1 Modelo Computacional de Programas Lógicos

Considere o seguinte programa:

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

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

Qual será o resultado do goal `porta_and(Entrada1, Entrada2, Saida)?`, a leitora pode se perguntar. 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?". Te convido a refletir por alguns momentos sobre essa questão.

A leitora pode imaginar que, se houvessem muitos programas com goals de resultados incalculáveis, programação lógica não seria tão útil e difícilmente 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 a variável toma o valor de um outro termo. Uma forma de pensar sobre isso é que, antes da substituição, a variável "tem uma vida só sua" (é irrestrita) e que, depois, sua vida é, na verdade, "a vida de outro" (é restrita). Mais precisamente, temos:

Definição 1.1. *Substituição*
Dado um termo p(a_1, ..., a_n), onde 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 "=" denota que a_i "se torna um outro nome" para k_j e dizemos que a_i é unificado com k_j. Uma substituição ι sobre um programa P é escrita Pι.

Convém fazer algumas observações a respeito do que foi dito:

1. A relação "$A = B$" deve ser entendido 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 (onde $A = b$ não é o mesmo que $b = A$);
2. O símbolo "=" expressa a relação de dois termos terem o mesmo valor, se algum;
3. Essa relação é transitiva: se A, B e C são variáveis e se $A = B$ e $B = C$, então $A = C$ (se A "é outro nome para B" e B "é outro nome" para C, A "é outro nome" para C);
4. Pelo item (2), não podemos fazer $A = 1$ e $A = 2$: isso dá 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**.

Dito isso, podemos expressar mais claramente 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. O processo pelo qual esse objetivo é realizado é chamado **processo de resolução**.

Unificação exerce um papel fundamental na programação lógica. Na prática, ele pode processos de atribuição de valores, gerenciamento de memória, invocação de funções do sistema, entre outros. O primeiro estudo formal sobre unificação é devido a John A. Robinson, que depois de provar que existe um algoritmo de unificação, gerou o algoritmo de que temos conhecimento.

O algoritmo dele é um tanto ineficiente, e não será estudado aqui. No lugar disso, para entender como o processo de unificação (e, consequentemente, o de resolução) pode funcionar, usaremos o **algoritmo de Martelli-Montanari**:

(a) Escolha uma das proposições $p_i(t_1, ..., t_n)$ presentes no goal (lembre-se que um goal é entendido como uma conjunção de proposições);

(b) Das proposições presentes no programa, escolha não-determinísticamente uma que tenha a forma de uma das abaixo e realize a ação especificada:

1. $f(k_1, ..., k_m) = p_i(t_1, ..., t_n)$, onde $m = n$, $f = p_i$: troque as variáveis t_l por s_l;
2. $f(k_1, ..., k_m) = p_i(t_1, ..., t_n)$, onde $f \neq p_i$: delete essa equação e pare, retorne com falha;
3. $x = x$: delete a equação;
4. $x = y$, onde x é uma variável, mas y não: troque x por y em todas as proposições consideradas;
5. $x = y$, onde x é variável de y (isto é, y é algo como $y(a_1, ..., x, ..., a_n)$): pare e retorne com falha.

(c) Se existe mais algum p_i a ser escolhido, volte ao item (a), se não retorne com sucesso.

Intuitivamente, esse algoritmo tenta provar o goal a partir do programa de forma construtiva: isto é, tenta construir uma solução por meio de substituições e, se não chegar a uma contradição irrecuperável, termina com sucesso, "retornando" (não no sentido de uma função que retorna um valor, mas no de "mostrar" ao usuário do programa) a substituição realizada (na verdade, extritamente falando, ele não precisa "retornar" as substituições, mas assumiremos que retorna).

No item (a), escolhemos um dos termos do goal para provar:como discutido anteriormente, o goal é verdadeiro se cada um de seus termos o for e, provando todos os p_i, provamos o goal.

Em seguida, no item (b), escolhemos não-determínisticamente[1] uma das cláusulas (digamos, a cláusula m_j) do programa para provar p_i. Essa operação consiste na tentativa de unificação de p_i com m_j.

Os itens de 1 a 5 são referentes a um passo da unificação. O item 5 merece uma explicação um pouco mais detalhada. O que ela diz é 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 exerce uma função primariamente estrutural e simbólica. Sem esse item, 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 círculo sem fim. Com um processo desses, não dá para provar um goal e, portanto, se retorna falha.

[1] 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 cláusulas. Quando dizemos que a escolha é não-determinística, queremos dizer que, se existem 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, mas, ainda assim, é uma abstração importante e leva a aplicações interessantes, como as dos, assim chamados, *programas não-determinísticos*.

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

1. Escolha um das proposições do goal: neste case, o goal só tem uma proposição (que é `resistor(energia, n1)`);
2. Escolha uma das proposições do programa: podemos escolher a primeira (que é `resistor(energia, n1).`)
3. Ela é do tipo 1, mas não tem variáveis, então não é preciso atualizar valor algum;
4. Como o goal era de apenas uma proposição, não há mais o que fazer, e é retornado sucesso, com uma substituição vazia.

O ideal seria que, se um goal pode ser provado por um programa, o processo de prova seguisse um caminho direto, sem tentar unificações infrutíferas. Na prática, isso só é realizável para situações muito específicas e, no geral, serão tentadas diversas unificações infrutíferas antes de se chegar a um objetivo.

Para lidar com isso, é, no geral, criado um *choice point* logo antes de se tentar uma unificação. Se a tentativa resulta em falha, o processo volta ao estado de antes da tentativa, e a possibilidade daquela unificação é eliminada. O goal só falha quando todas as possibilidades foram eliminadas.