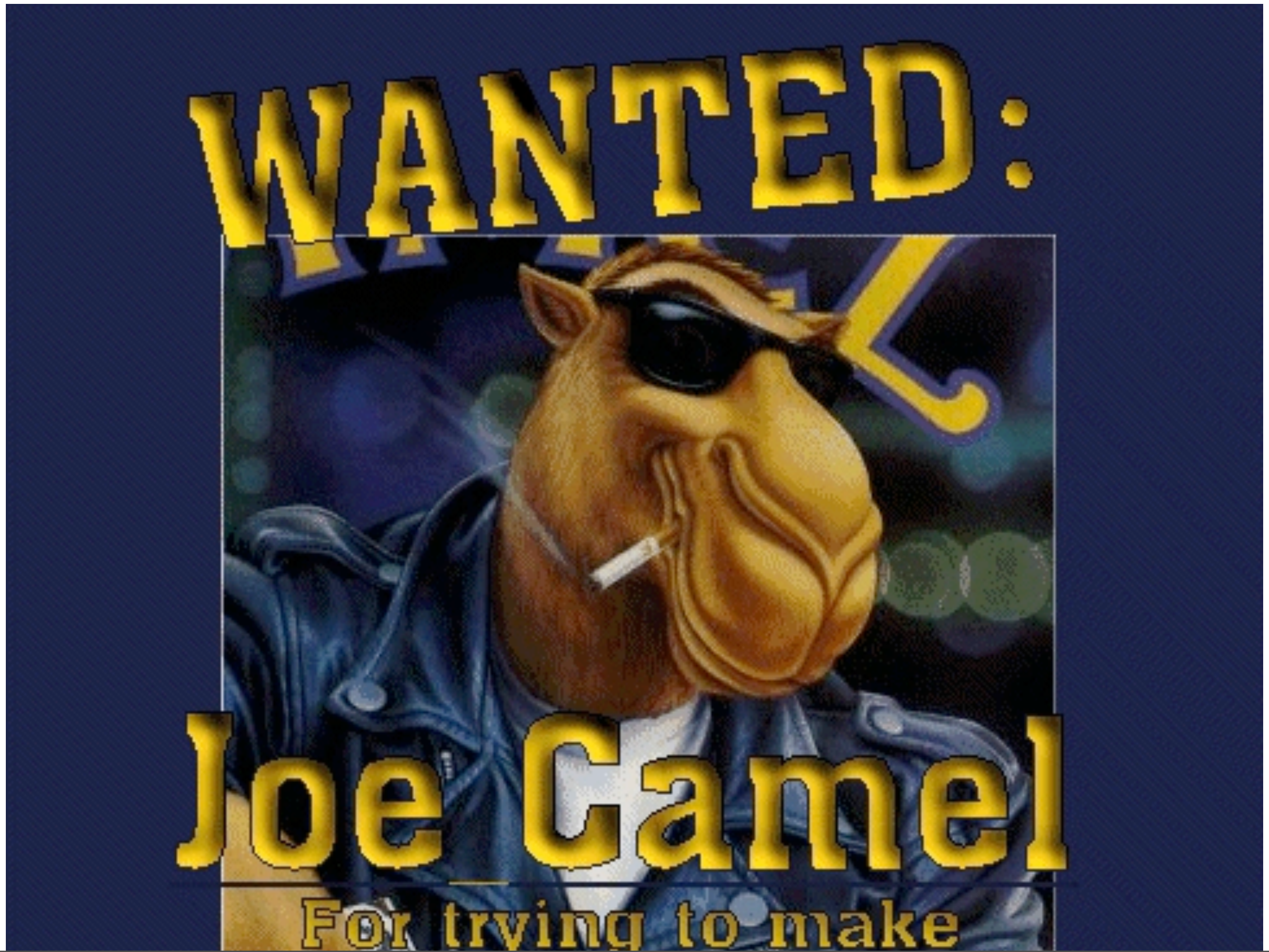
J-O-Caml (4)

jean-jacques.levy@inria.fr

pauillac.inria.fr/~levy/qinghua/j-o-caml

Qinghua, November 27



Plan of this class

- polymorphic mutable data
- modules and signatures
- manipulating terms as AST
- reading bitmaps
- ariane 5 story

