Objective Caml:

the language and its type system

Part 2: objects and classes

Overview of objects

An object encapsulates:

- An internal state:
  - one or several *instance variables*;
  - accessible only to the object's methods;
  - mutable or not.
- A collection of *methods*:
  - functions that operate on the internal state;
  - accessible from the outside;
  - they interface the outside world and the internal state.

The only operation on objects is *method sending*, written

object#*method* [*arguments*]

Example: a motor vehicle

Internal state:

- Position of the throttle
- Position of the brake pedal
- Current gear of the gear box
- Rotation speed of the engine

Methods:

- Change position of throttle and brakes
- Shift gears
- Compute current speed of the vehicle

Classes and objects

A class is a template for objects:

- The class defines the methods.
- The class declares the instance variables and define their initial values.
- The class takes as argument some initialization parameters on which initial values of instance variable depend.

The construct `new` returns a new object – a new *instance* of the class:

`new` *class-name* [*initialization parameters*]

```
class motorbike =
  object
    val mutable gear = 0

    method shift =
      if gear > 6 then failwith "shift" else gear <- gear + 1

    method downshift =
      if gear <= 0 then failwith "downshift" else gear <- gear - 1

    method current_gear = gear
  end

let m = new motorbike
     val m : motorbike = <obj>

m#shift; m#shift; m#downshift; m#current_gear
     - : int = 1
```

Extended example: unified input/output

A class encapsulating output on the terminal:

```
class standard_output =
  object
    method put_char c = print_char c
    method put_string s = print_string s
    method put_int n = print_int n
  end

let stdout = new standard_output
     val stdout : standard_output = <obj>

stdout#put_string "Hello!\n"
     Hello!
     - : unit = ()
```

Polymorphism over objects

```
let lire obj francs =
  obj#put_int francs;
  obj#put_string " francs = ";
  obj#put_int (francs * 286);
  obj#put_string " lire\n"

     val lire :
     < put_int : int -> 'a; put_string : string -> 'b; .. > -> int -> 'b
```

`lire` applies to any object containing at least the methods `put_int` and `put_string` with the given types.

```
lire stdout 322;;
     322 francs = 92092 lire
     - : unit = ()
```

What is the type of `stdout`?

The definition of the class `standard_output` also defines a type abbreviation for the type of objects of this class:

```
type standard_output =
    < put_char : char -> unit;
      put_string : string -> unit;
      put_int : int -> unit >
```

File output

```
class fileout filename =
  object
    val chan = open_out filename
    method put_char c = output_char chan c
    method put_string s = output_string chan s
    method put_int n = output_string chan (string_of_int n)
    method seek pos = seek_out chan pos
    method close = close_out chan
  end
```

In addition to the `put_` operations seen before, we add random access (`seek`) and explicit closing (`close`).

Using the class `fileout`:

```
let f = new fileout "/tmp/myfile";;
f#put_string "Hello, world!\n";;
lire f 1234;;
f#seek 7;;
f#put_string "monde";;
f#close;;
```

The call `lire f 1234` is well-typed because `f` provides the required methods `put_string` and `put_int` (and more).

Default implementations of methods

When we know how to do `put_char`, we also know how to implement `put_string` (print each character in turn) and `put_int` (convert the integer to a string, then print the string).

However, on certain kinds of output, we have more efficient implementations (e.g. `put_string` on a file, where we can write a whole block in one operation).

- We can use a *virtual class* to define the default implementations of methods once and for all.
- We then use *inheritance* between classes to reuse the default implementations in a particular output class.
- In that output class, we can *redefine* methods for which we have a better implementation.

The virtual class of “writers”

```
class virtual writer =
  object (self)
    method virtual put_char : char -> unit
    method put_string s =
      for i = 0 to String.length s - 1 do self#put_char s.[i] done
    method put_int n =
      self#put_string (string_of_int n)
  end
```

This class is marked “virtual” because some of its methods are not defined (`put_char`); thus we can’t create instances of this class.

The identifier `self` is bound (at the time of method invocation) to the object itself. It allows one method to call another method of the same object.

