Programming with
Objective Caml

* * *

Pascal Manoury

– Sept. 2002 –

* * *

Shanghai Jiao Tong University
Department of Computer Science and Engineering

The Objective Caml language

In few words

Fully *functionnal + imperative* controls

High level *modules* structuration facilities

Object oriented extension with parametrized classes

Concurrent and network API's

⋆

Strongly typed with *polymorphic types*

Powerfull type *inference* algorithm

⋆

Efficient automatic *garbage collection*

⋆

Bytecode and *native* compilers

Interactive *toplevel*

`http://caml.inria.fr`

Genealogy

The ML familly

80-81 ML: *Meta-Language* for LCF proof assistant
(R. Milner)

84-... CAM: *Categorical Abstract Machine* (P-L. Curien)
“a compiling technique for ML” (G. Cousineau)

Standard ML design (R. Milner)

87-... Caml’s first implementation $\not\approx$ SML
(A. Suarez, P. Weiss, M. Mauny)

90-91 Zinc abstract machine + native code compiler
⇒ Caml Light (X. Leroy, D. Doligez)

96-... Objective Caml (J. Vouillon, D. Rémy)
Modules, objects, ...

Thanks to G. Cousineau’s
A brief history of Caml (as I remember it)
`http://www.pps.jussieu.fr/~cousinea/Caml/caml_history.html`

The unavoidable “`Hello world`”

- Let’s the text file `hello.ml` contain the phrase

```
print_string "Hello world\n"
```

The “program” simply consists of an *expression*:
the *application* of the *function* `print_string` to the string *argument* `"Hello world\n"`

Note the lack of parenthesis

- Compile it with command

```
[shell-prompt] ocamlc -o hello hello.ml
```

⇒ file `hello`: *ocamlrun*[†] script text executable

- Run your new “hello” command

```
[shell-prompt] ./hello
Hello world
[shell-prompt]
```

[†] use `ocamlopt` compiler to get a *native code*

“`Hello world`” at the ***toplevel***

- Run the ocaml interactive toplevel

```
[shell-prompt] ocaml
        Objective Caml version 3.04

#
```

- Tape in the expression to evaluate

```
# print_string "Hello world\n" ;;
Hello world
- : unit = ()
```

- What happened ?

1. the expression was
   (a) parsed and type-checked
   (b) compiled (to byte-code)
   (c) evaluated
   ⇒ `Hello world` side effected
2. the toplevel displays
   (a) the *type* of the result: `unit` (≈ the `void` of C)
   (b) the *value* of the result: `()` (THE value for *nothing*)

What did we learn ?

From the “Hello world” example:

1. any expression has a *value*
2. the (unique) value for *nothing* id written `()`; it belongs to type `unit`
3. strings are as usual (`" \n`)
4. `print_string` is a *function* which returns the value `()` when applied to a `string`

Check the type of `print_string` using the toplevel (and learn more):

```
# print_string ;;
- : string -> unit = <fun>
```

1. the *function* `print_string` *is a value* (written `<fun>`)
2. it has the type of functions from (values of type) `string` to `unit`
   (written `string -> unit`)

Note the double semi-colons (`;;`) used at toplevel
⇒ *“lets do the job !”*

A polyglot world

File `hello1.ml`

```
let string_select n =
  if n = 0 then "Hello"
  else if n = 1 then "Ni Hao"
  else "Bonjour" (* french is the default :*)
;;

print_newline () ;
print_endline " - Menu -" ;
print_endline "0: english" ;
print_endline "1: chinese" ;
print_endline "2: french" ;
print_string "\nYour favorite: " ;
print_string ("\n"
              ^(string_select (read_int ()))
              ^" \"world\"\n") ;
print_newline ()
```

Two parts in this program:

1. a *function definition*: `string_select`
   (as a *conditional expression*)
2. a *"main"* expression
   (a *sequence* of printings)

Note: both of function's body and "main" sequence *are expressions*

What's new ?

Minor novelties

- Objective Caml knows *integers* and *boolean*

  type `int` ($[-2^{30}, 2^{30}-1]$ or $[-2^{62}, 2^{62}-1]$)

  type `bool` (values: `true`, `false`)

- New printing functions:

```
print_newline :  unit -> unit
print_endline :  string -> unit
```

- The concatenation string operator

  `^ :  string -> string -> string` (infix)

- An input function (for integers)

```
read_int :  unit -> int
```

applied to *nothing* (i.e. the value `()`), returns the `stdin` input

- The comments are opened with `(*` and closed with `*)`

Nested comments are allowed.

What's new again ?

The *sequence* control operator

Syntax: $\boxed{exp_1 \ ; \ exp_2}$

Effect of `e1 ; e2`:

evaluates the expression `e1` *and then* evaluates the expression `e2`

Value of `e1 ; e2`:

the value of `e2`

Remarks:

- I did not say *"executes the instruction…"*
  ⇒ no instructions but expressions of type `unit`
- `e1 ; e2` is itself an expression
- the compiler warns you when `e1` has not type `unit`

Associates to right:

`e1 ; e2 ; e3` = `e1 ; (e2 ; e3)`

What's the good news ?

One can define functions in Objective Caml !

Syntax: `let` $id\ id_1 \ldots id_n$ `=` exp

Semantics (first approach)

Consider `let f x = e`

Static (typing)

if `e` has type $\mathtt{T}_2$, assuming `x` has type $\mathtt{T}_1$ then

`f` has type $\mathtt{T}_1$ `->` $\mathtt{T}_2$

Dynamic (evaluation)

for any value of `x` (with right type)

the value of `f x` is equal to the value of `e`

Simple exemple (using the toplevel):

```
# let to_the_square n = n * n ;;
val to_the_square : int -> int = <fun>
```

Remark:

- no need to precise any type
- no need to precise any *return*

Conditional control structure

Syntax: `if` exp_0 `then` exp_1 `else` exp_2

- conditional constructs are *expressions*:

```
# "Hello " ^ (if true then "China" else "world") ;;
- : string = "Hello China"
```

Typing:

- exp_0 must have type `bool`
- exp_1 and exp_2 may have *any* type, but *the same*

```
# if true then "Hello world" else 0;;
This expression has type int but is here used with
type string
```

Control and value:

The condition is evaluated first and then, either the "true" alternative, either the "false" one.

```
# if 0 = 0 then 0/2 else 2/0 ;;
- : int = 0
```

When things go wrong

Exceptions

Assume the following silly answer to our “hello” program:

```
 - Menu -
0: english
1: chinese
2: french

Your favorite: any
Fatal error: exception Failure("int_of_string")
```

The program stops *raising an exception* due to an unexpected input.

Catching exceptions

Syntax: **`try`** exp_1 **`with`** exn **`->`** exp_2

A protected **`read_int`** with default value

```
let read_int_with_default n =
  try read_int ()
  with (Failure "int_of_string") -> n
;;
```

Compound data structures

Arrays and `for` loops

```
let menu_tab = [| "english"; "chinese"; "french" |] ;;

let print_menu () =
  print_newline () ;
  print_endline " - Menu -" ;
  for i=0 to 2 do
    Printf.printf"%d: %s\n" i menu_tab.(i)
  done;
  print_string "\nYour favorite: "
;;
...
print_menu ();
Printf.printf "\n %s \"world\"\n\n"
              (string_select (read_int_with_default 2))
```

Minor (not so) novelty

- Formatted output *à la* C: `Printf.printf`
  ⇒ from the *module* `Printf`
- constant definition (`menu_tab`)
  ⇒ as "functions" with no argument

Array's basics

Constants

Syntax: `[|` *exp*$_0$ `;` … `;` *exp*$_n$ `|]`

- *exp*$_0$, …, *exp*$_n$ may have *any* type, but the *same*
- the length is $n + 1$
- the first index is 0, the last is n
  (exception `Invalid_argument "Array.get"` if outside of range)

The type `array` is a *parametrized type*:

`[| e1; ...; en |]` has type written `t array` if `t` is the (common) type of `e1`, …, `en`.

Access

Syntax: *exp*$_1$`.(` *exp*$_2$ `)`

- *exp*$_1$: any expression which value is an array
- *exp*$_2$: any expression which value is an integer

`For loop`

Syntax: `for` *id* `=` exp_1 `to` exp_2 `do` exp_3 `done`

Typing:

- exp_1 and exp_2 must have type `int`
- exp_3 may have any type (but the compiler warns you if it has not type `unit`)
- the loop expression itself has type `unit`

Value: the constant `()`

Effect: Pascal-like

The loop index is local and "read only"

Decreasing variant

Syntax: `for` *id* `=` exp_1 `downto` exp_2 `do` exp_3 `done`

More compound data structures

```
let read_bound_int n =
  try
   let m = read_int () in
    if (m < 0) or (n < m) then n else m
  with (Failure "int_of_string") -> n
;;

let choice_tab =
  [| ("english", "Hello");
     ("chinese", "Ni Hao");
     ("french",  "Bonjour") |] ;;

let max_choice = (Array.length choice_tab) - 1 ;;

let get_choice () =
  print_newline () ;
  print_endline " - Menu -" ;
    for i=0 to max_choice do
      Printf.printf" %d: %s\n" i (fst choice_tab.(i))
    done;
  print_string "\nYour favorite: " ;
read_int_with_default max_choice
;;

Printf.printf "\n %s \"world\"\n\n"
              (snd choice_tab.(get_choice ()))
```

What's new again and again ?

Local definition

Syntax: `let` id `=` exp_1 **`in`** exp_2

- Note: `let x = e1 in e2` in an *expression*
- Value: the value of `e2` where `x` has the value of `e1`
- Control: `e1` is evaluated *before* `e2` is

```
# let x = print_string"Hello" in
    print_string" world\n" ;;
Hello world
- : unit = ()
```

- Typing: the type of `e2` assuming that `x` has the type of `e1`

Module `Array`

val length : 'a array -> int

Return the length (number of elements) of the given array.

Note the unknown type variable notation `'a`

⇒ *polymorphic* arrays

Fully qualified name: `Array.length`

Product type

Pairs

Syntax: $\boxed{exp_1 \textbf{ , } exp_2}$

- Remark: (external) parenthesis are not mandatory, but I recommand them !

- Value: the value of `(e1 , e2)` is the pair of values of `e1` and `e2`

  ⇒ the comma (`,`) is the *constructor* of pair values

- Typing: `(e1 , e2)` has $\mathtt{T}_1$ `*` $\mathtt{T}_2$ if `e1` has type $\mathtt{T}_1$ and, `e2`, type $\mathtt{T}_2$

  ⇒ the star (`*`) is the *type constructor* of pair values

Accessors (pair operation)

val fst : 'a * 'b -> 'a

Return the first component of a pair.

val snd : 'a * 'b -> 'b

Return the second component of a pair.

