Programmation extrême avec Objective Caml

Jean-Christophe Filliâtre

équipe Démons

Séminaire IA, 14 mars 2006

l'équipe Démons et Objective Caml

une longue histoire

idée de cet exposé : partager notre **enthousiasme**

Introduction

l'équipe Démons et Objective Caml

une longue histoire

idée de cet exposé : partager notre **enthousiasme**

Plan

1. L'équipe Démons et Ocaml
   - enseignement
   - logiciels développés
   - concours de programmation
2. Objective Caml
   - historique
   - points forts
   - quelques idiomes

Enseignement d'Ocaml à Orsay

- L2
  - Approche Fonctionnelle
  - Principes d'Interprétation des Langages (PIL)
  - Langages et Génie Logiciel (LGL)
  - Combinatoire pour l'informatique
- L3 et M1
  - Informatique théorique
- M1
  - Cours de mise à niveau
  - Compilation (utilisation en TD)
  - Projet de compilation

l'équipe Démons participe activement à ces enseignements

Enseignement d'Ocaml à Orsay

- L2
  - Approche Fonctionnelle
  - Principes d'Interprétation des Langages (PIL)
  - Langages et Génie Logiciel (LGL)
  - Combinatoire pour l'informatique
- L3 et M1
  - Informatique théorique
- M1
  - Cours de mise à niveau
  - Compilation (utilisation en TD)
  - Projet de compilation

l'équipe Démons participe activement à ces enseignements

Logiciels développés dans l'équipe Démons

- **Coq** (assistant de preuve) [118000]
- **CiME** (boite à outils de réécriture) [64000]
- **Why** / **Krakatoa** / **Caduceus** (preuve de programmes) [42000]
- **bibtex2html** [4700]
  notamment pour les pages web de l'équipe
  (avec **hlins** et **hevea**, autres programmes Ocaml)
- applications et bibliothèques diverses par dizaines
  dont **ocamlgraph**
- ... et chaque année le **concours de programmation de l'ICFP !**

Logiciels développés dans l'équipe Démons

- **Coq** (assistant de preuve) [118000]
- **CiME** (boite à outils de réécriture) [64000]
- **Why** / **Krakatoa** / **Caduceus** (preuve de programmes) [42000]
- **bibtex2html** [4700]
  notamment pour les pages web de l'équipe
  (avec **hlins** et **hevea**, autres programmes Ocaml)
- applications et bibliothèques diverses par dizaines
  dont **ocamlgraph**
- ... et chaque année le **concours de programmation de l'ICFP !**

Logiciels développés dans l'équipe Démons

- **Coq** (assistant de preuve) [118000]
- **CiME** (boite à outils de réécriture) [64000]
- **Why** / **Krakatoa** / **Caduceus** (preuve de programmes) [42000]
- **bibtex2html** [4700]
  notamment pour les pages web de l'équipe
  (avec **hlins** et **hevea**, autres programmes Ocaml)
- applications et bibliothèques diverses par dizaines
  dont **ocamlgraph**
- ... et chaque année le **concours de programmation de l'ICFP !**

Logiciels développés dans l'équipe Démons

- **Coq** (assistant de preuve) [118000]
- **CiME** (boite à outils de réécriture) [64000]
- **Why** / **Krakatoa** / **Caduceus** (preuve de programmes) [42000]
- **bibtex2html** [4700]
  notamment pour les pages web de l'équipe
  (avec **hlins** et **hevea**, autres programmes Ocaml)
- applications et bibliothèques diverses par dizaines
  dont **ocamlgraph**
- ... et chaque année le **concours de programmation de l'ICFP !**

Logiciels développés dans l'équipe Démons

- **Coq** (assistant de preuve) [118000]
- **CiME** (boite à outils de réécriture) [64000]
- **Why** / **Krakatoa** / **Caduceus** (preuve de programmes) [42000]
- **bibtex2html** [4700]
  notamment pour les pages web de l'équipe
  (avec **hlins** et **hevea**, autres programmes Ocaml)
- applications et bibliothèques diverses par dizaines
  dont **ocamlgraph**
- ... et chaque année le **concours de programmation de l'ICFP !**

Logiciels développés dans l'équipe Démons

- **Coq** (assistant de preuve) [118000]
- **CiME** (boite à outils de réécriture) [64000]
- **Why** / **Krakatoa** / **Caduceus** (preuve de programmes) [42000]
- **bibtex2html** [4700]
  notamment pour les pages web de l'équipe
  (avec **hlins** et **hevea**, autres programmes Ocaml)
- applications et bibliothèques diverses par dizaines
  dont **ocamlgraph**
- ... et chaque année le **concours de programmation de l'ICFP !**

Concours de programmation de l'ICFP

année	pgms Ocaml	place
2005	21/161	**5^{e}**
2004	24/218	prix du jury
2003	22/118	1er (- de 24h)
2002	21/160	1er
2001	34/171	3^{e}
2000	6/39	1er et 2^{e}
1999	6/37	1er
1998	—	2^{e}

cette année : 21–24 juillet (`http://icfpcontest.org/`)

Concours de programmation de l'ICFP

année	pgms Ocaml	place
2005	21/161	**5^{e}**
2004	24/218	prix du jury
2003	22/118	1er (- de 24h)
2002	21/160	1er
2001	34/171	3^{e}
2000	6/39	1er et 2^{e}
1999	6/37	1er
1998	—	2^{e}

cette année : 21–24 juillet (`http://icfpcontest.org/`)

Programmation extrême

concours sur **72 heures** $\Rightarrow$ pratique de la **programmation extrême**

- développer **rapidement** et **collaborativement**
- mise en œuvre d'algorithmes complexes
  - plus court chemin, tri topologique, alpha-beta, etc.
  - lexeurs et parseurs
  - compilateurs
  - interfaces graphiques
- **correction** du programme primordiale
- **efficacité** du programme parfois importante

Programmation extrême

concours sur **72 heures** $\Rightarrow$ pratique de la **programmation extrême**

- développer **rapidement** et **collaborativement**
- mise en œuvre d'algorithmes complexes
  - plus court chemin, tri topologique, alpha-beta, etc.
  - lexeurs et parseurs
  - compilateurs
  - interfaces graphiques
- **correction** du programme primordiale
- **efficacité** du programme parfois importante

Programmation extrême

concours sur **72 heures** $\Rightarrow$ pratique de la **programmation extrême**

- développer **rapidement** et **collaborativement**
- mise en œuvre d'algorithmes complexes
  - plus court chemin, tri topologique, alpha-beta, etc.
  - lexeurs et parseurs
  - compilateurs
  - interfaces graphiques
- **correction** du programme primordiale
- **efficacité** du programme parfois importante

Programmation extrême

concours sur **72 heures** $\Rightarrow$ pratique de la **programmation extrême**

