Interpretation and Compilation of Programming Languages
Part 6 - Imperative languages

João Costa Seco

April 16, 2014

In this lecture we discuss the main features of imperative languages, and how we can define a semantic function in a language with state variables.

We support the fundamental notions of memory and memory operations, which are independent from the operations on the execution environment. We address the important concepts of L-value and R-values and the implicit coercion mechanisms usually used in languages. Important notions like aliasing and memory cell lifetime are also addressed.

We present language constructs usually associated to imperative languages, like loops and sequencing of commands. We discuss how important typing is in the efficient treatment of state variables, and motivate for the existence of stack and heap variables. We build an imperative language with mutable cells, and define its operational and typing semantics.

1 Imperative languages

Programming languages are many times divided into the world of imperative and functional languages (among other language paradigms). The main distintive feature between these worlds is the presence (or predominance) of side effects. Imperative programs operate by iteratively changing the state (memory) of a machine, while functional programs operate by inductively (recursively) defining the results of computations. For instance, the factorial of a natural number can be computed iteratively (e.g. in Javascript), where the result can only be understood by simulating the execution, and analysing the intermediate states.

```
function factorial( n ) {
```

```
  int f = 1;
  int i = 1;
  while( i <= n ) {
    f = f * i;
    i = i + 1;
  }
  return f;
}
```

Or it can be recursively (inductively) defined in OCaml, as follows

```
let rec factorial n = if n = 0 then 1 else n * factorial (n-1)
```

Or in Haskell using a case analysis style,

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n-1)
```

which corresponds more closely to the mathematical notion of the factorial function. Whilst the first example is based on the key notions of state variables and sequences of statements (or commands), the second relies on the key mechanism of recursion. In pure functional languages, a given expression always yields the same result, no matter when, or how many times it is evaluated. The same does not happen in imperative languages, where results are obtained by a succession of state changes, and depends on its initial state.

2 Memory model

In order to define the semantics of imperative languages, it is crucial to define a model of the memory, an abstraction that focuses on the key interactions of a language semantics and memory. The intuitive notion of memory is a (mutable) mapping from memory location identifiers to values. A memory is a dynamic (runtime) data structure maintained by the operational semantics function. This notion is independent and complementary to the notion of runtime environment.

The memory model we define next is an abstraction for the memory layer of a program. It abstracts the notions of memory allocation and deallocation, and management of its contents. The presented abstraction can be seen as the memory management layer of a virtual machine or operating system. It can, for instance, isolate memory management, monitoring, and garbage collection mechanisms.

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

Figure 1: Java interface for a memory data type.

Consider that a memory abstraction maintains a (potentially) infinite number of mutable memory cells. The main operations are memory allocation, setting/getting values to/from memory locations, and deallocation. One key notion, necessary in this model, are memory location identifiers, also known as memory references or pointers. Programming languages support a variety of memory manipulation mechanisms, from `Java` references, `OCaml` variables, to `C` and `C++` low-level pointers, where, besides the basic operations, arithmetic operations are also allowed.

Memory references are language (runtime) values whose meanings are given by concrete memories. A memory can therefore be defined by an abstract data type, with name *Mem* and the following constructs:

$$\begin{aligned} \texttt{new}: &\quad \mathit{Mem} \times \mathit{Value} \to \mathit{Ref} \times \mathit{Mem} \\ \texttt{free}: &\quad \mathit{Mem} \times \mathit{Ref} \to \mathit{Mem} \\ \texttt{set}: &\quad \mathit{Mem} \times \mathit{Ref} \times \mathit{Value} \to \mathit{Mem} \\ \texttt{get}: &\quad \mathit{Mem} \times \mathit{Ref} \to \mathit{Value} \end{aligned}$$

Which corresponds to the Java interface of Figure 1, that can be instantiated in an imperative way by an object that manages a memory. A possible instantiation of such interface is a singleton class that manages a global memory. In the definition of Figure 1 we consider the inductively defined type `Value`, and a subtype `RefValue` that denotes memory locations (see Figure 3). Consider the simple implementation of interface `Memory`, defined in Figure 2, where memory locations in the operational semantics are directly represented by values of class `RefValue`.

The operational semantics of an imperative language is based on a combination of an evaluation environment and a memory. The evaluation environment keeps the bindings of identifiers to their denotations, which is fixed during their life time. The memory maintains the contents of memory cells.

