Verifying programs and proofs
part V. More data-structures

Yves Bertot

September 2012

1 Introduction

Jusqu'à maintenant nous avons seulement étudié des preuves sur les nombres et les listes. Il est possible de définir des structures de données plus complexes, qui permettent de fabriquer des programmes plus efficaces. Nous pouvons illustrer cette progression en observant deux nouvelles structures: les nombres positifs (avec un algorithme d'addition) et les arbres binaires (avec un algorithme pour le tri des listes).

2 Recursion sur les nombres positifs

2.1 Les contraintes de programmation

Les nombres positifs sont représentés par un type de données avec trois constructeurs. Le constructeur nommé `xH` est utilisé pour représenter 1, le constructeur nommé `xO` est utilisé pour reprérsenter la fonction qui envoie x vers $2x$ et le constructeur nommé `xI` est utilisé pour représenter la fonction qui envoie x ver $2x + 1$.

```
Require Import ZArith Arith.

Print positive.
Inductive positive : Set :=
    xI : positive -> positive
  | xO : positive -> positive
  | xH : positive
```

Quand on écrit des algorithmes récursifs sur cette structure de données, on peut seulement avoir des appels récursifs sur la sous-composante des constructeurs `xO` et `xI`. En observant les nombres représentés, ceci correspond à autoriser seulement les appels récursifs sur la moitié du nombre donné en argument (arrondie vers le bas).

Si nous concevons des algorithmes, nous avons besoin de prendre cette contrainte en compte. Nous pouvons l'illustrer par l'opération qui additionne deux nombres.

2.2 Convertir un nombre positif en nombre naturel

Pour calculer le nombre naturel qui correspond à un nombre positif, nous pouvons nous contenter d'utiliser les interprétations associées a chaque constructeur. Chaque repose naturellement sur un appel récursif sur l'argument prévu par la structure de donnée.

```
Fixpoint pos_to_nat (x : positive) : nat :=
  match x with
    xH => 1
  | xO p => 2 * pos_to_nat p
  | xI p => S (2 * pos_to_nat p)
  end.
```

2.3 La fonction qui incrémente un nombre positif

Pour une fonction d'incrémentation, nous devons décomposer la description de la fonction en trois cas.

1. Pour le cas de base, l'entrée est `xH` (qui représente 1) et la sortie est `xO xH` (qui représente 2).

2. Pour le cas où l'entrée est `xO p` (cette entrée représente un nombre $2x_p$ où x_p est le nombre représenté par p), il suffit de retourner `xI p` (qui représente $2x_p + 1$).

3. Pour le cas où l'entrée est `xI p`, si x_p est le nombre représenté par `p`, il faut retourner `xO p'` où `p` est la représentation de $x_p + 1$. En fait `p'` peut simplement être obtenu par un appel récursif de la même fonction d'incrémentation sur `p`.

Tout ce raisonnement se condense dans la définition suivante :

```
Fixpoint add1 (x : positive) : positive :=
  match x with
    xH => xO xH
  | xO p => xI p
  | xI p => xO (add1 p)
  end.
```

2.4 Ajouter deux nombres positifs

Quand on ajoute deux nombres, nous pouvons utiliser le même algorithme que celui que nous avons appris à l'école primaire pour ajouter des nombres en base

10, sauf qu'ici la base est 2 et nous n'avons pas besoin de connaitre des tables d'addition.

D'un autre coté, l'addition telle que nous l'avons apprise à l'école peut faire intervenir une retenue. Donc, nous allons utiliser une fonction à trois arguments, le troisième argument étant une valeur booléenne indiquant la présence d'une retenue ou non. L'algorithm inclus une analyse de cas pour les deux arguments et la retenue, donc à la fin nous avons 18 cas ($18 = 3 \times 3 \times 2$).

Observons d'abord les cas où il n'y a pas de retenue.

1. Si l'un des arguments est 1, alors nous utilisons la fonction d'incrémentation `add1`.

2. Si les deux arguments numériques sont impairs, il y a une retenue pour l'appel récursif. Le calcul suit le schéma suivant:

$$(2x + 1) + (2y + 1) = 2(x + y + 1).$$

   Un appel récursif est calculé pour l'expression $(x + y + 1)$.