- développer **rapidement** et **collaborativement**
- mise en œuvre d'algorithmes complexes
  - plus court chemin, tri topologique, alpha-beta, etc.
  - lexeurs et parseurs
  - compilateurs
  - interfaces graphiques
- **correction** du programme primordiale
- **efficacité** du programme parfois importante

Programmation extrême

concours sur **72 heures** $\Rightarrow$ pratique de la **programmation extrême**

- développer **rapidement** et **collaborativement**
- mise en œuvre d'algorithmes complexes
  - plus court chemin, tri topologique, alpha-beta, etc.
  - lexeurs et parseurs
  - compilateurs
  - interfaces graphiques
- **correction** du programme primordiale
- **efficacité** du programme parfois importante

Objective Caml, objectivement

Historique

fin des années 70, Milner introduit **ML** = **M**eta **L**anguage
destiné au développement d'un assistant de preuve (LCF)

années 80, **CAML** développé à l'INRIA dans le même but
devient **Caml Light** (92) puis enfin **Objective Caml** (96)

autres langages de la famille ML

- Standard ML
  - SML-NJ
  - MLton
  - Moscow ML
- Haskell

Historique

fin des années 70, Milner introduit **ML** = **M**eta **L**anguage
destiné au développement d'un assistant de preuve (LCF)

années 80, **CAML** développé à l'INRIA dans le même but
devient **Caml Light** (92) puis enfin **Objective Caml** (96)

autres langages de la famille ML

- Standard ML
  - SML-NJ
  - MLton
  - Moscow ML
- Haskell

Historique

fin des années 70, Milner introduit **ML** = **M**eta **L**anguage
destiné au développement d'un assistant de preuve (LCF)

années 80, **CAML** développé à l'INRIA dans le même but
devient **Caml Light** (92) puis enfin **Objective Caml** (96)

autres langages de la famille ML

- Standard ML
  - SML-NJ
  - MLton
  - Moscow ML
- Haskell

Quelques succès d'Ocaml

- le synchroniseur de fichiers **Unison**
- le client pair-à-pair **MLdonkey**
- le Langage de Modélisation Financière de **LexiFi**
- le système de communication distribuée **Ensemble**
- l'assistant de preuve **Coq**
- l'analyseur statique **ASTRÉE**
- le vérificateur de programmes C **SLAM** de Microsoft
- la bibliothèque **FFTW** de Transformées de Fourier Discrètes

Styles de programmation

Ocaml intègre tout à la fois la programmation

- fonctionnelle
- impérative
- générique
- modulaire
- orientée object

aucun style n'est privilégié ; ils se **combinent** plutôt

Styles de programmation

Ocaml intègre tout à la fois la programmation

- fonctionnelle
- impérative
- générique
- modulaire
- orientée object

aucun style n'est privilégié ; ils se **combinent** plutôt

Points forts d'Ocaml

- système de types puissant
  - **typage statique**, avec **inférence** de types et **polymorphisme**
  - la grande majorité des erreurs est détectée à la compilation
  - note : seul langage à combiner objets et typage statique avec inférence
- types algébriques et **filtrage**
  - particulièrement adapté au calcul symbolique
- **GC**
- compilateurs **bytecode** et **natif**
  - compilation séparée
  - code produit efficace, comparable à gcc
  - nombreuses architectures supportées

Points forts d'Ocaml

- système de types puissant
  - **typage statique**, avec **inférence** de types et **polymorphisme**
  - la grande majorité des erreurs est détectée à la compilation
  - note : seul langage à combiner objets et typage statique avec inférence
- types algébriques et **filtrage**
  - particulièrement adapté au calcul symbolique
- GC
- compilateurs **bytecode** et **natif**
  - compilation séparée
  - code produit efficace, comparable à gcc
  - nombreuses architectures supportées

Points forts d'Ocaml

- système de types puissant
  - **typage statique**, avec **inférence** de types et **polymorphisme**
  - la grande majorité des erreurs est détectée à la compilation
  - note : seul langage à combiner objets et typage statique avec inférence
- types algébriques et **filtrage**
  - particulièrement adapté au calcul symbolique
- **GC**
- compilateurs **bytecode** et **natif**
  - compilation séparée
  - code produit efficace, comparable à gcc
  - nombreuses architectures supportées

Points forts d'Ocaml

- système de types puissant
  - **typage statique**, avec **inférence** de types et **polymorphisme**
  - la grande majorité des erreurs est détectée à la compilation
  - note : seul langage à combiner objets et typage statique avec inférence
- types algébriques et **filtrage**
  - particulièrement adapté au calcul symbolique
- **GC**
- compilateurs **bytecode** et **natif**
  - compilation séparée
  - code produit efficace, comparable à gcc
  - nombreuses architectures supportées

Filtrage

Tri par insertion

```
let rec sort = function
  | [] → []
  | x :: l → insert x (sort l)

and insert x = function
  | [] → [x]
  | y :: l → if x < y then x :: y :: l else y :: insert x l
```

```
val sort : 'a list → 'a list = <fun>
val insert : 'a → 'a list → 'a list = <fun>
```

Tri par insertion

```
let rec sort = function
  | [] → []
  | x :: l → insert x (sort l)

and insert x = function
  | [] → [x]
  | y :: l → if x < y then x :: y :: l else y :: insert x l
```

```
val sort : 'a list → 'a list = <fun>
val insert : 'a → 'a list → 'a list = <fun>
```

Types algébriques et filtrage

on peut définir de nouveaux types comme sommes de produits étiquetés
$\approx$ union de structures en C

exemple : arbres binaires

```
type tree = Empty | Node of tree * string * tree
```

filtrage = construction `switch` généralisée

le typage garantie

- la bonne formation des valeurs
- l'examen de tous les cas

Types algébriques et filtrage

on peut définir de nouveaux types comme sommes de produits étiquetés
$\approx$ union de structures en C

exemple : arbres binaires

```
type tree = Empty | Node of tree * string * tree
```

filtrage = construction `switch` généralisée

le typage garantie

- la bonne formation des valeurs
- l'examen de tous les cas

Types algébriques et filtrage

on peut définir de nouveaux types comme sommes de produits étiquetés
$\approx$ union de structures en C

exemple : arbres binaires

```
type tree = Empty | Node of tree * string * tree
```

filtrage = construction `switch` généralisée

le typage garantie

- la bonne formation des valeurs
- l'examen de tous les cas

Types algébriques et filtrage

on peut définir de nouveaux types comme sommes de produits étiquetés
$\approx$ union de structures en C

exemple : arbres binaires

```
type tree = Empty | Node of tree * string * tree
```

