Interpretação e Compilação de Linguagens de Programação
Abstração Funcional

11 de Abril de 2013

1 Introdução

Nesta aula introduz-se a noção de abstração funcional em linguagens de programação. A abstração funcional é um mecanismo fundamental das linguagens de programação, sendo o cálculo-lambda (proposto por Curch na década de 1930) o seu representante por excelência. Entre muitos outros desenvolvimentos, o cálculo lambda inspirou o desenvolvimento na década de 60 de linguagens como o Lisp e o Algol, que formam a base de todas as linguagens de programação modernas.

O mecanismo de abstração permite definir novas operações, tipos de dados ou outros elementos, usando a própria linguagem de programação. As abstrações são definidas abstraindo alguns dos seus elementos. A criação de abstrações permite não só a modularidade de programas, mas também a reutilização de código e a definição de funções recursivas (que é a base da expressividade das linguagens funcionais). O mecanismo de abstração pode ser aplicado a vários níveis ou a vários elementos sintáticos de uma linguagem de programação. Por exemplo as abstrações de expressões são funções; as abstrações de comandos são procedimentos, as abstrações de classes (parametrizando em tipos) são classes genéricas (Java,C♯), as abstrações de classes (parametrizando a superclasse) definem mixins (Scala), as abstrações de módulos (parametrizando módulos importados) definem functores (ML, Haskell), etc.

2 Abstração funcional

Quando se declara um identificador associa-se um identificador a uma denotação e a definir o âmbito dessa declaração. Numa abstração define-se o contexto de uma ligação de um parâmetro, a expressão que define o corpo da abstração, sem definir qual a sua denotação. Só é possível avaliar a expressão quando for conhecida a denotação do seu parâmetro (e de todos os outros

`int f(int x} { return x+1; }`	C
`Function f(x:Integer):Integer` `Begin f := x End`	Pascal
`fun x -> x+1`	ML
`f(x) = x+1`	Haskell
`\x -> x+1`	Haskell (anonymous)
`x => x+1`	C♯
`(lambda (x) (x+1))`	Lisp
$\lambda x.x + 1$	Lambda calculus

Figura 1: Definição de funções

identificadores livres). A denotação é conhecida na aplicação da abstração a um argumento.

As definições da Figura 1 correspondem à mesma abstração em diversas linguagens. Sendo que a diferença entre as várias definições reside na declaração obrigatória de um nome nas linguagens C, Pascal e Haskell (noção clássica de função) ou de abstração funcional ou função anónima (ML, Haskell, C♯, etc.). Existem paralelos por exemplo ao nível das linguagens com classes que suportam classes anónimas (Java, C♯, Scala).

Definimos a linguagem *CALCF* com duas construções novas, a definição de uma abstração com um parâmetro (`fun` $x \to e$) e a aplicação (de uma abstração a um argumento) ($e\ e'$). A sintaxe abstrata da linguagem é definida pelo tipo de dados da Figura 2. A linguagem *CALCF* é definida por extensão da linguagem *CALCI* e não tem as construções imperativas. Na realidade, a definição adotada é composicional tornando as duas extensões ortogonais e de fácil combinação.

Definimos o conjunto dos identificadores livres de uma abstração como sendo o conjunto dos identificadores livres da expressão abstraída excluindo os parâmetros da mesma.

$$free(\lambda x.e) = free(e) \backslash \{x\}$$

A função de identificadores livres está definida em Haskell na Figura 3.

Relembre que os parâmetros são identificadores locais cuja ligação é instanciada na única operação que as abstrações permitem, a aplicação. A existência de nomes locais permite-nos definir uma relação de equivalência sintática entre abstrações se existir uma renomeação dos seus parâmetros

```
data CalcF =
    Num Int
  | Add CalcF CalcF
  | Sub CalcF CalcF
  | Mul CalcF CalcF
  | Div CalcF CalcF

  | Id String
  | Decl String CalcF CalcF

  | Fun String CalcF
  | Call CalcF CalcF
```

Figura 2: Sintaxe abstrata

```
free :: CalcF -> [String]
free (Id x)        = [x]
free (Decl x e e') = (free e) ++ (filter (\y -> x /= y) (free e'))
free (Num n)       = []
free (Add e e')    = (free e) ++ (free e')
free (Sub e e')    = (free e) ++ (free e')
free (Mul e e')    = (free e) ++ (free e')
free (Div e e')    = (free e) ++ (free e')
free (Fun x e )    = (filter (\y -> x /= y) (free e))
free (Call e e')   = (free e) ++ (free e')
```

Figura 3: Nomes livres com abstrações

que as torna sintaticamente iguais.

$$\lambda x.x + w =_\alpha \lambda y.y + w$$

Chama-se equivalência alfa ($=_\alpha$) a esta relação.

