LÓGICA, RESTRIÇÕES E OTIMIZAÇÃO

Príncipios de otimização relacional

por

Éricles Lima, Marina Andretta

ICMC-USP, São Carlos, 2018

Conteúdo

Introdução

Motivação

"Optimization and constraint programming are beginning to converge, despite their very different origins. Optimization is primarily associated with mathematics and engineering, while constraint programming developed much more recently in the computer science and artificial intelligence communities. The two fields evolved more or less independently until a very few years ago. Yet they have much in common and are applied to many of the same problems. Both have enjoyed considerable commercial success. Most importantly for present purposes, they have complementary strengths, and the last few years have seen growing efforts to combine them. Constraint programming, for example, offers a more flexible modeling framework than mathematical programming. It not only permits more succinct models, but the models allow one to exploit problem structure and direct the search. It relies on such logic-based methods as domain reduction and constraint propagation to accelerate the search. Conversely, optimization brings to the table a number of specialized techniques for highly structured problem classes, such as linear programming problems or matching problems. It also provides a wide range of relaxations, which are often indispensable for proving optimality." – J.N.Hooker

Notação

Ao decorrer do presente texto, usaremos as seguintes convenções:

- Palavras em língua estrangeira estarão *like this*, exceto quando explicitamente dito o contrário;
- A eventual definição de algum termo ou expressão terá o termo ou definição **assim**, com o objetivo de facilitar sua localização; **assim**
- Quando quisermos destacar algum termo por qualquer outro motivo, ele estará *assim*;
- Para trechos de código escritos no meio do texto, será usada `esta fonte`.

Documentação de códigos

As observações nesta subseção servem mais como referência e seriam melhor apreciadas depois de lidos os primeiros capítulos (em particular, o capítulo inicial).

Uma particularidade do Prolog é que, no caso geral, a mesma variável pode ser tanto "de entrada" quanto "de saída" e, como argumento de um funtor,

pode ser tanto instanciada como não instanciada. No entanto, com alguma frequência, um programa é otimizado ou construído para apenas um dos casos. Isto é, frequentemente espera-se que uma variável já esteja instanciada ao ser associada a um funtor (ou o contrário, espera-se que ela não esteja instanciada), ou que uma variável só sirva como variável de saída.

Dada a forma como programas lógicos são escritos, esses casos podem não ser claros e deseja-se que sejam documentados. Para tanto, usaremos a convenção de, na documentação do funtor ou restrição (quando existente, geralmente na forma de um comentário logo antes do funtor ou restrição), indicar os seguintes modos:

- "Variáveis de entrada", isto é, unificadas a outro termo do programa, serão indicadas por um prefixo "+" (como em `f(+X)`);
- Exceto quando são esperadas estarem unificadas com termos base, quando serão indicadas pelo prefixo "++" (como em `f(++X)`);
- "Variáveis de saída", isto é, cuja expectativa é de não estarem unificadas a termo algum, serão indicadas pelo prefixo "-" (como em `f(-X)`);
- Quando não for necessária uma distinção entre "variável de entrada" ou "de saída", a variável será indicada pelo prefixo "?", como o Y em `f(++X, ?Y, -Z)`.

Adicionalmente, o ECL^iPS^edisponibiliza `comment/2`, muito útil tanto para a documentação do código quanto para, possível auxílio do compilador. Não usaremos essa funcionalidade neste texto, por questões de claridade, mas sua utilização é incentivada para códigos "em produção" (consulte o manual para detalhes, assim como para obter um *style guide*, que pode ser de interesse).

Organização

Nosso objetivo principal aqui é introduzir o paradigma de programação por restrições como uma extensão natural do paradigma de programação lógica, com um foco em resolução de problemas, em particular problemas de otimização. Para isso, começamos do básico, tentando dar uma justificativa às escolhas feitas no texto e usando, para isso, de exemplos e código da bibliografia. Caso seja encontrado qualquer irregularidade, erro, ou queira fazer alguma observação, sinta-se livre para enviar um e-mail a ericlesaquileslima at gmail dot com, ou contatá-lo pelo meio que te for mais conveniente.

O texto pode ser razoavelmente bem dividido em três partes:

- Capítulos do 0 ao 6: Desenvolvimento de conceitos básicos e introdução a Programação Lógica, com alguns exemplos;

- Capítulos 7 ao 11: Desenvolvimento de alguns conceitos básicos de programação por restrições, com alguns exemplos;
- Capítulos 8 adiante: Desenvolvimento de mais conceitos de programação por restrições, com um maior foco em resolução de problemas e otimização.

Vale notar que, apesar de usarmos nos exemplos, principalmente, Prolog e ECL^iPS^ee desenvolvermos no texto os conceitos básicos necessários para sua compreensão, este texto não pretende substituir um manual ou qualquer outro texto feito especificamente para esses sistemas.

Agradecimentos

Isto que está lendo provavelmente não existiria não fosse o auxílio proveniente do Programa Unificado de Bolsas da USP e aos incentivos da Marina Andretta, à época minha orientadora, que resolvou apoiar e fazer o que pôde para ajudar o no desenvolvimento projeto. A eles, só tenho gratidão a expressar.
– Éricles Lima

Imagem de capa retirada de
https://kathleenhalme.com/explore/oak-clipart-tamarind-tree/#gal_post_875_oak-clipart-tamarind-tree-1.png, acesso em Setembro de 2018.

0 Dando nome aos bois

Foi a influência da filosofia grega que fez da matemática uma ciência. De fato, os primeiros matemáticos gregos estão entre os primeiros filósofos, como é o caso de Tales e Pitágoras. A noção de que um fato matemático pode ser demonstrado é fruto da interação entre matemática e filosofia. Afinal, uma demonstração é essencialmente um argumento para esclarecer como um certo fato é consequência de algo que já conhecemos. E se há alguma coisa que os filósofos gregos gostavam de fazer era argumentar.

S. C. Coutinho - Números Inteiros e Criptografia RSA

Nosso objetivo final é aprendermos a resolver e desenvolver problemas de otimização por restrições lógicas. Para tanto, precisaremos de algumas ferramentas, talvez a mais importante das quais, que será a base de outras, é a programação lógica. Programação lógica é um paradigma de programação, como o de programação funcional, a de programação procedural, entre outros, mas, em princípio, lidaremos com ela como a linguagem Prolog (de *"Programmation Logique"*). Isso rende a vantagem de lidarmos com algo concreto, enquanto perdemos pouco do poder de abstração. Um detalhe que vale notar é que existem vários diferentes "sabores" de Prolog, cada um com suas peculiaridades, na maior parte de natureza técnica. Ignoraremos essas peculiaridades sempre que possível (este texto não tem a intenção de ser um texto técnico sobre Prolog), nos focando, sempre que possível, no "padrão de facto" Edinburgh Prolog.

Como dizia Sussman [3] , quando se quer aprender uma nova linguagem, as perguntas básicas que precisamos perguntar são

1. Quais são as "coisas" primitivas da linguagem?
2. Quais são seus meios de combinação, com os quais podemos usar as primitivas de maneira estruturada?
3. Quais são seus meios de abstração, com os quais podemos usar programas menores para criar programas maiores?

A resposta à primeira pergunta é que o Prolog tem uma primitiva: o **funtor**. **funtor**
O termo *funtor* foi introduzido por Rudolf Carnap, filósofo alemão do círculo

de Viena, em seu *Logische Syntax der Sprache*[1] e indica uma *palavra função* (não confundir com "uma função"), que contribui primariamente com a sintaxe de uma sentença (em contraste com *palavras conteúdo*, que contribuem primariamente com o significado): resumidamente, em sua concepção original, funtores expressam a estrutura relacional que palavras tem umas com as outras. O termo atualmente também é usado em outras áreas (além de em linguística e programação lógica), como em *Teoria de Categoria*, com significado, em essência, semelhante.

Para nossos propósitos, é uma simplificação razoável dizer que funtores expressam relações. Na linguagem natural, somos restritos a usar os funtores presentes na língua usada (mesmo uma "licença poética" não vai muito longe disso). Na programação lógica, os funtores só precisam ser sintaticamente corretos. Em Prolog, um funtor f de aridade n (aridade é a quantidade de parâmetros, ou "símbolos relacionados pelo funtor"), dito `f/n`, é escrito com uma sintaxe semelhante à de uma função:

`f(`a_1`, ..., `a_n`)`

em que os argumentos a_i são outros funtores. Em particular, símbolos e números são ditos funtores de aridade zero, também chamados **átomos***. Quando um funtor `f/n` não é um átomo, ele, na presença de seus argumentos, é dito um **termo composto** de **funtor principal** f. Mais em geral, um **termo** é um funtor ou variável (o que é uma variável e como se comporta veremos mais para frente) e, quando corresponder a um átomo ou uma variável, dizemos que ele é atômico.

átomos

termo composto

funtor principal

termo

Dados funtores `f/3` e `p/1`, exemplos de seus usos são:

- f(a,b,c)
- f(1,a,8)
- f(p(9),c,e)

Os exemplos acima são úteis como primeiro exemplo de "como escrever um funtor" mas, como colocados, não têm significado. Isto porque funtores expressam relações, mas relações arbitrárias entre símbolos e números arbitrários não têm necessidade de existir. A maneira de dizermos que uma dada relação existe em Prolog é por meio de um **fato**. Um fato em Prolog é expresso como um funtor, junto de seus argumentos, seguido de um ponto. Por exemplo

fato

```
mais(1,2,3).
```

diz que "é verdade que mais(1,2,3)" ou, mais naturalmente, "é verdade que existe a relação 'mais' entre 1, 2 e 3, nessa ordem" (vale notar que a ordem é importante: `mais(1,2,3)` não é o igual a `mais(3,2,1)`, em que por "igual

*Em Prolog, nem todo símbolo pode ser tratado como átomo (embora os usuais sejam). Quando na dúvida, recomendamos testar antes de confiar.

a" entende-se "idêntico a"). Ocasionalmente, é muito mais conveniente que um funtor receba argumentos pela direita e pela esquerda, quase como um operador, como `1 f 2`. É possível definir funtores dessa forma, mas nos restringiremos ao modo usual, pelo qual a última relação fica `f(1,2)`.

Eventualmente pode ser que, na ocasião de a relação `p(a,b)` ser verdadeira, outras relações também sejam. Em particular, pode ser que, em linguagem matemática, $f(1, l) \Rightarrow p(a, b)$. Esse tipo de relação é escrita em Prolog como:

```
p(a,b) :- f(1,l).
```

ou, se ($f(1, l)$ e $q(f)$) $\Rightarrow p(a, b)$, escrevemos

```
p(a,b) :- f(1,l), q(f).
```

na qual as vírgulas entre os termos fazem o papel do "e" lógico. Esse tipo de estrutura é chamada **regra** e é o mais importante meio de combinação em programação lógica. O que está para a esquerda de ":-" em uma regra é dita "cabeça da regra", ou só "cabeça", se a regra for clara do contexto, e o que está à direita de ":-" é dito o "corpo da regra". Quando não for necessário distinguir entre fatos e regras, usaremos o termo **cláusula*** para nos referir a qualquer um dos dois.

regra

cláusula

Um **programa lógico** consiste em um conjunto de cláusulas. Um exemplo é o seguinte:

programa lógico

```
pao(de).
pao(de, queijo).
com(cafe) :- tem(cafe).
tem(cafe) :- cafe.
cafe.
```

Código 1: Café

Note que os nomes usados para os funtores são arbitrários. O seguinte programa tem essencialmente o mesmo significado do ponto de vista computacional:

```
queijo(lua).
queijo(lua, pao).
coma(terra) :- gem(terra).
gem(terra) :- terra.
terra.
```

Código 2: queijo

Pelo código 1, por exemplo, podemos dizer que "é verdade que pao(de,

*Note que usamos "cláusula" em um sentido diferente do da lógica clássica (em que "cláusula" é definida como sendo uma disjunção de proposições).

queijo) e que com(cafe)". De fato, a pergunta mais geral que podemos fazer a um programa lógico é se existe alguma relação r entre os termos a_1, ..., a_n.

E essa constitui a maneira primária de se utilizar um programa lógico, que pode ocorrer, por exemplo, em um *prompt* no computador. Por meio de questões, ou buscas [*], às vezes também chamadas objetivos ou "**goals**" (daqui para frente preferiremos usar "goal", sem itálico[†], exceto quando for conveniente utilizar "busca" ou "objetivo"). Um goal é escrito como um fato e é o que se busca "provar" a partir do programa. Intuitivamente, pode-se pensar no programa lógico como um conjunto de axiomas e no goal como uma hipótese que se quer provar a partir desses axiomas.

goals

Assim como dizemos que uma busca foi um sucesso se foi encontrado o que se gostaria de encontrar e uma falha caso contrário, diremos que um goal relativo a algum programa pode ter o status de sucesso se ele pode ser provado a partir do programa, e de falha se não. Note que, em Prolog, "falhar" não é o mesmo que "quebrar". A diferença é essencialmente a mesma entre receber uma resposta de "não" a uma pergunta e receber uma resposta de "não entendo sua pergunta". Se foi um goal mal escrito (se o compilador "não entende a pergunta"), pode ocorrer um erro (isto é, o goal gera um erro quando não é "compreensível" a partir da gramática utilizada pelo compilador usado)[‡] e o programa pode não ser executado.

Mais especificamente, um goal é uma conjunção[§], escrita como $p\_1, p\_2, ..., p\_n$, em que p_i é entendida como uma proposição que pode ser verdadeira ou falsa [¶] (ou, interpretando como busca, que pode ter sucesso ou falha).

Um ponto interessante sobre goals e também sobre regras é que são como cláusulas de Horn[‖] (isto é, cláusulas do tipo C se A_1 e ... e A_n). Como pode imaginar, a teoria existente sobre cláusulas de Horn é de alguma importância para programas lógicos. Mais especificamente, dada uma disjunção[**] com ao menos um termo não negado, esta é dita uma *cláusula de Horn*. Se existe exatamente um termo não negado, a seguinte identidade é de simples constatação:

$$X_1 \vee \neg X_2 \ldots \neg X_n \Leftrightarrow [(X_2 \wedge \ldots X_n) \Rightarrow X_1]$$

em que o símbolo $\neg$ corresponde à negação lógica usual.

---

[*] Porque busca é um termo apropriado deve ficar cada vez mais claro no decorrer do texto.

[†] Lembre-se que, em geral, usaremos termos em língua estrangeira, em itálico.

[‡] Se um goal que gera erro é de fato um goal, é discutível, mas não entraremos nesse mérito.

[§] Isto é, uma sequência de proposições unidas por um *e* lógico. Algo como `p_1` e ... e `p_n`, em que os `p_i` são proposições.

[¶] Na verdade, como veremos, uma interpretação mais próxima da realidade do programa lógico é uma baseada na lógica construtivista, mas, por enquanto, é suficiente pensar em um goal como uma conjunção no sentido clássico.

[‖] Também, apesar de mais raramente, chamadas *cláusulas de McKinsey*. Elas foram usadas primeiramente por McKinsey, como notado por Horn [2]. É importante notar que a "cláusula" usada aqui não é a mesma "cláusula" definida anteriormente, mas a "cláusula" da lógica clássica

[**] Proposições unidas por "ou".

Para melhor diferenciarmos goals de fatos, goals serão, aqui, escritos como: `p?`

apesar de, na natureza*, não haver essa distinção (neles, goals são, como fatos, escritos com um ponto final no lugar de com um ponto de interrogação). Na verdade, a falta de distinção entre goals, fatos e termos de uma cláusula, como deve ficar mais claro nos próximos capítulos, faz sentido, já que no processo de resolução (que será explicado depois) eles cumprem papel semelhante.

Agora, considere o seguinte código:

```
arvore_b(vazio).
arvore_b(arvore(A, D, E)) :-
  arvore_b(D),
  arvore_b(E).
```

Código 3: Árvore Binária

O goal *arvore_b*`(vazio)?` tem sucesso, já que é um dos fatos. O goal *arvore_b*`(arvore(raiz, vazio, vazio))?` também tem, já que *arvore_b*`(a, B, C)?` tem sucesso se *arvore_b*(A) e *arvore_b*(B) tiverem, o que ocorre. Perceba que, neste caso, o goal não tem sucesso segundo a primeira cláusula apenas e nem segundo a segunda. Ambas contribuem para a definição de *arvore_b*.

Utilizamos sempre a convenção de que termos capitalizados (isto é, com inicial maiúscula) denotam variáveis, enquanto os demais denotam constantes (a leitora descobrirá que, por notável coincidência, o Prolog faz uso da mesma convenção). Termos que não contém variáveis são ditos **termos base**. Variáveis serão explicadas mais a fundo no futuro.

termos base

No geral, o que um programa lógico deveria fazer (isto é, o que a programadora tem em mente ao escrevê-lo) pode não ser a princípio claro. Numa tentativa de aliviar isso, usaremos aqui a convenção de usar nomes significativos, assumindo o significado usual (a não ser quando dito o contrário), para os elementos relevantes. Assim, por exemplo, o seguinte trecho:

```
filho(c,b).
pai(Pai, Filho) :- filho(Filho, Pai).
```

Código 4: Pai e Filho

Indica que Pai tem a relação *pai* com Filho se Filho tem a relação *filho* com Pai (em outras palavras, um Pai é pai de um Filho se Filho é filho de Pai). Mas não é assumido que o interpretador do programa tenha conhecimento sobre o que é *pai* ou *filho*, ou seja, essa interpretação é relevante para a leitora do programa apenas (o que também significa que a leitora do programa pode

*Isto é, no Prolog padrão.

ler *pai* e *filho* de várias formas e um programa pode, assim, ser interpretado de diversas maneiras).

Com essa discussão em mente, será ocasionalmente útil ter o *significado* de um programa lógico definido de forma algo mais precisa:

Definição 0.1. ***Significado de um programa lógico*** *é o conjunto de goals deduzíveis a partir dele (isto é, os goals que obtêm sucesso se aplicados ao programa).*

Significado de um programa lógico

Na verdade, essa é a definição procedural do significado de um programa lógico (apesar de que, como veremos posteriormente, essa distinção é imaterial) e, vale notar, ela é sempre bem definida (isto é, todo programa lógico tem um significado bem definido). Nesse contexto, o significado intencionado pela programadora (isto é, "o que ela quer dizer com o programa") é um conjunto de goals que pode ou não estar contido no significado do programa. Assim, podemos nos perguntar "Será que o programa diz tudo que se quer que ele diga?" (isto é, se o significado intencionado está contido no significado do programa), ou "Será que tudo o que ele diz é correto?" (isto é, se o significado do programa está contido no intencionado). No primeiro caso, se o significado intencionado estiver contido no do programa, dizemos que o programa é **completo** e, no segundo, que ele é **correto**.

completo

correto

Por exemplo, digamos que o programa 4 seja um trecho de um programa no qual a programadora deseja modelar as relações entre programas e especificações de software atuais e antigos, de forma que A é pai de B se A veio antes de B e B herda características de A, como partes do código ou especificações (como é possível ver, ela apenas começou a escrever o programa). Assim, estão no significado intencionado goals como `pai(gnu, linux)?` e `pai(dos, windows)?`, enquanto que o significado do programa é `pai(d,c)?`.

Fica a pergunta: esse programa é correto? E ele é completo?

Referências

[1] Carnap, Rudolf, Logische Syntax der Sprache, Wien (Viena): Julius Springer (1968)

[2] Horn, Alfred, On Sentences Which are True of Direct Unions of Algebras, J. Symbolic Logic 16 (1951), no. 1, pp 14

[3] Abelsson, Harold and Sussmann, Gerald J. and Sussman Julie Structure and Interpretation of Computer Programs, second edition MIT Press

1 Modelo Computacional de Programas Lógicos

Considere o seguinte código:

```
resistor(energia,n1).
resistor(energia,n2).
transistor(n2,ground,n1).
transistor(n3,n4,n2).
transistor(n5,ground,n4).

inversor(Entrada, Saida) :-
  transistor(Entrada, ground, Saida),
  resistor(energia, Saida).

porta_nand(Entrada1, Entrada2, Saida) :-
  transistor(Entrada1, X, Saida),
  transistor(Entrada2, ground, X),
  resistor(energia, Saida).

porta_and(Entrada1, Entrada2, Saida) :-
  porta_nand(Entrada1, Entrada2, X),
  inversor(X, Saida).
```

Código 5: Circuito

A leitora pode se perguntar qual o resultado do seguinte goal:

```
porta_and(Entrada1, Entrada2, Saida)?
```

Ela também pode se perguntar "Será que, dado um programa qualquer e um goal qualquer dá para "calcular" o resultado do goal?".

Se houvesse muitos programas com goals de resultados incalculáveis, programação lógica não seria tão útil e dificilmente teria sido feito um material como este (mais difícil ainda é o material ter sido feito e a leitora estar lendo), então esse não deve ser o caso, alguém poderia dizer.

Se o goal estiver expresso no programa apenas como um fato base, prová-lo é fácil: só precisamos checar se algum dos fatos é igual ao goal. No caso mais geral, entretanto, se o goal conter alguma variável ou só puder ser provado através de alguma regra, a situação pode ficar mais complicada.

Se o goal contiver variáveis, para prová-lo precisamos encontrar uma substituição para cada uma delas de forma que cada um dos termos do goal seja logicamente consistente com o programa. Aqui o que queremos dizer com "substituição" é que em todo lugar em que a variável aparecer, ela é substituída por um outro termo (em outras palavras, uma substituição S é um conjunto, possivelmente vazio, de atribuições de valores a variáveis). Uma forma de pensar sobre isso é que, antes da substituição, a variável "tem uma vida só sua", sendo

irrestrita, e que, depois, sua vida é, na verdade, "a vida de outro", ou seja, é, em algum sentido, restrita. Mais precisamente, temos:

Definição 1.1. *Dado um termo p(a_1, ..., a_n), onde os a_j, para $j \in J$, J algum conjunto indexador, são variáveis, uma* ***substituição*** *é um conjunto ι de* ***unificações****, escritas como $a_i = k_j$, onde k_j é uma variável ou um termo atômico e "=" indica que a_i é idêntica a k_j e dizemos que a_i é unificado com k_j. Uma substituição ι sobre um programa P é escrita $P\iota$.*

substituição

unificações

Observações.

1. Ao realizar uma substituição ι sobre um programa P, o resultado é ou programa Pι, sobre o qual podemos, em particular, fazer outras substituições;
2. A relação "$A = B$" deve ser entendida como usado em álgebra (isto é, como denotando uma relação simétrica de igualdade entre A e B) e não como geralmente usado em programação, como um operador de atribuição assimétrico (em que $A = b$ não é o mesmo que $b = A$);
3. O símbolo "=" expressa a relação de dois termos serem idênticos;
4. Essa relação é transitiva: se A, B e C são variáveis e se `A = B` e `B = C`, então `A = C` (se A é idêntico a B e B é idêntico a C, então A é idêntico a C);
5. Pelo item (2), não podemos fazer* `A = 1` e, logo em seguida, `A = 2`: isso resulta em falha, por inconsistência.

Se temos que existe alguma substituição ι (possivelmente vazia) para que $p(a_1, ..., a_n) = q(b_1, ..., b_n)$, dizemos que $p(a_1, ..., a_n)$ é unificável com $q(b_1, ..., b_n)$. "=" é o **símbolo de unificação**[†‡].

símbolo de unificação

Convém notar aqui que, em Prolog, funtores são *cidadãos de primeira classe*, o que significa que compartilham o direito e privilégio de ser nomeado por uma variável (isto é, uma variável pode receber qualquer funtor n-ário, não só átomos).

Normalmente, quando lidando com outros tipos de linguagens, também seria dito que:

*Na verdade, podemos. Mas é como se não pudéssemos. Isso vai ficar mais claro quando lidarmos com *backtracking*.

†Veremos depois mais predicados do tipo "a op b", onde op é o operador (no caso, op é =/2). Predicados desse tipo são chamados de infixos. Todo predicado infixo também pode ser usado no formato prefixo (como =(a, b)) e predicados infixos também podem ser definidos pelo programador, como já mencionado anteriormente.

‡ Rigorosamente, "predicados" são diferentes de "funtores", mas neste texto adotaremos a convenção usual de nos referir a um "funtor" como "predicado" quando quisermos enfatizar seu caráter procedural.

1. Cidadãos de primeira classe podem ser retornados por funções e passados como argumentos a elas e

2. Que podem ser incorporados em estruturas de dados.

Como em programação lógica, a princípio, não fazemos uso de funções*, não podemos fazer a afirmação 1, mas, na prática, o que dizemos é equivalente. Isso porque, apesar de um funtor `f/n` não retornar um valor propriamente dito, se queremos um "valor de retorno", sempre podemos fazer um `f/n+1`, cujo último argumento (ou qualquer outro, mas costumamos optar pelo último pela conveniência de fazer apenas uma escolha) seria o valor de retorno (como já foi visto no funtor `length/2`, por exemplo) e o termo nessa última posição, se for uma variável, pode ser unificado com um funtor. Quanto à afirmação 2, podemos dizer somente que os funtores são o tijolo e cimento das estruturas de dados em programação lógica.

Voltando ao tema das substituições, todas elas são iguais, mas algumas são mais iguais que outras. Em particular, dado um programa P e substituições ι e ν, se existe alguma substituição η tal que $(P\iota)\eta = P\nu$, dizemos que ι é uma substituição mais geral do que ν. A substituição mais geral é de especial importância, porque podemos expressar outras substituições em função dela.

Agora podemos expressar nosso objetivo de provar o goal a partir do programa como o de achar uma substituição tal que cada termo do goal seja unificável com alguma cláusula do programa. Mais precisamente, um goal é provado a partir do programa se é possível unificar cada termo do goal com alguma cláusula do programa de forma a preservar a consistência das cláusulas. O processo pelo qual esse objetivo é realizado é chamado **processo de resolução**.

processo de resolução

Unificação exerce um papel fundamental na programação lógica. Na prática, ele resume processos de atribuição de valores, gerenciamento de memória, invocação de funções e passagem de valores, entre outros. O primeiro estudo formal sobre unificação é devido a John A. Robinson [1], que depois de montar um algoritmo de unificação, gerou o primeiro de que temos conhecimento.

O algoritmo dele é um tanto ineficiente e só foi citado por considerações históricas. Usaremos um outro no lugar. Antes, vale lembrar que um programa lógico é um conjunto de regras que recebe um goal (ou uma busca) e retorna *sucesso* (ou, sim, ou verdadeiro, dependendo da preferência pessoal) se a busca tem sucesso ou *falha* (ou, não, ou falso...) se não.

Como discutido acima, para provar um goal a partir de um programa é suficiente que tenhamos um algoritmo de unificação. Esse algoritmo recebe uma equação do tipo $T_1 = T_2$, e devolve uma substituição mais geral† para as variáveis presentes caso tal substituição exista, ou falha, caso contrário. O

*Em programação lógica pura não há funções no sentido usual, mas nas encarnações concretas de programação lógica, elas costumam aparecer de um jeito ou de outro.

†É importante que seja a mais geral, para não perdermos possíveis soluções

algoritmo que utilizaremos faz uso de uma pilha para armazenar as sub-equações a serem resolvidas e de um espaço Γ para armazenar a substituições:

(a) Faça o *push*[*] da equação na pilha;

(b) Se a pilha estiver vazia, devolva sucesso. Se não, faça o *pop* de um elemento (uma equação) $T_1 = T_2$ da pilha. Realize uma das ações a seguir, segundo a equação retirada:

1. Se T_1 e T_2 forem termos unários idênticos, nada precisa ser feito;
2. Se T_1 é uma variável e T_2 um termo não contendo T_1, realize uma busca na pilha pelas ocorrências de T_1 e troque T_1 por T_2 (o mesmo é feito em Γ);
3. Se T_2 for uma variável e T_1 for um termo não contendo T_2, a ação tomada é análoga ao do passo anterior;
4. Se T_1 e T_2 forem termos compostos de mesmo funtor principal e aridade, $f(a_1, ..., a_n)$ e $p(b_1, ..., b_n)$, adicione as equações $a_i = b_i$ na pilha;
5. Em outro caso, devolva falha.

(c) Retorne ao passo (b).

Intuitivamente, esse algoritmo tenta provar a equação de forma construtiva: isto é, tenta construir uma solução por meio de substituições e, se não chegar a uma contradição, termina com sucesso, "devolvendo" (não no sentido de uma função que devolve um valor, mas no de "mostrar" ao usuário do programa) a substituição realizada.

Para provar um goal G, escolhemos não-deterministicamente[†] a cabeça de uma cláusula T do programa, construímos uma equação do tipo `G = T` e aplicamos o algoritmo acima. Caso ele devolva sucesso, fazemos o mesmo com cada termo do corpo da cláusula. Caso devolva falha, seleciona-se outra cláusula e é realizado o mesmo processo, até que não haja mais cláusulas a serem selecionadas, quando o goal retorna falha.

O passo 2.b do algoritmo merece uma explicação um pouco mais detalhada. Ele diz implicitamente que x não é unificável com algum `y(a_1, ...,x,`

[*]Os *push* e *pop* devem ser entendidos como realizadas em uma pilha: *push* põe um elemento na pilha, *pop* retira.

[†]No geral, podem existir várias escolhas possíveis e pode ser que, por algumas sequências de escolhas de cláusulas, nunca cheguemos a uma prova do goal, apesar de ele ser deduzível a partir de outras escolhas de cláusulas. Quando dizemos que a escolha é não-determinística, queremos dizer que, se existem mais de um conjuntos de escolhas que provam o goal, um desses conjuntos é escolhido (a escolha é feita entre as cláusulas que podem provar o goal, o que significa que, se ele é provável, ele é provado). Na prática, isso pode ser implementado apenas aproximadamente. Ainda assim, é uma abstração importante e leva a aplicações interessantes, na assim chamada, *programação não-determinística*. Em outros contextos, também podemos usar esse mesmo termo para nos referir a situações nas quais o programa tem, a princípio, mais de uma "escolha" (se não há "escolhas", dizemos que o contexto é determinístico).

`...a_n)`, isto é, com algum funtor que tome x como argumento. Pode parecer estranho a princípio, mas a estranheza some se se lembrar que funtor não é função: um funtor representa uma estrutura primariamente simbólica. Sem essa condição, se `X = y(a_1, ..., X, ..., a_n)`, então `X = y(a_1, ...,y(a_1, ..., x, ..., a_n), ..., a_n) = y(a_1, ..., y(a_1, ...,y(a_1, ...,y(a_1, ...,x, ..., a_n), ..., a_n) , ..., a_n) , ..., a_n)` em um ciclo sem fim. Com um processo desses, não dá para provar um goal e, portanto, é devolvida falha.

Para entender melhor, tome o exemplo do código **??**, no início deste Capítulo, e suponha que àquele código é submetido o goal `resistor(energia, n1)?` O algoritmo é aplicado como se segue:

1. Tentaremos a unificação do goal com a cláusula na primeira linha do programa: a equação `resistor(energia, n1) = resistor(energia, n1)` é posta na pilha;
2. Uma equação é retirada da pilha: a equação `resistor(energia, n1) = resistor(energia, n1)`;
3. A equação é formada por dois funtores termos compostos de mesmo funtor principal e mesma aridade: as equações `energia = energia` e `n1 = n1` são postas na pilha;
4. É retirada uma equação da pilha: a equação `energia = energia`. Como os dois lados da equação são idênticos, não há mais o que fazer;
5. É retirada outra equação da pilha: a equação `n1 = n1`. Como os dois lados da equação são idênticos, não há mais o que fazer;
6. A pilha está vazia: o programa devolve sucesso, com a substituição $\Gamma = \{\}$ (substituição vazia).

