1 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.

1.1 Semântica

Semântica tem a ver como 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 1.1. *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 programa como o seguinte:

Código 1: Natural

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

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 1.2. *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 1.3. *Dada uma interpretação I de um programa lógico, I é um 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 1.4. *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.

1.2 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[1] (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

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

[1] 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.

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.

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)`[2] 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}[3] [4];
     (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};

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

[3] 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.

[4] 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) uma checagem para evitar a não-terminação de um programa em um caso específico[5]. 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.

[5] Apesar de em implementações práticas essa checagem ser omitida por questões de eficiência, podendo ser ativada manualmente.

É um fato clássico[6] 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 1.5. *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 1.6. *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 1.1 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, e temos motivos para fazer isso.

Definição 1.7. *Um tipo é **completo** se é fechado pela relação de instância.* **completo**

Assim, um (número) natural completo (vide programa 1.1 deste Capítulo) é ou 0 ou um $s^n(0)$, para $n \in N$.

1.3 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

[6] 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 1.8. *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 1.9. *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

1.4 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[7]: na lógica clássica, com mais a adição de proposições permite a

[7]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..