Denotační sémantika

Marek Běhálek

Katedra informatiky, FEI,
Vysoká škola báňská – Technická universita Ostrava
17. listopadu 15, Ostrava-Poruba 708 33
Česká republika

29. listopadu 2015

WWW stránky

Webové stránky k předmětu naleznete na adrese:

http://www.cs.vsb.cz/behalek

Prezentace byla připravena jako opora k předmětu Paradigmata programování Materiály vycházi z původních materiálu k předmětu Funkcionální a logické programování doc. Dr. Ing. Miroslava Beneše.

Princip denotační sémantiky

- Denotační sémantika programovacího jazyka popisuje význam programu jako *funkci*

$$\text{program} : \mathit{Input} \longrightarrow \mathit{Output}$$

- **Princip kompozicionality** — význam složené konstrukce je dán kombinací významů jednotlivých složek
- Pro popis se používá λ-**kalkulu** — lze jednoduše vyjádřit také pomocí *funkcionálního jazyka*
- Princip kompozicionality - význam složené konstrukce je dán kombinací významů jednotlivých složek

$$[[E1 + E2]] = [[E1]] + [[E2]]$$

není vždy splněno – konstrukce pro paralelní výpočty

Abstraktní syntax

- Popisuje zjednodušenou strukturu programu bez detailů, které nenesou žádnou sémantickou informaci (priorita a asociativita operátorů, závorky, . . .)
  - ano - operátory, operandy
  - ne – závorky, priorita, asociativita, oddělovače
- Pro datovou realizaci abstraktní syntaxe programu se používá obvykle stromová struktura — **abstraktní syntaktický strom** (AST)

Sémantické funkce

- Přiřazují význam jednotlivým syntaktickým konstrukcím
- Syntaktické konstrukce jsou strukturovány do syntaktických **domén** (doména výrazů `Exp`, příkazů `Exp`, deklarací `Dec`, programů `Prog` apod.)
- Sémantické funkce jsou definovány pro každou doménu zvlášť:

$$e\ [\![1+1]\!] = 2$$
$$p\ [\![i = \mathit{read}; \mathit{write} 2 * i;]\!] = \lambda(x : \_).[2 * x]$$

Výrazy s konstantami

- významem výrazu je jeho hodnota:

$$e\ [\![1+1]\!] \ = \ 2$$

```
data Exp = Add Exp Exp
         | Mul Exp Exp
         | Neg Exp
         | Num Int

e :: Exp -> Int
e (Add e1 e2) = (e e1) + (e e2)
e (Mul e1 e2) = (e e1) * (e e2)
e (Neg e1)    = - (e e1)
e (Num x)     = x
```

Výrazy s proměnnými

- Význam výrazu (jeho hodnota) závisí na konkrétních hodnotách proměnných
- Funkce přiřazující proměnným hodnotu (valuační funkce) musí být parametrem sémantické funkce e — tato funkce modeluje paměť počítače s pojmenovanými buňkami

```
type Store = String -> Int
data Exp   = ...
           | Var String

e :: Exp -> Store -> Int

e (Add e1 e2) s = (e e1 s) + (e e2 s)
e (Num x)     _ = x
e (Var v)     s = s v
```

Výrazy s přiřazením

- Hodnoty proměnných (a tedy i valuační funkce) se mohou během vyhodnocení výrazu změnit
- Změněná valuační funkce musí být součástí funkční hodnoty sémantické funkce e (tj. výsledek musí být uspořádanou dvojicí hodnot)

```
data Exp = ...
         | Asgn String Exp

e :: Exp -> Store -> (Int, Store)
e (Asgn v e1) s = let (v1,s') = e e1 s
                      s'' v' = if v'==v then v1
                               else s v'
                  in (v1, s'')
e (Add e1 e2) s = let (v1,s') = e e1 s
                  in let (v2, s'') = e e2 s'
                     in (v1+v2, s'')
e (Num x)     s = (x, s)
```

Příkazy

- Příkaz neprodukuje (na rozdíl od výrazu) hodnotu, jeho významem je vedlejší efekt – změna stavu programu (např. hodnot proměnných)

```
data Com = Eval Exp
         | If    Exp Com
         | While Exp Com
         | Seq   Com Com

c :: Com -> Store -> Store
c (Eval e1) s = let (_,s') = e e1 s
                in s'
c (If e1 c1) s = let (v1,s') = e e1 s
                 in if v1 == 0 then s'
                    else c c1 s'
c (While s1 c1) s = let (v1,s') = e e1 s
                    in if v1 == 0 then s'
                       else c (While e1 c1) (c c1 s')
c (Seq c1 c2) s = c c2 (c c1 s)
```

