1 Alguns predicados não lógicos

Mesmo depois de termos visto tudo o que vimos, ainda temos alguns pontos a esclarecer. A começar pela avaliação do programa: computadores comuns não vem equipados com uma placa de clarevidência, e, assim, podem não conseguir adivinhar bem o caminho para a prova de um goal. Isto é, a hipótese de não-determinismo não segue na prática[1]. O ideal seria que, se um goal pode ser provado por um programa, o processo de prova seguisse um caminho direto, sem tentativas de unificações infrutíferas. Na prática, isso só é realizável para situações muito específicas e, no geral, serão tentadas diversas unificações infrutíferas antes de se chegar ao objetivo.

Para esse tipo de situação é criado o **choice point** logo antes de se tentar uma unificação. Se a tentativa resulta em falha, o processo volta ao estado anterior à tentativa, o que chamamos de **backtracking**, a possibilidade daquela unificação é eliminada e o processo continua (isto é, a mesma unificação não é tentada de novo). O goal falha quando todas as possibilidades foram eliminadas e tem sucesso quando não existirem mais unificações a serem feitas.

choice point

backtracking

Os *choice points* são um ponto chave na execução de um programa lógico. A quantidade deles está diretamente relacionada com a eficiência do programa: quanto mais *choice points*, em geral, mais ineficiente o programa é. Assim, se o mesmo goal puder ser provado de mais de uma forma diferente, é necessária a criação de *choice points* a mais, o que deve ser evitado (em outras palavras, programas determinísticos costumam ser mais eficientes). Da mesma forma, se existe mais de um resultado para uma computação (isto é, se o mesmo goal admite diversas substituições que resultam em sucesso), a eliminação das opções a mais implicariam na eliminação de *choice points*, o que seria vantajoso.

A quantidade de *choice points* tem muito a ver com a de "axiomas". Mais especificamente, tem a ver com a quantidade de cláusulas e com tamanho do corpo das cláusulas. Geralmente, um programa com menos axiomas resulta em melhor legibilidade e maior eficiência, no entanto, alguns desses axiomas podem representar restrições que podem levar uma falha mais cedo na computação, o que, na prática, diminuiria a quantidade de *choice points* (os que seriam criados, não houvesse essa falha, não serão mais) e de unificações desnecessárias, aumentando a eficiência do programa.

Veremos, então, métodos para lidar com essas situações. O mais importante, e mais polêmico, é o **corte**, `!/0` (um funtor de nome "!" e aridade 0). Intuitivamente, o que ele faz é se comprometer com a escolha atual, descartando as demais. Poderemos compreender melhor como ele funciona depois que desenvolvermos a interpretação de uma computação lógica como a busca em uma árvore, mas, enquanto isso, nossa interpretação intuitiva será o suficiente.

corte

Como um exemplo de seu uso, considere o seguinte programa:

[1] No sentido de que o programa pode não saber qual é "é a melhor escolha". Vale lembrar que, em outros contextos, não-determinismo indica apenas a presença de escolhas.

```
member(X,[X|Xs]).
member(X, [_|Xs]) :- member(X, Xs).
```

Código 1: Member 0

O símbolo "_" representa uma **variável anônima**. A utilizamos quando o nome daquela variável não é importante, o que acontece quando não a utilizarmos novamente (isso serve para não nos distrairmos com variáveis que não serão utilizadas: a variável anônima cumpre um papel meramente formal, de *placeholder*). Toda variável anônima é diferente, apesar de serem escritas iguais.

variável anônima

O goal `member(X, Xs)?` resulta em sucesso se X é um elemento de Xs[2]. Esse programa é usado primariamente de duas formas: para checar se um elemento é membro de uma lista; ou para gerar em X os elementos da lista. Por exemplo, para o goal `member(X, [a, b, c])?`, X pode assumir os valores de a, b ou c.

Se a programadora quiser saber apenas se um certo X faz parte de uma certa lista Xs, ela pode usar o corte, "cortando" as demais soluções:

```
member(X,[X|Xs]).
member(X, [_|Xs]) :- member(X, Xs), !.
```

Código 2: Member 1

O que o corte "diz" essencialmente é: "Tudo bem. Agora que chegamos a este ponto, não olhe para trás". A utilização desse novo programa para *member* é o mesmo do anterior, exceto que, agora, ao gerar membros da lista, só é gerado um elemento. Ao checar se outro elemento pertence a lista, assim que é obtido sucesso, o processo para.