From `writer` to `fileout` by inheritance

A simple implementation of file output inherits `put_string` and `put_int` from class `writer`, and defines only the other methods:

```
class fileout filename =
  object
    inherit writer
    val chan = open_out filename
    method put_char c = output_char chan c
    method seek pos = seek_out chan pos
    method close = close_out chan
  end
```

This class is functionally equivalent to the previous `fileout` class (implemented without inheritance):

```
let f = new fileout "/tmp/myfile";;
f#put_string "Hello, world!\n";;
f#close;;
```

From `writer` to `fileout`, optimized

In class `fileout`, we can provide a more efficient implementation of `put_string`, which then hides the default implementation inherited from `writer`:

```
class fileout filename =
  object
    inherit writer
    val chan = open_out filename
    method put_char c = output_char chan c
    method put_string s = output_string chan s
    method seek pos = seek_out chan pos
    method close = close_out chan
  end
```

We keep the default implementation of `put_int`, because it is efficient enough.

Late binding (open recursion)

Bindings between methods are (roughly speaking) determined when we do `new`, but not yet when the classes are defined.

This is called *late binding* by opposition with *static binding* used by default in the core language.

```
class c =
  object (self)
    method a = print_string "c"           let a() = print_string "c"
    method b = self#a                     let b() = a()
  end

class d =
  object
    inherit c
    method a = print_string "d"           let a() = print_string "d"
  end

let o = new d

o#b                                       b()
(* prints "d" *)                           (* prints "c" *)
```

`private` methods

A method declared `private` cannot be invoked from outside the object, but only from another method of the object. This is useful to define operations local to a class.

```
class stringout =
  object (self)
    inherit writer
    val mutable accu = String.create 20
    val mutable pos = 0
    method private redim =
      let newsize = String.length accu * 2 in
      let newaccu = String.create newsize in
      String.blit accu 0 newaccu 0 (String.length accu);
      accu <- newaccu
    method put_char c =
      if pos >= String.length accu then self#redim;
      accu.[pos] <- c;
      pos <- pos + 1
    method contents = String.sub accu 0 pos
  end
```

Subtyping between objects

Assume B has more methods than A, with the same types for common methods:

$A : \langle m_1 : \tau_1;\ \ldots;\ m_k : \tau_k \rangle \qquad B : \langle m_1 : \tau_1;\ \ldots;\ m_k : \tau_k;\ m_{k+1} : \tau_{k+1};\ \ldots \rangle$

Then B can be used anywhere in place of A.

We say that B's type is *subtype* of the type of A
(think about types as sets and subtypes as subsets).

Application: store objets of different types in an homoegeneous data structure (list, array), as long as all those objects have a common supertype.

Subtyping in Objective Caml

```
let w1 = new fileout "/tmp/myfile" in
let w2 = new stringout in

let writers = [(w1 :> writer); (w2 :> writer)] in

List.iter (fun w -> w#put_string "Hello") writers;
w1#close;
w2#contents
```

In Objective Caml, subtyping must be marked explicitly as a type coercion:

(*object* :> *type*)

(*object* : *source-type* :> *result-type*)

Polymorphism vs. subtyping

Polymorphism allows functions that apply to arguments of different types, but not heterogeneous data structures:

`List.rev [1;2];;`	*- : int list = [2;1]*
`List.rev [true;false];;`	*- : bool list = [false;true]*
`[ [1]; [true] ];;`	*Type error*

`int list` and **`bool list`** do not have a common supertype.

Subtyping is required to support heterogeneous data structures.

However, polymorphism generally suffices for writing generic functions over objects.

Referencing the superclass

After generic outputs, we now do generic inputs in the same way:

```
class virtual reader =
  object (self)
    method virtual get_char : char
    method get_string n =
      let s = String.create n in
      for i = 0 to n - 1 do s.[i] <- self#get_char done;
      s
    method get_int =
      let rec read accu =
        match self#get_char with
          '0'..'9' as c -> read (10 * accu + Char.code c - Char.code '0')
        | _ -> accu
      in read 0
  end
```

Problem: `get_int` reads one extra character at the end!

