8.3 The Tracing Model

There are various ways of looking at the method by which Prolog attempts to satisfy goals. We have introduced a model in terms of the "flow of satisfaction" through boxes representing goals. Here we present the model of Prolog execution used by a number of debugging aids, such as the trace facility. This model is largely due to our colleague Lawrence Byrd, and for this reason the model is known as the Byrd Box model. Although Prolog systems differ in the debugging aids provided (and the Prolog standard does not specify what they should be), the following description should roughly correspond to what happens with many Prolog systems.

When the trace facility is used, the Prolog system prints out information about the sequence of goals in order to show where the program has reached in its execution. However, in order to understand what is happening, it is important to understand when and why the goals are printed. In a conventional programming language, the key points of interest are the entries to and exits from functions. But Prolog permits non-deterministic programs to be written, and this introduces the complexities of backtracking. Not only are clauses entered and exited, but backtracking can suddenly reactivate them in order to generate an alternative solution. Furthermore, the cut goal "!" indicates which goals are committed to having only one solution. One of the major confusions that novice programmers face is what actually occurs when a goal fails and the system suddenly starts backtracking. We hope this has been adequately explained in the previous chapters. However, the previous chapters discussed not only control flow, but also how variables are instantiated, how goals match against clause heads in the database, and how subgoals are satisfied. The tracing model describes the execution of Prolog programs in terms of four kinds of *events* that occur:

CALL. A CALL event occurs when Prolog starts trying to satisfy a goal. In our diagrams, this is when an arrow enters a box from the top.

EXIT. An EXIT event occurs when some goal has just been satisfied. In our diagrams, this is when the arrow emerges from the bottom of a box.

REDO. A REDO event occurs when the system comes back to a goal, trying to resatisfy it. In our diagrams, this is when the arrow retreats back into a box from the bottom.

FAIL. A FAIL event occurs when a goal fails. In our diagrams, this is when the arrow retreats upwards out of a box.

The debugging aids tell us about when events of these four kinds occur in the execution of our programs. These events will take place for all of the various goals that Prolog considers during the execution. So that we can distinguish which events are happening to which goals, each goal is given a unique integer identifier, its *invocation* *number*. Below we shall show some goals together with their invocation numbers in square brackets.

Let us now take a look at an example. Consider the following definition of the predicate descendant:

```
descendant(X, Y) :- offspring(X, Y).
descendant(X, Z) :- offspring(X, Y), descendant(Y, Z).
```

This piece of program derives descendants of a person, provided that there are offspring facts in the database, such as

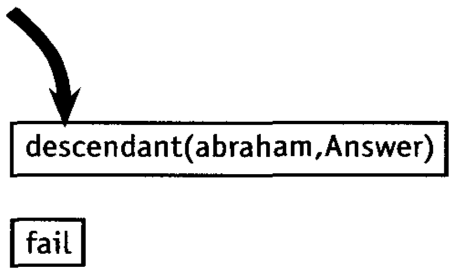
```
offspring(abraham, ishmael).
offspring(abraham, isaac).
offspring(isaac, esau).
```

The first clause of descendant states that Y is a descendant of X if Y is an offspring of X. The second clause states that Z is a descendant of X if Y is an offspring of X and if Z is a descendant of Y. We shall consider the question:

```
?- descendant(abraham, Answer), fail.
```

and we shall follow the control flow to see when the various kinds of events occur. It is important that you try to follow the trace we are about to look at by thinking about the flow of satisfaction entering and leaving the boxes for the goals. We will periodically display the current state in diagram form.