Vstup a výstup

- Stav programu je tvořen nejen okamžitými hodnotami proměnnných, ale také nezpracovaným vstupem a již vyprodukovaným výstupem.
- Místo typu `Store` je třeba v předchozích definicích používat typ `State` zahrnující i vstup a výstup programu

```
type Input  = [Int]
type Output = [Int]
type State  = (Store, Input, Output)

data Exp = ...
         | Read

e :: Exp -> State -> (Int, State)
e Read (s, x:xs, o) = (x, (s, xs, o))

data Com = ...
         | Write Exp
c :: Com -> State -> State
c (Write e1) s = let (v1, (s', i, o)) = e e1 s
                 in (s', i, v1:o)
```

Kontinuace

- Význam „zbytku programu“ od aktuálního místa v programu až po jeho ukončení lze modelovat **kontinuací** — funkcí, která na základě aktuálního stavu vrátí výsledek celého programu
- Sémantická funkce obdrží jednu nebo více kontinuací, kterým po vyhodnocení může (a nemusí) předat řízení
- Kontinuace výrazu obdrží hodnotu výrazu, stav programu po jeho výpočtu a vrátí výsledek programu:

$$\mathbf{type}\ \mathsf{ECont} = \mathbf{Int} \mathrel{->} \mathsf{State} \mathrel{->} \mathsf{Output}$$

- Kontinuace příkazu obdrží stav po provedení příkazu a vrátí výsledek programu:

$$\mathbf{type}\ \mathsf{CCont} = \mathsf{State} \mathrel{->} \mathsf{Output}$$

Kontinuace

- Sémantická funkce pro výraz obdrží zpracovávaný výraz, stav před jeho výpočtem a kontinuaci, které má předat nový stav a výsledek; vrací výsledek celého programu („vyrobený“ obvykle kontinuací):

```
e :: Exp -> State -> ECont -> Output
e (Add e1 e2) s ec =
   e e1 s (\v1 s' ->
      e e2 s' (\v2 s'' -> ec (v1+v2) s''))
```

- Sémantická funkce pro příkaz obdrží zpracovávaný příkaz, stav před jeho provedením a kontinuaci, které má předat nový stav; vrací výsledek celého programu:

```
c :: Com -> State -> CCont -> Output
c (Seq c1 c2) s cc = c c1 s (\s' ->
                        c c2 s' (\s'' -> cc s''))
c (If e1 c1) s cc =
   e e1 s (\v1 s' -> if v1 == 0 then cc s'
                     else c c1 s' cc)
```

Použití kontinuací

- *reakce na chybové stavy* — funkce nepředá řízení žádné kontinuaci a tím ukončí vyhodnocení
- *definice významu skokových příkazů* (`break`, `goto`) — při skoku se předá řízení kontinuaci odpovídající cílovému místu skoku
- *definice významu volání podprogramu* — kontinuace odpovídající zbytku programu za příkazem volání se uschová pro použití v příkazu `return`

Program

- Modely výpočtu
  1. Program nejprve provede výpočet a pak zobrazí výsledek nebo chybu
  2. Program vypisuje průběžně výsledky a na konci oznámí úspěch nebo chybu
- Syntax a sémantika programu
  - program je tvořen posloupností příkazů

$$\mathbf{type}\ \text{Prog} = \text{Com}$$

  - vstupem programu je posloupnost čísel, výstupem posloupnost čísel zakončená příznakem OK nebo chybovou zprávou

$$\begin{aligned}&\mathbf{type}\ \text{Input} = [\mathbf{Int}]\\&\mathbf{data}\ \text{Value} = \text{I}\ \mathbf{Int}\\&\qquad\qquad\quad \mid \text{OK}\\&\qquad\qquad\quad \mid \text{Err}\ \mathbf{String}\\&\mathbf{type}\ \text{Output} = [\text{Value}]\end{aligned}$$

Stav program

- stav programu je tvořen obsahem proměnných a nepřečteným vstupem; na počátku není žádná proměnná definována

```
type Store = String -> Maybe Int
data State  = State {store :: Store,  input :: Input}

emptyStore ::  Store
emptyStore id = Nothing

initialState   ::  Input  -> State
initialState   inp  = State {store=emptyStore, input=inp}
```

Vyhodnocení programu

- vyhodnocení programu

```
p :: Prog -> Input -> Output
p body inp =
      c body (initialState inp) (\_ -> [OK])
```

- ošetření nedefinované proměnné:

```
e :: Exp -> State -> ECont -> Output
e (Var v) s ec = case (store s) v of
      Just v1 -> ec v1 s
      Nothing -> [Err ("Nedef: "++v)]
```

- realizace příkazu `Write`:

```
c :: Exp -> State -> CCont -> Output
c (Write e1) s cc = e e1 s (\v1 s' ->
    (I v1):(cc (State (store s') (input s'))))
```