⇒ *polymorphic* functions

A more fair choice

```
let check_bound n m =
  if (m < 0) or (n < m) then
      failwith "Out of bound"
    else m
;;

let rec read_bound_int n =
  try
   check_bound n (read_int ())
  with
    _ -> begin
          print_string "Try again: ";
          read_bound_int n
         end
;;
```

Two new features:

1. recursive loop (`let rec`)
2. raising exception (`failwith`)

Minor novelties

- `begin` and `end` are (resp.) left and right parenthesis
- the special exception *pattern* (char. _) catches any (here: `Failure "int_of_string"` or `Failure "Out of bound"`)

Recursive definitions

Self reference

Syntax: $\boxed{\texttt{let rec}\ id\ id_1 \ldots id_n\ \texttt{=}\ exp}$

The defining expression *exp* can make usage of the defined identifier *id*.

Example the exponentiation x^n

```
# let rec expn x n =
    if n = 0 then 1
    else x * (expn x (n - 1))
val expn : int -> int -> int = <fun>
```

Typing

if `e` has type $\texttt{T}_2$,
assuming `x` has type $\texttt{T}_1$ and `f` has type $\texttt{T}_1$ `->` $\texttt{T}_2$ then
`f` has type $\texttt{T}_1$ `->` $\texttt{T}_2$

Evaluation:

for any value of `x` (with right type)
the value of `f x` is equal to the value of `e`
(where the value of `f x` is equal to the value of `e` !)

More on recursive definitions

- recursive definitions must be *explicitly recursive*

  `let` $\neq$ `let rec`

```
# let expn x n =
    if n = 0 then 1
    else x * (expn x (n - 1)) ;;
Unbound value expn
```

- Don't define silly values

```
# let rec x = x ;;
This kind of expression is not allowed as
right-hand side of ‘let rec’
```

```
# let app f x = f x ;;
val app : ('a -> 'b) -> 'a -> 'b = <fun>
# let rec x = app x ;;
This kind of expression is not allowed as right-hand
side of ‘let rec’
```

Or do it properly

```
# let rec f x = f x ;;
val f : 'a -> 'b = <fun>
```

- in the first and second case: compiler can't assign value to **`x`**
- while in the third: value of `f` is a *closure* (see forward p.50)

Exceptions

A function to raise exceptions (module `Pervasives`)

val failwith : string -> 'a

Raise exception Failure with the given string.

Any type as result ⇒ can be used any where

- Exceptions are *values* and belong to a (special) type

type exn

The type of exception values.

- `Failure` is a *constructor* of type `exn`

exception Failure of string

Exception raised by library functions to signal that they are undefined on the given arguments.

```
# 2 / 0 ;;
Exception: Division_by_zero.
```

- Some other predifined exceptions : `Invalid_argument`; `Division_by_zero`; `End_of_file`; etc.

Remark: the capitalized initial

Exceptions again

The general raising exception *builtin* function

`val raise : exn -> 'a`

Raise the given exception value

Defining a *new* exception

Syntax: `exception` *Id* [`of` *type*]

Notice again the capitalized initial: *mandatory*

Catching several exceptions

Syntax:

```
try exp with
   Exn_1 -> exp_1
 | ⋮
 | Exn_n -> exp_n
```

- Typing: *exp*, exp_1, …and exp_n may have any type, but the same; Exn_1, dots, Exn_n have type `exn`
- Value: the one of *exp*, or the one of exp_i if *exp fails with* Exn_i, or some other uncatched exception
- Control: *exp* is evaluated fisrt, and then one of the exp_i's, if needed

Refined error handling

```
exception Out_of_bound of int ;;

let check_bound n m =
  if (m < 0) or (n < m) then
    raise (Out_of_bound m)
  else m
;;

let rec read_bound_int n =
  try
   check_bound n (read_int ())
  with
      Out_of_bound m ->
        (Printf.printf "%d is out of bound: " m;
         read_bound_int n)
    | Failure "int_of_string" ->
        (Printf.printf "Please give a number: ";
         read_bound_int n)
    | e ->
        (Printf.printf "Unknown error: "; raise e)
;;
```

- Note the last exception case:

the *variable* `e` stands for any exception other than `Out_of_bound m` and `Failure "int_of_string"`;

⇒ it is *reraised*

Exceptions as control

Beware values of exceptions are not their raising

```
# let check_bound n m =
    if (m < 0) or (n < m) then
      Out_of_bound m
    else m
  ;;
This expression has type int but is here used with
type exn
```

- Raising an exception causes a *break* in computation

The execution of the programm

```
try
  for i=0 to 10 do
    if i < 5 then Printf.printf"(%d)" i
    else raise Exit
  done
with
    Exit -> print_endline "\nBye bye\n"
```

will give the output

```
(0)(1)(2)(3)(4)
Bye bye

```

Labeled product

Pascal's `records` or C `struct`

Type definition

Syntax: $\boxed{\texttt{type}\ id\ \texttt{=}\ \{\ id_1\ \texttt{:}\ ty_1\ \texttt{;}\ \ldots\ \texttt{;}\ id_n\ \texttt{:}\ ty_n\ \}}$

where

- *id* is the name of a *new* type
- $id_1 \ldots id_n$ name labels of components
- $ty_1 \ldots ty_n$ are their respective types

Values

Syntax: $\boxed{\{id_1\ \texttt{=}\ exp_1\ \texttt{;}\ \ldots\ \texttt{;}\ id_n\ \texttt{=}\ exp_n\ \}}$

Exemple:

```
type menu_data = { lang : string; word : string; }
let choice_tab =
  [| { lang = "english"; word = "Hello" } ;
     { lang = "chinese"; word = "Ni Hao" } ;
     { lang = "french" ; word = "Bonjour" } |]
```

The constant `choice_tab` has type `menu_data array`

Labeled product (continued)

The labels free from the order

```
# { lang = "chinese"; word = "Ni Hao" }
  = { word = "Ni Hao"; lang = "chinese"} ;;
- : bool = true
```

Labels give access to components

Syntax: $\boxed{\textit{exp}\ .\ \textit{id}}$

where

- *id* must be the name of a label in a known record type
- *exp* any expression of this type

```
let print_item i item =
   Printf.printf" %d: %s\n" i item.lang ;;

let get_choice () =
  print_endline "\n - Menu -" ;
  for i=0 to max_choice do
    print_item i choice_tab.(i)
  done;
  print_string "\nYour favorite: " ;
  read_bound_int max_choice ;;

Printf.printf "\n %s \"world\"\n\n"
              choice_tab.(get_choice ()).word
```

Mutable data structure

The value of record's fields can be modified in-place when declared so

Syntax: `mutable` *id* `:` *ty*

Assignment

Syntax: exp_1`.`*id* `<-` exp_2

- Typing exp_2 has the type declared for *id*
- Effect the value of field the *id* of the record exp_1 becomes the value of exp_2
- Value the value of the assignment itself is `()`

Array's cells are also *mutable*

Syntax: exp_1`.(`exp_2`)` `<-` exp_3

- Typing exp_1 has type `t array` (for any `t`); exp_2 has type `int`; exp_3 has type `t`
- Effect: the exp_2-th cell of exp_1 takes the value exp_3

A predefined type for references

Parametrized record type with a mutable field

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

The type of references (mutable indirection cells) containing a value of type 'a.

Creation

val ref : 'a -> 'a ref

Return a fresh reference containing the given value.

Access

val ! : 'a ref -> 'a

!r returns the current contents of reference r. [. . .]

Assignment

val := : 'a ref -> 'a -> unit

r := a stores the value of a in reference r. [. . .]

An other safe `read_int_bound`

```
let read_or_ignore r =
  try r := read_int () with _ -> ()
;;

let rec read_bound_int n =
  let m = ref (-1) in
    read_or_ignore m;
    while (!m < 0) || (n < !m) do
      print_string"Bad input, try again: ";
      read_or_ignore m
    done;
    !m
;;
```

Notice: the “pseudo procedure”

`val read_or_ignore :  int ref -> unit`

The `While` loop

Syntax: **`while`** exp_1 **`do`** exp_2 **`done`**

- exp_1 must have type `bool`
- exp_2 may have any type (warning if not `unit`)

Remark: no `Repeat` loop

Higher order iteration

Objective Caml is also fully functional

From module `Array`

val iter : ('a -> unit) -> 'a array -> unit

Array.iter f a applies function f in turn to all the elements of a. It is equivalent to f a.(0); f a.(1); ...; f a.(Array.length a - 1); ().

val iteri : (int -> 'a -> unit) -> 'a array -> unit

Same as Array.iter, but the function is applied to the index of the element as first argument, and the element itself as second argument.

```
let print_item i r =
  Printf.printf" %d: %s\n" i r.lang
;;

let get_choice () =
  print_endline "\n - Menu -" ;
  Array.iteri print_item choice_tab ;
  print_string "\nYour favorite: " ;
  read_bound_int max_choice
;;
```

Functional expressions

Anonymous functions

Syntax: `fun` *id* `->` *exp*

- Typing: `fun x -> e` has type $\texttt{T}_1$ `->` $\texttt{T}_2$ if `e` has type $\texttt{T}_2$, assuming `x` has type $\texttt{T}_1$
- Value: a *closure* (code + environment)

Syntactic sugar: `fun` id_1 id_2 `->` *exp*

stands for: `fun` id_1 `->` `fun` id_2 `->` *exp*

```
Array.iteri
  (fun i r -> Printf.printf" %d: %s\n" i r.lang)
  choice_tab
```

The truth on function's definitions

Syntactic sugar: `let` id_1 id_2 `=` *exp*

stands for: `let` id_1 `=` `fun` id_2 `->` *exp*

Syntactic sugar: `let rec` id_1 id_2 `=` *exp*

stands for: `let rec` id_1 `=` `fun` id_2 `->` *exp*

Functions are values

(For the "`fun`")

Mathematical definitions

- function composition: $(f \circ g)(x) = f(g(x))$
- function iteration (recursive): $\begin{cases} f^0 = id \\ f^{n+1} = f \circ f^n \end{cases}$

  where id is the identity function

Objective Caml definitions

```
# let fun_comp f g = fun x -> f (g x) ;;
val fun_comp :
   ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
# let rec fun_pow f n =
  if n = 0 then (fun x -> x)
  else fun_comp f (fun_pow f (n - 1))
;;
val fun_pow :
   ('a -> 'a) -> int -> 'a -> 'a = <fun>
```

Functions and data

Data structures may contain functions

```
type menu_data = { label : string;
                   action : unit -> unit } ;;
```

Possible handling choice functions:

- access to a (sub)menu

```
let  print_item = Printf.printf" %d: %s\n" ;;

let print_menu menu =
  print_endline "\n - Menu -" ;
  Array.iteri print_item menu ;
  print_string "\nYour choice: " ;
  let i = read_bound_int ((Array.length menu) - 1) in
    menu.(i).action ()
;;
```

- print a message

```
let print_msg s =
  Printf.printf "\n %s \"world\"\n\n" s
;;
```

Note how **`print_item`** id defined (apparently) without argument

Functions and data (continued)

Define the generic actions: `unit -> unit`

```
let menu_fun m = fun () -> print_menu m
;;

let msg_fun s = fun () -> print_msg s
;;
```

Define menu and submenu:

```
let western_menu =
  [| { label = "english";
         action = msg_fun "Hello" } ;
       { label = "french" ;
         action = msg_fun "Bonjour" } |]
;;

let main_menu =
  [| { label = "easter";
         action = msg_fun "Ni Hao" } ;
       { label = "western";
         action = menu_fun western_menu } |]
;;
```

Launch all

```
(menu_fun main_menu) ()
```

The `List` data structure

Parametrized recursive sum type

- A list of elements of type `t` is

1. either the *empty list* (noted `[]`)
2. either the list obtained by adding an element `x` (of type `t`) to an *already built list* `xs` (noted `x::xs`)

- Predefined type in Objective Caml

type 'a list = [] | :: of 'a * 'a list

The type of lists whose elements have type 'a.

