Languages and Compilation

Based on the Jean-Christophe Filliâtre's Courses
given at École Polytechnique & École Normale Supérieure

Lecture 8 - evaluation strategies and parameter passing

Léon Gondelman

AALBORG UNIVERSITY | COPENHAGEN | 2025

- **concepts**: evaluations strategies, parameter passing
- **illustration**: parameter passing modes of
  - Java
  - OCaml
  - Python
  - C
  - C++
- **lab session:**
  - continuing the previous lab (compiling a small language to MIPS)
  - help with the projects

evaluation strategies, parameter passing

when **declaring** a function

```
function f(x1, ..., xn) =
  ...
```

variables `x1,...,xn` are called the **formal parameters** of `f`

and when **calling** this function

```
f(e1, ..., en)
```

expressions `e1,...,en` are called the **actual parameters** of `f` in a language with in-place modifications, an assignment

```
e1 := e2
```

modifies a memory location designated by `e1`

the expression `e1` is limited to certain constructs,
and assignments such as

```
42 := 17
true := false
```

do not make sense

an expression that is legal on the left-hand side of an assignment is called a **left value** the evaluation strategy of a language defines the order in which computations are performed

this can be defined using a formal semantics (see lecture 2)

the compiler must obey the evaluation strategy in particular, the evaluation strategy **may** specify

- **when** actual parameters are evaluated
- the evaluation **order** of operands and actual parameters

some aspects of evaluation may be **left unspecified**

this allows the compiler to perform more aggressive optimizations
(such as reordering computations)

we distinguish

- **eager evaluation**: operands / actual parameters are evaluated before the operation / the call

  examples: C, C++, Java, OCaml, Python

- **lazy evaluation**: operands / actual parameters are evaluated only when needed

  examples: Haskell, Clojure
  but also Boolean operators `&&` and `||` in most languages

an imperative language has to adopt an eager evaluation, to ensure that side effects are performed consistently with the source code

for instance, the Java code

```
int r = 0;
int id(int x) { r += x; return x; }
int f(int x, int y) { return r; }

{ System.out.println(f(id(40), id(2))); }
```

prints 42 since both arguments of `f` are evaluated an exception is made for Boolean operations && and || in most languages, which is really useful

```
void insertionSort(int[] a) {
  for (int i = 1; i < a.length; i++) {
    int v = a[i], j = i;
    for (; 0 < j && v < a[j-1]; j--)
      a[j] = a[j-1];
    a[j] = v;
  }
}
```

non-termination is also a side effect

for instance, the Java code

```
int loop() { while (true); return 0; }
int f(int x, int y) { return x+1; }

{ System.out.println(f(41, loop())); }
```

does not terminate, even if argument y is not used a **purely functional** language (= without imperative features) may adopt any evaluation strategy, since an expression will always evaluate to the same value (this is called **referential transparency**)

in particular, it may adopt a lazy evaluation the Haskell program

```
loop () = loop ()
f x y = x
main = putChar (f 'a' (loop ()))
```

terminates (and prints a)

the semantics also defines the way parameters are passed in a function call

several approaches:

- **call by value**
- **call by reference**
- **call by name**
- **call by need**

(we also say **passing by value**, etc.)

new variables receive the **values** of actual parameters

```
function f(x) =
  x := x + 1

main() =
  int v := 41
  f(v)
  print(v)    // prints 41
```

formal parameters denote the **same left values** as actual parameters

```
function f(x) =
  x := x + 1

main() =
  int v := 41
  f(v)
  print(v)    // prints 42
```

actual parameters are **substituted** to formal parameters, textually, and thus are evaluated only if necessary

```
function f(x, y, z) =
  return x*x + y*y

main() =
  print(f(1+2, 2+2, 1/0)) // prints 25
  // 1+2 is evaluated twice
  // 2+2 is evaluated twice
  // 1/0 is never evaluated
```

actual parameters are evaluated only if necessary,
and **at most once**

```
function f(x, y, z) =
  return x*x + y*y

main() =
  print(f(1+2, 2+2, 1/0)) // prints 25
  // 1+2 is evaluated once
  // 2+2 is evaluated once
  // 1/0 is never evaluated
```

a few words on Java

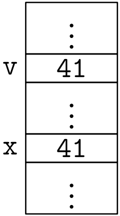
Java uses an eager evaluation, with **call by value**

