Le foncteur sonne toujours deux fois

Jean-Christophe Filliâtre & Pierre Letouzey

LRI, 3 juillet 2003

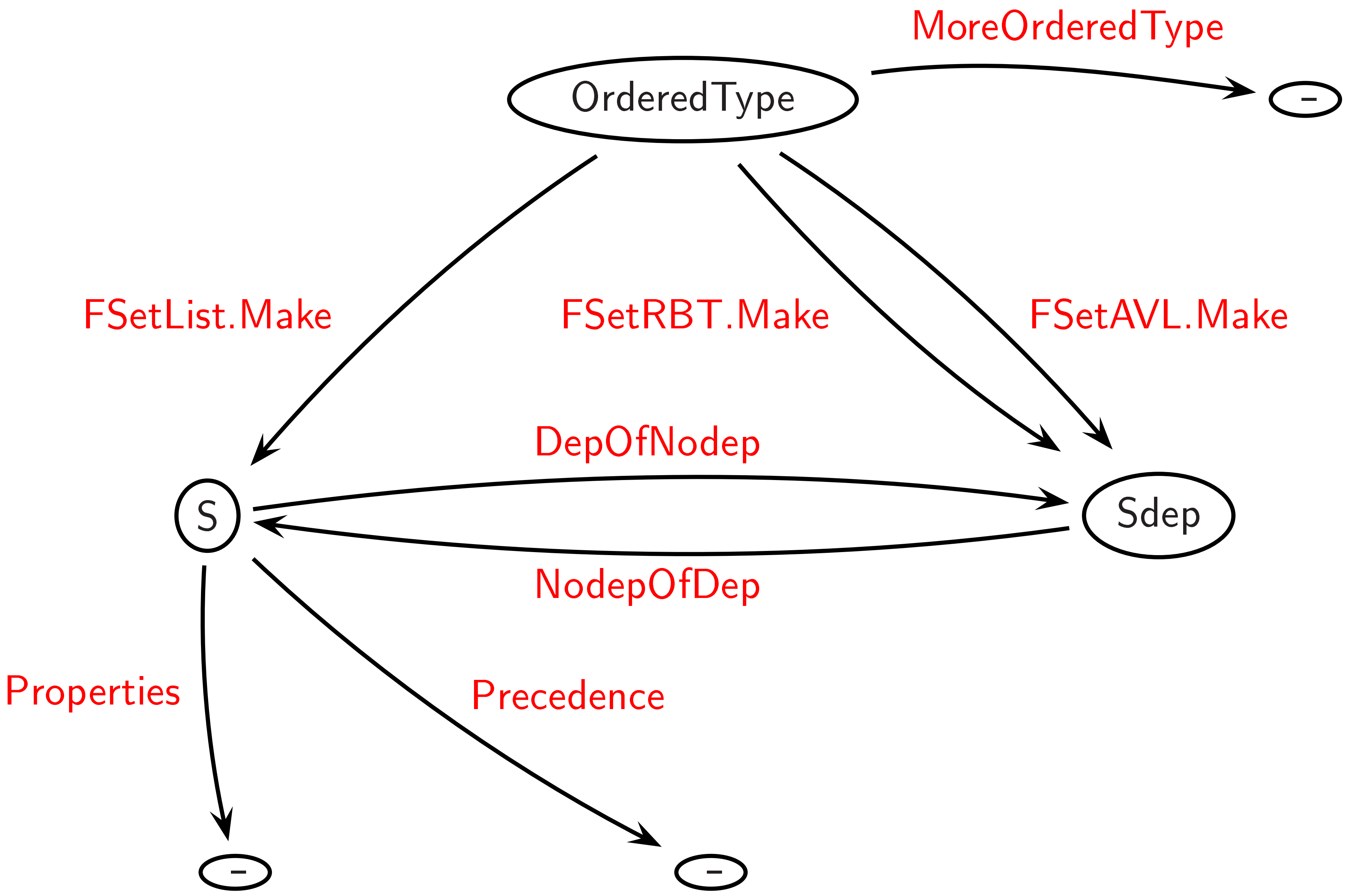
Motivations

Une bibliothèque d'ensembles finis « à la Caml »

Plan

1. Interfaces
   – Types ordonnés
   – Une signature ensembliste « fonctions + axiomes »
   – Une signature ensembliste « tout en un »
2. Listes triées
3. Arbres équilibrés
   – Arbres rouges et noirs
   – AVL

