1 Restrições ativas

Regras de propagação como as vistas no Capítulo passado possibilitam o que é chamado "restrição ativa". Sem essas regras de propagação, tínhamos que buscar a solução de nossos problemas "na mão", isto é, por aquele método primitivo de testar cada solução e dascartar as que falham. Isso ainda era verdade com o uso da bilioteca *suspend*. Considere, por exemplo, a restrição `suspend:(X or Y), X = 0.`. Essa restrição tem a solução $X = 0, Y = 1$, mas, na forma como está escrita, a restrição não nos permite encontrar essa solução (`X or Y` estará suspenso até que Y receba um valor, o que não acontece, já que `or/2` só lida com variáveis instanciadas). Existem, no entanto, no sistema ECL^iPS^e, bibliotecas que lidam com a restrição de forma "ativa" (isto é, fazem uso de propagações). Veremos como lidar com duas delas momentaneamente, a *ic_symbolic* e a *ic*.

Considere a seguinte situação (adaptada de [1]):

Depois da queda do comunismo em Absurdolândia, vários clubes de debate "Preto e Branco" cresceram rapidamente pelo país. Eles eram clubes exclusivos: apenas antigos *Colaboradores Secretos* (do antigo Serviço de Segurança Comunista) ou antigos *Oposicionistas Honrados* (que costumavam serem caçados pelos antigos membros Serviço de Segurança Comunista). Esse tipo de associação fez bastante sucesso, já que proveu um solo fértil para discussões contraditórias, amadas pelo público, pela mídia televisiva e por jornalistas. Isso também impulsionou o consumo de todo aquele tipo de bebida que tem uma merecida reputação de facilitar o entendimento de assuntos complicados. A atratividade das discussões foi ainda mais realçada pelo conhecimento comum de que *Oposicionistas Honrados* sempre dizem a verdade, enquanto que *Colaboradores Secertos* diziam alternadamente a verdade e a mentira. O bem conhecido tablóide local "Notícias da Cloaca" delegou a um dos clubes um Jornalista Celebrado para fazer uma reportagem promovendo a ideia de reconciliação entre os inimigos do passado. Infelizmente, o Jornalista Celebrado encontrou um problema: no momento de sua chegada, o clube tinha apenas três membros, dos quais Membro1 e Membro2 argumentavam ferozmente, evidentemento porque pertenciam a diferentes grupos de membros. O jornalista, não querendo importunar os adversários, perguntou ao Membro3, que não tomava parte no argumento, se ele costumava ser um *Oposicionista Honrado* ou um *Colaborador Secreto*. Infelizmente, Membro3 já tinha tomado muito das bebidas supramencionadas e resmungou algo ininteligível. O Jornalista Celebrado perguntou então aos dois membros restantes sobre o que o Membro3 havia dito. Membro1, que talvez devido a alguma habilidade que havia praticado, pôde entender a resposta do Membro3, afirmou que o Membro3 disse que havia sido um *Oposicionista Honrado*. No entanto, Membro2 primeiro disse que Membro3 foi um *Colaborador Secreto* e, então, adicionou que o Membro3 havia mentido. O Celebrado Jornalista tem informação suficiente para inferir quem é quem?

Para resolver esse problema, faremos uso das bibliotecas ic e ic_symbolic, a

qual é uma adição à biblioteca ic, implementando variáveis e restrições sobre sobre domínios simbólicos ordenados (como já mencionado anteriormente). Para modelarmos esse problema, faremos uso das seguintes observações básicas:

- Membro1 e Membro3 disseram alguma coisa;
- Membro2 disse duas coisas;
- Cada coisa que foi dita pode ser verdadeira ou falsa;
- Sabemos o que Membro1 e Membro2 disseram, mas só sabemos o que o Membro3 possivelmente disse.

Assim, temos uma variável para cada membro, que é ou um *Oposicionista Honrado* ou um *Colaborador Secreto*, assim como para cada coisa dita por eles (daqui para frente referida como "resmungo"), que pode ser verdadeira ou falsa. Mais importante são as relações entre essas variáveis.