Referencing the superclass, contd

The problem is easily fixed if we can do `seek` on the input: back off one character after reading the integer.

```
class virtual seekable_reader =
  object (self)
    inherit reader as super
    method virtual seek : int -> unit
    method virtual pos : int        (* current reading position *)
    method get_int =
      let res = super#get_int in
      self#seek (self#pos - 1);
      res
  end
```

The `as super` clause gives the name `super` to methods of the superclass. `super#get_int` then references the implementation of `get_int` in the superclass. We can thus redefine an inherited method while re-using the inherited implementation.

Reading from files

```
class filein filename =
  object
    inherit seekable_reader
    val chan = open_in filename
    method get_char = input_char chan
    method seek pos = seek_in chan pos
    method pos = pos_in chan
    method close = close_in chan
  end
```

Multiple inheritance

A class can inherit from several classes at once:

```
class pipe =
  object
    val q = Queue.create ()
    inherit writer
    inherit reader
    method put_char c = Queue.add c q
    method get_char =
      try Queue.take q
      with Queue.Empty -> raise End_of_file
  end
```

An instance of `pipe` can be used both as a writer and as a reader:

```
let p = new pipe;;
lire p 100;;
p#get_string 13;;             - : string = "100 francs = "
p#get_int;;                   - : int = 28600
```

Subtyping and inheritance

Subtyping:

- allows to consider different implementations with the same interface
- supports generic functions
- allows mixing different classes in an homogeneous data structure.

Inheritance:

- supports incremental programming and code reuse;
- provides late binding.

In many object-oriented languages, subtyping is tied to inheritance: B is subtype of A if and only if B inherits from A.

In contrast, OCaml separates entirely subtyping and inheritance:

- B can be subclass of A without inheriting from A. (Independent implementations of the same interface, or of interfaces in subtyping relation.)

- B can inherit from A without being a subtype of A. (Happens with self type + binary methods.)

Inheritance is not subtyping (Cook et al, 1989)

```
class point =
  object (self: 'selftype)
    val x = 0
    method coord = x
    method equal (p : 'selftype) = (p#coord = x)
  end

class colorpoint =
  object (self: 'selftype)
    inherit point as super
    val c = "black"
    method colour = c
    method equal (p : 'selftype) = (p#coord = x) && (p#colour = c)
  end
```

`colorpoint` is not a subtype of `point`; otherwise the following bad code would typecheck:

```
let p = new point and cp = (new colorpoint :> point) in cp#equal p
```

Parameterized classes

Like data types, classes can be parameterized by types, which makes them more polymorphic.

Typical example: *container classes*.

```
class ['a, 'b] hashtable size =
  object
    val buckets = (Array.make size [] : ('a * 'b) list array)
    method find key =
      let h = (Hashtbl.hash key) mod (Array.length buckets) in
      List.assoc key buckets.(h)
    method add key data =
      let h = (Hashtbl.hash key) mod (Array.length buckets) in
      buckets.(h) <- (key, data) :: buckets.(h)
end
```

Object encoding of data types

```
class virtual ['a] list = object (self)
  method virtual isnil : bool
  method virtual head : 'a
  method virtual tail : 'a list
  method length =
    if self#isnil then 0 else 1 + self#tail#length
end
```

```
class 'a nil = object
  inherit ['a] list
  method isnil = true
  method head = failwith "nil"
  method tail = failwith "nil"
end
```

```
class 'a cons h0 t0 = object
  val h = h0
  val t = t0
  inherit ['a] list
  method isnil = false
  method head = h
  method tail = t
end
```

This encoding is popular in Java, but nearly useless in Caml, as built-in data types are more convenient.

Type systems for objects

Main difficulty: combining subtyping (implicit subsumption) with type inference.

Several approaches:

1. Full implicit subsumption, no or partial type inference (most typed OO languages: Java, Eiffel, . . .)

2. Full ML-style type inference, explicit subsumption (`obj :> ty`), implicit polymorphic instantiation (Objective Caml)

3. Full implicit subsumption and full type inference (Scott Smith et al)

We will cover 2 (in details) and 3 (more quickly).