```
class MemoryImpl implements Memory {

  static Memory mem = new Memory();

  private Memory() {}

  RefValue new(Value v) { return new RefValue(v); }
  void free(RefValue r) { /*No need in a GC language */ }
  void set(RefValue r, Value v) { r.set(v); }
  Value get(RefValue r) { return r.get(); }
}
```

Figure 2: Sketch of a Java class implementing a memory.

```
class RefValue implements Value {
  Value content;

  RefValue(Value content) {...}
}
```

Figure 3: Java reference values.

Denotations kept in the environment may include memory locations. Figure 5 depicts a sample environment and memory in the marked program spot of Figure 4.

Recall that the direct manipulation of memory references is a crucial mechanism to allow the construction of dynamically allocated data structures. It also opens the door to the essential scenarios where there are more than one path from a local (stack) variable to a memory location (e.g. transverse a linked list, define a graph, etc.). This effect is called *aliasing*, and it can also be the source of programming errors. Check the function `revert` in Figure 6, that copies the contents of an array to another, in the reverse index order. If this function is called using a single array as actual parameters (where `a` and `b` refer to the same locations), then the final contents of the array (`a`) will not be as expected.

Memory allocation in imperative languages is found in many different places, and can be categorised according to the kind of memory handling mechanisms, the life time of memory cells, and the way cells are managed. The declaration of a local variable, of `int` type, in `C`, and the declaration of an imperative variables in `OCaml`, are completely different. The allocation