1.1 Programas "gera-e-testa"

Programação não determinística não serve apenas para o desenvolvimento da teoria de computação de programas lógicos, também serve como uma abstração útil para a criação de programas interessantes.

Imagine que você se encontra em uma situação problemática e gostaria de resolver o problema. Um procedimento possível é gerar uma provável solução e, então, testar se ela de fato resolve o problema. Se formos traduzir isso para programação lógica, teríamos algo como:

Para algum *gera* e algum *testa*. A hipótese de não-determinísmo está na esperança de que será gerada uma solução que passa no teste, o que não é, a princípio, claro ser possível. Na prática, isso seria aproximadamente resolvido com o artifício do *backtracking*, que veremos posteriormente.

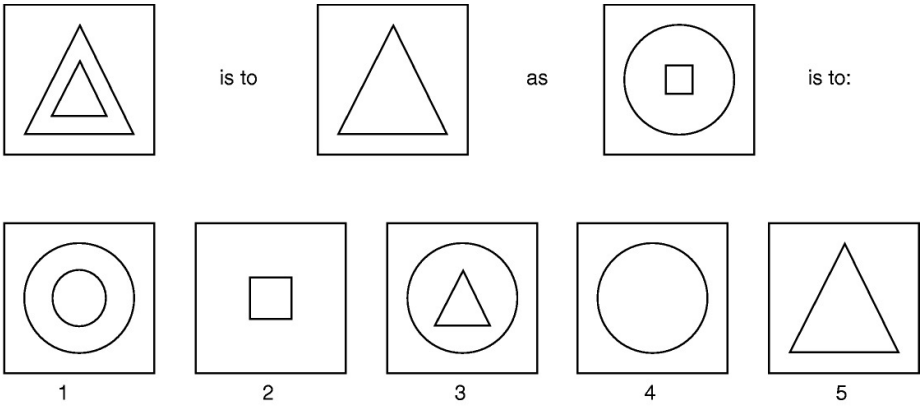
```
encontra(X) :-
  gera(X),
  testa(x).
```

Código 6: Encontra

Gera-e-testa é um modelo comum para a resolução de vários problemas. Frequentemente, no entanto, o *testa* está mesclado com o *gera*, de modo a tornar o procedimento mais eficiente*. Muitas vezes a programadora não precisa se preocupar com isso, tornando essa uma abstração útil. Talvez isso fique mais claro com o seguinte exemplo.

O exemplo de programa gera-e-testa que usaremos é o "ANALOGY". Considere o problema de encontrar analogias geométricas, como o mostrado na figura 7 †:

Figura 1: Retirado de [http://cs-alb-pc3.massey.ac.nz/notes/59302/l01.html], acesso em 03/11/2017.



Um possível algoritmo para resolver esse problema é o seguinte:

1. Ache uma operação que relaciona os objetos‡ para os quais conhecemos a relação "is_to";
2. Aplique essa operação no objeto dado para gerar um outro objeto;
3. Cheque se o objeto gerado está entre as opções listadas:
   - Se não estiver, volte ao passo (1);

*Na verdade, em geral, tenta-se pôr o teste tão dentro do gerador quanto possível, levando a um gasto menor de tempo de processamento com soluções inúteis.

†Esse problema foi retirado da edição de 1942 do "Teste psicológico para calouros de faculdade", do conselho americano de educação [2].

‡Usaremos "objeto" como um termo geral para nos referir a algo a que não queremos nos dar ao trabalho de definir rigorosamente.

- Caso contrário, termine.

No problema específico mostrado, podemos ver que, na primeira linha, a relação entre o primeiro diagrama e o segundo é que o segundo é o primeiro quando se retira a figura no centro. Assim, uma resposta ao problema seria encontrar um diagrama na segunda linha que corresponda ao terceiro da primeira menos a figura do centro (isto é, um círculo dentro de um quadrado, o diagrama 4 na segunda linha).

O programa a seguir implementa esses passos em um programa lógico*:

```
analogy(is_to(A, B), is_to(C, X), Answers) :-
  match(A, B, Operation),
  match(C, X, Operation),
  member(X, Answers).

match(inside(Figure1, Figure2),
      inside(Figure3, Figure2),
      exclude_center) :-
  Figure1 = inside(Figure5, Figure6),
  Figure3 = Figure6.

match(inside(Figure1,Figure2),
      inside(Figure2,Figure1),
      invert).
```

Código 7: Analogy

Esse programa é muito específico: ele toma a analogia entre apenas dois objetos e, a partir disso, cria uma analogia com um terceiro. Uma generalização é possível, mas, para nossos propósitos, isso é o suficiente.

Para que ele funcione, a maneira como os diagramas são representados é fundamental. Estando representados apropriadamente, `match/3` nos dá a operação que relaciona um objeto A com um B. Com isso em mãos, só precisamos aplicar a mesma operação ao termo C, por meio de `match/2`, achando X, o objeto que queríamos. Vale ressaltar que `match/2` está sendo usado de duas maneiras diferentes†: para encontrar a relação entre dois termos e para "fabricar" um termo com uma relação desejada. O predicado `member/2` ainda não foi explicado, e só o será melhor entendido posteriormente: por enquanto, é suficiente assumir que

*O "=" usado nesse programa é o mesmo da "substituição" mencionada acima e é a relação de identidade: A = B ⇔ A é idêntico a B.

†Esse tipo de comentário provém de uma leitura procedural: do ponto de vista estritamente lógico, `match/2` apenas expressa uma relação, que pode ser verdadeira ou falsa (isto é, pode existir ou não existir). Pensar do ponto de vista lógico é conveniente para fazermos programas mais elegantes e gerais, mas, sem uma leitura procedural adequada, não conseguiríamos trabalhar com alguns dos programas que veremos mais para frente.

`member(A, B)` se B for uma lista (essa coisa entre colchetes, que será explicada adiante) da qual A faz parte (no caso, de C fazer parte de *Answers*)*.

Caso esteja se perguntando qual a relação com o modelo do gera-e-testa discutido anteriormente: o primeiro *match* ajuda o segundo a gerar o *member* testa se o resultado é válido.

O seguinte programa realiza um teste ao programa anterior:

```
test_analogy(Name, X) :-
  figures(Name, A, B, C),
  answers(Name, Answers),
  analogy(is_to(A,B), is_to(C,X), Answers).

figures(test1,
        [ inside(inside(triangle, triangle),square),
          inside(triangle, square),
          inside(inside(square, circle),square) ]).

answers(test1,
        [ inside(inside(circle, circle), square),
          inside(square, square),
          inside(inside(triangle, circle), square),
          inside(circle, square),
          inside(triangle, square) ]).
```

Código 8: Test Analogy

O goal `test_analogy(test1, X)?` tem o resultado esperado.

Talvez tenha estranhado que os últimos programas estejam todos em inglês. O programa Analogy (um parecido, em espírito, com o usado aqui) foi apresentado como a tese de doutorado de Thomas Evans [2], no MIT. Preferimos manter o nome original (analogy) e com o nome veio o resto.

Alguém poderia dizer que a maior parte da "inteligência" do programa está na representação utilizada. Vale notar, entretanto, que, diferente do programa discutido aqui, o original não tomava figuras geométricas como primitivas e tinha que criar um tipo de representação por conta própria. Como isso foi feito está além do escopo deste texto.

*Caso o mistério te incomode, considere fazer uma visita ao Capítulo 3.

Referências

[1] J.A. Robinson (Jan 1965), "A Machine-Oriented Logic Based on the Resolution Principle", Journal of the ACM, 12 (1): 23–41.

[2] T.G. Evans, "A Program for the Solution of Geometric-Analogy Intelligence Test Questions", Semantic Information Processing , M. Minsky, ed., MIT Press, 1968, pp. 271–351.

2 Teoria de Programação Lógica

Um dos motivos para a programação lógica parecer atrativa são as suas raízes em teorias matemáticas, já bem desenvolvidas e compreendidas. O presente capítulo visa fazer uma rápida introdução a essas raízes. Esta introdução não é de modo nenhum completa e tem por objetivo apresentar apenas uma visão geral. Olhando por alto, existem três aspectos mais importantes nessa teoria, dos quais faremos um breve resumo: o da semântica, o da corretude e o da complexidade de um programa lógico.

2.1 Semântica

Semântica tem a ver com o significado. Em nosso contexto, nos importamos com o significado do programa. No Capítulo 0 começamos uma discussão informal nesse sentido. Como foi lá notado, a definição que demos de "significado" é, na realidade, a de significado procedural. Aqui, vamos trabalhar de uma maneira um pouco diferente.

O significado declarativo é baseado na assim chamada teoria de modelos. Para trabalharmos com ela, precisamos antes de alguma terminologia:

Definição 2.1. *O **universo Herbrand** de um programa lógico P, denotado U(P), é o conjunto de termos base formados pelas constantes e símbolos de funtores que aparecem em P.* **universo Herbrand**

Por exemplo, se temos um código como o seguinte:

```
natural(0).
natural(s(P)) :-  natural(P).
```

Código 9: Natural

então U(P) é o conjunto formado por 0, `natural/1` e suas combinações:

$$U(P) = \{0, s(0), s(s(0)), s(s(s(0))), ...\}$$

Definição 2.2. *A **base Herbrand** de um programa lógico P, denotada B(P), é o conjunto de goals base formados por predicados em P e o termos de U(P).* **base Herbrand**

Chamaremos um subconjunto de B(P) de *interpretação* e um subconjunto consistente com o programa, de *modelo*:

Definição 2.3. *Dada uma interpretação I de um programa lógico, I é um modelo**Modelo** se, para cada cláusula de P,* `a :-` b_0`, ...,` b_n*, $a \in I$ se b_0, ..., $b_n \in I$.* **Modelo**

Podem existir vários modelos diferentes para um programa lógico, então faz sentido falar de um modelo mínimo.

Definição 2.4. *Dado um programa P, um modelo mínimo para P, m(P), é um modelo tal que $\forall M_i$, M_i modelo de P, m(P) $\subseteq M_i$. Tal modelo é chamado de* ***significado declarativo*** *de P.*

significado declarativo

Mostrar que essa semântica e a anteriormente apresentada são equivalentes foge de nosso escopo atual.

2.2 Correção do Programa

Pelo que vimos na seção anterior, todo programa tem um significado bem definido. Apesar disso, esse significado pode ou não corresponder ao intencionado pela programadora. Querer tratar matematicamente o "significado intencionado pela programadora" é querer tirar conclusões rigorosas de algo não rigorosamente definido: vencemos essa dificuldade no Capítulo inicial dizendo que o significado intencionado é um conjunto de goals. Não nos questionamos se isso é mesmo possível, essa questão é deixada de exercício para a leitora diligente.

Supondo que o "significado intencionado pela programadora" é conjunto de goals, definimos, também no Capítulo inicial, o que chamamos de *programa correto* e *programa completo*. Caso não se lembre, um programa é correto em relação a um significado intencionado se o significado do programa está contido no intencionado e, completo, se ele contém o intencionado: um programa é correto e completo se o significado intencionado for igual ao do programa. Geralmente gostaríamos que os programas fossem corretos e completos, mas isso nem sempre é possível de se obter.

À parte do significado, outra questão importante é se ele termina com relação a algum goal: não adianta muito ter um goal no significado intencionado e no do programa se o processo de computação desse goal não termina. Talvez não seja intuitivo que isto é possível com as definições dadas, mas, ao nos lembrarmos que uma busca pode gerar mais de um resultado (e, de fato, pode gerar infinitos resultados), podemos ter uma ideia de que isso depende da forma como o goal é computado. Mas antes de lidarmos com a questão de terminação, precisamos de uma representação melhor do processo de computação de um resultado:

Podemos representar a computação de um goal G = G_0 em relação a um programa P como uma sequência (possivelmente infinita) de triplas (G_i, Q_i, C_i), em que G_i é um goal, Q_i um goal simples ocorrendo em G_i (um goal, no geral, é uma conjunção: um goal simples não) e C_i uma cláusula tal como `A :-` B_1`,` `...,` B_n`.` de P* (possivelmente com uma renomeação de variáveis, para que variáveis diferentes tenham nomes diferentes). G_{i+1} é o goal obtido quando se faz Q_i como o corpo de C_i (lembre-se que Q_i é um goal em G_i) e se aplica o unificador mais geral de Q_i com A, a cabeça de C_i; ou *sucesso* se Q_i for o único goal em G_i e C_i é vazio; ou *falha* se G_i e a cabeça de C_i não são unificáveis.

Se não estiver claro, pare e pense sobre isso por alguns minutos... se continuar escuro, considere ir tomar um suco de laranja e voltar depois

*No futuro revisitaremos esse *framework* teórico, deixando essa notação uma pouco mais leve. Mas vê-la dessa forma antes será importante para compreender melhor a utilidade do que virá depois.

Dizemos que uma computação termina se $\exists n > 0 : G_n = sucesso$ ou $G_n = falha$. Relacionado a isso é o *traço* de uma computação: dizemos que o traço de uma computação (G_i, Q_i, C_i) é a sequência (Q_i, Γ), em que Γ é a parte do unificador mais geral computado no passo i, restrito às variáveis em Q_i.

traço

Dizemos que um programa é *terminante* se todo goal em seu significado pode ser provado em uma quantidade finita de passos. Isto é, se $G \in M(P) \Rightarrow \exists n \in N.G_n \in \{sucesso, falha\}$, em que M(P) é o significado de P.

Para obtermos um exemplo concreto, é útil termos o passo a passo do processo de resolução de um goal. Por exemplo, por exemplo, considere o goal `porta_and(En1, En2, Sai)?` submetida ao Código 1, do Capítulo 1.

O processo de resolução, baseado no algoritmo visto no Capítulo 1, é como se segue:

- Construímos a equação `porta_and(En1, En2, Sai) = porta_and(En1, En2, Saida)`, que é posta na pilha;
- $\Gamma = \{\}$;
    1. É realizado o *pop* da equação `porta_and(En1, En2, Sai) = porta_and(En1, En2,Saida)`;
    2. (Passos b.2 e b.3, três vezes sucessivamente) Γ = {Entrada1 = En1, Entrada2 = En2, Saida = Sai};
    3. Construímos as seguintes equações e as colocamos na pilha:
        - `porta_nand(En1, En2, X) = porta_nand(Entrada10, Entrada20, Saida0)`* e:
        - `inversor(X, Sai) = inversor(Entrada00, Saida00)`;
    4. (a) Realizamos um *pop*, retirando a equação `porta_nand(En1, En2, X) = porta_nand(Entrada10, Entrada20, Saida0)` da pilha;
        (b) Γ += {X = Saida0, Entrada10 = En1, Entrada20 = En2}† ‡;
        (c) Construímos as seguintes equações e as colocamos na pilha:
            - `transistor(En1, X0, X) = transistor(n3,n4,n2)`;
            - `transistor(En2, ground, X0) = transistor(n5,ground,n4)`;
            - `resistor(energia, X) = resistor(energia,n1)`
        (d) i. Realizamos um *pop* da equação `transistor(En1, X0, X) = transistor(n2, ground, n1)`;
            ii. Γ += {En1 = n3, X0 = n4, X = n2};

*Os dois lados da equação, pertencendo a cláusulas diferentes, fazem uso de variáveis, a priori, diferentes. Assim, para evitar problemas, renomeamo-las.

†Por conveniência, usaremos C += B, onde C e B são dois conjuntos, para denotar que C' = C ∪ B e posterior renomeação de C' para C.

‡Não se esqueça que quando adicionamos uma substituição, também a fazemos na pilha e em Γ.

iii. Realizamos um *pop* da equação `transistor(En20, ground, X0) = transistor(n5, ground, n4)`;
iv. Γ += {En20 = n5, X0 = n4};

5. (a) Realizamos um *pop* da equação `inversor(n2, Sai) = inversor(Entrada00, Saida00)`;
   (b) Γ += {Entrada00 = n2, Saida00 = Sai};
   (c) Construímos as seguintes equações e as colocamos na pilha:
      - `transistor(n2, ground, Sai) = transistor(n2,ground,n1)`;
      - `resistor(energia, Sai) = resistor(energia,n1)`;
   (d) Realizamos um *pop* da equação `transistor(n2, ground, Sai) = transistor(n2,ground,n1)`;
   (e) Γ += {Sai = n1};
   (f) Realizamos um *pop* da equação `resistor(energia, n1) = resistor(energia,n1)`;

- Por fim, se tivermos feito todos os passos corretamente, temos que En1 = n3, En2 = n5 e Sai = n1.

Com base nisso, qual seria o traço dessa computação? O programa é terminante?

Vimos anteriormente (no Capítulo 1, o passo b.5 do algoritmo de unificação) explicitamente uma checagem para evitar a não-terminação de um programa em um caso específico*. Mesmo essa checagem não seria o suficiente para evitar a não-terminação de um programa. Em particular, em programas recursivos (muito comuns em programação lógica) é fácil criar um programa para o qual existam goals cuja busca não termina no sentido usual (isto é, no sentido de o programa não terminar a computação de um goal posto pela programadora). Mas o sentido definido acima é ainda mais fraco: por exemplo, na prática, o goal `natural(X)?` pode terminar com a substituição $\iota = \{X = 0\}$, por exemplo, e a computação terminaria. Mas outra computação poderia levar a outro ι, por exemplo $\iota = \{ X = s(0)\}$. Mais no geral, como é possível construir um traço não terminável para uma computação nesse goal, esse programa não é terminante.

Na prática, o tipo de caso de não-terminação mencionado acima seria provavelmente evitado, mas, ainda assim, é importante: primeiro porque mostra que, se um programa for não terminante, existirão resultados (unificações) corretas, mas, na prática, inatingíveis (quais seriam atingíveis ou não, nesse contexto, depende de como se faz a busca pela solução) e, segundo, porque ele nos mostra que a forma de computar o goal pode ser fundamental. Ademais, o caso acima poderia ser evitado porque é simples, mas em programas mais complicados esse tipo de situação pode se tornar um problema real.

*Apesar de em implementações práticas essa checagem ser frequentemente omitida por questões de eficiência, podendo ser ativada manualmente.

É um fato clássico[†] que não pode existir um algoritmo que diga se um programa qualquer termina ou não. Felizmente, não só nossa definição de terminação é especial, mas nossos programas também serão especiais e, eventualmente, poderemos dizer se ele termina ou não e em quais circunstâncias.

Dizemos que um termo A é uma *instância* de um termo B se existe uma substituição ι tal que Aι = B. Com isso, temos as seguintes definições:

Definição 2.5. *Um **domínio** D é um conjunto de goals fechados pela relação de instância: se $A \subseteq D$ e $B = A\iota$ para alguma substituição ι, então $B \subseteq D$.* **domínio**

Definição 2.6. *Um **domínio de terminação** D de um programa P é um domínio tal que qualquer computação de qualquer goal em D termina em P.* **domínio de terminação**

Por exemplo, a Base Herbrand para o programa 9 é um domínio de terminação. No geral, gostaríamos que um programa tivesse um domínio de terminação contido no seu significado intencionado. Para um programa lógico interessante no geral isso não poderá ser obtido. Felizmente, as linguagens de programação com que lidaremos são restritivas o suficiente para que possamos obter esse tipo de resultado no futuro.

Pode ser útil buscarmos achar, para programas lógicos, domínios de terminação. Para isso, usaremos o conceito de *tipo*: um tipo é um conjunto de termos.

Entenda isso como uma definição mais informal. Poderíamos, pela definição, tratar o `match/2`, introduzido no programa Analogy, do Capítulo anterior, como um tipo, mas não temos, atualmente, motivos para fazer isso. Analogamente, podemos chamar arvore_b, no programa Árvore Binária do Capítulo 0 de um tipo,

Definição 2.7. *Um tipo é **completo** se é fechado pela relação de instância.* **completo**

Assim, um (número) natural completo (vide programa 9 deste Capítulo) é ou 0 ou um $s^n(0)$, para $n \in N$.

2.3 Complexidade

Programas lógicos no geral podem ser usados de várias formas diferentes, o que pode mudar a natureza de sua complexidade. Para analisar a complexidade de um programa de modo mais geral, tomaremos goals em seu significado e veremos como eles são derivados.

Para isso, precisaremos do conceito de *comprimento de uma prova*. Quando submetemos um goal a um programa P, o processo de tentativa de goal define implicitamente uma árvore. Se, para cada termo do goal, há apenas uma

[†]Por clássico, entenda "comumente visto em classe", para uma classe comum de um curso apropriado.

cláusula no programa que o prova, dizemos que tal computação é *determinística*. A árvore determinada por uma computação determinística é essencialmente uma lista.

Interpretadores abstratos diferentes podem construir diferentes árvores de busca, o que depende de quais cláusulas são escolhidas primeiro para a prova do goal. Por exemplo, um interpretador pode fazer uma busca por largura: logo depois de realizada a criação do ponto de escolha, o interpretador volta e tenta outra unificação com a consequente criação de outro ponto de escolha, continuando assim até que não haja mais possibilidades nesse "nível", na ocasião de que ele volta ou primeiro ponto criado e continua nesse "segundo nível".

O comprimento de uma prova de um goal é definido como a altura dessa árvore.

Definição 2.8. *O* ***tamanho de um termo*** *é o número de símbolos em sua representação textual. Constantes e variáveis de um símbolo tem tamanho 1. O tamanho de termos compostos é um mais a soma dos tamanhos de seus argumentos.*

tamanho de um termo

Definição 2.9. *Um programa P tem* ***complexidade por comprimento*** *C(n) se qualquer goal de tamanho n no significado de P tem alguma prova de comprimento menor ou igual a C(n).*

complexidade por comprimento

2.4 Negação

Existem duas interpretações fundamentais para um programa lógico:

1. Que a falha de uma busca indica que o goal correspondente não é provável pelo programa;
2. Ou, que a falha de uma busca indica que o goal correspondente é falso.

À segunda interpretação corresponde a assim chamada *hipótese de mundo fechado*: tudo o que é conhecível é conhecido. Nesse contexto, dizer que uma computação falha implica dizer que "se não está no programa, não é verdade". Nessa linha, se quiséssemos implementar algo como uma negação lógica, a qual denotaremos `not` (no lugar de "não", porque em programas de verdade é usado "not", não "não"), o caminho mais natural é dizer que, dado um goal G, `not G?` tem sucesso se G falha.

Not

Vale notar que essa forma de negação tem várias diferenças com a negação da lógica clássica. Por exemplo, a negação da lógica clássica é uma relação monótona*: na lógica clássica, com mais a adição de proposições permite a

*Relações monótonas são as que preservam ordem (isto é, se R é uma relação entre A e B, $\preceq$ a ordem em A e $\preceq$' a ordem em B, temos: $a \preceq b \Rightarrow Ra \preceq' Rb$) e, quando não dito o contrário, é assumido que a ordem é a induzida pela inclusão: $A \preceq B \Leftrightarrow A \subseteq B$.

derivação de, pelo menos, a mesma quantidade de conclusões (se $b \Rightarrow a$, então, para toda proposição c, $b \wedge c \Rightarrow a$), o que segue não é verdade com o nosso `not`.

Assim, por exemplo, podemos escrever algo como `a :- not a.` que indica que `a` tem sucesso se `a` tem falha. Na prática, se existem outras cláusulas contribuindo para a definição de `a`, o resultado dessa computação vai depender muito da ordem de avaliação do interpretador, que no geral não é não-determinística.

Vale notar que essa não é a única forma de negação. Uma outra possibilidade seria a de possibilitar a `fail/1`, um átomo que representa falha, compor a cabeça de uma cláusula, no que poderíamos fazer algo como `fail :-` a_1`, ...,` a_n`.`. Esse tipo de negação tem sua utilidade, por exemplo, em sistemas de diagnóstico de falhas (em particular, porque, por esse sistema, é possível saber o que causou a falha, o que seria mais difícil de outra forma). Mas não faremos uso dele e, quando falarmos sobre negação, nos referimos à negação por falha.

3 Listas

Antes de prosseguirmos em outros aspectos da programação lógica, ocasionalmente será útil sabermos trabalhar com **listas**: **listas**

Definição 3.1. *Usaremos a seguinte definição formal de lista:*

- `.()` *(o "funtor .") é uma lista** *(o que chamamos "lista vazia", é o mesmo que "[]" das seguintes convenções);*
- `.(A, B)` *é uma lista se B é uma lista*

Como é chato ficar escrevendo coisas como `.(A, .(B, []))` *usaremos as seguinte convenções:*

- `[A,B]` *é o mesmo que* `.(A, .(B, .()))`*;*
- `[A]` *é o mesmo que* `.(A, .())` *(o funtor . é de aridade 2, a não ser quando não recebe argumentos);*
- `[A, B, C, ...]` *é o mesmo que* `.(A, .(B, .(C, ...)))`*;*
- `[A|B]` *indica que A é o primeiro elemento da lista (também chamado de* ***cabeça da lista****) e B é o resto da lista, também chamado de* ***corpo da lista*** *.* **cabeça da lista** **corpo da lista**

Dado isso, podemos escrever um programa para adicionar um elemento na lista como o seguinte:

```
append([], A, A).
append([X|Y], A, [X|C]) :- append(Y, A, C).
```

Código 10: Append

Esse é um programa clássico e, por isso, escolhemos manter seu nome clássico, que será usado sem itálico. Os dois elementos iniciais de append são listas e o final é o resultado de se "juntar as duas listas": cada elemento da primeira lista está, ordenadamente, antes da cada elemento da segunda. Para exercitar, pense em qual seria o resultado do goal `append([cafe, queijo], [goiabada], L)?`.

Para uma melhor compreensão, será instrutivo analisarmos o seguinte programa:

*Na verdade, isso não é estritamente padrão e implementações diferentes podem usar funtores diferentes. Essa é outra razão para não usarmos essa definição na prática, mas sim a convenção seguinte. Apesar disso, é importante se lembrar que a lista não é, estritamente falando, diferente de um funtor.

```
% length(Xs, N) :-
%   N eh a quantidade de elementos na lista Xs
%

lenght([X|Xs], N) :-
  length(Xs, K),
  N is 1 + K.
length([], 0).
```

Código 11: Length 0

3.0.1 Operadores aritméticos

Mas antes, precisamos entender o que significa `N is M + 1.`. Esse **is** é o operador de atribuição aritmético e ele não funciona como os demais predicados: para algo como `A is B`, se B for uma constante numérica e A é uma variável, o comportamento é o esperado (A assume o valor de B); se *A* for uma constante numérica e *B* for outra constantes, ocorre falha. Em outras ocasiões, ocorre erro. Disso segue que `is/2` não é simétrico: por exemplo, `A is 5` resulta em A = 5, mas `5 is A.`, em que A é uma variável não instanciada, resulta em erro. Ademais, quando algum operador aritmético* *op* é usado com `is/2`, o operador realiza a operação esperada: `A is 2 + 3` e `A is 10 - 8` resultam em A = 5, por exemplo. Vale notar que esse comportamento é diferente do `=/2`, por exemplo. Assim, o seguinte programa pode não ter o resultado intuitivamente esperado:

is

```
length([X|Xs], N) :-
  length(Xs, K),
  N = 1 + K.
length([], 0).
```

Código 12: Length 1

Intuitivamente, esperaríamos que o *N* fosse unificado, por um processo recursivo, com o número correspondente ao tamanho da lista. Isso não ocorre, porque `=/2` tem um efeito puramente simbólico e não realiza operações aritméticas: o que teríamos em *N* seria algo como uma *string* de símbolos *1+(1+(1+0))*, ao invés da avaliação dessa *string*, ou seja, 3.

Outro exemplo é o do operador ou exclusivo (o *xor*), que também não age da maneira como estamos acostumados: gostaríamos que `1 is A xor 0` seja equivalente a `A is 1`. Veremos mais tarde outras formas de contornarmos isso. Por ora, podemos lidar com isso por meio de predicados de meta-programação

*Operadores aritméticos que temos à disposição são: `+/2`, `-/2`, `*/2`, `//2`, `∧/2`, `-/1`, `abs/1`, `sin/1`, `cos/1`, `max/2`, `sqrt/1`, `<</1`, `>>/1`. O funcionamento de muitos deles é claro pelo símbolo, o dos demais será explicado na medida que forem usados.

(os quais serão melhor explicados no Capítulo 6):

```
% ou_exclusivo(X, Y, Z) :-
%   se ao menos dois dos argumentos nao sao variaveis,
%   o resultado eh o esperado
%

ou_exclusivo(X, Y, Z) :- nonvar(X), nonvar(Y), Z is X xor Y.
ou_exclusivo(X, Y, Z) :- nonvar(Z), nonvar(Y), X is Z xor Y.
ou_exclusivo(X, Y, Z) :- nonvar(X), nonvar(Z), Y is Z xor X.
```

Código 13: XOR

Temos à nossa disposição também *operadores de comparação* (também chamados de operadores relacionais), que funcionam de maneira semelhante e também possibilitam o uso de operadores aritméticos à direita do símbolo, na maneira usual. Eles serão de alguma importância:

- `=:=/2`, de igualdade;
- `=/=/2`, de desigualdade;
- `>/2` , de "maior que";
- `>=/2`, de "maior ou igual que";
- `</2` , de "menor que";
- `=</2`, de "menor ou igual que";

Voltando ao código 11. O goal `length(Xs, N)?`, onde *Xs* é uma lista e *N* uma variável, resulta em *N* tomando o valor da quantidade de elementos de *Xs* (o seu "tamanho"). Mas, o goal `length(Xs, N)?`, onde N é um número natural positivo e Xs uma variável resulta em erro ao chegar no trecho `N is K + 1`, uma vez que, como dito anteriormente, `N is K`, onde *N* é um número e *K* não, resulta em erro.

Para o seguinte programa, esse já não é o caso.

```
length([X|Xs], N) :-
  N > 0,
  K is N -1,
  length(Xs, K).
length([], 0).
```

Código 14: Length 2

O goal `length(Xs, N)?`, para esse programa, resulta em erro se N não for um número. Entretanto, se for um natural positivo, `length(Xs, N)?` resulta

em sucesso e Xs se torna uma lista de N elementos. Se Xs for uma lista e N um natural positivo, `length(Xs, N)?` resulta em falha se Xs tem uma quantidade de elementos diferente de N e sucesso se Xs tem uma quantidade de elementos igual a N.

Nota-se que, apesar dessa diferença procedural, a leitura declarativa do programa é essencialmente a mesma. A diferença decorre da maneira como os operadores aritméticos funcionam em Prolog e leva a outras situações parecidas, o que eventualmente se tornará um incoveniente grande demais. Veremos como lidar com esse tipo de incoveniente de maneiras diferentes no Capítulo de inspeção de estruturas colocar "(Capítulo x)". e, depois, no de restrições lógicas. quiser procurar o capitulo.