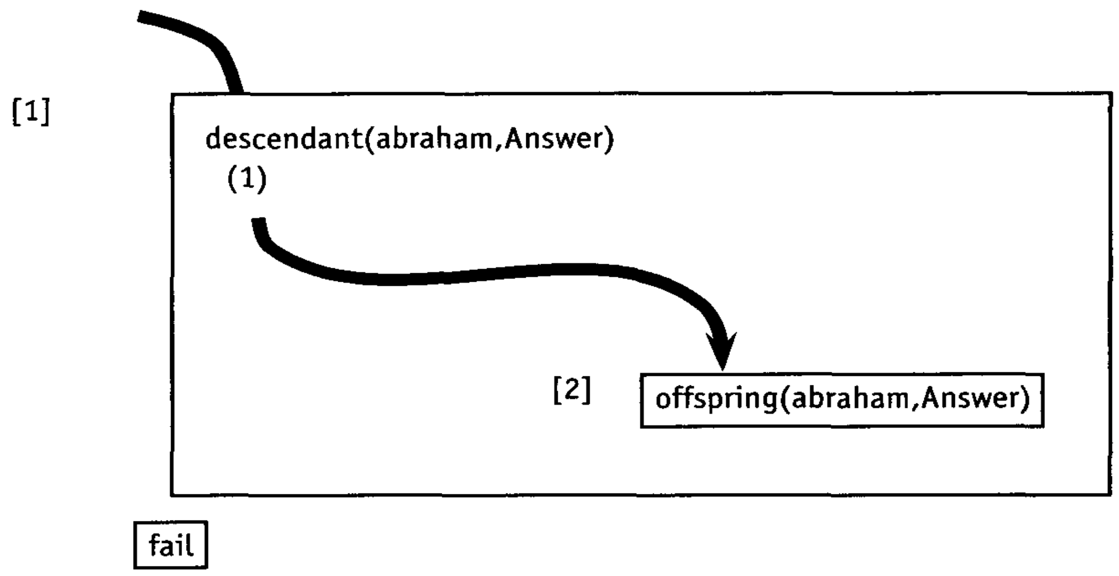
The first goal in the question is followed by a fail. The purpose of this is to force all possible backtracking behaviour out of the descendant goal. The question as a whole can therefore never succeed. However, the point of this trace is to observe the execution flow induced by the failure of the second goal (the fail). We begin with (as yet unentered) boxes for the two goals:



The first event is that the descendant goal is CALLed. This is invocation number 1 (shown in square brackets).

```
[1] CALL: descendant(abraham,Answer)
[2] CALL: offspring(abraham,Answer)
```

We have matched the first clause of the descendant procedure and this results in a CALL of a goal for offspring. The situation is now as follows, with the arrow moving downwards:



We continue:

```
[2] EXIT: offspring(abraham,ishmael)
```

Immediate success on the first clause, and so the goal EXITs.

```
[1] EXIT: descendant(abraham,ishmael)
```

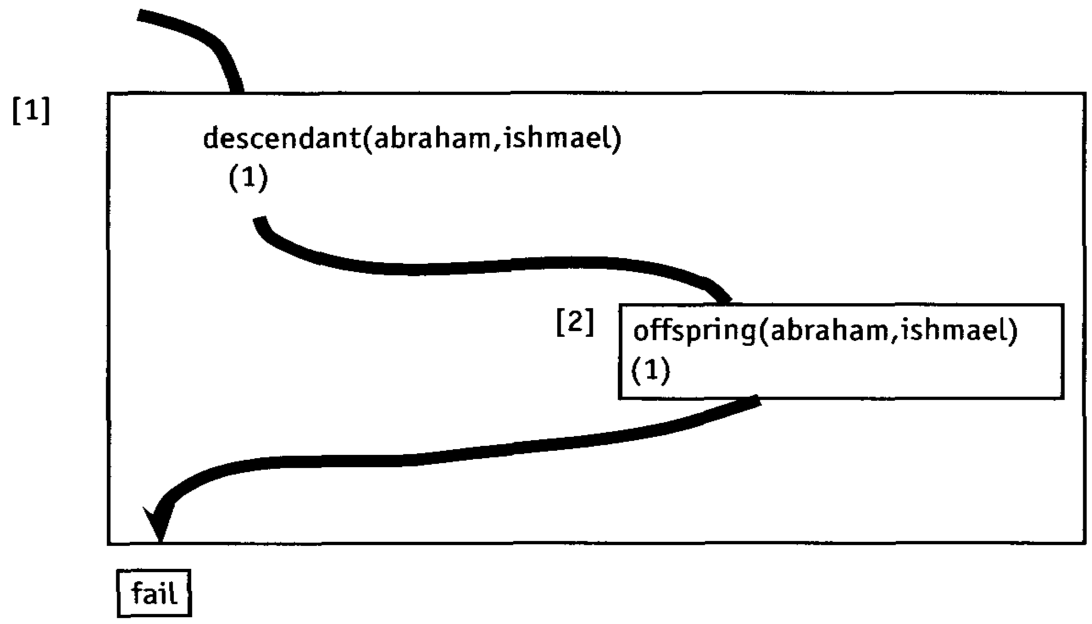
And thus we have satisfied the first descendant clause.

```
[3] CALL: fail
[3] FAIL: fail
[1] REDO: descendant(abraham,ishmael)
```