3. Si les deux arguments numériques sont pairs, il n'y a pas de retenue dans l'appel récursif, suivant le schéma suivant:

$$2x + 2y = 2(x + y).$$

   Un appel récursif est calculé pour $(x + y)$.

4. Si l'un seulement des arguments numériques est impair, alors le résultat est impair et il n'y a pas de retenue dans l'appel récursif.

$$(2x + 1) + 2y = 2(x + y) + 1$$

   L'appel récursif sert à calculer $(x + y)$

Observons maintenant le cas où il y a une retenue en entrée.

1. Si les deux arguments numériques en entrée sont 1, alors le résultat est 3.

2. Sinon, si l'un des arguments numériques est 1, nous pouvons utiliser `add1` sur la moitié de l'autre argument, puis réutiliser le même constructeur :

$$(2x) + 1 + 1 = 2(x + 1),$$

$$(2x + 1) + 1 + 1 = 2(x + 1) + 1.$$

3. Si les deux arguments numériques sont impairs et supérieurs à 1, alors il y a une retenue dans le résultat, le calcul suit le schéma suivant :

$$(2x + 1) + (2y + 1) + 1 = 2(x + y + 1) + 1$$

4. Si les deux arguments numériques sont pairs, alors il n'y a pas de retenue. Le schéma de calcul est le suivant:

$$2x + 2y + 1 = 2(x + y) + 1$$

L'appel récursif est utilisé pour calculer $(x + y)$

5. Si l'un des arguments numérique est pair et pas l'autre, alors le résultat est pair et il y a une retenue dans l'appel récursif.

$$(2x + 1) + 2y + 1 = 2(x + y + 1)$$

Nous pouvons condenser toutes ses réflexions dans la description suivante:

```
Fixpoint pos_add (x y : positive) (c : bool) : positive :=
 match x, y, c with
   xI x', xI y', false => xO (pos_add x' y' true)
 | xO x', xI y', false => xI (pos_add x' y' false)
 | xI x', xO y', false => xI (pos_add x' y' false)
 | xO x', xO y', false => xO (pos_add x' y' false)
 | xH, y, false => add1 y
 | x, xH, false => add1 x
 | xI x', xI y', true => xI (pos_add x' y' true)
 | xO x', xI y', true => xO (pos_add x' y' true)
 | xI x', xO y', true => xO (pos_add x' y' true)
 | xO x', xO y', true => xI (pos_add x' y' false)
 | xH, xH, true => xI xH
 | xH, xI y, true => xI (add1 y)
| xH, xO y, true => xO (add1 y)
 | xI x, xH, true => xI (add1 x)
| xO x, xH, true => xO (add1 x)
end.
```

2.5 Prouver que les fonctions sont correctes

Nous allons d'abord prouver que la fonction `add1` est correcte. Nous utilisons la conversation des nombres positifs vers les nombres naturels pour exprimer la correction. Comme d'habitude, nous montrons que la fonction est correcte en montrant qu'elle est cohérente avec d'autres fonctions.

```
Lemma add1_correct :
  forall x, pos_to_nat (add1 x) = S (pos_to_nat x).
Proof.
```

Cette preuve est faite par récurrence sur le nombre positif. Parce que le type inductif a trois cas, cette preuve a aussi trois cas.

```
induction x.
3 subgoals

  x : positive
  IHx : pos_to_nat (add1 x) = S (pos_to_nat x)
  ============================
   pos_to_nat (add1 x~1) = S (pos_to_nat x~1)

subgoal 2 is:
 pos_to_nat (add1 x~0) = S (pos_to_nat x~0)
subgoal 3 is:
 pos_to_nat (add1 1) = S (pos_to_nat 1)
```

Les notations `x~1`, `x~0` et 1 représentent respectivement `xI x`, `xO x` et `xH`.

Pour le premier cas, nous pouvons forcer le calcul de la fonction récursive de la façon suivante, ce fait apparaître le terme de l'hypothèse de récurrence.

```
simpl.
  ...
  ============================
   pos_to_nat (add1 x) + (pos_to_nat (add1 x) + 0) =
   S (S (pos_to_nat x + (pos_to_nat x + 0)))
rewrite IHx; ring.
```

Après réécriture avec l'hypothèse de récurrence, nous obtenons une égalité qui est résolue facilement par la tactique `ring`.