Exercices

- Conway sequences - solution 2

```
# let conway x =
    let rec conway1 n a x = match x with
      | [ ] -> if n > 0 then [1;a] else [ ]
      | b :: y -> if a = b then conway1 (n+1) a y else
            if n > 0 then n :: a :: conway1 1 b y else conway1 1 b y
    in conway1 0 0 x ;;
```

Polymorphism

- variables and functions are polymorphic in Caml
- type inference gives the principal type (unique most general)
- no need to keep type information at runtime, since strong type checking

  differs from Lisp, Scheme, Java which keep and check type information
- in Caml, polymorphism only appears around `let` statements

```
# let id = function x -> x in
  print_int (id 43); print_string (id "jocaml") ;;
43jocaml- : unit = ()
# function x -> x ;;
- : 'a -> 'a = <fun>
```

Polymorphic mutable data

- mutable values can't have real polymorphic types (see below)
- they are not considered as real values

  only values have true polymorphic types
- mutable values have ``once polymorphic'' types

```
# let succ x = x + 1 ;;
val succ : int -> int = <fun>
# let loc = ref (function x -> x) in
  loc := succ; !loc "jocaml" ;;
  Characters 55-63:
    loc := succ; !loc "jocaml" ;;
                      ^^^^^^^^
Error: This expression has type string but an expression was expe
cted of type int
# ref (function x -> x) ;;
- : ('_a -> '_a) ref = {contents = <fun>}
```

Modules and Signatures

- module declaration groups related functions

```
# module FIFO = struct
    type 'a t = {mutable hd: 'a list; mutable tl: 'a list}
    let create() = {hd = [ ]; tl = [ ]}
    let add f a = f.tl <- a :: f.tl
  end;;
module FIFO :
  sig
    type 'a t = { mutable hd : 'a list; mutable tl : 'a list; }
    val create : unit -> 'a t
    val add : 'a t -> 'a -> unit
  end
#
```

- qualified names to refer to functions and types

```
# let f = FIFO.create();;
val f : '_a FIFO.t = {FIFO.hd = []; FIFO.tl = []}
# FIFO.add f 3;;
- : unit = ()
# f;;
- : int FIFO.t = {FIFO.hd = []; FIFO.tl = [3]}
```

Modules and Signatures

- implementation may be hidden by forcing signature

```
# module FIFO = (struct
    type 'a t = {mutable hd: 'a list; mutable tl: 'a list}
    let create() = {hd = [ ]; tl = [ ]}
    let add f a = f.tl <- a :: f.tl
  end :
  sig
    type 'a t
    val create : unit -> 'a t
    val add : 'a t -> 'a -> unit
  end) ;;
module FIFO : sig type 'a t val create : unit -> 'a t val add :
'a t -> 'a -> unit end
```

- qualified names to refer to functions and types

```
# let f = FIFO.create() ;;
val f : '_a FIFO.t = <abstr>
# FIFO.add f 3;;
- : unit = ()
# f ;;
- : int FIFO.t = <abstr>
# FIFO.add f "jocaml";;
Characters 11-19:
  FIFO.add f "jocaml";;
```

Modules and Signatures

- modules group set of type, exception, variable, function definitions
- type of a module is its signature
- signature can be restricted by giving it explicitly
- hiding implementation of some types produce abstract types
- several functions may also be hidden (usually auxiliary functions)
- abstract types may have several implementations (FIFO as circular buffers, FIFO as lists)
- if type is abstract, the user of this type will not see differences between implementations
- signatures are described in the Ocaml libraries
- compiled signatures are in files with suffix `.cmi`
- modules may be nested

Exercice

- Write `remove` function which removes the head of the queue in FIFO module (with creation of `EmptyQueue` exception)

- Give an alternative implementation of FIFOs with circular buffers.

- Give a module definition for addition and multiplication for big numbers (as in exercic lecture 2)

Reading bitmaps

- function to read bitmaps on standard input + 2 useful functions

  [format is: width(w) height(h) and h lines of w numbers]

```
let ncols = read_int() in
let nlignes = read_int() in
let b = bmap_read nlignes ncols in
bmap_display b;
pause();
```

```
let bmap_display b =
  let bi = make_image b in
    draw_image bi margin margin;;
```

```
let pause () =
  match wait_next_event [Button_down] with
      _ -> () ;;
```