filtrage = construction `switch` généralisée

le typage garantie

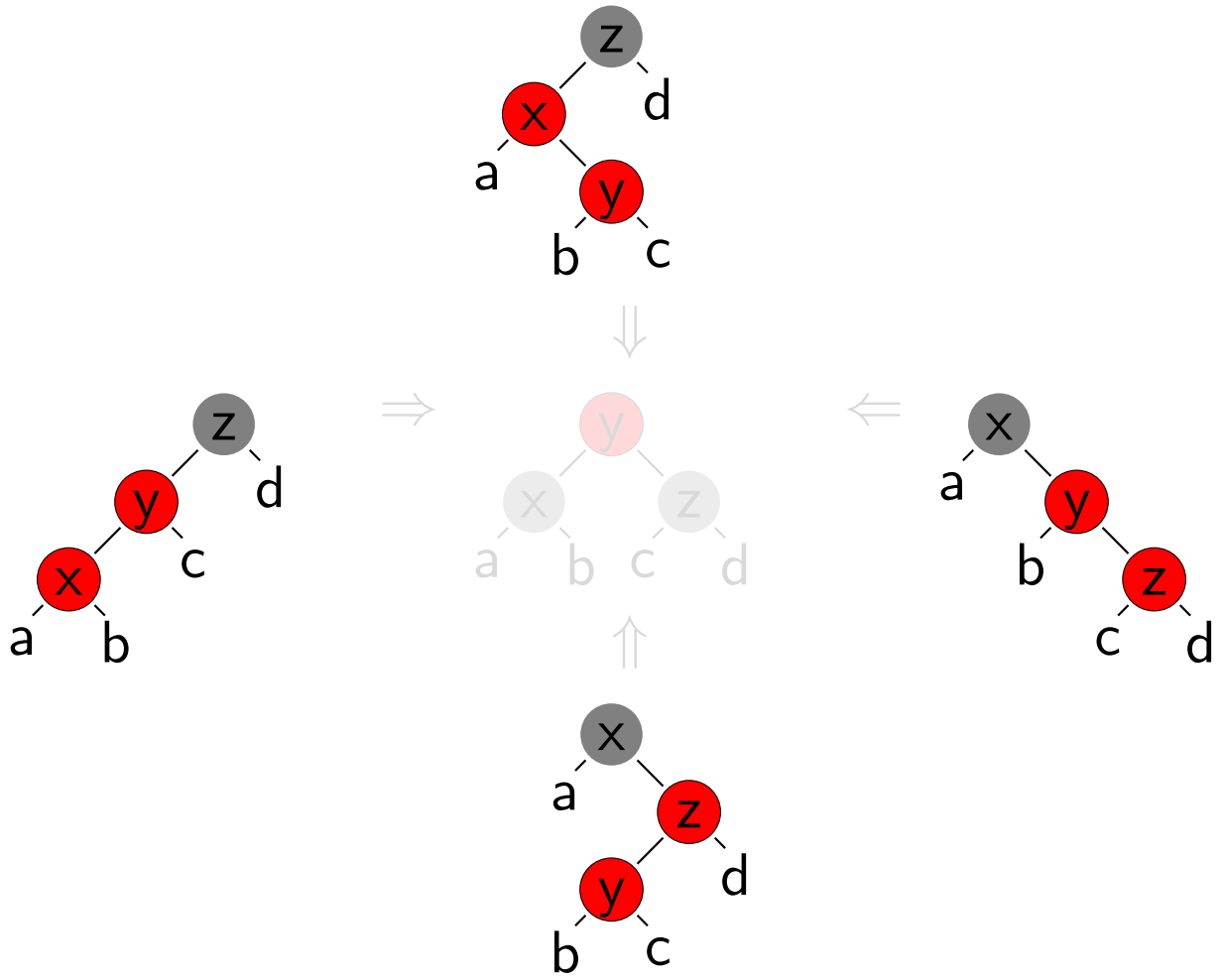
- la bonne formation des valeurs
- l'examen de tous les cas

Arbres rouges et noirs

```
type color = Red | Black
type 'a t = Empty | Node of color * 'a t * 'a * 'a t
```

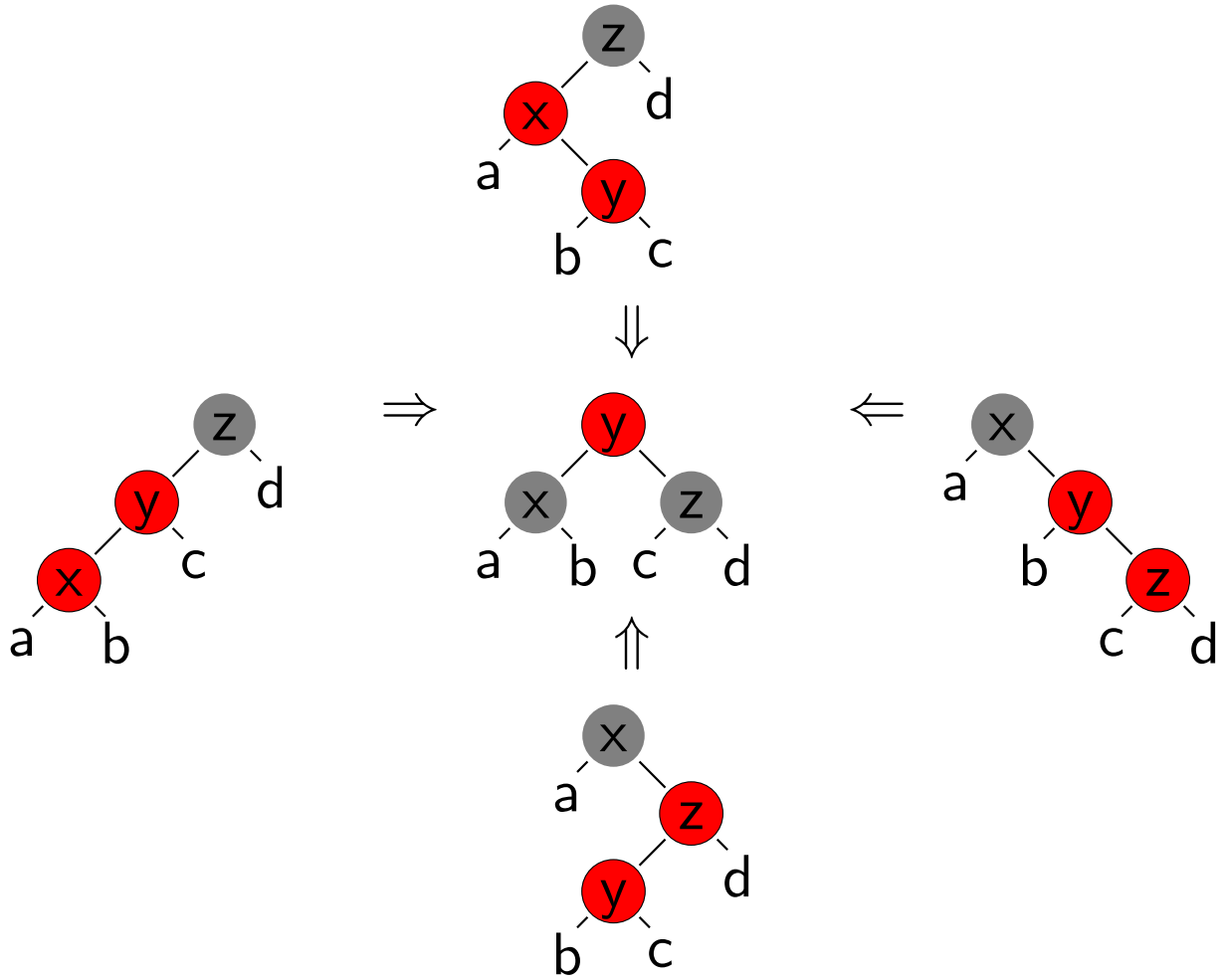
Arbres rouges et noirs

```
type color = Red | Black
type 'a t = Empty | Node of color * 'a t * 'a * 'a t
```