Architecture de la bibliothèque



I

Interfaces

Type ordonné en Caml

```
module type OrderedType =
  sig
    type t
    val compare : t -> t -> int
  end
```

Type ordonné en Coq

```
Inductive Compare [X:Set;lt,eq:X->X->Prop;x,y:X]:Set:=
  | Lt : (lt x y) -> (Compare lt eq x y)
  | Eq : (eq x y) -> (Compare lt eq x y)
  | Gt : (lt y x) -> (Compare lt eq x y).

Module Type OrderedType.
  Parameter t : Set.
  Parameter eq, lt : t -> t -> Prop.
  Parameter compare : (x,y:t)(Compare lt eq x y).
  ... eq_refl, eq_sym, eq_trans, lt_trans, lt_not_eq ...
End OrderedType.
```

Extraction des types ordonnés

```
type 'x compare =
  | Lt
  | Eq
  | Gt

module type OrderedType =
 sig
  type t

  val compare : t -> t -> t compare
 end
```

La signature S en Caml

```
module type S =
 sig
  type elt
  (** The type of the set elements. *)

  type t
  (** The type of sets. *)

  val empty: t
  (** The empty set. *)

  val is_empty: t -> bool
  (** Test whether a set is empty or not. *)

  [...]
```

La signature S en Caml (suite)

```
val mem: elt -> t -> bool
(** [mem x s] tests whether [x] belongs to the set [s]. *)

val add: elt -> t -> t
(** [add x s] returns a set containing all elements of [s],
   plus [x]. If [x] was already in [s], [s] is returned unchanged. *)

val singleton: elt -> t
(** [singleton x] returns the one-element set containing only [x]. *)

val remove: elt -> t -> t
(** [remove x s] returns a set containing all elements of [s],
   except [x]. If [x] was not in [s], [s] is returned unchanged. *)

[...]
```

Une signature « fonctions + axiomes » : fonctions

```
Module Type S.

  Declare Module E : OrderedType.
  Definition elt := E.t.

  Parameter t : Set.

  Parameter empty: t.

  Parameter is_empty: t -> bool.

  Parameter mem: elt -> t -> bool.

  Parameter add: elt -> t -> t.
```

Une signature « fonctions + axiomes » : cas particuliers

- `compare : t -> t -> int`
- `iter : (elt -> unit) -> t -> unit`
- `cardinal : t -> int`
- `min_elt, max_elt, choose : t -> elt`

Une signature « fonctions + axiomes » : spécifications

```
Parameter In : elt -> t -> Prop.
Parameter In_1: (E.eq x y) -> (In x s) -> (In y s).

(** Specification of [mem] *)
Parameter mem_1: (In x s) -> (mem x s)=true.
Parameter mem_2: (mem x s)=true -> (In x s).

(** Specification of [add] *)
Parameter add_1: (In x (add x s)).
Parameter add_2: (In y s) -> (In y (add x s)).
Parameter add_3:
  ~(E.eq x y) -> (In y (add x s)) -> (In y s).
```

Une signature « fonctions + axiomes » : fold

```
val fold: (elt -> 'a -> 'a) -> t -> 'a -> 'a
(** [fold f s a] computes [(f xN ... (f x2 (f x1 a))...)],
    where [x1 ... xN] are the elements of [s]. The order in
    which elements of [s] are presented to [f] is unspecified. *)
```

$$\Downarrow$$

```
Parameter fold:  (A:Set)(elt -> A -> A) -> t -> A -> A.
(** Specification of [fold] *)
Parameter fold_1: (A:Set)(i:A)(f:elt->A->A)
 (EX l:(list elt) |
     (Unique E.eq l) /\
     ((x:elt)(In x s)<->(InList E.eq x l)) /\
     (fold f s i)=(fold_right f i l)).
```

Une signature « tout en un »

```
Module Type Sdep.

  Declare Module E : OrderedType.
  Definition elt := E.t.

  Parameter t : Set.

  Parameter In:elt -> t -> Prop.
  Parameter eq_In:(s:t)(x,y:elt)(E.eq x y) -> (In x s) -> (In y s).
  Definition Empty:=[s](a:elt)~(In a s).
  Definition Add:=[x,s,s'](y:elt)(In y s')<->((E.eq y x)\/(In y s)).

  Parameter empty : { s:t | Empty s }.
  Parameter is_empty : (s:t){ Empty s }+{ ~(Empty s) }.
  Parameter mem : (x:elt) (s:t) { (In x s) } + { ~(In x s) }.
  Parameter add : (x:elt) (s:t) { s':t | (Add x s s') }.
```

