1 Restrições passivas e implementações no Eclipse

Antes de sairmos por aí resolvendo CSPs e COPs, será útil termos a distinção entre restrições ativas e passivas. Resumidamente, restrições ativas podem alterar o estado das variáveis, enquanto que restrições passivas não podem e são mais usadas para fins de testes. Por exemplo, `r(a,X) = r(Y,b).` é uma restrição ativa: X precisa tomar o valor de b, e Y de a. Mas `4*X < Y + 2` é uma restrição passiva.

Por enquanto, nos preocuparemos mais com restrições passivas e veremos exemplos de sua utilização no sistema ECL^iPS^e. Valeria deixarmos aqui uma breve introdução à história desse sistema e uma explicação mais detalhada de porquê escolhemos usá-lo e não outro, mas nos contentamos em dizer que o ECL^iPS^e é uma expansão suficientemente completa do Prolog para lidar melhor com restrições. Não cabe darmos aqui uma detalhada mostra do que ele adiciona, mas convém falarmos brevemente sobre como alguns de seus iteradores funcionam, já que os usaremos extensivamente daqui em diante.

Enquanto em Prolog o único método iterativo é a recursão, no ECL^iPS^e dispomos de algumas opções a mais. Em particular, temos:

- `foreach(El, Lista) do busca(El).`
  Itera Busca(El) ordenadamente sobre cada elemento El de Lista;
- `fromto(Prim, In, Out, Ult) do busca(In, Out).`
  Itera Busca(In, Out) de In = Prim até Out = Ult.

Existem diversos outros iteradores para propósitos diferentes, todos eles seguindo o padrão *(iterador do busca)*. Iteradores podem ser postos em conjunto como *(iterador1, iterador2, ..., iteradorn do busca)*. Ao fazer isso, todos os iteradores dão o passo junto, por assim dizer, e o conjunto de iteradores para quando qualquer um deles chegar ao fim. Também podemos aninhar iteradores (como se colocássemos um *for* dentro de outro em uma linguagem convencional) da seguinte forma: *(iterador1 do (iterador 2 do ... (iteradorn do busca)))*.

1.1 Backtracking no Prolog/Eclipse

Já discutimos rapidamente a busca por *backtracking* antes, agora veremos como implementá-la. Para tanto, precisamos decidir qual será o método de ramificação usado, qual a ordenação das variáveis e qual a ordenação dos valores de cada variável.

O método de ramificação usado aqui será o *labelling*. O próximo passo é decidir a ordem das variáveis. Isso pode ter um grande impacto na busca, apesar de a quantidade de folhas na árvore de busca continuar sendo a mesma para qualquer ordem: a diferença está na quantidade de nós internos na árvore. Por exemplo, para um CP nas variáveis X e Y, em que X pode tomar dois

valores e Y pode tomar 3 valores diferentes, a quantidade de folhas na árvore de busca é 3×4=12. Fazendo o *labelling* do X antes do Y, temos dois nós internos, enquanto que, fazendo o *labelling* do Y antes do X, temos três nós internos. A presença de maior quantidade de nós internos na árvore de busca torna a busca mais difícil, sendo razão razoável para que busquemos fazer antes o *labelling* das variáveis com menor domínio. Veremos depois que, na presença de restrições ativas, a ordenação das variáveis pode ter grande influência no desempenho do algoritmo.

Mencionamos que uma forma de descrever a escolha de valores de forma mais geral é por meio de alocação de crédito. Uma parte de um programa que implementa essa ideia é o seguinte, que assume que os valores de cada domínio já estão ordenados segundo uma preferência:

```
% Busca(Lista, Credito) :-
%   Busca por solucoes com um dado credito

busca(Lista, Credito) :-
  ( fromto(Lista, Vars, Resto, []),
    fromto(Credito, CreditoAtual, NovoCredito, _)
  do
    escolhe_vari(Vars, Vari-Dominio, Resto),
    escolhe_val(Dominio, Val, CreditoAtual, NovoCredito),
    Vari = Val
  ).

escolhe_val(Dominio, Val, CreditoAtual, NovoCredito) :-
  compartilha_credito(Dominio, CreditoAtual, DomCredLista),
  member(Val-NovoCredito, DomCredLista).
```

