Du fonctionnel à l'impératif en Caml

Un exemple pédagogique, plutôt que
La Pédagogie de l'informatique!

Pierre Weis
Pierre.Weis@inria.fr

INRIA – 28 avril 1998

PLAN

1. Le tri des listes.

2. Le tri des vecteurs.

3. Passage de l'un à l'autre par réécriture de programme.

Considérations sur la récursivité

let prog(n) = ... prog(n - 1) ...

Difficile: équivalent à prouver par récurrence.

Programmation: descente infinie à la Fermat.

Intuitivement: utilise un programme pas fini.

Pas de “si P(n)”: utilisation effective.

Considérations impératif-fonctionnel

Fonctionnel: calcul d'un résultat

Impératif: obtenir un état de la machine

Fonctionnel: difficile, il faut programmer un calcul complet, obtenir le résultat directement

Impératif: difficile, il faut programmer des modifications successives, mais on avance pas à pas en s'approchant du résultat

Considérations impératif-fonctionnel

Impératif: le temps intervient, mais on détaille la suite des opérations

Fonctionnel: le temps n'intervient pas, la suite des opérations est implicite

Fonctionnel: preuve simple (récurrence sur la structure)

Impératif: preuve complexe (invariants de boucles)

Tri fonctionnel des listes

Filtrage:

```
let rec tri = function
 | [] ->
 | x :: l ->
;;
```

Anticipation:

```
let rec tri = function
 | [] -> []
 | x :: l -> ... tri l
;;
```

Délégation:

```
let rec tri = function
 | [] -> []
 | x :: l -> insert x (tri l)
;;
```

Insertion

Filtrage:

```
let rec insert e = function
 | [] ->
 | x :: l ->
;;
```

Anticipation:

```
let rec insert e = function
 | [] -> [e]
 | x :: l -> ... insert e l
;;
```

Programmer c'est prouver: résultat trié ?

```
let rec insert e = function
 | [] -> [e]
 | x :: l ->
   if e < x then e :: x :: l else x :: insert e l
;;
```

Tri impératif des vecteurs

Le prof: tri par insertion sur les vecteurs

```
let tri v =
 let len = vect_length v in
 for i = len - 2 downto 0 do
  let x = v.(i)
  and k = ref (i + 1) in
  while !k < len && x > v.(!k) do
   v.(!k - 1) <- v.(!k);
   k := !k + 1
  done;
  v.(!k - 1) <- x
 done;;
```

Le prof: c'est la même chose que pour les listes, mais la preuve de correction est plus difficile.

Tri impératif des vecteurs

L'élève: comment ça la même chose ???

```
let rec tri = function
 | [] -> []
 | x :: l -> insert x (tri l)

and insert e = function
 | [] -> [e]
 | x :: l ->
   if e < x then e :: x :: l else x :: insert e l;;


let tri v =
 let len = vect_length v in
 for i = len - 2 downto 0 do
  let x = v.(i)
  and k = ref (i + 1) in
  while !k < len && x > v.(!k) do
   v.(!k - 1) <- v.(!k);
   k := !k + 1
  done;
  v.(!k - 1) <- x
 done;;
```

Motivations

Prouver l'équivalence
Calculer sur les programmes
Comprendre leur sémantique

Méthode: dériver le programme impératif

Étapes:

- Traduction :
  - Type : l'algorithme fonctionnel sur les vecteurs
  - Filtrage : expansion en tests élémentaires
- Réécriture : introduction des boucles, simplifications (nécessite des démonstrations et de la sémantique).

Fonctionnel expansé

Expansion du filtrage: null hd tl

Expansion des primitives: cons

```
let null = function
 | [] -> true | _ -> false;;

let cons x l = x :: l;;


let rec tri list =
 (* | [] -> *)
 if null list then [] else
 (* | x :: l -> *)
 let x = hd list
 and l = tl list in
 insert x (tri l)

and insert x list =
 if null list then cons x [] else
 let y = hd list
 and l = tl list in
 if x <= y then x :: y :: l else y :: (insert x l);;
```

Primitives de filtrages sur vecteurs

Pas de filtrage sur les vecteurs:

On écrit les primitives de filtrage.

```
let null_vect v = vect_length v = 0;;

let hd_vect v =
 if null_vect v then failwith "hd_vect" else v.(0);;

let tl_vect v =
 if null_vect v then failwith "tl_vect"
 else sub_vect v 1 (vect_length v - 1);;

let cons_vect x v =
 let res = make_vect (vect_length v + 1) x in
 for i = 0 to vect_length v - 1 do
  res.(i + 1) <- v.(i)
 done;
 res;;

hd_vect : 'a vect -> 'a = <fun>
tl_vect : 'a vect -> 'a vect = <fun>
cons_vect : 'a -> 'a vect -> 'a vect = <fun>
```

Tri fonctionnel sur vecteur

```
let rec tri v =
 if null_vect v then [||] else
 let x = hd_vect v and l = tl_vect v in
 insert x (tri l)

and insert x v =
 if null_vect v then cons_vect x [||] else
 let y = hd_vect v and l = tl_vect v in
 if x <= y then cons_vect x v
 else cons_vect y (insert x l);;

tri : 'a vect -> 'a vect = <fun>
insert : 'a -> 'a vect -> 'a vect = <fun>

let v = [|1;0;5;6;2;4;3|];;
v : int vect = [|1; 0; 5; 6; 2; 4; 3|]

tri v;;
- : int vect = [|0; 1; 2; 3; 4; 5; 6|]

v;;
- : int vect = [|1; 0; 5; 6; 2; 4; 3|];;
```

Bilan

L'élève: même programme que pour les listes, d'accord

Mais n'opère pas en place!

Le prof: Il suffit de changer les primitives, sans changer l'algorithme, ni même le programme

Trier en place

Idée: les primitives opèrent en place

`cons` n'alloue pas

`cons x v` écrit `x` dans `v`, rend `v` ?

En fait, on alloue dans le vecteur
$\longrightarrow$ pointeur d'allocation explicite
$\longrightarrow$ sous-vecteur: support, indice de départ
$\longrightarrow$ `(v, i)` = `sub_vect v i (vect_length v - i)`
$\longrightarrow$ `(v, 0)` = `v`

`Cons x (v, i)` rend `(v, i - 1)`

Trier en place

Réécriture de cons:

```
let cons_vect x v =
 let res = make_vect (vect_length v + 1) x in
 for i = 0 to vect_length v - 1 do
  res.(i + 1) <- v.(i)
 done;
 res;;
```

Allocation en place:

test + affectation + changement d'indice

$\Rightarrow$

```
let cons_vect x (v, i) =
 if i <= 0 then failwith "cons_vect" else
 begin
  v.(i - 1) <- x;
  v, i - 1
 end;;

cons_vect : int -> 'a vect * int -> 'a vect * int = <fun>
```

On réécrit les autres primitives

Réécriture de null:

```
   let null_vect v = vect_length v = 0;;
```

$\Rightarrow$

```
let null_vect (v, i) = i >= vect_length v;;

null_vect : 'a vect * int -> bool = <fun>
```

Réécriture de hd:

```
let hd_vect v =
 if null_vect v then failwith "hd_vect"
 else v.(0);;
```

⇒

```
let hd_vect (v, i) =
 if null_vect (v, i) then failwith "hd_vect"
 else v.(i);;
hd_vect : 'a vect * int -> 'a = <fun>
```

Réécriture de tl:

```
let tl_vect v =
 if null_vect v then failwith "tl_vect"
 else sub_vect v 1 (vect_length v - 1);;
```

⇒

```
let tl_vect (v, i) =
 if null_vect (v, i) then failwith "tl_vect"
 else v, i + 1;;
tl_vect : 'a vect * int -> 'a vect * int = <fun>
```

Que devient le tri?

```
let rec tri v =
 if null_vect v then v else
 let x = hd_vect v and l = tl_vect v in
 insert x (tri l)

and insert x v =
 if null_vect v then cons_vect x v else
 let y = hd_vect v and l = tl_vect v in
 if x <= y then cons_vect x v
 else cons_vect y (insert x l);;

tri : 'a vect * int -> 'a vect * int = <fun>
insert : 'a -> 'a vect * int -> 'a vect * int = <fun>

tri ([|1;0;5;6;2;4;3|], 0);;
- : int vect * int = [|0; 1; 2; 3; 4; 5; 6|], 0

let v = [|1;0;5;6;2;4;3|];;
v : int vect = [|1; 0; 5; 6; 2; 4; 3|]

tri (v, 0);;
- : int vect * int = [|0; 1; 2; 3; 4; 5; 6|], 0

v;;
- : int vect = [|0; 1; 2; 3; 4; 5; 6|]
```

Impératif

Tri en place: le résultat est l'argument.