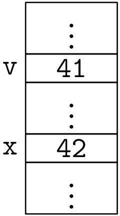
evaluation order is left-to-right

a value is

- either of a primitive type (Boolean, character, machine integer, etc.)
- or a pointer to a heap-allocated object

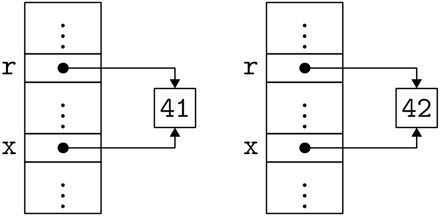
```c
void f(int x) {
  x = x + 1;
}

int main() {
  int v = 41;
  f(v);
  // v is still 41
}
```





an object is allocated on the heap

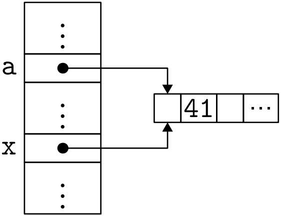
```
class C { int f; }

void incr(C x) {
  x.f += 1;
}
void main () {
  C r = new C();
  r.f = 41;
  incr(r);
  // r.f now is 42
}
```

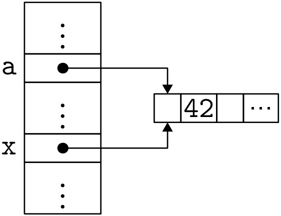


this is still **call by value**,
with a value that is an (implicit) pointer to an object

an array is an object

```
void incr(int[] x) {
  x[1] += 1;
}
void main () {
  int[] a = new int[17];
  a[1] = 41;
  incr(a);
  // a[1] now is 42
}
```





we can **emulate call by name** in Java, by replacing parameters with functions; for instance, the function

```
int f(int x, int y) {
  if (x < 0 || x == 0) return 42; else return y + y;
}
```

can be turned into

```
int f(Supplier<Integer> x, Supplier<Integer> y) {
  if (x.get() < 0 || x.get() == 0)
    return 42;
  else
    return y.get() + y.get();
}
```

and called like this

```
int v = f(() -> 0, () -> { throw new Error(); });
```

more efficiently, we can **simulate call by need** in Java

```
class Lazy<T> implements Supplier<T> {
  private T cache = null;
  private Supplier<T> f;

  Lazy(Supplier<T> f) { this.f = f; }

  public T get() {
    if (this.cache == null) {
      this.cache = this.f.get();
      this.f = null; // allows the GC to reclaim f
    }
    return this.cache;
  }
}
```

(this is memoization)

and we use it like this

```
int w = f(new Lazy<Integer>(() -> 1),
          new Lazy<Integer>(() -> { ...takes time... }));
```

a few words on OCaml

OCaml has an eager evaluation, with **call by value**

evaluation order is left unspecified

a value is

- either of a primitive type (Boolean, character, machine integer, etc.)
- or a pointer to a heap-allocated block (array, record, non constant constructor, etc.)

left values are array elements

```
a.(2) <- true
```

and mutable record fields

```
x.age <- 42
```

OCaml's "mutable variables" (aka references) are records

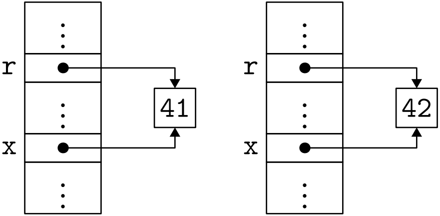
```
type 'a ref = { mutable contents: 'a }
```

and operations := and ! are defined as

```
let (!)  r   = r.contents
let (:=) r v = r.contents <- v
```

a reference is allocated on the heap

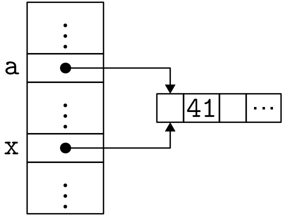
```
let incr x =
  x := !x + 1

let main () =
  let r = ref 41 in
  incr r
  (* !r now is 42 *)
```

