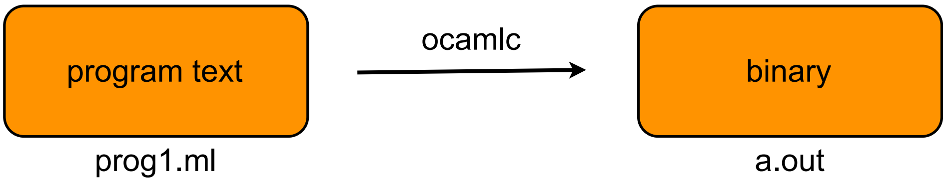
J-O-Caml (2)

jean-jacques.levy@inria.fr
pauillac.inria.fr/~levy/qinghua/j-o-caml
Qinghua, November 20

Plan of this class

- compiling programs
- list processing
- pattern-matching
- new data types
- labeling algorithm

Compiling programs



- `ocamlc -o prog1` produces `prog1` (executable byte code)
- `ocamlc -c prog1.ml` produces `prog1.cmo` (byte code)
- `ocamlopt -c prog1.ml` produces `prog1.cmx` (binary)
- also files `prog1.cmi` (compiled interfaces -- see later)

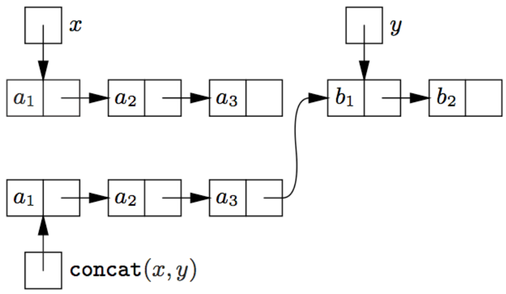
List processing

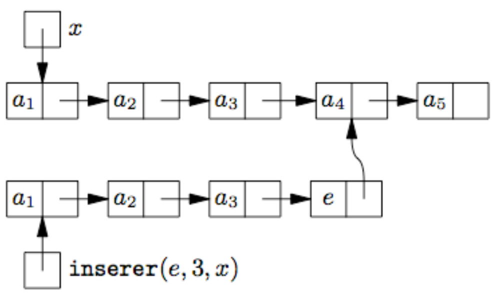
```
# let rec length x = match x with
  | [ ] -> 0
  | a :: x' -> 1 + length x' ;;

# let rec concat x y = match x with
  | [ ] -> y
  | a :: x' -> a :: concat x' y ;;

# let rec reverse x = match x with
  | [ ] -> [ ]
  | a :: x' -> concat (reverse x') [a] ;;

# let rec insert a n x =
    if n = 0 then a :: x else match x with
    | [ ] -> [ ]
    | b :: x' -> b :: insert a (n-1) x' ;;
```





List processing

```
# let rec length x = match x with
  | [ ] -> 0
  | a :: x' -> 1 + length x' ;;
    val length : 'a list -> int = <fun>
# let rec concat x y = match x with
  | [ ] -> y
  | a :: x' -> a :: concat x' y ;;
    val concat : 'a list -> 'a list -> 'a list = <fun>
# let rec reverse x = match x with
  | [ ] -> [ ]
  | a :: x' -> concat (reverse x') [a] ;;
    val reverse : 'a list -> 'a list = <fun>
# let rec insert a n x =
    if n = 0 then a :: x else match x with
    | [ ] -> [ ]
    | b :: x' -> b :: insert a (n-1) x' ;;
      val insert : 'a -> int -> 'a list -> 'a list = <fun>
```


List processing

```
# let rec length x = match x with
  | [ ] -> 0
  | a :: x' -> 1 + length x' ;;
    val length : 'a list -> int = <fun>
# let rec concat x y = match x with
  | [ ] -> y
  | a :: x' -> a :: concat x' y ;;
    val concat : 'a list -> 'a list -> 'a list = <fun>
# let rec reverse x = match x with
  | [ ] -> [ ]
  | a :: x' -> concat (reverse x') [a] ;;
    val reverse : 'a list -> 'a list = <fun>
# let rec insert a n x =
    if n = 0 then a :: x else match x with
    | [ ] -> [ ]
    | b :: x' -> b :: insert a (n-1) x' ;;
      val insert : 'a -> int -> 'a list -> 'a list = <fun>
```


- polymorphic types
- `'a list` (say ``alpha list") is type of list of any type, e.g. `int list`, `bool list`, `...`

Exercices on lists

Exercices on lists

- Arbitrary precision numbers can be implemented by lists (little endian or big endian style). Write addition and multiplication algorithms.
- Conway sequence of lists starts with list `[1]`. Then next list in sequence is obtainte by reading previous list. Therefore `[1; 1]`, `[2; 1]`, `[1; 2; 1; 1]`, `[1; 1; 1; 2; 2; 1]`.... Print lists of Conway sequence. Is there any unlucky number ?

