Interpretação e Compilação de Linguagens (de Programação)

21/22

Luís Caires (http://ctp.di.fct.unl.pt/~lcaires/)

Mestrado Integrado em Engenharia Informática

Departamento de Informática

Faculdade de Ciências e Tecnologia

Universidade Nova de Lisboa

Imperative Languages

Till now, expressions of our language have denoted pure values. and every expression always denotes an immutable fixed value in a given scope.

Imperative languages (C, Java, …) introduce mutable values (memory cells) and fundamental imperative operations:

- Allocation of memory (var x:Integer; int x;)
- Operations to read / write memory (e.g., x := 2, y = y + 2)

- Memory Model (new, set, get, free)
- Environment versus memory
- Aliasing
- L-value e R-value
- lifetime versus scope
- References, pointers, etc
- Intepreter for imperative language

Basic Memory Model

- Memory: dynamic store of memory cells, each with a mutable content.
- Each memory cell has a unique designator (the cell reference or address)
- We assume general cells, that can store any value of the language.
- The memory contais a pool of unused cells (the free pool), the other cells are considered in use by the executing program.
- A cell reference is also a value of a special (opaque) data type ref
- Interface for memory $\mathcal{M}$

new:	$\mathcal{M} \times \text{Value} \rightarrow \textbf{ref}$
set:	$\mathcal{M} \times \textbf{ref} \times \text{Value} \rightarrow \textbf{void}$
get:	$\mathcal{M} \times \textbf{ref} \rightarrow \text{Value}$
free:	$\mathcal{M} \times \textbf{ref} \rightarrow \textbf{void}$

Basic Memory Model

- Operations for a memory $\mathcal{M}$, functionally defined

new: $\mathcal{M} \times \text{Value} \rightarrow \mathcal{M} \times \textbf{ref}$

Guess back a reference for a newly allocated from the free pool.

set: $\mathcal{M} \times \textbf{ref} \times \text{Value} \rightarrow \mathcal{M}$

Mutates (changes) the value stored in the cell ref . The previous value is lost.

get: $\mathcal{M} \times \textbf{ref} \rightarrow \mathcal{M} \times \text{Value}$

Returns the value stores in the cell ref.

free: $\mathcal{M} \times \textbf{ref} \rightarrow \mathcal{M}$

releases the cell ref to the free pool.

Environment versus Memory

- The environment gives the value associated to every identifier declared in the program and reflects the static structure of such program (nesting of scopes).
- In the environment the binding between an identifier and its value is fixed and immutable. The value is bound just once using the assoc() operation.

- The memory contains a set of mutable cells, each cell is named by a reference value and holds a value.
- The value stored in a reference may be changed during execution, using assignment operations (e,.g., X := E),
- We may refer to a reference using an identifier (usually called a "state variable"), The binding between the name of a “state variable” and its associated memory location is defined by the ambiente. This binding is immutable in its scope.

Environment versus Memory

Environment

Identifier	Value
PI	3,14
x	loc_0
k	loc_1
j	loc_1
TEN	10

Memory

Reference	Stored Value
loc_0	25
loc_1	12
loc_2	loc_1
...	...
loc	0

Environment versus Memory

Environment

Identifier	Value
PI	3,14
x	0x00FF
k	0x0100
j	0x0100
TEN	10

Memory

Reference	Stored Value
0x00FF	25
0x0100	12
0x0102	0x0100
...	...
0xFFFF	0

Memory Model (properties)

Identifier	Value
PI	3,14
x	loc_0
k	loc_1
j	loc_1
TEN	10

Reference	Stored Value
loc_0	25
loc_1	12
loc_2	loc_1
...	...
loc	0

The same memory cell may be bound to different names (aliasing).

Aliasing

- Different names / expressions may refer to the same memory cell.

```
class A {
  int x;
  boolean equals(A b) { return x == b.x}
}
A a = new A(); a.equals(a);


int x = 0;
void f(int* y) { *y = x+1; }
...
f(&x);
// x = ?
```

Memory Model (properties)

Identifier	Value
PI	3,14
x	loc_0
k	loc_1
j	loc_1
TEN	10

Reference	Stored Value
loc_0	25
loc_1	12
loc_2	loc_1
...	...
loc	0

References are values (first-class references)

A cell may store a reference to other cell, allowing the manipulation of dynar
data structures, and even cyclic data structures.