Ensuite le deuxième cas apparait. Dans ce cas, la tactique `simpl` retourne une égalité simple.

```
  ...
 ============================
   pos_to_nat (add1 x~0) = S (pos_to_nat x~0)
simpl.
  ...
 ============================
   S (pos_to_nat x + (pos_to_nat x + 0)) =
   S (pos_to_nat x + (pos_to_nat x + 0))
reflexivity.
```

Le troisième cas est aussi résolu facilement par réflexivité.

```
reflexivity.
Qed.
```

Lorsque que l'addition est correcte, il est plus malin d'utiliser la régularité de la fonction pour couvrir plusieurs cas à la fois. Nous effectuons une preuve par récurrence sur le premier argument et par cas sur le deuxième et le trois arguments. Ceci produit 18 sous-buts d'un coup. Ensuite une partie de ces cas est résolu directement par simplification et en utilisant la tactique `ring` sur les égalités produites.

```
Lemma pos_add_correct :
  forall x y c, pos_to_nat (pos_add x y c) =
    pos_to_nat x + pos_to_nat y + if c then 1 else 0.
Proof.
induction x as [x' | x' | ]; intros [y' | y' | ] [ | ];
  simpl; try ring.
```

Dans cette tactique composée, la notation `intros [ y' | y' | ]` est une abbréviation pour `intros y; destruct y as [ y' | y' | ]`.

Ceci ne laisse que 14 buts, donc 4 buts ont déjà été résolus par `ring`. Nous voyons que certains de ces buts mentionnent `add1`, donc il devrait être utile de réécrire avec le théorème de correction pour cette fonction. Reprenons la preuve avec la tactique composée suivante.

```
induction x as [x' | x' | ]; intros [y' | y' | ] [ | ];
  simpl; try rewrite add1_correct; try ring.
```

Ceci ne laisse que 8 buts, donc nous avons réussi à résoudre systématiquement 6 buts supplémentaires. L'idée suivante est d'utiliser l'hypothèse de récurrence, quand elle existe. Reprenons avec la tactique combinée suivante.

```
induction x as [x' | x' | ]; intros [y' | y' | ] [ | ];
  simpl; try rewrite add1_correct; try rewrite IHx'; try ring.
```

Ceci résout le but.

```
Qed.
```

Notez bien que la preuve n'aurait progressé aussi bien si nous avions executé les étapes suivantes `intros x y c; induction x` comme étapes initiales. Ceci produit des hypothèses de récurrences plus faibles, trop faibles pour le problème que nous voulons résoudre.

3 Structures arborescentes

Les listes sont des structures sympatiques pour notre travail de programmation et de preuve. Elles sont faciles à comprendre et leur forme est quand même assez proche de la forme des tableaux, qui sont utilisés partout dans les langages de programmations conventionnels. Donc, si notre tâche était seulement de raisonner sur des programmes conventionnels, ces listes seraient tout ce dont nous avons besoin. Néanmoins, nous avons besoin parfois de construire des algorithmes plus efficaces dans le cadre de la programmation fonctionnelle. Pour cette tâche, les listes ne sont pas satisfaisante, parce que le coût moyen pour récupérer une donnée dans une liste est proportionnel à la longueur de cette liste.

Quand on utilise un arbre binaire, ou une structure de donnée avec plus d'une branche, nous pouvons avoir plus d'efficacité, parce que accéder à une donnée (si nous savons où chercher), peut avoir un coût logarithmique dans le nombre d'éléments stocké dans l'arbre. Nous allons illustrer cela avec deux implémentations d'algorithmes de tri.

3.1 tri par insertion

Les deux fonctions qui suivent fournissent un tri pour des listes contenant des entiers.

```
Fixpoint insert (a : Z) (l : list Z) :=
  match l with
    nil => a::nil
  | b::tl => if Zle_bool a b then a::b::tl else b::insert a tl
  end.

Fixpoint sort (l : list Z) :=
  match l with nil => nil | a::tl => insert a (sort tl) end.
```

Pour prouver que ces fonctions sont correctes, nou pouvons prouver deux faits: le tri ne perd pas de données et la liste résultat est réellement triée. Voici comment nous pouvons exprimer ces deux faits.

```
Fixpoint count (x : Z) (l : list Z) :=
  match l with
    nil => 0
  | y::tl =>
    if Zeq_bool x y then 1 + count x tl else count x tl
  end.

Fixpoint sorted (l : list Z) :=
  match l with
    a :: (b :: tl as l') =>
    if Zle_bool a b then sorted l' else false
  | _ => true
  end.

Lemma sort_perm : forall x l, count x l = count x (sort l).
Admitted.

Lemma sort_sorted : forall l, sorted (sort l) = true.
Admitted.
```

Vous êtes invités à effectuer ces deux preuves en exercice.