New data types

- products

```
# let move (x, y) (dx, dy) = (x +. dx, y +. dy);;

# move (2.1, 3.0) (0.5, 0.5);;

# let dotProduct (x, y) (x', y') = (x *. x' +. y *. y');;

# let crossProduct (x, y) (x', y') = (x *. y' -. x *. y');;
```

New data types

- products

```
# let move (x, y) (dx, dy) = (x +. dx, y +. dy);;
val move : float * float -> float * float -> float * float = <fun>
# move (2.1, 3.0) (0.5, 0.5);;
- : float * float = (2.6, 3.5)
# let dotProduct (x, y) (x', y') = (x *. x' +. y *. y');;
val dotProduct : float * float -> float * float -> float = <fun>
# let crossProduct (x, y) (x', y') = (x *. y' -. x *. y');;
val crossProduct : float * 'a -> 'b * float -> float = <fun>
```

New data types

- sums

```
# type complex = Cartesian of float * float | Polar of float * float ;;

# let polar_of_cartesian c = match c with
  | Cartesian (x, y) -> Polar (sqrt(x *. x +. y *. y), atan (y /. x))
  | Polar (rho, theta) -> c ;;

# let cartesian_of_polar c = match c with
  | Polar (rho, theta) -> Cartesian (rho *. (cos theta), rho *. (sin theta))
  | _ -> c ;;

# let add c c' =
    let c1 = cartesian_of_polar c and c1' = cartesian_of_polar c' in
    match c1, c1' with
      | Cartesian(x, y), Cartesian(x', y') -> Cartesian(x +. x', y +. y')
      | _ -> failwith "Impossible" ;;
```

New data types

- sums

```
# type complex = Cartesian of float * float | Polar of float * float ;;
type complex = Cartesian of float * float | Polar of float * float
# let polar_of_cartesian c = match c with
  | Cartesian (x, y) -> Polar (sqrt(x *. x +. y *. y), atan (y /. x))
  | Polar (rho, theta) -> c ;;
    val polar_of_cartesian : complex -> complex = <fun>
# let cartesian_of_polar c = match c with
  | Polar (rho, theta) -> Cartesian (rho *. (cos theta), rho *. (sin theta))
  | _ -> c ;;
    val cartesian_of_polar : complex -> complex = <fun>
# let add c c' =
    let c1 = cartesian_of_polar c and c1' = cartesian_of_polar c' in
    match c1, c1' with
      | Cartesian(x, y), Cartesian(x', y') -> Cartesian(x +. x', y +. y')
      | _ -> failwith "Impossible" ;;
        val add : complex -> complex -> complex = <fun>
```

New data types

- recursive types

```
# type tree = Empty of unit | Node of tree * int * tree;;

# let rec size a = match a with
  | Empty() -> 0
  | Node (left, _, right) -> 1 + size left + size right ;;

# size (Node(Empty(), 3, Node (Empty(), 4, Empty())));;
```

New data types

- recursive types

```
# type tree = Empty of unit | Node of tree * int * tree;;
type tree = Empty of unit | Node of tree * int * tree
# let rec size a = match a with
  | Empty() -> 0
  | Node (left, _, right) -> 1 + size left + size right ;;
    val size : tree -> int = <fun>
# size (Node(Empty(), 3, Node (Empty(), 4, Empty())));;
- : int = 2
```

New data types

- recursive types

```
# type tree = Empty of unit | Node of tree * int * tree;;

# let rec size a = match a with
  | Empty() -> 0
  | Node (left, _, right) -> 1 + size left + size right ;;

# size (Node(Empty(), 3, Node (Empty(), 4, Empty())));;
```