Language level imperative primitives

- Allocates a new cell, initialises it with value of expression E and returns the reference

  ref(E)

- We may this kind of operation more or less explicit in all imperative programming languages:

```
{
  int a = 2;
  MyClass m;
  …
}

new int[10];

malloc(sizeof(int));

new MyClass();
```

Language level imperative primitives

- Assignment of a value to a reference cell

E := F

Expression E denotes a reference cell, expression F denotes some value

- Assignments are present in programming languages in many forms:

```
a = a + 1

i := 2

b[x+2][b[x-2]] = 2

*(p+2) = y

myTable(i,j) = myTable(j,i)

Readln(MyLine);
```

Language level imperative primitives

- Dereference of the memory cell denoted by expression E.

!E

- We may find the presence of dereference in many forms:

```
i := !i + 1          (OCAML)

*p                   (C)

i = i + 1            (Java)

i++                  (Java)
```

Language level imperative primitives

- Dereference of the memory cell denoted by expression E.

!E

- We may find the presence of dereference in many forms:

```
i := !i + 1          (OCAML)

*p                   (C)

i = i + 1            (Java)

i++                  (Java)
```

references

Language level imperative primitives

- Dereference of the memory cell denoted by expression E.

!E

- We may find the presence of dereference in many forms:

```
i := !i + 1          (OCAML)

*p                   (C)

i = i + 1            (Java)

i++                  (Java)
```

denotes contents of i

(implicit dereference) L-Value e R-Value

- If expression E denotes a reference, most programming languages interprets E in a context dependent way

E := 2

- (Left-Value) To the left of the assignment symbol, denotes its true value (a reference)

E := E + 1

- (Right-Value) To the left of the assignment symbol, implicitly denotes the contents of the reference cell, without explicit dereference

E := !E + 1

(desreferenciação implícita) L-Value e R-Value

- If expression E denotes a reference, most programming languages interprets E in a context dependent way,
- NB. The terminology "L-Value" e "R-Value" although standard (you should know it) is not very sound.
- For example, consider, e.g.,

```
A[A[2]] := A[2] + 1
```

- In this statement, both subexpressions **A[2]**, one to the left and other to the rught are dereferenced implicitly, However A[A[2]] to the left is not dereferenced (so that it denotes a reference)

```
A[!A[2]] := !A[2] + 1
```

Explicit deference

- The explicit dereference operation **!E** allows the semantics of programs to be more precise, context free, and escapes any ambiguity.

```
A[!A[2]] := !A[2] + 1
```

- On the other hand, the use of dereference **!-** make programs more verbose. Most programming languages adopt implicit deference, for historical reasons.
- NB. The implicit dereference may be better understood as a coercion(cast) operation in which the interpreter compiler inserts the missing !
- To do this, types of expression must be known at evaluation / compilation time.
- In our language we will adopt uniform explicit dereference.

Language level imperative primitives

- All patterns of use mutable state in programming languages can be expressed using the basic imperative primitives

ref(E)	allocation
free(E)	release
E := E	assignement
! E	dereference

```
{
/* linguagem C */

  const int k = 2;
  int a = k;
  int b = a + 2;
  …
  b = a * b + k
  …
}
```

```
def
  k = 2
  a = ref(k)
  b = ref(!a+2)
in
  …
  b := !a * !b + k
  …

end
```

Operações Imperativas

- Todas as declarações e usos de variáveis mutáveis podem exprimir-se usando as primitivas

var(E **)**	instanciação
free(E **)**	libertação
E **:=** E	afectação
! E	desreferenciação

```
{
/* linguagem C */

  const int k = 2;
  int a = k;
  int b = a + 2;
  …
  b = a * b
  …
}
```

```
def
  k = 2
  a = var(k)
  b = var(!a+2)
in
  …
  b := !a * !b
  …
end
```

libertação implícita (das células atribuídas aos ids a e b)

Operações Imperativas

- Todas as declarações e usos de variáveis mutáveis podem exprimir-se usando as primitivas

var(E **)**	instanciação
free(E **)**	libertação
E **:=** E	afectação
! E	desreferenciação

```
{
/* linguagem C++ */

  int k = 2;
  const int *a = &k;
  int b = *a;
  … *a = k+b …

}
```

```
def
  k = var(2)
  a = k
  b = var(!a)
in
  … a := !k+!b …


end
```

Operações Imperativas

- Todas as declarações e usos de variáveis mutáveis podem exprimir-se usando as primitivas

var(E **)**	instanciação
free(E **)**	libertação
E **:=** E	afectação
! E	desreferenciação

```
{
/* linguagem C */

  int k = 2;
  int *a = &k;
  … k = k+*a …

}
```

```
def
  k = var(2)
  a = var(k)
in
  … k := !k+!!a …


end
```

Tempo de Vida (de uma célula)

O tempo de vida de uma célula é o tempo que medeia entre a sua criação / reserva usando var(_) e a sua libertação usando free(_).