3 Semântica operacional

Definimos agora a semântica operacional da linguagem *CALCF*. A intuição é que uma abstração denota uma expressão cuja avaliação está atrasada até ser conhecida a denotação do seu parâmetro, e a denotação de uma aplicação é a denotação do corpo da abstração, depois de instanciada a ligação do parâmetro para a denotação do argumento.

A semântica da linguagem *CALCF* é dada pela função de interpretação para expressões fechadas:

$$I : CALCF \longrightarrow Result$$

```
eval :: CalcF -> StateMaybe Int CalcF
...
eval (Fun x e ) = return (Fun x e)

eval (Call e e' ) =
      do
      f <- eval e
      a <- eval e'
      case f of
        Fun x e'' -> eval (subst a x e'')
        _ -> raise_error
```

Figura 4: Semântica operacional para a linguagem *CALCF*

Onde o conjunto *Result* é definido pelo sub-conjunto das expressões CALCF que correspondem aos literais inteiros e às abstrações fechadas. No fragmento de código Haskell da Figura 4 está definida a função `eval` para as novas construções da linguagem.

Consideremos o seguinte exemplo de avaliação de uma expressão

```
decl x=1 in
  decl f = fun y -> y+x in
    decl g = fun x -> x+f(x)
      in g(2)
```

Os passos intermédios da sua avaliação são os seguintes:
(caso `Decl`: substituição de `x` por `1`)

```
decl f = fun y -> y+1 in
  decl g = fun x -> x+f(x)
    in g(2)
```

(caso `Decl`: substituição de `f` por `fun y -> y+1`)

```
decl g = fun x -> x+((fun y -> y+1)(x))
   in g(2)
```

(caso `Decl`: substituição de `g` por `(fun x -> x+((fun y -> y+1) (x)))`

```
(fun x -> x+(fun y -> y+1) (x)) (2)
```

(caso `Call`: substituição de `x` por `2`)

```
2+((fun y -> y+1) (2))
```

(caso `Call`: substituição de `y` por `2`)

```
2+(2+1)
```

(caso `Add`)

```
5.
```

```
data Value =
     Number Int
  |  Abs String CalcF
  deriving (Eq,Show)

type Env = [(String,Value)]
...
eval :: CalcF -> Env -> StateMaybe Int Value
...
eval (Fun x e)   env = return (Abs x e)
eval (Call e e') env =
      do
      f <- eval e env
      a <- eval e' env
      case f of
        Abs x e'' -> eval e'' (assoc env x a)
        _ -> raise_error
```

Figura 5: Semântica operacional com ambientes para a linguagem *CALCF*.

4 Semântica com ambientes

Tal como foi visto anteriormente, é conveniente implementar um interpretador com recurso a estruturas de dados auxiliares que evitem a substituição explícita na árvore de sintaxe abstrata (re-escrita). Intuitivamente, a semântica da chamada de função é definida pela avaliação do corpo da função num ambiente onde o parâmetro está associado ao valor do argumento (ver Figura 5).

Usando essa semântica podemos observar novamente a avaliação do exemplo anterior:
(Ambiente inicial vazio `[]`)

```
decl x=1 in
  decl f = fun y -> y+x in
    decl g = fun x -> x+f(x)
      in g(2)
```

(caso `Decl`: Ambiente `[(x,1)]`)

```
decl f = fun y -> y+x in
  decl g = fun x -> x+f(x)
    in g(2)
```

(caso `Decl`: Ambiente `[(f, fun y->y+x),(x,1)]`)

```
decl g = fun x -> x+f(x)
   in g(2)
```

(caso `Decl`: Ambiente `[(g,fun x -> x+f(x)),(f, fun y->y+x),(x,1)]`)

```
g (2)
```

(caso `Call`: avaliação do corpo da função no ambiente
`[(x,2),(g,fun x -> x+f(x)),(f, fun y->y+x),(x,1)]`)

```
x+f(x)
```

(caso `Call`: avaliação do corpo da função no ambiente
`[(y,2),(x,2),(g,fun x -> x+f(x)),(f, fun y->y+x),(x,1)]`)

```
x+(y+x)
```

(casos `Id` e `Add`)

```
6
```

Para a mesma expressão observamos denotações diferentes. A noção de ligação de um identificador que foi proposta nas semânticas anteriores quebrou-se na última semântica. A ocorrência ligada do identificador `x` na expressão `fun y->y+x` que devia corresponder à ocorrência ligante de `x` ao valor `1`, corresponde nesta semântica à ligação do identificador `x` ao valor `2` na avaliação do corpo da abstração. As duas semânticas são portanto diferentes.

