Using the Prolog Graphical Tracer

Chris Mellish,
Division of Informatics,
University of Edinburgh

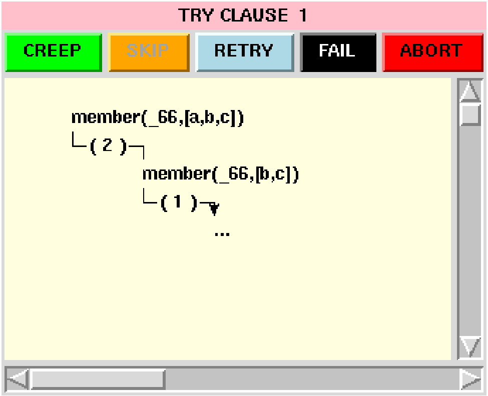
November 20, 2000

Abstract

This document describes the graphical tracer used in AI1 Prolog - what it does, how it models the flow of control in Prolog and how to use it. Note that there are pictures in this document which do not come out in the web version.

1 What the Tracer is

Normally when you present a goal to Prolog you get little information about what is going on whilst it is being processed. This is all right when the program is working as expected or when you are a Prolog expert, but it does not help you to develop an intuition for the flow of control in Prolog execution. Sicstus Prolog provides some good tracing facilities, but these only show a local snapshot of what is going on. These are meant primarily to help experts debug complex programs, rather than for learners to use on simple programs. The graphical tracer is an attempt to show what is going on using a simple graphical display which is updated as the program runs. The following shows an example of the display:



At the top of the window is a message (here “TRY CLAUSE 1”) summarising the current state of the program and what it is doing. At the bottom is a large area including a picture representing the complete state of the program. In the middle are four coloured buttons. The tracer runs by “single stepping” through the execution, i.e. redisplaying the program state after small steps. You click on the buttons (usually the green one marked “CREEP”) to make the program progress from one step to the next.

2 Displaying the Execution State

This section explains the model of Prolog execution which is reflected in the display produced by the tracer. This corresponds closely to the model described in Clocksin and Mellish sections 2.6 and 8.3.

The basic convention of the display is that there is a line, more or less continuously extending from the top to an intermediate point at which there is an arrow. The items above the arrow represent goals which are currently in the middle of being attempted and goals which have already been satisfied. The items below the arrow represent goals which have not yet been attempted, but which need to be satisfied in order for the current attempt to succeed. If the arrow points downwards, then the system is about to move on to new goals, whereas if it points upwards then it is about to backtrack to previous choices because a failure has occurred. The indentation of the goals indicates where they came from - when a goal is matched against a clause, the introduced subgoals are indented one position to the right from the original goal. Numbers written between parentheses indicate a clause that has been chosen to satisfy the goal appearing immediately above. The clauses for a predicate are numbered from 1 upwards, in the order in which they are written in the program. Goals are displayed as if by the Prolog `write` predicate. This means that uninstantiated variables appear as numbers preceded by underscores. If two such numbers are the same, then this indicates that the two variables currently share (when one becomes instantiated to a value, the other one will follow).

For example, in the example above, the original query was something like:

```
?- member(X,[a,b,c]).
```

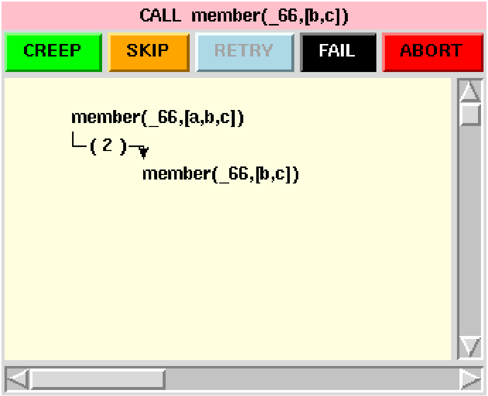
The original query is shown in the top line of the main graphic. The second clause for `member` was chosen and this introduced a subgoal (indented one to the right) of the form `member(X,[b,c])`. The first arguments in the two goals share. The first clause is being tried for the subgoal. If it matches, then new subsubgoals will appear below the arrow. Clicking on the “CREEP” button will result in the next program state being shown.