Ele assume que Lista é uma lista de pares variável-domínio e precisa ser completada pelas escolhas de `escolhe_var/3` e `compartilha_credito/3`. O seguinte exemplo de `compartilha_credito/3` corresponde à escolha dos N primeiros valores, se N for menor que o tamanho do domínio; ou do domínio todo, caso contrário:

```
% compartilha_credito(Dominio, N, DomCredLista) :-
%    Admite apenas os primeiros N valores.

compartilha_credito(Dominio, N, DomCredLista) :-
  ( fromto(N, AtuCredito, NovoCredito, 0),
    fromto(Dominio, [Val|Tail], Tail, _),
    foreach(Val-N, DomCredLista),
    param(N)
  do
    ( Tail = [] ->
```

```
      NovoCredito is 0
    ;
      NovoCredito is AtuCredito - 1
    )
  ).
```

Essa escolha ocorre atribuindo aos primeiros N valores do domínio o mesmo crédito, de N. Outra escolha de compartilha_crédito, possivelmente mais natural, é a que envolve a atribuição de N créditos ao primeiro valor, N/2 ao segundo, e assim por diante:

```
compartilha_credito(Dominio, N, DomCredLista) :-
  ( fromto(N, AtuCredito, NovoCredito, 0),
    fromto(Dominio, [Val|Tail], Tail, _),
    foreach(Val-NovoCredito, DomCredLista),
  do
    ( Tail = [] ->
      NovoCredito is 0
    ;
      NovoCredito is AtuCredito fix(ceiling(AtuCredito/2))
    )
  ).
```

Nesse código, o `fix(ceiling(AtuCredito/2))` retorna o maior inteiro menor ou igual que AtuCredito/2.

1.2 Variáveis não-lógicas

Ocasionalmente será útil, como uma medida da eficiência de um programa, quantificar coisas como a quantidade de sucessos em uma computação ou a quantidade de *backtrackings*. Para isso, o ECL^iPS^e permite a utilização de variáveis não-lógicas e oferece quatro meios de lidar com elas:

- `setval/2`;
- `incval/1`;
- `getval/1`;
- `decval/1`;

O que define uma variável como não-lógica é que seu valor não muda com o *backtracking*. Além disso, variáveis não-lógicas não são capitalizadas e a única forma de mudar ou acessar o valor delas é por meio de um dos predicados acima.

Segue uma implementação de nosso programa de busca que conta a quantidade de *backtrackings*[1]:

```
busca(Lista, Backtrackings) :-
  inicia_backtrackings,
  ( fromto(Lista, Vars, Resto, []),
  do
    escolhe_vars(Vars, Vari-Dominio, Resto),
    escolhe_vals(Dominio, Val),
    Vari = Val,
    conta_backtrackings
  ),
  pega_backtrackings(Backtrackings).

inicia_backtrackings :-
  setval(backtrackings, 0).

pega_backtrackings(B) :-
  getval(backtrackings, B).

conta_backtrackings :-
  on_backtracking(incval(backtrackings)).

on_backtracking(_).
on_backtracking(Q) :-
  once(Q),
  fail.
```

Esse programa explora a forma como é feito o *backtracking* e, por isso, a ordem em que foi posta é crucial. Vale notar que ele conta todos os backtrackings que ocorrem na busca, possibilitando contar a quantidade de nós na árvore.

