Interpretação e Compilação de Linguagens de Programação
Linguagens Imperativas

15 de Março de 2013

1 Introdução

As linguagens imperativas distinguem-se das linguagens ditas funcionais pela existência de efeitos laterais na avaliação das suas sub-expressões ou sub-programas. Nas linguagens funcionais, as expressões denotam sempre valores puros e imutáveis, e a avaliação de uma expressão denota sempre o mesmo valor. Nas linguagens imperativas os identificadores denotam por vezes localizações de memória mutáveis e a avaliação de expressões ou comandos, depende não só da denotação dos seus identificadores livres mas também do estado corrente da memória.

Para poder estudar as linguagens imperativas apresentamos primeiro um modelo de memória que vamos integrar na definição da semântica operacional das linguagens. Discutimos a diferença entre ambiente e memória, a noção de aliasing, de tempo de vida de uma célula de memória. Apresenta-se uma linguagem de programação com células mutáveis, *CALCState*, e a respectiva semântica operacional.

2 Modelo de memória

As operações fundamentais de uma linguagem imperativa estão relacionadas com a alocação (e dealocação) de localizações de memória, a sua modificação e consulta. A definição de uma linguagem imperativa pressupõe a definição precisa destas operações em relação a uma definição precisa do que é uma memória. Assim, pode-se definir uma memória como um conjunto potencialmente infinito de células mutáveis, i.e. cujo conteúdo pode mudar no decorrer da execução de um programa. A cada célula de memória corresponde um identificador, um apontador ou endereço de memória (C,C++) ou referência (Java, OCaml, etc.). As referências são valores cuja denotação só faz sentido em relação a um estado da memória concreto. Assim podemos definir uma memória como um tipo abstracto de dados *Mem* com as

```
interface Memory {
  Ref new(Value v);
  void free(Ref r);
  void set(Ref r, Value v);
  Value get(Ref r);
}
```

Figura 1: Interface Java para o modelo de memória.

seguintes operações:

$$\texttt{new} : \mathit{Mem} \times \mathit{Value} \rightarrow \mathit{Ref} \times \mathit{Mem}$$

$$\texttt{free} : \mathit{Mem} \times \mathit{Ref} \rightarrow \mathit{Mem}$$

$$\texttt{set} : \mathit{Mem} \times \mathit{Ref} \times \mathit{Value} \rightarrow \mathit{Mem}$$

$$\texttt{get} : \mathit{Mem} \times \mathit{Ref} \rightarrow \mathit{Value}$$

Note-se que as operações são aqui definidas de forma funcional, e que devolvem o estado seguinte da memória após a operação. Num modelo imperativo e orientado por objetos, tal poderia ser especificado pelo interface da Figura 1, sendo o tipo `Value` atribuído aos valores da linguagem, e o tipo `Ref` (subtipo de `Value`) atribuído aos valores que são referências.

O desenho de uma linguagem imperativa resulta da utilização combinada de um ambiente e de uma memória. Sendo que o ambiente mantém a ligação (imutável) entre um identificador e a sua denotação ou valor e a memória mantém a relação (mutável) entre uma localização (ou referência) e um valor. Na Figura 3 pode-se observar o ambiente e a memória no ponto assinalado no programa da Figura 2. Com a utilização livre de referências memória é possível construir estruturas de dados dinâmicas, e também é possível delinear cenários onde existem mais do que uma maneira de aceder à mesma posição de memória. Este fenómeno é conhecido por *aliasing* e está na base da programação imperativa, no entanto, pode também levar a problemas. Veja-se por exemplo a função `revert` da Figura 4, que copia de um array para outro invertendo as posições dos elementos. Se esta função for chamada usando um só array para os dois parâmetros (onde `a` e `b` referem as mesmas localizações de memória), o conteúdo do array resultado (`a`) não será o esperado.

A alocação de memória nas linguagens imperativas aparece de variadas formas e com nuances diferentes em relação ao tipo de memória alocada, ao tempo de vida dessa memória e como esse tempo de vida é gerido. A declaração de uma variável local de tipo inteiro em C, ou a alocação de uma variável de tipo inteiro em OCaml é completamente diferente. A alocação de um objeto em Java ou em C++ é também diferente, sendo que em C++, um objeto pode ser alocado na stack ou explicitamente na heap. A alocação de um array é diferente em Java ou em C.

```
const int TEN = 10;

int main(void) {
    int s = 0;
    int a[TEN] = {1,2,3,4,5,6,7,8,9};
    int *b = a;

    for(int i = 0; i < TEN; i++) {
        s = s + *b; // <--
        b++;
    }
    printf("%d\n",s);
}
```

Figura 2: Excerto de programa imperativo.

