Preuves & Programmes
Un cas d'école : preuve de correction de programmes fonctionnels de tris dans le système Coq

Pascal Manoury
IBP-LITP
45-55 203
Université Pierre et Marie Curie
4 Place Jussieu
75252 PARIS Cedex
FRANCE
manoury@litp.ibp.fr

Introduction

Nous nous intéresserons ici à l'implémentation fonctionnelle de quelques algorithmes de tris pour les listes d'entiers naturels. Notre but est de donner des implémentations certifiées correctes de ces algorithmes. Ce travail a été réalisé en utilisant le système d'aide à la démonstration Coq [5]. Ce système, développé à l'INRIA, est basé sur le formalisme du Calcul des Constructions Inductives [10].

Notre méthode de travail est la suivante :

(a) écrire un programme de tri implémentant chacun des algorithmes choisis,

(b) prouver que chacun des programmes satisfait la spécification attendue d'un programme de tri.

Formulée ainsi, notre méthode n'a rien d'original. Par exemple, les méthodes développées par C.A.R. Hoare [6] ou par E.W. Dijkstra [4] procèdent de la même idée. Cependant, les formalismes mis en œuvre par ces derniers et celui que nous avons utilisé se distinguent de la façon suivante. La logique de Hoare (pour ne citer qu'elle) vient s'ajouter comme un commentaire sur le texte d'un programme et les deux niveaux de langage, celui de la logique et le langage

de programmation, sont hétérogènes. Dans notre formalisme, le langage dans lequel sont exprimées les propriétés attendues et celui dans lequel sont écrits les programmes sont identiques.
Cette homogénéïté entre langage logique et langage de programmation se retrouve dans le sytème `nqthm` développé initialement par R.S. Boyer et J S. Moore [1]. Cependant, elle est obtenue au prix de *"l'embrigadement"* de la logique dans un langage de programmation (LISP). En contraste, le formalisme que nous utilisons est *primitivement* un langage logique qui permet de surcroît d'exprimer des programmes.

Cette possiblité nous est offerte par la correspondance que l'on peut établir entre les preuves formalisées par le système de la *déduction naturelle* [11] et le formalisme d'expression des fonctions récursives qu'est le λ-*calcul* [7].
Le formalisme du λ-calcul contient essentiellement deux constructions qui correspondent à deux notions importantes de la construction de programmes :

- *l'abstraction* d'une variable x par rapport à un terme t, notée $\lambda x\ t$, que l'on peut interpréter comme une fonction de paramètre x et de corps t;

- *l'application* d'un terme t à un terme u, que l'on note $(t\ u)$ et qui correspond à l'application d'une fonction à son argument.

Ces deux règles de construction de programmes ont leur pendant au niveau des règles du raisonnement :

- l'abstraction correspond au raisonnement suivant : *"si, sous hypothèse A, j'ai démontré B alors la formule $A \rightarrow B$ est démontrée"*;

- l'application correspond à : *"si j'ai démontré $A \rightarrow B$ et si j'ai démontré A alors B est démontrée"*

De cette façon, à chaque preuve d'un énoncé logique, on peut associer un λ-terme. Sur cette base ont été développés un certain nombre de systèmes de logique formelle dont le Calcul des Constructions. Ce sont ces systèmes, dits systèmes de λ-*calcul typé*, qui permettent l'homogénéïsation du langage de programmation (le λ-calcul) et du langage de spécification (les formules logiques).

Le Calcul des Constructions original [3] a été enrichi de constructions spécifiques permettant, en particulier, la définition de certains types de données tels les listes, les arbres, etc... sous une forme proche de celle utilisée dans les langages de la famille ML [9]. A ces définitions de type de données sont associés des *opérateurs de récurrence* permettant l'écriture de fonctions récursives. Ces opérateurs de récurrence sont le pendant calculatoire du raisonnement par récurrence structurelle sur les objets du type de données [10].