Une signature « tout en un » : fold

```
Parameter fold :
 (A:Set)(f:elt->A->A)(s:t)(i:A)
 { r : A | (EX l:(list elt) |
                (Unique E.eq l) /\
                ((x:elt)(In x s)<->(InList E.eq x l)) /\
                r = (fold_right f i l)) }.
```

Extraction de la signature S

```
module type S =
 sig
  module E : OrderedType

  type elt = E.t

  type t

  val empty : t

  val is_empty : t -> bool

  val mem : elt -> t -> bool

  val add : elt -> t -> t
  [...]
```

Extraction de la signature Sdep

```
module type S =
 sig
  module E : OrderedType

  type elt = E.t

  type t

  val empty : t sig0

  val is_empty : t -> sumbool

  val mem : elt -> t -> sumbool

  val add : elt -> t -> t sig0
  [...]
```

Passerelle utilisant le module Caml

```
module Make(X:OrderedType) : S with module E = X and type elt = X.t
 =
 struct
   module E = X
   module M = Set.Make(struct
                         type t = X.t
                         let compare x y = c2i (X.compare x y)
                        end)
   include M
   let compare s s' = i2c (compare s s')
   let cardinal s = i2n (cardinal s)
   let max_elt s = try Some (max_elt s) with Not_found -> None
   let min_elt s = try Some (min_elt s) with Not_found -> None
   let choose s = try Some (choose s) with Not_found -> None
 end
```

II

Listes triées

Pourquoi une implantation naive ?

- debugger l'interface ...
- ... et le faire vite !
- utilisé a posteriori dans les arbres Rouges-Noirs

Raw : un premier foncteur sur des listes simples

Pour l'instant `t = (list elt)`

Mais on fait comme si toutes les listes étaient triées :

```
Fixpoint union [s:t] : t -> t := Cases s of
  nil => [s']s'
| (cons x l) =>
     Fix union_aux { union_aux / 1 : t -> t := [s']Cases s' of
         nil => s
       | (cons x' l') => Cases (E.compare x x') of
            (Lt _ ) => x::(union l s')
          | (Eq _ ) => x::(union l l')
          | (Gt _) => x'::(union_aux l')
          end
       end}
 end.
```

Raw : un premier foncteur sur des listes simples

On rajoute ce qu'il faut de préconditions dans les specs :

```
Lemma union_1 : (s,s':t)(Hs:(Sort s))(Hs':(Sort s'))(x:elt)
   (In x (union s s')) -> (In x s)(In x s').
```

Et on prouve que les listes restent toujours triées :

```
Lemma union_sort :
 (s,s':t)(Hs:(Sort s))(Hs':(Sort s'))(Sort (union s s')).
```

Make : un deuxième foncteur sur des listes triées

```
Record sorted_list : Set :=
 { this :> Raw.t ; sorted : (sort E.lt this) }.

Definition t := sorted_list.

[...]

Definition union := [s,s':t]
 (Build_sorted_list (Raw.union_sort (sorted s) (sorted s'))).

Definition union_1 := [s,s':t]
 (Raw.union_1 (sorted s) (sorted s')).
```

Extraction de Raw : R.A.S.

```
type elt = E.t
type t = elt list

let rec union s x =
  match s with
    | Nil -> x
    | Cons (x0, l) ->
        let rec union_aux s' = match s' with
          | Nil -> s
          | Cons (x', l') ->
              (match E.compare x0 x' with
                 | Lt -> Cons (x0, (union l s'))
                 | Eq -> Cons (x0, (union l l'))
                 | Gt -> Cons (x', (union_aux l')))
        in union_aux x
```

Extraction de Make : R.A.S.

```
type slist =
    Raw.t
    (* singleton inductive, whose constructor was Build_slist *)

type t = slist

let this s = s

let union s s' = Raw.union (this s) (this s')
```

III

Arbres équilibrés

Quels arbres

- Arbres rouges et noirs
- AVL « à la Caml »

Arbres rouges et noirs

- arbres binaires de recherche équilibrés parmi les plus simples
- L. J. Guibas & R. Sedgewick
  A dichromatic framework for balanced trees, 1978
- Un codage par des nœuds binaires des arbres 2-3-4
  (cf Algorithms chapitre 15)

