B.5 Informação sobre predicados

Durante a interacção com o PROLOG é comum ter-se a necessidade de consultar a definição de um predicado. O PROLOG fornece o meta-predicado `listing/1` que aceita como argumento o nome de um predicado definido no nosso programa e que tem como efeito mostrar a definição desse predicado no monitor.[2]

Exemplo B.5.1 Considerando o programa do Exemplo 7.3.1, podemos obter a seguinte interacção:

```
?- listing(ant).

ant(A, B) :-
        ad(A, B).
ant(A, C) :-
        ant(A, B),
        ad(B, C).
Yes
```

B.6 Rastreio de predicados

O PROLOG apresenta vários mecanismos para a depuração de programas, alguns dos quais são apresentados nesta secção.

Para compreender o rastreio da execução de predicados em PROLOG é importante começar por discutir os vários eventos que têm lugar quando o PROLOG está a tentar provar um objectivo. Distinguem-se quatro eventos durante a tentativa de provar um objectivo (apresentados na Figura B.3):

- `Call`. Este evento ocorre no momento em que o PROLOG inicia a prova de um objectivo. Este evento corresponde à passagem por um nó da árvore SLD no sentido de cima para baixo. O objectivo indicado corresponde ao primeiro literal do objectivo em consideração.

[2] Este predicado também pode mostrar o conteúdo completo de um ficheiro, se o argumento que lhe é fornecido corresponder ao nome de um ficheiro.

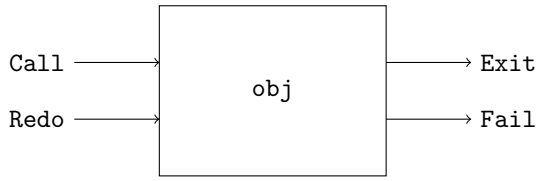

Figura B.3: Eventos associados à prova do objectivo `obj`.

- `Exit`. Este evento ocorre no momento em que o PROLOG consegue provar o objectivo, ou seja, quando o objectivo tem sucesso.
- `Redo`. Este evento ocorre no momento em que o PROLOG, na sequência de um retrocesso, volta a considerar a prova de um objectivo.
- `Fail`. Este evento ocorre no momento em que o PROLOG falha na prova de um objectivo.

O meta-predicado de sistema `trace/1`, aceita como argumento o nome de um outro predicado e permite seguir o rasto da avaliação do predicado especificado.

Exemplo B.6.1 Consideremos o programa do Exemplo 7.3.1:

```
ad(marge, bart).
ad(srB, marge).

ant(X, Y) :- ad(X, Y).

ant(X, Z) :- ant(X, Y), ad(Y, Z).
```

Recorrendo ao predicado `trace/1` para solicitar o rastreio dos predicados `ad` e `ant`:

```
?- trace(ad).
%         ad/2: [call, redo, exit, fail]
Yes
[debug]  ?- trace(ant).
%         ant/2: [call, redo, exit, fail]
Yes
```

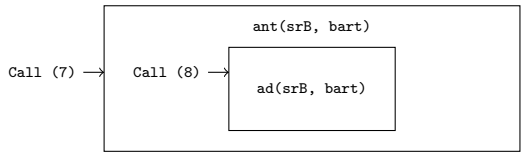

Figura B.4: Invocações para provar `ant(srB, bart)` e `ad(srB, bart)`.

a resposta do PROLOG a estas solicitações indica-nos que os quatro eventos associados à execução de predicados (`call`, `redo`, `exit` e `fail`) serão mostrados. O indicador `[debug] ?-` diz-nos que o PROLOG se encontra em modo de depuração.

Ao seguir o rasto da execução de um predicado, o PROLOG associa cada execução de um objectivo com um identificador numérico unívoco, chamado o *número da invocação*, o qual é apresentado, entre parênteses, sempre que o rastreio de um predicado está a ser utilizado.

Pedindo agora ao PROLOG para provar que `ant(srB, bart)`, obtemos a seguinte interacção:

```
[debug]  ?- ant(srB, bart).
 T Call: (7) ant(srB, bart)
 T Call: (8) ad(srB, bart)
 T Fail: (8) ad(srB, bart)
 T Redo: (7) ant(srB, bart)
 T Call: (8) ant(srB, _L172)
 T Call: (9) ad(srB, _L172)
 T Exit: (9) ad(srB, marge)
 T Exit: (8) ant(srB, marge)
 T Call: (8) ad(marge, bart)
 T Exit: (8) ad(marge, bart)
 T Exit: (7) ant(srB, bart)
Yes
```

As duas primeiras linhas do rastreio destes predicados:

```
 T Call: (7) ant(srB, bart)
 T Call: (8) ad(srB, bart)
```

indicam-nos que se iniciou uma prova do objectivo `ant(srB, bart)`, a qual

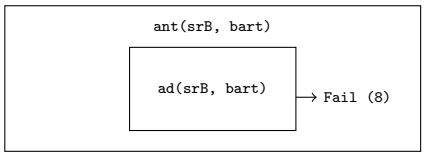

Figura B.5: Falhanço da primeira tentativa de provar o objectivo inicial.

