Lecture #3 on Object-Oriented Modeling

Thierry Géraud

EPITA Research and Development Laboratory (LRDE)

2006

Outline

1. Sub-typing
   - Extension
   - Modification
   - Class v. Interface

2. Imperative so Be Careful
   - A Simple Example
   - Consequences

Outline

Outline

Simple Test

with:

```
type t = { i : int, f : int -> float }
foo : (arg : t) -> float =
  arg.f(arg.i)
  end
```

the following program is clearly correct!

```
type t' = { i : int, f : int -> float, b : bool }
var : t' // = initialization
res : float = foo(var)
```

Extending is Sub-typing (1/2)

when

$$t \; = \; \{ \, \tau_1, \, .., \, \tau_n \, \}$$

and (with extra fields)

$$t' \; = \; \{ \, \tau_1, \, .., \, \tau_n, \, \tau_{n+1}, \, .., \, \tau_m \, \}$$

we have

$$t' <: t$$

put differently:
when t is expected, giving more than t is ok

Extending is Sub-typing(2/2)

sub-classing can be an extension process

with `base` being a class, you can write:

```
class derived : public base
{
  // here:
  // - the contents of 'base' is not "modified"
  // - extra attributes and methods are added
};
```

so

- `base` is extended
- the result of this extension is defined as the `derived` class

Extension and substitution

when “sub-classing” means “only extending”,
whenever a base class is used,
we can substitute it with any sub-class

adding extra (different and new) features through inheritance
is totally type-safe

An example that rocks

```
#include <iostream>

struct shape {
  float x, y;
  void translate(float dx, float dy) { x += dx; y += dy; }
};

struct circle {
  float x, y;
  float r;
};

int main() {
  circle* c = new circle;
  c->x = 4;  c->y = 0;
  shape* s = (shape*)c;
  s->translate(1, 1);
  std::cout << c->x << ' ' << c->y << std::endl;
}
```

is really type-safe!

What about Modifying?

If we do not extend

$$t \; = \; \{ \, \tau_1, \; .., \; \tau_n \, \}$$

we can try to modify it such as in

$$t' \; = \; \{ \, \tau'_1, \; .., \; \tau'_n \, \}$$

same question: can we have $t' <: t$?

Outline

What about Modifying?

when

$$t \ = \ \{\, \tau_1,\ ..,\ \tau_n \,\} \quad \text{and} \quad t' \ = \ \{\, \tau'_1,\ ..,\ \tau'_n \,\}$$

we have

$$t' <: t \quad \text{iff} \quad \forall i = 1..n,\ \ \tau'_i <: \tau_i$$

put differently:
when t is expected, giving more* than t is ok

* precisely, giving more “field by field”

The Case of Methods (1/3)

Consider

$$\begin{array}{lllll} t & = \{\tau\} & \text{with} \quad \tau & \text{being} & f : a \to b \\ t' & = \{\tau'\} & \text{with} \quad \tau' & \text{being} & f : a' \to b' \end{array}$$

where a, a', b, and b' being types

we have $t' <: t$ iff we have $\tau' <: \tau$

the question turns out to be:

- when have we $(a' \to b') <: (a \to b)$?
- what should respectively be the relationship
  - between a and a'?
  - between b and b'?

The Case of Functions (1/2)

so with

```
foo : (f : ftype) -> void =
  v : a
  w : b = f(v) // hyp. H: this is a valid call
  end
```

when is the following code valid?

```
f’ : f’type
foo(f’) // valid call if H is true
```

The Case of Functions (1/3)

let us substitute `f` with `f'`

in the following incomplete code

```
foo : (f' : f'type) -> void =
  v  : a
  v' : a' ...
  w' : b' = f'(v')  // ok since f' : a' -> b'
  w  : b  ...
  end
```

- we want the inner function call to be ok
- so we have modified the original code so that
  - `f'` expects a'
  - `f'` returns b'