Then we try to satisfy fail, and, as might be expected, this goal FAILs. The arrow retreats back out of the fail box and back into the descendant box above. Here is a picture of where we are now. The arrow is retreating upwards out of the fail box.

Continuing:

```
[2] REDO: offspring(abraham,ishmael)
[2] EXIT: offspring(abraham,isaac)
```



An alternative clause is chosen for the offspring goal, and so the arrow can move down out of this box again.

```
[1] EXIT: descendant(abraham,isaac)
[4] CALL: fail
[4] FAIL: fail
[1] REDO: descendant(abraham,isaac)
```

Again, fail causes us to reject this solution and to start backtracking. Notice that this was a completely new invocation of fail (we entered it afresh from "above").

```
[2] REDO: offspring(abraham,isaac)
[2] FAIL: offspring(abraham,Answer)
```

This time, offspring cannot offer us another match and so we continue backtracking, the arrow retreating upwards out of the offspring box.

```
[5] CALL: offspring(abraham,Y)
```

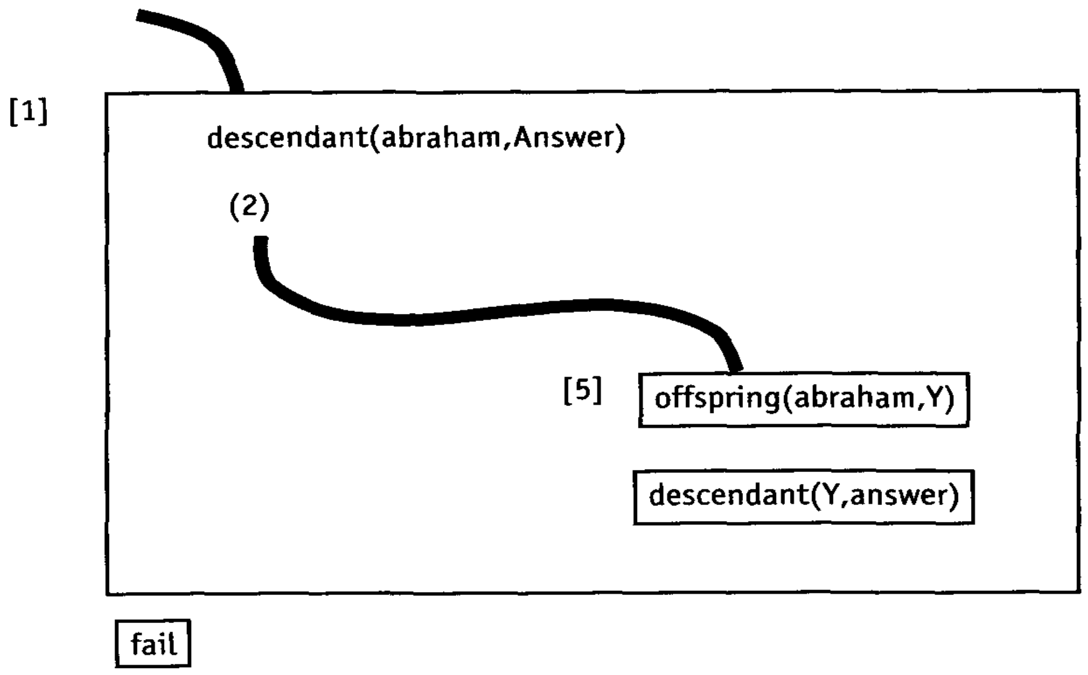
What has happened here is that Prolog has chosen the second descendant clause and this is a completely new offspring invocation corresponding to the first subgoal:

