FUN: uma pequena linguagem funcional

Pedro Vasconcelos

17 de Abril de 2020

Definir uma linguagem funcional mínima

1. Escolher uma estratégia de avaliação para o cálculo-λ:
   - ordem aplicativa (*call-by-value*) ou ordem normal (*call-by-name*/*lazy evaluation*)
   - formas normais fracas: não reduzir dentro de λs
2. Acrescentar constantes e operações primitivas:
   - inteiros, operações aritméticas
   - boleanos, caracteres, ...
3. Acrescentar estruturas de dados:
   - pré-definidos: pares, tuplos, listas
   - tipos definidos pelo programador

Plano

1. Linguagem FUN

2. Semântica operacional
   - Interpretador natural
   - Interpretador com ambientes
   - Interpretador CPS

Plano

1. Linguagem FUN

2. Semântica operacional
   - Interpretador natural
   - Interpretador com ambientes
   - Interpretador CPS

Uma pequena linguagem funcional

- Avaliação por ordem aplicativa (*call-by-value*)
- Valores inteiros primitivos
- Operações aritméticas ($+, -, \times$)
- Expressão condicional **ifzero**
- Definições locais **let**. . . **in**. . .
- Definições recursivas usando um operador ponto-fixo **fix**. . .
- Um programa é uma expressão sem variáveis livres
- Omitimos: outros tipos de dados, I/O

Sintaxe de expressões

$$
\begin{array}{rcll}
e & ::= & x & \text{variáveis} \\
& | & \lambda x.\, e & \text{abstração} \\
& | & e_1\ e_2 & \text{aplicação} \\
& | & n & \text{inteiros} \\
& | & e_1 + e_2 & \text{operadores} \\
& | & e_1 - e_2 & \\
& | & e_1 \times e_2 & \\
& | & \textbf{ifzero}\ e_0\ e_1\ e_2 & \text{condicional} \\
& | & \textbf{let}\ x = e_1\ \textbf{in}\ e_2 & \text{definição local} \\
& | & \textbf{fix}\ e & \text{ponto-fixo}
\end{array}
$$

Exemplo: condições

$$\lambda x.\ \lambda y.\ \textbf{ifzero}\ x\ y\ x$$

Exemplo: ordem superior

$$\textbf{let } \mathit{twice} = \lambda f.\ \lambda x.\ f\ (f\ x)$$
$$\textbf{in } \mathit{twice}\ (\lambda x.\ x + 1)\ 42$$

Exemplo: factorial recursivo

$$\textbf{let } \mathit{fact} = \textbf{fix } \lambda f.\ \lambda n.\ \textbf{ifzero } n\ 1\ (n \times f\ (n-1))$$
$$\textbf{in } \mathit{fact}\ 10$$

Plano

1. Linguagem FUN

2. Semântica operacional
   - Interpretador natural
   - Interpretador com ambientes
   - Interpretador CPS

Semântica operacional

Interpretador natural

$$e \Downarrow v \qquad\qquad e \text{ reduz-se a } v$$

Valores (formas normais fracas)

$$\begin{array}{rcll} v & ::= & n & \text{inteiros} \\ & | & \lambda x.\, e & FV(\lambda x.\, e) = \emptyset \end{array}$$

Interpretador natural

Valores e aplicações

$$\frac{}{n \Downarrow n} \qquad n \in \mathsf{Int} \tag{1}$$

$$\frac{}{\lambda x.\, e \Downarrow \lambda x.\, e} \qquad FV(\lambda x.\, e) = \emptyset \tag{2}$$

$$\frac{e_1 \Downarrow \lambda x.\, e' \qquad e_2 \Downarrow v \qquad e'[v/x] \Downarrow u}{e_1\ e_2 \Downarrow u} \tag{3}$$

Interpretador natural

Condicional

$$\frac{e_0 \Downarrow 0 \qquad e_1 \Downarrow v}{\textbf{ifzero}\ e_0\ e_1\ e_2 \Downarrow v} \tag{4}$$

$$\frac{e_0 \Downarrow n \qquad e_2 \Downarrow v}{\textbf{ifzero}\ e_0\ e_1\ e_2 \Downarrow v} \quad n \in \mathrm{Int} \wedge n \neq 0 \tag{5}$$

Interpretador natural

Operações primitivas

$$\frac{e_1 \Downarrow n \qquad e_2 \Downarrow m}{e_1 + e_2 \Downarrow k} \qquad n, m \in \mathsf{Int} \wedge k = n + m \tag{6}$$

Regras semelhantes para $-$ e $\times$.