Ambiente	
`TEN`	10
`main`	...
`s`	ℓ_0
`a`	ℓ_1
`b`	ℓ_2
`i`	ℓ_3
`printf`	...

Memória	
ℓ_0	0
ℓ_1	$\{1, 2, \ldots\}$
ℓ_2	ℓ_1
ℓ_3	0

Figura 3: Instante do ambiente e memória no ponto assinalado na Figura 2

Outra diferença que se pode observar entre várias linguagens é na desreferenciação de referências de memória. A expressão na linguagem C, `x=x+1` tem duas ocorrências do identificador `x`, mas onde as duas ocorrências têm denotações diferente. A ocorrência à esquerda do operador de afetação (`=`) denota a localização de memória a ser modificada, enquanto a ocorrência à direita denota o valor contido na localização de memória associada a `x`. Neste tipo de linguagens (C, Java, C#, etc.) há uma conversão implícita entre localizações e o seu conteúdo. A um valor do tipo localização, utilizado num contexto onde se pretende modificar o estado da memória, dá-se a categoria de *L-value* e ao correspondente conteúdo dá-se o nome de *R-value*. A conversão implícita de *L-value* para *R-value* faz-se de acordo com o contexto da ocorrência do identificador. Uma expressão equivalente em OCaml, onde as referências são tratadas explicitamente seria `x:=!x+1` onde `!x` significa exactamente o conteúdo da referência denotada por `x`. Na secção B do apêndice encontra-se a semântica de uma linguagem com desreferênciação implícita. Em geral, um *L-value* é convertido para um *R-value* quando o contexto de uso é o de um valor e não o de uma localização de memória. Note-se que na

```
void revert(int *a, int* b, int size) {
    for(int i = 0; i < size; i++) {
        a[i] = b[size-1-i];
    }
}
```

Figura 4: Função `revert`.

expressão `a[a[2]]:=a[1]`, as expressões `a[2]` e `a[1]` são convertidas para *R-value*. No processo de compilação, a geração de código para a conversão entre referências e o seu conteúdo é guiada pelo resultado da análise de tipos.

A alternativa à desreferenciação implícita de referências baseada em valores *L-value* e *R-value* é a alocação, manipulação direta de apontadores e desreferênciação explícita como na linguagem OCaml, ou na utilização de apontadores em C. A expressão acima seria escrita `a[!a[2]] := !a[1]`.

3 Linguagem CALCState

Apresentamos agora uma linguagem imperativa *CALCState* definida por extensão da linguagem *CALCI* com operações para lidar diretamente com a memória.

1. `num` : $Integer \rightarrow CALCState$
2. `add` : $CALCState \times CALCState \rightarrow CALCState$
3. `sub` : $CALCState \times CALCState \rightarrow CALCState$
4. `mul` : $CALCState \times CALCState \rightarrow CALCState$
5. `div` : $CALCState \times CALCState \rightarrow CALCState$
6. `id` : $String \rightarrow CALCState$
7. `decl` : $String \times CALCState \times CALCState \rightarrow CALCState$
8. `var` : $CALCState \rightarrow CALCState$
9. `assign` : $CALCState \times CALCState \rightarrow CALCState$
10. `deref` : $CALCState \rightarrow CALCState$
11. `free` : $CALCState \rightarrow CALCState$
12. `seq` : $CALCState \times CALCState \rightarrow CALCState$
13. `while` : $CALCState \times CALCState \rightarrow CALCState$
14. `If` : $CALCState \times CALCState \times CALCState \rightarrow CALCState$

```
newtype StateMaybe s a = SM( s -> (s, Maybe a) )

instance Monad (StateMaybe s)
  where
    -- (>>=) :: StateMaybe s a -> (a -> StateMaybe s b) -> StateMaybe s b
    (SM p) >>= k  =  SM( \s0 -> case p s0 of
                                  (s1, Just a) -> let (SM q) = k a in q s1
                                  (s1, Nothing) -> (s1,Nothing))

    -- return :: a -> StateTrans s a
    return a = SM( \s -> (s, Just a) )
```

Figura 5: Monad StateMaybe.

A semântica desta linguagem baseia-se na utilização de uma memória modelada pelas operações apresentadas na secção 2 e onde se registam os efeitos das expressões `var`, `assign` e `free`. Numa semântica com efeitos, é importante definir a sequência pela qual as sub-expressões são avaliadas. A semântica é aqui definida pelo extrato de código ML da Figura 6 por uma função que aceita como parâmetro um ambiente e uma memória e denota um valor e a memória resultante após aplicação dos efeitos da expressão.