3.0.2 Flattening

Uma característica importante da lista, que lhe dá a flexibilidade necessária para poder representar muitos tipos de dados diferentes é que ela é "fechada sob a relação de instanciação", isto é, não existe problema em fazer uma lista de listas. Assim, uma lista perfeitamente válida é `[[[a,b],c],[]]`. Algumas vezes, entretanto, será útil assumirmos que uma lista L só contenha "não-listas" como elementos. Para tanto, podemos fazer uso do funtor `flatten/1`, que pode ser implementado como se segue:

```
flatten([X|Xs], Ys) :-
  flatten(X, Ys1),
  flatten(Xs, Ys2),
  append(Ys1, Ys2, Ys).
flatten(X, X) :-
  atomic(X),
  X \= [].
flatten([], []).
```

Código 15: Flatten

O `atomic/1` usado é avaliado como sucesso se seu argumento for uma constante (`atomic(A)?` resulta em sucesso se A for um funtor de aridade zero).

3.0.3 Lista Completa

Agora, voltando rapidamente a um assunto discutido no Capítulo passado, temos a definição de **lista completa**: **lista completa**

Definição 3.2. *Uma lista L é completa se toda instância $L\iota$ satisfaz a definição de lista dada. Se existem instâncias que não a satisfazem, ela é dita incompleta.*

Por exemplo, a lista `[a,b,c]` (menos popularmente conhecida como `.(a,.(b,.(c,[]))))` é completa: a `[a,b|Xs]` (menos popularmente conhecida como `.(a,`

`.(b,Xs))`), não. Isso porque *Xs* não tem, a princípio, obrigação de ser uma lista. Listas incompletas são chamadas assim porque elas podem "ser completadas" para se tornarem uma lista (ou não, veja que algo como `[a|b]` pode parecer uma lista pela forma como foi escrito, mas, pela definição acima, não o é).

3.1 Listas de diferença

Estruturas de dados incompletas, no geral, podem ser bem importantes e úteis. Um exemplo interessante são as **listas de diferença**, uma estrutura de dados que pode simplificar e aumentar a eficiência de programas que lidam com listas.

listas de diferença

Listas de diferença têm esse nome porque são criadas como a diferença de duas listas. Por exemplo, dizemos que a diferença entre as listas `[a,b,c]` e `[c]` é a lista `[a,b]`. A diferença entre duas listas incompletas `[a,b|Xs]` e `Xs` é equivalente à lista `[a,b]` e, mais no geral, a diferença entre duas listas incompletas `[`x_0`, ...,`x_i`|Xs]` e `Xs` é equivalente a `[`x_0`, ...,`x_i`]`. A diferença entra as listas *Ys* e *Xs*, com `Ys = [Zs|Xs]`, é denotada `Ys\Xs`, em que *Ys* é dita a *cabeça* e *Xs* a *cauda*. Na prática, poderíamos definir um funtor tal como `lista_diff/2`, o que seria potencialmente mais eficiente, mas a notação anterior será mais conveniente pelo momento. Se eficiência for uma preocupação, termos como `Ys\Xs` poderiam ser substituídos automaticamente por outro funtor apropriado.

É importante notar que qualquer lista *L* pode ser trivialmente representada na forma de lista de diferença como `L\[]`. Fazer a concatenação de uma lista de diferença `Ys\Xs` com uma `Zs\Ws` só é possível quando *Xs* seja unificável com *Zs*, sendo, nessa ocasião, ditas **listas compatíveis** e, nesse caso, resulta na lista de diferença `Zs\Xs`. Esse fato é capturado no seguinte código:

listas compatíveis

```
append_dl(Xs\Zs, Ys\Xs, Ys\Zs).
```

Código 16: Append dl

Perceba que, enquanto no código 10 a concatenação realiza uma quantidade de operações linear no tamanho da lista, no código 16 a concatenação é realizada em uma quantidade constante de operações.

Outro exemplo de programa que poderia ser melhorado com o uso de listas de diferença é o `flatten/2`. Se queremos realizar o *flatten* de uma lista Xs e temos um `flatten_dl/2` que realiza o flatten em uma lista diferença, sabemos que `flatten(Xs,Ys)` é o mesmo que `flatten_dl(Xs\[], Ys\[])`. Um `flatten_dl/2` pode ser como o seguinte:

Perceba agora que o passo em que usamos `append_dl/2`, no código 17, pode ser feito de maneira implícita, resultando no seguinte

que parece melhor do que nosso `flatten/2` original. Essa mudança poderia ser obtida automaticamente com uma aplicação de um *unfolding*. *Unfolding* é

```
flatten_dl([X|Xs], Ys\Zs) :-
  flatten_dl(X, As\Bs), flatten_dl(Xs, Cs\Ds),
  append_dl(As\Bs, Cs\Ds, Ys\Zs).
flatten_dl(X,[X|Xs]\Xs) :-
  atomic(X),
  X \= [].
flatten_dl([],Xs\Xs).
```

Código 17: Flatten dl 0

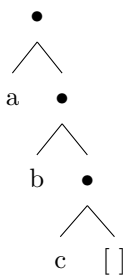
```
flatten_dl([X|Xs], Ys\Zs) :-
  flatten_dl(X, Ys\Bs), flatten_dl(Xs, Bs\Zs).
flatten_dl(X,[X|Xs]\Xs) :-
  atomic(X),
  X \= [].
flatten_dl([],Xs\Xs).
```

Código 18: Flatten dl 1

um tipo de "transformação programática" que consiste, em termos gerais, na substituição de um goal por sua definição e é o contrário de *folding*, que consiste na substituição do corpo de uma cláusula por sua cabeça. Essas transformações são úteis na otimização de código e para outras coisas, que fogem do nosso escopo atual.

Vistos os exemplos de listas de diferença dados, é justo dizer que a ideia de estruturas de diferença parecem boas e nos perguntar se ela não é generalizável para tipos de dados diferentes de listas. Para tanto, precisamos desenvolver uma representação um pouco melhor de o que a lista é e como ela funciona. Uma lista, como a definimos acima, é uma forma de representar uma árvore. Por exemplo, a lista `[a,b,c]` se parece com:

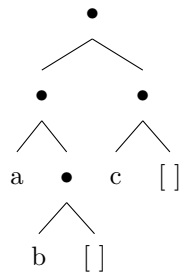
Árvore 2: Lista simples



em que • representa o funtor `./2` da lista. Na verdade, funtores em geral são árvores, não só os de lista, mas funtores de lista tem essa "cara especial". Uma

lista como `[[a,b],c]` seria como:

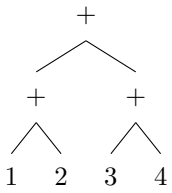
Árvore 3: Lista aninhada



O que `flatten/2` faz é uma transformação em árvores como essa, transformando uma lista aninhada como a da Figura 3 em uma simples, como a da Figura 2. No caso, o resultado de flatten na lista da Figura 3 (que é uma árvore) seria a lista da Figura 2.

Agora, para vermos como uma estrutura de diferença pode ser útil em outras ocasiões, considere o seguinte exemplo. Em Prolog, a operação de soma é associativa à esquerda, o que significa que a operação `1 + 1 + 1 + 1` é tomada como `(((1 + 1) + 1) + 1)`. Por razões técnicas, podemos querer que ela seja normalizada como associativa à direita. Ou seja, se temos algo como `(1 + 2) + (3 + 4)`, dado pela árvore

Árvore 4: Soma



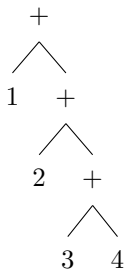
queremos que isso se torne `(1 + (2 + (3 + 4)))`, dado pela árvore da Figura 5.

O que precisamos é de uma forma de normalizar essa operação. Para tanto, precisamos de um novo funtor (já que o "+" já está em uso). Definiremos o funtor `++/2` como um operador infixo, da seguinte forma:

```
:- op(500, xfy, ++).
```

Resumidamente, essa linha nos diz que o funtor `++/2` é um operador binário (podemos usá-lo na forma `A ++ B`), de prioridade 500 (quanto maior o número,

Árvore 5: Soma normalizada



menor a prioridade de avaliação, sendo a menor prioridade possível dada por 1200) e associativo à direita (yfx seria associativo à esquerda e xfx seria não-associativo).

Com esse funtor em mãos, definimos a "soma de diferença" de maneira análoga à lista de diferença, isto é, como `S1++S2`, em que *S1* e *S2* são somas normalizadas incompletas. Nesse contexto, o número 0 faz o papel da lista vazia e `S1++0` é equivalente a *S1*. Assim, podemos definir o seguinte código:

```
normalize(Exp, Norm) :- normalize_ds(Exp, Norm++0).

normalize_ds(A+B, Norm++Space) :-
  normalize_ds(A, Norm++NormB),
  normalize_ds(B, NormB++Space).

normalize_ds(A, (A+Space)++Space) :-
  atomic(A).
```

Código 19: Soma normalizada

O goal `Normalize(Exp, Norm)` tem sucesso se *Norm* é a versão normalizada da expressão *Exp*. Perceba a semelhança entre esse normalizador e o `flatten/2`: a transformação feita na árvore é essencialmente a mesma. De uma expressão `A+B`, é como se tivéssemos normalizado A, normalizado B e, então, concatenado o resultado (como seria uma operação de concatenação de "somas de diferença"?).

Fica como exercício a seguinte questão: qual seria o comportamento esperado nas operações usuais de listas de diferença `Xs\Zs` quando $Xs \subset Zs$ (isto é, quando os elementos de *Xs* pertencem a *Zs* mas alguns de *Zs* podem não pertencer a *Xs*)?

Leituras Adicionais

A pesquisa e prática em programação lógica frequentemente é muito próxima à de linguagens funcionais e esse é o caso na área de transformações de programas. As transformações de *folding/unfolding* foram primeiro adaptadas para a programação lógica por Tamaki e Sato [1]. Mais informações podem ser encontradas em [2].

[1] Tamaki, H. and Sato, T., "Unfold/Fold Transformations of Logic Programs", Proc. Second International Conference on Logic Programming, pp. 127-138, Uppsala, Sweden, 1984.

[2] Roychoudhury, A. and Kumar Narayan K. and Ramakrishnan C.R. and Ramakrishnan I.V., "Beyond Tamaki-Sato Style Unfold/Fold Transformations for Normal Logic Programs", International Journal of Foundations of Computer Science, World Scientific Publishing Company

4 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*. 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 à eficiência do programa: quanto mais *choice points*, em geral, mais ineficiente o programa é†. 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 (assim, 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 poderia ser 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. Comumente, um programa com menos axiomas resulta em melhor legibilidade e maior eficiência, dada a menor quantidade de *choice points*. 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 a falha causada pelos axiomas a mais, não serão mais) e de unificações desnecessárias, aumentando a eficiência do programa. Vemos então que esse não é um assunto tão simples, e merece a atenção da programadora.

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

corte

*No sentido de que o programa pode não saber qual é "a escolha que daria certo". Vale notar que, em outros contextos, não-determinismo indica apenas a presença de escolhas, e que a programação lógica pura não indica como lidar com elas.

†Principalmente devido ao *overhead* que lembrar das escolhas que poderia ter feito causa

de uma computação lógica como a busca em uma árvore, mas, enquanto isso, nossa interpretação intuitiva será o suficiente.

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

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

Código 20: Member 0

O símbolo "_" (um *underline*) 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, um *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*. 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 21: 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 à 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 do programa. 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

*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 ou compilador "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

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.

Apesar disso, se deixarmos de lado o preconceito (que alguns têm com coisas "impuras"), 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 22: 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 23: SES

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

Código 24: OR

Perceba que, diferentemente do que ocorre em programas puramente lógicos*, 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 em Prolog†.

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

*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.

†Note que isso é dependente da implementação. Você pode vir a usar ferramentas relacionadas a programação lógica em que esse não é o caso.

‡Poderíamos pensar como o processo de *backtracking* como um nosso "ou" natural, mas às vezes pode ser conveniente usar um "ou" em uma cláusula, seja por efeito de legibilidade ou mesmo para evitar o *backtracking*.

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

4.1 Outras Opções

Covêm notar que a perspectiva anteriormente apresentada (usando *choice points* e *backtrackings*), usada no Prolog padrão, não é a única possível. Que 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, deveriam ser imateriais (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* como os aqui apresentadas.

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* 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].

*Convém termos uma ideia da diferença entre "implicação" e "inferência": "A implica B" não implica que "de A se infere B", mas, ter uma prova de "A implica B", sim.

Leituras Adicionais

A pesquisa e prática em programação lógica frequentemente é muito próxima à de linguagens funcionais e esse é o caso na área de transformações de programas. As transformações de *folding/unfolding* foram primeiro adaptadas para a programação lógica por Tamaki e Sato [1]. Mais informações podem ser encontradas em [2].

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

5 Programas como árvores

Será conveniente termos uma interpretação mais visual de programas lógicos. Isso nos dará uma ideia melhor de em qual ordem ocorrem as tentativas de unificações e como lidar com elas. Considere o seguinte programa, representando um pedaço da árvore genealógica da dinastia Julio-Claudiana, junto com a relação ancestral*

```
filhx(julia_caesaris, augustus).
filhx(julia_caesaris, scribonia).

filhx(agrippina, marcus_vipsanius).
filhx(agrippina, julia_caesaris).

filhx(caius_caesar, agrippina).          % Calígula
filhx(caius_caesar, germanicus).

filhx(julia_drusilla, caius_caesar).
filhx(julia_drusilla, caesonia).

filhx(nero, agrippina).                  % Nero
filhx(nero, gnaeus_ahenobarbus).

ancestral(A, B) := filhx(B, A), !.
ancestral(A, B) := filhx(B, C), ancestral(A,C).
```

Código 25: Ancestral 0

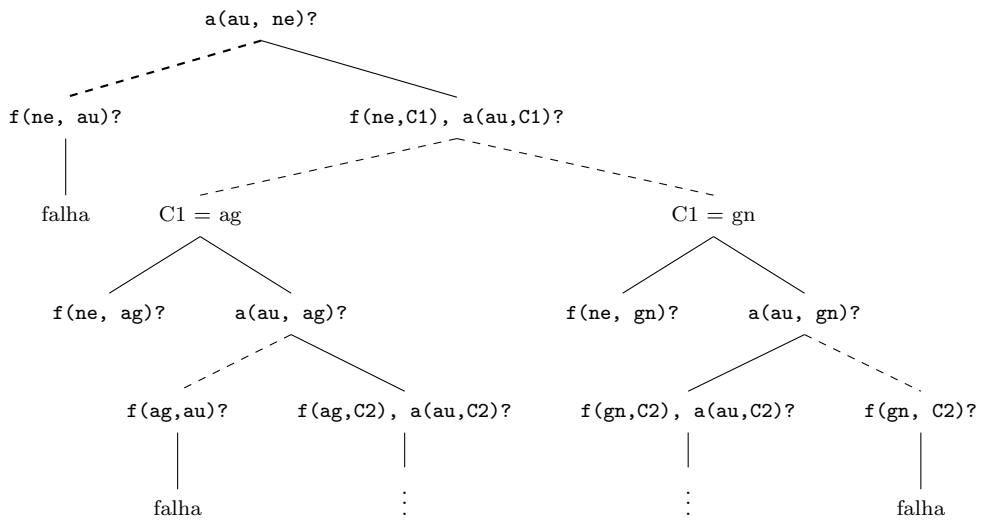
Por conveniência, no lugar das relações filhx/2 e ancestral/2 e dos nomes mostrados acima, usaremos os mostrados no Código 26 .

*Os romanos, assim como outros povos, tinham o hábito de atribuir outro nome às pessoas, com base em características físicas, psicológicas, de caráter ou de hábito. Assim, por exemplo, Claudius significa "manco", Calígula significa "bota de (pequeno) soldado", Augustus significa "majéstico" ou "venerável" e, Sulla, significa "manchado".

```
f(ju, au).
f(ju, sc).

f(ag, ma).
f(ag, ju).

f(ca, ag).
f(ca, ge).

f(ju_d, ca).
f(ju_d, cae).

f(ne, ag).
f(ne, gn).

a(A, B) := f(B, A), !.
a(A, B) := f(B, C), a(A,C).
```

Código 26: Ancestral 1

O goal `a(au, ne)?` define implicitamente uma árvore, que começa da forma a seguir.



A árvore inteira não será colocada por questões de espaço, mas continua de maneira semelhante. Os ramos tracejados indicam um "ou" lógico, enquanto que os sólidos indicam um "e" lógico. O que isso significa é que falha em um dos ramos dos arcos sólidos indica que todo o arco é falho, mas uma falha em

um ramo tracejado só afeta o ramo tracejado.

A execução de um programa lógico consiste em uma busca, em uma árvore como essa, por uma folha de "sucesso". Se, ao invés de "sucesso", é encontrada "falha", é feito o *backtracking*. Quando alguém fala sobre um "modelo computacional de programa lógico", fala essencialmente sobre como percorrer essa árvore e como fazer esse *backtracking*. Como pode ver, ela pode crescer muito em largura a cada nível (não cresceu tanto no exemplo acima porque colocamos uma quantidade reduzida de substituições). Por isso, é compreensível que preferamos uma busca em profundidade do que em largura. Isso pode, é claro, levar a problemas técnicos e filosóficos, já que impõe uma escolha arbitrária sobre a ordem de avaliação das cláusulas e dos termos de cada cláusula, o que pode levar a comportamentos inesperados à programadora desatenta a esse modelo*. É claro, dizer apenas que a busca é em profundidade não é o suficiente: precisamos dizer que ela é em profundidade e à esquerda.

O seguinte exemplo, um homem reclamando de sua vida, mostra que questões de parentesco podem ser mais complicadas do que isso (o exemplo é baseado na história retirada de [1], frequentemente atribuída a Mark Twain):

Me casei com uma viúva, que tinha uma filha crescida. Meu pai, que nos visitava frequentemente, se apaixonou com a filha. e a tomou como sua esposa. Isso fez do meu pai o meu filho adotado, e, de minha filha adotada, minha madrasta.
Depois de um ano, minha esposa deu à luz um filho, que se tornou o irmão adotado do meu pai e, ao mesmo tempo, meu tio, já que ele era o irmão de minha madrasta.
Mas a esposa de meu pai, isto é, minha filha adotada, também deu à luz um filho. Então, ele era meu irmão e também meu neto, já que ele era o filho de minha filha.
Isso quer dizer que eu me casei com minha avó, já que ela era a mãe de minha mãe. Como o marido de minha esposa, eu também era o neto adotado dela.
Nossos amigos dizem que eu sou meu próprio avô. Isso é verdade?

O personagem dessa história, não sabendo Prolog, teve uma certa dificuldade em responder a essa questão. Mas nós, como que por reflexo, fazemos o seguinte programa:

*Convém lembrar que não estamos lidando com programação paralela. No modelo de programação paralela a avaliação pode ocorrer de maneira diferente.

```
% pai(?Pai, ?Filhx).
% mae(?Mae, ?Filhx).
% avoh(?Avó, ?Netx).
% avo(?Avô, ?Netx).
% irmao(?Pai, ?Irmao1, ?Irmao2).
% tio(?Pai, ?Tio, ?Sobrinhx).

avo(Avo, Neto):-
         pai(Avo, Avo_filho),
         pai(Avo_filho, Neto).

avoh(Avoh,Neto):-
         mae(Avoh, Filha_da_avoh),
         mae(Filha_da_avoh, Neto).

irmao(Pai, Irmao1, Irmao2):-
       pai(Pai, Irmao1),
       pai(Pai, Irmao2).

tio(Pai, Tio, Sobrinho):-
       irmao(_,Pai,Tio),
       pai(Tio, Sobrinho).

pai(meu_pai, eu).

pai(eu, meu_pai).

pai(eu, filho_meu_e_de_minha_esposa).

pai(meu_pai, filho_de_minha_filha_adotiva).

mae(minha_filha_adotiva, eu).

mae(minha_esposa, minha_filha_adotiva).

% O filho meu e de minha esposa é o irmão adotado de meu pai
% O filho meu e de minha esposa é meu tio
% Filho de minha filha adotiva é meu irmão
% Filho de minha irmã adotiva é meu neto
% Minha esposa é minha avó
% Eu sou meu avô
              irmao(_, meu_pai, filho_meu_e_de_minha_esposa).
              tio(filho_meu_e_de_minha_esposa, meu_pai, eu),
              irmao(_, filho_de_minha_filha_adotiva, eu),
              avo(eu, filho_de_minha_filha_adotiva),
              avoh(minha_esposa, eu),
              avo(eu, eu).
```

Código 27: Família

O último bloco de código na verdade não faz parte do programa, mas foi deixado por conveniência. Ele indica o goal (na verdade, só queremos saber se o pobre coitado é avô de si mesmo, mas os outros termos foram deixados por completeza). Você tem ideia de como é a árvore de busca para esse goal?

Leituras Adicionais

A pesquisa e prática em programação lógica frequentemente é muito próxima à de linguagens funcionais e esse é o caso na área de transformações de programas. As transformações de *folding/unfolding* foram primeiro adaptadas para a programação lógica por Tamaki e Sato [1]. Mais informações podem ser encontradas em [2].

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

[2] Russell, Bertrand (1919), Introduction to Mathematical Philosophy, George Allen and Unwin, London, UK. Reprinted, John G. Slater (intro.), Routledge, London, UK, 1993
Esse livro está datado em alguns pontos, mas permanece interessante. Está gratuitamente disponível (em inglês) em http://people.umass.edu/klement/russell-imp.html

6 Predicados de Inspeção de Estrutura

Predicados de inspeção de estrutura nos passam informações sobre um termo específico. Por exemplo, será um dado termo atômico, numérico, constante, variável? Ou será um funtor composto? Se for, qual será seu funtor principal, qual sua aridade e quais seus argumentos? Os predicados de inspeção de estrutura respondem esse tipo de questão.

6.1 Predicados de tipos

Alguns dos, assim chamados, predicados de tipo são:

- `integer/1`;
- `real/1`;
- `atom/1`;
- `compound/1`;

Cada um deles pode ser interpretado como uma lista infinita de átomos. Por exemplo, `integer/1` pode ser interpretado como:

```
integer(0).  integer(1).  integer(2).  integer(3), ....
```

A partir desses predicados podemos criar outros. Por exemplo, podemos fazer

```
numero(X) :- integer(X);real(X).
```

ou

```
constante(X) :- numero(X); atom(X).
```

6.2 Acesso de termos compostos

Queremos ser capaz, além de lidar com tipos, lidar com funtores. Para acessar o funtor principal, temos o predicado `functor/3`. O goal `functor(Termo, F, Aridade)?` tem sucesso se o funtor principal de *Termo* tem aridade *Aridade* e nome *F*. Assim, por exemplo, `functor(f(`X_1`, ..., `X_n`), f, n)`, onde os X_i são variáveis, tem sucesso, já que o funtor principal é `f/n`.

Pode-se usar esse predicado para, entre outras coisas, realizar a decomposição e criação de termos:

1. `functor(tio(a,b), X, Y)?` tem a solução $\{X = \text{tio},\ Y = 2\}$;
2. `functor(F, tio, 2)` tem a solução $\{F = \text{tio}\}$.

Note que, no item 2 acima, se o goal fosse `functor(F, tio, N)?`, teríamos erro, já que o interpretador não consegue adivinhar a aridade de um funtor a partir do nome. Analogamente, `functor(F, tio(a, b), N)?` resultaria em erro, já que `functor` espera um átomo como segundo argumento, não um funtor composto. Mas, `functor(tio, X, N)?` teria sucesso, com {X = tio, N = 0}.

Similar a `functor/3` é o `arg/3`: `arg(N, F(`X_1`, ..., `X_n`), Q)?` tem sucesso se o Enésimo argumento de F é o Q. Assim como functor, `arg/3` é comumente usado para decomposição e criação de termos:

- Para decompor um termo, `arg/3` acha um argumento particular de um termo composto;
- Para criar um termo, ele instancia um argumento variável de um termo.

Por exemplo, `arg(1, tio(a,b), X)?` tem sucesso com $\{X = \text{a}\}$, enquanto que `arg(1, tio(X,b), a)?` tem sucesso com $\{X = \text{a}\}$.

Exemplos um pouco mais interessantes são os seguintes:

```
% subtermo(Sub, termo) :-
%   Sub eh um termo que aparece em termo
%

subtermo(Termo, Termo).
subtermo(Sub, Termo) :-
  compound(Termo),
  functor(Termo, _, N),
  subtermo(N, Sub, Termo).

subtermo(N, Sub, Termo) :-
  arg(N, Termo, Arg),
  subtermo(Sub, Arg).
subtermo(N, Sub, Termo) :-
  N > 1,
  N1 is N - 1,
  subtermo(N1, Sub, Termo).
```

Código 28: Subtermo

Outro predicado de inspeção de estrutura é o, assim chamado (por razões históricas obscuras), *univ*, escrito como `=../2`*. Um exemplo de seu uso é `Termo =..  [f,a,b]?`, que faz `Termo = f(a, b)`. O *univ* pode ser usado de essencialmente duas formas diferentes:

*Predicados para acesso e construção de termos têm origem na família Prolog de Edimburgo (que tem se tornado o padrão de fato) e alguns dos nomes usados vêm de lá. Em particular, a forma "=.." para *univ* vem do Prolog-10, onde era usado ",.." no lugar de "|" em listas (`[a, b,.. Xs]` no lugar de `[a, b|Xs]`).

```
% substituto(Velho, Novo, Velt, Novt) :-
%   Novt eh o resultado de trocar
%   todas as ocorrencias de Velho no termo Velt por Novo
%

substituto(Ve, No, Ve, No).
substituto(Ve, _, Vet, Vet) :-
  constante(Vet),
  Vet \= Ve.

substituto(Ve, No, Vet, Not) :-
  compound(Vet),
  functor(Vet, T, N),
  functor(Not, T, N),
  substituto(N, Ve, No, Vet, Not).

substituto(N, Ve, No, Vet, Not) :-
  N > 0,
  arg(N, Vet, Arg),
  substituto(Ve, No, Arg, Arg1),
  arg(N, Not, Arg1),
  N1 is N - 1,
  substituto(N1, Ve, No, Vet, Not).
substituto(0, _, _, _, _) :- !.
```

Código 29: Substituto

1. Com um funtor ao lado esquerdo e uma variável no direito: a variável é unificada com `[f|v]`, onde f é o funtor principal e V a lista de seus argumentos;

2. Com uma variável ao lado esquerdo e uma lista no direito: se a lista é `[a,` b_1`, ...,` b_n`]`, a variável é unificada com `a(`b_1`, ...,` b_n`)`.

O seguinte programa mostra um exemplo da utilidade de *univ*:

6.3 Predicados de Meta-programação

Digamos que você queira fazer um interpretador de Prolog em Prolog (algumas possíveis aplicações disso são *debuggers*). O mais simples possível é dado a seguir:

```
solve(Goal) :- Goal.
```

Esse interpretador, sozinho, é de pouca utilidade. Nos dá a possibilidade de acessar quase nada do programa. Se quisermos fazer melhor, precisaremos

```
% map(P, Xs, Ys) :-
%   Ys eh o resultado de se aplicar P a cada elemento de Xs
%   se P for P/n para n > 1, Xs precisa ser uma lista de listas
%
% map/3 usa apply/2 como auxiliar
%
% apply(P, [X1, ..., Xn]) :-
%   P(X1, ..., Xn)? tem sucesso
%

apply(P, Xs) :-
  Goal =.. [P|Xs],
  Goal.

map(_, [], []).
map(P, [X|Xs], [Y|Ys]) :-
  list(X),
  flatten([X|Y], Z),
  apply(P, Z),
  map(P, Xs, Ys).

map(P, [X|Xs], [Y|Ys]) :-
  apply(P, [X, Y]),
  map(P, Xs, Ys).
```

Código 30: Apply

de predicados que nos digam mais sobre predicados (ou seja, meta-predicados, predicados de meta-programação).

Um predicado de meta-programação básico é o `clause/2`. O goal `clause(Head, Body)?` é verdadeiro se *Head* for unificável com a cabeça de uma cláusula e *Body* com seu respectivo corpo (se *Head* for um fato, *Body* é unificável com *true*).

Com isso, podemos fazer um meta-interpretador, algo mais elaborado:

```
solve(true).

solve((A,B)) :-
  solve(A), solve(B).
solve(A) :-
  clause(A,B), solve(B).
```

Código 31: Meta 1

Precisamos dos parênteses a mais em `solve((A,B))` por razões técnicas (não queremos confundir o compilador). Esse interpretador teria que ser estendido se

quisermos que faça tudo o que é esperado de um interpretador Prolog (ele não lida apropriadamente com cortes, por exemplo, ou com entradas pelo teclado). Isso poderia ser corrigido sem grandes dificuldades.

O Prolog nos possibilita não só facilmente escrever interpretadores para a própria linguagem (o que pode ser útil, por exemplo, na construção de *debuggers*, de sistemas especializados, de programas autocorretores, de programas que "se explicam" em termos de porquês e comos entre outros), mas também nos possibilita facilmente escrever interpretadores para outras linguagens e para dialeto dessas. Por exemplo, alguém poderia argumentar que o Prolog, sendo uma linguagem de programação lógica, não lida bem com situações em que a incerteza é uma parte importante. Mas, com ele, podemos facilmente criar nossa própria linguagem para fazer isso. Para tanto, suponhamos, por exemplo, que cláusulas com uma certeza (a probabilidade de estar correta) C sejam representadas pelo funtor `clause_c/3`, em que `clause_c(Head,Body,C)?` é verdade quando *Head* unifica com uma cabeça de cláusula cujo corpo unifica com *Body* e cujo "fator de certeza" unifica com C, e considere o seguinte:

```
solve(true, 1, Limit) :- !.

solve((G1, G2), C, Limit) :-
  !, solve(G1, C1, Limit), solve(G2, C2, Limit),
  C is min(C1, C2).

solve(G, C, Limit) :-
  clause_cf(G, B, C1), C1 > Limit,
  Limit1 is Limit/C1, solve(B, C2, Limit1),
  C is C1 * C2.
```

Código 32: Meta 2

Um goal `solve(Goal,C,Limit)?` submetido a esse interpretador resulta em sucesso se a "confiança" C de *Goal* for maior do que o limite inferior *Limit*. Alguns pontos válidos de se notar nesse programa são que, numa conjunção `A,B`, a nossa confiança na conjunção é tomada como o mínimo entre a de A e a de B e que se, para provar um goal *Goal*, é preciso passar por uma cláusula com fator de certeza *C1*, a prova do corpo da cláusula terá um fator de certeza menor, dado por `Limit/C1`. Isto porque queremos que nossa "confiança" (pensando de forma probabilística) *C1* na cláusula vezes a confiança *C2* (pense na probabilidade de eventos independentes) na resolução do corpo da cláusula seja maior que *Limit*.

6.3.1 Meta-predicados de variáveis

Outro predicado de meta-programação básico é o `var/1`: o goal `var(T)?` tem sucesso se T é uma variável não instanciada e falha caso contrário. Sua irmã, `nonvar/1`, funciona de maneira análoga. Por exemplo, `var(a)?` e `var([X|Xs])?`

falham, enquanto que `var(X)?` tem sucesso, se X é uma variável não instanciada. Esse predicado nos permite fazer, por exemplo, programas como o seguinte:

```
length(Xs, N) :- nonvar(Xs), length1(Xs, N).
length(Xs, N) :- var(Xs), nonvar(N), length2(Xs, N).
```

Código 33: Length mais geral

em que `length1/2` e `length2/2` são dados pelos length no (Capítulo 4).