Cependant, l'écriture de programmes dans ce langage enrichi reste abstruse. Nous avons donc introduit, dans le système Coq une clause de définition de

fonctions dans une syntaxe également proche de celle de ML. Toutes les définitions (même purement fonctionnelles) qu'il est possible d'écrire en ML ne peuvent être admises. Rappelons en effet que les seuls λ-termes admis dans le Calcul des Constructions sont ceux qui correspondent à une preuve. Autrement dit, pour être transformée en λ-terme, la définition d'une fonction doit pouvoir être extraite d'une preuve. Cette preuve est essentiellement une preuve de terminaison de la fonction visée, elle est recherchée automatiquement par un algorithme dédié [8].
Si notre langage de programmation est en fin de compte un noyau fonctionnel trés restreint de ML, il est apparu néanmoins suffisamment riche pour pouvoir permettre une écriture *"naturelle"* de nos programmes de tri.

Après avoir donné un peu de syntaxe, nous introduisons les principaux types de données dont nous aurons besoin. Nous énonçons ensuite quelques propriétés utiles concernant ces types de données ainsi que des fonctions de bases les manipulant.
Nous présentons alors chacun des programmes de tri auxquels nous nous sommes intéressé ainsi que les grandes lignes de leur preuve de correction. La donnée des programmes et leur preuve de correction seront souvent accompagnées d'un commentaire dans lequel nous tentons de dégager un niveau plus abstrait du fonctionnement des programmes ainsi qu'une analyse de la méthode de preuve associée. En particulier, nous mettons l'accent sur le rapport entre fonctions définies par *itération* et la notion d'*invariant* qu'elles satisfont.

1 Un peu de syntaxe

Les termes ou formules du système Coq que nous manipulerons principalement sont de trois sortes :

1. `A -> B` qui correspond à l'implication logique (si `A` alors `B`),
2. `(x:A)B` qui correspond à la quantification universelle (pour tout `x` *de type* `A`, `B`),[1]
3. `[x:A]B` qui correspond à l'abstraction de la variable `x` (de type `A`) par rapport à `B`.

2 Structures de données

Nous donnons dans cette section la définition des principales structures de données utilisées dans notre développement ainsi que la façon dont elles peuvent être déclarées en Coq.

1. et que nous écrirons parfois pour tout `x`$\in$`A`, `B`.

2.1 Les listes

L'ensemble des listes d'objets de type `A`, noté `(list A)` est défini inductivement par :

(1) `(Nil A)`∈`(list A)`, où `(Nil A)` représente la liste vide.

(2) Si `a`∈`A` et `x`∈`(list A)` alors `(Cons A a x)`∈`(list A)`, où `(Cons A a x)` représente la liste obtenue en rajoutant `a` devant `x`,

(3) toute liste est obtenue soit par *(1)*, soit par *(2)*.

Les clauses *(1)* et *(2)* de cette définition introduisent ce que l'on appelle les *constructeurs* de l'ensemble `(list A)`. Dans la vie courante, les langages de programmation ne demandent pas toujours au programmeur de préciser le type des objets que contiennent les listes : on écrirait simplement `Nil` et `(Cons a x)`. Cette surcharge syntaxique est le prix à payer pour pouvoir intégrer dans un même formalisme langage de programmation et langage logique.
La clause *(3)*, dite *clause de fermeture*, autorise à raisonner sur les listes selon une forme de raisonnement dite *récurrence structurelle* : pour établir qu'une propriété `P` est vérifiée par toute liste `x`, il faut et il suffit de montrer que

(i) `(Nil A)` vérifie `P` et

(ii) pour tout `a`∈`A`, pour tout `y`∈`(list A)`, si `y` vérifie `P` alors `(Cons A a x)` vérifie également `P`.

Le vernaculaire de Coq nous permet de définir l'ensemble `(list A)` par la clause suivante :

```
Inductive list [A:Set] : Set :=
 Nil : (list A)
|Cons : A -> (list A) -> (list A).
```

Le système se charge lui-même de définir le principe de récurrence structurelle associé qui s'énonce :

```
(A:Set)(P:(list A)->Prop)
 (P (Nil A))->((y:A)(l:(list A))(P l)->(P (Cons A y l)))
  ->(l:(list A))(P l)
```

REMARQUE : Cette formule appelle sans doute quelques commentaires. La première quatification - `(A:Set)` - vient de ce que nous avons déclaré un type de listes *polymorphe* et, en conséquence, notre principe d'induction doit valoir pour n'importe quel ensemble d'objets `A`. Le type `Set` donné à `A` est, dans Coq, le type des objets sur lesquels ont désire *calculer*; ou encore, le type des *types de données*. Il s'oppose au type `Prop` que l'on trouve dans la seconde quatification -

(P:(list A)->Prop) - qui, lui, correspond au type des *valeurs logiques* (valeurs de vérité). Ainsi, le type (list A)->Prop doit être lu comme le type des *prédicats* (unaires) sur les listes, ou encore, le type des *fonctions propositionnelles* (unaires) sur les listes.

