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. A teoria de programação lógica junta a base matemática com um experiência em computação e engenharia. Olhando por alto, existem x aspectos mais importantes nessa teoria: a semântica semântica, corretude e complexidade de um programa lógico. Investigaremos cada um desses aspectos, nessa ordem.

1.1 Semântica

A semântica é o significado do programa. No primeiro capítulo começamos uma discussão informal nesse sentido. Como foi lá notado, a definição que lá demos de "significado" é, na realidade, a de significado procedural. Aqui, vamos trabalhar com de uma maneiro um pouco diferente.

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

Definição 1.1. *Universo Herbrand*
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.

Por exemplo, se temos um programa como o seguinte:

Código 1: Natural

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

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

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

Definição 1.2. *Base Herbrand*
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).

Definição 1.3. *Interpretação*
Uma interpretação I de um programa lógico P é um subconjunto de B(P).

Definição 1.4. *Modelo*
Dada uma interpretação I de um programa lógico, I é um modelo se, para cada termo de P, `a :-` b_0`, ...,` b_n, $a \in I$ *se* b_0, ..., $b_n \in I$.

Assim, um modelo é uma interpretação consistente com as regras do programa.

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

Definição 1.5. *Significado declarativo*
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.*

1.2 Correção do Programa

Pelo que vimos na seção anterior, todo programa tem um significado bem definido, o qual não é correto nem incorreto. Apesar disso, esse significado pode ou não corresponder ao intencionado pela programadora. Querer tratar matemáticamente o "significado intencionado pela programadora" é querer tirar conclusões rigorosas de algo não rigorosamente definido. Ultrapassamos 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 fica de exercício para a leitora diligente.

Supongo 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 o intencionado está contido no do programa: um programa é correto e completo se o significado intencionado igual ao significado do programa. Geralmente, gostaríamos que os programas sejam 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. Vimos anteriormente (no capítulo 1, o item 5 do algoritmo de Martelli-Montanari) uma checagem para evitar a não-terminação de um programa em um caso específico, mas, nas implementações práticas, essa checagem é omitida por questões de eficiência (apesar de poder ser ativado manualmente). 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.

Por exemplo, se omitirmos o primeiro fato do programa 1, o goal `natural(X)?` não termina. É um fato clássico[1] que não se pode dizer se um programa qualquer termina. Felizmente, no geral não lidaremos com programas quaisquer e, eventualmente, poderemos dizer se ele termina ou não e em quais circustâncias.

Mas, antes de prosseguirmos, a seguinte construção será ocasionalmente útil:

Definição 1.6. *Lista*

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

Usaremos a seguinte definição formal de lista:

- `.()` *(o "funtor .") é uma lista*[2]*;*
- `.(A, B)` *é uma lista se B é uma lista*

Como é chato ficar escrevendo coisas como `.(A, .(B, C))` *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 arguentos);*
- `[A, B, C, ...]` *é o mesmo que* `.(A, .(B, .(C, ...)))`*;*
- `[A|B]` *indica que A é o primeiro elemento da lista e B é o resto da lista (B é uma lista).*

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

Código 2: Append

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

Esse é um programa clássico e, por isso, escolhemos manter seu nome clássico, que, daqui para frente 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, diga qual seria o resultado do goal `append([cafe, queijo], [goiabada], L)?`.

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

Definição 1.7. *Domínio*
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* $B \subseteq D$*.*

Definição 1.8. *Domínio de terminação*
Um domínio de terminação D de um programa P é um domínio tal que qualquer computação de qualquer goal em D termina em P.

[2]Na verdade, isso não é extritamente 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, extritamente falando, diferente de um funtor.

No geral, gostaríamos que um programa tenha um domínio de terminação contido no seu significado intencionado. Para um programa lógico interessante, no geral isso não pode ser obtido. Mas, felizmente, as linguagens de programação com que lidaremos são restritivas o suficiente para que possamos obter esse tipo de resultado no futuro.

Mas antes disso, é útil tentarmos 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 append, introduzido no programa 2 deste capítulo, como um tipo, mas não temos, atualmente, motivos para fazer isso. Análogamente, podemos chamar arvore_b, no programa 3 do capítulo 0 de um tipo e temos motivos para fazer isso.

Definição 1.9. *Tipo completo*
Um tipo é completo se é fechado pela relação de instância.

Assim, um (número) natural completo (vide programa 1 deste capítulo) é ou 0 ou um $s^n(0)$, para $n \in N$.

Definição 1.10. *Lista Completa*
Uma lista L é completa se toda instância Lι satisfaz a definição dada: se existem instâncias que não a satisfazem, ela é dita incompleta

1.3 Complexidade

Programas lógicos no geral podem ser usados de várias formas diferentes, o que pode mudar a natureza de sua complexidade. Para analizar 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 bf 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 cláusula no programa que o prova, dizemos que tal computação é bf 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 hajam 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 1.11. *Tamanho de um termo*
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.

Definição 1.12. *Complexidade por Comprimento*
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).