Interpretador natural

Definições locais e operador ponto-fixo

$$\frac{(\lambda x.\, e_2)\ e_1 \Downarrow v}{\textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Downarrow v} \tag{7}$$

$$\textbf{fix}\ \lambda f.\, \lambda x.\, e \Downarrow \lambda x.\, e[(\textbf{fix}\ \lambda f.\, \lambda x.\, e)/f] \tag{8}$$

Observações

- $e \not\Downarrow v$ quando nenhuma regra se aplica
  (e.g. se e é uma variável livre)
- Apenas substituimos variáveis por valores
  - não têm variáveis livres;
  - logo: não pode ocorrer captura de variáveis

$$((\lambda x\, M)\, N) \to_\beta M[N/x] \qquad \text{onde } BV(M) \cap \underbrace{FV(N)}_{=\emptyset} = \emptyset$$

Sintaxe abstrata em Haskell

```
data Term = Var Ident
          | Lambda Ident Term
          | App Term Term
          | Const Int
          | Term :+ Term
          | Term :- Term
          | Term :* Term
          | IfZero Term Term Term
          | Let Ident Term Term
          | Fix Term
            deriving (Eq, Show)

type Ident = String  -- identificadores
type Value = Term    -- valores são termos
```

Interpretador em Haskell

Definido por recursão sobre termos.

```
eval1 :: Term -> Value
```

Funções auxiliares:

```
apply :: Value -> Value -> Value                    -- redução-beta
subst :: Term -> Ident -> Value -> Term             -- substituição
primitive ::  (Int->Int->Int) -> Value -> Value -> Value
                                                    -- operações aritméticas
```

Segue-se uma demonstração...

Observações

- Usamos o operador `$!` para forçar a avaliação por ordem aplicativa.

  $f \ \$! \ x = f \ x$ mas efetua *sempre* a avaliação de x

- Caso contrário: a linguagem FUN "herdaria" a redução não-estrita do meta-interpretador (Haskell).[1]

[1] cf. John Reynolds, *"Definitional Interpreters and Higher-Order Programming Languages"*.

Plano

1. Linguagem FUN

2. Semântica operacional
   - Interpretador natural
   - Interpretador com ambientes
   - Interpretador CPS

Eliminar as substituições

Problema

`eval1` efectua múltiplas substituições sobre a mesma expressão

Solução: em vez de fazer substituições vamos registar separadamente os valores de variáveis livres num ambiente.

Ambientes

Associações de valores a variáveis:

$$\rho = [x_1 \mapsto v_1, x_2 \mapsto v_2, \ldots, x_n \mapsto v_n]$$

dom ρ	domínio de ρ (conjunto de variáveis);
ρ_x	é o valor associado a x no ambiente ρ (se $x \in$ dom ρ);
$\rho[x \mapsto v]$	é o ambiente que associa v a x e actua como ρ noutras variáveis.

Closures

Expressões-lambda podem conter variáveis livres e.g.

let mult $= \lambda x.\, \lambda y.\, x \times y$	**let** mult $= \lambda x.\, \lambda y.\, x \times y$
in mult 2	**in** mult 10

Os valores funcionais são *pares* de termos e ambientes e.g.

$$(\lambda y.\, x \times y,\, [x \mapsto 2]) \qquad (\lambda y.\, x \times y,\, [x \mapsto 10])$$

Esta representação designa-se por closure.

Interpretador com ambientes

Interpretador

$$\rho \vdash e \Downarrow v$$

Valores

$$\begin{array}{lcll} v & ::= & n & \text{inteiros} \\ & | & (\lambda x.\, e,\, \rho) & \textit{closures} \end{array}$$

Interpretador com ambientes

Valores e aplicações

$$\frac{}{\rho \vdash n \Downarrow n} \qquad n \in \mathsf{Int} \tag{9}$$

$$\frac{}{\rho \vdash x \Downarrow \rho_x} \qquad x \in \mathsf{dom}\ \rho \tag{10}$$

$$\frac{}{\rho \vdash \lambda x.\, e \Downarrow (\lambda x.\, e,\ \rho)} \tag{11}$$

$$\frac{\rho \vdash e_1 \Downarrow (\lambda x.\, e',\ \rho') \qquad \rho \vdash e_2 \Downarrow v \qquad \rho'[x \mapsto v] \vdash e' \Downarrow u}{\rho \vdash e_1\ e_2 \Downarrow u} \tag{12}$$