Na definição da semântica, a presença de efeitos é clara pela dependência introduzida pelos parâmetros da função de interpretação. No caso da expressão `Add (e,e')` a avaliação da expressão `e` é realizada com base na memória inicial (`mem`) e os seus efeitos são registados na memória que resulta da sua avaliação (`mem'`). A avaliação da sub-expressão `e'` é feita num momento posterior, a partir da memória `mem'` e resulta no efeito final da expressão de adição, a memória `mem''`. Deste encadeamento de chamadas, resulta a especificação de que o operando esquerdo da expressão é avaliado primeiro que o operando direito. Na semântica Haskell apresentada na Figura 5 essa sequênciação fica encapsulada pela utilização do Monad `StateMaybe`, que também trata a propagação de erros da função de execução.

Numa implementação de um interpretador da linguagem *CalcState* numa linguagem imperativa com gestão de memória integrada como Java ou mesmo em OCaml, é mais eficaz usar os mecanismos de gestão de memória já existentes (ver Figuras 7 e 8).

```
type loc = ...

type value =
   Num of int
 | Ref of loc

let rec eval exp env mem =
  match exp with
    ...
  | Add (e,e') ->
      let (v1,mem') = eval e env mem in
      let (v2,mem'') = eval e' env mem' in
      (Num ((toNum v1)+(toNum v2)),mem'')
    ...
  | Var e ->
      let (v,mem') = eval e env mem in
      let (r,mem'') = new_loc mem' v in
      (r,mem'')
  | Assign e e' ->
      let (r,mem') = eval e env mem in
      let (v,mem'') = eval e' env mem' in
      let mem''' = set_loc mem' r v in
      (v,mem''')
  | Deref e ->
      let (r,mem') = eval e env mem in
      (get_loc mem' r,mem')
  | Free e ->
      let (r,mem') = eval e env mem in
      let mem''' = free_loc mem' r v in
      (0,mem'')
```

Figura 6: Semântica usando gestão de memória explícita.

```
type value =
   Num of int
 | Ref of value ref

let rec eval exp env =
  match exp with
    ...
  | Add (e,e') ->
      let v1 = eval e env in
      let v2 = eval e' env in
      Num ((toNum v1)+(toNum v2)
    ...
  | Var e ->
      let v = eval e env in
      let r = ref v in r

  | Assign (e,e') ->
      let r = eval e env in
      let v = eval e' env in
      (toRef r) := v

  | Deref e ->
      let r = eval e env in !(toRef (r)

  | Free e -> (0,mem) (* Não é necessário *)
```

Figura 7: Semântica usando referências OCaml.

```
interface Value {}

class IntValue {
  public final int n;

  IntValue(int n) { this.n = n; }
}

class RefValue {
  Value v;

  RefValue(Value v) { this.v = v; }
  Value get() { return v; }
  void set(Value v) { this.v = v; }
}

class ASTVar {
  ASTNode exp;
  ASTVar(ASTNode exp) { this.exp = exp; }
}
```

Figura 8: Semântica usando referências OCaml.

A Código Haskell

```
module CalcState where

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

import StateMaybeMonad

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

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

  | Id String
  | Decl String CalcState CalcState

  | Var CalcState
  | Deref CalcState
  | Assign CalcState CalcState
  | Free CalcState

  | If CalcState CalcState CalcState
  | While CalcState CalcState
  | Seq CalcState CalcState

  | Equal CalcState CalcState
  | Not CalcState
  deriving (Eq,Show)

type Env = [(String,Value)]

type Mem = [(Int,Value)]

{- Note that the implementation of the memory as a list is for demonstration purposes

type Loc = Int

data Value =
    Number Int
  | Ref Loc
```

```
  | Boolean Bool
  deriving (Eq,Show)

{- Environment manipulation functions -}

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)

{- State manipulation functions -}

newloc :: Value -> StateMaybe Mem Value
newloc v = SM( \m ->
               let loc = length m in ((loc,v):m, Just (Ref loc)))

getloc :: Loc -> StateMaybe Mem Value
getloc l = SM( \m -> (m, Just ((head [v | (l',v)<-m, l==l'])))))

setloc :: Loc -> Value -> StateMaybe Mem Value
setloc loc v = SM (\m ->
                let m' = [(loc,v)] ++ [(l',v) | (l',v) <- m, l' /= loc ] in
                (m',Just v))

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

toRef :: StateMaybe Mem Value -> StateMaybe Mem Int
toRef (SM(f)) = SM( \s0 -> case f s0 of
                           (s1, Just (Ref n)) -> (s1,Just n)
                           (s1, Just (Number n)) -> (s1,Nothing)
                           (s1, Just (Boolean b)) -> (s1,Nothing)
                           (s1, Nothing) -> (s1,Nothing))

toBool :: StateMaybe Mem Value -> StateMaybe Mem Bool
toBool (SM(f)) = SM( \s0 -> case f s0 of
```

```
                          (s1, Just (Ref n)) -> (s1,Nothing)
                          (s1, Just (Number n)) -> (s1,Nothing)
                          (s1, Just (Boolean b)) -> (s1,Just b)
                          (s1, Nothing) -> (s1,Nothing))


{- Operational Semantics -}


eval :: CalcState -> Env -> StateMaybe Mem 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 (Equal e e') env =
     do
     l <- toNum (eval e env)
     r <- toNum (eval e' env)
     return (Boolean (l == r))

eval (Not e) env =
     do
     l <- toBool (eval e env)
     return (Boolean (not l))


eval (Var e) env =
     do
     v <- eval e env
     newloc (v)

eval (Deref e) env =
     do
     loc <- toRef (eval e env)
     getloc(loc)

eval (Assign e e') env =
     do
     loc <- toRef (eval e env)
     v <- eval e' env
     setloc loc v

eval (If e e' e'') env =
     do
     v <- eval e env
     case v of
          Boolean True -> eval e' env
          Boolean False -> eval e'' env

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