- Em muitas situações, o tempo de vida da célula concide com o âmbito do(s) seu(s) identificador.

criação da célula k

```
{
/* linguagem C */

  int k = 2;
  ...
}
```

```
int *f(int j) {
  int k = j;
  return &k;
}
```

libertação implícita da célula k

Tempo de Vida (de uma célula)

O tempo de vida de uma célula é o tempo que medeia entre a sua criação / reserva usando **var(_)** e a sua libertação usando **free(_)**.

- Noutras situações, o tempo de vida da célula extravasa o âmbito do(s) seu(s) identificador.

âmbito de k

```
{
/* linguagem C */

  static int k;
  ...
}
```

O tempo de vida da célula associada a k é o tempo do programa

reserva da célula para k

```
/* linguagem Java */
Integer f(int j) {
  Integer k = new Integer(j);
  return k;
}
```

reserva de novo objecto Integer

há libertação implícita da célula de k mas o objecto sobrevive ao bloco!

Tempo de Vida (de uma célula)

O tempo de vida de uma célula é o tempo que medeia entre a sua criação / reserva usando **var(_)** e a sua libertação usando **free(_)**.

- Noutras situações, o tempo de vida da célula extravasa o âmbito do(s) seu(s) identificador.

âmbito de k

```
{
/* linguagem C */

  static int k = 2;
  ...
}
```

O tempo de vida da célula associada a k é o tempo do programa

reserva da célula para k

```
/* linguagem C */
int* f(int j) {
  int *k = malloc(sizeof(int));
  *k = 2;
  return k;
}
```

reserva de novo bloco de memória

há libertação implícita da célula de k mas a memória reservada predura.

Uma linguagem imperativa (CALCS)

- Consideramos uma só categoria sintáctica para expressões (EXP):

num:	Integer → EXP	
bool:	Boolean → EXP	
id:	String → EXP	
add:	EXP × EXP → EXP	
var:	EXP → EXP	
deref:	EXP → EXP	(!E)
if:	EXP × EXP × EXP → EXP	
while:	EXP × EXP → EXP	
assign:	EXP × EXP → EXP	(x := y + z)
seq:	EXP × EXP → EXP	(S1 ; S2)
def:	String × EXP × EXP → EXP	

Language level imperative primitives

- All patterns of use mutable state in programming languias can be expressed using the basic imperative primitives

ref(E **)**	allocation
free(E **)**	release
E **:=** E	assignement
! E	dereference

```
{
/* linguagem C */

  int a = 2;
  int b = &a;
  …
  a = a + *b
  …
}
```

```
def
  a = ref(2)
  b = ref(a)
in
  …
  a := !a + !!b
  …

end
```

The language CALCS (abstract syntax)

num: Integer → CALCS

bool: Integer → CALCS

id: String → CALCS

add: CALCS × CALCS → CALCS

gt: CALCS × CALCS → CALCS

def: (String × CALCS)+ × CALCS → CALCS

seq: CALCS x CALCS → CALCS

if: CALCS × CALCS × CALCS → CALCS

while: CALCS × CALCS → CALCS

ref: CALCS → CALCS

deref: CALCS → CALCS

assign: CALCS × CALCS → CALCS

println: CALCS → CALCS

The language CALCS (concrete syntax)

```
def
    N = ref(676)
in
    while (!N ~= 1) do
         if (2*(!N/2) = !N) then
             N := !N/2
        else
             N := 3*!N + 1
        end;
        println !N
   end;
  println “HELLO”
end
```

The language CALCS (concrete syntax)

```
def T = 10 in
def a = ref(0) in
  while (!a < T) do
     a := !a + 1;
  end
end
end
```

```
def a = ref(2) in
  def b = ref(!a) in
    def c = a in
      a := !b + 2;
      c := !c + 2
    end
  end
end
```

Sematics of CALCS (schematic)

Algorithm eval( ) that computes the denotation (value) of any **open** CALCS expression:

$$\mathit{eval}: \mathrm{AST} \times \mathrm{ENV} \times \mathrm{MEM} \rightarrow \mathrm{VAL} \times \mathrm{MEM}$$

AST = open programs

ENV = Environments (funções ID → VAL)

MEM = Memories

VAL = Values

Val = Boolean U Integer U Ref

The evaluation map expresses that in general a CALCS program P produces a resulting value and performs a (side) effect (in memory).