2.2 Les arbres binaires

Nous utiliserons deux structures d'arbres binaires, comme structures intermédiaires pour le traitement des listes à trier : des arbres binaires dont les nœuds sont étiquetés; des arbres binaires dont les feuilles sont étiquetées.

Le premier type de données est défini par :

```
Inductive lnb_tree [A:Set] : Set :=
 Lf : (lnb_tree A)
|Br : A -> (lnb_tree A) -> (lnb_tree A) -> (lnb_tree A).
```

Le principe de récurrence structurelle associé au type lnb_tree est le suivant :

```
(A:Set)(P:(lnb_tree A)->Prop)
 (P (Lf A))
 ->((y:A)
    (t1:(lnb_tree A))(P t1)
    ->(t2:(lnb_tree A))(P t2)
    ->(P (Br A y t1 t2)))
  ->(t:(lnb_tree A))(P t)
```

Le type des arbres binaires dont les feuilles sont étiquetées est défini par :

```
Inductive llb_tree2 [A:Set] : Set :=
 Empty_lf : (llb_tree2 A)
|Some_lf : A -> (llb_tree2 A)
|Br2 : (llb_tree2 A) -> (llb_tree2 A) -> (llb_tree2 A).
```

Remarquons que nous nous donnons la possibilité d'étiqueter effectivement une feuille (constructeur Some_lf) ou non (constructeur Empty_lf).
Le principe de récurrence structurelle est :

```
(A:Set)(P:(llb_tree A)->Prop)
 (P (Empty_lf A))
 ->((y:A)(P (Some_lf A y)))
 ->((t1:(llb_tree A))(P t1)
    ->(t2:(llb_tree A))(P t2)
    ->(P (Br2 A t1 t2)))
  ->(t:(llb_tree A))(P t)
```

3 Propiétés attendues des listes

Nous énonçons dans cette section les propriétés qui nous serviront à spécifier la correction de nos programmes de tri. Nous donnons également deux autres propriétés dont nous aurons besoin comme auxiliaires pour effectuer certaines preuves.

3.1 Qu'est-ce qu'une liste triée ?

De façon intuitive, une *liste triée*, selon un certain ordre < sur `A`, est une liste `x` telle que : pour tout `a`∈`A`, pour tout `b`∈`A`, si `a` apparaît *"avant"* `b` dans `x` alors `a<b`. Dans la mesure où nous nous intéressons à une relation d'ordre entre les éléments des listes et puisque les relations d'ordre sont transitives, pour vérifier qu'une liste est triée, il suffit en fait de vérifier que si `a` et `b` sont deux éléments *"consécutifs"* de `x` alors `a<b`.

Formellement, nous définissons une propriété plus générale `R_list` que l'on peut énoncer comme : une liste `x` est une `R_list`, pour une certaine relation binaire `R2` sur `A`, si pour tout `a`∈`A`, pour tout `b`∈`A`, si `a` et `b` sont deux éléments *"consécutifs"* de `x` alors `(R2 a b)`.

Cette propriété peut être définie inductivement sur la structure des listes par :

(1) `(Nil A)` est une `R_list`,

(2) pour tout `a`∈`A`, `(Cons A a (Nil A))` est une `R_list`,

(3) pour tout `a1`∈`A`, pour tout `a2`∈`A`, pour tout `x`∈`(list A)`, si `(R2 a1 a2)` et si `(Cons A a2 x)` est une `R_list` alors `(Cons A a1 (Cons A a2 x))` est une `R_list`.