Symbols `[]` and `::` are the *constructors* of type `list`

Syntactic sugar: $\boxed{\texttt{[}\ exp_1\ \texttt{;}\ \ldots\texttt{;}\ exp_n\ \texttt{]}}$

stands for: exp_1`::`$\ldots$ exp_n`::[]`

Predefined

val @ : 'a list -> 'a list -> 'a list

List concatenation.

Recursive programming with `list`'s

A non empty list has two components:
its *head* and its *tail*

- From The Objective Caml manual – module `List`

val hd : 'a list -> 'a

Return the first element of the given list. Raise Failure "hd" if the list is empty.

val tl : 'a list -> 'a list

Return the given list without its first element. Raise Failure "tl" if the list is empty.

Case analysis

Using toplevel

```
# let rec index a ns =
  if ns = [] then   (* ns is the empty list *)
    raise Not_found
  else              (* ns has form hd::tl *)
    if a = (List.hd ns) then 0
    else 1 + (index a (List.tl ns))
;;
val index : 'a -> 'a list -> int = <fun>
```

Remark: polymorphic function; available for `int list`, `string list`, `int list list`, etc.

The ML way

Pattern matching

```
# let rec index a xs =
  match xs with
      []
      -> raise Not_found
    | x::ys
      -> if a = x then 0 else 1 + (index a ys)
;;
val index : 'a -> 'a list -> int = <fun>
```

Two effects:

1. case analysis (`ns = []` or else)
2. access and bind to names the components
   - `x` for `(List.hd xs)`
   - `ys` for `(List.tl xs)`

- Alternative with the *any pattern* (char. `_`)

```
let rec index a xs =
  match xs with
      x::ys
      -> if a = x then 0 else 1 + (index a ys)
    | _ -> raise Not_found
```

More on pattern matching

What can be *matched* ?

Any value for which one can write *patterns* :)

What is a pattern ?

A "quasi-constant" expression which may contain variables and the *any* character `_`

Side condition: patterns are *linear*; variables can't occur several times in a given pattern.

What is *matching* ?

- intuitively: *pat* matches *expr* if the value of *exp* has the shape of *pat*
- (more) formaly *pat* matches *expr* if *pat* can be equalized to the *constant expression* which denotes *expr*'s value by replacing variables of *pat* with constants.

For instance: `[1]@[2]` (which is equal to `1::[2]`)

· matches `n::ns` with `n`=`1` and `ns`=`[2]`

· does not match `n::ns` because `[]`≠`[2]`

BNF for patterns

Extract of The Objective Caml manual

$$\begin{array}{rcl}
\textit{pattern} & ::= & \textit{value-name} \\
 & | & \_ \\
 & | & \textit{constant} \\
 & | & \textit{ncconstr-name}\ \textit{pattern} \\
 & | & \textit{pattern}\texttt{::}\textit{pattern} \\
 & | & \texttt{[}\ \textit{pattern}\ \{\texttt{;}\ \textit{pattern}\}\ \texttt{]} \\
 & | & \texttt{[|}\ \textit{pattern}\ \{\texttt{;}\ \textit{pattern}\}\ \texttt{|]} \\
 & \vdots & \ldots \\
\textit{constant} & ::= & \textit{int-literal} \\
 & | & \textit{float-literal} \\
 & | & \textit{char-literal} \\
 & | & \textit{string-literal} \\
 & | & \textit{bool-literal} \\
 & | & \textit{cconstr-name} \\
 & | & \texttt{()} \\
 & | & \texttt{[]} \\
\textit{value-name} & ::= & \textit{lowercase-ident} \\
\textit{cconstr-name} & ::= & \textit{capitalized-ident} \\
\textit{ncconstr-name} & ::= & \textit{capitalized-ident}
\end{array}$$

Once more on pattern matching

Syntax:

`match` *exp* `with`
  pat_1 `->` exp_1
| ⋮
| pat_n `->` exp_n

The `match` constructs are *expressions*

- Typing:
  - *exp*, pat_1, …and pat_n must have the same type
  - exp_1, …and exp_n must have the same type (may be differenr than the one of *exp*)
  - the whole type is the exp_i's one .
- Control: *exp*, then patterns are processed from 1 to n until the i-th matches the value of *exp* and then exp_i
- Value: the value of the first exp_i where variables of pat_i are given by the matching

Exception: `Match_failure` is raised if none of the patterns match *exp*.

⇒ The compiler warns you when this may happen

More advanced patterns usage

- Deep patterns: remove duplicates 0's of an `int list`

```
let rec rem_dup0's ns =
  match ns with
      0::0::ns -> rem_dup0's (0::ns)
    | 0::n::ns -> 0::n::(rem_dup0's ns)
    | n::ns -> n::(rem_dup0's ns)
    | _ -> []
```

- Uneeded values: keep the odd rank elements of a list

```
let rec skip xs =
  match xs with
      _::x::xs -> x::(skip xs)
    | _ -> []
```

Note that the last case plays for both pattern `[]` and `[x]`

- Matching two values: propositional arrow

```
let implies b1 b2 =
  match (b1, b2) with
      (false,_) -> true
    | _ -> b2
```

Matching two values = matching their *pair*

Note: the usage of `b2` in the last case

Sum types

Disjoint union

A type mixing `int` and `float`

```
type num =
    Inum of int
  | Fnum of float
;;
```

Notice: the capitalized initial; *mandatory*

Pattern matching facilities

The addition for `num`'s values

```
# let add_num x1 x2 =
  match x1, x2 with
      Inum n1, Inum n2 -> Inum (n1 + n2)
    | Inum n1, Fnum f2
        -> Fnum ((float_of_int n1) +. f2)
    | Fnum f1, Inum n2
        -> Fnum (f1 +. (float_of_int n2))
    | Fnum f1, Fnum f2 -> Fnum (f1 +. f2)
;;
val add_num : num -> num -> num = <fun>
```

Type conversion (`float_of_int`) embeded in *constructors*

Algebraic data types

Binary trees with parametrized labels

```
type 'a btree =
    Empty
  | Node of 'a btree * 'a * 'a btree
;;
```

Programming example: the list of labels

```
# let rec list_of_btree t =
  match t with
      Empty -> []
    | Node(t1, x, t2)
      -> (list_of_btree t1)@[x]@(list_of_btree t2)
;;
val list_of_btree : 'a btree -> 'a list = <fun>
```

Note: the polymorphic type

Recursion over trees

A tricky version of `list_of_btree`

```
let rec list_of_btree t =
  match t with
      Empty -> []
    | Node(Empty, x, t2) -> x::(list_of_btree t2)
    | Node(Node(t1, x1, t2), x2, t3)
        -> list_of_btree
             (Node(t1, x1, Node(t2, x2, t3)))
```

Insertion in a heap (balanced tree)

```
# let rec ins_heap x t =
  match t with
      Empty -> Node(Empty, x, Empty)
    | Node(t1, y, t2) ->
        if x < y then
          Node(t2, x, ins_heap y t1)
        else
          Node(t2, y, ins_heap x t1)
;;
val ins_heap : 'a -> 'a btree -> 'a btree = <fun>
```

Note: the polymorphic type

⇒ the test operator `<` is *polymorphic*

Functional model

λ-calculus

A. Church 1932: theory of computable functions

- Three basics constructs

1. atoms (variables or constantes)
2. application: $(t\ u)$
3. functional abstraction: $\lambda x.t$

- Abstracted variable's scope: binding

x is *bound* in $\lambda x.t$

A variable not bound (in a term) is *free*

- Renaming bound variables: α-conversion

Intuitively: $\lambda x.x + 1$ and $\lambda y.y + 1$ are the same function

Fact: it is always possible to rename a bound variable with an unused name

Computation model

- Substitution (of a term to free variables): $t[u/x]$

By case on t

- $x[u/x] = u$
- $y[u/x] = y$, if x and y are distinct variables
- $(t_1\ t_2)[u/x] = (t_1[u/x]\ t_2[u/x])$
- $(\lambda x.t)[u/x] = \lambda x.t$
- $(\lambda y.t)[u/x] = \lambda z.t[z/y][u/x]$, if x and y are distinct variables and z not free in u.

- A distinguished application: the *redex*

$$(\lambda x.t\ u)$$

- A model of computation: β-reduction

substitution of *formal* parameter by *actual* argument
$(\lambda x.t\ u)$ evaluates to $t[u/x]$

- Normal form: a *value* is reached when all redexes has been reduced

- $\lambda f.\lambda x.(f\ (f\ x))$ is in normal form
- $(\lambda f.\lambda x.(f\ x)\ \ \lambda y.y)$ is not

Data encoding

Booleans

- $\mathsf{true} = \lambda x.\lambda y.x$
- $\mathsf{false} = \lambda x.\lambda y.y$
- $\mathsf{if} = \lambda x.\lambda y.\lambda z.(x\ y\ z)$
- $\mathsf{and} = \lambda x.\lambda y.(\mathsf{if}\ x\ y\ x)$
- etc.

Notation: $(t_1\ t_2\ t_3)$ short and for $((t_1\ t_2)\ t_3)$.