Reading bitmaps

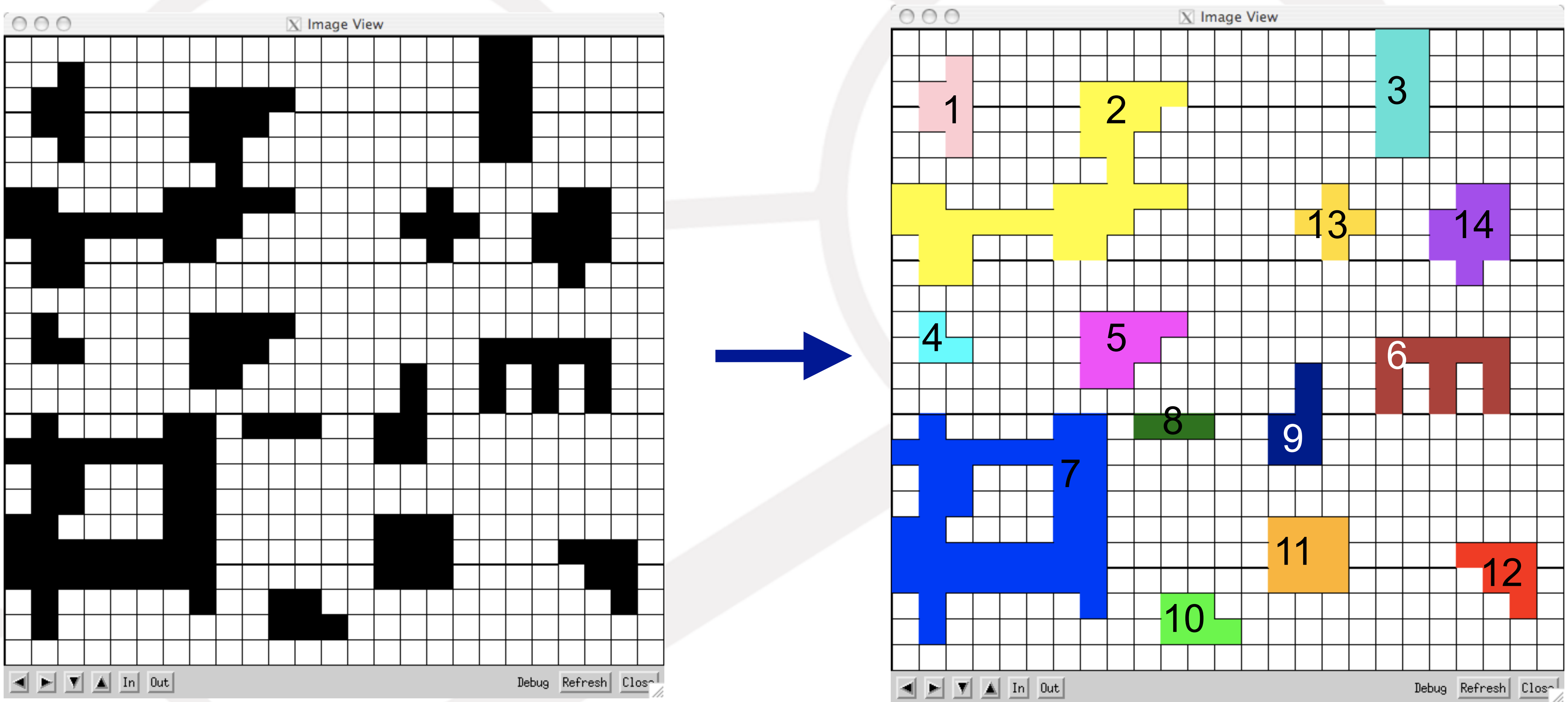
- format is: nlines and ncolumns

```
let bmap_read nlignes ncols =
  let b = Array.make_matrix nlignes ncols 0 in
  for i = 0 to nlignes - 1 do
    let s = read_line() in
    let xs = ref (Str.split (Str.regexp "[ \t]+") s) in
    for j = 0 to ncols - 1 do
      let c = int_of_string (List.hd !xs) in
        b.(i).(j) <- rgb c c c;
        xs := List.tl !xs;
    done;
  done;
  b;;
```

Combien d'objets dans une image?

Jean-Jacques Lévy
INRIA

Labeling



16 objects in this picture

Algorithm

1) first pass

- scan pixels left-to-right, top-to-bottom giving a new object id each time a new object is met

2) second pass

- generate equivalences between ids due to new adjacent relations met during scan of pixels.

3) third pass

- compute the number of equivalence classes

Complexity:

- scan twice full image (linear cost)

- try to efficiently manage equivalence classes (Union-Find by Tarjan)