A semântica da Figura 5 implementa o que se chama **resolução dinâmica de nomes**. Este tipo de resolução de nomes não respeita o princípio da substutividade enunciado anteriormente. E pode causar erros de execução que não são esperados pelo programador. Considere como exercício determinar a denotação da expressão:

```
let f = let x = 1 in fun y -> y + x in f(2)
```

A resolução dinâmica de nomes torna-se por vezes mais fácil de implementar e existe em ambientes simplificados como shell scripts e outros semelhantes, e até em versões iniciais da linguagem JavaScript. No entanto torna-se pouco prática por poder originar comportamentos inesperados e erros de execução.

5 Semântica com fechos

O problema da semântica com resolução dinâmica de nomes está na quebra das ligações declaradas que acontece ao armazenar expressões abertas no ambiente. Para implementar uma semântica segundo o princípio da substutividade, ou seja, com **resolução estática de nomes**, é necessário que o ambiente guarde apenas valores contendo expressões fechadas. Para isso introduzimos a noção de fecho. Um fecho (para uma abstração) é um triplo do conjunto

$$String \times CALCF \times Env$$

```
type Env a = [(String,a)]

data Value =
     Number Int
  |  Closure String CalcF (Env Value)
  deriving (Eq,Show)

eval :: CalcF -> Env Value -> StateMaybe Int Value
...
eval (Fun x e) env = return (Closure x e env)

eval (Call e e') env =
      do
      f <- eval e env
      a <- eval e' env
      case f of
        Closure x e'' env' -> eval e'' (assoc env' x a)
        _ -> raise_error
```

Figura 6: Semântica operacional com fechos para a linguagem *CALCF*.

Sendo que os triplos contêm um identificador (parâmetro), uma expressão e um ambiente. Num fecho, o conjunto de identificadores livres da expressão é um sub-conjunto do conjunto que contém o parâmetro e os identificadores para os quais o ambiente fornece uma denotação.

Assim, uma abstração tem como denotação um fecho, contendo o ambiente da sua definição, que contém a denotação para todos os seus nomes livres. A aplicação de uma abstração consiste em estender esse mesmo ambiente com a ligação entre parâmetro e argumento. A semântica que contempla esta modificação está resumida na Figura 6. Dada esta semântica, a avaliação da expressão do exemplo anterior corresponderia ao esperado na semântica baseada em substituição.
(Ambiente inicial vazio `[]`)

```
decl x=1 in
  decl f = fun y -> y+x in
    decl g = fun x -> x+f(x)
      in g(2)
```

(caso `Decl`: Ambiente `[(x,1)]`)

```
decl f = fun y -> y+x in
  decl g = fun x -> x+f(x)
    in g(2)
```

(caso `Decl`: Ambiente `[(f, fun y->y+x,[(x,1)]),(x,1)]`)

```
decl g = fun x -> x+f(x)
   in g(2)
```

(caso `Decl`: Ambiente
`[(g,fun x -> x+f(x),[(f, fun y->y+x,[(x,1)]),(x,1)]),(f, fun y->y+x,[(x,1)]),(x,1)]`)

```
g (2)
```

(caso `Call`: avaliação do corpo da função no ambiente do fecho
`[(x,2),[(f, fun y->y+x,[(x,1)]),(x,1)]`)

```
x+f(x)
```

(caso `Call`: avaliação do corpo da função no ambiente do fecho
`[(y,2),(x,1)]`)

```
y+x  (3)
```

(casos `Id` e `Add`)

```
2+3  (5)
```

Esta semântica corresponde ao que foi inicialmente definido usando substituição e resolução estática de nomes.

Como complemento a estas notas pode consultar os slides fornecidos, nomeadamente na animação da avaliação de expressões usando ambientes e fechos (ICLP-7, pp 52-62).