Arbres rouges et noirs

```
type color = Red | Black
type 'a t = Empty | Node of color * 'a t * 'a * 'a t
```



Arbres rouges et noirs

```
let balance = function
  | Black, Node (Red, Node (Red, a,x,b), y, c), z, d
  | Black, Node (Red, a, x, Node (Red, b,y,c)), z, d
  | Black, a, x, Node (Red, Node (Red, b,y,c), z, d)
  | Black, a, x, Node (Red, b, y, Node (Red, c,z,d)) →
      Node (Red, Node (Black, a,x,b), y, Node (Black, c,z,d))
  | c, l, x, r →
      Node (c, l, x, r)
```

Manipulations symboliques

le filtrage facilite les **manipulations symboliques**

exemples :

- arbres de syntaxe abstraite (compilateurs, ...)
- structures de données arborescentes
- etc.

représentation mémoire efficace par le compilateur Ocaml
aussi bien qu'un (très) bon programmeur C

Manipulations symboliques

le filtrage facilite les **manipulations symboliques**

exemples :

- arbres de syntaxe abstraite (compilateurs, ...)
- structures de données arborescentes
- etc.

représentation mémoire efficace par le compilateur Ocaml
aussi bien qu'un (très) bon programmeur C

Ordre supérieur

L'essence de la programmation fonctionnelle

programmation fonctionnelle = fonctions comme valeurs de première classe

fonctions passées en argument ou renvoyées

est-ce vraiment différent des pointeurs de fonctions ?

L'essence de la programmation fonctionnelle

programmation fonctionnelle = fonctions comme valeurs de première classe

fonctions passées en argument ou renvoyées

est-ce vraiment différent des pointeurs de fonctions ?

Un tri par insertion plus générique

```
let rec sort lt = function
  | [] → []
  | x :: l → insert lt x (sort lt l)

and insert lt x = function
  | [] →
      [x]
  | y :: l →
      if lt x y then x :: y :: l else y :: insert lt x l
```

```
val sort : ('a → 'a → bool) → 'a list → 'a list
val insert : ('a → 'a → bool) → 'a → 'a list → 'a list
```

Un tri par insertion plus générique

```
let rec sort lt = function
  | [] → []
  | x :: l → insert lt x (sort lt l)

and insert lt x = function
  | [] →
      [x]
  | y :: l →
      if lt x y then x :: y :: l else y :: insert lt x l
```

```
val sort : ('a → 'a → bool) → 'a list → 'a list
val insert : ('a → 'a → bool) → 'a → 'a list → 'a list
```

Fonctions anonymes

la construction `fun x -> e` introduit une fonction anonyme

exemple :

```
sort (fun x y → x <= y) [3;2;7;1;6]
```

le corps `e` peut dépendre de variables locales
$\Rightarrow$ strictement plus puissant que les pointeurs de fonctions

utilisation typique : les itérateurs

Fonctions anonymes

la construction `fun x -> e` introduit une fonction anonyme

exemple :

```
sort (fun x y → x <= y) [3;2;7;1;6]
```

le corps `e` peut dépendre de variables locales
⇒ strictement plus puissant que les pointeurs de fonctions

utilisation typique : les itérateurs

Fonctions anonymes

la construction `fun x -> e` introduit une fonction anonyme

exemple :

```
sort (fun x y → x <= y) [3;2;7;1;6]
```

le corps `e` peut dépendre de variables locales
$\Rightarrow$ strictement plus puissant que les pointeurs de fonctions

utilisation typique : les itérateurs

Fonctions anonymes

la construction `fun x -> e` introduit une fonction anonyme

exemple :

```
sort (fun x y → x <= y) [3;2;7;1;6]
```

le corps `e` peut dépendre de variables locales
$\Rightarrow$ strictement plus puissant que les pointeurs de fonctions

utilisation typique : les itérateurs

Les itérateurs

en Java, on écrit typiquement

```
for (Iterator i = t.iterator(); i.hasNext(); ) {
  ... traiter i.next() ...
}
```

en Ocaml, on utilise une fonction telle que

```
List.iter : ('a → unit) → 'a list → unit
```

exemple :

```
let print l =
  let n = List.length l in
  List.iter (fun x → printf "%d" (x/n)) l
```

Les itérateurs

en Java, on écrit typiquement

```
for (Iterator i = t.iterator(); i.hasNext(); ) {
  ... traiter i.next() ...
}
```

en Ocaml, on utilise une fonction telle que

```
List.iter : ('a → unit) → 'a list → unit
```

exemple :

```
let print l =
  let n = List.length l in
  List.iter (fun x → printf "%d" (x/n)) l
```

Les itérateurs

en Java, on écrit typiquement

```
for (Iterator i = t.iterator(); i.hasNext(); ) {
  ... traiter i.next() ...
}
```

en Ocaml, on utilise une fonction telle que

```
List.iter : ('a → unit) → 'a list → unit
```

exemple :

```
let print l =
  let n = List.length l in
  List.iter (fun x → printf "%d" (x/n)) l
```

Plus court chemin dans un graphe

```
let shortest_path succ v1 v2 =
  let visited = Hashtbl.create 97 in
  let q = Prqueue.create () in
  Prqueue.add (0, v1) q;
  while true do
    if Prqueue.is_empty q then raise Not_found;
    let (w,v) = Prqueue.min q in
    if v = v2 then begin
      Printf.printf "chemin de longueur %d\n" w; exit 0
    end else if not (Hashtbl.mem visited v) then begin
      Hashtbl.add visited v ();
      succ v (fun vv → Prqueue.add (w+1, vv) q)
    end
  done
```

```
: ('a → ('a → unit) → unit) → 'a → 'a → unit
```