- recursive polymorphic types

```
# type 'a tree = Empty of unit | Node of 'a tree * 'a * 'a tree ;;

# let rec size a = match a with
  | Empty() -> 0
  | Node (left, _, right) -> 1 + size left + size right ;;

# let a =  Node(Empty(), 3, Node (Empty(), 4, Empty()));;

# let b =  Node(Empty(), "nihao", Node (Empty(), "bushi", Empty()));;


# size b;;
- : int = 2
# size a;;
- : int = 2
```

New data types

- recursive types

```
# type tree = Empty of unit | Node of tree * int * tree;;
type tree = Empty of unit | Node of tree * int * tree
# let rec size a = match a with
  | Empty() -> 0
  | Node (left, _, right) -> 1 + size left + size right ;;
    val size : tree -> int = <fun>
# size (Node(Empty(), 3, Node (Empty(), 4, Empty())));;
- : int = 2
```

- recursive polymorphic types

```
# type 'a tree = Empty of unit | Node of 'a tree * 'a * 'a tree ;;
type 'a tree = Empty of unit | Node of 'a tree * 'a * 'a tree
# let rec size a = match a with
  | Empty() -> 0
  | Node (left, _, right) -> 1 + size left + size right ;;
    val size : 'a tree -> int = <fun>
# let a =  Node(Empty(), 3, Node (Empty(), 4, Empty()));;
val a : int tree = Node (Empty (), 3, Node (Empty (), 4, Empty ()))
# let b =  Node(Empty(), "nihao", Node (Empty(), "bushi", Empty()));;
val b : string tree =
  Node (Empty (), "nihao", Node (Empty (), "bushi", Empty ()))
# size b;;
- : int = 2
# size a;;
- : int = 2
```

New data types

- recursive polymorphic types (alternative definition for binary trees)

```
# type 'a tree = Leaf of 'a | Node of 'a tree * 'a * 'a tree ;;

# let rec flatten a = match a with
  | Leaf (x) -> [ x ]
  | Node (tleft, _ , tright) -> List.append (flatten tleft) (flatten tright) ;;

# let a = Node ( Leaf (3), 5, Node (Leaf (4), 7, Leaf (6))) ;;

# flatten a ;;

# let rec flatten a = match a with
  | Leaf (x) -> [ x ]
  | Node (tleft, _ , tright) -> (flatten tleft) @ (flatten tright) ;;
```

New data types

- recursive polymorphic types (alternative definition for binary trees)

```
# type 'a tree = Leaf of 'a | Node of 'a tree * 'a * 'a tree ;;
type 'a tree = Leaf of 'a | Node of 'a tree * 'a * 'a tree
# let rec flatten a = match a with
  | Leaf (x) -> [ x ]
  | Node (tleft, _ , tright) -> List.append (flatten tleft) (flatten tright) ;;
    val flatten : 'a tree -> 'a list = <fun>
# let a = Node ( Leaf (3), 5, Node (Leaf (4), 7, Leaf (6))) ;;
val a : int tree = Node (Leaf 3, 5, Node (Leaf 4, 7, Leaf 6))
# flatten a ;;
- : int list = [3; 4; 6]
# let rec flatten a = match a with
  | Leaf (x) -> [ x ]
  | Node (tleft, _ , tright) -> (flatten tleft) @ (flatten tright) ;;
    val flatten : 'a tree -> 'a list = <fun>
# flatten a;;
- : int list = [3; 4; 6]
```

Caring about space

- with an extra argument as an accumulator of the result

```
# let flatten a =
    let rec flatten1 a res = match a with
      | Leaf (x) -> x :: res
      | Node (tleft, _, tright) -> flatten1 tleft (flatten1 tright res)
    in flatten1 a [ ] ;;
        val flatten : 'a tree -> 'a list = <fun>
# flatten a;;
- : int list = [3; 4; 6]
```