7 Restrições

Como vimos anteriormente, uma das ideias iniciais da programação lógica era de permitir à programadora expressar *o que* o programa faz, sem ter que se preocupar muito em *como* ele o faz, o que fazemos expressando a lógica do programa em termos de relações. Na maior parte dos paradigmas de programação usuais, há uma dificuldade em expressar essas relações, ou restrições*, entre os objetos definidos no programa.

Com a programação lógica, isso é um pouco aliviado. Por exemplo, se soubermos a gramática do Prolog, não é difícil ler o programa Member (copiado a seguir, para referência) como expressando uma relação que existe entre um termo X e um Xs se Xs puder ser escrito como `[X|Xs]` ou, se for possível escrever Xs como `[Y|Ys]` e X tiver a relação member com Ys.

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

Código 34: Member

Mas vimos também que operações aritméticas, no Prolog, não funcionam de maneira relacional (ao fazermos uma soma, por exemplo, o "+" funciona como uma função, retornando um valor, no lugar de como uma restrição ou relação). Isso é grave, porque expressões aritméticas aparecem de várias formas em problemas reais e não ser capaz de expressá-las relacionalmente poderia levar a um código muito maior, redundante e difícil de manter.

Para se convencer disso, considere, como um exemplo, a equação da lei de Ohm: I = V/R (onde I é a corrente, V a tensão e R a resistência). Claramente, essa equação expressa a relação entre I, V e R, assim como as restrições provenientes dessa relação, indicando que, fixando quaisquer duas das variáveis, a terceira também é fixada e, fixando uma, as duas outras obedecem a uma restrição (por exemplo, se V = 10 volts, temos que $I \times R = 10$). Atualmente, não conseguimos expressar esse tipo de relação em código sem algum esforço.

Vimos uma forma limitada de lidarmos com isso no Capítulo anterior, mas precisaremos de algo mais poderoso se quisermos lidar com problemas mais complexos. Antes disso, será útil nos abstrairmos um pouco disso para vermos a programação por restrições de uma forma mais ampla.

7.1 Domínios

Restrições não se limitam a restrições aritméticas. Existem vários tipos de restrições, cada uma possivelmente agindo em diferentes domínios. O domínio é

*Estarmos tratando relação como sinônimo a restrição é um abuso de linguagem usado aqui por sua conveniência, mas é importante lembrar que são coisas diferentes.

o que determina as formas legítimas de restrição e o que elas significam. Restrições são escritas usando constantes (como 0, ou 1) e símbolos que agem como funções (como "+" ou "-"). O domínio determina a sintaxe das restrições: quais símbolos de restrições podem ser usados, quais e quantos são os argumentos de cada símbolo e a ordem em que são escritos.

Dois exemplos de domínios que conhecemos são o dos Reais e o dos Inteiros, com os símbolos de restrições usuais (isto é: "+", "<", etc.). Outros exemplos de domínios são o das Árvores e o dos Booleanos. São domínios de grande importância e, por isso, discorreremos momentaneamente um pouco sobre eles logo mais. Em se tratando de domínios aritméticos, a não ser quando dito o contrário, assumiremos que lidamos com o domínio dos Reais.

Definido o domínio, dada uma restrição, precisamos saber o que queremos dela. Algumas das opções mais comuns são:

1. Checar se a restrição é satisfazível (isto é, se existe alguma substituição para a qual a restrição é verdadeira);
2. Encontrar uma substituição que respeite as restrições;
3. Otimizar a substituição encontrada por meio de uma função (comumente chamada *função custo* (apesar de muitas vezes não podermos interpretar essa função como algum "custo" de forma natural)).

Claramente, se conseguimos (2), conseguimos (1) e, se conseguimos (2), conseguimos (1) e (2). Frequentemente, conseguir (1) será equivalente a conseguir (2), porque seguimos um método construtivo. Nem sempre, no entanto, conseguiremos (1). Sabemos, por exemplo, que no domínio dos Inteiros existem restrições que não se sabe se podem ser satisfeitas.

Ou talvez possamos, a princípio, dizer se a restrição em um domínio seja satisfazível, mas na prática isso se torne computacionalmente inviável se a restrição for muito complicada ou complexa. O problema de descobrir se uma restrição booleana pode ou não ser satisfeita (mais conhecido como SAT, de *Propositional Satisfiability Testing*), é um famoso problema NP-difícil e ainda hoje um tema de ativa pesquisa (veja, por exemplo, [3]).

Esse tipo de constatação rápida já nos dá uma ideia do tipo de pergunta que precisaríamos fazer dado um domínio, assim como da variedade de tipos de restrição que existem, mesmo em um domínio.

Daqui para frente denotaremos problemas de otimização como COP (de *Constraint Optimization Problem*), de satisfação de restrições como CSP (de *Constraint Satisfaction Problem*) e, quando não for necessário fazer distinção, apenas CP. Até agora, temos lidado com CPs como contendo uma restrição, mas será conveniente lidar com eles como contendo uma conjunção de restrições:

Se x_0 é um símbolo de restrição no domínio, junto de seus argumentos, diremos que é uma **restrição primitiva**. Uma **restrição composta** C é a

restrição primitiva

restrição composta

conjunção de restrições primitivas: x_0, ..., x_n (em que as vírgulas, como usual, são lidas como um e lógico), em que x_i é uma restrição primitiva. Assim, um COP é descrito por uma tupla (C, f), enquanto que um CSP é descrito por algum C, em que C é uma restrição composta. Daqui para frente nos referiremos a uma substituição nas variáveis de um CP que respeite às restrições como uma solução desse CP.

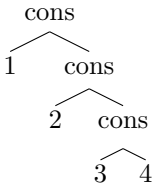
7.2 Árvores

Como você logo perceberá (ou talvez já o tenha percebido), já vimos restrições por árvores antes, elas só estavam um pouco disfarçadas.

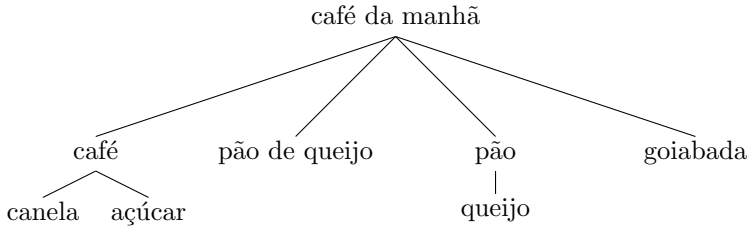
Um **construtor de árvore** é uma sequência de caracteres começando com um caractere minúsculo. Definimos uma árvore recursivamente como: uma constante é uma árvore (de altura 1); um construtor de árvore com um conjunto de n $\geq$ 1 árvores é uma árvore.

construtor de árvore

Árvores são comumente representadas na forma de diagramas como os seguintes:



ou



Que podem ser reescritos, de forma mais compacta como `cons(1, cons(2, cons(3, 4)))`* e `café da manhã(café(canela, açúcar), pão de queijo, pão(queijo), goiabada)`†. Como pode ver, é essencialmente a mesma representação que temos para um funtor (junto com seus argumentos): funtores são árvores, árvores são funtores (neste contexto).

Um algoritmo muito próximo ao algoritmo de unificação, que vimos no Capítulo 1 , serve para resolver restrições em árvore do tipo $T_1 = T_2$, em que

*A leitora de olhos afiados vai notar que é mais ou menos assim que representamos listas, trocando o "cons" pelo ".".

†Estamos usando caracteres especiais aqui para fins de expressividade, mas seu uso em códigos de computador deve ser evitado.

T_1 e T_2 são árvores. É interessante notar que um algoritmo para resolução de restrições em árvores foi dado por Herbrand[4] de forma independente ao do algoritmo de unificação de Robinson.

Vale notar também outro detalhe importante naquele algoritmo: ele realiza um procedimento que frequentemente fazemos ao buscar resolver um problema, isto é, transformando uma pergunta, potencialmente complicada e difícil, em uma trivial, para a qual sabe-se a resposta. Essa é uma instância de um processo de **normalização**, isto é, um processo de transformar uma restrição em outra equivalente, mas mais tratável. Processos de normalização frequentemente exercem um papel importante. Voltaremos a esse tema depois.

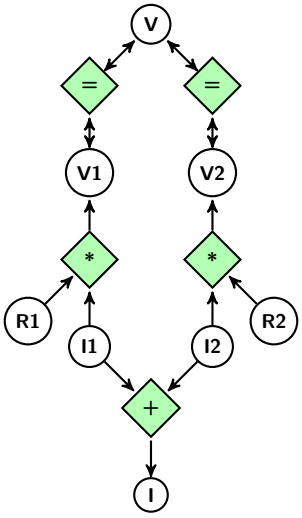
normalização

7.3 Ideias de resolução

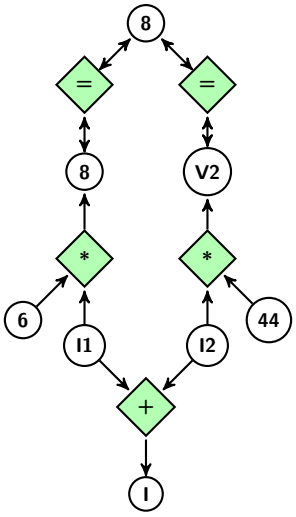
Nossa discussão sobre problemas envolvendo restrições até agora foi uma discussão geral e apesar de muitas vezes, para resolver esses problemas, ser preferível usar métodos específicos ao domínio e ao tipo de problema, muitas vezes esses problemas são heterogêneos o suficiente para não permitir isso ou não fazem uso de uma sintaxe que permita o uso de métodos específicos. É útil então termos alguma ou algumas formas gerais de resolver esses problemas. Eles são baseados em busca.

7.3.1 Propagação local

Propagação local é melhor explicada por um desenho. O grafo a seguir representa a modelagem de um circuito elétrico. As "caixinhas" representam as relações de restrições e as "bolinhas", os dados ou variáveis. Por exemplo, uma "caixinha" com sinal de adição indica que os dados (no caso, as variáveis) que entram por ela (o que é indicado pela seta) somadas devem ser iguais às variáveis, ou dados, de saída (indicado por outra seta).



Esse grafo é uma outra forma de representar a restrição `V=V1, V=V2, V1=I1*R1, V2=I2*R2, I=I1+I2`*. Considere as restrições adicionais de que `V=8, R1=6, R2=44`. Na propagação local, podemos considerar as arestas como veículos das restrições, deixando-as passar de nó em nó de forma apropriada, passo a passo. Por exemplo, dadas as restrições acima, a restrição de `V=8` pode ser propagada para `V1=8`, e teríamos:



Disso, podemos continuar a propagação, obtendo `I1 = 8/6`. E assim vai.

Propagação local funciona muito bem para resolver alguns problemas, mas simplesmente não consegue resolver outros. Por exemplo, considere a restrição

*Exemplo adaptado de [6], pág. 36.

`A=B+2*Z, A=3*K-B, K=5, Z=7`. Propagação local detectaria que `K=5` e `Z=7`, mas não encontraria o valor de A ou B. Isso ocorre porque para tanto seria preciso fazer uso de mais de uma restrição por vez, o que a propagação local não faz.

Mais geralmente, dizemos que um **propagador** é uma função monotônica não-crescente de domínio para domínio, i.e. se D e D' são domínios tais que $D \subseteq D'$ e f é um propagador, obrigatoriamente $f(D') \subseteq f(D)$ e $f(D) \subseteq D$. Um propagador é dito correto em relação a uma restrição `r/n` com variáveis de domínios D_i se as soluções para a restrição nos domínios originais são as mesmas que as soluções para a restrição em $f(D_i)$. Perceba que essa é uma noção fraca, já que a função identidade a satisfaz. Propagação local é um tipo de propagador correto (veremos outros mais para frente). No frigir dos ovos, o que um propagador faz impedir que o resolvedor teste todas as possibilidades descartando algumas possibilidades que não dariam certo.

propagador

No geral, em um CSP, provê-se um propagador para cada restrição. Nesse contexto, um **resolvedor por propagação** para um conjunto de propagadores P aplica os propagadores em P a cada restrição até que não haja mudanças a serem obtidas. Em outras palavras, um resolvedor *solv* por propagação recebe como argumentos um conjunto de propagadores P e um de restrições R e retorna um conjunto de domínios que é o maior ponto fixo dos propagadores em P com respeito à relação de inclusão (assumindo que tal conjunto de domínios exista, o que geralmente é o caso).

resolvedor por propagação

7.4 Busca Top-Down

Propagação local pode ser uma técnica muito útil para resolver alguns problemas, e pode ficar mais poderosa ainda se combinada com uma estratégia de *ramificação*. A essa combinação chamaremos de busca *top-down*.

Intuitivamente, a propagação de restrições ajuda a transformar o problema em outro mais simples. O passo de ramificação serve, então, para partir o problema em problemas menores, aos quais poderemos aplicar a propagação de restrições novamente, seguido por outra ramificação, e assim vai, gerando uma árvore de busca. A forma padrão de busca top-down é chamada de **busca por backtracking**. Essa forma de busca funciona simplesmente gerando a dita árvore e fazendo a travessia dela (frequentemente, faremos a travessia enquanto geramos a árvore, não depois). Vale notar que, na presença de propagação de restrições, ocasionalmente pode ser útil adicionar à escrita do problema restrições implícitas, que podem auxiliar na propagação.

busca por backtracking

A forma mais comum de *backtracking* começa ordenando as variáveis do problema, usualmente por algum modelo heurístico, com as ramificações tomando a forma de divisões no domínio de cada variável. Uma forma de fazer isso é a chamada marcação (ou, mais comumente, *labelling*), que corresponde à ramificação do domínio de cada variável em seus elementos constituintes (o que só pode ser

feito, é claro, quando o domínio for finito), assegurando que todos os valores de cada variável serão explorados. A marcação pode tornar um *método de busca incompleto* em um completo (isto é, um método que pode não encontrar uma solução quando ela existe em um que sempre encontra alguma solução se ela existe).

A ordem de exploração das variáveis é, então, escolhida por meio de uma *heurística de escolha de valor*. Frequentemente não será viável termos um método de busca completo, então é essencial que foquemos a nossa atenção nos valores que parecem mais promissores. Esse "foco de atenção" pode frequentemente ser descrito na forma de uma política de alocação de crédito, fornecendo maior crédito às escolhas aparentemente mais promissoras.

7.5 Branch and Bound and Cut

Os métodos de busca discutidos anteriormente são apropriados para CSPs. Para COPs, precisaremos fazer algumas modificações. Uma opção é o **branch and bound** e *branch, bound and cut*. O *branch and bound* genérico funciona da seguinte forma (sendo F o conjunto de soluções para o problema de otimização inteira):

branch and bound

1. F é particionado em n partes $(F_1, \ldots, F_n)$;
2. Um F_k é selecionado:
   (a) Se ele não for factível, é deletado;
   (b) Caso contrário, compute um limitante inferior $b(F_k)$ de sua função custo;
   (c) Se $b(F_k) \geq U$, o limitante superior do problema global F_k é deletado;
   (d) Caso contrário, obtenha uma solução ótima do problema inicial, ou ramifique F_k em outros subproblemas e os adicione à lista de subproblemas a serem selecionados ou deletados.

Note que, para esse método, é essencial, primeiro, a existência de uma função $b(F)$ razoavelmente eficiente, segundo, um método de ramificação razoavelmente eficiente, assim como um para o conhecimento do limitante superior U. Para um problema geral, podem existir várias escolhas para cada um desses fatores e uma "boa escolha" pode afetar decisivamente a eficiência do algoritmo.

O limitante superior U pode ser inicializado com infinito ou com o custo de alguma solução factível e é atualizado com o custo da melhor solução encontrada até o momento. A escolha de b pode ser um pouco mais complicada e será ilustrada com uma instância de um exemplo de problema de programação inteira linear.

Dado um problema de programação inteira linear tal como

$$\begin{aligned} &\text{minimize } c'x, \\ &\text{tal que } Ax = b, \\ &x \geq 0, \\ &x \in \mathbb{Z}, \end{aligned} \tag{1}$$

uma possível função b é o custo ótimo do problema relaxado

$$\begin{aligned} &\text{minimize } c'x, \\ &\text{tal que } Ax = b, \\ &x \geq 0. \end{aligned} \tag{2}$$

Se uma solução x^* do problema relaxado for inteira (isto é, se todas as componentes de x^* forem inteiras), essa solução é ótima para o problema inteiro inicial. Caso contrário, se algum dos x_j para $j \in J$ forem fracionários, podemos ramificar o problema adicionando as restrições $x_k \leq \lfloor x_k \rfloor$ ou $x_k \geq \lceil x_k \rceil$, para algum $k \in J$. Como x^* não faz parte do conjuntos de soluções de nenhuma das ramificações, o custo ótimo delas será estritamente maior.

Vale notar que, se o problema relaxado original foi resolvido por meios do método simplex, o ramificado poderia ser resolvido sem grandes dificuldades pelo simplex dual, já que só difere do problema original (relaxado) pela adição de uma restrição.

O *branch and bound* para problemas de programação linear inteira poderia ser aprimorado pela adição de *planos de corte*, traduzidos na forma de restrições que diminuem o espaço de busca. O uso de cortes ditos "profundos" podem tornar a execução de *branch and bound* muito mais rápida (mas podem ser difíceis de encontrar). Para mais detalhes sobre técnicas de programação inteira, veja, por exemplo, [1].

O ECL^iPS^e vem com uma biblioteca para métodos *branch and bound*, a qual veremos com mais detalhes posteriormente.

7.6 Projeção

Uma ideia implícita em nossa discussão sobre *branch and bound* pode ser vantajosamente generalizada. Considere uma restrição qualquer em função de X, Y e Z, que denotaremos `uma_restrição_qualquer(X, Y, Z)`.

Digamos que só o valor de X nessa restrição seja de interesse. Se tivermos que

```
uma_restrição_qualquer(X, Y, Z) :- X > Y, Y > Z, Z > 0.
```

é a única cláusula contribuindo para a definição de `uma_restrição_qualquer(X, Y, Z)`, temos então que X $>$ 0 é a única restrição que nos é de interesse, uma vez que é a restrição de em função de X mais simples que é compatível com a restrição original, no sentido de que a partir de qualquer substituição tal que X $>$ 0 podemos aumentar essa substituição com valores de Y e Z de modo a respeitar a restrição original.

Isso motiva a nossa definição de projeção:

Definição 7.1. *Uma substituição ι é dita uma* ***solução parcial*** *de um CP, que chamaremos C, se existe alguma substituição ρ tal que $\iota \cup \rho$ é uma solução de C.*

solução parcial

Definição 7.2. *Dizemos que uma restrição R_0 em função das variáveis X_0 a X_n é uma* ***projeção*** *da restrição R_1, em função das variáveis X_0 a X_m, com $m > n$, se toda solução de R_0 é uma solução parcial de R_1.*

projeção

Nem todo domínio admite projeções incondicionalmente. O domínio de árvores, por exemplo, não admite: podemos fazer projeções em algumas restrições nesse domínio, mas não em todas.

O *branch and bound* funciona porque, ao fazer uma ramificação, o que se faz na verdade é uma projeção em uma ou mais variáveis.

Não é difícil ver que a projeção é uma forma de simplificação e, dessa forma, podemos ver o *branch and bound* como uma sequência de simplificações de tipos diferentes.

Simplificações e projeções têm um papel importante na resolução de CPs, então será útil termos uma definição mais rigorosa. Mas antes precisamos da noção de equivalência:

Definição 7.3. *Duas restrições* ***restrições equivalentes*** *C_1 e C_2 são equivalentes em relação ao conjunto de variáveis V, o que denotaremos por $C_1 \leftrightarrow C_2$, se toda solução de C_1 restrita a V é uma solução parcial de C_2 e se toda solução de C_2 restrita a V é uma solução parcial de C_1.*

restrições equivalentes

Definição 7.4. *Var(X) denota o conjunto de variáveis de um CP X qualquer. Sejam uma restrição composta C e o conjunto de variáveis de interesse inter(C) $\subset$ var(C). Dizemos que C' é uma* ***simplificação*** *de C em inter(C) se inter(C) = var(C) e toda solução de C' é uma solução parcial de C.*

simplificação

Vale notar que, se dispomos de um algoritmo de simplificação nos moldes dessa definição, também dispomos de um algoritmo de solução: basta simplificar o CP em inter(C) = $\emptyset$.

7.7 Equivalência

Para terminar esta seção, consideremos o problema de equivalência de CPs, o qual é fortemente ligado ao problema de implicação e de bi-implicação. Para

tanto, voltamos à questão de normalização:

Definição 7.5. *Seja simp uma função que recebe um CP e um conjunto V de variáveis de interesse e retorna um CP. Dizemos que simp realiza um processo de* ***normalização canônica*** *em um dado domínio se para todo CP C no domínio e todo $V \subset var(C)$, simp(C,V) é uma simplificação de C e se $C_1 \leftrightarrow C_2 \Rightarrow simp(C_1, V) = simp(C_2, V)$.*

normalização canônica

Com um normalizador canônico, conseguimos responder sem grandes dificuldades a questão de equivalência: dois CPs são equivalentes se a normalização canônica de cada um segundo suas variáveis for idêntica. Daí, também conseguimos saber se um CP implica o outro, ou se o outro implica o um.

O algoritmo de unificação dado no Capítulo 1 é um algoritmo de normalização, como já foi notado, mas não descreve um processo de normalização canônica. Isso ocorre porque não tomamos cuidado suficiente com os nomes das variáveis.

Fica a questão: como poderíamos transformá-lo em um processo de normalização canônica?

7.8 Programação por restrições lógicas

Programação por restrições não precisa, a princípio, ser realizada com programação lógica. De fato, existem implementações de programação por restrições nos mais diversos paradigmas de programação. No entanto, programação lógica parece ser o paradigma mais natural para ser tomado base para programação por restrições. Um primeiro motivo já deve ser claro, nomeadamente que programação lógica é feita com base em um tipo importante de restrição, a em árvores, o que dá à programação lógica e, por extensão, à programação por restrições lógicas, o poder de uma linguagem de programação completa.

Mais no geral, no entanto, programação por restrições lógicas não se refere a uma linguagem ou paradigma, mas a um conjunto de linguagens. Cada uma dessas linguagens trabalha com um esquema diferente, o qual é determinado pelo domínio, pelos simplificadores de restrição e pelos resolvedores com que trabalha. Mais no geral, dado um domínio D, uma linguagem nesse conjunto, dita $CLP(D)$, é uma com simplificadores e resolvedores no domínio D. Exemplos são $CLP(\Re)$, no domínio dos números reais e $CLP(FD)$*, em domínios finitos.

Assim, uma linguagem de programação lógica *vanilla* seria uma $CLP(Árvore)$, onde as únicas restrições primitivas são igualdade e desigualdade (apesar de que desigualdade só está presente de forma implícita e pode ser implementada com base no símbolo de igualdade). Na realidade, cada $CLP(D)$ é feito com base no $CLP(Árvore)$ permitindo que as folhas sejam elementos em D e, eventualmente,

*FD vem de *Finite Domains*

tomando a adição de restrições e simplificadores e resolvedores específicos*. Prolog III, por exemplo, faz uso de um tipo de Simplex para a resolução de problemas de otimização com restrições lineares.

*Um exemplo de $CLP(D)$ em um D algo mais exótico pode ser encontrado em [2] e o de um CLP que foge do esquema $CLP(D)$ em [5]

Leituras Adicionais

A pesquisa e prática em programação lógica frequentemente é muito próxima à de linguagens funcionais e esse é o caso na área de transformações de programas. As transformações de *folding/unfolding* foram primeiro adaptadas para a programação lógica por Tamaki e Sato [1]. Mais informações podem ser encontradas em [2].

[1] Bertsimas, Dmitris e Tsitsiklis, John N., "Introduction to Linear Optimization", Athena Scientific.

[2] Dundua, Besik; Florido Mário; Kutsia Temur; Marin, Mircea (2015), "CLP(H): Constraint Logic Programming for Hedges", arXiv:1503.00336.

[3] Editores A. Biere, M. Heule, H. Van Maaren, T. Walsh (2009), "Handbook of Satisfiability", IOS Press.

[4] J. Herbrand (1930), "Recherches sur la theorie de la demonstration", PhD thesis, Universite de Paris, France, 1930.

[5] Mamede, Margarida; Monteiro Luís (1992), "A Constraint Logic Programming Scheme for Taxonomic Reasoning".

[6] Marriott, Kim; Peter J. Stuckey (1998), "Programming with constraints: An introduction" [S.l.]: MIT Press.

8 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 por si só não podem e, assim, 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, já que X e Y precisam estar instanciadas para a restrição ser usada.

Por enquanto, nos preocuparemos mais com restrições passivas e veremos exemplos de sua utilização no sistema ECL^iPS^e, que 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 daqui em diante.

Nossa exposição é baseada primariamente em [1] e em [3]. Na realidade, apesar de existirem muitos iteradores diferentes no ECL^iPS^e, todos são feitos com base na mesma construção, chamada `do/2`. Ademais, apenas uma das especificações de iteradores é mesmo fundamental. Uma chamada `( fromto(From, In, Out, To) do Body ).` é traduzida como:

```
do__1(Last, Last) :- !.
do__1(In, Last) :- Body, do__1(Out, Last).

do__1(From, To)?
```

Código 35: Fromto

É importante notar que, com *fromtos* aninhados, cada um mapeia a um acumulador diferente.

A partir desse iterador, podemos criar outros (que, na verdade, são abreviações). Por exemplo, `( foreach(X,Lista) do Body ).` é uma abreviação de `( fromto(Lista, [X|Xs], Xs, []) do Body ).`, e `(foreacharg(X,S) do Body).` é uma abreviação para `N1 is arity(S) + 1, ( fromto(1,I,I1,N1), param(S) do arg(I,S,X), I1 is I+1, Body ).`

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 se-

guindo 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` *(iterador2* `do` *... (iteradorn* `do` *busca)))*.

8.1 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 $3\times4=12$. 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).
```

Código 36: Busca 0

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
    )
  ).
```

Código 37: Partilha 0

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))
    )
  ).
```

Código 38: Partilha 1

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

8.2 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*[*]:

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

```
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.
```

Código 39: Busca 1

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).
```

Código 40: Backtracking

8.3 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 (assumidos como o padrão), podemos usar um $ no lugar de #, como em `X $:: 2..10` (vale repetir que o uso de intervalos inteiros é o padrão, o que significa que, na falta de uma símbolo como # ou $, o intervalo é entendido como sendo em inteiros). 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).
```

Código 41: XOR

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

8.4 Outras Bibliotecas

Como lidamos, nesta seção, com a biblioteca *suspend*, vale a pena fazermos um breve comentário sobre as demais bibliotecas. Isso é útil não só pelos motivos práticos (no uso das bibliotecas), mas também como um resumo das restrições que veremos mais para frente.

- A biblioteca *ic* (de *interval constraint*)* provê um resolvedor de restrições misto inteiro/real.
- A biblioteca *branch and bound* provê um *framework* para resolver problemas por *branch and bound* muito customizável.
- A biblioteca *eplex* provê otimização para problemas de LP e MIP (*linear programming* e *mixed integer programming*, respectivamente). Existem outras bibliotecas nomeadas *eplex_x*, para usar o resolvedor x específico (exemplos para valores de x são "cplex" e "gurobi").
- A biblioteca *ic_global* provê restrições globais sobre listas de inteiros.

*Esse nome vem do fato de que essas restrições atuam sobre intervalos, e as operações aritméticas realizadas sob elas são operações de intervalos. Veja [2] para mais detalhes.

- A biblioteca *ic_symbolic* provê um resolvedor para restrições sobre domínios simbólicos ordenados.
- A biblioteca *fd* provê um resolvedor para domínios finitos no geral.

Citamos algumas das que consideramos importantes. Mais detalhes sobre essas bibliotecas (e sobre as demais, não citadas aqui) podem ser encontrados em `http://eclipseclp.org/doc/bips/lib/fd/index.html`*

A biblioteca *ic_global* merece umas palavras a mais. Foi anteriormente mencionada a distinção entre restrições ativas e passivas, mas existe outra distinção, que às vezes pode ser mais significativa:

- **Restrições elementares** são as que agem sobre uma quantidade predeterminada de variáveis. Elas estão disponíveis, por exemplo, nas bibliotecas *ic* e *branch_and_bound*; **Restrições elementares**
- **Restrições globais** são as que agem sobre uma quantidade indeterminada de variáveis. Elas estão disponíveis, por exemplo, na biblioteca *ic_global* (e também em outras, não citadas aqui). **Restrições globais**

Um exemplo de restrição global é o `alldiferent/n`, que será vista adiante.

*Acessado em 22/02/2018.

Leituras Adicionais

A pesquisa e prática em programação lógica frequentemente é muito próxima à de linguagens funcionais e esse é o caso na área de transformações de programas. As transformações de *folding/unfolding* foram primeiro adaptadas para a programação lógica por Tamaki e Sato [1]. Mais informações podem ser encontradas em [2].

[1] Schimpf, Joachim, https://groups.google.com/forum/?hl=en#!msg/comp.lang.prolog/UYfgRxUWbGo/OKuxfWPEDqoJ

[2] T. Hickey and Q. Ju and M. H. van Emden, "Interval Arithmetic: from Principles to Implementation", Journal of the ACM

[3] Schimpf, Joachim, "Logical Loops", IC-Parc, Imperial College, London

[4] Schimpf, Joachim e Shen, Kish, "ECL^iPS^e- From LP to CLP", Theory and Practice of Logic Programming

9 Propagação de restrições em domínios finitos

Nesta seção trabalharemos primariamente com restrições em domínios finitos. Domínios finitos são importantes porque costumam ser bons para modelar decisões, o que é algo com que gostaríamos que o computador ajudasse.

Um exemplo simples e bem conhecido é o problema de coloração de um mapa: dado um conjunto finito de cores (de tamanho 15, por exemplo), precisamos colorir um mapa (digamos, o do Cazaquistão) de modo que nenhuma de suas regiões receba a mesma cor que outra com que faça fronteira*. Outro exemplo bem conhecido é o do "casamento a moda antiga" (menos popularmente conhecido como o "problema da correspondência bipartida"). Nesse problema, temos um conjunto de homens, um de mulheres a relação `gosta/2`, que existe quando um indivíduo i gosta de outro j. O problema é separar esses grupos de homens e mulheres em casais que se gostam.

Esses dois exemplos tem a particularidade de terem restrições primitivas binárias e, por isso, são chamados de CSPs binários. Um ponto interessante em CSPs binários é que sempre podem ser representados como um grafo não direcionado: cada variável (cada indivíduo no segundo exemplo ou cada região do Cazaquistão, no primeiro) é representada como um nó e cada restrição como um arco entre suas variáveis. Mais em geral, restrições n-árias CSPs podem ser representados como um multigrafo (isto é, um grafo em que podem existir mais de uma aresta entre dois vértices).

Em particular, problemas como os de roteamento e criação de cronogramas costumam ser facilmente expressos como CPs em domínio finito, o que indica sua importância comercial.

Sendo de uso tão amplo, essa classe de problemas (a de CPs em domínios finitos) foi estudada por diferentes comunidades científicas. A comunidade de Inteligência Artificial desenvolveu técnicas de consistência por arco e por nó, para CSPs, a comunidade de programação por restrições desenvolveu técnicas de propagação de limites e a comunidade de pesquisas operacionais desenvolveu técnicas de programação inteira. Daremos uma olhada em cada uma dessas abordagens a seguir.

9.1 Consistência por nó e por arco

Resolução de CSPs por consistência por nó e por arco acontece em tempo polinomial (mas, possivelmente, de forma incompleta)†. A ideia aqui é diminuir os domínios das variáveis, transformando o problema em outro equivalente (com as mesmas soluções). Se o domínio de alguma variável for vazio, é o fim do CSP.

*Incidentalmente, acontece que esse problema é essencialmente o mesmo que as companhias de aviação tem para alocar seu tráfico aéreo.

†Mas ela, assim como as demais formas de consistência vistas aqui pode ser usada em conjunto com *backtracking*, gerando um resolvedor completo, mas não mais em tempo polinomial.

Essa forma de resolução é dita baseada em consistência porque ele funciona propagando informações dos domínios de cada variável para os demais, tornando-os "consistentes" entre si.

Definição 9.1. *Uma restrição r/n é dita* ***consistente por nó*** *se $n > 1$ ou, X sendo for uma variável em r, se para cada d no domínio de X,* `X=d,r(X).` *resulta em sucesso. Uma restrição composta é dita consistente por nó se cada uma de suas restrições primitivas o é.* **consistente por nó**

Definição 9.2. *Uma restrição r/n é dita* ***consistente por arco*** *se $n \neq 2$ ou, se r é uma restrição nas variáveis X e Y e se D_x é o domínio de X e D_y o de Y, então $x \in D_x$ implica que existe $y \in D_y$ tal que* `X=x, Y=y, r(X,Y).` *resulta em sucesso. Uma restrição composta é dita consistente por arco se cada uma de suas restrições primitivas o é.* **consistente por arco**

Vale notar que essas noções de consistência são noções fracas no sentido de que um CSP pode não ser satisfazível e ainda manter consistência por arco e por nó.

Não é difícil escrever um código para manter consistência por arco e por nós. A seguir segue um exemplo. Ele é para fins demonstrativos: para algoritmos mais eficientes veja [5]. O funtor `apply/2` é o definido no Capítulo 6.

