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

porta_and(Entradal, Entrada2, Saida) :-
  nand_gate(Entradal,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 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 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" (ou, é irrestrita) e que, depois, sua vida é, na verdade, "a vida de outro" (ou, é restrita). Mais precisamente, temos:

Definição 1.1. *Substituição*
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 "=" denota 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 $P\iota$.

Convém fazer algumas observações a respeito do que foi dito:

1. 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 (onde $A = b$ não é o mesmo que $b = A$);
2. O símbolo "=" expressa a relação de dois termos serem idênticos;
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 idêntico a B e B é idêntico a C, então A é idêntico a C);
4. Pelo item (2), não podemos fazer $A = 1$ e $A = 2$: 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**[1].

Todas as substituições 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 logo mais.

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 regras. O processo pelo qual esse objetivo é realizado é chamado **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, que depois de provar que existe 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

[1] Ou, dependendo do contexto, de substituição, mas isso não deve alterar a compreensão do texto.

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

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

(a) Primeiro 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[2] 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

[2] 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 da assim chamada *programação não-determinística*.

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. Ela 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 exerce uma função primariamente estrutural e simbólica. Sem isso, 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 Circuito, 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).