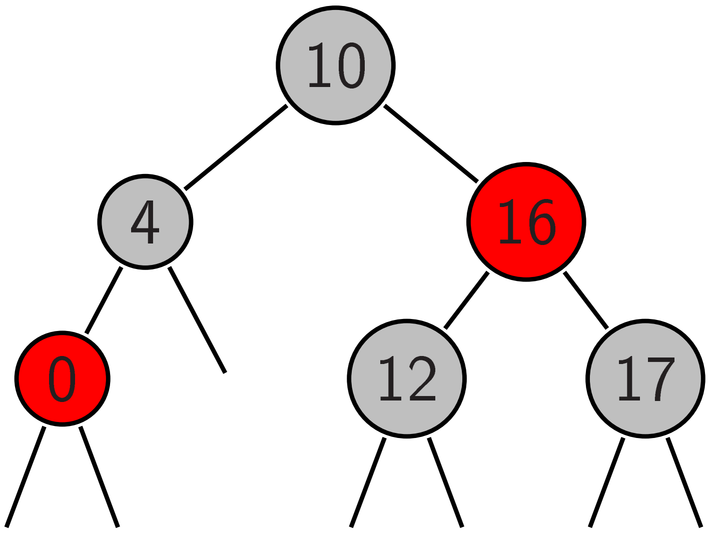
Arbres rouges et noirs : définition

Arbres binaires dont les nœuds sont rouges ou noirs

Invariants :

1. Un nœud rouge n'a pas de fils rouge
2. Tout chemin de la racine à une feuille contient le même nombre de nœuds noirs

Arbres rouges et noirs : exemple



En Coq

```
Module Make [X : OrderedType] <: Sdep with Module E := X.

  Inductive color : Set := red : color | black : color.

  Inductive tree : Set :=
  | Leaf : tree
  | Node : color -> tree -> X.t -> tree -> tree.
```

Occurrence dans un arbre, modulo X.eq

```
Inductive In_tree [x:elt] : tree -> Prop :=
| IsRoot : (l,r:tree)(c:color)(y:elt)
           (X.eq x y) -> (In_tree x (Node c l y r))
| InLeft : (l,r:tree)(c:color)(y:elt)
           (In_tree x l) -> (In_tree x (Node c l y r))
| InRight : (l,r:tree)(c:color)(y:elt)
            (In_tree x r) -> (In_tree x (Node c l y r)).
```

Arbres binaires de recherche

```
Definition lt_tree [x:elt; s:tree]
  := (y:elt)(In_tree y s) -> (X.lt y x).
Definition gt_tree [x:elt; s:tree]
  := (y:elt)(In_tree y s) -> (X.lt x y).


Inductive bst : tree -> Prop :=
| BSLeaf :
    (bst Leaf)
| BSNode : (x:elt)(l,r:tree)(c:color)
    (bst l) -> (bst r) ->
    (lt_tree x l) -> (gt_tree x r) ->
    (bst (Node c l x r)).
```

Arbres rouges et noirs

```
Inductive rbtree : nat -> tree -> Prop :=
| RBLeaf :
    (rbtree O Leaf)
| RBRed : (x:elt)(l,r:tree)(n:nat)
    (rbtree n l) -> (rbtree n r) ->
    (is_not_red l) -> (is_not_red r) ->
    (rbtree n (Node red l x r))
| RBBlack : (x:elt)(l,r:tree)(n:nat)
    (rbtree n l) -> (rbtree n r) ->
    (rbtree (S n) (Node black l x r)).
```