Mencionamos anteriormente que o corte é usado para fins de eficiência. Quando ele é usado somente para esse fim, ou seja, se ele só serve para fazer o programa rodar melhor, não interferindo no seu significado, dizemos que é um **corte verde**. Quando o corte não é verde, dizemos que ele é vermelho. O que acabamos de ver foi um **corte vermelho**: goals como `member(b,[a,b,c])?` e `member(c,[a,b,c])?` não estão mais no significado. Perceba que o corte não é um predicado lógico, mas sim operacional: ele nos diz algo sobre "como" rodar um programa mas, a princípio, não sobre "o que" ele é.

corte verde

corte vermelho

O corte é, provavelmente, o predicado não lógico mais importante e polêmico no Prolog. O que o torna polêmico é justamente o fato de ser não lógico. Dissemos anteriormente que a ideia da programação lógica era lidar com a parte da lógica (o "o que fazer"), abstraindo a parte procedural (o "como fazer"). O corte é um exemplo em que isso não foi obtido.

[2]Estamos assumindo tacitamente, neste programa e nos demais, que a ordem de execução é "de cima para baixo, da direita para a esquerda" (o interpretador "enxerga" primeiro a cláusula que aparece "antes"), que é como as implementações usuais de Prolog funcionam. É importante, entretanto, notar que não é assim por necessidade e existem outras "ordens" de avaliar programas Prolog

Apesar disso, se deixarmos de lado nosso preconceito, o corte pode contribuir de maneiras interessantes para a lógica do programa.

Por exemplo, o `not/1` pode ser implementado de maneira similar à seguinte:

```
not(Goal) :- Goal, !, fail.
not(Goal).
```

Código 3: Not

Onde o predicado `fail/0` é um *built-in* do Prolog que falha sempre.

Similarmente, podemos, a partir do corte, gerar outras formas de controle, familiares a programadores de linguagens procedurais:

```
se_entao_senao(A, B, R) :-
  A, !, B.
se_entao_senao(A, B, R) :-
  R.
```

Código 4: SES

```
or(A, B) :- A, !.
or(A,B)  :- B.
```

Código 5: OR

Perceba que, diferentemente do que ocorre em programas puramente lógicos[3], a ordem das cláusulas e dos termos em cada cláusula podem ser fundamentais. Pode ser que, por exemplo, na cláusula `p(a) :- b, c.`, b resulte em falha e c em um processo interminável. Neste caso, a mudança de ordem seria fatal.

Os programas acima demonstram uma possível forma de se definir, respectivamente, o "se, então, senão" e o "Ou"[4], mas essas construções já existem no Prolog por padrão, como:

- se, então, senão: `A -> B ; R.` (lê-se: se A, então B, senão R);
- ou: `A ; B.` (lê-se: A ou B).

1.1 Outras Opções

Na realidade, é preciso notar que a perspectiva anterior (usando *choice points* e *backtrackings*) não é a única possível (mas é a usada no Prolog padrão). Que

[3] Você pode se perguntar se algum do programa nesse texto é puramente lógico. Alguns dos mais simples, como o *Natural*, do Capítulo 2, podem ser, mas a situação geral é que os programas que veremos são só "meio que" lógicos.

[4] Na verdade, poderíamos pensar como o processo de *backtracking* como o nosso "ou" natural, mas às vezes pode ser conveniente usar um "ou" em uma cláusula, seja por efeito de legibilidade ou para evitar o *backtracking*.

ela poderia não ser a única possível deve ser intuitivo, já que, a princípio, se começamos a descrever a programação lógica de forma realmente lógica, a ordem em que as cláusulas são postas, assim como a ordem dos termos em cada cláusula, não deveriam importar (na lógica clássica, $A \vee B$ é o mesmo que $B \vee A$), o que deixa de ser o caso quando introduzimos artifícios tais como *choice points* e *backtracking*.

Artifícios desse tipo são necessários (ou, ao menos, parecem necessários), no entanto, quando queremos transformar um programa lógico em algo que possa ser executado. Em outras palavras, não é suficiente termos um conjunto de relações que impliquem alguma outra relação: precisamos de um mecanismo para construir uma prova de que o conjunto de relações implica de fato a outra relação. Uma prova (como interpretamos aqui) é uma sequência de passos que deriva inferências[5] a partir de relações conhecidas anteriormente.

O que isso significa é que a escolha do *backtracking* para resolver "tentativas infrutíferas", como posto acima, é uma entre outras e, possivelmente, não a mais eficiente para todos os casos. Atualmente, existem outros métodos como *dynamic backtracking*, *partial order dynamic backtracking* e *backjumping*, com funcionamentos diferentes.

No entanto, entendemos que, no que se segue, fazer uso de outro mecanismo que não o *backtracking* adicionaria complexidade na exposição e pouco *insight* ao entendimento. Caso o leitor interessado queira consultar detalhes, uma boa fonte de consulta é [1]

[5] Convém termos uma ideia da diferença entre "implicação" e "inferência": "A implica B" não implica que "de A se infere B". Ter uma prova de "A implica B", sim.

Leituras adicionais

[1] Final Technical Report, "DYNAMIC BACKTRACKING", February 1997, University of Oregon