A Semântica com substituição

```
module CalcF where

{- ©2013 João Costa Seco, ICL DI-FCT-UNL -}

import StateMaybeMonad

{- Definition of the Abstract Syntax of the language CalcF -}

data CalcF =
    Num Int
  | Add CalcF CalcF
  | Sub CalcF CalcF
  | Mul CalcF CalcF
  | Div CalcF CalcF

  | Id String
  | Decl String CalcF CalcF
```

```
  | Fun String CalcF
  | Call CalcF CalcF
  deriving (Eq,Show)

{- Examples -}

a = Id "x"

b = Decl "x" (Num 1) (Add a (Num 1))

c = Decl "x" (Id "x") b

d = Fun "y" (Add (Id "y") (Num 1)) {- fun y -> y+1 -}

e = Call (Fun "x" (Call (Id "x") (Num 2))) d

{- Free identifiers -}

free :: CalcF -> [String]
free (Id x)        = [x]
free (Decl x e e') = (free e) ++ (filter (\y -> x /= y) (free e'))
free (Num n)       = []
free (Add e e')    = (free e) ++ (free e')
free (Sub e e')    = (free e) ++ (free e')
free (Mul e e')    = (free e) ++ (free e')
free (Div e e')    = (free e) ++ (free e')
free (Fun x e )    = (filter (\y -> x /= y) (free e))
free (Call e e')   = (free e) ++ (free e')


{- Substitution function, does not avoid identifier capturing -}

subst :: CalcF -> String -> CalcF -> CalcF

subst e x (Id y) | x == y            = e
subst e x (Id y) | x /= y            = Id y

subst e x (Decl y e' e'') | x == y   = (Decl y (subst e x e') e'')
subst e x (Decl y e' e'') | x /= y   = (Decl y (subst e x e') (subst e x e''))

subst e x (Num n)                    = Num n
subst e x (Add e' e'')               = Add (subst e x e') (subst e x e'')
subst e x (Sub e' e'')               = Sub (subst e x e') (subst e x e'')
```

```
subst e x (Mul e' e'')              = Mul (subst e x e') (subst e x e'')
subst e x (Div e' e'')              = Div (subst e x e') (subst e x e'')

subst e x (Fun y e') | x == y       = (Fun y e')
subst e x (Fun y e') | x /= y       = (Fun y (subst e x e'))

subst e x (Call e' e'')             = Call (subst e x e') (subst e x e'')


toNum :: StateMaybe Int CalcF -> StateMaybe Int Int
toNum (SM(f)) = SM( \s0 -> case f s0 of
                           (s1, Just (Num n)) -> (s1,Just n)
                           (s1, Just _) -> (s1,Nothing)
                           (s1, Nothing) -> (s1,Nothing))

 ------------------------------------------------------------------------------
{- Semantics for closed expressions and capture avoiding substitutions         -}
 ------------------------------------------------------------------------------

eval :: CalcF -> StateMaybe Int CalcF

eval (Num n) = return (Num n)

eval (Add e e') =
     do
     l <- toNum (eval e)
     r <- toNum (eval e')
     return (Num (l+r))

eval (Sub e e') =
     do
     l <- toNum (eval e)
     r <- toNum (eval e')
     return (Num (l-r))

eval (Mul e e') =
     do
     l <- toNum (eval e)
     r <- toNum (eval e')
     return (Num (l*r))

eval (Div e e') =
     do
     l <- toNum (eval e)
```

```
     r <- toNum (eval e')
     if r == 0 then raise_error
     else return (Num (div l r))

eval (Decl x e e') =
     do
     x' <- newId
     e'' <- return (subst (Id x') x e')
-- replacing the declared identifier by a new one to avoid capturing of identifiers
     eval (subst e x' e'')

eval (Id x) = raise_error

eval (Fun x e ) = return (Fun x e)

eval (Call e e' ) =
      do
      f <- eval e
      a <- eval e'
      case f of
        Fun x e'' -> eval (subst a x e'')
        _ -> raise_error

runstate :: CalcF -> Maybe CalcF
runstate e = let SM(f) = (eval e) in snd (f 0)
```