Ce que l'on définit en Coq par :

```
Inductive R_list : (list A)->Prop :=
 R_list_Nil : (R_list (Nil A))
|R_list_Cons_Nil : (a:A)(R_list (Cons A a (Nil A)))
|R_list_Cons_Cons : (a1,a2:A)(x:(list A))
 (R2 a1 a2)->(R_list (Cons A a2 x))
 ->(R_list (Cons A a1 (Cons A a2 x))).
```

La propriété à laquelle nous nous intéressons concerne les listes d'entiers naturels, elle est définie par particularisation de `R_list` de la façons suivante :

```
Syntactic Definition sorted_list := (R_list nat le).
```

où `le` est la relation d'ordre usuelle sur les entiers naturels.

REMARQUE : la clause `Syntactic Definition` de `Coq` est l'analogue du `#define` de C ; elle permet simplement d'engendrer une "macro" syntaxique. Nous utilisons ici ce mécanisme pour avoir à éviter d'ouvrir explicitement la définition de `sorted_list` lors du développement des preuves.

3.2 Qu'est-ce qu'une permutation ?

Une façon mathématiquement élégante d'énoncer qu'un liste `x` est une permutation d'une liste `y` est de dire que l'identité est une bijection entre `x` et `y`. Si elle est élégante, cette définition s'avère trés lourde à utiliser dans notre cadre formel. Nous avons donc choisi une formulation alternative de la propriété visée.

Nous dirons que la liste `x` est une *permutation* de la liste `y` si et seulement si, pour tout `a`$\in$`A`, le *nombre d'occurrences* de `a` dans `x` est égal au *nombre d'occurrences* de `a` dans `y`.

Pour définir formellement notre notion de permutation, il nous faut préciser ce qu'est le nombre d'occurences de `a` dans une liste `x`. Là encore, nous utilisons à une définition inductive :

(1) pour tout `a`$\in$`A`, le nombre d'occurences de `a` dans `(Nil A)` est `O`,

(2) pour tout entier `n`, pour tout `a`$\in$`A`, pour tout `x`$\in$`(list A)`, si `n` est le nombre d'occurrences de `a` dans `x` alors `n+1` est le nombre d'occurrences de `a` dans `(Cons A a x)`.

(3) pour tout entier `n`, pour tout `a1`$\in$`A`, pour tout `a2`$\in$`A`, pour tout `x`$\in$`(list A)`, si `a1`$\neq$`a2` et si `n` est le nombre d'occurrences de `a1` dans `x` alors `n` est aussi est le nombre d'occurrences de `a1` dans `(Cons A a2 x)`.

Ce qui se définit en `Coq` par :

```
Inductive Nocc_list : nat -> A -> (list A) -> Prop :=
 Nocc_Nil : (a:A)(Nocc_list O a (Nil A))
|Nocc_SCons : (n:nat)(a:A)(x:(list A))
 (Nocc_list n a x) -> (Nocc_list (S n) a (Cons A a x))
|Nocc_Cons : (n:nat)(a1,a2:A)(x:(list A))
 ~(a1=a2) -> (Nocc_list n a1 x)
 -> (Nocc_list n a1 (Cons A a2 x)).
```

3.3 Minoration et majoration d'une liste

Pour mener à bien certaines preuves, nous aurons également besoin d'exprimer le fait qu'un élément minore ou majore tous les éléments d'une liste. Ces deux propriétés sont en fait un cas particulier de la propriété plus générale : tous les éléments d'une liste satisfont un prédicat donné. Cette propriété est définie en Coq par :

```
Definition R_map :=
 [x:(list A)](a:A)(Mem A a x)->(R1 a).
```

où `R1` est un prédicat (unaire) quelconque sur le domaine `A` et `Mem` le prédicat d'appartenance d'un élément (`a`) à une liste (`x`) d'objets de type `A`.

Les propriétés de minorations, *resp.* majoration se définissent alors :

```
Definition min_list := [n0:nat](R_map nat (le n0)).
```

```
Definition maj_list := [n0:nat](R_map nat (ge n0)).
```

REMARQUE : `min_list` et `maj_list` sont en fait des *fonctions propositionnelles* puisqu'elles dépendent de l'entier abstrait `n0`. Elles ne deviendront des *propositions* que lorsque l'on les utilisera avec un entier particulier.
Remarquons également que l'on fait ici un usage important de *l'application partielle* d'un prédicats à ces *"arguments"*. Par exemple, la propriété "être plus grand que `n0`" est exprimée par application partielle du prédicat `le` au seul l'entier `n0`.