Computing: let A and B be to termes:

$$\begin{aligned}
(\mathsf{if}\ \mathsf{true}\ A\ B) &= (\lambda x.\lambda y.\lambda z.(x\ y\ z)\ \mathsf{true}\ A\ B) \\
&\quad \text{- by definition -} \\
&= (\mathsf{true}\ A\ B) \\
&\quad \text{- after three reductions -} \\
&= (\lambda x.\lambda y.x\ A\ B) \\
&\quad \text{- by definition -} \\
&= A \\
&\quad \text{- after two reductions -}
\end{aligned}$$

Remark: $(\mathsf{if}\ t\ A\ B)$ where t is a boolean could be simpler be written $(t\ A\ B)$

ML’s evaluation model

- Weak head normal form: don’t reduce “under” λ’s

i.e. $\lambda x.t$ is in whnf what ever t can be

- Reduction strategy: *call-by-value*

i.e. reduce the argument before passing it to the function

Reduction rule:

$$\text{if } t \text{ reduces to } \lambda x.t' \text{ and } u \text{ reduces to } u' \text{ then}$$
$$(t\ u) \text{ reduces to } t'[u'/x]$$

Significant for side effects

```
# let x = print_endline "WHEN EVALUATED" ;;
WHEN EVALUATED
val x : unit = ()
# let x = fun () -> print_endline "WHEN APPLIED" ;;
val x : unit -> unit = <fun>
# x () ;;
WHEN APPLIED
- : unit = ()
# let x = fun y ->
    print_endline " THEN THE FUNCTION’S BODY" ;;
val x : ’a -> unit = <fun>
# x (print_string "THE ARGUMENT FIRST,") ;;
THE ARGUMENT FIRST, THEN THE FUNCTION’S BODY
- : unit = ()
```

Environment and closure

Delayed substitution

Mutually recursive definition

- *Environment*: pairs of variable and closure

$$E = (x_1, v_1); \ldots ; (x_n, v_n)$$

- *Closure*: pairs of term and environment

$$\langle t, E \rangle$$

Well founded: empty environment

Computation rules $\quad E \vdash t \Rightarrow v$

$$\begin{array}{ll} \mathit{Variable}: & \ldots ; (x, v); \ldots \vdash x \Rightarrow v \\ \\ \mathit{Abstraction}: & E \vdash \lambda x.t \Rightarrow \langle \lambda x.t, E \rangle \\ \\ \mathit{Application}: & \left\{ \begin{array}{l} \text{if } \ E \vdash t \Rightarrow \langle \lambda x.t', E' \rangle \\ \quad E \vdash u \Rightarrow v_1 \\ \quad (x, v_1); E \vdash t' \Rightarrow v_2 \\ \text{then } E \vdash (t\ u) \Rightarrow v_2 \end{array} \right. \end{array}$$

Closures are values

Abstract machine

Implements
call-by-value reduction to weak head normal form

A stack to store

1. closures to record in the environment, noted $\langle u, e\rangle$
2. closures to evaluate now, noted (t, e)

Transitions

Term	*Env.*	*Stack*
x	$\ldots (x, \langle t, e\rangle) \ldots$	s
t	e	s
$(t\ u)$	e	s
u	e	$(t, e) : s$
$\lambda x.t$	e	$\langle u, e'\rangle : s$
t	$(x, \langle u, e'\rangle) : e$	s
$\lambda x.u$	e	$(t, e') : s$
t	e'	$\langle \lambda x.u, e\rangle : s$

Control and reduction strategy

In ($\mathsf{if}\ t\ u_1\ u_2$) we *don't want* to evaluate u_1 and u_2 before.

$\Rightarrow$ *call-by-name*: "reduce the function first"

$\Rightarrow$ a different kind of application, noted $[t\ u]$

`if` t `then` A `else` B is translated as $[[t\ u_1]\ u_2]$

New transition

Term	*Env.*	*Stack*
$[t\ u]$	e	s
t	e	$\langle u, e\rangle : s$

Computing the conditional

Assume that e_0 containts the boolean encodings
ε denotes the empty stack

Term	*Env.*	*Stack*
$[[\mathsf{true}\ u_1]\ u_2]$	e_0	ε
$[\mathsf{true}\ u_1]$	e_0	$\langle u_2, e_0\rangle : \varepsilon$
true	e_0	$\langle u_1, e_0\rangle : \langle u_2, e_0\rangle : \varepsilon$
$\lambda x.\lambda y.x$	e_0	$\langle u_1, e_0\rangle : \langle u_2, e_0\rangle : \varepsilon$
$\lambda y.x$	$(x, \langle u_1, e_0\rangle) : e_0$	$\langle u_2, e_0\rangle : \varepsilon$
x	$(y, \langle u_2, e_0\rangle) : (x, \langle u_1, e_0\rangle) : e_0$	ε
u_1	e_0	ε

Typed λ-calculus

(simply)

Simple type expressions

1. atoms (variables or constants)
2. arrow type: $\tau_1 \rightarrow \tau_2$

Typing rules

- Typing environment: $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$
- Typing judgement $\Gamma \vdash t : \tau$
- Rules

$$\mathit{Atoms}: \quad \frac{}{x : \tau, \Gamma \vdash x : \tau}$$

$$\mathit{Abstraction}: \quad \frac{x : \tau_1, \Gamma \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \rightarrow \tau_2}$$

$$\mathit{Application}: \quad \frac{\Gamma \vdash \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash u : \tau_1}{\Gamma \vdash (t\ u) : \tau_2}$$

Polymorphism

Only defined variables have generalized type

Remark:

$$\texttt{let } x \texttt{ = } u \texttt{ ;; } t \qquad \approx \qquad \texttt{let } x \texttt{ = } u \texttt{ in } t$$

Type schemas

$\forall \alpha_1 \ldots \alpha_n . \tau$ where $\alpha_1 \ldots \alpha_n$ are variables of τ

Rules

Instance:

$$\frac{}{\Gamma, x : \forall \alpha_1 \ldots \alpha_n . \tau \vdash x : \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]}$$

Generalization:

$$\frac{\Gamma \vdash u : \tau_1 \qquad \Gamma, x : \forall \alpha_1 \ldots \alpha_n . \tau_1 \vdash t : \tau_2}{\Gamma \vdash \texttt{let } x \texttt{ = } u \texttt{ in } t : \tau_2}$$

$$\frac{\Gamma, x : \tau_1 \vdash u : \tau_1 \qquad \Gamma, x : \forall \alpha_1 \ldots \alpha_n . \tau_1 \vdash t : \tau_2}{\Gamma \vdash \texttt{let rec } x \texttt{ = } u \texttt{ in } t : \tau_2}$$

Type inference

Type expressions:

- variables: `'a`, `'b`, etc.
- constants: `unit`, `bool`, `int`, etc.
- type expression constructors: `__->__`, `__*__`, `__list`, etc.

Basic contexts: Γ_0

`true`:`bool`, `false`:`bool`, `not`:`bool -> bool`, . . . ,
. . . , `-1`:`int`, `0`:`int`, `1`:`int`, `+`:`int -> int -> int`, . . . ,
`[]`:`'a list`, `::`:`'a -> 'a list -> 'a list`, . . . ,
`fst`:`'a * 'b -> 'a`, etc.

Slove the problem:

> *“given a typing context* Γ *and a term* t*, find a type expression* τ *such that* $\Gamma_0 \cup \Gamma \vdash t : \tau$ *is derivable”*

Typing is *syntax directed*

$\Rightarrow$ reduce to first order unification

$\Rightarrow$ *decidable*

$\Rightarrow$ most general type

Closure and mutable values

- Reference embeded in a closure's *environment*:
the reference (the pointer) remains the same; its value changes

```
# let cpt =
  let c = ref 0 in
    fun () -> incr c; !c
;;
val cpt : unit -> int = <fun>
# cpt() ;;
- : int = 1
# cpt() ;;
- : int = 2
```

This does not work …

```
# let bad_cpt () =
  let c = ref 0 in
    incr c; !c
;;
val bad_cpt : unit -> int = <fun>
# bad_cpt() ;;
- : int = 1
# bad_cpt() ;;
- : int = 1
```

…because `let c` is in the *code* of `bad_cpt`; not in its *environemnt*

Weak type variable

- Mutable values can't have polymorphic type

⇒ it would break type safety

Assume it were possible:

```
let x = ref [] in
  x := 1::!x;
  x := true::!x
```

would cause problems !

- Solution: temporary polymorphic type with *weak type variable* `'_a`

```
# let x = ref [] ;;
val x : '_a list ref = {contents = []}
# x := 1::!x ;;
- : unit = ()
# x ;;
- : int list ref = {contents = [1]}
```

After (first) assignment, **x** has monomorphic type `int list`

Modules in Objective Caml

Sofware structuration

- *physical* structuration
  ⇒ compilation units (files)
- *logical* structuration
  ⇒ explicit syntax for modules

Module's name: capitalized identifier (`List`, etc.)

Two components:

- a *signature*: list of declarations
  ⇒ names of units exported by the module:
  types, exceptions, values, modules, etc.
- an *implementation*: sequence of definitions
  all that's needed

Acces to modules units:

- explicit access: fully qualififed names `M.x`
- implicit access: directive `open M`

Linking

- compiler: `ocamlc m.cmo p.ml`
- toplevel: `#load "m.cmo";; #use "p.ml" ;;`

Signature and structure

Structure

Syntax: `module` *Id* `= struct` … `end`

Signature

Syntax: **`module type`** *Id* **`= sig`** … **`end`**

Inferred signature

```
# module M1 =
  struct
    type t = int * int * int
    let make d m y = d,m,y
  end
;;
module M1 :
  sig type t = int * int * int
      val make : 'a -> 'b -> 'c -> 'a * 'b * 'c end
# let d = M1.make 1 1 1970 ;;
val d : int * int * int = 1, 1, 1970
```

- The most general type is inferred

Signature and structure (continued)

Constrained signature

```
# module type T2 =
  sig
    type t = int * int * int
    val make : int -> int -> int -> t
  end

 module M2 = (M1:T2)
;;
module M2 : T2
```

- `make` has the intended type

```
# M2.make;;
- : int -> int -> int -> M2.t = <fun>
```

- type `M2.t` is “open”

```
# let d = M2.make 1 1 1970 ;;
val d : M2.t = 1, 1, 1970
# match d with x,y,z -> z ;;
- : int = 1970
```

Signature and structure (continued)

Type abstraction

```
# module type T3 =
  sig
    type t
    val make : int -> int -> int -> t
  end

  module M3 = (M1:T3)
;;
module M3 : T3
# M3.make ;;
- : int -> int -> int -> M3.t = <fun>
```

- the definition of values of type `M3.t` is *unreachable*

```
# let d = M3.make 1 1 1970 ;;
val d : M3.t = <abstr>
# match d with x,y,z -> z ;;
This pattern matches values of type 'a * 'b * 'c
but is here used to match values of type M3.t
```

Remark: we used three times the *same structure* (`M1`) to create three *differents modules*, `M1`, `M2` and `M3`