Closed object types

When all methods of an object are known, the type of an object is just the names and types of its methods.

```
class point =
  object
    val mutable x = 0
    method pos = x
    method move d = x <- x + d
  end
```

```
new point : < pos: int; move: int → unit >
```

The order of methods in an object type does not matter.

Instance variables do not show up in the object type, as they are accessible only from the methods of the object, but not from outside.

Open object types

When only some methods of an object are required (e.g. an object parameter to a function), the object type is the names and types of these methods, plus an *extension variable* (row variable)

```
let translate pt = pt#move 5; pt
translate : < move: int → α; ρ > → < move: int → α; ρ >
```

ρ can be instantiated to any object type to satisfy type equality constraints:

```
translate (new point)
```

ρ *becomes* `pos: int`, α *becomes* `int`

```
translate: < move: int → int; pos: int > → < move: int → int; pos: int >
```

```
translate (new point): < move: int → int; pos: int >
```

Recursive types

Object types are naturally recursive (infinite but regular):

```
class point =
  object (self)
    val mutable x = 0
    method pos = x
    method move d = (x <- x + d; self)
  end

new point: < pos: int;
             move: int -> < pos: int;
                            move: int -> < pos: int;
                                           move: int -> < ... > > > >
```

```
new point: μα. < pos: int; move: int -> α >
```

$\mu\alpha.\tau[\alpha]$ is the unique finite or infinite type T such that $T = \tau[T]$.

The type algebra

Type expressions:	$\tau ::= \texttt{int}, \texttt{bool}, \ldots$	base types
	$\mid \alpha$	type variables
	$\mid \tau_1 \rightarrow \tau_2$	function types
	$\mid \langle \varphi \rangle$	object types
	$\mid \mu\alpha.\tau$	recursive types
Row expressions:	$\varphi ::= \emptyset$	the empty row
	$\mid \rho$	row variables
	$\mid m : \tau;\ \varphi$	$(m : \tau)$ plus whatever is in φ

Equations on types and rows: left commutativity, roll/unroll.

$$\begin{aligned} m_1 : \tau_1;\ m_2 : \tau_2;\ \varphi &= m_2 : \tau_2;\ m_1 : \tau_1;\ \varphi \\ \mu\alpha.\tau &= \tau\{\alpha \leftarrow \mu\alpha.\tau\} \end{aligned}$$

Structural invariants on rows

- A given method name m appears only once in a row

```
Bad:   < m : int; m : bool; ∅ >
Good:  < m : int; m2 : bool; ∅ >
```

- A given row variable is always preceded by the same set of method names

```
Bad:   < m : int; ρ > and < ρ >
Good:  < m : int; ρ > and < m : bool; ρ >
```

Kinding of rows

The invariants are enforced by *kinding rules*.

The *kind* κ of a row expression is the set of method names that must not appear in it.

$$\vdash \rho :: K(\rho) \qquad \vdash \emptyset :: \kappa \qquad \frac{\vdash \tau \ \texttt{ok} \qquad \vdash \varphi :: \kappa \cup \{m\}}{\vdash (m : \tau;\ \varphi) :: \kappa}$$

$$\frac{\vdash \varphi :: \emptyset}{\vdash \langle \varphi \rangle \ \texttt{ok}}$$

A pure object calculus (no classes)

Expressions:

$a ::= c$	constants
$\mid x$	identifiers
$\mid \texttt{fun}\ x \to a$	function abstraction
$\mid a_1\ a_2$	function application
$\mid \texttt{let}\ x = a_1\ \texttt{in}\ a_2$	the `let` binding
$\mid \texttt{obj}(s)\langle \texttt{field}\ f_i = a_i; \texttt{method}\ m_j = a_j \rangle$	object construction
$\mid a\#m$	method invocation