Plus court chemin dans un graphe

```
let shortest_path succ v1 v2 =
  let visited = Hashtbl.create 97 in
  let q = Prqueue.create () in
  Prqueue.add (0, v1) q;
  while true do
    if Prqueue.is_empty q then raise Not_found;
    let (w,v) = Prqueue.min q in
    if v = v2 then begin
      Printf.printf "chemin de longueur %d\n" w; exit 0
    end else if not (Hashtbl.mem visited v) then begin
      Hashtbl.add visited v ();
      succ v (fun vv → Prqueue.add (w+1, vv) q)
    end
  done
```

```
: ('a → ('a → unit) → unit) → 'a → 'a → unit
```

Partition d'une liste

```
let partition f l =
  let rec part ((l1,l2) as acc) = function
    | [] → acc
    | x :: l → part (if f x then x::l1,l2 else l1,x::l2) l
  in
  part ([],[]) l

let rec fold_left f acc = function
  | [] → acc
  | x :: l → fold_left f (f acc x) l

let partition f =
  List.fold_left
    (fun (l1,l2) x → if f x then x::l1,l2 else l1,x::l2)
    ([],[])
```

Partition d'une liste

```
let partition f l =
  let rec part ((l1,l2) as acc) = function
    | [] → acc
    | x :: l → part (if f x then x::l1,l2 else l1,x::l2) l
  in
  part ([],[]) l

let rec fold_left f acc = function
  | [] → acc
  | x :: l → fold_left f (f acc x) l

let partition f =
  List.fold_left
    (fun (l1,l2) x → if f x then x::l1,l2 else l1,x::l2)
    ([],[])
```

Partition d'une liste

```
let partition f l =
  let rec part ((l1,l2) as acc) = function
    | [] → acc
    | x :: l → part (if f x then x::l1,l2 else l1,x::l2) l
  in
  part ([],[]) l

let rec fold_left f acc = function
  | [] → acc
  | x :: l → fold_left f (f acc x) l

let partition f =
  List.fold_left
    (fun (l1,l2) x → if f x then x::l1,l2 else l1,x::l2)
    ([],[])
```

Modules et foncteurs

Modules

les **modules** d'Ocaml sont une réponse aux problèmes

- de **modularité** : découpage du code, espace de noms
  - chaque fichier est un module
  - modules dans les modules (`module M = struct ... end`)
- d'**encapsulation** : occulter la représentation de certains types
  - l'interface d'un module exporte sa partie publique (`module M : sig ... end = ...`)
- de **généricité** : éviter au mieux la duplication de code
  - fonctions des modules vers les modules (les **foncteurs**)

Modules

les **modules** d'Ocaml sont une réponse aux problèmes

- de **modularité** : découpage du code, espace de noms
  - chaque fichier est un module
  - modules dans les modules (`module M = struct ... end`)
- d'**encapsulation** : occulter la représentation de certains types
  - l'interface d'un module exporte sa partie publique (`module M : sig ... end = ...`)
- de **généricité** : éviter au mieux la duplication de code
  - fonctions des modules vers les modules (les **foncteurs**)

Modules

les **modules** d'Ocaml sont une réponse aux problèmes

- de **modularité** : découpage du code, espace de noms
  - chaque fichier est un module
  - modules dans les modules (`module M = struct ... end`)
- d'**encapsulation** : occulter la représentation de certains types
  - l'interface d'un module exporte sa partie publique (`module M : sig ... end = ...`)
- de **généricité** : éviter au mieux la duplication de code
  - fonctions des modules vers les modules (les **foncteurs**)

Exemple de module : les tables de hachage

```
module Hashtbl : sig

  type ('a, 'b) t

  val create : int → ('a, 'b) t

  val add : ('a, 'b) t → 'a → 'b → unit

  val find : ('a, 'b) t → 'a → 'b
  ...

  val iter : ('a → 'b → unit) → ('a, 'b) t → unit
  ...
end
```

Tables de hachage génériques

tables de hachage **génériques** i.e. paramétrées par la fonction de hachage et la fonction d'égalité

- passer les fonctions en argument ?

```
val add : ('a → int) → ('a,'b) t → 'a → 'b → unit
val find : ('a → int) → ('a → 'a → bool) → ...
```

  ⇒ propice aux incohérences

- passer les fonctions à la création ?
  ⇒ nécessite de les stocker dans la structure de données

- la bonne solution est **un foncteur** (≈ template C++)

Tables de hachage génériques

tables de hachage **génériques** i.e. paramétrées par la fonction de hachage et la fonction d'égalité

- passer les fonctions en argument ?

```
val add : ('a → int) → ('a,'b) t → 'a → 'b → unit
val find : ('a → int) → ('a → 'a → bool) → ...
```

  ⇒ propice aux incohérences

- passer les fonctions à la création ?
  ⇒ nécessite de les stocker dans la structure de données

- la bonne solution est **un foncteur** (≈ template C++)

Tables de hachage génériques

tables de hachage **génériques** i.e. paramétrées par la fonction de hachage et la fonction d'égalité

- passer les fonctions en argument ?

  ```
  val add : ('a → int) → ('a,'b) t → 'a → 'b → unit
  val find : ('a → int) → ('a → 'a → bool) → ...
  ```

  ⇒ propice aux incohérences

- passer les fonctions à la création ?
  ⇒ nécessite de les stocker dans la structure de données

- la bonne solution est **un foncteur** (≈ template C++)

Tables de hachage génériques

tables de hachage **génériques** i.e. paramétrées par la fonction de hachage et la fonction d'égalité

- passer les fonctions en argument ?

```
val add : ('a → int) → ('a,'b) t → 'a → 'b → unit
val find : ('a → int) → ('a → 'a → bool) → ...
```

  ⇒ propice aux incohérences

- passer les fonctions à la création ?
  ⇒ nécessite de les stocker dans la structure de données

- la bonne solution est **un foncteur** (≈ template C++)

Tables de hachage génériques

```
module Hashtbl.Make
  (X : sig type elt
           val hash : elt → int
           val equal : elt → elt → bool end)
  : sig
      type 'a t
      val create : int → 'a t
      val add : 'a t → X.elt → 'a → unit
      val find : 'a t → X.elt → 'a
      ...
    end
```

Tables de hachage génériques

```
module Hashtbl.Make
  (X : sig type elt
           val hash : elt → int
           val equal : elt → elt → bool end)
  : sig
      type 'a t
      val create : int → 'a t
      val add : 'a t → X.elt → 'a → unit
      val find : 'a t → X.elt → 'a
      ...
    end
```

Codage

```
module Make(X : ...) = struct

  type t = X.elt list array

  let create n = Array.create n []

  let add t x =
    let i = (X.hash x) mod (Array.length t) in
    t.(i) <- x :: t.(i)

  ...

end
```