B Semântica com ambientes (dynamic scoping)

```
module CalcF where

{- ©2013 João Costa Seco, ICL DI-FCT-UNL -}

import StateMaybeMonad

{- Definition of the Abstract Syntax of the language CalcF -}

data CalcF =
    Num Int
  | Add CalcF CalcF
  | Sub CalcF CalcF
  | Mul CalcF CalcF
  | Div CalcF CalcF
```

```
  | Id String
  | Decl String CalcF CalcF

  | Fun String CalcF
  | Call CalcF CalcF
  deriving (Eq,Show)

data Value =
     Number Int
  |  Abs String CalcF
  deriving (Eq,Show)

{- Examples -}

a = Id "x"

b = Decl "x" (Num 1) (Add a (Num 1))

c = Decl "x" (Id "x") b

d = Fun "x" (Add (Id "x") (Num 1))

toNum :: StateMaybe Int Value -> StateMaybe Int Int
toNum (SM(f)) = SM( \s0 -> case f s0 of
                           (s1, Just (Number n)) -> (s1,Just n)
                           (s1, Just (Abs _ _)) -> (s1,Nothing)
                           (s1, Nothing) -> (s1,Nothing))

 ------------------------------------------------------------------------------
{- Semantics for open expressions with environments                            -}
 ------------------------------------------------------------------------------

{- Environment managment -}

type Env = [(String,Value)]

find :: String -> Env -> Maybe Value
find x env = if l == [] then Nothing else Just (head l)
             where l = [ v | (k,v) <- env, k == x]

assoc :: Env -> String -> Value -> Env
assoc env x l = ((x,l):env)

{- -}
```

```
eval :: CalcF -> Env -> StateMaybe Int Value

eval (Id x) env =
        case find x env of
          Nothing -> raise_error
          Just v -> return v

eval (Decl x e e') env =
        do
        l <- eval e env
        eval e' (assoc env x l)

eval (Num n) env = return (Number n)

eval (Add e e') env =
     do
     l <- toNum (eval e env)
     r <- toNum (eval e' env)
     return (Number (l+r))

eval (Sub e e') env =
     do
     l <- toNum (eval e env)
     r <- toNum (eval e' env)
     return (Number (l-r))

eval (Mul e e') env =
     do
     l <- toNum (eval e env)
     r <- toNum (eval e' env)
     return (Number (l*r))

eval (Div e e') env =
     do
     l <- toNum (eval e env)
     r <- toNum (eval e' env)
     if r == 0 then raise_error
     else return (Number (div l r))

eval (Fun x e) env = return (Abs x e)

eval (Call e e') env =
      do
```

```
      f <- eval e env
      a <- eval e' env
      case f of
        Abs x e'' -> eval e'' (assoc env x a)
        _ -> raise_error

runstate :: CalcF -> Maybe Value
runstate e = let SM(f) = (eval e []) in snd (f 0)
```

C Semântica com fechos (static scoping)

```
module CalcF where

{- ©2013 João Costa Seco, ICL DI-FCT-UNL -}

import StateMaybeMonad

{- Definition of the Abstract Syntax of the language CalcF -}

data CalcF =
    Num Int
  | Add CalcF CalcF
  | Sub CalcF CalcF
  | Mul CalcF CalcF
  | Div CalcF CalcF

  | Id String
  | Decl String CalcF CalcF

  | Fun String CalcF
  | Call CalcF CalcF
  deriving (Eq,Show)

{-

(fun x -> x+1) (y+2)

-}

type Env a = [(String,a)]

data Value =
```

```
    Number Int
  |  Closure String CalcF (Env Value)
  deriving (Eq,Show)

toNum :: StateMaybe Int Value -> StateMaybe Int Int
toNum (SM(f)) = SM( \s0 -> case f s0 of
                           (s1, Just (Number n)) -> (s1,Just n)
                           (s1, Just _) -> (s1,Nothing)
                           (s1, Nothing) -> (s1,Nothing))

{- Environment managment -}


find :: Eq a => String -> Env a -> Maybe a
find x env = if l == [] then Nothing else Just (head l)
             where l = [ v | (k,v) <- env, k == x]

assoc :: Env a -> String -> a -> Env a
assoc env x l = ((x,l):env)

{- -}


eval :: CalcF -> Env Value -> StateMaybe Int Value

eval (Id x) env =
        case find x env of
          Nothing -> raise_error
          Just v -> return v

eval (Decl x e e') env =
        do
        l <- eval e env
        eval e' (assoc env x l)

eval (Num n) env = return (Number n)

eval (Add e e') env =
     do
     l <- toNum (eval e env)
     r <- toNum (eval e' env)
     return (Number (l+r))

eval (Sub e e') env =
```

```
     do
     l <- toNum (eval e env)
     r <- toNum (eval e' env)
     return (Number (l-r))

eval (Mul e e') env =
     do
     l <- toNum (eval e env)
     r <- toNum (eval e' env)
     return (Number (l*r))

eval (Div e e') env =
     do
     l <- toNum (eval e env)
     r <- toNum (eval e' env)
     if r == 0 then raise_error
     else return (Number (div l r))

eval (Fun x e) env = return (Closure x e env)

eval (Call e e') env =
      do
      f <- eval e env
      a <- eval e' env
      case f of
        Closure x e'' env' -> eval e'' (assoc env' x a)
        _ -> raise_error

runstate :: CalcF -> Maybe Value
runstate e = let SM(f) = (eval e []) in snd (f 0)
```