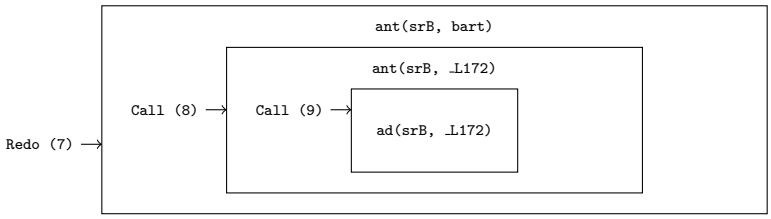

Figura B.6: Segunda tentativa de provar o objectivo inicial.

originou a prova do objectivo `ad(srB, bart)`. Este aspecto é apresentado esquematicamente na Figura B.4, a qual deverá sem comparada com a árvore SLD originada para este objectivo e apresentada na Figura 7.2. Na Figura B.4, se um rectângulo correspondente a uma invocação de um predicado, é desenhado dentro de um rectângulo correspondente a uma outra invocação, estão, isso significa que o objectivo que corresponde à invocação apresentada no rectângulo exterior originou a invocação do objectivo que está representada no rectângulo interior.

O objectivo `ad(srB, bart)` falha, o que é traduzido pela seguinte linha do rastreio do predicado

```
T Fail: (8) ad(srB, bart)
```

(Figura B.5), o que faz com que uma nova prova para o objectivo `ant(srB, bart)` seja tentada. Este facto é ilustrado pelas seguintes linhas apresentadas no rastreio dos predicados e cujos efeitos se ilustram na Figura B.6.

```
T Redo: (7) ant(srB, bart)
T Call: (8) ant(srB, _L172)
T Call: (9) ad(srB, _L172)
```

Nesta tentativa, é utilizada a variante da cláusula: `ant(srB, bart) :-`

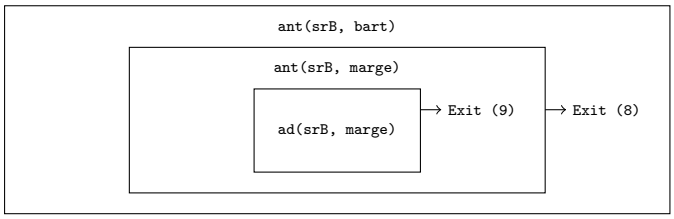

Figura B.7: Resultado da prova do objectivo `ant(srB, _L172)`.

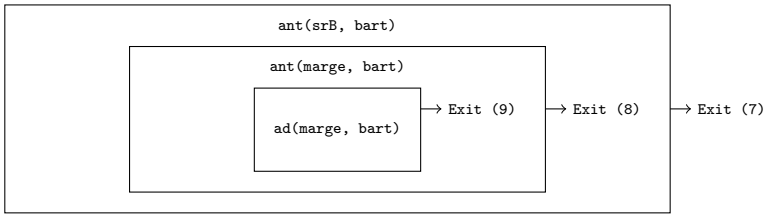

Figura B.8: Resultado da prova do objectivo inicial.

`ant(srB, _L172), ad(_L172, bart)`. Esta nova tentativa de prova corresponde ao ramo da árvore SLD que se mostra na Figura 7.3.

O objectivo `ad(srB, _L172)` unifica com `ad(srB,marge)` o que origina a seguinte informação durante o rastreio (correspondendo à Figura B.7):

```
 T Exit: (9) ad(srB, marge)
 T Exit: (8) ant(srB, marge)
```

Tendo provado o primeiro literal no corpo da cláusula `ant(srB, bart) :- ant(srB, _L172), ad(_L172, bart)` com a substituição {`marge/_L172`}, o PROLOG tenta agora provar o segundo literal, com sucesso como o indicam as seguintes linhas do rastreio dos nossos predicados (Figura B.8):

```
 T Call: (8) ad(marge, bart)
 T Exit: (8) ad(marge, bart)
 T Exit: (7) ant(srB, bart)
Yes
```

Indicação	Significado
"return"	siga para o próximo passo da execução ("creep")
`s`	continue a execução sem mostrar o rastreio de predicados até à próxima execução do predicado neste objectivo
`a`	aborta a execução
`i`	ignora o presente objectivo, fazendo com que este tenha sucesso
`n`	termina o modo de depuração, continuando a prova
`?`	mostra as indicações disponíveis

Tabela B.1: Indicações possíveis durante o rastreio interactivo.

Quando se solicita ao PROLOG que siga o rasto da execução de um ou mais predicados, este entra em modo de depuração, mudando o indicador para `[debug] ?-`. O modo de depuração pode ser terminado através da execução do predicado `nodebug/0`.