```
eval (While e e') env =
     do
     v <- eval e env
     case v of
          Boolean True -> eval (Seq e' (While e e')) env
          Boolean False -> return (Boolean False)

runeval :: CalcState -> Env -> Maybe Value
runeval e env = let SM(m) = (eval e env) in snd (m [])

a = (Decl "x" (Var (Num 0)) (Seq (While (Not (Equal (Deref (Id "x")) (Num 10))) (Assi
```

B Conversão implícita de referências

```
module CalcStateImplicit where

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

import StateMaybeMonad

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

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

  | Id String
  | Var String CalcState CalcState

  | Assign String CalcState

  | If CalcState CalcState CalcState
  | While CalcState CalcState
  | Seq CalcState CalcState

  | Equal CalcState CalcState
  | Not CalcState
  deriving (Eq,Show)
```

```
type Loc = Int

type Env = [(String,Loc)]

type Mem = [(Loc,Value)]

{- Note that the implementation of the memory as a list is for demonstration purposes

data Value =
    Number Int
  | Boolean Bool
  deriving (Eq,Show)

{- Environment manipulation functions -}

find :: String -> Env -> Maybe Loc

find x env = if l == [] then Nothing else Just (head l)
             where l = [ v | (k,v) <- env, k == x]

assoc :: Env -> String -> Loc -> Env

assoc env x l = ((x,l):env)

{- State manipulation functions -}

newloc :: Value -> StateMaybe Mem Loc
newloc v = SM( \m ->
               let loc = length m in ((loc,v):m, Just loc))

getloc :: Loc -> StateMaybe Mem Value
getloc l = SM( \m -> (m, Just ((head [v | (l',v)<-m, l==l'])))))

setloc :: Loc -> Value -> StateMaybe Mem Value
setloc loc v = SM (\m ->
                let m' = [(loc,v)] ++ [(l',v) | (l',v) <- m, l' /= loc ] in
                (m',Just v))

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

```
toBool :: StateMaybe Mem Value -> StateMaybe Mem Bool
toBool (SM(f)) = SM( \s0 -> case f s0 of
                          (s1, Just (Number n)) -> (s1,Nothing)
                          (s1, Just (Boolean b)) -> (s1,Just b)
                          (s1, Nothing) -> (s1,Nothing))


{- Operational Semantics -}


eval :: CalcState -> Env -> StateMaybe Mem Value

eval (Id x) env =
        case find x env of
          Nothing -> raise_error
          Just loc -> getloc loc

eval (Var x e e') env =
        do
        v <- eval e env
        l <- newloc v
        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 (Equal e e') env =
     do
     l <- toNum (eval e env)
     r <- toNum (eval e' env)
     return (Boolean (l == r))

eval (Not e) env =
     do
     l <- toBool (eval e env)
     return (Boolean (not l))

eval (Assign x e') env =
     do
     v <- eval e' env
     case find x env of
          Nothing -> raise_error
          Just loc -> setloc loc v

eval (If e e' e'') env =
     do
     v <- eval e env
     case v of
          Boolean True -> eval e' env
          Boolean False -> eval e'' env

eval (Seq e e') env =
     do
     eval e env
     eval e' env

eval (While e e') env =
     do
     v <- eval e env
     case v of
          Boolean True -> eval (Seq e' (While e e')) env
          Boolean False -> return (Boolean False)

runeval :: CalcState -> Maybe Value
```

```
runeval e = let SM(m) = (eval e []) in snd (m [])

a = (Var "x" (Num 0) (Seq (While (Not (Equal (Id "x") (Num 10))) (Assign "x" (Add (Id
```