Typing rules for the pure object calculus

$$\frac{E = \ldots, \texttt{field}\ f : \tau, \ldots}{E \vdash f : \tau} \qquad \frac{E = \ldots, \texttt{self}\ s : \tau, \ldots}{E \vdash s : \tau}$$

$$\frac{\bar{E} \vdash a_i : \tau_i \qquad \bar{E} \oplus \{\texttt{field}\ f_i : \tau_i\} \oplus \{\texttt{self}\ s : \langle m_j : \tau_j; \emptyset\rangle\} \vdash a_j : \tau_j}{E \vdash \texttt{obj}(s)\langle \texttt{field}\ f_i = a_i; \texttt{method}\ m_j = a_j\rangle : \langle m_j : \tau_j; \emptyset\rangle}$$

$$\frac{E \vdash a : \langle m : \tau; \varphi\rangle}{E \vdash a\# m : \tau}$$

$\bar{E}$ is E without any `field` and `self` bindings.

Evaluation rules

The "self application" semantics: when evaluating $a\#m$, we bind the self parameter to the object a itself in the body of the method.

$$o\#m \ \stackrel{*}{\rightarrow} \ a\{s \leftarrow o, f_i \leftarrow v_i\}$$

if $o = \texttt{obj}(s)\langle\texttt{field}\ f_i = v_i; \texttt{method}\ m = a; \ldots\rangle$.

Values and evaluation contexts:

Values: $\quad v ::= \ldots \mid \texttt{obj}(s)\langle\texttt{field}\ f_i = v_i; \texttt{method}\ m_j = a_j\rangle$

Contexts: $\quad E ::= \ldots \mid \texttt{obj}(s)\langle\texttt{field}\ f_i = v_i; \texttt{field}\ f = E;$
$\texttt{field}\ f_k = a_k; \texttt{method}\ m_j = a_j\rangle$

Type inference

Same approach as in the first lecture: set up type constraints and solve them by unification.

- If $a = \texttt{obj}(s)\langle\texttt{field } f_i = a_i; \texttt{method } m_j = a_j\rangle$:
  add $T_a = T_s = \langle m_j : T_{a_j}; \emptyset\rangle$
  and $T_{f_i} = T_{a_i}$

- If $a = b\#m$: add $T_b = \langle m = T_a; \rho\rangle$

Solving the constraints

This is a first-order unification problem modulo equations:

- Fold-unfold of recursive types: unification between rational trees (graphs)

- Left commutativity in rows: unification modulo a regular equational theory

To unify $\langle m : \tau; \varphi \rangle$ and $\langle m : \tau'; \varphi' \rangle$:
  unify $\tau = \tau'$ and $\varphi = \varphi'$.

To unify $\langle m : \tau; \varphi \rangle$ and $\langle n : \tau'; \varphi' \rangle$ when $m \neq n$:
  unify $\varphi = n : \tau'; \rho$ and $\varphi' = m : \tau; \rho$ where ρ is fresh.

We have principal unifiers, thus principal types.

Subtyping coercions

To handle subtyping, we just assume given constants $C_{\tau,\sigma} : \forall \alpha_1, \ldots, \alpha_n . \tau \to \sigma$ for all τ, σ such that $\tau <: \sigma$.

The subtyping relation $<:$ is standard:

$$\texttt{int} <: \texttt{int} \qquad \alpha <: \alpha \qquad \frac{\tau' <: \tau \qquad \sigma <: \sigma'}{\tau \to \sigma <: \tau' \to \sigma'} \qquad \frac{\varphi <: \varphi'}{\langle \varphi \rangle <: \langle \varphi' \rangle}$$

$$\varphi <: \emptyset \qquad \rho <: \rho \qquad \frac{\tau <: \tau' \qquad \varphi <: \varphi'}{(m : \tau;\ \varphi) <: (m : \tau';\ \varphi')}$$