Type des ensembles

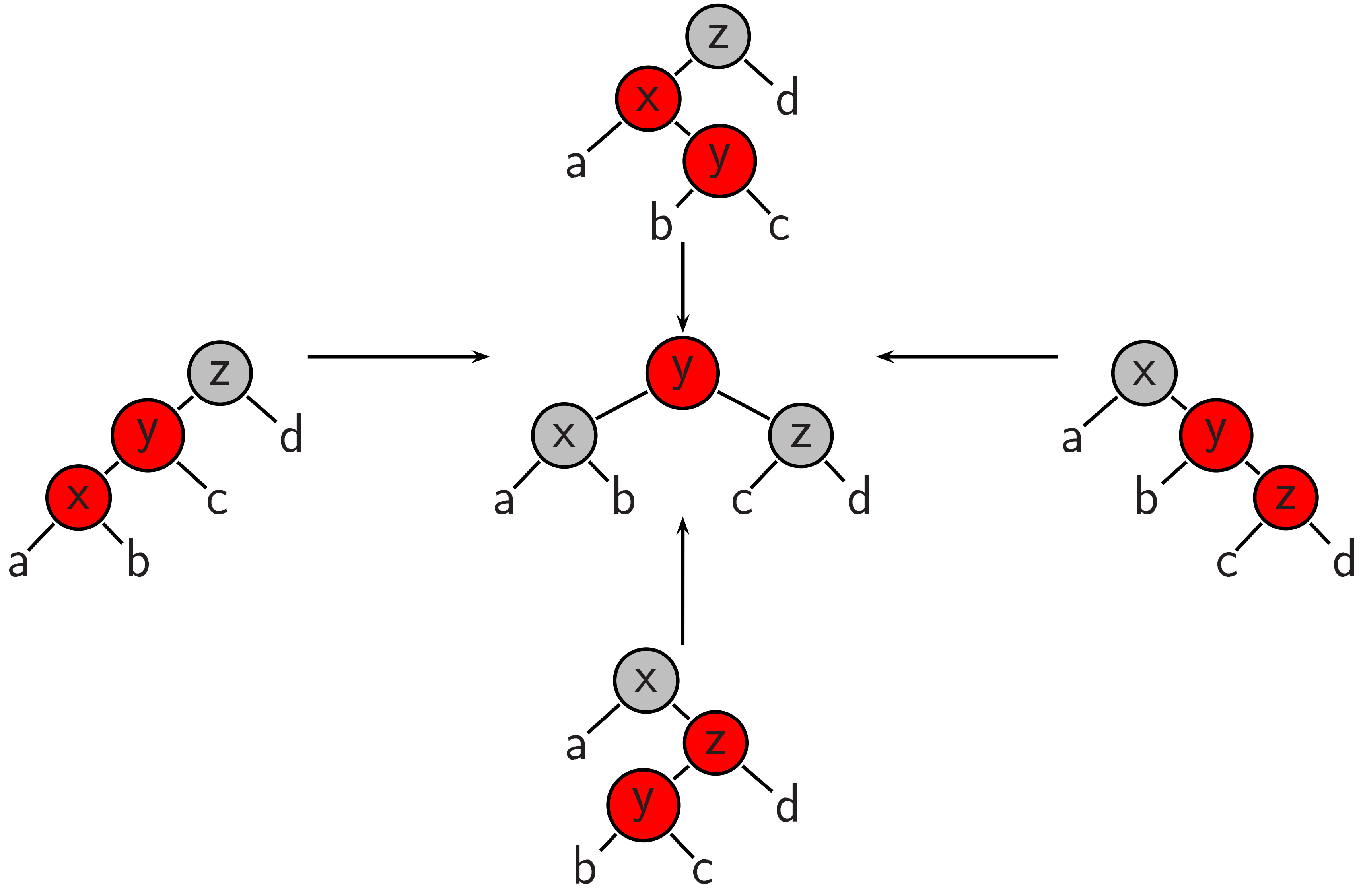
```
Record t : Set := t_intro {
  the_tree  :> tree;
  is_bst    : (bst the_tree);
  is_rbtree : (EX n:nat | (rbtree n the_tree)) }.
```

Note : l'entier n n'apparaît pas dans le code extrait

Arbres rouges et noirs : insertion

```
let add x s =
  let rec ins = function
    | Leaf ->
        Red (Leaf, x, Leaf)
    | Node (color, a, y, b) as s ->
        let c = Ord.compare x y in
        if c < 0 then balance (color, ins a, y, b)
        else if c > 0 then balance (color, a, y, ins b)
        else s
  in
  match ins s with
    | Node (_, a, y, b) -> Node (Black, a, y, b)
    | Leaf -> assert false
```

Arbres rouges et noirs : équilibrage



Fonction balance

```
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,a,x,b ->
      Node (c,a,x,b)
```

En pratique : deux fonctions lbalance et rbalance spécialisées

$\Rightarrow$ version vérifiée avec Coq

Preuve dans Coq de l'insertion

- Arbres presque rouges et noirs (éventuel conflit à la racine)

```
Inductive almost_rbtree : nat -> tree -> Prop :=
...
```

