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

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

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

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

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

Returns the value stores in the cell ref.

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

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

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

```
{
/* C language */

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

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

end
```

Language level imperative primitives

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

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

```
{
/* C language */

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

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

implicit release (of cells denoted by a e b)

Language level imperative primitives

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

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

```
{
/* 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
```

Language level imperative primitives

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

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

```
{
/* C */

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

}
```

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


end
```

Lifetime vs Scope

The lifetime of a memory cell is the time (during program execution) that intermediates bewteen its allocation **new(_)** and its release **free(_)**.

- Sometimes, a memory cell lifetime concides with the execution of the scope of declaration of its associated identifier.

allocation of cell associated to k

```
{
/* C */

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

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

(implicit) release of cell associated to k

Lifetime vs Scope

The lifetime of a memory cell is the time (during program execution) that intermediates bewteen its allocation **new(_)** and its release **free(_)**.

- Most often, a cell lifetime leaks out of the scope in which it is created.

scope of k

```
{
/* linguagem C */

  static int k;
  ...
}
```

In this example, the lifetime of k is the whole program execution

allocation of cell associated to k

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

allocation of Integer object

Here the cell associated to k is implicitly released at the end of scope but the Integer object may survive

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

new: CALCS → CALCS

deref: CALCS → CALCS

assign: CALCS × CALCS → CALCS

println: CALCS → CALCS

The language CALCS (concrete syntax)

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

```
def a = new(2) in
  def b = new(!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 ∪ Integer ∪ 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:

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

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( new(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:

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

```
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:

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

```
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 ASTNew 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");
  }
}
```

Compilation Schemes (CALCS)

Compilation Schemes

We will use compilation schemes to define our compiler

A compilation scheme defines the sequence of instructions generated for a programming language construct

$$[[\ E\ ]]D = \ldots \text{ code sequence } \ldots.$$

- E : program fragment
- D: environment (associates identifiers to "coordinates")

$[[\ E_1 + E_2\ ]]D =$

$[[\ E_1\ ]]D$

$[[\ E_2\ ]]D$

iadd

Compilation Schemes (rel ops)

in the JVM, we represent booleans by integers
(0 - false, 1 - true)

```
[[ E1 > E2 ]]D =
[[E1]]D
[[E2]]D
isub
ifgt L1
sipush 0
goto L2
L1: sipush 1
L2:
```

Compilation Schemes (bool ops)

```
[[ E1 && E2 ]]D =
[[E1]]D
[[E2]]D
iand
```

```
[[ E1 || E2 ]]D =
[[E1]]D
[[E2]]D
ior
```

Compilation Schemes (conditionals)

[[**if** E1 **then** E2 **else** E3]]D =

```
[[E1]]D
ifeq L1
[[E2]]D
goto L2
L1: [[E3]]D
L2:
```

Compilation Schemes (while loop)

```
[[ while E1 do E2 end ]]D =
L1: [[E1]]D
ifeq L2
[[E2]]D
pop
goto L1
L2:
```

Short Circuit Evaluation of Conditionals

Here we explain an alternative, more efficient, compilation scheme for conditionals, in which the value of boolean expressions is not explicitly computed, but directly results in a control flow choice between two jump labels

Short Circuit Evaluation of Conditionals

The code [[BE]] generated by a boolean typed expression BE will leave a boolean value on top of the stack

If BE appears inside a conditional expression, such as an if-then-else or while expression, such boolean value will be immediately consumed by a ifxx branch instruction

Short circuit evaluation of conditionals "skips over" the intermediate boolean value, and compiles a boolean typed expression BE relative to two given labels

- TL (true label)
- FL (false label)

Code [[BE (TL, FL)]] does not change the stack, but will:

- jump to TL if the value of BE is true
- jump to FL if the value of BE is false

Let's see the definition of [[BE (TL, FL)]] for the various boolean valued expressions BE

Short Circuit Evaluation of Conditionals

Code [[BE (TL, FL)]] does not change the stack, but will:

- jump to TL if the value of BE is true
- jump to FL if the value of BE is false

```
[[ true ( TL, FL) ]]D =
goto TL
```

```
[[ false ( TL, FL) ]]D =
goto FL
```

Short Circuit Evaluation of Conditionals

Code [[BE (TL, FL)]] does not change the stack, but will:

- jump to TL if the value of BE is true
- jump to FL if the value of BE is false

[[~ E2, **TL**, **FL**]]D =

[[E1, FL, TL]]D

Short Circuit Evaluation of Conditionals

Code [[BE (TL, FL)]] does not change the stack, but will:

- jump to TL if the value of BE is true
- jump to FL if the value of BE is false