The program pauses at the four “ports” corresponding to the “box model” of Prolog execution. These announce events of CALL, EXIT, REDO and FAIL which may occur with respect to a goal. The current event is displayed in the message above the buttons. An extra event, corresponding to the trying of a clause, is also shown. This is not part of the “box model”, but it is useful to see the program pause at these points.

These five "ports" will now be described in more detail, with an example of the display for each.

2.1 CALL

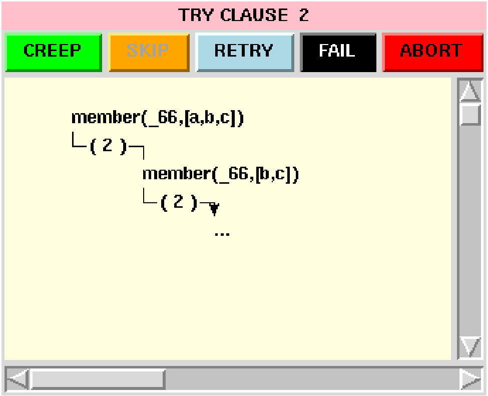
A CALL event occurs when the system moves on to the next unattempted goal. The arrow is shown moving downwards, just above the goal that is about to be attempted:



In the example, the system is just about to attempt the subgoal for `member(X,[b,c])`. If the predicate has clauses, it will next move on to try them in turn. Otherwise, it will move to FAIL.

2.2 Trying a Clause

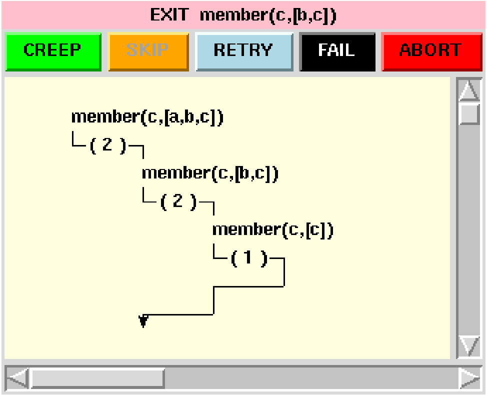
The clauses for a predicate are tried in turn. The tracer pauses as each one is considered.



In the example, the second clause is being considered. If the head of the clause actually matches the goal, then next any effects of the matching on variables will be shown and the subgoals introduced by the clause will be displayed. If there are no subgoals, then the system will move to EXIT the goal; otherwise it will move to CALL the first subgoal. If the head of the clause does not match, the system will move to try the next clause. If it runs out of clauses, it will move to FAIL the goal.

2.3 EXIT

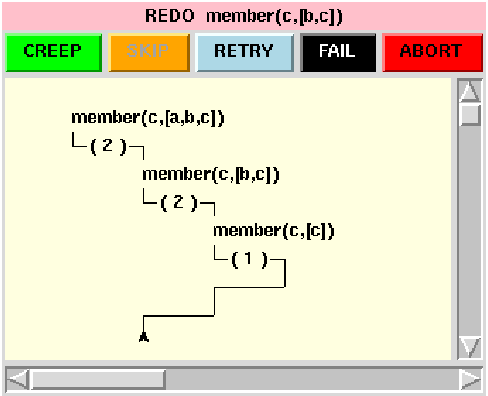
When a goal has matched the head of a clause and all the subgoals have been satisfied, the goal EXITs. The line moves out to the level of indentation corresponding to the goal and starts moving downwards (towards the next goal, if there is one).



In the example, the goal `member(X,[b,c])` has succeeded with `X=c`. Next the system will move to CALL the next outstanding goal if there is one. If not (as in this example), it will indicate an EXIT of the goal that invoked this one (here, `member(X,[a,b,c])`).

2.4 REDO

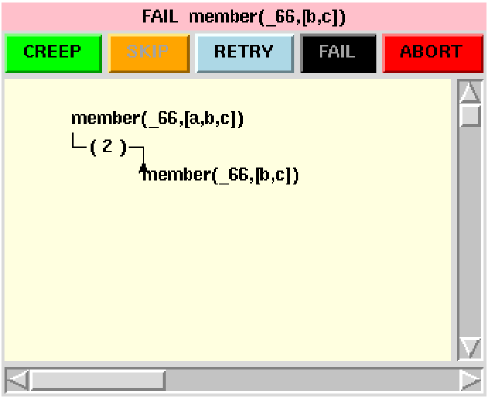
A REDO occurs for a goal when it has already succeeded once but some later goal then failed. Backtracking now takes place, to see if there is an alternative way of satisfying this goal.



