École Polytechnique

INF564 – Compilation

Jean-Christophe Filliâtre

evaluation strategies
compilation of object-oriented languages
evaluation strategies
some terminology

when *declaring* a function

\[
\text{function } f(x_1, \ldots, x_n) = \\
\ldots
\]

variables \(x_1, \ldots, x_n\) are called the *formal parameters* of \(f\)

and when *calling* this function

\[
f(e_1, \ldots, e_n)
\]

expressions \(e_1, \ldots, e_n\) are called the *actual parameters* of \(f\)
some terminology

in a language with in-place modifications, an assignment

\[ e_1 := e_2 \]

modifies a memory location designated by \( e_1 \)

the expression \( e_1 \) is limited to certain constructs,
for assignments such as

\[
\begin{align*}
42 & := 17 \\
true & := false
\end{align*}
\]

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

dthis 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

```java
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

```java
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

```java
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
purely functional programming

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 can adopt a lazy evaluation
the Haskell program

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

terminates (and prints a)
parameter passing

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

```plaintext
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

```plaintext
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

```plaintext
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**

```plaintext
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
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
}

diagram showing allocation on the heap

diagram showing allocation on the heap

diagram showing allocation on the heap

diagram showing allocation on the heap

diagram showing allocation on the heap

diagram showing allocation on the heap

diagram showing allocation on the heap

diagram showing allocation on the heap

diagram showing allocation on the heap

diagram showing allocation on the heap

diagram showing allocation on the heap

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

```java
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

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

can be turned into

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

and called like this

```java
int v = f(() -> 0, () -> { throw new Error(); });
```
more efficiently, we can **simulate call by need** in Java

```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

```java
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

\[
a.(2) \leftarrow \text{true}
\]

and mutable record fields

\[
x.age \leftarrow 42
\]
reminder: an OCaml reference is a record

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

and operations := and ! are defined as

```ocaml
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

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

but

```ocaml
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

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

can be turned into

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

and called like this

```ocaml
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

```ocaml
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

```ocaml
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

\[
\text{let } f \ x \ y = \\
\quad \text{if force } x = 0 \text{ then } 42 \text{ else force } y + \text{force } y
\]

and we call it with

\[
\text{let } v = f \ (\text{ref (Frozen (fun () \to 1)))} \\
\quad (\text{ref (Frozen (fun () \to \ldots \text{takes time\ldots})}))
\]

note: OCaml has a \textbf{lazy} construct that does something similar (but in a more subtle 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

```python
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

```python
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

\[
m = \left[ \left[ 0 \right] \times 3 \right] \times 2
\]

but

\[
m = \left[ \left[ 0 \right] \times 3 \right] \text{ for } _\_ \text{ in range}(2)]
\]
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 $\tau$

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

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

• we have records, called structures, such as

```c
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 \( (*e).x \)
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 \texttt{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

```c
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

```c
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

```c
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

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

and it will be deallocated when the function exits

or allocated on the heap, as follows

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

and it has to be deallocated with `free` (see lecture 9)
we cannot assign arrays, only pointers

so we can’t write

```c
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

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

and this is exactly the same as

```c
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 \textbf{call by value} by default

but we also have \textbf{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
f’s actual parameter has to be a left value

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

int main() {
    f(41); // <- error (not a left value)
}
```
we can pass structures by reference

```c
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

```c
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);
}
```
<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>OCaml</th>
<th>Python</th>
<th>C</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>integer by value</td>
<td>integer by value</td>
<td>—</td>
<td>integer by value</td>
<td>integer by value</td>
</tr>
<tr>
<td>ref</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>pointer by value</td>
<td>pointer by value</td>
</tr>
<tr>
<td>(object)</td>
<td>pointer by value</td>
<td>pointer by value</td>
<td>pointer by value</td>
<td>pointer by value</td>
<td>pointer by value</td>
</tr>
<tr>
<td>(ref, array, etc.)</td>
<td>pointer by value</td>
<td>pointer by value</td>
<td>pointer by value</td>
<td>pointer by value</td>
<td>pointer by value</td>
</tr>
</tbody>
</table>

Jean-Christophe Filliâtre
compiling OO languages
let us explain

- how an object is represented
- how a method call is implemented

let us use Java as an example (for the moment)
representing objects

an object is an heap-allocated block, containing

• its class
• the values of its fields

the value of an object is a pointer to the block

simple inheritance allows us to store the value of some field $x$ at some fixed position in the block: own fields are placed after inherited fields
class Vehicle {
    static int start = 10;
    int position;
    Vehicle() { position = start; }
    void move(int d) { position += d; }
}

class Car extends Vehicle {
    int passengers;
    void await(Vehicle v) {
        if (v.position < position)
            v.move(position - v.position);
        else
            move(10);
    }
}

class Truck extends Vehicle {
    int load;
    void move(int d) {
        if (d <= 55) position += d; else position += 55;
    }
}
**overriding** is the ability to redefine a method in a subclass (so that objects in that subclass behave differently)

example: in class `Truck`

```java
class Truck extends Vehicle {
    ...
    void move(int d) { ... }
}
```

the method `move`, inherited from class `Vehicle`, is **overridden**
representing objects

each object is a heap-allocated block, as follows

<table>
<thead>
<tr>
<th>Vehicle</th>
<th>Car</th>
<th>Truck</th>
</tr>
</thead>
<tbody>
<tr>
<td>position</td>
<td>position</td>
<td>position</td>
</tr>
<tr>
<td>passengers</td>
<td></td>
<td>load</td>
</tr>
</tbody>
</table>

note the absence of field `start`, which is static and thus allocated elsewhere (for instance in the data segment)
field access

for each field, the compiler knows its position, that is the offset to add to the object pointer

if for instance field position is at offset +16, then expression e.position is compiled to