Cet algorithme de tri n'est pas trop mauvais sur les listes qui sont presque triées, mais il a une complexité quadratique en moyenne. Observons le cas le pir, qui apparait lorsque la liste en entrée est triée dans l'ordre inverse.

Par exemple, observons les calculs effectués lorsque la liste a trier est `4::3::2::1::nil`. L'algorithme trie d'abord `3::2::1::nil`, et pour cela il trie `2::1::nil`, ce qui requiert une comparaison et produit `1::2::nil`, puis il insère `3` dans cette liste, donc `3` est comparé avec `1` puis `2`. Cela fait deux comparaisons et produit `1::2::3::nil`. Puis il insère 4 dans cette liste et cela requiert 3 comparaisons

pour placer 4 à la fin du résultat. En extrapolant pour une liste de longueur `n`, nous voyons que le nombre de comparaisons requises serait le nombre

$$\sum_{i=1}^{n}(i-1) = \frac{n(n-1)}{2}.$$

En pratique, sur mon ordinateur il faut 0,6 secondes pour trier une liste de mille éléments et 3 secondes pour trier une liste de deux mille éléments.

3.2 Tri par fusion

Pour trier une grande liste de nombres, il est plus efficace de décomposer cette liste en deux listes de longueur approximativement égale, de trier les sous-listes et de fusionner ces deux listes triées. Pendant la phase de fusion, nous pouvons nous assurer que le nombre de comparaisons nécessaires reste petit. Un tel algorithme peut être programmé en Coq de la façon suivante.

```
Fixpoint merge_aux (a : Z) (l' : list Z)
  (f : list Z -> list Z) (l : list Z) :=
  match l with
  | b :: l' =>
    if Zle_bool a b then f l else b :: merge_aux a f l'
  | nil => l'
  end.

Fixpoint merge (l1 : list Z) : list Z -> list Z:=
  match l1 with
  | a :: l1' =>
    fix m2 l2 : list Z :=
    match l2 with
    | b :: l2' =>
      if Zle_bool a b then a::merge l1' l2 else b::m2 l2'
    | nil => l1
    end
  | nil => fun l2 => l2
  end.
```

Ce bout de code utilise une technique avancée puisqu'il repose sur l'utilisation d'une fonction d'ordre supérieur (une fonction qui prend une autre fonction en argument). La fonction `merge_aux` insère dans le résultat tous les éléments de la deuxième liste qui sont plus petits que `a`. Lorsque tous ces éléments sont épuisés, elle appelle l'autre fonction qui est chargée de fusionner le reste de la première liste avec n'importe quelle liste.

Nous pouvons ensuite construire un algorithme qui prend en entrée un arbre binaire et produit une liste qui contient toutes les valeurs incluses dans cet arbre. Pour chaque nœud binaire de l'arbre, nous groupons ensemble les listes obtenues pour les sous arbres en faisant une fusion à l'aide de la fonction merge. Ceci garantit que le résultat final est trié.

```
Fixpoint bintolist (t : bin) : list Z :=
  match t with
    L x => x::nil
  | N t1 t2 => merge (bintolist t1) (bintolist t2)
  end.
```

Pour trier une liste il suffit donc de fabriquer un arbre binaire qui contient toutes les valeurs dans cette liste, puis de re-transformer cet arbre binaire en liste. Pour bénéficier de l'efficacité de l'algorithme, il faut être assez malin pour fabriquer un arbre dont toutes les branches ont approximativement la même longueur. Une technique maline est fournie par les deux fonctions suivantes:

```
Fixpoint inst (x : Z) (t : bin) :=
  match t with
    L y => N (L x) (L y)
  | N t1 t2 => N (inst x t2)  t1
  end.

Fixpoint insl l t :=
  match l with
    nil => t
  | a::tl => inst a (insl tl t)
  end.

Definition msort l :=
  match l with
    nil => nil
  | a::tl => bintolist (insl tl (L a))
  end.
```

En pratique, il faut 0,02 secondes pour trier une liste de mille éléments avec cette fonction et 0.04 pour trier une liste de deux mille éléments.

