Linguagens de Programação

Semântica Operacional

Samuel da Silva Feitosa

Aula 19

Análise Semântica

Introdução

- Como descrever uma linguagem de programação?
  - **Sintaxe:** como se vê o programa, sua forma e estrutura. Vimos a pouco o uso de **gramáticas** para especificar construções válidas da linguagem.
  - **Semântica:** o que os programas fazem, seus comportamentos e significados.

Sintaxe + Semântica: em conjunto definem uma linguagem

Semântica

- Não existe uma única forma aceitável para definição da semântica de uma Linguagem de Programação.
- Existem muitos critérios que devem ser considerados para a criação de uma metodologia e definição da notação para a semântica.
  - Programadores precisam entender o que os comandos significam.
  - Desenvolvedores de compiladores precisam saber exatamente o que as construções da Linguagem de Programação fazem.
  - Provas de corretude deveriam ser possíveis.
  - Geradores de compiladores deveriam ser suportados.
  - Detecção de ambiguidades e inconsistências.

Semântica Formal

- Existem três tipos de semânticas formais.
  - Semântica **operacional**: define o programa como sua execução em uma máquina abstrata interpretada.
  - Semântica **axiomática**: define o programa através de asserções sobre como os estados são alterados em cada passo.
  - Semântica **denotacional**: tenta capturar o que a computação (de forma matemática) faz em cada passo.

Semântica Operacional

- Para compreender os detalhes da semântica operacional, vamos definir uma pequena Linguagem de Programação.
  - Expressões aritméticas e booleanas.
  - Operações de adição (+) e conjunção lógica (^).
- Sintaxe da linguagem que usaremos:

$n ::=$	$0, 1, 2, ...$	numeric constants
$e ::=$		expressions:
	$true$	constant true
	$false$	constant false
	n	numeric constant
	$e + e$	math operator
	$e \wedge e$	logic operator

Linguagem de Expressões

- A sintaxe abstrata apresentada diz que é possível construir uma expressão de 5 maneiras.
  - As primeiras três linhas dizem que **true**, **false** e **n** são possíveis constantes.
  - A quarta linha diz que, se pudermos construir expressões $\mathbf{e_1}$ e $\mathbf{e_2}$, então também é possível construir $\mathbf{e_1 + e_2}$.
  - A última linha é similar. Se $\mathbf{e_1}$ e $\mathbf{e_2}$ são expressões, então $\mathbf{e_1 \text{ ^ } e_2}$ também é uma expressão.

Valores Permitidos

- Além da definição da sintaxe da linguagem, também define-se um subconjunto de expressões que representam valores finais da avaliação.
  - Nesta linguagem que estamos definindo, valores são apenas as constantes booleanas e numéricas.

$v ::=$		values:
	$true$	true value
	$false$	false value
	nv	numeric values
$nv ::=$	$0, 1, 2, ...$	numeric definitions

Implementação em Haskell

- Para representar a *árvore de sintaxe abstrata* (AST) em Haskell, usamos nosso conhecido *algebraic data type* (ADT).

```haskell
-- Árvore de sintaxe abstrata
data Expr = BTrue
          | BFalse
          | Num Int
          | Add Expr Expr
          | And Expr Expr
          deriving Show
```

Semântica Small-step

- Com a semântica de passo-pequeno (small-step), as computações são representadas através de um sistema de inferência lógica.
  - O propósito é descrever como passos individuais de computação acontecem.
  - São representados como um sistema de transição.
- Usam-se **regras de inferência** para descrever o comportamento de um programa.
  - Conclusão segue a partir de uma série de premissas.

$$\frac{premise_1 \quad premise_2 \quad \ldots \quad premise_n}{conclusion}$$

Semântica da LP de expressões

- Meta-variáveis para valores (**v**, **bv**, **nv**).
- **e**, $\mathbf{e_1}$, **e'**, etc., representam expressões.
- Relação de transição **e** $\longrightarrow$ **e'**, significa:
  - **e** avalia para **e'** em um passo de redução.

$$\frac{e_1 \longrightarrow e_1'}{e_1 + e_2 \longrightarrow e_1' + e_2} \text{ [S-Add}_1\text{]} \qquad \frac{e_2 \longrightarrow e_2'}{nv + e_2 \longrightarrow nv + e_2'} \text{ [S-Add}_2\text{]}$$

$$nv_1 + nv_2 \longrightarrow nv_1 \oplus nv_2 \text{ [S-Add]} \qquad \frac{e_1 \longrightarrow e_1'}{e_1 \wedge e_2 \longrightarrow e_1' \wedge e_2} \text{ [S-And}_1\text{]}$$

$$true \wedge e_2 \longrightarrow e_2 \text{ [S-And}_2\text{]} \qquad false \wedge e_2 \longrightarrow false \text{ [S-And}_3\text{]}$$

Detalhamento das regras

- **S-Add$_1$:** Se temos uma expressão **e$_1$ + e$_2$**, primeiro avalia-se **e$_1$** produzindo uma expressão **e$_1$'**, resultando em outra expressão **e$_1$' + e$_2$**.
- **Se-Add$_2$:** Similar, porém considera que a expressão da esquerda já é um valor (**nv**). Assim, avalia **e$_2$**, produzindo um **e$_2$'**, e o resultado da expressão é **nv + e$_2$'**.
- **S-Add:** Se ambas as expressões forem valores, efetua a adição entre **nv$_1$** e **nv$_2$**.
- **S-And$_1$:** Mesmo propósito de **S-Add$_1$**.
- **S-And$_2$ e S-And$_3$:** Se a expressão da esquerda representa o valor **true**, então o resultado é **e$_2$**. Se o valor da expressão da esquerda for **false**, o resultado é **false**.

Implementação do Interpretador

- A partir das regras semânticas, podemos implementar uma função em Haskell que avalia um passo de redução.

```haskell
-- Função que avalia um passo de execução
step :: Expr -> Maybe Expr
step (Add (Num n1) (Num n2)) = Just (Num (n1 + n2))
step (Add (Num n1) e2) = case (step e2) of
                           Just e2' -> Just (Add (Num n1) e2')
                           Nothing  -> Nothing
step (Add e1 e2) = case (step e1) of
                     Just e1' -> Just (Add e1' e2)
                     Nothing  -> Nothing
step (And BTrue e2) = Just e2
step (And BFalse _) = Just BFalse
step (And e1 e2) = case (step e1) of
                     Just e1' -> Just (And e1' e2)
                     Nothing  -> Nothing
```

Considerações Finais

- Nesta aula estudamos conceitos de semântica de linguagens de programação.
- Focamos especificamente em semântica formal.
  - Definimos uma mini-linguagem de programação.
  - Modelamos a semântica operacional da mesma.
  - Utilizamos a abordagem *small-step*.
- Também vimos:
  - Como representar ASTs em Haskell.
  - Como implementar um interpretador para a linguagem que estamos modelando.