Arbres équilibrés

```
module Set.Make
  (X : sig type elt
           val compare : elt → elt → int end)
  : sig
      type t
      val empty : t
      val add : X.elt → t → t
      val union : t → t → t
      ...
    end

module S = Set.Make(String)
```

Arbres équilibrés

```
module Set.Make
  (X : sig type elt
           val compare : elt → elt → int end)
  : sig
      type t
      val empty : t
      val add : X.elt → t → t
      val union : t → t → t
      ...
    end

module S = Set.Make(String)
```

Foncteur pour le plus court chemin

```
module Make
  (G : sig
         type node
         val succ : node → (node → unit) → unit
       end)
  =
  struct

    let shortest_path v1 v2 =

      ... G.succ ...

  end
```

Persistance

Persistance

la plupart des structures de données sont **immuables** :

- une valeur n'est pas affectée par une opération
- mais une **nouvelle** valeur est retournée

```
let l = [1; 2; 3]
```



```
let l' = 0 :: l
```

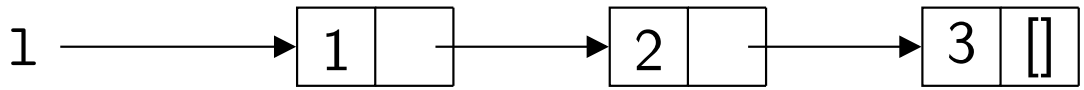

pas de copie, mais **partage**

Persistance

la plupart des structures de données sont **immuables** :

- une valeur n'est pas affectée par une opération
- mais une **nouvelle** valeur est retournée

```
let l = [1; 2; 3]
```




```
let l' = 0 :: l
```

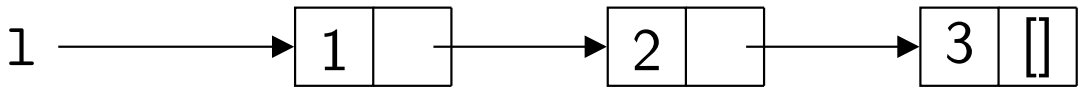

pas de copie, mais **partage**

Persistance

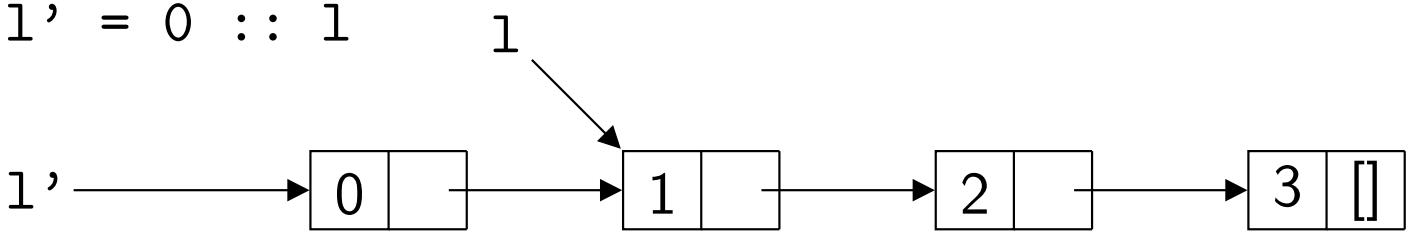
la plupart des structures de données sont **immuables** :

- une valeur n'est pas affectée par une opération
- mais une **nouvelle** valeur est retournée

```
let l = [1; 2; 3]
```




```
let l' = 0 :: l
```



pas de copie, mais **partage**

Concaténation de deux listes

```
let rec append l1 l2 = match l1 with
  | [] → l2
  | x :: l → x :: append l l2
```

```
let l = [1; 2; 3]
let l' = [4; 5]
let l'' = append l l '
```


blocs de `l` **copiés**, blocs de `l'` **partagés**

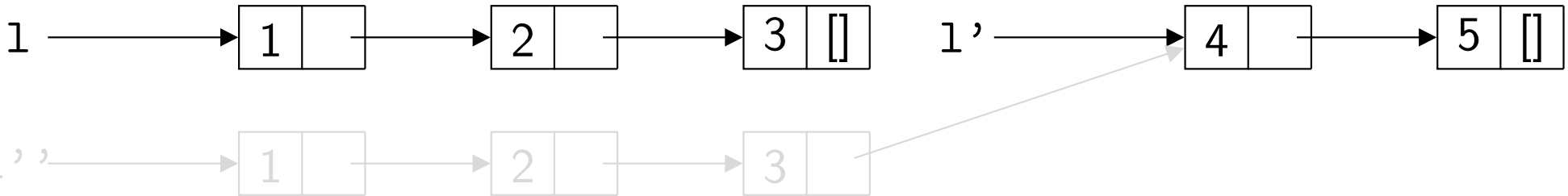
Concaténation de deux listes

```
let rec append l1 l2 = match l1 with
  | [] → l2
  | x :: l → x :: append l l2

let l = [1; 2; 3]
let l' = [4; 5]
let l'' = append l l '
```


blocs de `l` **copiés**, blocs de `l'` **partagés**

Concaténation de deux listes

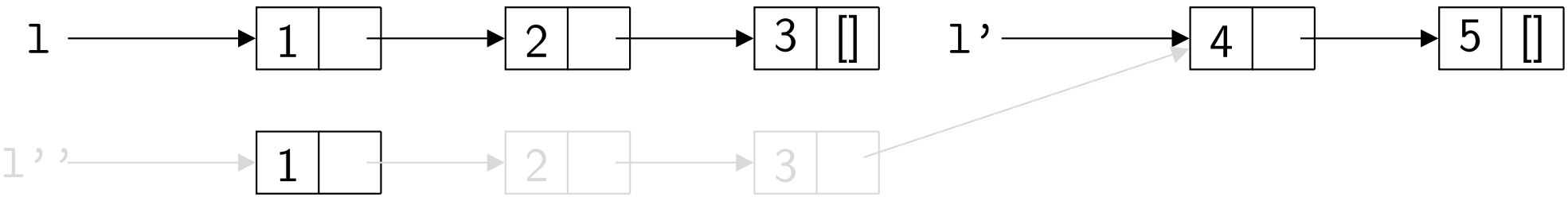
```
let rec append l1 l2 = match l1 with
  | [] → l2
  | x :: l → x :: append l l2

let l = [1; 2; 3]
let l' = [4; 5]
let l'' = append l l '
```