Interface and implementation

Compilation units

- Implementation file `m1.ml`, will give module `M1`

```
type t = int * int * int
let make d m y = d,m,y
```

- Interface file `m2.mli`, analogous to signature `T2`

```
type t = int * int * int
val make : int -> int -> int -> t
```

To get module `M2`: copy `m1.ml` into file `m2.ml` and compile both `m2.mli` and `m2.ml`

- Interface file `m3.mli`, analogous to signature `T3`

```
type t
val make : int -> int -> int -> t
```

…copy `m1.ml` into file `m3.ml` and compile

Restricted access

Aim: give two differents access to a *shared ressource*

```
module Cpt =
  struct
    let x = ref 0
    let reset () = x := 0
    let next () = incr x; !x
  end
```

- the "super-user" abilities: can reset the counter

```
module type CPT_ADM =
  sig
    val reset : unit -> unit
    val next : unit -> int
  end

module CptAdm = (Cpt:CPT_ADM)
```

- the user: can only get the next value

```
module type CPT_USR =
  sig
    val next : unit -> int
  end

module CptUsr = (Cpt:CPT_USR)
```

Genericity

A module may depend on a parameter

- Assume a (undetermined) type with conversion functions

```
module type ENCODING =
  sig
    type t
    val to_string : t -> string
    val of_string : string -> t
  end ;;
```

We only need here a *signature*

- Some generic (in|out)put module

```
module StdIO (V:ENCODING) =
  struct
    let writeln x = print_endline (V.to_string x)
    let readln () = V.of_string (read_line())
  end ;;
```

Aim: build a module for a given concrete type

Genericity (continued)

- Lists of integers and strings

```
module IntList =
  struct
    type t = int list
    let to_string ns =
      String.concat " " (List.map string_of_int ns)
    let of_string s =
      List.map int_of_string (Xstring.split ' ' s)
  end
```

- Build a module defining (in|out)put functions for `int list`'s

```
module IntListStdIO = StdIO (IntList)
```

Remark: while type `t` is abstract in signature `ENCODING`, it is not in module `IntListStdIO`

```
# IntListStdIO.writeln;;
- : IntList.t -> unit = <fun>
# IntListStdIO.writeln [1;2;3] ;;
1 2 3
- : unit = ()
```

Functors definition and usage

Functional model

Abstraction

Syntax:

$\boxed{\texttt{module}\ Id_1\ \texttt{(}\ Id_2\ \texttt{:}\ SIG\ \texttt{)}\ \texttt{=}\ \texttt{struct}\ \ldots\ \texttt{end}}$

where

- Id_1 is the name of the defined functor
- Id_2 is the module taken as (formal) parameter
- SIG is the signature of Id_2 (a name or a construct `sig`...`end`): *MANDATORY*

Application

Syntax: $\boxed{\texttt{module}\ Id_1 = Id_2\ \texttt{(}\ STRUC\ \texttt{)}}$

where

- Id_1 is the name of the defined module
- Id_2 is a functor
- $STRUC$ is a module (a name or a construct `struct`...`end` or ... a functor application !)

More genericity

- Signature of I/O's basics

```
module type IOSIG =
  sig
    val writeln : string -> unit
    val readln : unit -> string
  end
;;
```

- A signature embeding (in|out)channel

```
module type IOChanPair =
  sig
    val ic : in_channel
    val oc : out_channel
  end
;;
```

- A functor preparing (in|out)put on channels

```
module ChanIO (Chan:IOChanPair) =
  struct
    let writeln s =
      output_string Chan.oc s;
      output_char Chan.oc '\n'
    let readln () =
      input_line Chan.ic
  end
;;
```

More genericity (continued)

Generic (in|out)put

Functors may have more than one argument

```
module GenIO (V:ENCODING) (IO:IOSIG) =
  struct
    let writeln x = IO.writeln (V.to_string x)
    let readln () = V.of_string (IO.readln ())
  end
;;
```

- Lets do it on std(in|out)

```
module StdIO =
  ChanIO(struct let oc=stdout let ic=stdin end)
;;

module IntListStdIO = GenIO (IntList) (StdIO)
;;
```

```
# open IntListStdIO ;;
# writeln [1;2;3] ;;
1 2 3
- : unit = ()
# readln() ;;
1 2 3
- : IntList.t = [1; 2; 3]
```

Modules and type sharing

Assume a module to compute some “digest” of a data

```
module type DIGEST =
  sig
    type t
    val to_int : t -> int
  end
```

Aim: define a new encoding merging the one of previous encoding and the cerificate given by “digest”

This will fail:

```
# module NewEncoding (E:ENCODING) (D:DIGEST) =
  struct
    let to_string x =
      (string_of_int (D.to_int x)) ^ (E.to_string x)
    (* ... *)
  end
;;
This expression has type D.t but is here used with
type E.t
```

As abstracted, types `D.t` and `E.t` are differents.

Modules and type sharing (continued)

Types constraints

We have to state explicitely the required the type equality

```
# module NewEncoding (E:ENCODING)
                     (D:DIGEST with type t=E.t) =
  struct
    let to_string x =
      (string_of_int (D.to_int x)) ^ (E.to_string x)
    (* ... *)
  end
;;
module NewEncoding :
  functor (E : ENCODING) ->
  functor (D : sig type t = E.t
               val to_int : t -> int end) ->
      sig val to_string : D.t -> string end
```

(with `functor` ≈ `fun`)

The type checker has created the right dependent signature for module parameter `D`

Modules and type sharing (continued)

Type constraint is checked on concrete types when functor is applied

Bad usage:

```
# module BoolListDigest =
  struct
    type t = bool list
    let to_int = List.length
  end ;;

# module BoolListNewEncoding =
    NewEncoding (IntList) (BoolListDigest) ;;
Signature mismatch:
Modules do not match:
  sig type t = bool list
      val to_int : 'a list -> int end
is not included in
  sig type t = IntList.t
      val to_int : t -> int end
Type declarations do not match:
  type t = bool list
is not included in
  type t = IntList.t
```

Objective Caml Libraries

All language's predifined and builtins belongs to modules

- Basics (always "open" and linked): module `Pervasive`
- Standard (automaticaly linked): 33 modules
  mainly data structures
  (`Array`, `List`, `String`, `Stack`, etc.)
  utilities, etc.
- "other libs": 9 modules
  `Unix` contains networking API
  `Threads` concurent programming
  etc.

All is well documented

Objects in Objective Caml

Foreword

Class: specification, definition of a set of objects
(see below)

Object: element or *instance* of a class
(see above)

Inheritance: relation between classes, extension or specialisation

Field or attribute: data belonging to an object

Method: action, function belonging to objects

Message passing: activation of a method by the revieving object

Objects in Objective Caml

Class declaration

Syntax:

```
Class id id1 ... idn =
   object
   ...
     val id = expr
   ...
     val mutable id = expr
   ...
     method id id1 ... idn = expr
   ...
   end
```

- class declaration header (`Class`)
  - *id* is the name of the class
  - $id_1 \ldots id_n$ are the optional parameters needed when instances are created
- field variables declaration (`val`)
  a field's value may be `mutable`
- methods declarations (`method`)
  like functions

Class and type

Still statically infered

```
# class cpt =
  object
    val mutable c = 0
    method incr () = c <- c+1
    method reset () = c <- 0
    method get () = c
  end
;;
class cpt :
  object
    method get : unit -> int
    method incr : unit -> unit
    method reset : unit -> unit
    val mutable c : int
  end
```

Type of instances will be

- named as the class (`cpt`)
- defined as method's names *together* with their type

Note: although they were displayed, variables are ignored in the type

Class and type (continued)

Class with parameters

Counters with initial and step values

```
class cpt c0 s =
  object
    val mutable c = c0
    method incr () = c <- c+s
    method reset () = c <- c0
    method get () = c
  end
;;
class cpt :
  int ->
  int ->
  object
    method get : unit -> int
    method incr : unit -> unit
    method reset : unit -> unit
    val mutable c : int
  end
```

The functional type `int -> int -> object ...end` stands for the type of the *instance constructor*

The class's type itself is still `object get : unit -> int ...end`

Instances and their usage

Creating an instance

Syntax: $\boxed{\texttt{new}\ id\ exp_1 \ldots exp_n}$

- *id* is the name of the class
- $exp_1 \ldots exp_n$ are the initial values of class parameters

```
# let c = new cpt 0 1 ;;
val c : cpt = <obj>
```

Message passing: access to a method

Syntax: $\boxed{exp_1\texttt{\#}id}$

```
# c#get ;;
- : unit -> int = <fun>
```

Applying method to its arguments

```
# c#get() ;;
- : int = 0
# c#incr() ; c#get() ;;
- : int = 1
```

Warning: variables are not accessible

```
# c#c ;;
This expression has type cpt
It has no method c
```

Inheritance

Syntax: `inherit` *id* $exp_1 \ldots exp_n$

Adding methods

```
class cpt1 c0 s =
  object
    inherit cpt c0 s
    method to_string =
      Printf.sprintf "< init=%d; step=%d; value=%d >"
                     c0 s c
  end
```

Notes: a method may have no parameter;
inherited variables are usuable

- Inherited methods are available

```
# let c = new cpt1 0 1 ;;
val c : cpt1 = <obj>
# c#incr(); c#get() ;;
- : int = 1
```

and there is new one

```
# c#to_string ;;
- : string = "< init=0; step=1; value=1 >"
```

Self references

Method can't be used without an object
⇒ generic name for *any* instance

Self-name must be *declared*

Syntax: `object ( id )`

```
class gensym =
  object(self)
    inherit cpt 0 1
    val txt = "X"
    method sym = txt^(string_of_int c)
    method next = self#incr(); self#sym
  end
```

Note: the name "`self`" is not mandatory but *standard*

Initializer

Execute some code at creation time

Syntax: `initializer` *exp*

```
# class verbose_gensym =
  object(self)
    inherit gensym
    initializer
      Printf.printf
        "Hello, I'm a new gensym for %s symbols\n" txt;
      Printf.printf
        "my initial value is %s\n" self#sym
  end
```

...

```
# let vs = new verbose_gensym ;;
Hello, I'm a new gensym for X symbols
my initial value is X0
val vs : verbose_gensym = <obj>
```

Initializers may use parameters, variables and methods definined by the class

Redefining

One can redefine : variables and methods

```
# class gensym' =
  object(self)
    inherit gensym
    val txt = "Y"
    method sym = txt^(string_of_int c)
  end ;;
```

...

```
# let s = new gensym ;;
val s : gensym = <obj>
# s#sym ;;
- : string = "X0"
# let s' = new gensym' ;;
val s' : gensym' = <obj>
# s'#sym ;;
- : string = "Y0"
```