Interpretador com ambientes

Condicional, operações primitivas

$$\frac{\rho \vdash e_0 \Downarrow 0 \qquad \rho \vdash e_1 \Downarrow v}{\rho \vdash \textbf{ifzero}\ e_0\ e_1\ e_2 \Downarrow v} \tag{13}$$

$$\frac{\rho \vdash e_0 \Downarrow n \qquad \rho \vdash e_2 \Downarrow v}{\rho \vdash \textbf{ifzero}\ e_0\ e_1\ e_2 \Downarrow v}\ n \in \mathsf{Int} \wedge n \neq 0 \tag{14}$$

$$\frac{\rho \vdash e_1 \Downarrow n \qquad \rho \vdash e_2 \Downarrow m}{\rho \vdash e_1 + e_2 \Downarrow k}\ n, m \in \mathsf{Int} \wedge k = n + m \tag{15}$$

Interpretador com ambientes

Operador ponto-fixo

$$\frac{}{\rho \vdash \textbf{fix}\ \lambda f.\, \lambda x.\, e \Downarrow v} \qquad v = (\lambda x.\, e,\ ?)$$

Qual o ambiente desta *closure*?

Interpretador com ambientes

Operador ponto-fixo

$$\frac{}{\rho \vdash \textbf{fix}\ \lambda f.\, \lambda x.\, e \Downarrow v} \qquad v = (\lambda x.\, e,\ \rho[f \mapsto v])$$

Interpretação operacional: construimos uma *closure cíclica*.

Ambientes em Haskell

```
-- ambientes
type Env = [(Ident,Value)]

-- valores
data Value = Int Int
           | Closure Term Env
           deriving (Eq,Show)
```

`Env` e `Value` são mutuamente recursivos.

Ambientes em Haskell (cont.)

```
-- ambiente vazio
[] :: Env

-- operações
-- acresentar uma entrada
(:) :: (Ident,Value) -> Env -> Env

-- pesquisar uma variável
lookup :: Ident -> Env -> Maybe Value
```

Interpretador com ambientes em Haskell

```
eval2 :: Term -> Env -> Value
```

Nota: não necessita de fazer substituições.

Segue-se uma demonstração...

Observações

- Tal como antes, usamos `$!` para forçar avaliação por ordem aplicativa
- O interpretador de Haskell diverge se tentarmos mostrar uma *closure* cíclica

Interpretador CPS

Vamos definir um interpretador em que a ordem de avaliação é explicita usando “continuation passing style” (CPS).

Continuações

Uma continuação é uma função que representa o resto da computação.

Usando ordem superior podemos re-escrever qualquer função com a sua continuação como argumento.

Continuações (cont.)

Exemplo: a função factorial

```
fact :: Int -> Int
```

Vamos definir `factCPS` tal que

```
factCPS n k = k (fact n)
```

em que a continuação é o argument `k`.

Como o resultado do factorial é inteiro, as continuações são funções com o seguinte tipo:

```
type Cont = Int -> Int
```

Continuações (cont.)

Estilo directo

```
fact :: Int -> Int
fact n
  | n>0 = n * fact (n-1)
  | otherwise = 1
```

CPS

```
factCPS :: Int -> Cont -> Int
factCPS n k
  | n>0 = factCPS (n-1) (\r -> k (n*r))
  | otherwise = k 1
```

Continuações (cont.)

Para calcular factoriais passamos a *função identidade* como continuação inicial.

```
> factCPS 10 id
3628800
```

Note que `factCPS` é diretamente recursiva (*tail-recursive*) mas `fact` não é.

Interpretador CPS

Re-escrevemos o interpretador com ambiente em CPS:

```
eval3 :: Term -> Env -> Cont -> Value
```

Como o resultado é do tipo `Value`, as continuações têm o seguinte tipo:

```
type Cont = Value -> Value
```

A ordem de avaliação é explicita no interpretador CPS pelo composição de continuações.