Frequentemente, no entanto, pode ocorrer um *backtracking* entre mais de um nível. Isso ocorre quando, logo depois de realizar um, é realizado outro *backtracking*. Uma medida melhor de eficiência pode ser uma contagem de *backtrackings* que conta uma sequência ininterrupta como sendo apenas um. Um `conta_backtracking/0` que faz isso é dado a seguir:

```
conta_backtrackings :-
  setval(single_step,true).
conta_backtrackings :-
  getval(single_step,true),
  incval(backtrackings),
  setval(single_step,false).
```

[1] Esse `once/1`, usado no programa, definido como `once(Goal) :- Goal, !.`

1.3 A biblioteca suspend

Voltando a resoluções de CPs aritméticos e booleanos, introduzimos a biblioteca ECL^iPS^e *suspend*. A biblioteca *suspend* lida com restrições aritméticas suspendendo a avaliação delas até que as variáveis tenham sido instanciadas e possam ser avaliadas. Caso elas não se tornem instanciadas até o fim da busca, o resultado é uma restrição, que é o que queríamos.

No ECL^iPS^e existem duas formas de se usar uma biblioteca: pode-se colocar um `:-library(nome_da_biblioteca).` no início do arquivo utilizado ou, ao usar um predicado da biblioteca nome_da_biblioteca, colocar `nome_da_biblioteca:(...).`. Um exemplo de uso de *suspend* é: `suspend:(2 < Y + 4), Y = 3.`, que resultaria em erro em Prolog puro. Caso a biblioteca *suspend* já tenha sido carregada, essa restrição pode ser reescrita como `2 $< Y + 4, Y = 3.`, em que o $ indica que a restrição é usada tal como na biblioteca *suspend*.

Essa biblioteca também lida com restrições booleanas (para as quais os valores de variáveis são 0 ou 1 e os símbolos de restrições são tais como `or/2`, `and/2`, `neg/1` e `=>/2`, de implicação) e permite a declaração de variáveis de formas distintas.

Uma delas é por meio do *range*: `suspend(X :: 2..10).`, ou `suspend(X #:: 2..10).` que gera uma variável X cujo valor é restrito ao intervalo de inteiros entre 2 e 10. Se a biblioteca já estiver carregada, que é o que assumiremos daqui para frente, essa restrição pode ser escrita como `X :: 2..10.`. Se quisermos usar intervalos reais no lugar de intervalos de inteiros (que é o padrão), podemos escrever algo como `X $:: 2..10.`. Alternativamente, pode-se usar a restrição `integers/1` ou `reals/1` para restringir a variável ou lista de variáveis a assumir valores nos inteiros ou reais, respectivamente.

A biblioteca *suspend* também permite a criação de suspensões arbitrárias pelo usuário a partir de `suspend/3`. O primeiro argumento de `suspend/3` indica a restrição a ser suspensa, o segundo indica a prioridade da suspensão (o que nos dá a ordem de execução de restrições que deixam a suspensão juntas) e o terceiro a condição de saída da suspensão, escrito como `Termo -> Condicao` (dizendo que na ocorrência da *Condicao*, em relação a Termo, a restrição deixa a suspensão), em que "varCondicao" geralmente é *inst*, indicando que o Termo é instanciado. Um exemplo é `suspend(X =:= 21, 2, X -> inst)`, indicando que a restrição de que X =:= 21 está em estado de suspensão até que X seja instanciada.

Talvez se lembre do programa Ou Exclusivo do Capítulo 6. Com o uso de `suspend/3`, podemos reimplementá-lo sem recorrer ao *backtracking*:

```
% ou_exclusivo(X, Y) :-
%   sucesso se X xor Y eh 1, falso caso contrario

ou_exclusivo(X, Y) :-
  ( nonvar(X) ->
      sus_y_xor(X,Y),
    ;
      suspend(sus_y_xor, 3, X->inst)
  ).

sus_y_xor(X,Y) :-
  ( nonvar(Y) ->
      xor(X,Y)
    ;
      suspend(xor(X,Y), 3, Y->inst)
  ).

xor(1, 0).
xor(0, 1).
```

Note que, agora, nosso `ou_exclusivo/2` não é mais uma operação aritmética, agindo como um predicado relacional como os demais.