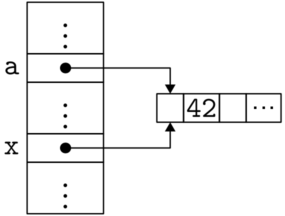


this is still **call by value**,
with a value that is an (implicit) pointer to a mutable data

an array is allocated on the heap

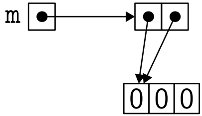
```
let incr x =
  x.(1) <- x.(1) + 1

let main () =
  let a = Array.make 17 0 in
  a.(1) <- 41;
  incr a
  (* a.(1) now is 42 *)
```





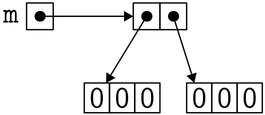
to build a matrix, do not write

```
let m = Array.make 2 (Array.make 3 0)
```



but

```
let m = Array.make_matrix 2 3 0
```



we can **simulate call by name** in OCaml, by replacing parameters with functions

for instance, the function

```
let f x y =
  if x = 0 then 42 else y + y
```

can be turned into

```
let f x y =
  if x () = 0 then 42 else y () + y ()
```

and called like this

```
let v = f (fun () -> 0) (fun () -> failwith "oups")
```

we can also **simulate call by need** in OCaml

we first introduce a type to represent lazy computations

```
type 'a value = Value  of 'a
              | Frozen of (unit -> 'a)

type 'a by_need = 'a value ref
```

and a function to evaluate a computation when it is not yet done

```
let force l = match !l with
  | Value  v -> v
  | Frozen f -> let v = f () in l := Value v; v
```

(this is memoization)

then we define function `f` as follows

```
let f x y =
  if force x = 0 then 42 else force y + force y
```

and we call it with

```
let v = f (ref (Frozen (fun () -> 1)))
          (ref (Frozen (fun () -> ...takes time...)))
```

note: OCaml has a `lazy` construct that does something similar (but in a more subtle and more efficient way)

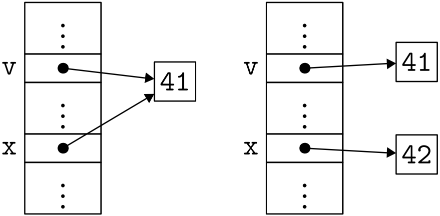
a few words on Python

Python has an eager evaluation, with **call by value**

evaluation order is left-to-right
(but right-to-left for an assignment)

a value is a pointer to a heap-allocated object an integer is an **immutable** object

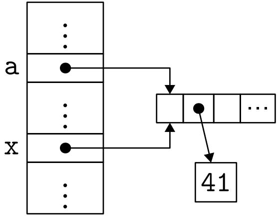
```
def f(x):
    x += 1

v = 41
f(v)
print(v) # prints 41
```

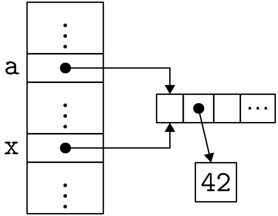


this is still **call by value**,
with a value that is an (implicit) pointer to an object

an array is a mutable object

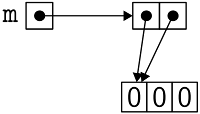
```
def incr(x):
    x[1] += 1

a = [0] * 17
a[1] = 41
incr(a)
# a[1] now is 42
```





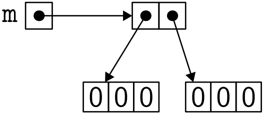
to build a matrix, do not write

```
m = [[0] * 3] * 2
```



but

```
m = [[0] * 3 for _ in range(2)]
```



execution models of Java, OCaml, and Python are **very close**

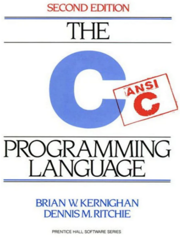
even if their surface languages are way different

a few words on C

C is an imperative language that is considered low-level, notably because pointers and pointer arithmetic are explicit

conversely, C can be considered as a high-level assembly language

a book that is still relevant:
The C Programming Language
by Brian Kernighan and Dennis Ritchie



the C language has an eager evaluation, with **call by value**

evaluation order is left unspecified

- we have primitive types such as `char`, `int`, `float`, etc.