In the example, a failure has occurred through the user asking for an alternative solution for the original query. The arrow is now pointing upwards. If there are subgoals, the system will next move to REDO them in turn. You can think of the line retreating upwards along the path it has come. When it comes to a place indicating the choice of a clause, it will undo the effects of the clause choice (variable bindings and introduced subgoals) and move to try the subsequent clauses. If one matches, it will start to move forward again; otherwise it will have to retreat further.

2.5 FAIL

A FAIL occurs when a goal has tried all possibilities and none of them led to a successful solution. The system will continue backtracking.



In the example, all possibilities for the subgoal `member(X,[b,c])` have been tried. Backtracking must now see whether there are alternative clauses that could be used for the original goal.

3 How to Run the Tracer

To run the tracer, copy the file `.sicstusrc` into your home directory:

```
% cp ~ai1teach/prolog/.sicstusrc ~
```

Having this file will mean that the tracer code is always loaded when you run Prolog. If for some reason at a later point you want to run Prolog without having the tracer code present (and without some of the effects that it has – see below), then you can simply do:

```
% rm ~/.sicstusrc
```

to return to the standard setup.

To trace the execution of a goal, precede the normal query with a `?` sign, for instance:

```
?- ?member(X,[a,b,c]).
```

This will then create a window for the tracer (if you haven't already created one) and start running your query. You may need to adjust the position of the window on the screen so that you can see both it and the window that you are using to type into Prolog. You may also need to resize or scroll within the window in order to see everything.

Once your query is running, you make the execution progress by clicking on the buttons. Initially just use the CREEP button, to simulate what would happen in a normal Prolog execution, step by step. When the system comes to EXIT or FAIL the original goal (and you have clicked to move on), it displays any solutions in your original window and asks you whether you want other solutions, in the normal way. It only makes sense to click on the buttons whilst your query is actually being executed, not when you are being prompted for something from the keyboard.

4 More Complex Facilities

The other buttons allow you to alter the flow of control in ways similar to those permitted by the Sicstus tracing mechanism. They are not all applicable at all ports.

SKIP allows you to avoid seeing the details of the execution of a given goal and for the system to run until that goal either EXITs or FAILs.

RETRY allows you to go right back to the start of satisfying a goal and try it again (for instance, so that you can look at it in more detail).

FAIL causes the current goal immediately to FAIL.

ABORT causes the current execution to be aborted.

The tracer can display information in two different sizes, large and small. Small is the default, and is what you would use for normal screen use. Large would be appropriate if you wanted to project the display onto a larger screen, for instance for a lecture. The predicates `tcl_SMALL` and `tcl_LARGE` (neither of which takes arguments) can be used to adjust the size for future tracings.

5 Things to look out for

The execution of system predicates is shown as if you had chosen to "skip" them. The tracer will not work properly with programs using disjunction or other meta-predicates, or with programs that use multiple modules. Actually it will "skip" the execution of meta-predicates, not treating any embedded cuts properly.

If you click on the buttons more than is needed, the clicks may be stored up and the appropriate actions then invoked whenever the system next needs an action to be selected. This may produce strange behaviour. The best thing is to watch out for when control returns to the keyboard and then not click any more.

For more advanced Prolog users: The tracer code uses the Prolog term expansion capability to ensure that all predicates are dynamic. This works with DCGs, but will not work if you have your own `term_expansion` definitions. Having all predicates be dynamic means that normal execution will not be as fast as without and that certain operations (e.g. retracting clauses) that would normally produce errors don't.

6 Limitations and Possible Extensions

There are many aspects of Prolog execution that are not covered, e.g. meta-predicates, constraints, blocking. The display of variable names often ends up with long numbers, and this could be made more user-friendly. Spypoints and other control options based on Sicstus tracing could be introduced. Modules and multiple files could be handled properly.

Extra facilities that could be useful might be rerunning the last step and showing unification in detail. Colour could be used more productively and scrolling could be automatic. The graphical notation was originally designed with a more tty-oriented display in mind, and it could not be liberated from these constraints. There could be a more elegant way of dealing with different sizes.