```
% consistente_por_no([Dominio-Restricao|DsRs],
%                     [NovosDominio-Restricao|DnsRs]) :-
%   NovosDominios sao consistentes por no com suas respectivas restricoes
%

consistente_por_no([], []).
consistente_por_no([D-Res|DsRs],[Dn-Res|DnsRs]) :-
  functor(Res, _, N),
  (
    N \== 1 ->
    Dn = D
  ;
    consistente_por_no_primitivo(D, Res, Dn),
  ),
  consistente_por_no(DsRs, DnsRs).


consistente_por_no_primitivo([], _, []).
consistente_por_no_primitivo([D1|Ds], R, [D1|Dn]) :-
  apply(R, [D1]), !,
  consistente_por_no_primitivo(Ds, R, Dn).


consistente_por_no_primitivo([D|Ds], R, Dn) :-
  consistente_por_no_primitivo(Ds, R, Dn).
```

Código 42: Consistência por nó

```
% consistente_por_arco([Dominios-Restricoes|DsRs],
%                       [NovosDominios-Restricoes|DnsRs]) :-
%   NovosDominios sao consistentes por arco com suas respectivas restricoes
%

consistente_por_arco([],[]).
consistente_por_arco([D1-D2-Res|DsRs],[D1n-D2n-Res|DnsRs]) :-
  consistente_por_arco_primitivo([D1-D2], Res, [D1n-D2n]),
  consistente_por_arco(DsRs, _).

consistente_por_arco([_|Cs],[Cns]).
  consistente_por_arco(Cs,Cns).


consistente_por_arco_primitivo([], _, []).
consistente_por_arco_primitivo(_, [], []).
consistente_por_arco_primitivo([D11|D1s]-[D22|D2s], R, [Dx|D1ns]-[Dy|D2ns]) :-
  (
    apply(R, [D11, D22]) ->
    (
      !, consistente_por_arco_primitivo(D1s-D2s, R, D1ns-D2ns),
      Dx = D11, Dy = D22
    )
  ;
    (
      consistente_por_arco_primitivo([D11]-D2s,R, _-_) ->
      (
        !, Dx = D11,
        consistente_por_arco_primitivo(D1s-D2s, R, D1ns-D2ns)
      )
    ;
      (
        consistente_por_arco_primitivo(D1s-[D22],R, _-_) ->
        (
          !,Dy = D22,
          consistente_por_arco_primitivo(D1s-D2s, R, D1ns-D2ns)
        )
      )
    )
  ).

consistente_por_arco_primitivo([D11|D1s]-[D22|D2s], R, D1ns-D2ns) :-
 consistente_por_arco_primitivo(D1s-D2s, R, D1ns-D2ns).
```

Código 43: Consistência por arco

O `consistente_por_no/2` recebe uma lista de pares "Domínio-Restrição". Se a restrição for unária, ela checa cada valor do domínio. Os valores que resultam em falha são retirados do domínio. `consistente_por_arco/2` tem alguns casos a mais, mas é essencialmente a mesma coisa.

9.2 Consistncia por limite

As noções de consistência desenvolvidas a cima funcionam bem para restrições em uma ou duas variáveis, mas se quisermos usar algo do tipo para mais variáveis, precisaremos generalizar um pouco:

Definição 9.3. *Uma restrição* `r/n` *nas variáveis* X_1, ..., X_n *é dita* ***consistente por hiper-arco*** *se para cada* x_i *no domínio de* X_i, *existem* x_j *nos domínios de* X_j *tais que para todo* $j \neq i$ *entre 1 e n,* X_i `=` x_i`,` X_j `=` x_j`.` *resulta em sucesso. Uma restrição composta é dita consistente por hiper-arco se cada uma de suas restrições o é.*

consistente por hiper-arco

Infelizmente, manter consistência por hiper-arco é algo caro demais para se fazer em um problema geral. Para encontrarmos uma nova checagem de consistência realmente útil, precisaremos restringir o domínio com que lidamos.

Dizemos que um **CSP é aritmético** se o domínio de cada variável é uma união finita de intervalos finitos de números inteiros e se as restrições são aritméticas. Muitos CSPs podem ser modelados como aritméticos de forma natural e muitos outros podem ser transformados em CSPs aritméticos por uma mudança de variáveis apropriada. Por exemplo, se o problema tem a ver com escolhas, uma mudança de variáveis natural é denotar cada escolha por um número. No problema da coloração do mapa do Cazaquistão, por exemplo, ao invés de denotar as cores como "vermelho", "azul", etc., podemos denotá-las como "1", "2", etc., obtendo resultados equivalentes.

CSP é aritmético

Lidando com CSPs aritméticos, podemos definir a noção de **consistência por limites**. A ideia é limitar o domínio de uma variável por limitantes inferiores e superiores. As seguintes convenções de notação serão convenientes:

consistência por limites

- $min_D(X)$:= algum x tal que para todo $y \in D$, $y \geq x$;
- $max_D(X)$:= algum x tal que para todo $y \in D$, $y \leq x$.

Dadas essas convenções, podemos construir uma definição de consistência por limites. Por incrível que pareça, existem três noções de consistência por limites, uma incompatível com a outra. Não obstante, os pesquisadores frequentemente confundem uma noção com a outra (os motivos de tal confusão serão explicados a seguir).

As três noções de consistência por limites serão denotadas $bounds(\mathbb{D})$, $bounds(\mathbb{Z})$ e $bounds(\mathbb{R})$. Intuitivamente, para $bounds(\mathbb{D})$, dentro de cada limitante no

domínio de uma variável há "suporte inteiro" para os valores dos domínios das outras variáveis ocorrendo na mesma restrição. Para *bounds*($\mathbb{Z}$), o "suporte inteiro" precisa ser apenas dentro do intervalo definido pelos limitantes inferiores e superiores das outras variáveis. Para *bounds*($\mathbb{R}$), os "suportes" podem assumir valores reais no intervalo definido pelos limitantes inferiores e superiores das outras variáveis.

Mais formalmente, temos as seguintes definições:

Definição 9.4. *Uma restrição* `r/n` *nas variáveis* $X_1, \ldots, X_n$ *é dita* ***bounds($\mathbb{D}$) consistente*** *se, para cada* X_i*, existem* ***valores inteiros*** x_j *no domínio de* X_j*, para* $j \neq i$*, tais que* $\{X_k = x_k : 1 \leq k \leq n\}$ *é uma solução de* `r/n`*.*

***bounds*($\mathbb{D}$) consistente**

Uma restrição composta é dita bounds($\mathbb{D}$) consistente se cada uma de suas restrições o é.

Definição 9.5. *Uma restrição* `r/n` *nas variáveis* $X_1, \ldots, X_n$ *é dita* ***bounds($\mathbb{Z}$) consistente*** *se, para cada* X_i*, existem* ***valores inteiros*** x_j*, para* $j \neq i$*, satisfazendo* $min_D(X_j) \leq x_j \leq max_D(X_j)$*, tais que* $\{X_k = x_k : 1 \leq k \leq n\}$ *é uma solução inteira de* `r/n`*.*

***bounds*($\mathbb{Z}$) consistente**

Uma restrição composta é dita bounds($\mathbb{Z}$) consistente se cada uma de suas restrições o é.

Definição 9.6. *Uma restrição* `r/n` *nas variáveis* $X_1, \ldots, X_n$ *é dita* ***bounds($\mathbb{R}$) consistente*** *se, para cada* X_i*, existem* ***valores reais*** x_j*, para* $j \neq i$*, satisfazendo* $min_D(X_j) \leq x_j \leq max_D(X_j)$*, tais que* $\{X_k = x_k : 1 \leq k \leq n\}$ *é uma solução real de* `r/n`*.*

***bounds*($\mathbb{R}$) consistente**

Uma restrição composta é dita bounds($\mathbb{R}$) consistente se cada uma de suas restrições o é.

É claro pelas definições que *bounds*($\mathbb{D}$) $\Rightarrow$ *bounds*($\mathbb{Z}$) $\Rightarrow$ *bounds*($\mathbb{R}$). Existem, no entanto, problemas práticos com *bounds*($\mathbb{D}$) e *bounds*($\mathbb{Z}$), nomeadamente que, para restrições lineares, por exemplo, manter consistêncai por *bounds*($\mathbb{D}$) ou por *bounds*($\mathbb{Z}$) é um problema NP-completo, enquanto que manter consistência por *bounds*($\mathbb{R}$) pode ser feito em tempo linear.

Também vale notar que um conjunto de domínios D é consistente em *bounds*($\mathbb{Z}$) ou *bounds*($\mathbb{R}$) para uma restrição `r/n` se, e só se, range(D) é consistente em *bounds*($\mathbb{Z}$) ou em *bounds*($\mathbb{R}$) para `r/n`, em que range(D) é definido como $range(D) := \{[min_{D_{x_i(X)}} \ldots max_{D_{x_{i(X_i)}}}]\}$, D_{x_i} o domínio da variável de `r/n` X_i. Essa propriedade é muito usada para não reexecutar propagadores de limites sem necessidade, e não vale para *bounds*($\mathbb{D}$).

Um problema com consistência em *bounds*($\mathbb{R}$) é que pode não ser claro como interpretar uma restrição inteira nos reais.

Com tantas diferenças entre essas noções, alguém poderia nos perguntar "Como alguém poderia confundi-las?". O fato é que, para muitas restrições, elas são equivalentes. Essas restrições são ditas monótonas. Para definir uma

restrição monótona, será conveniente ter uma definição mais refinada do que é um domínio. Daqui para frente, um **domínio** D de uma restrição `r/n` será tido como uma função do conjunto de variáveis de `r/n` para os conjuntos de valores que essa variável pode receber. Por exemplo, para `r(X,Y) :- X = Y.`, em que X pode assumir os valores 1 e 2 e, Y, os valores 2 e 3, D(X) = {1, 2} e D(Y) = {2, 3}. Por abuso de notação, diremos que $\Gamma \in D$ se Γ é outra função domínio tal que $\Gamma(X_i) \subseteq D(X_i)$ para $1 \leq i \leq n$.

domínio

Definição 9.7. *Uma restrição* `r/n` *é dita monótona com respeito às suas variáveis* X_i *se, e só se, existe uma ordem total*[*] $\prec_i$[†] *no domínio de* X_i, $D(X_i)$, *tal que, se para todo i,* $\Gamma \in D(X_i)$ *é uma solução de* `r/n`, *então também o é qualquer* Γ' *tal que, para* $i \neq j$, $\Gamma'(X_j) = \Gamma(X_j)$ *e* $\Gamma'(X_i) \preceq_i \Gamma(X_i)$.

Restrições na forma de desigualdades lineares[‡] e de desigualdades do tipo $x_1 \times x_2 \leq x_3$, com $min_D(x_i) \geq 0$, são monótonas, por exemplo.

Mais detalhes sobre consistência por limites podem ser vistos em [2]. A seguir é elaborado um exemplo de um método de propagação por limites.

Considere a restrição `X = Z + Y`. Ela pode ser escrita nas formas

$$X = Z + Y,\ Y = X - Y,\ Z = X - Y$$

Podemos ver que:

$$X \geq min_D(Y) + min_D(Z), X \leq max_D(Y) + max_D(Z) \quad (3)$$
$$Y \geq min_D(Y) + min_D(Z), Y \leq max_D(Y) + max_D(Z) \quad (4)$$
$$Z \geq min_D(Y) + min_D(Z), Z \leq max_D(Y) + max_D(Z) \quad (5)$$

Podemos usar essa observação para tentar diminuir os domínios de X, Y e Z. Com essa ideia, obtemos o seguinte programa:

---

[*]Lembrete: uma ordem total em um conjunto S é uma relação $\preceq$ que satisfaz, para todo $a, b, c \in S$: $a \preceq a$, $a \preceq b \wedge b \preceq a \Rightarrow a = b$, $a \preceq b \wedge b \preceq c \Rightarrow a \preceq c$ e $a \preceq b \vee b \preceq a$.

[†]A ordem pode mudar segundo a variável.

[‡]Exceto desigualdades da forma $x \neq y$, mas essas são muito fáceis de lidar, além de serem equivalentes para $bounds(\mathbb{D})$, $bounds(\mathbb{Z})$ e $bounds(\mathbb{R})$.

```
bounds_consistent_addition([], []).
bounds_consistent_addition([Dx,Dy,Dz], [Dnx, Dny, Dnz]) :-
  min_member(Dx, Xmin),
  min_member(Dy, Ymin),
  min_member(Dz, Zmin),

  Xm is max(Xmin, Ymin + Zmin),
  XM is min(Xmax, Ymax + Zmax),
  new_domain(Xm, XM, Dx, Dnx),

  Ym is max(Ymin, Xmin - Zmax),
  YM is min(Ymax, Xmax - Zmin),
  new_domain(Ym, YM, Dy, Dny),

  Zm is max(Zmin, Ymin - Ymax),
  ZM is min(Zmax, Xmax - Ymin),
  new_domain(Zm, ZM, Dz, Dnz).


new_domain(Vm, VM, Ds, Dn) :-
  Vm =< VM,
  (member(Vm, D) -> append([Vm], Dn) ; true),
  Vm is Vm + 1,
  new_domain(Vm, VM, D, Dn).
new_domain(_, _, _, []).
```

Código 44: Consistência por limites

Observações semelhantes podem ser feitas para outros tipos de restrições aritméticas. Para restrições do tipo $X \neq Z$ e $X \neq min(Z, Y)$, isso é especialmente simples de ser feito. Para restrições não lineares do tipo $X < Z \times Y$, isso pode ser especialmente complicado, ainda mais se Z e Y puderem assumir valores positivos e negativos, mas ainda pode ser feito.

Como é meio chato escrever uma regra para cada caso de restrição dessas para a manutenção de consistência por limites, o que é mais usual em um sistema que ofereça esse tipo de consistência é que suporte apenas uma quantidade reduzida dessas restrições, sendo as demais transformadas em versões equivalentes às quais essas restrições se apliquem, o que não é difícil de se fazer. Isso, no entanto, está sujeito ao potencial inconveniente de que restrições equivalentes mas escritas de formas diferentes podem oferecer oportunidades diferentes para a diminuição de domínio de cada restrição e a reescrita pode tornar um domínio que poderia originalmente ser grandemente simplificado, em um que sofra apenas uma pequena alteração (e o usuário pode ficar frustrado ao ver que uma diminuição de domínio "óbvia" não foi feita).

Apesar disso, a aplicação de consistência por limites frequentemente é útil.

Um programa que realiza essa aplicação é simples de se fazer: ele toma cada restrição primitiva e os domínios de suas respectivas variáveis e aplica um algo como o mostrado no código 44. Assim, temos um mecanismo de busca incompleto. Torná-lo um mecanismo completo é simples e pode ser feito com a adição do *backtracking*.

9.3 Consistência global

Consistência por limites também podem ser aplicadas a restrições em duas ou em uma variável, mas, nesse caso, consistência por arco ou por nó costumam resultar em diminuição maior nos domínios. Um problema geral, no entanto, pode ser composto por uma combinação de restrições de diferente aridade, tornando vantajosa a aplicação de diferentes noções de consistência.

No entanto, um potencial problema com abordagens baseadas em consistência é que elas consideram apenas uma ou um pequeno número de restrições e variáveis por vez, enquanto que frequentemente muitas restrições e variáveis oferecem informações sobre as demais.

Tome, por exemplo, a restrição `X` $\neq$ `Y, Y` $\neq$ `Z, X` $\neq$ `Z`, equivalente a dizer que as variáveis X, Y e Z são todas diferentes. Dos métodos de consistência que vimos, o mais indicado a essa restrição é o de consistência em arco, já que $\neq$ é de aridade 2. Mas esse método é muito fraco para restrições de desigualdade, o que significa que, na prática, resolveríamos isso por *backtraking*, que tem um crescimento assintótico exponencial.

Restrições desse tipo são tão usadas que receberam o nome especial de `alldiferent/1`* em vários sistemas CLP e é, talvez, a restrição mais estudada. Restrições que fazem uso da informação no domínio de muitas variáveis para realizar a atualização de cada domínio são chamadas restrições globais (como notado anteriormente). No caso do `alldifferent/1`, podemos notar que essa restrição é equivalente ao supramencionado problema de correspondência bipartida e que existem algoritmos eficientes para lidar com ele.

Assim como nas restrições vistas anteriormente, os domínios das variáveis em `alldifferent/1` são importantes na decisão de como resolvê-la. Em particular, se as variáveis são inteiras um algoritmo de propagação baseado em consistência por limites atinge um bom desempenho. Se as variáveis não são inteiras, no entanto, uma algoritmos baseados em consistência por hiper-arco podem ser usados. A seguir é apresentada os fundamentos de um tal algoritmo. Esse material é baseado em [1].

*Na verdade, a aridade dessa restrição pode ser arbitrária, mas, por simplicidade, assumimos que as variáveis que devem ser diferentes entre si estão organizadas em uma lista, tornando a aridade igual a um.

9.3.1 Alldifferent

Precisaremos das seguintes definições:

Definição 9.8. *Um **grafo** é uma tupla $G = (V, A)$, de vértices e arestas (V é um conjunto de vértices e A de arestas). Em um grafo não orientado, uma aresta é uma tupla de vértices.*

grafo

Dado um grafo $G = (V, E)$, um **pareamento**[*] M em G é um conjunto de arestas cujos vértices aparecem em apenas uma aresta de M (em outras palavras, M é um pareamento em G se os vértices em (V,M) tem no máximo grau 1).

pareamento

Definição 9.9. *Um pareamento M em G é dito máximo se para todos os pareamentos P de G, $|P| \leq |M|$.*

A teoria de pareamento é relevante para nosso problema porque, para qualquer restrição r/n com variáveis X_j, de domínios D_j, a informação de que $X_j \in D_j$ pode ser expressa por um grafo bipartido $(\cup_{j=1}^{n} X_i \cup (\cup_{i=1}^{n} D_i), E)$, onde $(X_i, d) \in E \Leftrightarrow d \in D_i$. Esse grafo é conhecido como **grafo valor** e pode ser construído em tempo polinomial.

grafo valor

É fácil ver que a restrição `alldifferent/n` tem solução se e somente se existe um pareamento máximo de tamanho n em seu grafo valor. Incidentalmente, existe um algoritmo que, dado um grafo, encontra um pareamento máximo em $O(\sqrt[2]{|X|} \times |E|)$.

Ademais, dado um pareamento máximo, existem algoritmos eficientes que tornam `alldifferent/n` hiper arco consistente (cada aresta em um pareamento máximo corresponde a uma atribuição de valor a uma variável da restrição).

Para desenvolvermos essas afirmações um pouco melhor, as seguintes definições serão uteis.

Definição 9.10. *Dado um pareamento P em G, um vértice e em G é dito pareado se $e \in P$, ou livre caso contrário.*

Definição 9.11. *Um pareamento P em G é dito uma **cobertura** para os vértices do G se todo v vértice de G pertence a P.*

cobertura

Definição 9.12. *Dado um pareamento P em G, um **caminho (ou ciclo) alternante** de G é um caminho (ou ciclo) cujos vértices são alternadamente pareadas e livres.*

caminho (ou ciclo) alternante

Teorema 9.1. *Um vértice pertence a algum pareamento máximo P de tamanho n se, e somente se, para qualquer pareamento máximo P', ele ou pertence a P' ou a um caminho alternante em P' de comprimento par que começa em um vértice livre em P', ou a um ciclo alternante em P' de comprimento par.*[†]

[*] Também comumente conhecido como *matching*.

[†] Se esse teorema não soa intuitivo, você pode fazer uns desenhos e se convencer de sua veracidade (a prova real não vai ser muito diferente dos desenhos que fizer).

Disso segue que quando uma aresta não pertencer a algum circuito ou caminho alternante, podemos extraí-la do grafo original. Para sabermos se tal ocasião acontece, podemos transformar o grafo em um grafo direcionado bipartido G' da seguinte forma: dado G=(V,A), nas arestas em A que são pareadas com P a aresta é orientada das variáveis para os valores e, nas demais, dos valores ara as variáveis.

Assim, podemos fazer uso do seguinte:

Teorema 9.2. *Todo circuito direcionado de G' tem comprimento par e corresponde a um circuito alternante par de G. Além disso, todo caminho em G' que for ímpar pode ser estendido em um caminho par, que corresponde a um caminho alternante par de G começando em um vértice livre.*

Para achar os caminhos procurados, podemos, então determinar os componentes fortemente conectados de G', assim como os caminhos direcionados em G' começando num vértice livre.

Um resumo do algoritmo é como se segue*:

1. É dada a restrição `alldifferent(X)`, onde $X = [X_1, ..., X_n]$;
2. Construímos então o grafo valor $G = (\cup_{j=1}^{n} X_i \cup (\cup_{i=1}^{n} D_i), E)$;
3. Computamos algum pareamento máximo de G, P;
4. Se $|P| < |X|$, falha;
5. Caso contrário, construa G';
6. Marque as arestas de G' que correspondem a arestas de G pertencentes a P como consistentes;
7. Encontre os componentes fortemente conectados de G' e marque as arestas nesses componentes como consistentes;
8. Encontre os caminhos direcionados que começam em um vértice livre e marque suas arestas como consistentes;
9. Para cada aresta de G' não marcada como consistente, remova a aresta correspondente em G.

9.4 Indexicals

Como deve ter dado para notar, uma operação chave em CLP(FD) é a propagação de restrições.

*Tem algumas sutilezas nele não comentadas aqui. Para saber mais, veja a bibliografia (em particular, [1] e, em [3], o Capítulo *Algorithms for matching*)

Em sistemas CLP(FD) iniciais, como o CHIP, as restrições eram primeiro transformadas em termos de forma canônica e, então, executados em um interpretador que, entre outras coisas, realiza a propagação de restrições (veja [6]). Devido ao sucesso na compilação de programas Prolog para a *máquina abstrata de Warren* (ou WAM, da sigla em inglês), que é o tipo de máquina abstrata mais usada na compilação de programas em Prolog, foram feitos esforços para a compilação de restrições em CLP(FD) nos mesmos moldes (sendo, na prática, uma extensão da WAM).

Nesse modelo, as restrições são traduzidas para códigos de baixo nível de modo que formas de propagação especializadas sejam usadas para cada tipo de restrição. Isso foi chamado de "modelo caixa-preta", já que o programador não sabe, a princípio, que tipo de propagação seria realizada em suas restrições. Na prática, esse modelo não se provou flexível o suficiente e foi abandonado.

indexicals

Uma construção chamada **indexicals**, mais flexível do que a anterior, foi então criada para auxiliar na compilação de restrições em CLP(FD). O modelo com *indexicals* é dito de "caixa de vidro", denotando seu caráter intermediário entre "o programador dita como as restrições são tratadas" e o seu contrário. No nosso contexto, um *indexical* é algo da forma `X in S`, onde X é uma variável de domínio finito e S é uma expressão.

Por exemplo, uma regra de propagação em consistência por limites para a restrição `X = Y + Z.` em *indexical* é:

X in $min(Y) + min(Z)..max(Y) + max(Z)$

A ideia é descrever o domínio de cada variável como uma função do domínio inicial por meio de construções como *in* e *min(Z)..max(Y)*.

Indexicals não são usados no ECL^iPS^e, mas são usados em outros sistemas e algumas outras formas de propagação fazem uso de métodos que tomam *indexicals* por base, por isso o discutimos brevemente por nesta ocasião.

Incidentalmente, os *indexicals* usados aqui provém de um conceito mais geral de *indexical* proveniente da filosofia da linguagem. Para entender esse conceito um pouco melhor, considere a seguinte situação (retirada de [4]):

Digamos que existem dois irmãos gêmeos, um que só diz a verdade e outro que só diz a mentira. Além disso, o primeiro irmão, que só diz a, verdade também tem uma crença perfeita do que é ou não verdade (isto é, todas as proposições que são verdadeiras ele crê serem verdade e analogamente o contrário). Em contrapartida, o outro irmão tem uma crença completamente imperfeita do que é verdade (o que é verdade ele crê não o ser e analogamente o contrário). Note que, ao serem feitos a mesma pergunta, os dois irmãos dariam a mesma resposta. Considerando essa situação e tendo em mente que o irmão mentiroso o deve dinheiro, um logicista pergunta a outro logicista "Você acha que é possível, fazendo perguntas de sim ou não, descobrir se um irmão é o mentiroso ou o verdadeiro?", ao que ele responde "Claramente não, já que eles dariam as mesmas respostas às mesmas questões". Você acha que o segundo logicista

estava correto?
Na verdade, não estava: Considere a pergunta "Você é o que diz a verdade?". Se ele o for, dirá que é. Se não o for, crerá que o é e dirá que não o é. Apesar disso, o segundo logicista estava correto ao dizer que eles dariam as mesmas respostas às mesmas questões: o "você" em "Você é o que diz a verdade?" se refere a "coisas" diferentes se usados com pessoas diferentes, então a pergunta feita ao mentiroso seria fundamentalmente diferente da feita ao não mentiroso. Aqui, "você" é um *indexical*.

O ponto é que um mesmo *indexical* pode significar coisas diferentes em contextos diferentes (na verdade, do ponto de vista filosófico, é isso que o define como sendo um *indexical*). No exemplo de *indexical* acima, os domínios iniciais de X, Y e Z poderiam mudar significativamente o que aquele *indexical* significa.

Leituras Adicionais

A pesquisa e prática em programação lógica frequentemente é muito próxima à de linguagens funcionais e esse é o caso na área de transformações de programas. As transformações de *folding/unfolding* foram primeiro adaptadas para a programação lógica por Tamaki e Sato [1]. Mais informações podem ser encontradas em [2].

[1] Basileos Anastasatos "Propagation Algorithms for the Alldifferent Constraint"

[2] C.W. Choi, W. Harvey, J.H.M. Lee, and P.J. Stuckey, "Finite Domain Bounds Consistency Revisited"

[3] Christos Papadimitriou (1998), "Combinatorial Optimization, Algorithms and Complexity", Dover Publications

[4] Smullyan, Raymond, "5000 B.C. and Other Philosophical Fantasies"

[5] E. Tsang (1930), "Foundations of Constraint Satisfaction", Academic Press.

[6] Neng-Fa Zhou (2006), "Programming Finite-Domain Constraint Propagator in Action Rules", Theory and Practice of Logic Programming, Vol. 6, No. 5, pp.483-508, 2006

10 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 ser 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, evidentemente 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,

    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).
```

Código 45: Preto e Branco

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

- As relações entre os Membros (termos simbólicos) são realizadas por restrições de *ic_symbolic* (vale lembrar, os "#" indicam que a restrição têm 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", na qual "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).

10.1 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`* para realizar a mesma função

*Perceba como ele foi usado no programa anterior.

(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),
```

Código 46: Indomain

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

```
%% 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)) ).
```

Código 47: Back Raso

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

10.2 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
    escolhe_var(Vars, Var, Resto),
    indomain(Var).
  ).

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

Código 48: Busca

Sendo um procedimento de busca tão comum, ele está disponível em algumas bibliotecas ECL^iPS^e(na *ic* e na *sd*, por exemplo) sob o nome `labeling/1` (o nome *labeling* é devido a razões históricas).

Leituras Adicionais

A pesquisa e prática em programação lógica frequentemente é muito próxima à de linguagens funcionais e esse é o caso na área de transformações de programas. As transformações de *folding/unfolding* foram primeiro adaptadas para a programação lógica por Tamaki e Sato [1]. Mais informações podem ser encontradas em [2].

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

11 Heurísticas

Nos métodos de busca vistos até agora, a ordem de variáveis adotada é a ordem apresentada na lista e a ordem de valores para cada variável é fixa. Para problemas maiores, pode ser mais conveniente considerarmos ordenações diferentes para cada tipo de problema. Essas ordenações são geralmente escolhidas por alguma heurística e, por isso, são chamadas ordenações heurísticas. Existem, naturalmente, duas classes de ordenações heurísticas: ordenação heurística de variáveis e de valores.

Antes de prosseguirmos, será útil falarmos sobre vetores, que serão utilizados nos programas a seguir.

11.1 Vetores

Listas são interessantes, mas, eventualmente, podemos querer realizar acesso em tempo constante a um membro qualquer, o que não é possível. Em outras linguagens, esse tipo de acesso é por costume realizado através de vetores. Vetores existem em Prolog padrão, de certa forma. Como os argumentos de funtores podem ser acessados em tempo constante por meio do funtor `arg/3`, a princípio, um funtor qualquer poderia ser usado como um vetor, o acesso aos membros do qual sendo feitos por meio de `arg/3`. Essa abordagem, no entanto, não só não parece muito elegante, como é pouco prática. Se temos algo que chamamos de vetor, gostaríamos de poder fazer algo como `Vetor[2] =/= 4`, por exemplo, o que com os meios do Prolog padrão não é possível.

Para esse tipo de situação, no ECL^iPS^e existe uma implementação de vetores como um açúcar sintático semelhante ao que vimos na implementação de listas, permitindo justamente esse tipo de utilização. Em particular, um vetor é escrito como uma variável seguida do funtor de lista (como em `Var[4]`), ou, se anônimo, como `[](A, B, C)`. Note que, assim, a representação de um vetor ou de uma lista vazia é a mesma. Para criar "vetores", possivelmente multidimensionais (na realidade, matrizes), existe o predicado `dim/2`, que pode ser usado ou para criar vetores ou para extrair sua dimensão.

11.2 As n-rainhas

Analisaremos o efeito de algumas heurísticas pela resolução do *problema das n-rainhas* quando $n > 3$*. É um problema por restrições tão bem conhecido que um estudante de programação por restrições poderia (talvez com razão) se sentir enganado se terminasse um livro sobre o tema sem saber como resolver esse problema em especial. A discussão a seguir é baseada, em grande parte, em [1].

*Quando $n = 2 ou n = 3$, o problema não tem solução.

Para quem não conhece, o *problema das n-rainhas* é o problema de alocar n rainhas em um tabuleiro como o de xadrez, mas $n \times n$, com a restrição de que as rainhas não têm permissão para se atacarem segundo as regras usuais de xadrez.

Existe um algoritmo polinomial para a resolução desse problema (que não veremos aqui), mas o que veremos muitas vezes é mais conveniente e até mais rápido. Antes de considerarmos o papel das heurísticas, vejamos como é uma solução mais simples. Notamos primeiro que, como temos n rainhas, precisa existir uma rainha em cada linha e em cada coluna. Ou seja, para cada coluna (ou cada linha) só precisamos indicar em qual linha (coluna) a rainha está, o que significa que a solução requer uma lista* de n posições, não uma matriz (como alguém poderia supor a princípio). O domínio de cada posição na lista é o intervalo inteiro $[1, n]$ e as restrições são que só pode existir uma rainha por linha (ou por coluna), o que se traduz na restrição `alldifferent(Cols)`, em que *Cols* é o vetor de colunas. Ademais, só pode existir uma rainha por linha, o que se traduz nas restrições† X_i `-` X_j `\ = \` `I - J` e X_i `-` X_j `\ = \` `J - I`, em que X_i e X_j são membros distintos da lista de colunas (ou linhas).

*Ou um vetor.

†Por conveniência, usamos X_i no lugar de $X[i]$.