Adding classes

Expressions:	$a ::= \ldots$	(the functional subset)
	$\mid \texttt{new}(c)$	object creation
	$\mid a\#m$	normal method invocation
	$\mid s\#m$	super method invocation
	$\mid \texttt{class}\ z = c\ \texttt{in}\ a$	class binding
Class expressions:	$c ::= z$	class name
	$\mid \texttt{fun}\ x \to c$	parameterized class
	$\mid c(a)$	class application
	$\mid \texttt{object}(x)\ b\ \texttt{end}$	basic class
Class bodies:	$b ::= \emptyset \mid d;\ \ b$	
Class components:	$d ::= \texttt{inherit}\ c\ \texttt{as}\ s$	
	$\mid \texttt{field}\ f = a$	
	$\mid \texttt{method}\ m = a$	

Class types

Class types: $\gamma ::= \texttt{object}(x : \tau)\ \delta\ \texttt{end} \mid \tau \to \gamma$

Class type bodies: $\delta ::= \emptyset$

$\mid \delta;\ \texttt{field}\ f : \tau$

$\mid \delta;\ \texttt{method}\ m : \tau$

$\mid \delta;\ \texttt{super}\ s : \delta$

Class type scheme: $\Sigma ::= \forall \alpha_1 \ldots \alpha_n.\ \gamma$

In $\texttt{object}(x : \tau)\delta\ \texttt{end}$, methods that are in τ but not in δ are virtual methods:

```
object(self: μα. <copy: α; getx: int; ρ)
  method getx: int
end
```

This class has one virtual method `copy`.

Typing rules in the presence of classes

$$\frac{E \vdash c : \texttt{object}(s : \tau)\ \delta\ \texttt{end} \qquad \tau = \langle \texttt{method}(\delta) \rangle}{E \vdash \texttt{new}(c) : \tau}$$

For class components: $E \vdash d : \delta$

For class bodies: $E \vdash b : \delta$

For class expressions: $E \vdash c : \gamma$

(Full rules in the paper by Rémy and Vouillon in the reading list.)

Full type inference with implicit subsumption

- From the program text, generate a set C of subtyping constraints $T_a <: T_b$ (instead of equality constraints as in ML-style inference).
- Do not solve those constraints C, as there is no single principal solution and the set of minimal solutions can be exponentially large.
- Instead, just check that C is consistent (admits at least one solution) and return C itself as “the” “type” of the program.

Generation of subtyping constraints

- If $a = x$: add $T_x <: T_a$
- If $a = c$: add $TC(c) <: T_a$
- If $a = \texttt{fun}\ x \to b$: add $T_a = T_x \to T_b$
- If $a = b(c)$: add $T_b = \alpha \to T_a$ and $T_c <: \alpha$
- If $a = \texttt{obj}(s)\langle\texttt{field}\ f_i = a_i; \texttt{method}\ m_j = a_j\rangle$: add $T_a = T_s = \langle m_j : T_{a_j}\rangle$ and $T_{f_i} = T_{a_i}$
- If $a = b\#m$: add $T_b <: \langle m = T_a\rangle$

Consistency check

First *saturate* the set of constraints C by the following structural rules:

- If $(\tau_1 <: \tau_2) \in C$ and $(\tau_2 <: \tau_3) \in C$, add $(\tau_1 <: \tau_3)$
- If $(\tau_1 \to \sigma_1 <: \tau_2 \to \sigma_2) \in C$, add $(\tau_2 <: \tau_1)$ and $(\sigma_1 <: \sigma_2)$.
- If $(\langle \ldots ; m : \tau_1; \ldots \rangle <: \langle \ldots ; m : \tau_2; \ldots \rangle) \in C$, add $(\tau_1 <: \tau_2)$.

Then check that the saturated set $\bar{C}$ does not contain any inconsistent constraints such as

$$\texttt{int} <: \tau \to \sigma \quad \text{or} \quad \tau \to \sigma <: \langle \ldots \rangle \quad \text{or} \quad \langle \ldots \rangle <: \texttt{int} \quad \text{or} \ldots$$

This suffices to guarantee that the program does not go wrong at run-time.

Problems with constraint-based inference

- The set of type constraints can be very large (e.g. the size of the program itself!)
- Simplifications are possible, but still result in large sets.
- Type errors are very hard to read.

Not yet a viable alternative, but needs to be investigated further.