blocs de `l` **copiés**, blocs de `l'` **partagés**

Concaténation de deux listes

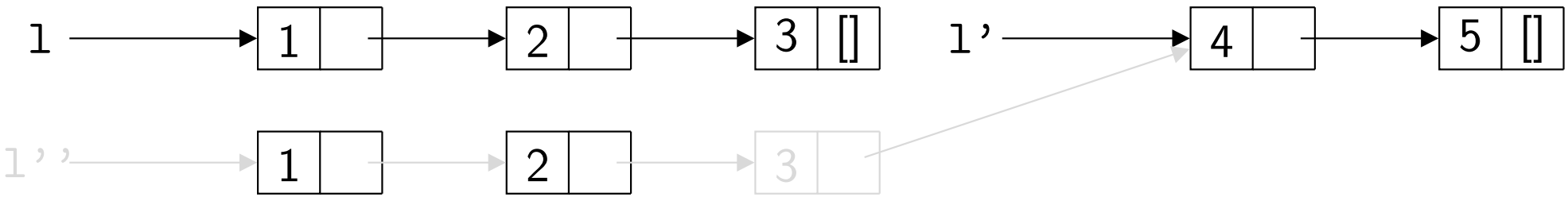
```
let rec append l1 l2 = match l1 with
  | [] → l2
  | x :: l → x :: append l l2

let l = [1; 2; 3]
let l' = [4; 5]
let l'' = append l l '
```



blocs de `l` **copiés**, blocs de `l'` **partagés**

Concaténation de deux listes

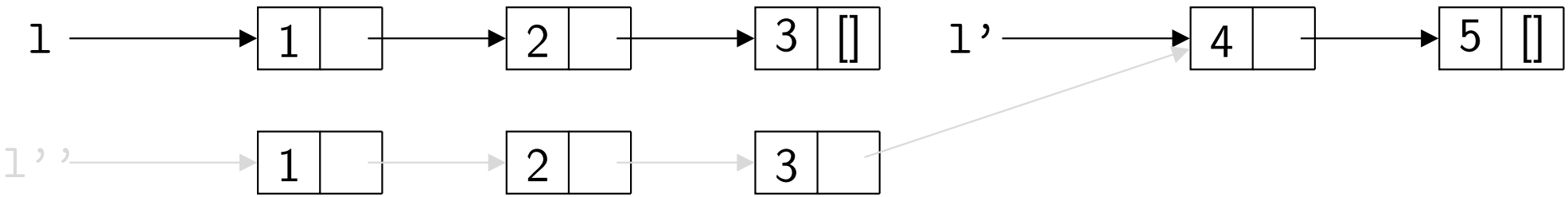
```
let rec append l1 l2 = match l1 with
  | [] → l2
  | x :: l → x :: append l l2

let l = [1; 2; 3]
let l' = [4; 5]
let l'' = append l l '
```



blocs de `l` **copiés**, blocs de `l'` **partagés**

Concaténation de deux listes

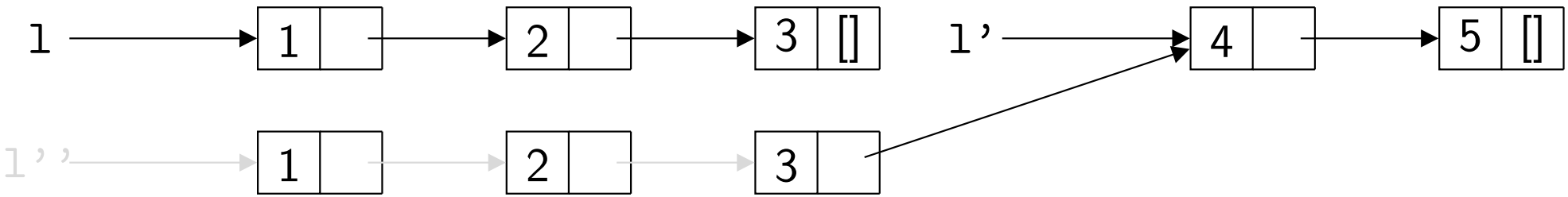
```
let rec append l1 l2 = match l1 with
  | [] → l2
  | x :: l → x :: append l l2

let l = [1; 2; 3]
let l' = [4; 5]
let l'' = append l l '
```



blocs de `l` **copiés**, blocs de `l'` **partagés**

Concaténation de deux listes

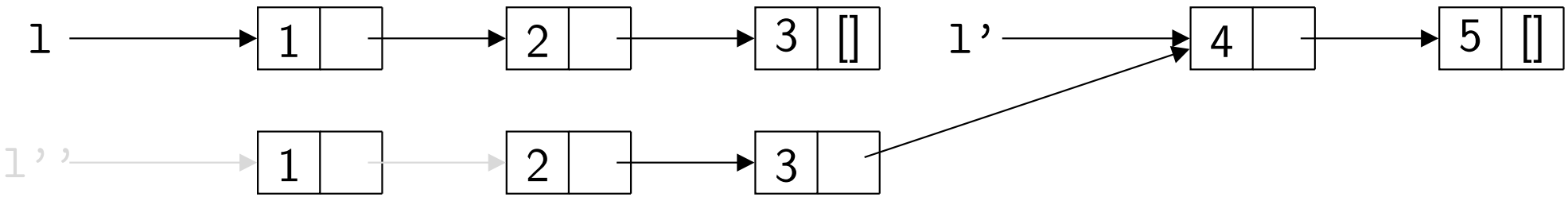
```
let rec append l1 l2 = match l1 with
  | [] → l2
  | x :: l → x :: append l l2

let l = [1; 2; 3]
let l' = [4; 5]
let l'' = append l l '
```



blocs de `l` **copiés**, blocs de `l'` **partagés**

Concaténation de deux listes

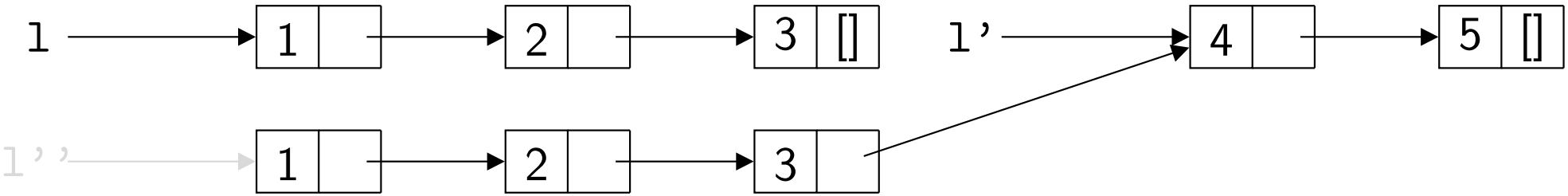
```
let rec append l1 l2 = match l1 with
  | [] → l2
  | x :: l → x :: append l l2

let l = [1; 2; 3]
let l' = [4; 5]
let l'' = append l l '
```