```
%% rainhas(-Rainhas, ++Numero)
% Rainhas contém a posição de cada rainha por coluna
%

:- lib(ic)

rainhas(Rainhas, Numero) :-
  dim(RainhaStruct,[Numero]),
  restricoes(RainhaStruct, Numero),
  struct_para_lista(RainhaStruct, Rainhas),
  busca(Rainhas).

restricoes(RainhaStruct, Numero) :-
  ( for(I,1,Numero),
    param(RainhaStruct,Numero)
  do
    RainhaStruct[I] :: 1..Numero,
    (for(J,1,I-1),
      param(I,RainhaStruct)
    do
      RainhaStruct[I] #\= RainhaStruct[J],
      RainhaStruct[I]-RainhaStruct[J] #\= I-J,
      RainhaStruct[I]-RainhaStruct[J] #\= J-I
    )
  ).

struct_para_lista(Struct, Lista) :-
  ( foreacharg(Arg,Struct),
    foreach(Var,Lista)
  do
    Var = Arg
  ).
```

Código 49: Queens

Esse programa primeiro declara o vetor *RainhaStruct*, impõe a ele as restrições mencionadas, transforma o vetor em lista e realiza a busca (como explicada no Capítulo 10).computador do autor, consegue resolver o problema por esse programa para N até 29, mas, adotando um tempo limite de 50 segundos, $N = 30$ já está fora de mão (leva mais de 50 segundos).

Alguém poderia dizer que isso ocorre porque esse código não foi feito de maneira inteligente, já que a assimetria do problema, presente no fato de que a alocação de uma rainha em uma coluna central, por exemplo, resulta em maiores restrições nas posições das outras rainhas, não está representada no código. Esse realmente é o caso e, para fazê-lo deixar de ser, criamos uma nova versão do

programa.

11.3 Ordenação de variáveis

A alteração consiste na adição de um argumento Heur (de heurística) no predicado `rainhas/2` (que vira `rainhas/3`). Esse predicado nos diz qual a heurística usada para ordenar as variáveis. Podemos definir o `rainhas/2`, do exemplo passado, como um `rainhas/3` no qual o último argumento é "naive" (indicando que a heurística usada é, por assim dizer, boba).

```
busca(Rainhas, naive) :- labeling(Rainhas).
```

Código 50: Heurística Ingênua

Uma heurística mais esperta nesta situação seria começar a testar as variáveis do meio. Chamaremos essa heurística de *middle_out*: ela faz com que a busca comece do meio e vá alternando entre os vizinhos do lado esquerdo e direito, a partir do meio.

```
:- lib(lists).

busca(Rainhas, middle_out) :-
  middle_out(Rainhas, RainhasDeSaida),
  labeling(RainhasDeSaida).

middle_out(Lista, ListaDeSaida) :-
  halve(Lista, PrimeiraMetade, UltimaMetade),
  reverse(PrimeiraMetade, RevPrimeiraMetade),
  splice(UltimaMetade, RevPrimeiraMetade, ListaDeSaida).
```

Código 51: Heurística Meio

Aqui usamos a biblioteca *lists*, que contém, entre outros predicados, `halve/3`, `reverse/2` e `splice/3`. O que o primeiro e o segundo desses predicados faz é claro. O que o terceiro faz é juntar em *ListaDeSaida* as *UltimaMetade* e *RevPrimeiraMetade*, colocando um membro de cada alternadamente (como uma função *merge* do *merge sort*).

À parte dessa heurística, existe a *first_fail*, que seleciona como a próxima variável a que tem menos valores restantes no domínio. Para implementar essa heurística, usaremos o predicado da biblioteca *ic* `delete(-X, +List, -R, ++Arg, ++Select)`, que remove uma variável X da lista de variáveis *List*, deixando o resultado em R. O parâmetro *Arg* indica se a lista é uma lista de fato ou um funtor (no segundo caso, os argumentos do funtor são tratados como se fossem elementos de uma lista), e o *Select* indica o parâmetro de seleção (uma lista dos parâmetros possíveis pode ser encontrada na documentação do ECL^iPS^e).

No caso, nosso parâmetro *Select* é *first_fail*. Nossa busca com *first_fail* fica da seguinte forma:

```
busca(Lista, first_fail) :-
  ( fromto(Lista, Vars, Resto, [])
  do
    delete(Var, Vars, Resto, 0, first_fail),
    indomain(Var)
  ).
```

Código 52: Heurística First Fail

O parâmetro 0 indica que de fato lidamos com uma lista.

A experiência prática (que você pode realizar) indica que, para instâncias pequenas (para N pequenos), a diferença entre usar o *fisrt_fail* ou não é pequena. Para N grandes, entretanto, a diferença é notável.

A priori, não temos motivos para não usar o *first_fail* e o *middle_out* juntos. Chamaremos essa heurística de *moff*:

```
busca(Lista, moff) :-
  middle_out(Lista, ListaDeSaida),
  ( fromto(ListaDeSaida, Vars, Resto, [])
  do
    delete(Var, Vars, Resto, 0, first_fail),
    indomain(Var)
  ).
```

Código 53: Heurística Meio + First Fail

11.4 Heurísticas de ordenação de valor

A mesma observação usada para a heurística *middle_out* vale para a ordem dos valores escolhidos: a escolha de valores próximos ao centro restringem mais os valores de outras variáveis e têm a chance de falhar mais cedo. O ECL^iPS^e oferece um predicado `indomain/2` que nos ajuda nisso: seu segundo argumento nos dá um certo controle sobre a ordem em que os valores são testados. Em particular, `indomain(Var, middle)` começa a enumeração das variáveis pelo meio do domínio de *Var*. Existem também `indomain(Vars, min)` e `indomain(Vars, max)` que começam pelos menores e maiores valores, respectivamente (usando a ordem do domínio, pela qual "menor" é o que aparece antes na lista).

Podemos assim combinar nossa heurística anterior (*moff*) com esta (que chamaremos de *moffmo*):

```
busca(Lista, moffmo) :-
  middle_out(Lista, ListaDeSaida),
  ( fromto(ListaDeSaida, Vars, Resto, [])
  do
    delete(Var, Vars, Resto, 0, first_fail),
    indomain(Var)
  ).
```

Código 54: Heurística Meio First Fail Meio

Na tabela a seguir consta a quantidade de *backtrackings* por heurística para alguns valores de N:

Árvore 6: Retirado de [1]

	naive	middle_out	first_fail	moff	moffmo
8-queens	10	0	10	0	3
16-queens	542	17	3	0	3
32-queens	—	–	4	1	7
75-queens	—	–	818	719	0
120-queens	—	–	—	—	0

11.5 Outras Considerações

O problema das n-rainhas é simétrico, já que se uma solução tem a n-ésima rainha no m-ésimo quadrado da coluna i, outra solução teria a n-ésima rainha no m-ésimo quadrado da linha i. Perceba que uma solução é equivalente à outra, a menos de rotações de 90º do tabuleiro. Assim, quando a heurística de ordenação de variáveis *middle_out* sucede sem *backtrackings*, a heurística de ordenação de valores `indomain(Var, middle)` também deveria. Percebendo isso, podemos criar uma heurística de ordenação de valores que faz *backtrackings* como a *middle_out*. Depois de aplicada, uma "rotação" deve nos mostrar as mesmas soluções e na mesma ordem.

Para tanto, fazemos pequenas modificações no nosso programa. No lugar de iterar sobre a lista de variáveis, iteraremos sobre os valores do domínio. Ademais, no lugar de selecionar um valor para a variável atual (pelo `indomain/1`), selecionaremos uma variável para um valor (pelo `member/2`). Por fim, "rodamos" o resultado, para que os resultados produzidos sejam os mesmos produzidos pelo *middle_out*. Um código representando o que foi discutido é como segue:

```
rainhas(rotate, Rainhas, Numero) :-
  dim(RainhaStruct,[Numero]),
  constraints(RainhaStruct, Numero),
  struct_para_lista(RainhaStruct, QList),
  busca(QLista, rotate),
  rotate(RainhaStruct, Rainhas).

busca(QLista, rotate) :-
  middle_out_dom(QLista, MOutDom),
  ( foreach(Val, MOutDom),
    param(QLista)
  do
    member(Val, QLista)
  ).

middle_out_dom([Q | _], MOutDom) :-
  get_domain_as_lista(Q,OrigDom),
  middle_out(OrigDom, MOutDom).

rotate(RainhaStruct, Rainhas) :-
  dim(RainhaStruct,[N]),
  dim(RRainhas,[N]),
  ( foreachelem(Q,RainhaStruct,[I]),
    param(RRainhas)
  do
    subscript(RRainhas,[Q],I)
  ),
  struct_para_lista(RRainhas, Rainhas).
```

Código 55: Heurística Roda

Curiosamente, os resultados para o *rotate* são muito melhores do que para *middle_out*, como pode ser visto na seguinte tabela, mostrando a quantidade de *backtrackings* por N e por heurística:

Árvore 7: Retirado de [1]

	middle_out	rotate
8-queens	0	0
16-queens	17	0
24-queens	194	12
27-queens	1161	18
29-queens	25120	136

Isso ocorre porque o comportamento da propagação de restrições não é o mesmo para linhas e colunas. Se os valores para uma coluna são excluídos, exceto um, essa variável é instanciada ao valor restante. Esse é o comportamento exibido por *rotate*.

No entanto, no comportamento exibido por *middle_out*, se todos os valores de uma linha são excluídos exceto um, a propagação de restrições não instancia a variável. Isso ocorre porque as variáveis são representadas por colunas.

Uma forma de conseguir a mesma quantidade de propagação é usar uma redundância na formulação do problema. Neste caso, poderíamos adicionar outro tabuleiro ligado ao primeiro de modo que, se uma rainha i é posta no quadrado j de um tabuleiro, no outro, uma rainha j é posta no quadrado i. Técnicas desse tipo se mostraram muito eficientes.

11.6 O Predicado Search

Espero que a leitora não fique triste ao saber que grande parte do trabalho desenvolvido até agora poderia ser substituído pelo uso do predicado `search/6`. Ele tem 6 argumentos: *search(+Lista, ++Arg, ++Select, +Escolha, ++Metodo, +Opcao)*. O primeiro argumento, *Lista*, é uma lista de variáveis de decisão (as variáveis do problema) quando *Arg* é 0, ou uma lista de funtores compostos quando $Arg > 0$. Assumiremos que *Arg* é 0. O argumento *Select* representa uma heurística (feita pelo usuário ou não) de ordenação de variáveis, enquanto que o *Escolha*, uma de ordenação de valores. Pelo *Metodo*, podemos selecionar a "forma" pela qual a busca deve ser feita: algumas opções são *complete* (que realiza uma busca por todos os valores), e *sbds* (que faz uso da biblioteca SBDS de quebra de simetria para excluir partes simétricas da árvore de busca)*. O argumento *Opcao* é usado para passar parâmetros adicionais†, algumas das quais são *node(daVinci)*, para criar um desenho da árvore de busca usando a ferramenta de desenho daVinci, e *backtracks(-N)*, que nos dá em N a quantidade de *backtrackings* que ocorreram durante a busca.

*Recomendamos a consulta ao manual do ECL^iPS^e para mais detalhes sobre esses e outros métodos de busca.

†Novamente, é recomendado checar o manual do ECL^iPS^e para mais informações.

Leituras Adicionais

A pesquisa e prática em programação lógica frequentemente é muito próxima à de linguagens funcionais e esse é o caso na área de transformações de programas. As transformações de *folding/unfolding* foram primeiro adaptadas para a programação lógica por Tamaki e Sato [1]. Mais informações podem ser encontradas em [2].

[1] Krzysztof R. Apt and Mark Wallace "Constraint Logic Programming using ECLiPSe" Cambridge University Press, The Edinburgh Building, Cambridge CB2 8RU, UK

12 O Problema do General Bárbaro

Para ver se realmente compreendemos as ideias que vimos anteriormente, é sempre útil ver se conseguimos aplicá-las a algo concreto. Para isso, desenvolveremos aqui um problema específico (o "problema do general bárbaro"[*]). Mas faremos isso não com vistas em colocá-lo diretamente no *framework* desenvolvido até agora, mas sim em entendê-lo, pensar em diversos métodos de resolvê-lo e comparar esses métodos.

12.1 O Problema

O general romano Sulla, em suas campanhas pela Grécia, fez (entre outras coisas) o que qualquer outro general bárbaro faria em seu lugar, isto é, saqueou tesouros locais para obter vantagem na guerra (diferente de outros generais bárbaros, Sulla prometeu devolver parte do saque). Mas isso constituiu um problema, já que ele tinha uma grande variedade de tesouros para escolher levar ou não, enquanto que a capacidade de carga de seu exército era muito limitada, umas vez que dispunha de (relativamente) poucos navios. Mas o que criava a maior dificuldade nesse projeto era não o volume dos tesouros, mas o seu peso.

O valor de cada peça de tesouro é dado por uma função complicada de interesses internos e externos (no que influenciam, entre outros fatores, o preço pelo qual Sulla poderia vender dado item, no valor que cada soldado atribui ao item, no valor que seus conterrâneos atribuiriam ao item e na reputação que carregar um item lhe conferiria) e é sumarizada no que chamamos de *valor Sulla*, $\mathscr{S}$, de um item. Depois de calcular o $\mathscr{S}$ de cada item relevante, Sulla compilou a seguinte tabela preliminar com o peso médio e valor médio de cada item[†]:

Item	Valor	Peso	Quantidade
Livros	10	5	1000
Barras de Ouro	100	500	85
Estátuas de Ouro	500	4000	7
Jóias	50	50	200

Sem muito sucesso em resolver o problema, Sulla mandou que chamassem um matemático local para resolvê-lo. Theon de Smyrna estava por perto e foi convocado. No início, se sentiu incomodado por ter que deixar de lado seu trabalho com os sólidos de Platão em favor de ajudar um bárbaro a resolver um problema desse tipo inferior, como considerava, mas, notando que pode ser mais difícil para um morto desenvolver trabalhos com os sólidos de Platão, decidiu ajudar Sulla com seu problema.

[*] Também conhecido como "o problema da mochila", ou *knapsack problem*, em inglês.

[†] Não há evidências de que tal tabela tenha sido feita, mas é mantida aqui pela conveniência ao exemplo.

Ao ter a situação explicada, Theon deu uma leve risada e disse o que se segue. "Você me dá um problema grandioso e difícil, oh grande Sulla! Os Deuses devem ter nos posto juntos, porque eu devo ser um dos poucos helenos, se não o único, que sabe resolvê-lo! Isto porque vi meu pai o resolvendo quando era criança. Espero que não se incomode em ouvir por alguns momentos um velho como eu relembrar a infância, ainda mais que isso te será útil."

"Continue." - disse Sulla.

"Meu pai era um fazendeiro muito forte, mas também tinha o seus limites. Às vezes, precisava realizar trabalhos em partes distantes e, para isso, precisava levar suas ferramentas. Cada ferramenta faria o seu trabalho mais fácil, ou o ajudaria a terminá-lo mais rápido, ou lhe proporcionaria maior conforto ao realizá-lo. Cada uma, também, tinha o seu peso, e meu pai, forte como era, não conseguia carregar todas. Então, ele compilou uma tabela parecida com a sua, com um "valor" que cada ferramenta lhe proporcionaria, assim como seu peso."

"Mas, meu caro, esse problema parece muito mais simples do que o meu: seu pai só precisava decidir entre levar uma ferramenta ou não, enquanto que eu preciso decidir entre quantos de cada tipo de item levar!" - disse Sulla.

"É verdade que o problema de meu falecido pai parece muito mais simples do que o seu. Mas toda pessoa bem educada, como tenho certeza ser o seu caso, oh conquistador de nações!, sabe que não se pode confiar nas aparências. Neste caso, ver que os problemas são essencialmente os mesmos é muito simples. Toda criança sabe que qualquer número natural pode ser decomposto como uma soma de potências de dois. Por exemplo, $11 = 2^0 + 2^1 + 2^3$. Além disso, essa decomposição é única, cada potência aparecendo no máximo uma vez. Como pode ver, então, o seu problema é um problema de decisão: decidir quais potências de 2 comporão as quantidades de cada tesouro que levará. Esse problema só tem o posssível incoveniente de fazer uso de mais variáveis. Mas não muito mais: $\lfloor log_2(q_i) \rfloor$, em que q_i é a quantidade a ser levada do item i.*"

"Mas esse problema adiciona complicações extras que não me parecem tão necessárias." - disse Sulla.

"Não tanto assim, mas tudo bem. Não quero tomar muito mais tempo seu, então vou direto ao ponto. Um modelo mais geral ao problema que deseja resolver é o seguinte:

$$\begin{aligned} maximizar \sum_{i=1}^{n} q_i \mathscr{S}_i, \\ tal\ que \sum q_i p_i \leq C, \\ 0 \leq q_i \leq l_i,\ q_i\ inteiro \end{aligned}$$

*Estamos adicionando, aqui e mais adiante, notação um pouco mais moderna do que a que o velho Theon poderia ter usado, por conveniência ao exemplo.

em que q_i é a quantidade a ser levada do item i, p_i o peso do item i (em $\mathscr{S}$), C a capacidade máxima a ser levada e l_i limite máximo do item i a ser levado (não podemos levar mais do que l_i do item i, talvez porque não existam mais do que l_i desses itens).

Esse é um problema de programação inteira. Existem muitas formas de resolver problemas de programação inteira. Se tiver algum tempo sobrando, eu poderia desenvolver uma outra forma com você. Será divertido e intrutivo."

"Vá em frente." - disse Sulla.

"Imagine que você sabe resolver esse problema para $C' < C$ e para todo subconjunto dos itens (um subconjunto de $\{1, ..., n\}$) e tenha em mãos uma solução para ele. Se retirarmos dessa solução um item i, de peso p_i, deve ser claro que a solução restante é ótima para o problema cuja capacidade de $C - p_i$ e conjunto de itens $\{1, ..., n\} - \{i\}$. Isso indica que uma solução ótima goza de uma subestrutura ótima. Isto é, a remoção de itens dessa solução gera uma solução que é ótima para outro problema, de fato um subproblema, mais simples. Se temos a solução para algum desses subproblemas, só precisamos expandi-lo, considerando itens que não foram considerados. Para cada um desses itens, sabendo que a nova solução ótima $op(i, c)$ para uma capacidade c, ao considerar a adição de um item i, é:

$$op(i, c) = \begin{cases} op(i-1, c), & se\ p_i > c \\ max\{op(i-1, c), op(i-1, c-p_i) + \mathscr{S}_i\} & se\ p_i \leq c \end{cases}$$

Fazendo $op(0, C) := 0$ e computando os valores de $op(i, c)$ por essa recursão* temos o valor ótimo que o meu pai queria. Se quisermos resolver o seu problema, precisamos fazer apenas uma pequena alteração, como a seguinte:

$$op(i, c) = max_{q_i}\{op(i-1, c-q_i p_i) + q_i \mathscr{S}_i \mid c \geq q_i p_i\}$$

Não deve ser difícil de ver como isso nos leva ao resultado desejado."

"Oh, mas Theon! Devo ter me enganado ao achar que era um matemático, pois você me está parecendo mais um ator de teatro! Isso porque qualquer ator de teatro me daria uma resposta como essa, mas ela é longe de satisfatória. Mas é pior do que um ator de teatro, porque quem quer que fosse incumbido de realizar a tarefa pelo método que você propôs ficaria um tempo intolerável fazendo contas, tempo esse que é, de fato, morto. E você seria o responsável pela morte." - disse Sulla. tempo passado faz parte da nossa morte.

Theon, percebendo a gravidade da situação, disse o seguinte. "Essa formulação, como eu disse antes, tinha um caráter mais introdutório e instrutivo. Se quer uma mais séria, existem várias e em vários sabores. Uma delas em particular faz uso extenso de técnicas de programação inteira, que são desen-

*Conhecida como recursão de Bellman.

volvidas há muitas décadas*. Fazendo uso dessas técnicas, podemos gerar o seguinte programa:

```
:- lib(eplex).

modelo(Pesos, Valores, Limitantes, Capacidade, Quantidades, Peso, Valor) :-
    ( foreach(Limitante,Limitantes), foreach(Quantidade,Quantidades) do
      Quantidade $:: 0..Limitante
    ),
    integers(Quantidades),
    Quantidades*Pesos $= Peso,
    Peso $=< Capacidade,
    Quantidades*Valores $= Valor.

resolve(Pesos, Valores, Limitantes, Capacidade, Quantidades, Peso, Valor) :-
    modelo(Pesos, Valores, Limitantes, Capacidade, Quantidades, Peso, Valor),
    optimize(max(Valor), Valor).

principal :-
    dados(Nomes, Pesos, Valores, Limitantes, Capacidade),
    resolve(Pesos, Valores, Limitantes, Capacidade, Quantidades, Peso, Valor),
    ( foreach(Quantidade,Quantidades), foreach(Nome,Nomes) do
      ( Quantidade >0 -> printf("%d of %w%n", [Quantidade,Nome]) ; true )
    ),
    writeln(peso = Peso ; valor = Valor).
```

Código 56: Conquistador Bárbaro Eplex

Podemos, analogamente, fazer uso da extensa pesquisa em otimização por restrições, com algo como o seguinte programa:

*Infelizmente, não há evidências de desenvolvimentos de programação inteira na época de Theon, mas a fala é mantida aqui por sua conveniência.

```
:- lib(ic).
:- lib(branch_and_bound).

model(Pesos, Valores, Limitantes, Capacidade, Quantidades, Peso, Valor) :-
        ( foreach(Limitante,Limitantes), foreach(Quantidade,Quantidades) do
                Quantidade $:: 0..Limitante
        ),
        integers(Quantidades),
        Quantidades * Pesos $= Peso,
        Peso $=< Capacidade,
        Quantidades * Valores $= Valor.

resolve(Pesos, Valores, Limitantes, Capacidade, Quantidades, Peso, Valor) :-
        model(Pesos, Valores, Limitantes, Capacidade, Quantidades, Peso, Valor),
        Custo #= -Valor,
        minimize(search(Quantidades, 0, largest, indomain_reverse_split, complete, []),
                 Custo).

principal :-
        dados(Nomes, Pesos, Valores, Limitantes, Capacidade),
        resolve(Pesos, Valores, Limitantes, Capacidade, Quantidades, Peso, Valor),
        ( foreach(Quantidade,Quantidades), foreach(Nome,Nomes) do
                ( Quantidade > 0 -> printf("%d of %w%n", [Quantidade,Nome]) ;
                  true )
        ),
        writeln(peso=Peso ; valor=Valor).
```

Código 57: Conquistador Bárbaro IC

Para saber qual dos métodos é o mais interessante, podemos simplesmente tomar uma pequena quantidade dos itens iniciais, fazer um teste, e ver qual resolvemos mais rápido.

"Isso pode ser uma dificuldade, porque, como ainda não alcançamos todos os lugares de onde os itens serão subtraídos, só temos uma ideia muito vaga sobre quais itens existem e qual o valor de cada." - disse Sulla.

"Um problema facilmente remediado. Podemos fazer testes com valores aleatórios, como exemplificado aqui." e, ao dizer isso, Theon primeiro explicou a notação que usaria e, depois, mostrou algo equivalente ao seguinte código a Sulla:

```
:- lib(ic_prop_test_util).

aleat_dados(dados([],[],[],[],_), 0).
aleat_dados(dados([Contador|Ls], [P|Ps], [V|Vs], [Q|Qs], _), Contador) :-
        random_int_between(1, 50,  P),
        random_int_between(1, 100, V),
        random_int_between(1, 15,  Q),
        Novo_Contador is Contador - 1,
        aleat_dados(dados(Ls, Ps, Vs, Qs, _), Novo_Contador).


inic_dados(dados(Nomes, Pesos, Valores, Qtd, _)) :-
        aleat_dados(dados(Nomes, Pesos, Valores, Qtd, _), 100).