- a type $\tau*$ for pointers to values of type τ

  if p is a pointer of $\tau*$, then $*p$ stands for the value pointed to by p, of type τ

  if e is a left value of type τ, then $\&e$ is a pointer to its memory location, with type $\tau*$

- we have records, called *structures*, such as

```
struct L { int head; struct L *next; };
```

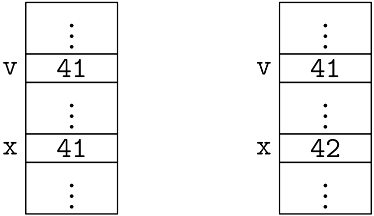
if `e` has type `struct L`, we write `e.head` for a field access in C, a left value is either

- `x`, a variable
- `*e`, the dereferencing of a pointer
- `e.x`, a structure field access
  if `e` is itself a left value

- `t[e]`, that is sugar for `*(t+e)`
- `e->x`, that is sugar for `(*e).x`

call by value

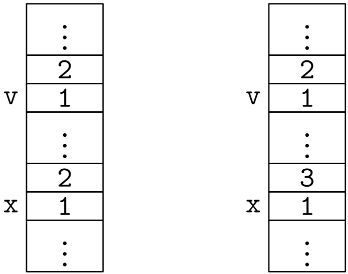
```
void f(int x) {
  x = x + 1;
}

int main() {
  int v = 41;
  f(v);
  // v is still 41
}
```



call by value means that **structures are copied** when passed to functions or returned

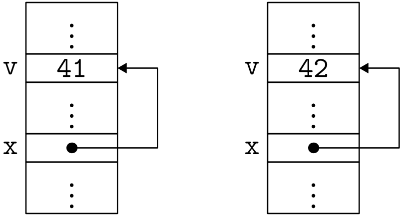
structures are also copied when variables of structure types are assigned, *i.e.* assignments such as `x = y`, where `x` and `y` have type `struct S`

```
struct S { int a; int b; };

void f(struct S x) {
  x.b = x.b + 1;
}

int main() {
  struct S v = { 1, 2 };
  f(v);
  // v.b is still 2
}
```



we can **simulate** a call by reference by passing an explicit pointer

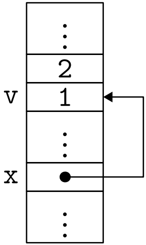
```
void incr(int *x) {
  *x = *x + 1;
}

int main() {
  int v = 41;
  incr(&v);
  // v now is 42
}
```

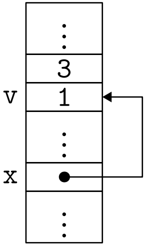


but this is still **call by value**

to avoid copies, we often use pointers to structures

```
struct S { int a; int b; };

void f(struct S *x) {
  x->b = x->b + 1;
}

int main() {
  struct S v = { 1, 2 };
  f(&v);
  // v.b now is 3
}
```





explicit pointer manipulation can be dangerous

```
int* p() {
  int x;
  ...
  return &x;
}
```

this function returns a pointer to a memory location on the stack (the stack frame of `p`) that is not meaningful anymore, and that is going to be reused for another stack frame

we call this a **dangling reference**

notation `t[i]` is syntactic sugar for `*(t+i)` where

- `t` is a pointer to a memory location containing consecutive integers
- `+` stands for **pointer arithmetic** (adding `4i` to `t` for an array of 32 bit integers)