Late binding

The code (of methods) to execute is choosen at *runtime*

```
# s#next ;;
- : string = "X1"
# s'#next ;;
- : string = "Y1"
```

Redefining (continued)

Beware: don't change types

```
class wrong_gensym =
  object(self)
    inherit cpt 0 1
    val txt = "X"
    method get () = txt^(string_of_int c)
  end
;;
This expression has type string but is here used
with type int
```

Beware: variables are *static*

```
# class wrong_gensym' =
  object(self)
    inherit gensym
    val txt = "Y"
  end
```

...

```
# (new wrong_gensym')#sym ;;
- : string = "X0"
```

Redefinition and self reference

Using former method's value to (re)define the new one

Syntax: `inherit` … `as` *id*

```
class gensym1 =
  object(self)
    inherit cpt1 0 1 as super
    val txt = "X"
    method sym = txt^(string_of_int c)
    method next = self#incr(); self#sym
    method to_string =
      Printf.sprintf "[ txt=\"%s\" %s ]"
                     txt super#to_string
  end
```

Note: the name "`super`" is not mandatory but *standard*

Multiple inheritance

Merging several classes in a new (sub)one

Assume

```
class cpt =
```

...

```
class mksym =
  object
    val txt = "X"
    method sym_of_num n = txt^(string_of_int n)
  end
```

- Define by merging

```
class gensym2 =
  object(self)
    inherit cpt 0 1
    inherit mksym
    method next =
      self#incr(); self#sym_of_num c
  end
```

Multiple inheritance and overloading

Assume

```
class cpt1 =
...
```

```
class mksym1 =
  object
    val txt = "X"
    method sym_of_num n = txt^(string_of_int n)
    method to_string =
      Printf.sprintf "< txt=\"%s\" >" txt
  end
```

- Must (and can) discriminate between `to_string`'s

```
class gensym3 =
  object(self)
    inherit cpt1 0 1 as super1
    inherit mksym1 as super2
    method next =
      self#incr(); self#sym_of_num c
     method to_string =
      Printf.sprintf "< %s %s >"
                     super1#to_string
                     super2#to_string
  end
```

Abstract classes

Specify a required but delayed method definition

Syntax: `class virtual` *id* . . .

Syntax: `method virtual` *id* `:` *ty*

A generic class for printable objects

```
class virtual printable =
  object(self)
    method print = print_string self#to_string
    method virtual to_string : string
  end
```

- An abstract class can't have instances

```
# new printable ;;
One cannot create instances of the virtual class
printable
```

Abstract classes (continued)

- Becoming abstract by inheritance

```
class virtual printable_gensym =
  object
    inherit gensym
    inherit printable
  end
```

- Becoming *concrete* by defining

```
class gensym4 =
  object
    inherit printable_gensym
    method to_string =
      Printf.sprintf"< text=%s value=%d>" txt c
  end
;;
```

- Becoming *concrete* by inheritance

```
class gensym5 =
  object
    inherit printable
    inherit gensym3
  end
```

Parametrized classes

Class may depend on a *type parameter*
⇒ polymorphism

Syntax: $\boxed{\texttt{class ['}\mathit{id}_1\texttt{,}\ \ldots\texttt{, '}\mathit{id}_n\texttt{]}\ \mathit{id}\ \ldots}$

where 'id_1, …, 'id_n are type variables

A generic class for stacks

```
class  ['a] stack =
  object
    val mutable s = ([] : 'a list)
    method push x = s <- x::s
    method pop =
      match s with
          [] -> failwith "Empty stack"
        | x::s' -> (s <- s'; x)
  end
```

Recall: classes define types, so type parameters must be declared

Type constraint

Syntax: $\boxed{\mathit{exp}\ :\ \mathit{ty}}$

Forces the compiler to check that *exp* do have type *ty*

Needed in `stack` definition

⇒ forces elements of `s` to belong to
THE declared type parameter `'a`

Remark: type constraint may be anywhere else, but somewhere

```
class  ['a] stack =
  object
    val mutable s = []
    method push (x : 'a) = s <- x::s
    method pop =
      match s with
          [] -> failwith "Empty stack"
        | x::s' -> (s <- s'; x)
  end
```

Type error

Caution: strange error message with type variables names

```
# class  ['elt] stack =
  object
    val mutable s = []
    method push x = s <- x::s
    method pop =
      match s with
          [] -> failwith "Empty stack"
        | x::s' -> (s <- s'; x)
  end ;;
Some type variables are unbound in this type:
  class ['a] stack :
    object
      method pop : 'b
      method push : 'b -> unit
      val mutable s : 'b list
    end
The method pop has type 'a where 'a is unbound
```

The `'a` in “`pop has type 'a`” must be read as `'b` !

Note also: my `'elt` has been replaced. Type variables are *bound variables*; their name may change.

Parametrized class usage

- Parametrized type ⇒ weak type variables

```
# let s = new stack;;
val s : '_a stack = <obj>
# s#push 1 ;;
- : unit = ()
# s ;;
- : int stack = <obj>
```

- Inheritance with type instanciation

Syntax: `inherit [` *ty* `]` *id*

```
# class  int_stack =
  object
    inherit [int] stack
    method add =
      match s with
  n1::n2::s' -> s <- (n1+n2)::s'
| _ -> ()
  end ;;
class int_stack :
  object
    method add : unit
    method pop : int
    method push : int -> unit
    val mutable s : int list
  end
```

Parametrized class usage (continued)

- Polymorphic inheritance

```
# class  ['a] stack1 =
  object
    inherit ['a] stack
    method app f =
      match s with
          x1::x2::s' -> s <- (f x1 x2)::s'
        | _ -> ()
  end ;;
class ['a] stack1 :
  object
    method app : ('a -> 'a -> 'a) -> unit
    method pop : 'a
    method push : 'a -> unit
    val mutable s : 'a list
  end
```

⇒ weak type

```
# let s = new stack1 ;;
val s : '_a stack1 = <obj>
# s#app ;;
- : ('_a -> '_a -> '_a) -> unit = <fun>
# s#push "Hello ";;
- : unit = ()
# s#app ;;
- : (string -> string -> string) -> unit = <fun>
```

Class define type

- class name used as type name

```
class int_stack_stack =
  object
    inherit [int_stack] stack
  end
```

- class name used as *parametrized* type name

```
class ['a] stack_stack =
  object
    inherit ['a stack] stack
  end
```

Note: the lack of `[ ]` around `'a` when `stack` is used as a *type name*

- type/class is not value/instance

```
class  printable_stack =
  object
    inherit [printable] stack
  end
```

Only instances of *concrete* `printable`'s would be pushed

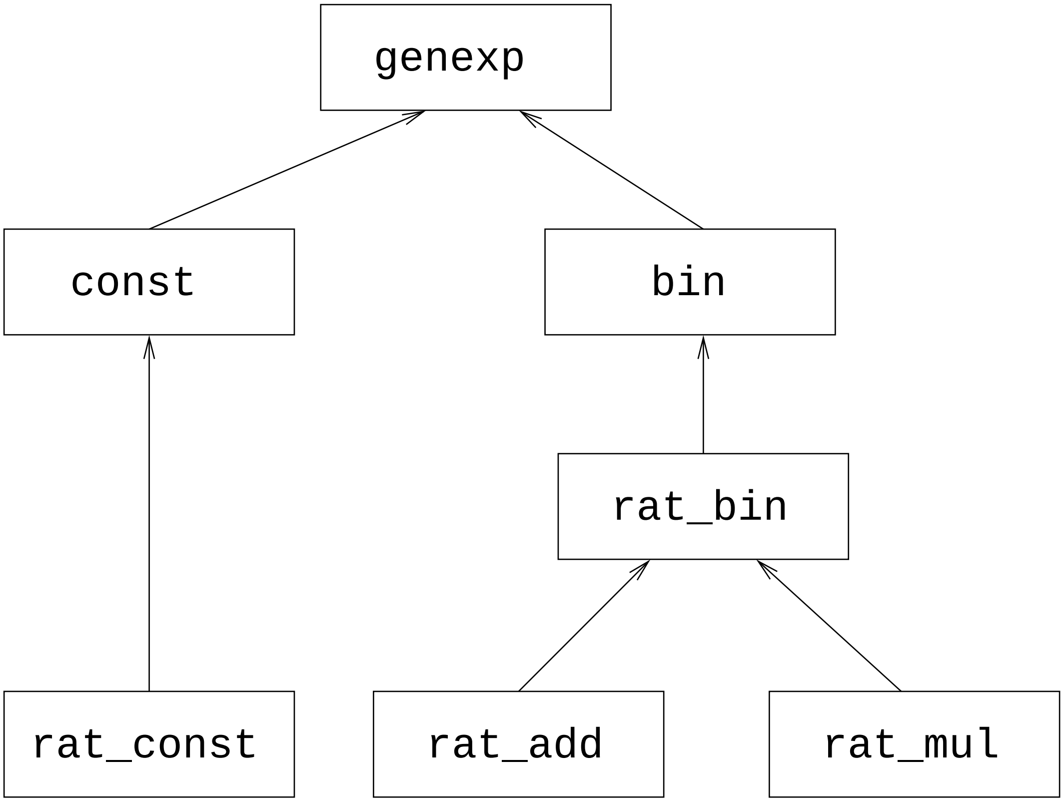
⇒ `printable_stack` *is not* itself abstract

Design example

From generic expressions

Featuring:

- an `eval` function
- a `print` method



Inheritance tree

Rational arithmetics

Parametrized abstract classes

- Common features as abstract methods

```
class virtual ['a] genexp =
  object
    method virtual eval : 'a
    method virtual print : unit
  end
```

- Constants: immediate value

```
class virtual ['a] const x =
  object
    inherit ['a] genexp
    method eval = x
  end
```

- Binary operations: recursive value and display

```
class virtual ['a] bin e1 e2  =
  object(self)
    inherit ['a] genexp
    method virtual printop : unit
    method virtual op : 'a -> 'a -> 'a
    method eval = self#op e1#eval e2#eval
    method print =
      print_char '(';
      e1#print; self#printop; e2#print;
      print_char ')'
  end
```

Full determination

- Rational constants

```
class rat_const x y =
  object
    inherit [int * int] const (x,y)
    initializer
      if y=0 then raise Division_by_zero
    method print =
      Printf.printf"[%d/%d]" x y
  end
```

Note: the use of `initializer`

- Usage

```
# let r = new rat_const 3 4 ;;
val r : rat_const = <obj>
# r#print ;;
[3/4]- : unit = ()
# r#eval ;;
- : int * int = 3, 4
# new rat_const 1 0 ;;
Exception: Division_by_zero.
```

(Re)abstract binary expressions

Factorizing common code for printing and evaluating rationals

- operation symbol as `char`
- reduce after computing

```
class virtual rat_bin e1 e2 =
  object(self)
    inherit [int * int] bin e1 e2
    method virtual sym : char
    method printop = print_char self#sym
    method virtual rawop :
      int * int -> int * int -> int * int
    method op r1 r2 =
      let (x, y) = self#rawop r1 r2 in
      let d = gcd x y in
        (x/d, y/d)
  end
```

with

```
let rec gcd m n =
  if n=0 then m
  else gcd n (m mod n)
```

Concrete binary expressions

- Addition

```
class rat_add e1 e2 =
  object
    inherit rat_bin e1 e2
    method sym = '+'
    method rawop (x1,y1) (x2,y2) =
      (x1*y2 + x2*y1, y1*y2)
  end
```

- Mutiplication

```
class rat_mul e1 e2 =
  object
    inherit rat_bin e1 e2
    method sym = '*'
    method rawop (x1,y1) (x2,y2) =
      (x1*x2, y1*y2)
  end
```

- Usage

```
# let rcst x y = new rat_const x y in
  let rplus x y = new rat_add x y in
  let rmult x y = new rat_mul x y in
  let e =
    (rmult (rplus (rcst 1 3) (rcst 3 5)) (rcst 1 2))
  in
    e#print; e#eval ;;
(([1/3]+[3/5])*[1/2])- : int * int = 7, 15
```

What about types ?