Introduction de l'appel initial à la procédure.

Définition locale des primitives.

```
let tri v =

 let null_vect (v, i) =
 let hd_vect (v, i) =
 let tl_vect (v, i) =
 let cons_vect x (v, i) =

 let rec tri v =
 and insert x v =
 in
 tri (v, 0);
 v;;

tri : 'a vect -> 'a vect = <fun>
```

Le tri sur vecteur

```
let tri v =

 let null_vect (v, i) = i >= vect_length v in
 let tl_vect (v, i) =
  if null_vect (v, i) then failwith "tl_vect"
  else v, i + 1
 and hd_vect (v, i) =
  if null_vect (v, i) then failwith "hd_vect"
  else v.(i)
 and cons_vect x (v, i) =
  if i <= 0 then failwith "cons_vect" else
  begin v.(i - 1) <- x; v, i - 1 end in

 let rec tri v =
  if null_vect v then v else
  let x = hd_vect v and l = tl_vect v in
  insert x (tri l)
 and insert x v =
  if null_vect v then cons_vect x v else
  let y = hd_vect v and l = tl_vect v in
  if x <= y then cons_vect x v
  else cons_vect y (insert x l) in

 (* Trier le vecteur complet *)
 tri (v, 0);
 v;;
```

Simplifications

v est invariant: variable globale aux fonctions
argument inutile

```
let tri v =

 let null_vect i =
 let hd_vect i =
 let tl_vect i =
 let cons_vect x i =

 let rec tri v =
 and insert x v =
 in
 tri 0;
 v;;

Entrée interactive:
> tri 0; v;;
> ^^^^^
Attention: cette expression est de type int,
mais est utilisée avec le type unit.
tri : 'a vect -> 'a vect = <fun>
```

Simplifications

Renommage des arguments et primitives
longueur du vecteur: constante nommée

```
let tri v =
 let len = vect_length v in

 let null i = i >= len
 and tl i = if i >= len then failwith "tl" else i + 1
 and hd i = if i >= len then failwith "hd" else v.(i)
 and cons x i =
  if i <= 0 then failwith "cons" else
  begin v.(i - 1) <- x; i - 1 end in

 let rec tri i =
  if null i then i else
  let x = hd i and l = tl i in insert x (tri l)

 and insert x j =
  if null j then cons x j else
  let y = hd j and l = tl j in
  if x <= y then cons x j
  else cons y (insert x l) in

 tri 0; v;;
tri : 'a vect -> 'a vect = <fun>
```

Expansion en ligne des primitives

```
let tri v =
 let len = vect_length v in

 let null i = i >= len
 and hd ... and tl ... and cons ... in

 let rec tri i =
  if i >= len then i else
  let x = (* hd i *)
   if i >= len then failwith "hd" else v.(i)
  and l = (* tl i *)
   if i >= len then failwith "tl" else i + 1 in
  insert x (tri l)

 and insert x j =
  if j >= len then cons x j else
  let y = (* hd j *)
   if j >= len then failwith "hd" else v.(j)
  and l = (* tl j *)
   if j >= len then failwith "tl" else j + 1 in
  if x <= y then cons x j
  else cons y (insert x l) in

 tri 0; v;;
```

Simplification

Définitions inutiles: null, hd tl

Tests redondants: `i >= len`

```
let tri v =
 let len = vect_length v in

 let cons x i =
  if i <= 0 then failwith "cons" else
  begin v.(i - 1) <- x; i - 1 end in

 let rec tri i =
  if i >= len then i else
  let x = (* hd i *) v.(i)
  and l = (* tl i *) i + 1 in
  insert x (tri l)

 and insert x j =
  if j >= len then cons x j else
  let y = (* hd j *) v.(j)
  and l = (* tl j *) j + 1 in
  if x <= y then cons x j
  else cons y (insert x l) in

 tri 0; v;;
```