Prouver que ce code est correct est plus complexe que pour le tri par insertion. Par exemple, je fournis ici une preuve complete que la fonction `merge` produit une liste triée. Cette preuve contient deux preuves par récurrences imbriquées, ce qui n'est pas surprenant, puis que la fonction `merge` fait appel à une deuxième fonction récursive `merge_aux`. Ici encore, nous devons prouver un énoncé plus fort. Nous prouvons non seulement que la fonction `merge` produit une liste triée, mais aussi que le premier élément du résultat est l'un des premiers éléments des deux listes en entrée.

Je fournis la preuve ici pour montrer qu'elle est faisable, mais vous pouvez vous contenter d'admettre ce résultat pour la suite.

```
Definition head_constraint (l1 l2 l : list Z) :=
  (exists l1', exists l2', exists l', exists a, exists b, exists c,
    l1 = a::l1' /\ l2 = b::l2' /\ l = c::l' /\
    (c = a \/ c = b)) \/
  (l1 = nil /\ exists l2', exists l', exists b, l2 = b::l2' /\ l = b::l') \/
```

```
  (l2 = nil /\ exists l1', exists l', exists a, l1 = a::l1' /\ l = a::l') \/
  (l1 = nil /\ l2 = nil /\ l = nil).

Lemma merge_sorted :
  forall l1 l2, sorted l1 = true -> sorted l2 = true ->
    sorted (merge l1 l2) = true /\
    head_constraint l1 l2 (merge l1 l2).
induction l1; intros l2.
 intros sn sl2;split;[exact sl2 | ].
 destruct l2 as [ | b l2'].
  right; right; right; repeat split; reflexivity.
 right; left;split;[reflexivity | exists l2'; exists l2'; exists b].
 split; reflexivity.
induction l2 as [ | b l2 IHl2].
 intros sal1 _; split.
  assumption.
 right; right; left; split;[reflexivity | ].
 exists l1; exists l1; exists a; split; reflexivity.
intros sal1 sbl2.
change (merge (a::l1) (b::l2)) with
  (if Zle_bool a b then a::merge l1 (b::l2) else b::merge (a::l1) l2).
case_eq (a <=? b)%Z.
 intros cab; split.
  destruct l1 as [ | a' l1].
   simpl merge.
   change ((if (a <=? b)%Z then sorted (b::l2) else false) = true).
   rewrite cab; assumption.
  assert (int: sorted (a'::l1) = true).
   simpl in sal1; destruct (a <=? a')%Z;[exact sal1 | discriminate].
  apply (IHl1 (b::l2)) in int.
   destruct int as [int1 int2].
   case_eq (merge (a'::l1) (b::l2)).
    intros q; rewrite q in int2.
    destruct int2 as [t2 | [t2 | [t2 | t2]]].
       destruct t2 as [abs1 [abs2 [abs3 [abs4 [abs5 [abs6 [_ [_ [abs9 _]]]]]]]]].
       discriminate.
      destruct t2 as [abs _]; discriminate.
     destruct t2 as [abs _]; discriminate.
    destruct t2 as [abs _]; discriminate.
   intros r l' qm.
   change ((if (a <=? r)%Z then sorted (r :: l') else false) = true).
   destruct int2 as [t2 | [t2 | [t2 | t2]]];
      try (destruct t2 as [abs _]; discriminate).
   destruct t2 as [l1' [l2' [l'' [u [v [w [q1 [q2 [qm' [H | H]]]]]]]]]].
    rewrite <- qm, int1.
    rewrite qm in qm'; injection qm';
     injection q1; injection q2; intros; subst.
    simpl in sal1; destruct (a <=? u)%Z; reflexivity || discriminate.
   rewrite qm in qm'; injection qm'; injection q1; injection q2.
   intros; subst; rewrite <- qm, int1, cab; reflexivity.
```

```
  assumption.
 left.
 exists l1; exists l2; exists (merge l1 (b :: l2)); exists a; exists b.
 exists a.
 repeat split; left; reflexivity.
intros cab.
assert (cab' : (b <=? a = true)%Z).
 apply Zle_imp_le_bool; rewrite Z.leb_nle in cab; omega.
assert (sl2 : sorted l2 = true).
 destruct l2 as [ | b' l2].
  reflexivity.
 simpl in sbl2; destruct (b <=? b')%Z; assumption || discriminate.
destruct (IHl2 sal1 sl2) as [int1 int2].
destruct l2 as [ | b' l2].
 split;[ | left; exists l1; exists nil; exists (merge (a::l1) nil); exists a;
          exists b; exists b; repeat split; right; reflexivity].
 destruct int2 as [t2 | [t2 | [ t2 | t2]]];
  try (destruct t2 as [abs _]; discriminate).
  destruct t2 as [ab1 [ab2 [ab3 [ab4 [ab5 [ab6 [_ [ab _]]]]]]]]; discriminate.
 destruct t2 as [_ [l1' [l'' [u [q qm]]]]].
 rewrite qm in int1 |- *.
 change ((if (b <=? u)%Z then sorted (u::l'') else false) = true).
 injection q; intros; subst.
 rewrite cab', int1; reflexivity.
destruct int2 as [t2 | [t2 | [t2 | t2]]];
  try (destruct t2 as [abs _]; discriminate).
destruct t2 as [l1' [l2' [l' [u [v [w [q [q' [qm [H | H]]]]]]]]]].
 injection q; injection q'; intros; subst.
 rewrite qm in *.
 split;[ | left; exists l1'; exists (v::l2'); exists (u::l');
             exists u; exists b; exists b; repeat split; right; reflexivity].
 change ((if (b <=? u)%Z then sorted (u::l') else false) = true).
 rewrite cab'; assumption.
injection q; injection q'; intros; subst.
rewrite qm in *.
split;[ | left; exists l1'; exists (v::l2'); exists (v::l');
             exists u; exists b; exists b; repeat split; right; reflexivity].
change ((if (b <=?v)%Z then sorted (v :: l') else false) = true).
simpl in sbl2; destruct (b <=? v)%Z; assumption || discriminate.
Qed.
```