Para a resolução do problema, notamos que os domínios a serem usados pela ic_symbolic precisam ser declarados, o que faremos por uso da construção `:- local domain(domain_name(`$domain\_value_1$`, ..., `$domain\_value_n$`))` em que "domain_name" representa o nome do domínio e $domain\_value_i$ representa o i-ésimo valor (simbólico) no domínio. O valor de cada variável de resmungo pode ser "verdadeiro" ou "falso", aqui representados pelos valores aritméticos 0 e 1. O valor de cada variável de Membro pode receber os valores "oposicionista_honrado" e "colaborador_secreto". O que cada Membro disse é uma afirmação sobre o grupo ao qual outro membro faz parte ou sobre a veracidade do que outro Membro disse. Essas observações são traduzidas no seguinte código:

```
:- lib(ic).
:- lib(ic_symbolic).
:- local domain(membro_do_clube(oposicionista_honrado, colaborador_secreto)).


% Oposicionistas Honrados sempre dizem a verdade
% Colaboradores Secretos podem dizer a verdade ou mentir
resmungo_unico(Membro, Verdade) :-
    (Membro &= oposicionista_honrado) => Verdade.

% Colaboradores secretos mentem e dizem a verdade alternadamente
resmungos_consecutivos(Membro, Verdade1, Verdade2) :-
    (Membro &= colaborador_secreto) #= (Verdade1 #\= Verdade2).

resmungos_consecutivos(Membro, Verdade1, Verdade2) :-
    (Membro &= colaborador_secreto) #= (Verdade1 #\= Verdade2).
```

```
%% resolve([?Membro_1, ?Membro_2, ?Membro_3])
% Membro_x é unificado com o nome do seu respectivo grupo
%

resolve([Membro_1, Membro_2, Membro_3]):-
    [Membro_1, Membro_2, Membro_3] &:: membro_do_clube,
    [Membro_3_possivelmente_disse, Membro_3_disse, Membro_1_disse,
     Membro_2_disse_primeiro, Membro_2_disse_entao] :: 0..1,
    Membro_1 &\= Membro_2,

    % O que Membro_3 possivelmente disse
    % O que Membro_1 disse
    % O que Membro_2 disse primeiro
    % O que Member_2 disse entao
    Membro_3_possivelmente_disse #= (Membro_3 &=oposicionista_honrado),
    resmungo_unico(Membro_3, Membro_3_possivelmente_disse),

    Membro_1_disse #= (Membro_3_disse #=Membro_3_possivelmente_disse),
    resmungo_unico(Membro_1, Membro_1_disse),

    Membro_2_disse_primeiro #= (Membro_3 &=colaborador_secreto),
    resmungo_unico(Membro_2, Membro_2_disse_primeiro),

    Membro_2_disse_entao #=(Membro_3_disse #= 0),
    resmungo_unico(Membro_2, Membro_2_disse_entao),

    resmungos_consecutivos(Membro_2, Membro_2_disse_primeiro, Membro_2_disse_entao),
    ic_symbolic:indomain(Membro_1),
    ic_symbolic:indomain(Membro_2),
    ic_symbolic:indomain(Membro_3),
    writeln("Membro_1":Membro_1),
    writeln("Membro_2":Membro_2),
    writeln("Membro_3":Membro_3),
    writeln("Membro_2_disse_primeiro":Membro_2_disse_primeiro),
    writeln("Membro_2_disse_entao":Membro_2_disse_entao).
```

É um código simples e que não precisa de muitos comentários, mas cabe alguns:

- As relações entre os Membros (termos simbólicos) são realizadas por restrições de ic_symbolic (vale lembrar, os "#" indicam que a restrição tem o significado usual, mas nos inteiros);
- As entre o que eles disseram (ou possivelmente disseram), por restrições aritméticas (de ic);

- Enquanto `[Membro_1, Membro_2, Membro_3] &:: membro_do_clube` indica o domínio das variáveis, `ic_symbolic:indomain(Membro_1)` faz a atribuição de valores (segundo as restrições);
- Em `resmungo_unico(Membro, Verdade):- (Membro &= oposicionista_honrado) => Verdade`, o símbolo "`=>`" é o de implicação como em lógica clássica;
- Restrições como `(Membro &= colaborador_secreto)` são reificadas, isto é, assumem um valor de "verdadeiro" ou "falso" (na prática, 0 ou 1);
- As linhas tais como `writeln("Membro_1":Membro_1)` escrevem "Membro1 : valor", onde "valor" é o valor de Membro1.

Uma particularidade desse código é que ele resolve o problema sem a necessidade de *backtracking*, no que é chamado de *backtracking-free search* (apesar de ser uma propagação e não uma busca propriamente dita). Essa foi uma situação excepcional, uma vez que geralmente é necessário fazer uma busca para se chegar à solução (na verdade, mesmo quando não é necessário usar busca, sua introdução pode agilizar o processo).

1.1 Backtracking raso

Busca por *backtracing* raso é um processo de busca no qual é permitido *backtracking*, mas de forma limitada: ao atribuir um valor a uma variável, o processo de propagação é desencadeado e, se ocorre uma falha, o próximo valor é tentado. Para realizar essa busca, precisamos de predicados que nos permitam o acesso do domínio atual de uma variável. Um desses predicados é o `get_domain_as_list/2`. Ele toma como primeiro argumento uma variável com um domínio e o segundo argumento é instanciado a uma lista com os valores contidos do domínio da variável. Com essa lista em mãos, podemos testar cada valor a partir de `member/2`: `get_domain_as_list(X, Dominio), member(X, Dominio)`. Essa combinação (de `get_domain_as_list/2 e member/2`) é tão usada que foi criado o predicado `indomain/1`[1] para realizar a mesma função (exceto que de forma mais eficiente). O predicado `indomain/1` age como se definido por:

```
%% indomain(+X)
% Insiste que a variável X faça parte de seu domínio

indomain(X) :-
  get_domain_as_list(X,member(X, Domain).
  member(X, Domain),
```

Em mãos desse predicado, podemos fazer o *backtracking* raso como se segue:

[1] Perceba como ele foi usado no programa anterior

```
%% backtrack_raso(+Lista)
% Para cada variável em Lista, tente atribuir um valor em seu domínio

backtrack_raso(Lista) :-
  ( foreach(Var,Lista) do once(indomain(Var)) ).
```

Note que busca por *backtracking* raso não é um resolvedor completo.

1.2 Busca por backtracking

Fazer uma busca por *backtracking* por meio da enumeração de todos os valores no domínio não é eficiente na presença de propagadores, já que os domínios diminuem (ou, ao menos, assim esperamos). No lugar disso, seria mais interessante fazer uma busca por *backtracking*que faça uso apenas dos valores nos domínios atuais. Essa observação nos leva a uma revisão do programa de busca apresentado no Capítulo 8:

```
%% busca_com_dom(+Lista)
% Faz a busca por backtracking nas variáveis de Lista
%

busca_com_dom(Lista) :-
  ( fromto(Lista, Vars, Resto, [])
  do
    choose_var(Vars, Var, Resto),
    indomain(Var).
  ).

choose_var(Lista, Var, Resto) :- Lista = [Var | Resto].
```

Leituras adicionais

[1] Niederliński, Antoni, “A gentle guide to constraint logical programming via Eclipse”, 3rd edition, Jacek Skalmierski Computer Studio, Gliwice, 2014