Expansion des expressions pures

```
let l = i + 1 in ... tri l
```

$\longrightarrow$ `tri (i + 1)`

```
let tri v =
 let len = vect_length v in
 let cons = ... in

 let rec tri i =
  if i >= len then i else
  let x = v.(i) in
  insert x (tri (i + 1))

 and insert x j =
  if j >= len then cons x j else
  let y = v.(j)
  (* and l = j + 1 *) in
  if x <= y then cons x j
  else cons y (insert x (j + 1)) in

 tri 0; v;;
```

Expansion de cons

```
cons x j ==
if j <= 0 then failwith "cons" else
begin v.(j - 1) <- x; j - 1 end
```

```
let tri v =
 let len = ... in
 let cons = ... in

 let rec tri i =
  if i >= len then i else
  let x = v.(i) in
  insert x (tri (i + 1))

 and insert x j =
  if j >= len then (* cons x j *)
   if j <= 0 then failwith "cons" else
    begin v.(j - 1) <- x; j - 1 end
  else
  let y = v.(j) in
  if x <= y then cons x j
  else cons y (insert x (j + 1)) in

 tri 0; v;;
```

Simplification

`insert` n'est jamais appelée avec $j = 0$

$j > 0$, on supprime `if j <= 0 ...`

```
let tri v =
 let len = ... in
 let cons = ... in

 let rec tri i =
  if i >= len then i else
  let x = v.(i) in
  insert x (tri (i + 1))

 and insert x j =
  if j >= len then (* cons x j *)
    begin v.(j - 1) <- x; j - 1 end
  else
  let y = v.(j) in
  if x <= y then (* cons x j *)
    begin v.(j - 1) <- x; j - 1 end
  else cons y (insert x (j + 1)) in

 tri 0; v;;
```

Expansion du dernier cons

On a:

```
cons y (insert x (j + 1)) ==
 let k = insert x (j + 1) in
 cons y k
```

Mais, par définition:

```
 cons y k ==
  if k <= 0 then failwith "cons" else
  begin v.(k - 1) <- y; k - 1 end
```

Donc:

```
cons y (insert x (j + 1)) ==
 let k = insert x (j + 1) in
  if k <= 0 then failwith "cons" else
  begin v.(k - 1) <- y; k - 1 end
```

Expansion du dernier cons

D'où:

```
let tri v =
 let len = vect_length v in

 let rec tri i =
  if i >= len then i else
  let x = v.(i) in
  insert x (tri (i + 1))

 and insert x j =
  if j >= len then
   begin v.(j - 1) <- x; j - 1 end else
  let y = v.(j) in
  if x <= y then
   begin v.(j - 1) <- x; j - 1 end else
  let k = insert x (j + 1) in
  if k <= 0 then failwith "cons" else
  begin v.(k - 1) <- y; k - 1 end in

 tri 0; v;;
```

Simplification

Suppression du test if $k <= 0$, car $k >= 1$

Lemme: tri $i = i$ et insert x $j = j - 1$.

$k =$ insert x $(j + 1)$ donc $k = j$ mais $j >= 1$

```
let tri v =
 let len = vect_length v in

 let rec tri i =
  if i >= len then i else
  let x = v.(i) in
  insert x (tri (i + 1))

 and insert x j =
  if j >= len then begin v.(j - 1) <- x; j - 1 end else
  let y = v.(j) in
  if x <= y then begin v.(j - 1) <- x; j - 1 end else
  let k = insert x (j + 1) in
  begin v.(k - 1) <- y; k - 1 end in

 tri 0; v;;
```

Substitution des résultats de fonctions

Une fonction produit un résultat *et* un effet

Pas d'effet: on peut remplacer par le résultat
Avec effet: il faut produire l'effet, puis remplacer par le résultat

Si $f(x)$ vaut e, alors:
f sans effet $\Rightarrow f(x) == e$
f a un effet $\Rightarrow f(x) == f(x); e$

Application: tri $i = i$, donc
```
insert x (tri (i + 1)) ==
tri (i + 1); insert x (i + 1)
```

Substitution des résultats de fonctions

```
let tri v =
 let len = ...

 let rec tri i =
  if i >= len then i else
  let x = v.(i) in
  insert x (tri (i + 1))
 ...
```