- Pattern-matching ML concentré dans une fonction

```
Inductive Conflict [s:tree] : Set :=
| NoConflict :
    ...
| RedRedConflict :
    (a,b,c:tree)(x,y:elt)
    ...
Definition conflict : (s:tree)(bst s) -> (Conflict s).
```

Spécification de {l,r}balance

```
Definition lbalance :
  (l:tree)(x:elt)(r:tree)
  (lt_tree x l) -> (gt_tree x r) ->
  (bst l) -> (bst r) ->
  { s:tree |
      (bst s) /\
      ((n:nat)((almost_rbtree n l) /\ (rbtree n r)) ->
              (rbtree (S n) s)) /\
      (y:elt)(In_tree y s) <->
             ((E.eq y x)(In_tree y l)\/(In_tree y r)) }.
Proof.
  Intros; Case (conflict l).
  ...
```

Spécification de `insert`

```
Definition insert :
  (x:elt) (s:t)
  { s':tree |
      (bst s') /\
      ((n:nat)(rbtree n s) ->
              ((almost_rbtree n s') /\
               ((is_not_red s) -> (rbtree n s')))) /\
      (y:elt)(In_tree y s') <-> ((E.eq y x)\/(In_tree y s)) }.
Proof.
  ...
```

Autres fonctions

Aucune autre fonction trouvée dans la littérature

- remove

  Solution classique : on se ramène à l'extraction du plus petit élément du sous-arbre droit

  Problème : la hauteur noire peut changer $\Rightarrow$ rééquilibrage

  Au final un code bien plus long que add (88 / 33)

- union, inter, diff, partition, filter, subset, equal, compare, for_all, exists

  passage de/vers des listes $\Rightarrow$ `FSetList.Make`

- empty, singleton, mem, elements, cardinal, fold, choose, min_elt, max_elt pas de problème

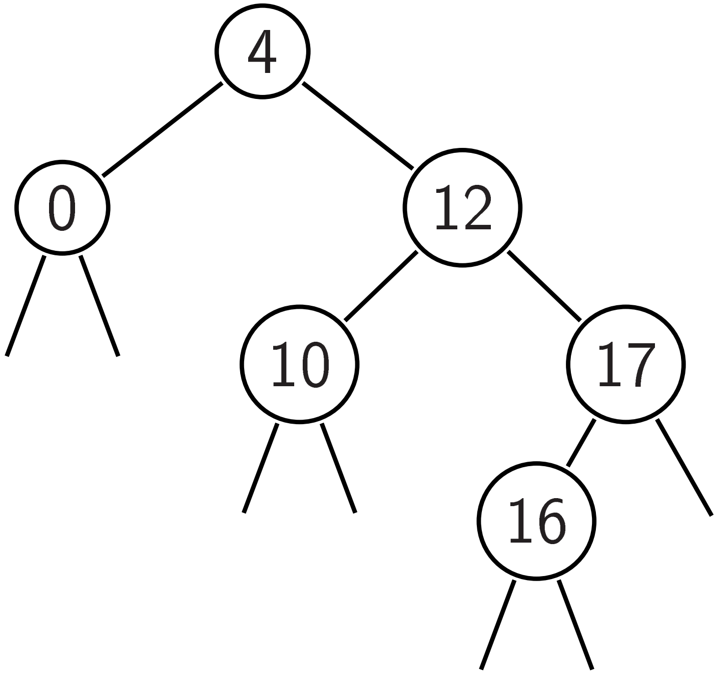
Arbres AVL

- G.M. Adel'son-Vel'skiĭ and E.M. Landis
  An algorithm for the organization of information, 1962

- Arbres binaires de recherche dont les hauteurs de sous-arbres frères diffèrent d'au plus 1

- modules `Set` et `Map` d'Ocaml : au plus 2
  $\Rightarrow$ le code vérifié est celui d'Ocaml

Exemple



Représentation

À la différence des arbres rouges et noirs, la hauteur est conservée dans la représentation

```
type t = Empty | Node of t * elt * t * int
```

```
Inductive tree : Set :=
| Leaf : tree
| Node : tree -> X.t -> tree -> Z -> tree.
```