```
[[ E1 && E2, TL, FL]]D =
[[E1, AuxLabel, FL]]D
AuxLabel:
[[E2, TL, FL]]D
```

```
[[ E1 || E2, TL, FL]]D =
[[E1, TL, AuxLabel]]D
AuxLabel:
[[E2, TL, FL]]D
```

Short Circuit Evaluation of Conditionals

Notice:

- in E1 && E2, code E2 is executed only if E1 yields true
- in E1 || E2, code E2 is executed only if E1 yields false

```
[[ E1 && E2, TL, FL]]D =
[[E1, AuxLabel, FL]]D
AuxLabel:
[[E2, TL, FL]]D
```

```
[[ E1 || E2, TL, FL]]D =
[[E1, TL, AuxLabel]]D
AuxLabel:
[[E2, TL, FL]]D
```

Short Circuit Evaluation of Conditionals

Code [[BE (TL, FL)]] does not change the stack, but will:

- jump to TL if the value of BE is true
- jump to FL if the value of BE is false

```
[[ E1 > E2, TL, FL]]D =
[[E1]]D
[[E2]]D
isub
ifgt TL
goto FL
```

Short Circuit Evaluation of Conditionals

Code [[BE (TL, FL)]] does not change the stack, but will:

- jump to TL if the value of BE is true
- jump to FL if the value of BE is false

```
[[ while E1 do E2 end]]D =
LStart:
[[ E1 (TL, FL) ]]D
TL:
[[ E2 ]]
pop
goto LStart
FL:
```

Short Circuit Evaluation of Conditionals

Code [[BE (TL, FL)]] does not change the stack, but will:

- jump to TL if the value of BE is true
- jump to FL if the value of BE is false

```
[[ if E1 then E2 else E3 end]]D =
[[E1 ( TL, FL) ]]D
TL: [[E2]]
goto LExit
FL: [[E3]]
LExit:
```

Compilation Schemes (reference cells)

Compilation Schemes (reference cells)

At runtime, we will represent a reference cell as a JVM class:

```
.class ref_of_type
.super java/lang/Object
.field public v typeJ
.end method
```

typeJ is the JVM type corresponding to the cell content type, e.g.

int maps to typeJ = I

bool maps to typeJ = Z

ref int maps to typeJ = Lref_of_int;

ref ref bool maps to typeJ = Lref_of_ref_int;

Compilation Schemes (reference cells)

At runtime, we will represent a reference cell as a JVM class:

```
.class ref_of_type
.super java/lang/Object
.field public v typeJ
.end method
```

typeJ will be the JVM type corresponding to the cell content type, e.g.

```
.class ref_of_int
.super java/lang/Object
.field public v I
.end method
```

Compilation Schemes (reference cells)

At runtime, we will represent a reference cell as a JVM class:

```
.class ref_of_type
.super java/lang/Object
.field public v typeJ
.end method
```

typeJ will be the JVM type corresponding to the cell content type, e.g.

```
.class ref_of_bool
.super java/lang/Object
.field public v Z
.end method
```

Compilation Schemes (reference cells)

At runtime, we will represent a reference cell as a JVM class:

```
.class ref_of_type
.super java/lang/Object
.field public v typeJ
.end method
```

typeJ will be the JVM type corresponding to the cell content type, e.g.

```
.class ref_of_ref_of_int
.super java/lang/Object
.field public v Lref_of_int;
.end method
```

Compilation Schemes (reference cells)

At runtime, we will represent a reference cell as a JVM class:

```
.class ref_of_type
.super java/lang/Object
.field public v typeJ
.end method
```

typeJ will be the JVM type corresponding to the cell content type, e.g.

```
.class ref_of_ref_of_ref_of_bool
.super java/lang/Object
.field public v Lref_of_ref_of_bool;
.end method
```

Compilation Schemes (new)

typeJ is the JVM type corresponding to the cell content type, e.g.

```
E[[ new E ]]D =
new ref_of_type
dup
invokespecial ref_of_type/<init>()V
dup
[[E]]D
putfield ref_of_type/v typeJ
```

Compilation Schemes (dereference)

typeJ is the JVM type corresponding to the cell content type, e.g.

```
E[[ ! E ]]D =
[[E]]D
getfield ref_type/v typeJ
```

Compilation Schemes (assign)

typeJ is the JVM type corresponding to the cell content type, e.g.

```
E[[ E1 := E2 ]]D =
[[E1]]D
[[E2]]D
putfield ref_type/v typeJ
```