Sematics of CALCS (schematic)

Algorithm eval( ) that computes the denotation (value) of any **open** CALCS expression:

$$\mathit{eval}: \mathrm{AST} \times \mathrm{ENV} \times \mathrm{MEM} \rightarrow \mathrm{VAL} \times \mathrm{MEM}$$

eval(**add(**E1, E2**)** , *env* , *m0*) ≜ **[** (*v1* , *m1*) = **eval**(E1, *env*, *m0*);
(*v2* , *m2*) = **eval**(E2, *env*, *m1*);
(*v1* + *v2* , *m2*) **]**

eval(**and(**E1, E2**)** , *env* , *m0*) ≜ **[** (*v1* , *m1*) = **eval**(E1, *env*, *m0*);
(*v2* , *m2*) = **eval**(E2, *env*, *m1*);
(*v1* && *v2* , *m2*) **]**

Sematics of CALCS (schematic)

Algorithm eval( ) that computes the denotation (value) of any **open** CALCS expression:

$$eval : AST \times ENV \times MEM \rightarrow VAL \times MEM$$

eval(**ref**(E) , env , m0) ≜ **[** (v1 , m1) = **eval**(E, env, m0);
(m1.**new**(v1), m1);
]

eval(**!**(E) , env , m0) ≜ **[** (ref , m1) = **eval**(E, env, m0);
(m1.**get**(ref) , m1) **]**

eval(E1 := E2 , env , m0) ≜ **[**(v1 , m1) = **eval**(E1, env, m0);
(v2 , m2) = **eval**(E2, env, m1);
m3 = m2.**set**(v1, v2) ;
(v2 , m3) **]**

Sematics of CALCS (schematic)

Algorithm eval( ) that computes the denotation (value) of any **open** CALCS expression:

$$\mathit{eval}: \mathrm{AST} \times \mathrm{ENV} \times \mathrm{MEM} \rightarrow \mathrm{VAL} \times \mathrm{MEM}$$

```
eval( seq(E1, E2) , env , m0) ≜ [(v1 , m1) = eval( E1, env, m0);
                                 eval( E2, env, m1)
                                ]
eval( if(E1, E2, E3) , env , m0) ≜
                 [  (v1 , m1) = eval( E1, env, m0);
                    if (v1 = true) then eval( E2, env, m1);
                                   else eval( E3, env, m1);
                 ]
```

Sematics of CALCS (schematic)

Algorithm eval( ) that computes the denotation (value) of any **open** CALCS expression:

$$eval: AST \times ENV \times MEM \rightarrow VAL \times MEM$$

```
eval( while(E1, E2) , env , m0) ≜
                    [(v1 , m1) = eval( E1, env, m0 );
                     if (v1 = T) then [ (v2 , m2) = eval( E2, env, m1);
                                  (v,m1) =eval( while(E1, E2), m2) ]
                                 else (F , m1) ]
```

NB. Here we interpret iteration (while) in terms of recursion.

Sematics of CALCS (schematic)

Algorithm eval( ) that computes the denotation (value) of any **open** CALCS expression:

$$\mathit{eval}: \mathrm{AST} \times \mathrm{ENV} \times \mathrm{MEM} \rightarrow \mathrm{VAL} \times \mathrm{MEM}$$

```
eval( def(s, EI, EB) , env , m0) ≜
        [(v1 , m1) = eval( EI, env, m0 );
         env = env.BeginScope();
         env.Assoc(s, v1);
         (v2 , m2) = eval(EB, env, m1);
         env = env.EndScope();
         (v2 , m2) ]
```

NB. The "functional" semantics explicitly specifies an order of evaluation, by threading memory use.

Interpreter for CALCS (Java implementation)

NB. In the object oriented implementation, we use global "reference" objects.

```
class VCell implements IValue
{  IValue v;
   VCell(IValue v0) { v = v0; }
   IValue get() { return v;}
   void set(IValue v0) { v = v0;}
}
```

Interpreter for CALCS (Java implementation)

```
class ASTRef implements ASTNode {
Value eval(Env<IValue> e)
   {  ASTNode exp;
      Value v1 = exp(e);
      return new VCell(v1);
   }
}
```

Interpreter for CALCS (Java implementation)

```
class ASTAssign implements ASTNode {
Value eval(Env<IValue> e)
   {  ASTNode lhs;
      ASTNode rhs;
      Value v1 = lhs.eval(e);
      if (v1 instanceof VCell) {
            Value v2= rhs.eval(e);
            ((VCell)v1).set(v2);
           return v2;
      }
     throw new RuntimeError("Assign: reference expected");
   }
}
```