```c
...  # we compile e in %rcx
movl 16(%rcx), %eax  # field position at +16
```

this is sound, even if the compiler only knows the static type of e, which may differ from the dynamic type (the class of the object)

it could even be a sub-class of Vehicule that is not yet defined!
the essence of OO languages lies in **dynamic method call** \( e.m(e_1, \ldots, e_n) \) (aka dynamic dispatch / message passing)

to do this, we build **class descriptors** containing addresses to method codes (aka **dispatch table**)

as for class fields, simple inheritance allows us to store the address of (code of) method \( m \) at a fixed offset in this descriptor

class descriptors can be allocated in the data segment; each object points to its class descriptor
class Vehicule
   { void move(int d) {...} }

class Car extends Vehicule
   { void await(Vehicule v) {...} }

class Truck extends Vehicule
   { void move(int d) {...} }

descr. Vehicule
   Vehicule_move

descr. Car
   Vehicule_move
   Car-await

descr. Truck
   Truck_move
to compile a call such as e.move(10)

1. we compile e; its value is a pointer to an object
2. this object contains a pointer to its class descriptor
3. inside, the code for method move is located at an offset (for instance +8) known from the compiler

```asm
... # compiler e into %rdi
movq $10, %rsi # parameter
movq (%rdi), %rcx # class descriptor
call *8(%rcx) # method move
```

as for field access, the compiler has no need to know the actual class of the object (the dynamic type)
if we write

```java
Truck v = new Truck();
((Vehicule)v).move();
```

this is the method `move` from class `Truck` that is called since the call is always compiled the same way

the cast only has an influence on the static type (existence of the method + overloading resolution)
In practice, the class descriptor for $C$ also contains points to the class that $C$ inherits from, called the **super class** of $C$.

This can be a pointer to the descriptor of the super class (for instance stored in the very first slot of the descriptor).

This allows subtyping tests at runtime (*downcast* or *instanceof*).
a few words on C++
let us reuse the vehicles example

```cpp
class Vehicle {
    static const int start = 10;

public:
    int position;
    Vehicle() { position = start; }
    virtual void move(int d) { position += d; }
};
```

`virtual` means that method `move` can be overridden
```cpp
class Car : public Vehicle {
public:
    int passengers;
    Car() {}
    void await(Vehicle &v) { // call by reference
        if (v.position < position)
            v.move(position - v.position);
        else
            move(10);
    }
};
```
class Truck : public Vehicle {
public:
    int load;
    Truck() {}
    void move(int d) {
        if (d <= 55) position += d; else position += 55;
    }
};
#include <iostream>
using namespace std;

int main() {
    Truck t; // objects are stack-allocated
    Car c;
    c.passengers = 2;
    cout << c.position << endl; // 10
    c.move(60);
    cout << c.position << endl; // 70
    Vehicle *v = &c; // alias
    v->move(70);
    cout << c.position << endl; // 140
    c.await(t);
    cout << t.position << endl; // 65
    cout << c.position << endl; // 140
}
on this example, object representation is not different from Java’s

<table>
<thead>
<tr>
<th>descr. Vehicle</th>
<th>descr. Car</th>
<th>descr. Truck</th>
</tr>
</thead>
<tbody>
<tr>
<td>position</td>
<td>position</td>
<td>position</td>
</tr>
<tr>
<td>passengers</td>
<td></td>
<td>load</td>
</tr>
</tbody>
</table>
but in C++, we also **multiple inheritance**

consequence: we cannot use anymore the principle that

- the object layout for the super class is a prefix of the object layout of the subclass
- the descriptor for the super class is a prefix of the descriptor for the subclass
class Rocket {
public:
    float thrust;
    Rocket() {}
    virtual void display() {}
};

class RocketCar : public Car, public Rocket {
public:
    char *name;
    void move(int d) { position += 2*d; }
};

representations of Car and Rocket are appended
multiple inheritance

in particular, a cast such as

```
RocketCar rc;
... (Rocket)rc ...
```

is compiled using pointer arithmetic

```
... rc + 12 ...
```

this is not a no-op anymore
let us now assume that Rocket also inherits from Vehicle

class Rocket : public Vehicle {
public:
    float thrust;
    Rocket() { } 
    virtual void display() {} 
};

class RocketCar : public Car, public Rocket {
public:
    char *name;
    ...
};

we now have two fields position
and thus a possible ambiguity

class RocketCar : public Car, public Rocket {
public:
    char *name;
    void move(int d) { position += 2*d; }
};

vehicles.cc: In member function ‘virtual void RocketCar::move(int)’
vehicles.cc:51:22: error: reference to ‘position’ is ambiguous
we have to say which one we want

class RocketCar : public Car, public Rocket {
public:
    char *name;
    void move(int d) { Rocket::position += 2*d; }
};
to have a single instance of Vehicle, we need to modify the way Car and Rocket inherit from Vehicle; this is **virtual inheritance**

```cpp
class Vehicle { ... };

class Car : public virtual Vehicle { ... };

class Rocket : public virtual Vehicle { ... };

class RocketCar : public Car, public Rocket {

there is no ambiguity anymore:

```cpp
public:
    char *name;
    void move(int d) { position += 2*d; }
};
```
three class diagrams

class Vehicle { ... };
class Car : Vehicle { ... };
class Rocket { ... };
class RocketCar : Car, Rocket { ... };
though Java only features simple inheritance, interfaces make method call more complex, in a way analogous to multiple inheritance

```java
interface I {
    void m();
}

class C {
    void foo(I x) { x.m(); }
}
```

when compiling `x.m()`, we have no idea what the class of object `x` will be
• lab 5
  • static typing of mini-C continued

• next lecture
  • optimizing compiler (1/3)