dados(Nomes, Pesos, Valores, Qtd, 400) :-
        init_dados(dados(Nomes, Pesos, Valores, Qtd, 400)).
```

Código 58: Problemas Aleatórios

"Parece que, pelo menos por ora, nosso problema está resolvido." - disse Sulla.

"Sim, o seu problema está."

12.2 Os Testes

Levando em conta a discussão anterior, foram feitos testes para checar a eficiência dos métodos discutidos. Eles foram realizados em um computador Inspiron 5447, versão A06, com barramento de 64 bits, produzido pela Dell. Esse computador conta com um processador Intel Core i7-4510U, a 2GHz e sistema operaiconal Debian Strech. A versão do sistema de programação usado é ECL^iPS^e 7.0 (de Julho de 2018).

Os testes foram executados da seguinte forma. O método que faz uso de programação inteira foi testado 8 vezes, com o teste i contando com 2^{i+4} tipos de itens.

Cada tipo de item tem um peso, um volume que ocupa e uma quantidade de itens disponíveis. Nos testes que fazem uso de programação inteira, o peso é escolhido aleatoriamente como um valor entre 1 e 50 (unidades de peso), o volume como um valor entre 1 e 100 (unidades de volume) e, a quantidade, como um valor entre 1 e 15. Dada a natureza aleatória dos dados, é necessário fazer o teste com a mesma quantidade de tipos de itens mais de uma vez, e os que têm mais tipos de itens devem ser testados mais do que os que têm menos tipos de itens. A maneira escolhida foi de executar j vezes o teste com 2^j itens.

Infelizmente, realizar os mesmos testes com o método que faz uso de *branch and bound* foi impraticável, dado o tempo computacional necessário para tanto ser, no geral, proibitivo para $i \geq 6$*. No lugar disso, então, foram executados três testes para esse, os dois primeiros como explicados anteriormente (no caso, fazendo $i + 4 = 4$ e $i + 4 = 5$) e o terceiro caso de teste foi feito com 50 itens, o qual foi executado 6 vezes (sendo a quantidade de itens e de testes realizados as únicas diferenças entre esta modalidade e a explanada acima).

Os resultados são como segue na Tabela 1 (medições de tempo com dois valores significativos). Os valores na coluna "Programação Inteira" são referentes aos respectivos tempos para a resolução dos problemas por meio de restrições provindas da biblioteca *eplex* de programação inteira†, enquanto que os valores na coluna "Branch and Bound" são referentes aos respectivos tempos para a resolução dos problemas por meio de restrições provindas da biblioteca *branch_and_bound*‡ . Como os testes para 2^7 itens ou mais se tornaram inviáveis se fazendo uso da biblioteca ECL^iPS^e *branch_and_bound*, os respectivos espaços na tabela foram substituídos por "NA".

* "No geral", porque, os dados sendo aleatórios, às vezes o programa termina rapidamente. Esse, entretanto, não é o caso geral.

†Essa biblioteca ECL^iPS^e é uma maneira geral de fazer interface com uma variedade de *solvers* para problemas de programação linear e inteira. Neste caso em particular, os *solvers* carregados são o *COIN-OR CLP*, versão 1.6 (linear) com *CBC* versão 2.9 (inteiro misto)[4].

‡A biblioteca ECL^iPS^e *branch_and_bound* provê predicados genéricos que implementam a busca por *branch and bound*, isto é,

1. achando uma solução,
2. adicionando uma restrição que requer uma solução melhor do que a melhor vista até então e
3. achar uma solução que satisfaz essa nova restrição.

Tempos	Programação inteira	Branch and Bound
Menor tempo para 2^4	0,00s	0,00s
Maior tempo para 2^4	0,01s	0,09s
Tempo médio para 2^4	0,00s	0,02s
Menor tempo para 2^5	0,00s	0,00s
Maior tempo para 2^5	0,01s	0,07s
Tempo médio para 2^5	0,00s	0,01s
Menor tempo para $2^6/50$	0,00s	0,08s
Maior tempo para $2^6/50$	0,01s	4,70s
Tempo médio para $2^6/50$	0,01s	1.81s
Menor tempo para 2^7	0,01s	NA
Maior tempo para 2^7	0,01s	NA
Tempo médio para 2^7	0,01s	NA
Menor tempo para 2^8	0,02s	NA
Maior tempo para 2^8	0,02s	NA
Tempo médio para 2^8	0,02s	NA
Menor tempo para 2^9	0,02s	NA
Maior tempo para 2^9	0,04s	NA
Tempo médio para 2^9	0,03s	NA
Menor tempo para 2^{10}	0,05s	NA
Maior tempo para 2^{10}	0,07s	NA
Tempo médio para 2^{10}	0,06s	NA
Menor tempo para 2^{11}	0,60s	NA
Maior tempo para 2^{11}	0,80s	NA
Tempo médio para 2^{11}	0,70s	NA
Menor tempo para 2^{12}	0,14s	NA
Maior tempo para 2^{12}	0,16s	NA
Tempo médio para 2^{12}	0,14s	NA

Tabela 1: Resultados

Leituras Adicionais

A pesquisa e prática em programação lógica frequentemente é muito próxima à de linguagens funcionais e esse é o caso na área de transformações de programas. As transformações de *folding/unfolding* foram primeiro adaptadas para a programação lógica por Tamaki e Sato [1]. Mais informações podem ser encontradas em [2].

[1] Kellerer, Hans e Pferschy, Ulrich e Pisinger, David, "Knapsack Problems", Springer

[2] Plutarch, "Sulla", traduzido por John Dryden

[3] ECL^iPS^e, http://www.eclipseclp.org/examples/

[4] COIN-OR, https://projects.coin-or.org/CoinBinary

13 Dualidade e relaxação

- Q: Qual é o contrário do "espaço dual"?
- R: O espaço "donothing".

Eduardo Tengan, em uma aula qualquer

A noção de dualidade não é uma noção perene só na álgebra linear e suas áreas adjacentes (conhecidas em conjunto como "matemática"), também ocupa um lugar especial na teoria de otimização. Em otimização, dualidade vem em diversas formas: dualidade de programação linear, de programação linear inteira (lagrangeana), entre outras. Frequentemente, os métodos de otimização mais confiavelmente eficicientes são métodos primal-dual.

A dualidade é um tema unificador porque podemos expressar todos os duais como dual de inferência ou de relaxação (frequentemente, como ambos). Um dual de inferência pode ser visto como o problema de inferir das restrições nas variáveis um maior limitante inferior na imagem da função objetivo (assumimos daqui para frente, sem perda de generalidade, que queremos minimizar a função). Assim, a busca ocorre sobre provas, no lugar de sobre valores das variáveis. Mais precisamente, se tivermos um problema de otimização da forma

$$\begin{aligned} &min\ f(x), \\ &R(x), \\ &x \in X, \end{aligned}$$

em que $R(x)$ denota as restrições sobre x, sendo X o seu domínio, seu respectivo **dual de inferência** nos coloca o problema de maximizar o limitante inferior em $f(x)$ que é inferível das restrições. Isso ocorre na forma de uma busca por uma prova do limitante ótimo (ou "bom o suficiente", se não pudermos pedir por um "ótimo"):

dual de inferência

$$\begin{aligned} &max\ v, \\ &C(x) \vdash^P (f(x) \geq v), \\ &v \in \mathbb{R}, P \in \mathcal{P}, \end{aligned}$$

em que $C(x) \vdash^P (f(x) \geq v)$ indica que P é uma prova de $(f(x) \geq v)$, usando o que é assumido em $C(x)$ (no caso, $C(x)$ indica as restrições). O domínio de P é uma família de provas $\mathcal{P}$, sendo (v, P) um par de solução dual. Quando o problema inicial (dito "**primal**") não tem soluções, o conjunto de soluções para

primal

o dual é ilimitado.

Quando o primal for um problema de programação linear, por exemplo, a família de provas pode consistir de combinações lineares não-negativas de restrições de desigualdade, sendo a solução dual identificada com os multiplicadores na combinação linear que deriva o maior limitante.

Direto da definição, podemos inferir o **princípio de dualidade fraca**: um valor factível v do dual de inferência nunca pode ser maior do que qualquer valor factível do primal, o que nos possibilita limitar o valor para o ótimo do primal. Ademais, a diferença (a "lacuna") entre os valores ótimos do primal e do dual é chamada **lacuna de dualidade**. A lacuna de dualidade pode ser igual a zero, mas não é garantido que esse possa ser o caso (isto é, não é garantido que as soluções ótimas primal e dual possam ter o mesmo custo). Em particular, pode ser que exista um maior limitante inferior a $f(x)$, mas esse limitante inferior não ser inferível em $\mathcal{P}$. Quando esse é o caso, $\mathcal{P}$ é dita incompleta. Mais em geral, se $C(x)$ implica C mas não existe $P \in \mathcal{P}$ que infira C, $\mathcal{P}$ é dita incompleta. Caso contrário, ela é dita completa. Quando $\mathcal{P}$ for completa, a lacuna de dualidade pode, ao menos em princípio, ser reduzida a zero pelo dual, o que constitui o princípio de **dualidade forte***.

princípio de dualidade fraca

lacuna de dualidade

dualidade forte

A ideia de dualidade por inferência provê um plano unificador para várias ideias centrais em otimização no geral e em programação por restrições em particular. Entre outras coisas, dualidade por inferência nos provê métodos para *aprendizado de cláusulas*, frequentemente muito úteis. Buscas com *aprendizado de cláusulas* cresceram de pesquisa em *aprendizado baseado em explicações*, um ramo de inteligência artificial que busca melhorar a eficiência de métodos de busca por *backtracking* por meio de "explicações de falha", na forma de novas restrições ao problema original. Para CSPs em geral, tais "explicações" são chamadas *nogood* (ou, às vezes, "conflitos"). Foi mostrado que uma grande quantidade de problemas que não podem (até o presente momento) ser resolvidos por outras técnicas podem ser resolvidos de maneira eficiente por *aprendizado de cláusulas*. Em particular, problemas (anteriormente) abertos em teoria de grupos foram resolvidos com essa técnica (entre outras). Para mais informações, veja [1]

13.1 Duais

Existem na literatura de programação por restrições, assim como na de programação matemática, diversas noções específicas de dual, para diversos tipos de problemas diferentes. Entre elas, destacam-se como especialmente úteis os duais *substituto* e *lagrangeano*, aos quais devotaremos alguma atenção nesta seção e na próxima, colocando ênfase especial na relação entre eles. Então,

*Os "princípios" de dualidade fraca e forte são às vezes referidos como teoremas (porque precisam ser provados). Suas provas, no entanto, são tão simples, que foram omitidas a fim de não aborrecer a leitora.

nossa atenção se voltará ao *dual por ramificação*, que será útil em nosso posterior estudo de técnicas de ramificação.

13.1.1 Dual Substituto

Seja um problema do tipo

$$\begin{aligned} & min f(x), \\ & g(x) \geq 0, \\ & x \in X, \end{aligned} \tag{6}$$

em que $g(x)$ não necessariamente é linear e $x \in X$ indica as demais restrições em x. Note que problemas de programação inteira mista são um caso particular em que $g(x)$ é uma função linear e $X = \mathbb{R}^n + \mathbb{Z}^m$. Obtemos o dual de inferência desse problema tomando combinações lineares não negativas e implicação como método de inferência, fazendo

$$\begin{aligned} & max\ v, \\ & (g(x) \geq 0) \vdash^P (f(x) \geq v), \\ & P \in \mathcal{P}, v \in \mathbb{R}, \end{aligned} \tag{7}$$

em que cada prova em $\mathcal{P}$ corresponde a um vetor u de multiplicadores e a prova P deduz $f(x) \geq v$ de $g(x) \geq 0$ quando a $ug(x) \geq 0 \Rightarrow f(x) \geq v$. Isto é o mesmo que dizer que o mínimo de $f(x)$ com $ug(x) \geq 0$ e $x \in X$ é no mínimo v. Daí segue que essa formulação é equivalente a

$$\begin{aligned} & max\ \sigma(u), \\ & \sigma(u) = min_{x \in X}\{f(x) | ug(x) \geq 0\}, \\ & u \geq 0. \end{aligned}$$

Pergunta: o *dual substituto* é completo?

13.1.2 Dual Lagrangeano

O dual lagrangeano do problema (6) é semelhante ao dual substituto, exceto que, no dual lagrangeano, as provas fazem uso de combinações lineares e *dominação*[*]. O dual é o problema (7), em que $\mathcal{P}$ consiste de provas que mesclam combinação

[*]Dizemos que uma desigualdade $g(x) \geq 0$ domina uma $h(x) \geq 0$ se $g(x) \leq h(x)\ \forall x$, ou se não existe x tal que $g(x) \geq 0$.

linear com dominação: $f(x) \geq v$ é inferido de $g(x) \geq 0$ se $\exists\, l \geq 0 : lg(x)$ domina $f(x) \geq v$.

Como dominação é um requisito mais fraco do que implicação, o dual lagrangeano resulta em um dual de inferência mais fraco do que o *substituto* e, assim, possibilita limitantes mais fracos em relação ao *substituto*. Apesar disso, para outros fatores o lagrangeano goza de propriedades mais interessantes do que o *substituto*, o que o torna atraente para uma grande variedade de problemas. Por exemplo, o conjunto de soluções factíveis para o *dual lagrangeano* é côncavo (o que não vale, no geral, para o *dual substituto*). Isso pode ser visto facilmente se, supondo que $lg(x) \geq 0$ é factível para $l \geq 0$, observarmos que o lagrangeano maximiza v com $l \geq 0$ e $lg(x) \geq f(x) - v\ \forall x \in X$. Reescrevendo $lg(x) \leq f(x) - v$ como $v \leq f(x) - lg(x)$ e fazendo $\lambda(l, x) = f(x) - lg(x)$, nosso lagrangeano é equivalente a

$$
\begin{aligned}
&max\ \lambda(l), && (8)\\
&\lambda(l) := inf_{x \in X}\{f(x) - lg(x)\}, && (9)\\
&l \geq 0, && (10)
\end{aligned}
$$

e o conjunto de ínfimos de funções afim (como em (9)) é um conjunto côncavo (λ é afim para x fixo). O vetor l é popularmente conhecido como vetor de multiplicadores de lagrange.

13.1.3 Dualidade por Ramificação

Intuitivamente, se temos uma árvore de busca para um problema de otimização, essa árvore vista "de baixo para cima", isto é, das folhas para a raíz, nos provê um certificado de otimalidade a uma solução ótima. Além disso, dada tal árvore de busca, cada subárvore dela constitui uma relaxação ao problema original e provê um limitante ao custo ótimo do problema original. Assim, parece natural definirmos um dual por ramificação ao problema de otimizar um $f(x)$ nas folhas de uma árvore de busca $\mathcal{T}$ da seguinte forma:

$$
\begin{aligned}
&max\ B(\mathcal{T}'),\\
&\mathcal{T}' \in \mathcal{T},
\end{aligned}
\qquad (11)
$$

em que B é o limitante de f em $\mathcal{T}'$ e $\mathcal{T}$ é uma coleção de subárvores de $\mathcal{T}$.

As estratégias de ramificação que veremos logo mais podem ser vistas como instâncias do dual de ramificação. Para mais detalhes sobre o dual de inferência veja [2].

13.2 Complexidade

Um certificado de factibilidade (infactilibidade/otimalidade) é um pedaço de informação que permite a verificação da factibilidade (infactibilidade/otimalidade) da instância de um problema. Dizemos que um problema de otimização pertence ao NP (de *nondeterministic polynomial*) se existe um certificado de factibilidade polinomial para qualquer instância factível do problema: a computação requerida para testar factibilidade, pelo certificado, é limitada por uma função polinomial do tamanho da instância do problema. Um problema de otimização é co-NP se existe um certificado polinomial de otimalidade para qualquer instância.

Se um problema de otimização primal for co-NP, seu dual de inferência é NP para alguma família de provas $\mathcal{P}$: se o problema for co-NP, seja $\mathcal{P}$ a família de provas P_i (que assumimos polinomiais) de otimalidade da instância i, de valor ótimo v_i, provê um certificado polinomial de factibilidade do dual (que é, portanto, NP).

Programação linear pertence a ambos NP e co-NP. Problemas desse tipo são tidos ter "boa caracterização", no sentido de que soluções factíveis e provas de otimalidade são facilmente escritas.

Leituras Adicionais

A pesquisa e prática em programação lógica frequentemente é muito próxima à de linguagens funcionais e esse é o caso na área de transformações de programas. As transformações de *folding/unfolding* foram primeiro adaptadas para a programação lógica por Tamaki e Sato [1]. Mais informações podem ser encontradas em [2].

[1] Beame Paul and Kautz Henry and Sabharwal Ashish, Towards Understanding and Harnessing the Potential of Clause Learning, Journal of Artificial Intelligence Research 22 (2004) 319-351.

[2] Hooker N. John, Integrated Methods for Optimization, Second Edition, Springer Verlag, 2012.

14 Desenvolvimento de Aeronaves por Programação Geométrica

Neste capítulo vamos ver como podemos usar um pouco do que vimos até agora para resolvermos um problema um pouco mais concreto do que os apresentados como exemplos. O problema a ser analisado é o de projeto (*design*) de aeronaves.

Pelas últimas décadas, métodos de otimização têm tomado cada vez maior importância no desenvolvimento de máquinas complicadas, mas ainda com muitas limitações. O problema é que, quando um projeto é tão grande e complicado quanto o do desenvolvimento de aeronaves, que contam com relações altamente não lineares entre muitas variáveis, a aplicação de um método homogêneo, ou mesmo de vários métodos heterogêneos, fica difícil. Não é pouco usual a aplicação de um método de otimização a uma parte específica do projeto (o da asa, por exemplo) e deixá-lo rodando por dias para obter, no final, apenas uma resposta aproximada, que pode ou não ser viável quando se leva em consideração o resto do projeto.

Para estudar como melhor lidar com esse tipo de situação, surgiu o campo de estudos de *Otimização Multidisciplinar de Projetos* (ou *Multidisciplinary Design Optimization*, **MDO** na sigla em inglês, que será usada daqui para frente). **MDO**
O objetivo dos métodos MDO são coordenar de maneira eficiente diferentes métodos de otimização para um projeto único, na esperança de assim obter projetos melhores ("melhores" nos termos definidos pelo ou pela projetista) do que os obtidos otimizando o projeto por partes isoladas. No entanto, mesmo depois de numerosos avanços em métodos MDO, aplicá-los a um projeto inteiro de uma aeronave ainda é impraticável na maior parte dos casos.

Em última instância, a utilização de métodos de otimização para esse tipo de problema (o de projetos como o de uma aeronave, como no exemplo) é feita a fim de se entender e analisar melhor o problema em mãos. Isso porque, no início, o problema a ser resolvido pelo projeto não é, no geral, bem posto (muitas vezes, os objetivos do projeto não são sequer definidos[1]), e é de interesse entender o formato de sua fronteira de Pareto*, a fim de se entender a relevância de cada característica (ou, poderíamos dizer, "de cada variável"). Isso, aliado ao fato de que geralmente um projetista gostaria de otimizar vários parâmetros, e não só um, aumenta a necessidade de se obter não só uma "resposta" otimizada, mas também um maneira de entender a vizinhança dessa resposta (isto é, realizar uma análise de sensibilidade).

Vale também notar que há um impasse entre a realização de uma análise de alta ou de baixa fidelidade. Para análises de alta fidelidade, existem duas opções:

*Uma alocação de recursos (codificada aqui como a atribuição de valores a variáveis) é dita Pareto eficiente se não é possível alterar tal alocação tal que um critério de otimalidade seja melhorado sem que outro seja piorado. Fronteira de Pareto é o conjunto de alocações Pareto eficientes.

- Fazer uma análise de uma parte específica do projeto (como o da asa, ou do motor), o que possibilitaria a quem cuidar dele fazer decisões mais informadas sobre essa parte específica;

- Fazer uma análise mais geral, envolvendo um subconjunto maior de "partes específicas", incorrendo em instâncias mais arbitrárias de problemas mais gerais, e leva, em geral, dias ou semanas para a obtenção de uma solução.

Esses tipos de problemas, assim como a possibilidade de *overfitting* * levam, em vários casos, a uma escolha por uma análise de baixa fidelidade.

Antes de seguirmos um pouco mais a fundo, convém obtermos uma ideia geral do desenvolvimento típico de uma aeronave. Esse desenvolvimento é composto por três etapas[1] (ou, pelo menos, assim os engenheiros o têm tratado por algumas décadas):

- Projetagem conceitual: etapa em que os objetivos do projeto são definidos, assim como requisitos de performance. Dependendo do projeto, esta etapa pode levar vários anos e pode envolver uma análise matemática razoavelmente detalhada;

- Projetagem preliminar: envolve uma análise mais aprofundada da configuração do projeto, identificação e resolução de problemas de interferência aerodinâmica e de instabilidade é realizada e, em um certo ponto, a configuração básica da aeronave é "congelada", de modo a permitir que times que trabalham com subsistemas possam trabalhar independentemente, sem afetar demais os outros times;

- Projetagem detalhada: envolve uma projetagem detalhada do projeto, desenhos em CAD, determinação de passagens hidráulicas, elétricas e de combustível, assim como fabricação de peças de produção†.

Cada uma dessas etapas pode envolver grandes problemas de otimização com restrições, envolvendo as dificuldades notadas acima.

14.1 Programação Geométrica

Nos últimos anos, técnicas de programação convexa têm se tornado cada vez mais eficientes (aproximando mesmo técnicas de programação linear). Apesar disso, por incrível que possa parecer, tais técnicas ainda são notavelmente ausentes em propostas MDO[1] e a abordagem mais comum a esses problemas

*Uma solução "está em" *overfitting* quando levar a excelentes resultados para uma comparativamente pequena quantidade de casos de teste, mas a soluções ruins quando em situações mais gerais.

†Peças que são pensadas para uso em um projeto específico.

é por meio de métodos gerais de programação não-linear. Assim, problemas muito gerais podem ser abordados, mas dificilmente os critérios de confiança e e eficiência são preenchidos.

No lugar disso, então, voltamos nossa atenção a métodos de otimização convexa e, em particular, a um método que se mostrou particularmente adequado para lidar com problemas de projeto de aeronave, o de programação geométrica*. Quando esse método é aplicável, entre as vantagens que ele oferece estão a obtenção de soluções globalmente ótimas (ou um certificado de infactibilidade, quando não existem tais soluções), com tempos de solução comparativamente curtos, que escalam para problemas maiores (isto é, quando o problema aumenta "um pouco", o tempo de solução aumenta só "um pouco"). Ele tem a desvantagem de não ser um método geral, mas como só estamos interessados (no momento) em resolver problemas de projetagem de aeronaves, essa desvantagem não nos entristece muito.

Um problema de programação geométrica em forma padrão pode ser posto da seguinte forma:

$$\begin{aligned} minimize \ & \sum_{i=1}^{n} f_0(x) \\ tal \ que \ & f_i(x) \leq \ 1, i = 1, ..., n, \\ & g_i(x) = 1, i = 1, ..., k, \\ & x \in R^n_+, \end{aligned}$$

em que f_i são posinômios ou monômios, os g_i são monômios e $x \in R^n$. Aqui, a definição de monômios (e de posinômios) é um pouco diferente do usual em álgebra:

- Um **monômio** $g(x)$ é algo da forma $g(x) = kx^\alpha$, $\alpha \in R^n, k \in R^n_+$†; **monômio**
- Um **posinômio** é algo da forma $f(x) = \sum g_i(x)$, em que cada g_i é um monômio (em particular, monômios são posinômios). **posinômio**

Note que esse problema não só é altamente não-linear, mas também é não-convexo. Nós podemos, no entanto, torná-lo convexo, com uma mudança de coordenadas apropriada. Note que, se f é um posinômio em x, $log(f(e^x))$ é convexo e, se g é um monômio em x, $log(g(e^x))$ é, não só convexo, mas também afim. Realizando essa mudança de variáveis, nosso problema de otimização pode ser expresso como

*Diferentemente do que se pode supor, o nome "programação geométrica" não é devido às suas propriedades geométricas no geral, mas sim à desigualdade geométrico-aritmética, que foi muito usada em sua análise.

†Aqui, usamos a convenção de que, se $x \in R^n$ e $\alpha \in R^n$, $x^\alpha := \prod x_i^{\alpha_i}$.

$$
\begin{aligned}
&minimize\ log \sum e^{\alpha'_{0k} x + b_{0k}} \\
&tal\ que\ log \sum e^{\alpha'_{ik} + b_{ik}} \leq 0 \\
&x \in R^n_+.
\end{aligned}
$$

Nessa nova formulação, estamos tomando α como a matriz de vetores que geram as potências dos monômios e posinômios. Note que, fazendo essa mudança de variáveis ($v = log(x)$), monômios se tornam funções afim e, posinômios, funções convexas.

É importante notar que monômios e posinômios são um conjunto fechado sob as operações de divisão por monômios, então desigualdades do tipo $f_1(x) = f_2(x)$, em que f_2 é um monômio e f_1 é um posinômio poderiam ser lidadas fazendo $\frac{f_1(x)}{f_2(x)} = 1$. Ademais, igualdades como $g(x) = 1$, se g é um monômio, podem ser lidadas fazendo $g(x) \leq 1$ e $\frac{1}{g(x)} \leq 1$*. A restrição de que k precisa ser positivo acaba sendo frequentemente satisfeita sem esforços mas, quando esse não é o caso, é preciso usar outro maquinário que não o da programação geométrica. Em particular, programação signomial†, que faz uso de uma estrutura parecida com a da geométrica, resolve esse caso (no geral, de forma mais eficiente do que um método de programação não linear genérico).

14.2 Um modelo simples de aeronave

A seguir é apresentado um modelo de aeronave simplificado, em grande parte baseado no modelo da tese de Hoburg[1]. A modelagem do problema será feita com base no pacote GPkit[2].

14.2.1 Modelo de peso e sustentação

Assumimos que a aeronave esteja em vôo de cruzeiro‡, de modo que o sustentação seja igual ao arrasto e a sustentação igual ao peso. O peso da aeronave é a soma do peso de carga útil P_0, o da asa P_a e do combustível P_c:

$$P \geq P_0 + P_a + P_c$$

Usamos o modelo abaixo para vôo estável§:

$$P_0 + P_a + \frac{P_c}{2} \leq \frac{1}{2}\rho S C_l V^2,$$

*Em particular, isso mostra que um programa geométrico qualquer pode ser expresso com desigualdades do tipo $f_i(x) \leq 1$, o que é a entrada que alguns *solvers* requerem por padrão.

†Um posinômio é um signômio que é restrito a ter coeficientes positivos. Programação signomial é a cujas restrições são signômios.

‡Quando as velocidades linear a angular da aeronave são constantes em um sistema de referência fixado ao corpo. É o mesmo que "vôo estável".

§O "l" abaixo vem de "*lift*", "sustentação" em inglês.

em que a sustentação da aeronave é igual ao peso dela com metade de combustível, C_l é o coeficiente de sustentação e V a velocidade do avião e ρ é uma constante física.

Aqui usamos a equação para arrasto:

$$Arrasto = D := \frac{1}{2}\rho S C_d V^2,$$

em que $\frac{1}{2}\rho V^2$ representa a pressão dinâmica, S corresponde à área da asa e C_d ao coeficiente de arrasto total[*], dado por:

$$C_{d_{fuse}} = \frac{CDA_0}{S} + kC_f\frac{S_{wet}}{S} + \frac{C_l^2}{\pi Ae},$$

em que CDA_0 corresponde à área de arrasto da fuselagem[†], k é um fator de forma que leva em conta arrasto de pressão, $\frac{S_{wet}}{S}$ é a razão de área molhada[‡] e e é o fator de eficiência de Oswald. CDA_0 é linearmente proporcional ao volume de combustível na fuselagem[§]:

$$V_{f_{fase}} \leq CDA_0 \times 10[metros].$$

Assumindo um fluxo de Blasius turbulento sobre uma placa plana, o coeficiente de fricção C_f pode ser aproximado por:

$$C_f = \frac{0,074}{Re^{0,2}},$$

em que $Re = \frac{\rho V}{\mu}\sqrt{\frac{S}{A}}$ é o número de Reynolds na corda média $\sqrt{\frac{S}{A}}$[¶].

Também gostaríamos que uma aeronave cheia de combustível seja capaz de voar a uma velocidade mínima:

$$P \leq \frac{1}{2}\rho V_{min}^2 S C_{l_{max}},$$

em que $C_{l_{max}}$ é o coeficiente de sustentação máximo.

Uma medida de performance útil, o tempo de vôo, é dada pela distância percorrida sobre velocidade:

$$T_{vôo} \geq \frac{Distância}{V}.$$

---

[*] O "d" vem de "*drag*", arrasto em inglês.

[†] Definida como a área normal ao fluxo de ar que geraria o mesmo arrasto que o avião.

[‡] A área em contato com o fluxo de ar.

[§] GPkit faz checagem de unidades, então precisamos corrigi-la explicitamente, do que o "[metros]" constitui um lembrete.

[¶] A representa a razão de aspecto, ie, a razão entre a distância entre as pontas das asas e a corda média

A razão de força de sustentação/arrasto é dada por:

$$\frac{L}{D} = \frac{C_l}{C_d},$$

em que C_l é o coeficiente de sustentação.

Essas restrições se traduzem como

```
# Modelo de peso e empuxo
restricoes += [P >= P_0 + P_a + P_f,
               P_0 + P_a + 0.5 * P_f <=
               0.5 * rho * S * C_l * V ** 2,
               P <= 0.5 * rho * S * C_lmax * V_min ** 2,
               T_voo >= Dist / V,
               LoD == C_L/C_D]
```

Assumimos que o Consumo Específico de Combustível por Unidade de Empuxo (CEUE) é constante e que ele provê tanto impulso quanto necessário. Como a força de impulso F é maior ou igual que a força de arrasto D, obtemos

$$P_f \geq CEUE \times T_{vôo} \times D.$$

E podemos completar nosso modelo da seguinte forma:

```
# Modelo de Impulso e Arrasto
C_D_fuse = CDA0 / S
C_D_wpar = k * C_f * S_wetratio
C_D_ind  = C_l ** 2 / (np.pi * A * e)
restricoes += [P_f >= CEUE * T_voo * D,
               D >= 0.5 * rho * S * C_D * V ** 2,
               C_D >= C_D_fuse + C_D_wpar + C_D_ind,
               V_f_fuse <= 10*units('m')*CDA0,
               Re <= (rho / mu) * V * (S / A) ** 0.5,
               C_f >= 0.074 / Re ** 0.2]
```

14.2.2 Volume do combustível

Definindo o volume requerido em função da densidade do combustível ρf, temos

$$V_c = \frac{P_c}{\rho_f g}.$$

Pelo modelo aerodinâmico, temos que

$$V_{c_{asa}}^2 \leq 0,0009 \frac{S\tau^2}{A \times Distância}.$$

O volume de combustível disponível $V_{c_{disp}}$ é limitado superiormente pelos volumes disponíveis na asa e na fuselagem:

$$V_{c_{disp}} \leq V_{c_{asa}} + V_{c_{fuse}}.$$

Note que, devido à restrição acima, o programa resultante será signomial.

Restringimos o volume total de combustível como menor do que o volume disponível:

$$V_{c_{disp}} \geq V_c.$$

```
# Modelo de Volume de Combustível
with SignomialsEnabled():
    restricoes +=[V_c == P_c / g / rho_c,
                  V_c_wing**2 <= 0.0009*S**3/A*tau**2,
                  # linear em b e tau, quadratico em chord
                  V_c_disp <= V_c_wing + V_c_fuse, #[SP]
                  V_c_disp >= V_c]
```

14.2.3 Acumulação de peso nas asas

O peso na superfície da asa é uma função da área da asa

$$P_{p_{surp}} \geq P_{p_{coef2}} S.$$

O peso estrutural na asa é uma expressão posinomial que leva em conta o alívio de cisalhamento devido à presença de combustível nas asas e o momento flexor:

$$P^2_{p_{strc}} \geq \frac{P^2_{p_{coeff1}}}{\tau^2}(N^2_{ult} AR^3((P_0 + \rho_f g V_{c_{fuse}})PS)).$$

O peso total da asa é limitado:

$$P_p \geq P_{p_{surp}} + P_{p_{strc}}.$$

```
# Modelo de Peso nas Asas
restricoes += [P_w_surf >= P_P_coeff2 * S,
               P_w_strc**2. >= P_P_coeff1**2./ tau**2. *
               (N_ult**2. * A ** 3. * ((P_0+V_f_fuse*g*rho_f) * P * S)),
               P_w >= P_w_surf + P_w_strc]

return restricoes
```

14.2.4 Possíveis objetivos

Dadas essas restrições, algumas potenciais funções objetivo (a serem minimizadas) com as quais poderíamos querer trabalhar são:

- P_c, o peso do combustível (o objetivo padrão);
- P, o peso total da aeronave;
- $\frac{P_c}{T_{vôo}}$, o peso do combustível dividido pelo tempo de vôo (podemos pensar nisso como uma medida de eficiência no uso do combustível);
- $P_c + c \times T_{vôo}$, uma combinação linear antre peso do combustível e tempo de vôo.

14.3 Modelo de asa

Como um exemplo simples de aplicação de programação geométrica, é apresentado a seguir um projeto de asa simples, tirado de [4]. As fórmulas usadas aqui são semelhantes às vistas na seção anterior. Queremos dimensionar uma asa de área total S, envergadura b e alongamento $A = \frac{b^2}{S}$. Gostaríamos de escolher esses parâmetros de forma a minimizar o arrasto total, $D = \frac{1}{2}\rho V^2 C_D S$. O coeficiente de arrasto é modelado como a soma de arrasto parasita da fuselagem, arrasto parasita da asa e arrasto induzido,

$$C_D = \frac{CDA_0}{S} + kC_f\frac{S_{wet}}{S} + \frac{C_L^2}{\pi Ae},$$

em que CDA_0 representa a área de arrasto da fuselagem, k é o fator de forma, que leva em conta o arrasto de pressão, S_wet/S é a razão de área molhada e e é o fator de eficiência de Oswald.

O coeficiente de fricção C_f é aproximado por:

$$C_f = \frac{0,074}{Re^{0,2}},$$

em que $Re = \frac{\rho V}{\mu}\sqrt{\frac{S}{A}}$. O peso total da aeronave é modelado como a soma de um peso fixo W_0 e o peso da asa W_w, $W = W_0 + W_w$. O peso da asa é modelado como:

$$W_w = 45,42S + 8,71 \times 10^{-5}\frac{N_{lift}b^3\sqrt{W_0 W}}{S\tau}m$$

em que N_{lift} é o fator de carga para dimensionamento estrutural e τ é a razão de espessura-para-corda do aerofólio.

Tomamos também que

$$W = \frac{1}{2}\rho V^2 C_L S$$

Além disso, a aeronave deve ser capaz de voar a uma velocidade mínima V_{min}

$$\frac{2W}{\rho V_{min}^2 S} \leq C_{L,max}.$$

Note que as restrições acima são expressões monomiais e posinomiais, de modo que podemos lidar com este modelo como um problema de programação geométrica.

Resolvendo o problema com o código no final da seção, obtemos um custo ótimo de 303,1 e os seguintes valores para as variáveis livres:

A	:	8,46	Alongamento
C_D	:	0,02059	Coeficiente de arrasto da asa
C_L	:	0,4988	Coeficiente de sustentação da asa
C_f	:	0,003599	Coeficiente de fricção
D	:	303,1 (N)	Força de arrasto total
Re	:	3,675e+06	Número de Reynold
S	:	16,44 (m^2)	Área total da asa
V	:	38,15 (m/s)	Velocidade de cruzeiro
W	:	7341 (N)	Peso total da aeronave
W_w	:	2401 (N)	Peso da asa

É importante para um projetista ter também ideia da sensibilidade de algumas variáveis (isto é, mudando um pouco o valor de uma variável, é de interesse saber quanto o valor da solução muda, se muito ou pouco). Os valores a seguir buscam refletir isso (os valores são referentes à derivada logarítmica $\frac{d(log(y)}{d(log(x)}$):

W_0	:	+1	Peso da aeronave sem asa
e	:	-0,48	Fator de eficiência de Oswald
$(\frac{S}{S_{wet}})$	:	+0,43	Razão de área molhada
k	:	+0,43	Fator de forma
V_{min}	:	-0,37	Velocidade de decolagem

Frequentemente o conhecimento da fronteira de Pareto é de grande importância. Para obtermos algum conhecimento sobre ela, podemos atribuir valores a algumas variáveis e checar como as soluções se comportam para tais valores. Fazemos isso com as variáveis V e V_{min}:

V	:	45	45	55	55	m/s	velocidade de cruzeiro
V_{min}	:	20	25	20	25	m/s	velocidade de decolagem

Obtemos, assim, os respectivos custos:

338 294 396 326

Os valores das variáveis livres são os seguintes:

A	:	6,2	8,84	4,77	7,16		Alongamento
C_D	:	0,0146	0,0196	0,0123	0,0157		Coeficiente de arrasto da asa
C_L	:	0,296	0,463	0,198	0,31		Lift coefficent of wing
C_f	:	0,00333	0,00361	0,00314	0,00342		Coeficiente de fricção
D	:	338	294	396	326	N	Força de arrasto total
Re	:	5,38e+06	3,63e+06	7,24e+06	4,75e+06		Número de Reynolds
S	:	18,6	12,1	17,3	11,2	m^2	Área total da asa
W	:	6,85e+03	6,97e+03	6,4e+03	6,44e+03	N	Peso total da aeronave
W_w	:	1,91e+03	2,03e+03	1,46e+03	1,5e+03	N	Peso da asa

As variáveis mais sensíveis, como apresentado pelo modelo (veja [2] para mais detalhes), são:

W_0	:	+0,92	+0,95	+0,85	+0,85	Peso da aeronave sem a asa
V_{min}	:	-0,82	-0,41	-1	-0,71	Velocidade de decolagem
V	:	+0,59	+0,25	+0,97	+0,75	Velocidade de cruzeiro
$(\frac{S}{S_{wet}})$	:	+0,56	+0,45	+0,63	+0,54	Razão de área molhada
k	:	+0,56	+0,45	+0,63	+0,54	Fator de forma

Programa

O programa usado para obter os dados acima é o seguinte:

```
#!/usr/bin/env python
#
# Based on program from
#   https://gpkit.readthedocs.io/en/latest/examples.html#simple-wing
#

import cPickle as pickle
import numpy as np
from gpkit import Variable, Model
pi = np.pi


# Constants
k = Variable("k", 1.2, "-", "form factor")
e = Variable("e", 0.95, "-", "Oswald efficiency factor")
mu = Variable("\\mu", 1.78e-5, "kg/m/s", "viscosity of air")
rho = Variable("\\rho", 1.23, "kg/m^3", "density of air")
tau = Variable("\\tau", 0.12, "-", "airfoil thickness to chord ratio")
N_ult = Variable("N_{ult}", 3.8, "-", "ultimate load factor")
V_min = Variable("V_{min}", 22, "m/s", "takeoff speed")
C_Lmax = Variable("C_{L,max}", 1.5, "-", "max CL with flaps down")
S_wetratio = Variable("(\\frac{S}{S_{wet}})", 2.05, "-", "wetted area ratio")
W_W_coeff1 = Variable("W_{W_{coeff1}}", 8.71e-5, "1/m",
                      "Wing Weight Coefficent 1")
W_W_coeff2 = Variable("W_{W_{coeff2}}", 45.24, "Pa",
                      "Wing Weight Coefficent 2")
CDA0 = Variable("(CDA0)", 0.031, "m^2", "fuselage drag area")
W_0 = Variable("W_0", 4940.0, "N", "aircraft weight excluding wing")

# Free Variables
D = Variable("D", "N", "total drag force")
A = Variable("A", "-", "aspect ratio")
S = Variable("S", "m^2", "total wing area")
V = Variable("V", "m/s", "cruising speed")
W = Variable("W", "N", "total aircraft weight")
Re = Variable("Re", "-", "Reynold's number")
C_D = Variable("C_D", "-", "Drag coefficient of wing")
C_L = Variable("C_L", "-", "Lift coefficent of wing")
C_f = Variable("C_f", "-", "skin friction coefficient")
W_w = Variable("W_w", "N", "wing weight")

constraints = []
```

```
# Drag model
C_D_fuse = CDA0/S
C_D_wpar = k*C_f*S_wetratio
C_D_ind = C_L**2/(pi*A*e)
constraints += [C_D >= C_D_fuse + C_D_wpar + C_D_ind]

# Wing weight model
W_w_strc = W_W_coeff1*(N_ult*A**1.5*(W_0*W*S)**0.5)/tau
W_w_surf = W_W_coeff2 * S
constraints += [W_w >= W_w_surf + W_w_strc]

# and the rest of the models
constraints += [D >= 0.5*rho*S*C_D*V**2,
                Re <= (rho/mu)*V*(S/A)**0.5,
                C_f >= 0.074/Re**0.2,
                W <= 0.5*rho*S*C_L*V**2,
                W <= 0.5*rho*S*C_Lmax*V_min**2,
                W >= W_0 + W_w]

print("SINGLE\n======")
m = Model(D, constraints)
sol = m.solve(verbosity=0)
print(sol.summary())
# save solution to a file and retrieve it
# sol.save("solution.p")
# print(sol.diff("solution.p"))

print("SWEEP\n=====")
N = 2
sweeps = {V_min: ("sweep", np.linspace(20, 25, N)),
          V: ("sweep", np.linspace(45, 55, N)), }
m.substitutions.update(sweeps)
sweepsol = m.solve(verbosity=0)
print(sweepsol.summary())
# sol_loaded = pickle.load(open("solution.p"))
# print(sweepsol.diff(sol_loaded))
```

Leituras Adicionais

A pesquisa e prática em programação lógica frequentemente é muito próxima à de linguagens funcionais e esse é o caso na área de transformações de programas. As transformações de *folding/unfolding* foram primeiro adaptadas para a programação lógica por Tamaki e Sato [1]. Mais informações podem ser encontradas em [2].

[1] Hoburg, W. Warren, "Aircraft Design Optimization as a Geometric Program" 2013.

[2] E. Burnell and W. Hoburg, "Gpkit software for geometric programming." https://github.com/convexengineering/gpkit, 2018. Version 0.7.0.

[3] B. Ozturk, "SimPleAC" https://github.com/convexengineering/gplibrary/blob/master/gpkitmodels/SP/SimPleAC/simpleac.pdf, 2018.

[4] Hoburg, W. Warren, "Geometric Program for Aircraft Design Optimization" 2014.

15 Programação Relacional em Scheme

Uma pessoa só tem certeza
daquilo que constrói

Giambattista Vico -
Historiador italiano, século XVIII

Neste capítulo, veremos uma linguagem de programação lógica com um sabor diferente do Prolog que vimos no início, chamada *miniKanren*, e veremos como implementá-la, por meio da linguagem *Scheme*.

Usaremos *Scheme* porque é uma linguagem que é pequena, o que significa que não será gasto muito esforço para apresentar o seu funcionamento. E é, ao mesmo tempo, poderosa, o que significa que não precisaremos de muito código para fazer o que nos propomos.

Esse capítulo tem um sabor diferente dos demais. Essa diferença pode ser visto rapidamente, pela cara do código, mas também de várias outras formas. Redefiniremos alguns termos usados anteriormente. Essas redefinições terão semelhanças e diferenças às definições originais, mas escolhemos não notar essas diferenças aqui, por acreditarmos serem claras o suficiente. Outro ponto que vale nota é que, aqui, buscamos apenas um maior entendimento e, para tanto, tentamos deixar o código como "implementado em primeiros princípios", evitando detalhes que poderiam deixá-lo mais eficiente, mas elevando os prerrequisitos para a compreensão do código, ou exigindo uma discussão mais completa sobre ele (o que preferimos evitar).

15.1 Introdução ao Scheme

Scheme é um *Lisp*[*]. O termo *Lisp* é às vezes usado para ser referir a uma linguagem de programação, mas o mais correto seria ser referida como a uma família de linguagens (de fato, dezenas de linguagens), todas com algumas características em comum, notadamente:

- São linguagens multi-paradigma, mas como um foco no paradigma de programação funcional, isto é, funções são "cidadãos de primeira classe";
- "Todo"[†] código *Lisp* (que não tenha erro de sintaxe) é avaliado para algum valor;
- Programas são expressos em notação polonêsa, em formato de listas[‡](listas

[*] De *LISt Processing*
[†] Com uma excessão bem natural, que será mencionada mais adiante.
[‡] Também conhecidas como *S-Expressions*, ou *Sexps*

em *Lisps* são delimitadas por parênteses)*:
`(+ 1 2)` ⇒3;

- O que nos leva a outro ponto: não existe diferença sintática entre a forma como programas *Lisp* são escritos e a forma como suas estruturas de dados são representadas. Diz-se, assim, que *Lisps* são homoicônicas (vale dizer, Prolog também é uma língua homoicônica), o que significa que a diferença entre dados e programa é "borrada", e programas podem, e frequentemente são, manipulados livremente.

Agora, vamos rapidamente introduzir a sintaxe principal de *Scheme*, com alguns exemplos†:

- Listas são representadas como `(`el_1 el_2 ... el_n`)`, em que el_i é o iésimo elemento da lista. No entanto, se escrevêssemos uma lista simplesmente assim, a lista seria confundida com uma aplicação de função (aplicação da função el_1 aos argumentos el_2 a el_n), então, para fins de desambiguação, é usada uma aspa simples, e a lista é escrita como $'(el_1 el_2 ... el_n)$, que é equivalente a `(list` $'el_1$ ... $'el_n$`)` . Essa aspa simples também pode ser usada para "evitar que um objeto seja avaliado"‡:
  A linha
  `a` ⇒erro
  resulta em erro se `a` não for uma variável, já que tentará ser avaliada, sendo que `a` não tem valor associado, mas a linha
  $'a \Rightarrow 'a$
  é avaliada como ela mesma.

- Uma estrutura de dados mais geral do que lista em *Scheme* é o que é chamado *cons pair* (que nós chamaremos daqui para frente simplesmente de "par").. A lista `'(a b c d)` é equivalente a `(cons a (cons b (cons c (cons d '()))))`, em que `'()` é a lista vazia Assim *cons* constrói uma estrutura de dados formada por um par. Para obter o primeiro elemento do par, usa-se *car* e, para obter o segundo, *cdr*§. Temos, por exemplo,
  `(car (cons 1 (cons 2 3)))` ⇒1
  `(cdr (cons 1 (cons 2 3)))` ⇒`(cons 2 3)` = `'(2 . 3)`
  `(car '(1 2 3 4))` ⇒1
  `(cdr '(1 2 3 4))` ⇒`'(2 3 4)`

*Usaremos daqui em diante a notação **codigo** ⇒**valor** para denotar que o código **código** avalia para o valor **valor**.

†Note que só introduziremos a parte da linguagem que nos será relevante, o que não é a linguagem toda.

‡Uma colocação mais correta seria "tornar objetos auto-avaliantes", mas não precisamos entrar em muitos detalhes de como isso funciona. Para nós é suficiente dizer que 'a é um símbolo.

§Esses nomes têm uma motivação histórica: eram nomes de registradores quando os primeiros *Lisps* estavam sendo criados.

Com isso, podemos definir um lista indutivamente como sendo ou a lista vazia, `'()`, ou um par, cujo *cdr* é uma lista*.

- Para definir funções, use **lambda**, ou λ†:
  `((λ (a b) (/ a b)) 1 3)` ⇒1/3
- Para definir constantes, use **define**:
  `(define divide (λ (a b) (/ 1 3)))`‡
  `(divide 1 3)` ⇒1/3
  `(define c (divide 1 3))`
  `(divide (divide 1 9) c)` ⇒1/3
- Um ponto importante, que usaremos muito logo mais é que, se quisermos criar listas com os valores das variáveis, no lugar de nomes simbólicos, podemos usar, no lugar da aspa simples a crase e preceder o nome da variável com uma vírgula:
  `(define x 10)`
  `‘(1 2 'x ,x)` ⇒`'(1 2 'x 10)`
- Para realizar execuções condicionais, use **cond**:

```
(cond
  ((< 1 0) (+ 3 4))
  ((< 0 1) (- 3 4))
  (else 0))

⇒-1
```

  Podem ser adicionadas quantas cláusulas do tipo `((condicao)(efeito))` se quiser (vale notar que elas são avaliadas sequencialmente), sendo que a última pode opcionalmente ser como `(else (efeito))`, ou `(#t (efeito))`§.
- Para adicionar variáveis locais, use *let*:

---

*Note que `(cons 2 3)`, por exemplo, não é uma lista. Esse tipo de estrutura é chamada *dotted list*, porque, para distingui-la de uma lista, é costumeiramente impressa como `'(2 . 3)`, mas, assim como com Prolog, é uma estrutura de dados diferente, que tem o nome *dotted **list*** como que por uma aparência acidental.

†Editores de texto atuais aceitam os dois tipos de entrada, mas optamos por usar λ. Esse uso do símbolo tem a seguinte origem: Bertrand Russel e Alfred Whitehead buscaram, no início do século XX lançar as bases lógicas da matemática eu seu trabalho *Principia Mathematica*. Lá, para denotar que uma variável é livre, ela recebia chapéu, como em *â(a + y)*. Mais tarde, Alonzo Church achou que seria mais conveniente ter fórmulas crescendo linearmente na horizontal, então decidiu mover o chapéu para o lado, obtendo *ˆa(a + y)*. Mas o chapéu flutuando parece engraçado, então Church o trocou pelo o símbolo não usado mais próximo que tinha, um Λ, como em *Λa(a + y)*. Mas Λ tem uma grafia muito parecida com outra letra comum, então ele acabou eventualmente trocando para λ em sua teoria, que acabou se chamando *cálculo λ* [2].

‡Um *açucar sintático* para essa construção é `(define (divide a b) (/ 1 3))`.

§Preferiremos a segunda opção. O *#t* é de *true* e existe, analogamente, o *#f*, de *false*.

```
(let ((a (+ 3 4))
      (b (cons 1 2)))
  (+ a (car b)))

⇒8
```

O *let* tem duas partes, a de definições, da forma `((variavel valor)(variavel valor) ... (variavel valor))`*. em seguida, a parte de valor, que nos dá o valor que *let* assume.

Como exemplo, o seguinte programa calcula o comprimento de uma lista (lembre que, como

Dada essa introdução, faremos uso dessas e outras construções da linguagem sem maiores comentários (exceto quando uma construção especialmente difícil ou complexa o justificar). Para uma introdução mais compreensiva à linguagem, veja [1].

15.2 A linguagem miniKanren

Nosso objetivo aqui é implementar miniKanren, uma linguagem de programação relacional. No lugar de descrever toda a linguagem e depois implementá-la, seguimos pelo caminho de mostrar um pequeno exemplo do que esperamos conseguir e, então, implementamos a linguagem. A esperança é que essa abordagem ofereça maior entendimento dos conceitos explorados.

O tipo de coisa que queremos poder fazer com miniKanren é como o seguinte†:

```
(defrel (teacup^o t)
  (disj_2 (≡ 'tea t) (≡ 'cup t)))
(run* x
  (teacup^o x))

⇒'(tea cup)
```

*Podem ser adicionadas quantas variáveis se quiser. As atribuições são feitas "paralelamente" (o que significa que a atribuição de valor a uma variável não influi no da outra, o que pode ser feito de forma paralela, ou não).

†Usaremos as convenções da literatura de usar subescritos e sobrescritos e símbolos matemáticos, como ≡, para representar as relações, na esperança de que isso clarifique a notação e deixe o texto menos pesado. Em particular, para diferenciar um objeto relacional de um funcional, o relacional terá um "o" sobrescrito (como em $relacao^o$). Ao escrever o programa para o computador ler, os sobrescritos e subescritos que forem alfa-numéricos ou "*" podem ser escritos normalmente, na frente do termo (como em *relacaoo*, ou $run*$). O símbolo ≡ é escrito "==" e, termos como $termo^\infty$, "$termo-inf$". Ademais, #u e #s devem ser trocados por *fail* e *succeed.*, respectivamente.

Veremos mais exemplos enquanto o construímos. A construção a seguir é em grande parte baseada em [3]. Para conferir detalhes de implementações completas, veja [4].

Como visto no primeiro exemplo, não seguimos, como em Prolog, uma convenção de nomeação de variáveis (em Prolog, a convenção era de que variáveis são capitalizadas). Assim, precisamos de algo para discerni-las e, para tal fim, o que usamos é o *fresh*, informando que a variável é "fresca".

Lembre-se que uma variável relacional não é a a mesma coisa que uma variável em uma linguagem de programação tradicional (não relacional). Para uma variável única, usamos*

```
(define (var name)(vector name))
```

usamos também†

```
(define (var? name)(vector? name))
```

Para evitar problemas como os de colisão de variáveis, as variáveis são locais, assim como em *Scheme*. Precisamos, então, de uma forma de modelar isso (note que a definição acima não reflete isso) e, o que usamos é o seguinte:

```
(define (call/fresh name f)
  (f (var name)))
```

Essa função espera, como segundo argumento, uma expressão λ, que recebe como argumento uma variável e produz como valor um goal, que tem acesso à variável criada.

Precisamos, agora, saber como associar um valor a uma variável. Diremos que o par `‘(,z . a)` é uma associação de a à variável z. Mais em geral, um par é uma associação quando o seu *car* é uma variável.

A uma lista de associações chamaremos *substituição*. Na substituição `‘((,x . ,z))`, a variável x é "fundida" (ou, na nossa linguagem anterior, unificada) à variável z. A substituição vazia é simplesmente uma lista vazia: `(define empty-s '())`. Nem toda lista de associações é uma substituição, no entanto. Isso porque, não aceitamos, em nossas substituições, associações com o mesmo *car*. Então, o seguinte não é uma substituição: `‘((,z . a)(,x . c)(,z . b))`.

Precisamos de dois procedimentos importantes para lidar com substituições: um para extênde-las e um para obter o valor de uma variável presente nela.

Para obter o valor associado a x, usamos *walk*, que deve se comportar como o seguinte:

---

*Usamos *vector* para que a unicidade da variável seja definida por sua posição de memória. Outra opção, seria distingui-las por valor, se nos assegurássemos que seu valor é único.

†Símbolos como "?" podem ser usados no meio do código asssim como outros, tais como "a" ou "b".

```
(walk y
  `((,z . a)(,x . ,w)(,y . ,z)))
```

$\Rightarrow a$

porque y está fundido a z, que está associado a a. O *walk* é como se segue*:

```
(define (walk v s)
  (let ((a (and (var? v)(assv v s))))
    (cond
      ((pair? a)(walk (cdr a) s))
      (else v))))
```

Esse código faz uso de *assv*, que ou produz $\#f$, se não há associação cujo *car* seja v na substituição s, ou produz o *cdr* de tal associação. Perceba que, se *walk* produz uma variável como valor, ela é necessariamente fresca (isto é, que não foi associada).

Para extender uma substituição, usamos *ext-s*, que faz uso de *occurs?*:

```
(define (ext-s x v s)
  (cond
    ((occurs? x v s) #f)
    (else (cons `(,x . ,v) s))))

(define (occurs? x v s)
  (let ((v (walk v s)))
    (cond
      ((var? v) (eqv? v x))
      ((pair? v)
        (or (occurs? x (car v) s)
            (occurs? x (cdr v) s)))
      (else #f))))
```

Esse *occurs?* realiza o "teste de ocorrência" (aquele que, como mencionamos anteriormenet, não é feito por padrão no Prolog por razões de eficiência, e que faz com que substituições do tipo `‘((,y . (,x))(,x . (a . ,y))` sejam inválidas).

Com isso em mãos, podemos definir nosso unificador:

```
(define (unify u v s)
  (let ((u (walk u s))(v (walk v s)))
    (cond
      ((eqv? u v) s)
      ((var? u) (ext-s u v s))
      ((var? v) (ext-s v u s))
```

*Lembre-se que `(and a b)` $\Rightarrow b$, se $a \neq \#f$.

```
((and (pair? u) (pair? v))
 (let ((s (unify (car u)(car v) s)))
   (and s
     (unify (cdr u)(cdr v) s))))
(else #f))))
```

15.3 Streams

Antes de continuarmos, precisamos tocar no modelo de avaliação de Scheme. Scheme é uma linguagem de "ordem aplicativa", o que significa que os argumentos de uma função são todos avaliados no momento em que a função é aplicada. Por esse motivo, os *and* e *or* usuais, por exemplo, não podem ser funções em Scheme. Uma alternativa à ordem aplicativa é a "ordem normal", que algumas outras linguagens de programação funcional adotaram. Linguagens de ordem normal só avaliam o argumento de uma função quando esse argumento for usada, atrasando a avaliação do mesmo (no que é chamado "avaliação tardia", ou "avaliação preguiçosa").

Avaliação preguiçosa é conveniente para diversas operações e pode ser emulada em linguagens de programação funcional de ordem aplicativa pelo uso de **streams**. *Streams* são definidos como sendo ou a lista vazia, ou um par cujo *cdr* é um *stream*, ou uma suspenção. **streams**

Uma **suspenção** é uma função formada por `(λ () corpo)`, em que `((λ () corpo))` é uma *stream*. Agora, se definirmos **suspenção**

```
(define (≡ u v)
  (λ (s)
    (let ((s (unify u v s)))
      (if s `(,s) '()))))
```

≡ produz um goal. Dois outros goals, *sucesso* e *falha*, são denotamos #s e #u:

```
(define #s
  (λ (s)
    `(,s)))

(define #u
  (λ (s)
    '()))
```

Um goal é uma função que recebe uma substituição e que, se retorna, retorna uma *stream* de substituições.

Como um exemplo, temos que `((≡x y) empty-s)` ⇒`(((,x . ,y)))`, uma lista com uma substituição (com uma associação).

Ao lidar com *Streams*, precisamos de funções especiais, já que não são simples listas. *Streams* são uteis para a representação de estruturas de dados infinitas, então, por isso, funções e variáveis para lidar com elas terão um ∞ sobrescrito, para diferenciá-las das funções para listas comuns. Assim, podemos definir $append^{\infty}$:

```
(define (append∞ s∞ t∞)
  (cond
    ((null? s∞) t∞)
    ((pair? s∞)
      (cons (car s∞
        (append∞ (cdr s∞) t∞))))
    (else (λ ()
            (append∞ t∞ (s∞))))))
```

Note que, na suspensção, a ordem dos argumentos é trocada. Com essa função, podemos fazer

```
(define (disj2 g1 g2)
  (λ (s)
    (append∞ (g1 s) (g2 s))))
```

em que $disj_2$ é uma disjunção (um *ou* lógico).

Veja agora a seguinte definição:

```
(define (never^o)
  (λ (s)
    (λ ()
      ((never^o) s))))
```

Esse é um goal que não sucede nem falha. Para entender porque a ordem dos argumentos é trocada na suspenção de $append^{\infty}$, compare o valor de s^{∞} em

```
(let ((s∞ ((disj2
                    (never^o)
                    (≡ 'olive x))
                    empty−s)))
  s∞)
```

com o valor de s^{∞} em

```
(let ((s∞ ((disj2
                    (≡ 'olive x))
```

```
                    (never^o)
                    empty-s)))
  s^∞)
```

Em contraste com $never^o$, aqui está $always^o$:

```
(define (always^o)
  (λ (s)
    (λ ()
      ((disj_2 #s (always^o)) s))))
```

Antes de continuar, será útil conhecer a função *map*:
`(map f '(`el_1 ... el_n`))` ⇒`'((f` e_1`) ... (f` el_n`))`

A lista construída por *map* é constrúida por *cons*. Mas existe também *map-append*, análoga a *map*, mas em que a lista resultante é construída por *append*. Usaremos um *append-map*, mas para *streams*, isto é, um $append - map^\infty$:

```
(define (append-map^∞ g s^∞)
  (cond
    ((null? s^∞) '())
    ((pair? s^∞)
      (append^∞ (g (car s^∞))
        (append-map^∞ g (cdr s^∞))))
  (else (λ ()
          (append-map^∞ g (s^∞))))))
```

Com isso, podemos definir a conjunção de dois goals, assim como definimos a disjunção:

```
(define (conj_2 g_1 g_2)
  (λ (s)
    (append-map^∞ g_2 (g_1 s))))
```

15.3.1 Voltando ao problema das variáveis

"Reificação" é um termo muito usado em contextos de programação relacional, assim como nos da teoria de Marx, e com significado semelhante. Notadamente, é "o ato de tratar algo abstrato como algo concreto, ou algo concreto como algo abstrato". No nosso contexto, pode-se argumentar que uma variável é algo "concreto", na medida que tem algum significado para quem programa. Apesar disso, temos a liberdade de, quando conveniente, tratá-la como algo abstrato, um *place-holder*, e, quando não apresentar valor, podemos denotá-la

simplesmente como um "_" seguido de um número, que serve para diferi-lo de outras variáveis*.

Isso é útil, por exemplo, para mostrarmos os valores das variáveis (e, quando a variável for fresca, podemos simplesmente mostrá-la como "$_n$"). Mais em geral, quando uma variável é criada fresca, não tem valor. Na verdade, "não ter valor" significa que a variável é fresca, e a forma de representarmos isso convenientemente é reificando-a. Para realizar isso, precisamos primeiro do *reify-name*†:

```
(define (reify-name n)
  (string → symbol
    (string-append "_"
      (number → string n))))
```

Com *reify-name*, podemos criar o *reify-s*, que recebe uma variável e uma substituição, inicialmente vazia, r:

```
(define (reify-s v r)
  (let ((v (walk v r)))
    (cond
      ((var? v)
       (let ((n (length r)))
         (let ((rn (reify-name n)))
           (cons `(,v . ,rn) r))))
      ((pair? v)
       (let ((r (reify-s (car v) r)))
         (reify-s (cdr v) r)))
      (else r))))
```

Aqui usamos *length* para produzir um número único em cada uso de *reify-name*.

Para continuar com nosso esquema de reificação, vamos precisar de uma versão ligeiramente diferente de *walk*, o qual chamaremos *walk**‡. Veja a definição:

```
(define (walk* v s)
  (let (v (walk v s))
    (cond
      ((var? v) v)
      ((pair? v)
        (cons
          (walk* (car v) s)
          (walk* (cdr v) s)))
```

*Nos programas, usaremos $_n$, em que n é um número natural.

†*string → symbol* é escrito *string− > symbol*.

‡Leia *walk star*.

```
      (else v))))
```

Note que *walk* e $walk^*$ só diferem se $walk^*$ *caminhar* a um par que contém alguma variável com associação na substituição s. Além disso, note que, se $walk^*$ produz um valor v ao *caminhar* por uma substituição s, temos garantia de que as variáveis em v são frescas.

Com isso em mãos, podemos substituir cada variável fresca por sua reificação, com

```
(define (reify s)
  (λ (s)
    (let ((v (walk* v s)))
      (let ((r (reify-s v empty-s)))
        (walk* v r)))))
```

15.4 Finalizando

Na seção anterior, definimos a coluna dorsal do miniKanren. Para fechar precisaremos usar as macros do *Scheme*. A palavra "macro", no geral, é usada (neste contexto de programação) para se referir a código que "escreve código", isto é, que realiza transformações no código a ser "enviado"ao compilador ou interpretador. Várias linguagens têm macros de tipos diferentes, mas poucas são tão poderosas como as macros que costumam estar presentes em linguagens Lisp. *Scheme* não tem na própria linguagem mecanismos de iteração, por exemplo (como um laço *for*, ou *while*), mas esses (assim como vários outros mecanismos de iteração) podem ser facilmente definidos por meio de macros.

Para começar, notamos que temos $disj_2$ e $conj_2$, que são a disjunção e conjunção, mas para apenas dois argumentos. Gostaríamos de realizar disjunções e conjunções com vários goals. A disjunção de n termos é definida induvivamente (a conjunção é análoga):

$$(disj) \Rightarrow \#u$$
$$(disj\ g) \Rightarrow g$$
$$(disj\ g_0\ g\ ...) \Rightarrow (disj_2\ g_0\ (disj\ g\ ...))$$

que se traduz como

```
(define-syntax disj
  (syntax-rules ()
    ((disj) #u)
    ((disj g) g)
    ((disj g0 g ...) (disj2 g0 (disj g ...)))))
```

Cada *defrel* define uma nova função:

```
(define-syntax defrel
  (syntax-rules ()
    ((defrel (name x ...) g ...)
     (define (name x ...)
       (λ (s)
         (λ ()
           ((conj g ...) s)))))))
```

Váriaveis frescas são criadas com:

```
(define-syntax fresh
  (syntax-rules ()
    ((fresh () g ...)(conj g ...))
    ((fresh (x0 x ...) g ...)
     (call/fresh 'x0
       (λ (x0)
         (fresh (x ...) g ...))))))
```

Para executar o goal, definimos run^*, que recebe uma lista de variáveis e um goal e, se terminar de executar, assume como valor uma lista com os valores de associção a tais variáveis de modo que o goal tenha sucesso (vale lembrar, tal valor pode ser uma variável reificada). Definimos também run, que recebe um número natural n, uma lista de variáveis e um goal, e se terminar de executar, assume como valor os n primeiros elementos de run^*.

Para termos essas definições, precisamos de $take^\infty$, que, quando dado um número n e uma stream s^∞, se algo, produz uma lista de, no máximo, n valores:

```
(define (take^∞ n s^∞)
  (cond
    ((and n (zero? n)) '())
    ((null? s^∞) '())
    ((pair? s^∞)
     (cons (car s^∞)
       (take^∞ (and n (sub1 n))
         (cdr s^∞))))
    (else (take^∞ n (s^∞)))))
```

note que, se $n = false$, $take^\infty$, se retornar, produz uma lista de *todos* os valores (pergunta: valores de que?).

Agora podemos definir

```
(define (run-goal n g)
  (take^∞ n (g empty-s)))
```

e

```
(define-syntax run
  (syntax-rules ()
    ((run n (x0 x ...) g ...)
     (run n q (fresh (x0 x ...)
                (≡ `(,x0 ,x ...) q) g ...)))
    ((run n q g ...)
     (let ((q (var 'q)))
       (map (reify q)
         (run-goal n (conj g ...)))))))

(define-syntax run*
  (syntax-rules ()
    ((run* q g ...)(run #f q g ...))))
```

Com isso, temos um implementação mínima de *miniKanren*. Algumas construções importantes foram deixadas de lado em favor da brevidade, como os $cond^e$, $cond^a$ e $cond^u$, mas a leitora interessada é convidada a checar [3] ou [4] para mais detalhes.

Leituras Adicionais

A pesquisa e prática em programação lógica frequentemente é muito próxima à de linguagens funcionais e esse é o caso na área de transformações de programas. As transformações de *folding/unfolding* foram primeiro adaptadas para a programação lógica por Tamaki e Sato [1]. Mais informações podem ser encontradas em [2].

[1] R. Kent Dybvyg, "The Scheme Programming Language - Fourth Edition", disponível em https://www.scheme.com/tspl4/, acesso em Setembro de 2018.

[2] Peter Norvig, "Paradigms of Artificial Inteligence Programming - Case Studies in Common Lisp", Morgan Kauffman Publishers, 1992.

[3] Daniel P. Friedman, William E. Byrd, Oleg Kiselyov, Jason Hemann, "The Reasoned Schemer - Second Edition", The MIT Press, 2018.

[4] Site do miniKanren http://minikanren.org/.