```
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);
}
```

Figure 4: Aliasing example.

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

Figure 5: Environment and memory snapshot of the program in Figure 4

of an object in `Java` or in `C++` are also different. In `C++`, an object can be allocated in the stack or explicitly in the heap. Also, allocation of arrays is different in `Java` or in `C`.

Other differences can observed in the way memory cells are accessed. For instance, `C` expression `x=x+1` contains two different occurrences of `x`, with two distinct denotations. The denotation in the left-hand side of the assignment denotes the memory location being modified, and the denotation on the right-hand side is the content of the memory location associated to identifier `x`. There is an implicit coercion between the memory location and its contents, that depends on the evaluation context. Languages like (`C`, `Java`, `C#`, etc., are examples of this category. We name values that denote a memory cell, as *L-value*s, and *R-value*s to the remaining values. The implicit conversion, from *L-value* to *R-value*, is performed according to the evaluation context. An equivalent expression in `OCaml`, that treats reference accesses (variables) explicitly, is the following: `x:=!x+1`. Expression `!x` denotes the content of the

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

Figure 6: Função `revert`.

reference denoted by `x`. In section **??** of an appendix to this document you may find the semantics of a language with implicit dereference of variables. In general, an *L-value* is converted to its *R-value* when its use context demands for its contents and not the memory location itself. However, consider the expression `a[a[2]]:=a[1]`, where expression `a[2]` and `a[1]` are converted to an *R-value*. In order to statically determine the coercion points, there is a need for a prior type analysis.

A simpler alternative to implicit dereference of variables, can be defined by forcing the explicit allocation and deallocation of variables.

An example, is the `OCaml` language, or the use of heap memory in `C` (`malloc` and `free`). The expression above would be written as follows: `a[!a[2]] := !a[1]`.

3 Language *CALCState*

We now present an imperative programming language called *CALCState* extending the language *CALCI* with commands and operations directly connected memory manipulation and the typical loop structures of imperative programming.

$\texttt{num} : \mathit{Integer} \to \mathit{CALCState}$
$\texttt{add} : \mathit{CALCState} \times \mathit{CALCState} \to \mathit{CALCState}$
$\texttt{sub} : \mathit{CALCState} \times \mathit{CALCState} \to \mathit{CALCState}$
$\texttt{mul} : \mathit{CALCState} \times \mathit{CALCState} \to \mathit{CALCState}$
$\texttt{div} : \mathit{CALCState} \times \mathit{CALCState} \to \mathit{CALCState}$
$\texttt{id} : \mathit{String} \to \mathit{CALCState}$
$\texttt{decl} : \mathit{String} \times \mathit{CALCState} \times \mathit{CALCState} \to \mathit{CALCState}$
$\texttt{var} : \mathit{CALCState} \to \mathit{CALCState}$
$\texttt{assign} : \mathit{CALCState} \times \mathit{CALCState} \to \mathit{CALCState}$
$\texttt{deref} : \mathit{CALCState} \to \mathit{CALCState}$
$\texttt{free} : \mathit{CALCState} \to \mathit{CALCState}$
$\texttt{seq} : \mathit{CALCState} \times \mathit{CALCState} \to \mathit{CALCState}$
$\texttt{while} : \mathit{CALCState} \times \mathit{CALCState} \to \mathit{CALCState}$
$\texttt{If} : \mathit{CALCState} \times \mathit{CALCState} \times \mathit{CALCState} \to \mathit{CALCState}$

The semantics of this language is based on a memory model, whose operations where already explored (see section **??**). Notice that in the syntax of this language, we have only one syntactical category (expressions). This is usually not the case in main-stream imperative languages, where (ideally) the effects on the memory are accomplished by commands (or statements), and where expressions are "pure" (do not cause effects on the state). Languages that separate commands from expressions usually belong to the family of Algol-derived languages. To keep the language more uniform and simpler we keep only one category, and define a language belonging to the so-called ML derived languages (only with expressions).

We introduce an expression to allocate a new state variable (a memory cell containing a language value) `var`, an expression to set a new value to an existing state variable, `assign`, an expression to retrieve the value of a state variable, `deref`, and an expression that releases the occupied memory, `free`. These are the expressions that actually produce an effect in the state of the memory. Notice that we abstract the occupied memory by each value, and assume an infinite number of available memory cells.

Given that our language has effects, it is important that its semantics specifies the exact order of evaluation of each sub-component of each language expression. In terms of general specification we can add an extra parameter to our semantic function to denote the current memory state, and give meaning to the denotations that may be stored in the environment. On the other hand, since results may now include memory locations, these results only

have meaning with relation to a memory. So, we define the signature of the evaluation function as follows:

$eval : CALCState \times ENV \times MEM \rightarrow Result \times Mem$

Where the Result set is now extended to include references.

$Result \triangleq Integer \cup Boolean \cup Ref$

Notice that all interpreters (including native code) make use of a memory abstraction, that is in charge of memory management mechanisms, like allocation, protection, and even garbage collection.

Consider the code in Figure 7 where the type loc is abstracted (defined in the memory module for instance). Notice that the order of execution is deterministically defined by the data dependency between the results and the parameter representing the memory.

In the case of expression `Add (e,e')` the evaluation of expression `e` is performed with relation to the initial memory (`mem`) and its effects are registered in the resulting memory (`mem'`). The sub-expression `e'` is evaluated with relation to that memory (`mem'`) modifying the memory that is yielded as result of the addition expression (`mem''`).

Notice that an implementation of an interpreter of language *CalcState*, using an imperative language with a suitable memory management mechanism, like Java or OCaml, it is enough (and easier) to use the native mechanisms (see Figures 8 and 9).

References can be managed (and implemented) in several different ways. If one uses an imperative language as the base for the interpreter, then the chaining of state changes is implicitly placed in the host language (Java). `RefValue` values (Figure 9) represented by objects act as memory cells.

4 Typing State Variables

In order to trap execution errors related to memory manipulation, in a simple an efficient way, we must adopt restrictions that are common in typed imperative languages. The most important is that a state variable only contains values of one single type. This leads to a set of types defined by the abstract data type in Figure 10,

Exercise 1. Define the type semantics of the given language.

```
type loc = ...

type value =
   Num of int
 | Bool of bool
 | 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
      (Num 0,mem'')
```

Figure 7: Semantics using explicit memory management.

```
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 -> Num 0 (* Não é necessário *)
```

Figure 8: Semantics using references in OCaml.

```
interface Value {}

class IntValue implements Value {
  public final int n;

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

class RefValue implements Value {
  Value v;

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

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

  Value eval(Env<Value> env) {
    return new RefValue(exp.eval(env));
  }
}
```

Figure 9: Semantics using references in Java.

```
type ty =
    IntType
  | BoolType
  | RefType of ty
```

Figure 10: Types for imperative languages