Caring about space

- with an extra argument as an accumulator of the result

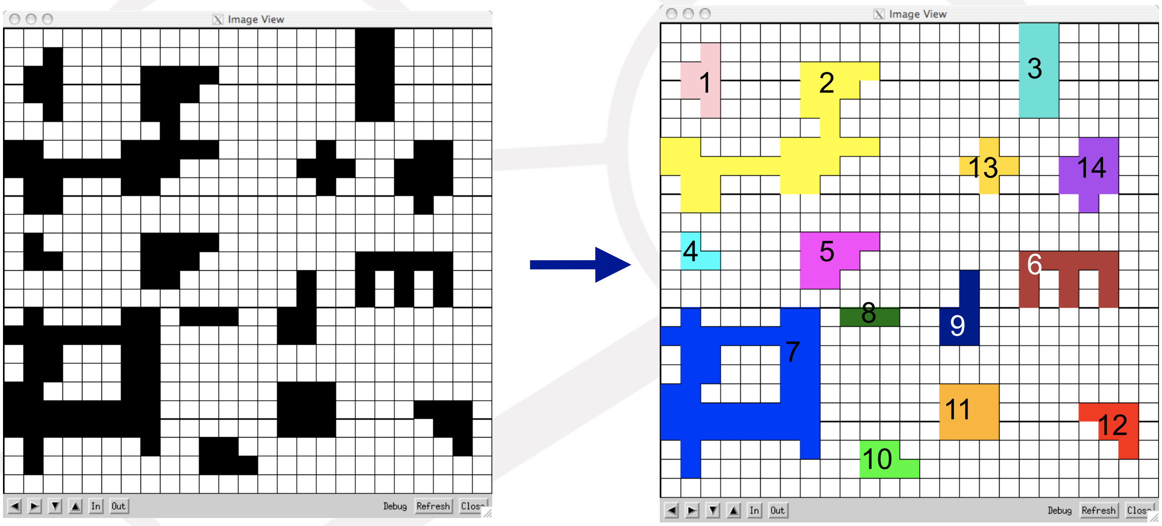
```
# let flatten a =
    let rec flatten1 a res = match a with
      | Leaf (x) -> x :: res
      | Node (tleft, _, tright) -> flatten1 tleft (flatten1 tright res)
    in flatten1 a [ ] ;;
        val flatten : 'a tree -> 'a list = <fun>
# flatten a;;
- : int list = [3; 4; 6]
```

Combien d'objets dans une image?

Jean-Jacques Lévy
INRIA

CENTRE DE RECHERCHE COMMUN INRIA MICROSOFT RESEARCH

Labeling



16 objects in this picture

CENTRE DE RECHERCHE COMMUN INRIA MICROSOFT RESEARCH

Naive algorithm

1) choose an unvisited pixel
2) traverse all similar connected pixels
3) and restart until all pixel are visited

Very high complexity:

- how to find an unvisited pixel ?
- how organizing exploration of connected pixels (which direction?)

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)

Animation

with Polka system for animated algorithms

CENTRE DE RECHERCHE COMMUN
INRIA MICROSOFT RESEARCH

Exercise for next class

- find an algorithm for the labeling algorithm

Equivalence classes

Equivalence classes

"Union-Find"

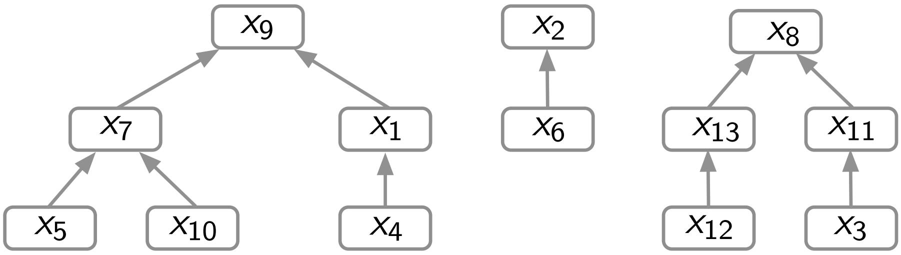
- objects $x_1, x_2, \ldots, x_n$
- equivalence*s*
- find the equivalence class of x_p
- 3 operations:
  - $\mathtt{NEW}(x)$ new object
  - $\mathtt{FIND}(x)$ find canonical representative
  - $\mathtt{UNION}(x, y)$ merge 2 equivalence classes

CENTRE DE RECHERCHE COMMUN
INRIA MICROSOFT RESEARCH

Equivalence classes

"Union-Find"