- Apparently strange

```
# let r_1_2 = new rat_const 1 2 ;;
val r_1_2 : rat_const = <obj>
# let e1 = new rat_add r_1_2 r_1_2 ;;
val e1 : rat_add = <obj>
# let e2 = new rat_mul e1 r_1_2 ;;
val e2 : rat_mul = <obj>
```

the type depends on the main operator (instance constructor)

⇒ three different types for expressions !

Types of instance constructors

- Constants

```
# new rat_const;;
- : int -> int -> rat_const = <fun>
```

looks good

- Addition

```
# new rat_add ;;
- : < eval : int * int; print : 'a; .. > ->
    < eval : int * int; print : 'b; .. > -> rat_add
= <fun>
```

what does it mean ?

Classes and types: more

Recall: object's type = its methods and their types

Type inference ⇒ most general type
⇒ *open* object-type

Syntax: $\boxed{\texttt{<}\ id_1\ \texttt{:}\ ty_1\texttt{;}\ \ldots\texttt{;}\ id_n\ \texttt{:}\ ty_n\texttt{;}\ \texttt{..>}}$

where the last `..` is a *reserved symbol*

Meaning:

(of `< eval : int * int; print : 'a; .. >`)

(the type of) an object with

- method `eval` of type `int * int`
- method `print` of undeterminated type `'a`
- and may be some other undeterminated methods `..`

The "`..`" at the end is some kind of *type variable*

⇒ possible object extension

Object-types compatibility

Simplified example

```
# class c0 =
  object method m = print_string "Hello" end
;;
class c0 :
   object method m : unit end
# class c1 o =
  object method m = o#m; print_string " world" end
;;
class c1 :
  < m : 'a; .. > -> object method m : unit end
# let o = new c1 (new c0) ;;
val o : c1 = <obj>
# o#m ;;
Hello world- : unit = ()
```

Question: why is `new c1 (new c0)` well typed ?

Answer: because

1. `new c1` expects a `< m : 'a; .. >`
2. `new c0` provides a `< m : unit >`
3. `'a` can take value `unit`
4. `..` can take value *nothing else*

Subtyping relation

Wider object-types compatibility

Syntax: (*exp* `:>` *oty*)

where *exp* is an object and *oty* an object-type

Meaning: (of `(o:>c)`)

- the object `o` provides *all methods* of `c`
- their type in `o` are *subtypes* of the one specified by `c`

Usage:

```
# r_1_2 ;;
- : rat_const = <obj>
# e1 ;;
- : rat_add = <obj>
# type rat_exp = (int*int) genexp ;;
type rat_exp = (int * int) genexp
# [ (r_1_2:>rat_exp); (e1:>rat_exp) ] ;;
- : rat_exp list = [<obj>; <obj>]
```

Note: how we used object-type name `genexp` to define a new one

More about object-types

```
# e2 ;;
- : rat_mul = <obj>
# [ e1; e2 ] ;;
- : rat_add list = [<obj>; <obj>]
```

works because `rat_mul` and `rat_add` are *short names* for the *same object-type*

```
rat_add =
rat_mul =
  < eval : int * int;
    op : int * int -> int * int -> int * int;
    print : unit; printop : unit;
    rawop : int * int -> int * int -> int * int;
    sym : char >
```

Verbose error message (don't be affraid about):

```
# [r_1_2; e1];;
This expression has type
  rat_add =
    < eval : int * int;
      op : int * int -> int * int -> int * int;
      print : unit; printop : unit;
      rawop : int * int -> int * int -> int * int;
      sym : char >
but is here used with type
  rat_const = < eval : int * int; print : unit >
Only the first object type has a method op
```

Concurrent programming

Concurrency:

simultaneous process sharing ressources
⇒ mutual exclusion
⇒ synchronisation

with Objective Caml

Three modules

- **`Thread`**: to create, run and stop process involved in concurrent applications
- **`Mutex`**: to create, lock and release critical sections
- **`Condition`**: to create, wait and send synchronisation signals

Additonnal module

- **`ThreadUnix`**: non blocking Unix I/O

Threads

"multiple threads of control (also called lightweight processes) that execute concurrently in the same memory space"

Creation: `val create :  ('a -> 'b) -> 'a -> t`

`Thread.create f x`

1. creates a new thread to excecute `(f x)` *concurrently* with the other threads of the program.
   Note: "the program" itself is a thread.
2. returns the handle (`Thread.t`) of the created thread.
3. terminates when `(f x)` returns (or fails)
4. the result ofOCtext(f x) (or its failure) is discarded and not directly accessible to the parent thread (the one who created)

Suspend: `val delay :  float -> unit`

`Thread.delay d`

1. suspends the execution of the calling thread for d seconds.

Threads

Let’s play with

File `pingpong.ml`

```
let ping t =
  for i=0 to 10 do
    print_string "ping";
    flush stdout;
    Thread.delay t
  done ;;
let pong t =
  for i=0 to 10 do
    print_string "PONG";
    flush stdout;
    Thread.delay t
  done ;;

print_endline"ping-pong go:";
Thread.create ping 0.1;
Thread.create pong 0.05;
Thread.delay 3.0;
print_newline()
```

Threads are not in the standard library

```
ocamlc -thread -custom -o pingpong \
       unix.cma threads.cma pingpong.ml \
       -cclib -lunix -cclib -lthreads
```

Let's play with threads

Run ping-pong game:

```
[unix-prompt] ./pingpong
ping-pong go:
pingPONGPONGpingPONGPONGpingPONGPONGpingPONGPONG
pingPONGpingPONGPONGpingpingpingpingping
[unix-prompt]
```

Delays:

- in `ping` ot `pong`, allow alternation
- in main expression, leave time for threads to execute

Changing delay parameters

```
Thread.create ping 0.01;
Thread.create pong 0.05;
```

changes the ditribution

```
[unix-prompt] ./pingpong
ping-pong go:
pingPONGpingpingpingPONGpingpingpingPONGpingping
pingPONGpingPONGPONGPONGPONGPONGPONGPONG
[unix-prompt]
```

Mutual exclusion

Critical section:

A piece of code that must not be interrupted
⇒ locks

Module `Mutex`:

val create : unit -> t

Return a new mutex.

val lock : t -> unit

Lock the given mutex. Only one thread can have the mutex locked at any time. A thread that attempts to lock a mutex already locked by another thread will suspend until the other thread unlocks the mutex.

val unlock : t -> unit

Unlock the given mutex. Other threads suspended trying to lock the mutex will restart.

Let's play with

Stamming players

```
let m = Mutex.create () ;;

let f s =
  for i=0 to 5 do
    Mutex.lock m;    (* begin critical section *)
    print_string s;
    Thread.delay 0.1;
    print_string s;
    flush stdout;
    Mutex.unlock m;  (* end critical section   *)
    Thread.delay (Random.float 0.3)
  done ;;

print_endline"ping-pong go:";
Thread.create f "ping";
Thread.create f "PONG";
Thread.delay 3.0;
print_newline()
```

Delays:

- between printing should allow the other thread to play but it will not, because of mutex
- randomized to introduce some perturbation in alternation

Stamming play

Let's run

```
ping-pong go:
pingpingPONGPONGpingpingPONGPONGPONGPONGpingping
PONGPONGpingpingPONGPONGPONGPONGpingpingpingping
```

Note that `ping` and `PONG` are always displayed twice

Changing loop's body by adding one more display

```
        Mutex.lock m;    (* begin critical section *)
        print_string s;
        Thread.delay 0.1;
        print_string s;
        print_string s;
        flush stdout;
        Mutex.unlock m;  (* end critical section   *)
```

will give laternation of three consecutive `ping` and `PONG`

```
ping-pong go:
pingpingpingPONGPONGPONGpingpingpingPONGPONGPONG
PONGPONGPONGpingpingpingPONGPONGPONGpingpingping
PONGPONGPONGPONGPONGPONGpingpingpingpingpingping
```

Synchronization

Waiting for a given condition

Alternation on a boolean flag

- `ping` plays when flag is `true` and set it to `false`
- `pong` plays when flag is `false` and set it to `true`

Wait and signal: module `Condition`

val create : unit -> t

Return a new condition variable.

val wait : t -> Mutex.t -> unit

`wait c m` *atomically unlocks the mutex* `m` *and suspends the calling process on the condition variable* `c`*. The process will restart after the condition variable* `c` *has been signalled. The mutex* `m` *is locked again before wait returns.*

val signal : t -> unit

`signal c` *restarts one of the processes waiting on the condition variable* `c`*.*

Using conditions

Fair and safe alternation

```
let m = Mutex.create () ;;
let c = Condition.create () ;;
let b = ref true ;;

let f (wait, s) =
  for i=0 to 10 do
    while wait () do Condition.wait c m done;
    print_string s; flush stdout;
    b := not !b;
    Condition.signal c;
    Mutex.unlock m;
  done ;;

print_endline"ping-pong go:";
Thread.create f ((fun () -> not !b), "ping");
Thread.create f ((fun () -> !b), "PONG");
Thread.delay 1.0;
print_newline()
```

Note: the mutex `c` is used both

- to protect the signal variable `c`
- to protect the modification of the flag `b`

Using conditions (continued)

Unfair but safe alternation

`ping` will play twice more than `pong`

Use an integer flag instead of a boolean

- `ping` plays when flag is more than zero, set substract 1 from the flag and do it one more
- `pong` plays when the flag is null and set it to 2