$\Rightarrow$

```
let tri v =
 let len = ...

 let rec tri i =
  if i >= len then i else
  let x = v.(i) in
  tri (i + 1); insert x (i + 1)
 ...
```

Substitution d'un mutable

On supprime la définition de x

(Dém: `v.(i)` n'est pas modifié par `tri (i + 1)`)

```
...
let rec tri i =
 if i >= len then i else
 let x = v.(i) in
 tri (i + 1); insert x (i + 1)
...
```

$\Rightarrow$

```
let rec tri i =
 if i >= len then i else
 begin tri (i + 1); insert v.(i) (i + 1) end
...
```

Vers le procédural complet

Les fonctions ne rendent plus de résultats

On recalculerait leur résultat au besoin.

```
 ...
 let rec tri i =
  if i >= len then () else
  begin tri (i + 1); insert v.(i) (i + 1) end

 and insert x j =
  if j >= len then v.(j - 1) <- x else
  let y = v.(j) in
  if x <= y then v.(j - 1) <- x else
(* let k = insert x (j + 1) in
   begin v.(k - 1) <- y; k - 1 end *)
  begin insert x (j + 1); v.(j - 1) <- y end in

 tri 0; v;;
```

Déplacement de définition

On expanse aussi y dans `insert`

On repousse sa définition

```
...
and insert x j =
 if j >= len then v.(j - 1) <- x else
 let y = v.(j) in
 if x <= y then v.(j - 1) <- x else
 begin insert x (j + 1); v.(j - 1) <- y end in
 ...
```

$\Rightarrow$

```
and insert x j =
 if j >= len then v.(j - 1) <- x else
 if x <= v.(j) then v.(j - 1) <- x else
 let y = v.(j) in
 begin insert x (j + 1); v.(j - 1) <- y end in
 ...
```

Déplacement d'instructions

Lemme insert x j n'affecte que les éléments d'indice $i >= j - 1$

Donc insert x $(j + 1)$ n'affecte pas v.$(j - 1)$

On inverse affectation et appel récursif:

```
...
and insert x j =
 if j >= len then v.(j - 1) <- x else
 if x <= v.(j) then v.(j - 1) <- x else
 let y = v.(j) in
 begin insert x (j + 1); v.(j - 1) <- y end in
 ...
```

$\Rightarrow$

```
and insert x j =
 if j >= len then v.(j - 1) <- x else
 if x <= v.(j) then v.(j - 1) <- x else
 let y = v.(j) in
 v.(j - 1) <- y;
 insert x (j + 1) in
...
```

Expansion de mutable

On expanse complètement la définition de y

```
...
and insert x j =
 if j >= len then v.(j - 1) <- x else
 if x <= v.(j) then v.(j - 1) <- x else
 let y = v.(j) in
 v.(j - 1) <- y;
 insert x (j + 1) in
...
```

$\Rightarrow$

```
and insert x j =
 if j >= len then v.(j - 1) <- x else
 if x <= v.(j) then v.(j - 1) <- x else
 begin
  v.(j - 1) <- v.(j);
  insert x (j+1);
 end in
...
```

Factorisation de tests

On factorise les deux premiers tests

```
...
and insert x j =
 if j >= len then v.(j - 1) <- x else
 if x <= v.(j) then v.(j - 1) <- x else
 begin
  v.(j - 1) <- v.(j);
  insert x (j+1);
 end in
...
```

$\Rightarrow$

```
and insert x j =
 if j >= len || x <= v.(j) then v.(j - 1) <- x
 else begin
   v.(j - 1) <- v.(j);
   insert x (j+1);
 end in
...
```

Inversion de tests

De Morgan, not <=, ...

```
...
and insert x j =
 if j >= len || x <= v.(j) then v.(j - 1) <- x
 else begin
   v.(j - 1) <- v.(j);
   insert x (j+1);
 end in
...
```

⇒

```
and insert x j =
 if j < len && x > v.(j)
 then begin
  v.(j - 1) <- v.(j);
  insert x (j + 1)
 end
 else v.(j - 1) <- x in
...
```