Para além do predicado de sistema `trace/1`, o PROLOG fornece outros predicados pré-definidos que auxiliam a tarefa de depuração. Apresentamos mais alguns destes predicados:

- `spy/1`. Este meta-predicado tem o efeito de efectuar o rastreio do predicado que é seu argumento. A diferença entre `spy/1` e `trace/1` reside no facto do meta-predicado `spy/1` originar um rastreio interactivo: no final de cada evento de execução, é solicitado ao utilizador que forneça uma indicação sobre o modo de prosseguir com a tarefa de rastreio. Algumas das possíveis indicações são apresentadas na Tabela B.1.[3]
- `trace/0`. Este predicado tem o efeito de ligar o mecanismo de rastreio. A partir do momento da execução deste predicado, todas as execuções de todos os predicados definidos no nosso programa são mostradas.

Exemplo B.6.2 Considerando, de novo, o programa do Exemplo 7.3.1, podemos gerar a seguinte interacção:

[3]Indicações adicionais podem ser consultadas no manual do PROLOG.

```
?- spy(ant).
% Spy point on ant/2
Yes

[debug]  ?- ant(X, bart).
   Call: (7) ant(_G312, bart) ? creep
   Call: (8) ad(_G312, bart) ? creep
   Exit: (8) ad(marge, bart) ? skip
   Exit: (7) ant(marge, bart) ? creep
X = marge ;
   Redo: (8) ad(_G312, bart) ? skip
   Redo: (8) ad(_G312, bart) ? skip
   Redo: (7) ant(_G312, bart) ? creep
   Call: (8) ant(_G312, _L172) ? creep
   Call: (9) ad(_G312, _L172) ? creep
   Exit: (9) ad(marge, bart) ? creep
   Exit: (8) ant(marge, bart) ? creep
   Call: (8) ad(bart, bart) ? creep
   Fail: (8) ad(bart, bart) ? creep
   Redo: (9) ad(_G312, _L172) ? creep
   Exit: (9) ad(srB, marge) ? creep
   Exit: (8) ant(srB, marge) ? creep
   Call: (8) ad(marge, bart) ? creep
   Exit: (8) ad(marge, bart) ? creep
   Exit: (7) ant(srB, bart) ? creep
X = srB
Yes
```


B.7 Informação de ajuda

Durante uma sessão com o PROLOG é possível obter informação de ajuda sobre vários tópicos. Para além do predicado pré-definido `listing` apresentado na secção B.5, existem dois outros predicados úteis, `help` e `apropos`:

- O meta-predicado pré-definido `help/1`, cujo argumento corresponde à entidade sobre a qual se pretende onter informação, permite consultar

o manual do PROLOG sobre predicados, funções e operadores existentes na linguagem.

Exemplo B.7.1 (Utilização de `help`)

```
?- help(is).
-Number is +Expr
    True  if Number has successfully  been unified with the number  Expr
    evaluates to.   If Expr evaluates to a float that can be represented
    using  an integer (i.e,  the value is  integer and within the  range
    that can be  described by Prolog's integer representation),  Expr is
    unified with the integer value.

    Note  that normally, is/2 should be used with unbound  left operand.
    If equality is to be tested, =:=/2 should be used.  For example:

             ?- 1 is sin(pi/2).   Fails!.   sin(pi/2) evaluates
                                  to the float  1.0, which does
                                  not unify with the integer 1.
             ?- 1 =:= sin(pi/2).  Succeeds as expected.
Yes
```


- O meta-predicado pré-definido `apropos/1`, aceita como argumento um tópico sobre o qual desejamos saber informação e mostra a informação existente no manual do PROLOG relacionada com o tópico indicado.

Exemplo B.7.2 (Utilização de `apropos`)

```
?- apropos(trace).
guitracer/0                 Install hooks for the graphical debugger
noguitracer/0               Disable the graphical debugger
gtrace/0                    Trace using graphical tracer
gdebug/0                    Debug using graphical tracer
gspy/1                      Spy using graphical tracer
trace/0                     Start the tracer
tracing/0                   Query status of the tracer
notrace/0                   Stop tracing
guitracer/0                 Install hooks for the graphical debugger
noguitracer/0               Disable the graphical debugger
trace/1                     Set trace-point on predicate
trace/2                     Set/Clear trace-point on ports
notrace/1                   Do not debug argument goal
spy/1                       Force tracer on specified predicate
```

```
leash/1                       Change ports visited by the tracer
visible/1                     Ports that are visible in the tracer
chr_trace/0                   Start CHR tracer
chr_notrace/0                 Stop CHR tracer
prolog_trace_interception/4   library(user) Intercept the Prolog tracer
prolog_skip_level/2           Indicate deepest recursion to trace
prolog_list_goal/1            Hook (user) Intercept tracer 'L' command
Section 12-2                  ''Intercepting the Tracer''
Yes
```