The arrow is now moving downwards again. Continuing:

```
[5] EXIT: offspring(abraham,ishmael)
[6] CALL: descendant(ishmael,Answer)
```

This provides a solution with which we now recursively call descendant. This gives us a new invocation of descendant.

```
[7] CALL: offspring(ishmael,Answer)
```



```
[7] FAIL: offspring(ishmael,Answer)
[8] CALL: offspring(ishmael,Y2)
[8] FAIL: offspring(ishmael,Y2)
[6] FAIL: descendant(ishmael,Answer)
```

Ishmael has no offspring (in this example), and so the offspring subgoals in both descendant clauses fail, thus failing the descendant goal.

```
[5] REDO: offspring(abraham,ishmael)
```

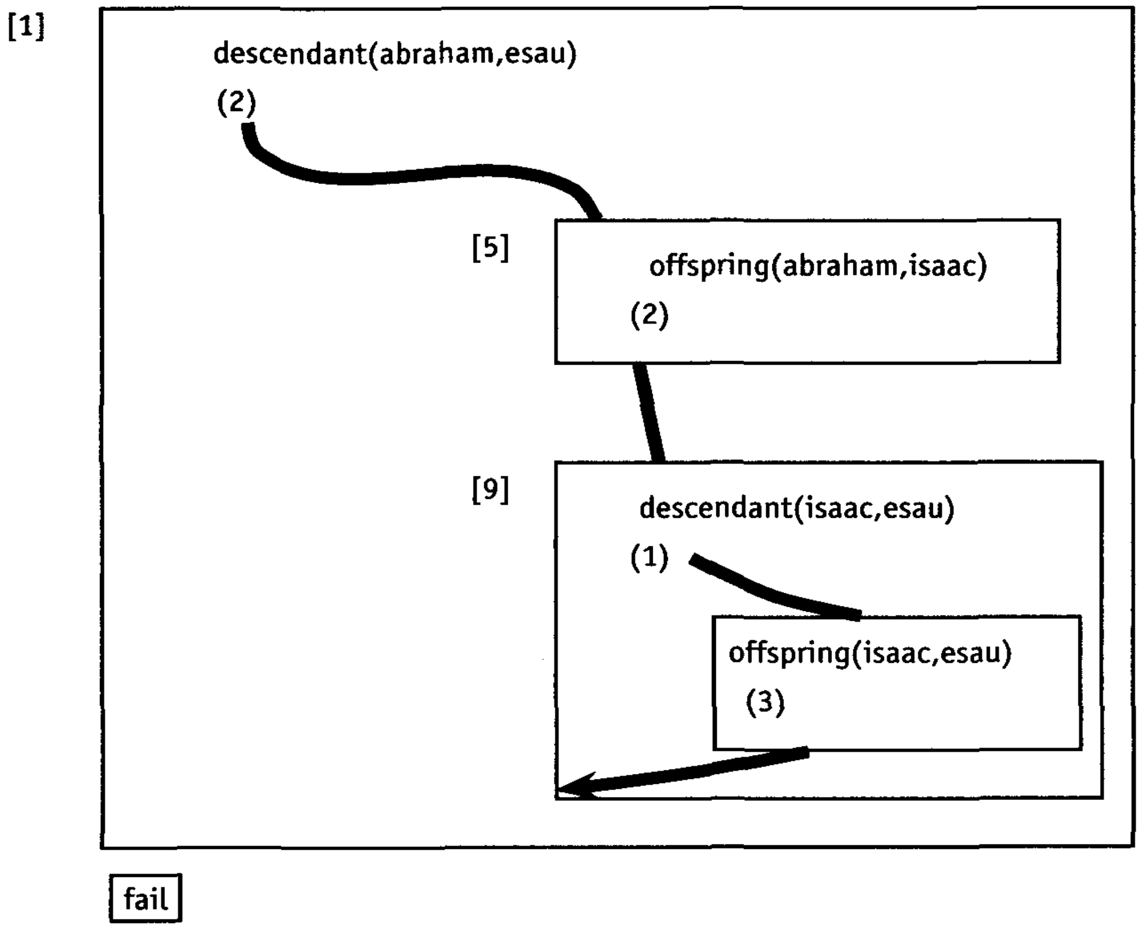
Back we go for an alternative.

```
[5] EXIT: offspring(abraham,isaac)
[9] CALL: descendant(isaac,Answer)
[10] CALL: offspring(isaac,Answer)
[10] EXIT: offspring(isaac,esau)
```