En utilisant le résultat sur `merge`, nous pouvons ensuite prouver que la fonction `bintolist` est correcte. C'est une preuve par récurrence sur les arbres binaires, donc il y a deux cas et le cas récursif a deux hypothèses de récurrence.

```
Lemma bintolist_sorted : forall t, sorted (bintolist t) = true.
induction t.
2 subgoals, subgoal 1 (ID 1478)

  x : Z
```

```
  ============================
   sorted (bintolist (L x)) = true

subgoal 2 (ID 1483) is:
 sorted (bintolist (N t1 t2)) = true
```

Le cas de base est plutôt simple. Quand l'arbre en entrée est une feuille, le résultat de `bintolist` est une liste à élément and la fonction `sorted` retourne toujours `true` pour les listes de de cette forme. La tactique `reflexivity` resout ce cas directement.

```
 reflexivity.
simpl.
  t1 : bin
  t2 : bin
  IHt1 : sorted (bintolist t1) = true
  IHt2 : sorted (bintolist t2) = true
  ============================
   sorted (merge (bintolist t1) (bintolist t2)) = true
```

Ici l'énoncé que nous devons prouver est la première partie de la conjonction fournie par `merge_sorted`. Nous utilisons la tactique `assert` pour appliquer ce théorème aux bons arguments. Notez que nous utilisons les deux hypothèses de récurrence pour satisfaire les conditions requises par `merge_sorted`.

```
assert (tmp := merge_sorted (bintolist t1) (bintolist t2) IHt1 IHt2).
destruct tmp as [tmp' _]; assumption.
Qed.
```

Nous pouvons maintenant conclure que la fonction produit bien une liste triée. Ici, le fait que `insl` construit un arbre équilibré ne joue aucun rôle dans la preuve. Même si la fonction `insl` produisait un arbre déséquilibré, le résultat serait correct, mais dans ce cas le calcul serait plus facile. C'est un exemple d'erreur qui n'est pas capturé par notre approche.

```
Lemma msort_sorted : forall l, sorted (msort l) = true.
intros [ | a l].
 reflexivity.
unfold msort; apply bintolist_sorted.
Qed.
```

Pour une preuve de correction plus complète, il faudrait aussi montrer que tous les éléments de la liste en entrée apparaissent en sortie. Nous laissons ce travail à faire en exercise.