Partial code

```
[..]
let n = ref 2 ;;
let ping () =
  for i=1 to 10 do
    while !n = 0 do Condition.wait c m done;
    print_string "ping"; flush stdout;
    n := !n-1;
    Condition.signal c; Mutex.unlock m
  done ;;
let pong () =
  for i=1 to 5 do
    while !n > 0 do Condition.wait c m done;
    print_string "PONG"; flush stdout;
    n := 2;
    Condition.signal c; Mutex.unlock m
  done ;;
[..]
```

Distributed programming

Distribution

Network communication
⇒ Internet protocol
Sockets
⇒ Client/Server

with Objective Caml

Modules `Unix`

- internet adresses and hosts database
- sockets API

Also: module `ThreadUnix`

Names and adresses

Internet adresses

Format: 32 bits usually written as `134.157.168.126`

- Abstract type `Unix.inet_addr`
- Conversion function:
  to strings: `Unix.string_of_inet_addr`
  from strings: `Unix.inet_addr_of_string`

Hosts data base

Correspondance between names and IP adresses

Host entry structure

```
type host_entry = {
  h_name : string;
  h_aliases : string array;
  h_addrtype : socket_domain;
  h_addr_list : inet_addr array; }
```

with

- `h_name`, `h_aliases`: official name and aliases
- `h_addrtype` adress type (should be `Unix.PF_INET`)
- `h_addr_list` internet adress list (may be several – gateways – but usually one)

Names and adresses (continued)

Hosts data base requests

val gethostbyname : string -> host_entry
Find an entry in hosts with the given name, or raise `Not_found`*.*

val gethostbyaddr : inet_addr -> host_entry
Find an entry in hosts with the given address, or raise `Not_found`*.*

val gethostname : unit -> string
Return the name of the local host.

Some utilities

```
# open Unix ;;
# let in_addr_of_name name =
  (gethostbyname name).h_addr_list.(0) ;;
val in_addr_of_name : string -> Unix.inet_addr = <fun>
# let name_of_in_addr in_addr =
  (gethostbyaddr in_addr).h_name ;;
val name_of_in_addr : Unix.inet_addr -> string = <fun>
# let gethostaddr () =
  in_addr_of_name (gethostname()) ;;
val gethostaddr : unit -> Unix.inet_addr = <fun>
```

I/O for Internet

The *sockets*

- Unix generic communication interface for processes
  ≈ special file descriptor
- within Objective Caml

```
type sockaddr =
  | ADDR_UNIX of string
  | ADDR_INET of inet_addr * int
```

- `ADDR_UNIX`: for local communication
- `ADDR_INET`: for (Inter)network communication
  - `inet_addr`: internet adress of the socket
  - `int`: *port number* of the socket
    ⇒ several sockets on one host

Note: the module `Unix` does not provide other socket *domain*.

Note again: we will use only Internet domain socket

Internet socket for TCP/IP

- Several possible socket's kind specifying the behaviour of the communication (⇒ protocols, no comment)

```
type socket_type =
  | SOCK_STREAM    (* Stream socket *)
  | SOCK_DGRAM     (* Datagram socket *)
  | SOCK_RAW       (* Raw socket *)
  | SOCK_SEQPACKET (* Sequenced packets socket *)
```

- Type for socket's domain

```
type socket_domain =
  | PF_UNIX (* Unix domain *)
  | PF_INET (* Internet domain *)
```

- Creation of a socket

val socket : socket_domain -> socket_type -> int -> file_descr

Create a new socket in the given domain, and with the given kind. The third argument is the protocol type; 0 selects the default protocol for that kind of sockets.

Our usage:

TCP/IP, reliable point to point communication

```
let tcp_socket () =
  Unix.socket Unix.PF_INET Unix.SOCK_STREAM 0 ;;
```

Using sockets

Quietly waiting connection

- When created, a socket has no address

val bind : file_descr -> sockaddr -> unit

Bind a socket to an address

- Configuation as a listening socket

val listen : file_descr -> int -> unit

Set up a socket for receiving connection requests. The integer argument is the maximal number of pending requests.

- Ready to accept connections

val accept : file_descr -> file_descr * sockaddr

Accept connections on the given socket. The returned descriptor is a socket connected to the client; the returned address is the address of the connecting client.

Actively asking connection

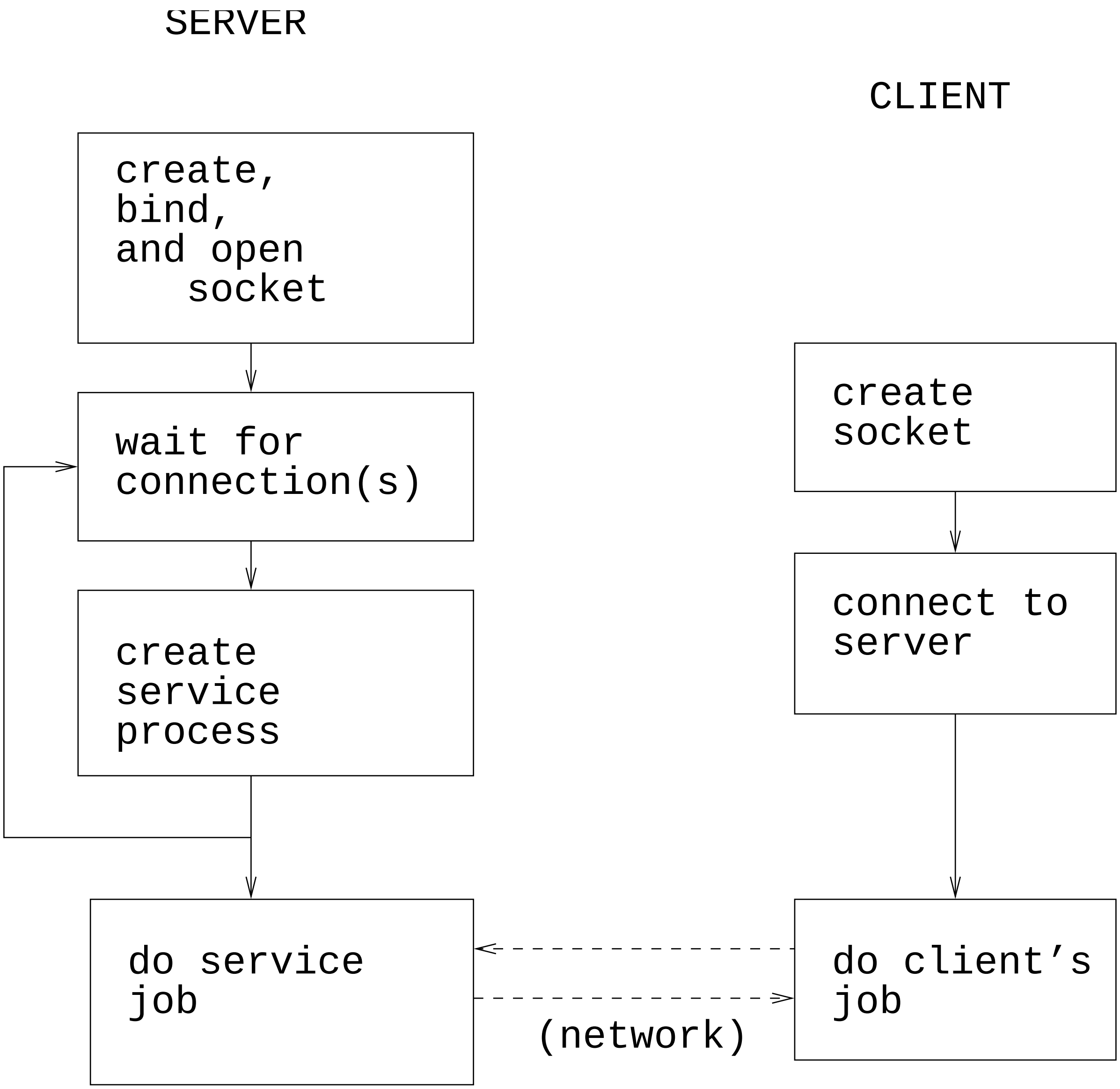
- Try to connect with an other socket

val connect : file_descr -> sockaddr -> unit

Connect a socket to an address.

Client-Server architecture

Asymmetric communicating processes



SERVER

CLIENT

create,
bind,
and open
socket

wait for
connection(s)

create
service
process

do service
job

create
socket

connect to
server

do client's
job

(network)

Distant `Hello`

A primitive service

On *port*: 12345

The server

1. waits for a client's connection,
2. when a cleint connects
   (a) build the answer "Hello machine.domain"
   (b) send it
3. loop

The client

1. gets the server's adress on the command line
2. connects to the server
3. wait the server's answer
4. print it

The `hellod` server

- utilities

```
open Unix ;;

let gethostaddr () =
  (gethostbyname(gethostname())).h_addr_list.(0)
;;

let tcp_socket () =
  socket PF_INET SOCK_STREAM 0
;;
```

Module `Unix` is open to lighten the writtings

- The service function: build and send the answer

```
let answer (c_sock, c_addr) =
  match c_addr with
    ADDR_INET(in_addr,_)
    -> (let s = Printf.sprintf
                  "Hello %s\n"
                  ((gethostbyaddr in_addr).h_name)
        in
          (ThreadUnix.write
             c_sock s 0 (String.length s)))
    | _ -> failwith "Unexpected UNIX domain"
;;
```

Note: the usage of non blocking `ThreadUnix.write`

The `Hellod` server (continued)

Main loop

- Creates, binds and configures the socket, then loops

```
let hellod () =
  let s_sock = tcp_socket () in
  let s_addr = ADDR_INET(gethostaddr (), 12345) in
    bind s_sock s_addr;
    listen s_sock 3;
    while true do
      Thread.create answer (accept s_sock)
    done
;;

hellod()
```

The arguments of the service's function are

- the "service socket", to write the answer
- the client adress to get its name to build the answer

Compiling command (with types output)

```
[unix-promt] ocamlc -thread -custom -i -o hellod \
             unix.cma threads.cma hellod.ml  \
             -cclib -lthreads -cclib -lunix
```

The `helloc` client

- Simple client function and main expression

```
open Unix ;;

let tcp_socket () =
  socket PF_INET SOCK_STREAM 0
;;

let hello server_name =
  let c_sock = tcp_socket () in
  let s_addr =
    (gethostbyname server_name).h_addr_list.(0)
  in
  let s_sock = ADDR_INET(s_addr, 12345) in
    connect c_sock s_sock;
    print_endline
      (input_line
         (in_channel_of_descr c_sock))
;;

if Array.length Sys.argv < 2 then
  Printf.eprintf "Usage : helloc hostname\n"
else
  hello Sys.argv.(1)
```

Note: the usage of `Sys.argv`

Contents