blocs de `l` **copiés**, blocs de `l'` **partagés**

Concaténation de deux listes

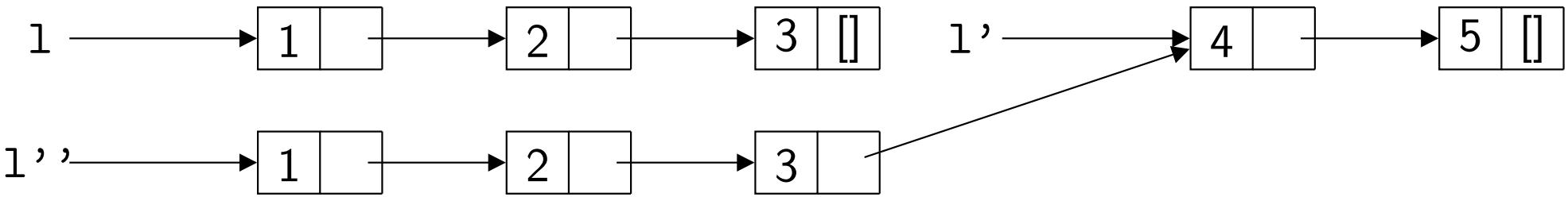
```
let rec append l1 l2 = match l1 with
  | [] → l2
  | x :: l → x :: append l l2

let l = [1; 2; 3]
let l' = [4; 5]
let l'' = append l l '
```



blocs de `l` **copiés**, blocs de `l'` **partagés**

Concaténation de deux listes

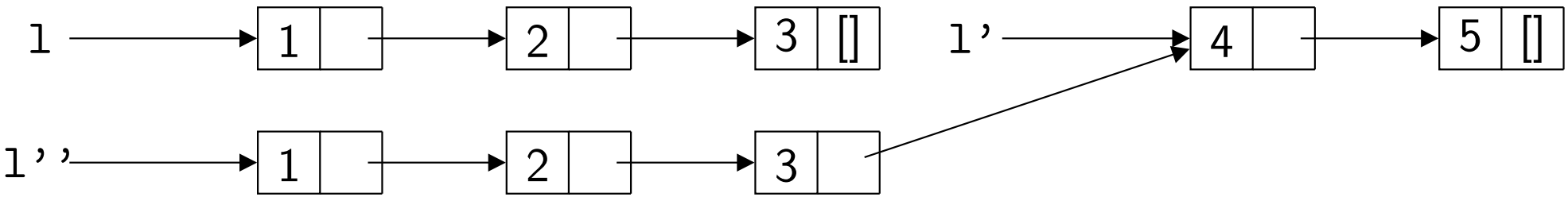
```
let rec append l1 l2 = match l1 with
  | [] → l2
  | x :: l → x :: append l l2

let l = [1; 2; 3]
let l' = [4; 5]
let l'' = append l l '
```



blocs de `l` **copiés**, blocs de `l'` **partagés**

Concaténation de deux listes

```
let rec append l1 l2 = match l1 with
  | [] → l2
  | x :: l → x :: append l l2

let l = [1; 2; 3]
let l' = [4; 5]
let l'' = append l l '
```



blocs de `l` **copiés**, blocs de `l'` **partagés**

Intérêts pratiques de la persistance

1. **correction** des programmes
   - code plus simple
   - raisonnement mathématique possible

2. outil puissant pour le **backtracking**
   - algorithmes de recherche
   - manipulations symboliques et portées
   - rétablissement suite à une erreur

Intérêts pratiques de la persistance

1. **correction** des programmes
   - code plus simple
   - raisonnement mathématique possible

2. outil puissant pour le **backtracking**
   - algorithmes de recherche
   - manipulations symboliques et portées
   - rétablissement suite à une erreur

Persistance et backtracking

exemple : recherche de solution

```
type state (* persistant *)
val success : state → bool
type move
val moves : state → move list
val move : state → move → state

let rec find e =
  success e || attempt e (moves e)
and attempt e = function
  | [] → false
  | d :: r → find (move e d) || attempt e r
```

Persistance et backtracking

exemple : recherche de solution

```
type state (* persistant *)
val success : state → bool
type move
val moves : state → move list
val move : state → move → state

let rec find e =
  success e || attempt e (moves e)
and attempt e = function
  | [] → false
  | d :: r → find (move e d) || attempt e r
```

Persistance et backtracking

autre exemple : **problème de la transaction**

avec une structure **modifiée en place**

```
try
  ... effectuer l'opération de mise à jour ...
with e →
  ... rétablir la base dans un état cohérent ...
  ... traiter ensuite l'erreur ...
```

Persistance et backtracking

autre exemple : **problème de la transaction**

avec une structure **modifiée en place**

```
try
  ... effectuer l'opération de mise à jour ...
with e →
  ... rétablir la base dans un état cohérent ...
  ... traiter ensuite l'erreur ...
```

Persistance et backtracking

avec une structure **persistante**

```
let bd = ref (... base initiale ...)
...
try
  bd := (... opération de mise à jour de !bd ...)
with e →
  ... traiter l'erreur ...
```

Persistance et effets de bords

persistant ne signifie pas sans effet de bord

$$persistant = observationnellement\ immuable$$

on a seulement l'implication dans un sens :

$$purement\ fonctionnel \Rightarrow persistant$$

Note :

- la persistance n'est pas propre à Ocaml
- on y est juste **invité naturellement**

Persistance et effets de bords

persistant ne signifie pas sans effet de bord

$$persistant = observationnellement\ immuable$$

on a seulement l'implication dans un sens :

$$purement\ fonctionnel \Rightarrow persistant$$

Note :

- la persistance n'est pas propre à Ocaml
- on y est juste **invité naturellement**

Persistance et effets de bords

persistant ne signifie pas sans effet de bord

$$persistant = observationnellement\ immuable$$

on a seulement l'implication dans un sens :

$$purement\ fonctionnel \Rightarrow persistant$$

Note :

- la persistance n'est pas propre à Ocaml
- on y est juste **invité naturellement**

Persistance et effets de bords

persistant ne signifie pas sans effet de bord

$$persistant = observationnellement\ immuable$$

on a seulement l'implication dans un sens :

$$purement\ fonctionnel \Rightarrow persistant$$

Note :

- la persistance n'est pas propre à Ocaml
- on y est juste **invité naturellement**

Conclusion

quelques points forts d'Ocaml illustrés au travers de traits idiomatiques

- filtrage
- ordre supérieur et itérateurs
- généricité et foncteurs
- persistance

beaucoup d'autres aspects encore

- objets
- variants polymorphes
- arguments nommés, optionnels
- ...

Conclusion

quelques points forts d'Ocaml illustrés au travers de traits idiomatiques

- filtrage
- ordre supérieur et itérateurs
- généricité et foncteurs
- persistance

beaucoup d'autres aspects encore

- objets
- variants polymorphes
- arguments nommés, optionnels
- ...

Conclusion

Ocaml est un langage qui

- marie de nombreux styles de programmation
- aide au développement **rapide** de programmes **corrects**