the first element of the array is thus `t[0]`, that is `*t` an array may be allocated on the stack, as follows

```
void f() {
  int t[10];
```

and it will be deallocated when the function exits

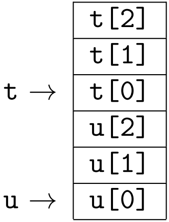
or allocated on the heap, as follows

```
int *t = malloc(10 * sizeof(int));
```

and it has to be deallocated with `free` we cannot assign arrays, only pointers

so we can't write

```
void p() {
  int t[3];
  int u[3];
  t = u;    // <- error
}
```



since `t` and `u` are (stack-allocated) arrays and arrays assignment is not possible when passing an array, we only pass a pointer (by value, as always)

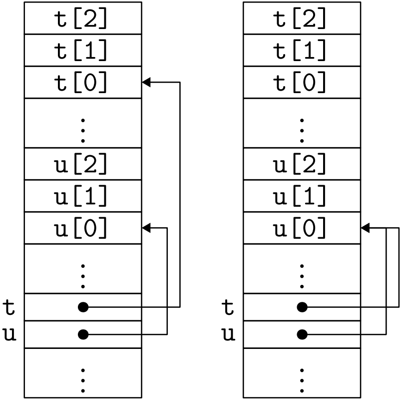
we can write

```
void q(int t[3], int u[3]) {
  t = u;
}
```

and this is exactly the same as

```
void q(int *t, int *u) {
  t = u;
}
```

and pointer assignment is possible



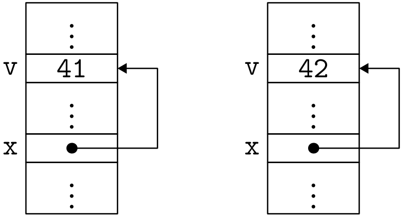
a few words on C++

in C++, we have (among other things) all the types and constructs of C with an eager evaluation

passing is **call by value** by default

but we also have **call by reference**
indicated with symbol & at the formal parameter site

```
void f(int &x) {
  x = x + 1;
}

int main() {
  int v = 41;
  f(v);
  // v now is 42
}
```



this is the compiler that

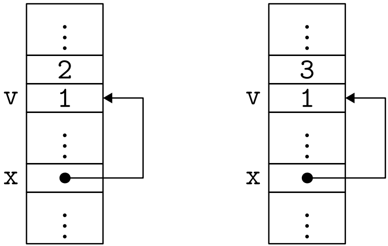
- passed a pointer to `v` at the call site
- dereferenced the pointer `x` in function `f`

the actual parameter has to be a left value

```
void f(int &x) {
  x = x + 1;
}

int main() {
  f(41); // <- error (not a left value)
}
```

we can pass structures by reference

```
struct S { int a; int b; };

void f(struct S &x) {
  x.b = x.b + 1;
}

int main() {
  struct S v = { 1, 2 };
  f(v);
  // v.b now is 3
}
```

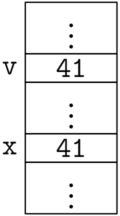
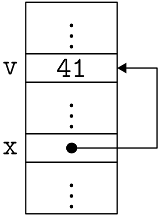
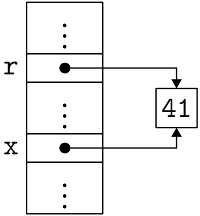


we can pass pointers by reference

for instance to insert an element into a mutable tree

```
struct Node { int elt; Node *left, *right; };

void add(Node* &t, int x) {
  if      (t == NULL ) t = create(NULL, x, NULL);
  else if (x < t->elt) add(t->left,  x);
  else if (x > t->elt) add(t->right, x);
}
```

summary

			
Java	integer by value	—	pointer by value (object)
OCaml	integer by value	—	pointer by value (`ref`, array, etc.)
Python	—	—	pointer by value (object)
C	integer by value	pointer by value	pointer by value
C++	integer by value	pointer by value integer by reference	pointer by value or by reference

- lab 8
  - compiling a small language to MIPS
  - help with the projects
- next lecture
  - OO languages compilation