Définition de bst comme pour les arbres rouges et noirs

Le prédicat avl

Difficulté propre à Coq : éviter l'utilisation de max et de Zabs pour favoriser Omega

```
Definition height_of_node [l,r:tree; h:Z] :=
  ((height l) >= (height r) /\ h = (height l) + 1) \/
  ((height r) >= (height l) /\ h = (height r) + 1).

Inductive avl : tree -> Prop :=
| RBLeaf :
    (avl Leaf)
| RBNode : (x:elt)(l,r:tree)(h:Z)
    (avl l) -> (avl r) ->
    `-2 <= (height l) - (height r) <= 2` ->
    (height_of_node l r h) ->
    (avl (Node l x r h)).
```

Type des ensembles

```
Record t : Set := t_intro {
  the_tree :> tree;
  is_bst : (bst the_tree);
  is_avl : (avl the_tree)
}.
```

La tactique AVL

```
(* [h] is a proof of [avl (Node l x r h)] *)
Tactic Definition AVL h :=
  (Generalize (height_non_negative h); Try Simpl);
  (Try Generalize (height_equation h)); Intros.



...
AVL avl_l; AVL H3; AVL s_avl; Omega.
...
```

Équilibrage

Repose sur une unique fonction bal qui accepte une différence d'au plus 3 à la racine et rétablit une différence d'au plus 2

```
Definition bal :
  (l:tree)(x:elt)(r:tree)
  (bst l) -> (avl l) -> (bst r) -> (avl r) ->
  (lt_tree x l) -> (gt_tree x r) ->
  `-3 <= (height l) - (height r) <= 3` ->
  { s:tree |
     (bst s) /\ (avl s) /\
     (* height may be decreased by 1 *)
     (((height_of_node l r (height s)) \/
       (height_of_node l r ((height s) + 1))) /\
     (* ...but is unchanged when no rebalancing *)
      (`-2 <= (height l) - (height r) <= 2` ->
       (height_of_node l r (height s)))) /\
     (* elements are those of (l,x,r) *)
     (y:elt)(In_tree y s) <->
            ((X.eq y x) \/ (In_tree y l) \/ (In_tree y r)) }.
```

La fonction add

Ensuite l'insertion est classique :

```
let rec add x = function
  | Empty -> Node(Empty, x, Empty, 1)
  | Node(l, v, r, _) as t ->
      let c = Ord.compare x v in
      if c = 0 then t else
      if c < 0 then bal (add x l) v r else bal l v (add x r)
```

Et là... surprise !

Les fonctions join et merge ne sont pas correctes

```
(* Same as bal, but repeat rebalancing until the final result
   is balanced. *)
let rec join l x r = match bal l x r with
  | Empty -> invalid_arg "Set.join"
  | Node(l', x', r', _) as t' ->
      let d = height l' - height r' in
      if d < -2 || d > 2 then join l' x' r' else t'
```

les arbres construits peuvent être mal équilibrés

$\Rightarrow$ il en va de même des fonctions remove, union, diff, inter

Réparation

X. Leroy a corrigé le code

⇒ fonctions join et merge un peu plus complexes, maintenant certifiées, ainsi que remove, diff, inter

```
let rec join l v r =
  match (l, r) with
    (Empty, _) -> add v r
  | (_, Empty) -> add v l
  | (Node(ll, lv, lr, lh), Node(rl, rv, rr, rh)) ->
      if lh > rh + 2 then bal ll lv (join lr v r) else
      if rh > lh + 2 then bal (join l v rl) rv rr else
      create l v r
```

Autres fonctions

Prouvées : union, cardinal, elements, filter, partition, subset

Restent à prouver : equal, compare, fold, exists, for_all

Une difficulté : la terminaison de `union` et de `subset`

```
let rec union s1 s2 = match (s1, s2) with
  | (Empty, t2) -> t2
  | (t1, Empty) -> t1
  | (Node(l1, v1, r1, h1), Node(l2, v2, r2, h2)) ->
        if h1 >= h2 then
          let (l2, _, r2) = split v1 s2 in
          join (union l1 l2) v1 (union r1 r2)
        else
          let (l1, _, r1) = split v2 s1 in
          join (union l1 l2) v2 (union r1 r2)
```

ce qui diminue : $\text{card}(s_1) + \text{card}(s_2)$

Statistiques

	RBT	AVL
Lignes Caml	314	235
Lignes Coq	1683	1840
Compil Coq	1 :19	9 :30

Un total de 7100 lignes de Coq

Conclusions personnelles

- Merci à Jacek pour les modules et surtout les foncteurs
- Merci à Benjamin pour CoqIDE
- Merci à Judicaël et Pierre pour Intuition et Ground