Introduction de boucles

```
let rec for_downto body start stop =
  let rec loop i =
   if i > stop then () else begin
    loop (i + 1);
    body i
   end in
  loop start;;
for_downto : (int -> unit) -> int -> int -> unit = <fun>
```

```
 ...
 let rec tri i =
  if i >= len then () else
  begin
   tri (i + 1);
   insert v.(i) (i + 1)
  end
 ...
 tri 0; v;;
```

loop = tri, start = 0, stop = len - 1,

body = `insert v.(i) (i + 1)`

```
 let rec tri () =
  for i = len - 1 downto 0 do
   insert v.(i) (i + 1)
  done
 ...
 tri (); v
```

Introduction de boucles

`insert` comporte une boucle `while`

```
let rec while_loop cond body =
 if cond () then
 begin
  body ();
  while_loop cond body
 end else ();;
```

Pour que la boucle s'arrête, la condition doit être mutable

Introduction de boucles

```
and insert x j =
 if j < len && x > v.(j) then
 begin
  v.(j - 1) <- v.(j);
  insert x (j + 1)
 end else v.(j - 1) <- x in
```

insert =

if cond then begin e1; insert end else e2

insert =

if cond then begin e1; insert end else ();

if not cond then e2

```
and insert x j =
 if j < len && x > v.(j) then
 begin
  v.(j - 1) <- v.(j);
  insert x (j + 1)
 end else v.(j - 1) <- x in
```

On implémente j par une référence k

L'appel récursif consiste à incrémenter cette référence

insert =
while cond do e1; incr k done;
if not cond then e2

Après la boucle cond est fausse, not cond est vraie

insert =
while cond do e1; incr k done;
e2

Introduction de la boucle while

```
and insert x j =
 if j < len && x > v.(j) then
 begin
  v.(j - 1) <- v.(j);
  insert x (j + 1)
 end else v.(j - 1) <- x in
```

$\Rightarrow$

```
and insert x j =
 let k = ref j in
 while !k < len && x > v.(!k) do
  v.(!k - 1) <- v.(!k);
  k := !k + 1
 done;
 v.(!k - 1) <- x in
```

Fusion des procédures

```
 let rec tri () =
  for i = len - 1 downto 0 do
   insert v.(i) (i + 1)
  done

 and insert x j =
  let k = ref j in
  while !k < len && x > v.(!k) do
   v.(!k - 1) <- v.(!k);
   k := !k + 1
  done;
  v.(!k - 1) <- x in
...
```

$\Rightarrow$

```
 let rec tri () =
  for i = len - 1 downto 0 do
   (* insert v.(i) (i + 1) *)
   let x = v.(i)
   and k = ref (i + 1) in
   while !k < len && x > v.(!k) do
    v.(!k - 1) <- v.(!k);
    k := !k + 1
   done;
   v.(!k - 1) <- x
  done in
 tri (); v;;
```

Simplification

```
...
 let rec tri () =
  for i = len - 1 downto 0 do
   ...
  done in
 tri (); v;;
```

La procédure locale tri n'est plus récursive

let f () = e in f () == e

```
...
  for i = len - 1 downto 0 do
   let x = v.(i)
   and k = ref (i + 1) in
   while !k < len && x > v.(!k) do
    v.(!k - 1) <- v.(!k);
    k := !k + 1
   done;
   v.(!k - 1) <- x
  done;
 v;;
```

Procédural complet

Pas de résultat

```
let tri v =
 let len = vect_length v in
  for i = len - 1 downto 0 do
   let x = v.(i)
   and k = ref (i + 1) in
   while !k < len && x > v.(!k) do
    v.(!k - 1) <- v.(!k);
    k := !k + 1
   done;
   v.(!k - 1) <- x
  done;;
```

Optimisation finale

Inutile de commencer à len - 1

```
let tri v =
 let len = vect_length v in
 for i = len - 2 downto 0 do
  let x = v.(i)
  and k = ref (i + 1) in
  while !k < len && x > v.(!k) do
   v.(!k - 1) <- v.(!k);
   k := !k + 1
  done;
  v.(!k - 1) <- x
 done;;
```

Correspond au programme fonctionnel:

```
let rec tri = function
 | x :: y :: l -> insert x (tri (y :: l))
 | l -> l
...
```

Bilan

Bilan: est-ce plus clairement les mêmes algorithmes ?

On a calculé sur les programmes

On a obtenu une preuve de l'algorithme impératif induite par celle de l'algorithme fonctionnel (!)

On a abordé la sémantique des programmes