1 Modelo Computacional de Programas Lógicos

Considere o seguinte código:

Código 1: Circuito

```
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(Entradal, X, Saida),
  transistor(Entrada2, ground, X),
  resistor(energia, Saida).

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

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

`porta_and(Entrada1, Entrada2, Saida)?`

Mais do que isso, ela pode se perguntar "Será que, dado um programa qualquer e um goal qualquer dá para "calcular" o resultado do goal?". Será instrutivo refletir por alguns momentos sobre essa questão.

A leitora pode imaginar que, 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.

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. Mas, se o goal contiver alguma variável ou só puder ser provado através de alguma regra, que é o caso geral, a situação fica mais complicada.

Se o goal contiver variáveis, para prová-lo o que precisamos é 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 substutuída por um outro termo (substituição essa que pode ser desfeita no processo de *backtracking*, a ser descrito posteriormente). 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 Pι.*

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[1] `A = 1` e, logo em seguida, `A = 2`: isso resulta em falha, por inconsistência.

Se temos que existe alguma substituição ι (possívelmente 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**[2].

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:

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

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

[2] 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.

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 será 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 não será estudado aqui. Usaremos um mais prático 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[3] para as variáveis presentes caso tal substituição exista, ou falha, caso contrário. O 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*[4] da equação na pilha;

[3] É importante que seja a mais geral, para não perdermos possíveis soluções

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

(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[5] 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, ...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

[5]No geral, podem existir várias escolhas possíveis e pode ser que, por algumas sequências de escolhas de cláusulas, nunca chegemos 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 situações mais práticas, 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).

fim. Com um processo desses, não dá para provar um goal e, portanto, é devolvida falha.

Para entender melhor, tome o exemplo do código 1, 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:

Código 2: Encontra

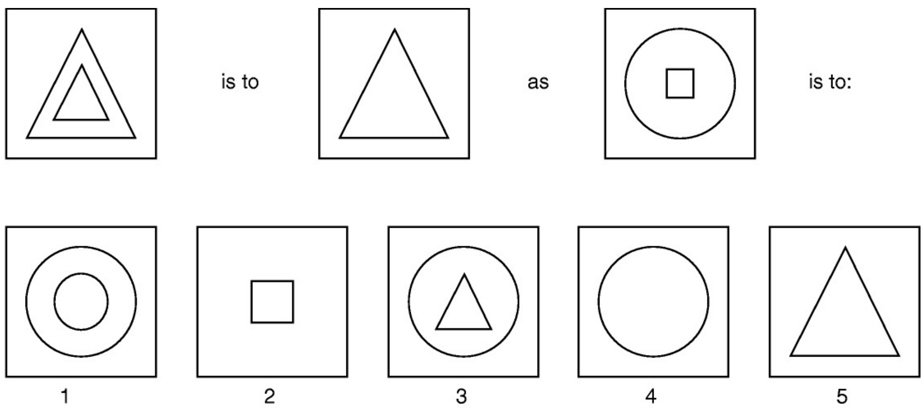
```
encontra(X) :-
  gera(X),
  testa(x).
```

Para algum *gera* e algum *testa*. A hipótese de não-determinismo 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.

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[6]. 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 1 [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[8] 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);
   - 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

[6] 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.

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

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

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[9]:

Código 3: Analogy

```
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).
```

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[10]: 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 `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*)[11].

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:

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

[10] 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 conseguiriamos trabalhar com alguns dos programas que veremos mais para frete.

[11] Caso o mistério te incomode, considere fazer uma visita ao Capítulo 3.

Código 4: Test Analogy

```
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) ]).
```

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.

Leituras adicionais

[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.