The Case of Functions (2/3)

the complete substitution of `f` with `f′` then is

```
foo : (f′ : f′type) -> void =
  v  : a
  v′ : a′ = a        // hyp. 1: this is ok
  w′ : b′ = f′(v′)  // ok since f′ : a′ -> b′
  w  : b  = b′       // hyp. 2: this is ok
  end
```

so

- `f′` expects a' and we give a
  so a should be a sub-type of a'
- `f′` returns b' when we expect b
  so b' should be a sub-type of b

The Case of Functions (3/3)

we have $(a' \rightarrow b') \;<:\; (a \rightarrow b)$

iff

$a' :> a$ and $b' <: b$

The Case of Methods (2/3)

we have $\{\, f : a' \to b' \,\} <: \{\, f : a \to b \,\}$

iff

$a' :> a$ and $b' <: b$

Example

```
struct object {
  virtual object* clone() = 0;
};

struct rabbit : public object {
  virtual rabbit* clone() { return new rabbit; }
};
```

The Case of Methods (2/3)

Exercise: is this program valid?

Example

```
struct animal {
  virtual void eat(food&) = 0;
};

struct cow : public animal {
  virtual void eat(grass& g) { /*...*/ }
};
```

Outline

An Abstract Class without Data

C

```
struct foo {
  virtual bool m() const = 0;
};
```

translation

```
type foo = {
  m : void -> bool
}
```

An Abstract Class with Data (1/3)

C

`class bar {`
`public:`
`  virtual bool m() const = 0;`
`private:`
`  int a;`
`};`

translation
?

An Abstract Class with Data (2/3)

The private part is not accessible so

- the definition (type) of the private part is not known
- but the precise definition exists

translation

$\exists$ `t. type bar = {`
`m : void -> bool,`
`t`
`}`

An Abstract Class with Data (3/3)

actually the *bar* is a sub-type by extension of *foo*

$$bar <: foo$$

foo is the type of the interface (public part) of class `bar`

there is a duality between

- the interface (the type)
- and the class (the implementation)

What is a Class?

A single entity with both interface and implementation: a stand-alone class!

Example

C++ code here:

```
class bar {
public:
  virtual bool m() const { return a > 50; }
private:
  int a;
};
```

we can (should) do better to enforce the duality...

Interface in C++

Example

C++ code here:

```
struct foo { // interface
  virtual void m() = 0;
};

class bar : public foo {
public:
  virtual bool m() const { return a > 50; }
private:
  int a;
};
```

too bad: the same mechanism is used for class inheritance and interface implementation

Interface in Java

Example

Java code here:

```
interface foo {
  void m();
}

class bar implements foo {
  public void m() { return a > 50; }
  private int a;
}
```

Module in Ocaml (1/2)

Example

Ocaml code here:

```
module type FOO =
sig
  type t
  val m : t -> bool
end;;

(* module type FOO = sig type t val m : t -> unit end *)
```

Module in Ocaml (2/2)

Example

Ocaml code here:

```
module Bar =
struct
  type t = { a : int }
  let m t = t.a > 50
end;;

(* module Bar : sig type t = { a : int; } val m : t -> b
```

Outline

Non Imperative types

with:

```
type odd = enum 1, 3, 5..
```

we have:

$$odd <: int$$

since the set of odd integers is included in $\mathbb{N}$

but we do not have an imperative type!

Imperative types

with:

```
type myint = {
  i : int
  set : (j : int) -> void = { i := j }
  get : void -> int = { i }
}
```

and the equivalent for odd integers:

```
type myodd = {
  i : odd
  set : (j : odd) -> void = { i := j }
  get : void -> odd = { i }
}
```

do we have *myodd* $<:$ *myint* ?

Outline

Restrictions and Inheritance

what about a `square` deriving from `rectangle`?

Covariant Methods

- what about feeding the cows?
- what about binary methods (like operator ==)?

Containers and Inheritance (1/2)

Example

what about this Java code?

```
public class Test
{
  public static void doit(Object[] arr, Object o)
  {
    arr[0] = o;
  }
  public static void main(String[] args)
  {
    Integer[] a = new Integer[1];
    a[0] = new Integer(0);
    doit(a, new Object());
  }
}
```

Containers and Inheritance (2/2)

<live!>