We get a new invocation of descendant and the offspring subgoal succeeds:

Continuing:

```
[9] EXIT: descendant(isaac,esau)
[1] EXIT: descendant(abraham,esau)
[11] CALL: fail
[11] FAIL: fail
[1] REDO: descendant(abraham,esau)
[9] REDO: descendant(isaac,esau)
```



This provides a final solution to the initial question, but the fail forces backtracking again and so back we come along the REDO paths.

```
[10] REDO: offspring(isaac,esau)
[10] EXIT: offspring(isaac,jacob)
[9] EXIT: descendant(isaac,jacob)
[1] EXIT: descendant(abraham,jacob)
```

The offspring subgoal has another alternative which produces another result for the initial descendant goal. As can be seen, this is Abraham's last possible descendant, however there is a certain amount of work left to be done. Let us continue to follow the control flow as it backtracks unsuccessfully back to the beginning.

```
[12] CALL: fail
[12] FAIL: fail
[1] REDO: descendant(abraham,jacob)
[9] REDO: descendant(isaac,jacob)
[10] REDO: offspring(isaac,jacob)
[10] FAIL: offspring(isaac,Answer)
[13] CALL: offspring(isaac,Y3)
```

We are now trying the second clause for descendant.

```
[13] EXIT: offspring(isaac,esau)
[14] CALL: descendant(esau,Answer)
```

Recur again.

```
[15] CALL: offspring(esau,Answer)
[15] FAIL: offspring(esau,Answer)
[16] CALL: offspring(esau,Y4)
[16] FAIL: offspring(esau,Y4)
[14] FAIL: descendant(esau,Answer)
[13] REDO: offspring(isaac,esau)
[13] EXIT: offspring(isaac,jacob)
[17] CALL: descendant(jacob,Answer)
```

Try jacob.

```
[18] CALL: offspring(jacob,Answer)
[18] FAIL: offspring(jacob,Answer)
[19] CALL: offspring(jacob,Y5)
[19] FAIL: offspring(jacob,Y5)
[17] FAIL: descendant(jacob,Answer)
[13] REDO: offspring(isaac,jacob)
[13] FAIL: offspring(isaac,Y3)
[9] FAIL: descendant(isaac,Answer)
[1] FAIL: descendant(abraham,Answer)

no
```

And that's the end of that. We hope that this exhaustive example has provided an understanding of the control flow involved in the execution of a Prolog program. You should have noticed that for any goal there is always only one CALL and FAIL, although there may be arbitrarily many REDOs and corresponding EXITs. In the next section, we look at the trace messages for a more complicated example: append.

Exercise 8.1: In the above model, no mention is made of how the cut goal "!" is handled. Extend the model to account for the action of cut.

8.4 Tracing and Spy Points

When you find that your program doesn't work (because it generates an error, just says "***no***" or produces the wrong answer), you will want to find out quickly where

Prof. William F. Clocksin
Oxford Brookes University
Department of Computing
Wheatley Campus
Oxford OX33 1HX, United Kingdom

Dr. Christopher S. Mellish
University of Edinburgh
Department of Artificial Intelligence
80 South Bridge
Edinburgh EH1 1HN, United Kingdom

ISBN 3-540-00678-8 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-00678-8 Springer-Verlag New York Berlin Heidelberg

ISBN 3-540-58350-5 4th Edition Springer-Verlag New York Berlin Heidelberg

Cataloging-in-Publication Data applied for

Clocksin, W.F. (William F.), 1955-
Programming in Prolog: using the ISO standard/W.F. Clocksin, C.S. Mellish.--5th ed. p.cm.
Includes bibliographical references and index.
ISBN 3-540-00678-8 (alk.paper)
1.Prolog (Computer program language) I.Mellish, C.S. (Christopher S.), 1954-II. Title.

QA76.73.P76C57 2003
005.13´3--dc21 2003044177

Printed on acid-free paper 41/3180XO - 5 4 3 SPIN 12626533