, a subnet, a state, or an arrow. In particular, backtracking can be switched off at various levels.
- Spider supports means for re-ordering of the arrows and for selection of an arrow or a group of arrows.
- Spider, unlike Prolog, has built-in tools for implementing heuristics [11]. Heuristics can be “automatic” – no external algorithm for implementing such heuristics is needed.

A more general comparison Spider vs. Prolog can be found in [6].
6. FUTURE RESEARCH

As shown in Sections 2.2 and 3.3 above, the effect of CUT cannot be currently completely simulated in Spider (as far as committing to a single path is concerned), and as a result the third common usage of CUT cannot be directly handled. Related features are the once predicate suggested in [3], and the “one-solution commit” operator [12]. Spider possesses a very powerful collection of control tools; however, it would be still worthy developing a system state or primitive to remedy the mentioned shortcoming.

Another avenue we intend to explore in the near future is conducting a similar comparative study of the Gõdel programming language [8], and first of all its pruning (the commit) operator. A very interesting feature of Gõdel is also the more intelligent computation rule which, in principle, may select (in a conjunction of literals) a literal that will hopefully yield high efficiency and avoidance of loops.

Another (very important and practical) aspect of future reports will be the comparison of solutions to typical representative problems written in Spider and Prolog, respectively.

An efficiency analysis should be also conducted, in two aspects – efficiency of a system’s interpreter, and efficiency of a problem’s solution.

REFERENCES

ABOUT THE AUTHORS
Emilia Golemanova (EGolemanova@ecs.ru.acad.bg) and Tzanko Golemanov (TGoelmanov@ecs.ru.acad.bg) are principal assistant professors in the Department of Computer Systems, University of Rousse, Bulgaria. Dr. Kostadin Kratchanov (kostadin.kratchanov@yasar.edu.tr) is associate professor in the Department of Computer Engineering, Yasar University, Turkey.
PROCEEDINGS
of the
International Conference
on Computer Systems and Technologies
and Workshop for PhD Students in Computing

CompSysTech’09

Ruse, Bulgaria, 18-19 June
PROCEEDINGS
of the
International Conference
on Computer Systems and Technologies
and Workshop for PhD Students in Computing

Edited by
B. Rachev
Technical University of Varna, Bulgaria
A. Smrikarov
University of Ruse, Bulgaria

CompSysTech’09

Ruse, Bulgaria, 18-19 June
Proceedings of the Bulgarian International Conference on Computer Systems and Technologies (CompSysTech'09) and PhD Workshop in Computing, 18-19 June 2008, held at the University of Ruse “Angel Kantchev” in Ruse, Bulgaria. This Conference is an International event in Computer Science and Engineering in Europe.

Papers included in these Proceedings were triple reviewed by independent referees.

ISSN 1313-9037

Copyright © 2009 by the Bulgarian Chapter of ACM–acmbul&UAI. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of the publisher.

Printed in Bulgaria