- with tree-like structure



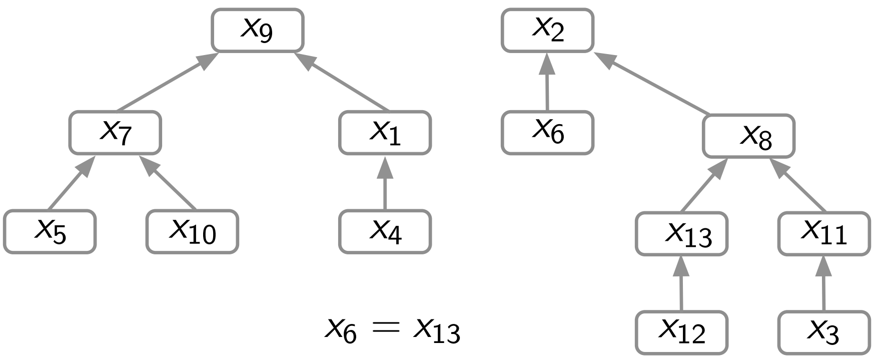
- given by the array of direct ancestors

`ancestor`	0	9	2	11	1	7	2	9	8	9	7	8	13	8

Equivalence classes

"Union-Find"

- unbalanced trees because of merges
  - $\mathtt{UNION}(x, y)$ merge 2 equivalence classes

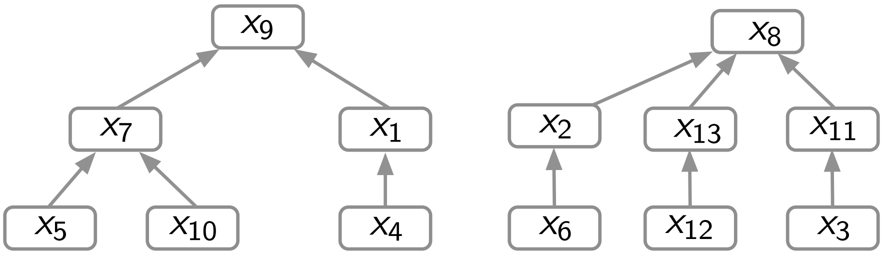


$x_6 = x_{13}$

Equivalence classes

"Union-Find"

- try to balance
  - $\texttt{UNION}(x, y)$ merge 2 equivalence classes



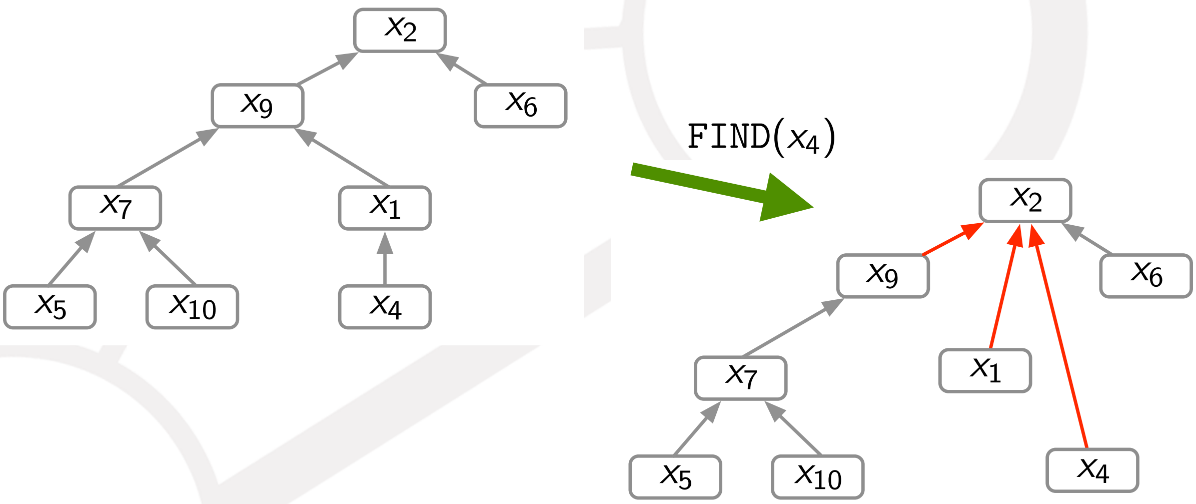
therefore keep height at each node

$O(\log n)$ opérations pour $\texttt{FIND}(x)$

Equivalence classes

"Union-Find"

- compression of path toward canonical representative
  - $\texttt{FIND}(x)$ with side-effect on tree structure



Exercice for next class

- find good primitives for graphics in Ocaml library
- design the overall structure of the labeling program