4 Fonctions utiles sur les listes et propriétés

Nous donnons dans cette sections deux fonctions utilitaires sur les listes ainsi que quelques propriétés qui s'avèreront utiles par la suite.

4.1 Le constructeur Cons

Rappelons qu'une `R_list` est une liste dont les éléments consécutifs sont deux à deux dans une certaine relation `R2`.

La première propriété que nous donnons ici nous dit que si un objet `a0` est dans la relation `R2` avec tous les éléments d'une `R_list x` alors on obtient une `R_list` en rajoutant `a0` devant `x` :

```
Lemma Cons_R_list : (a0:A)(x:(list A))
 (R_map (R2 a0) x) -> (R_list x)
 -> (R_list (Cons A a0 x)).
```

Nous utiliserons, en particulier le fait que si un entier `a0` minore les éléments d'une liste (d'entiers) triée `x` alors `(Cons nat a0 x)` est une liste triée.

Nous utiliserons également le fait que le nombre d'occurrences d'un élément dans une liste est insensible à la permutation des deux premiers éléments de celle-ci :

```
Lemma Nocc_Cons_Cons : (n:nat)(a0,a1,a2:A)(x:(list A))
 (Nocc_list n a0 (Cons A a1 (Cons A a2 x)))
 -> (Nocc_list n a0 (Cons A a2 (Cons A a1 x))).
```

Enfin nous aurons besoin de savoir que si deux listes x1 et x2 ont même nombre d'occurences d'un élément a1 alors il en va de même lorsque l'on rajoute un a2 quelconque devant x1 et x2 :

```
Lemma Nocc_Nocc_Cons : (a1:A)(x1,x2:(list A))
 ((n1:nat)(Nocc_list n1 a1 x1) -> (Nocc_list n1 a1 x2))
 -> (n2:nat)(a2:A) (Nocc_list n2 a1 (Cons A a2 x1))
    ->(Nocc_list n2 a1 (Cons A a2 x2)).
```

4.2 La concaténation

La fonction de concaténation de deux listes est définie simplement par :

```
Recursive Definition append [A:Set] :
  (list A) -> (list A) -> (list A) :=
 (Nil A) ys => ys
|(Cons A x xs) ys => (Cons A x (append A xs ys)).
```

Nous aurons esentiellement besoin de deux propriétés de la fonction append. La première est que si tous les éléments de deux listes x et y satisfont un prédicat R1 alors il en va de même pour les éléments de la concaténée de x et y :

```
Lemma R_map_append : (x,y:(list A))
 (R_map x) -> (R_map y) -> (R_map (append A x y)).
```

La seconde propriété utile de la fonction append est que le nombre d'occurences d'un élément a dans la concaténée de deux listes x1 et x2 est égal à la somme du nombre d'occurences de a dans x1 et dans x2 :

```
Lemma Nocc_append : (x1,x2:(list A))(a:A)(n1,n2:nat)
 (Nocc_list n1 a x1) -> (Nocc_list n2 a x2)
 -> (Nocc_list (add n1 n2) a (append A x1 x2)).
```

4.3 La fusion de deux listes

La fonction de fusion de deux listes est définie par :

```
Recursive Definition merge :
  (list nat) -> (list nat) -> (list nat) :=
```

```
 (Nil nat) ms => ms
|(Cons nat  n ns) (Nil nat) => (Cons nat  n ns)
|(Cons nat  n ns) (Cons nat  m ms)
  => (ifSet(list nat)
      (less_eq_nat n m)
      (Cons nat  n (merge ns (Cons nat  m ms)))
      (Cons nat  m (merge (Cons nat  n ns) ms))).
```

Nous aurons besoin de savoir que si un entier `n` minimise deux listes (d'entiers) `x` et `y` alors il en va de même pour la liste fusionnant `x` et `y` :

```
Lemma min_list_merge : (n:nat)(x,y:(list nat))
 (min_list n x) -> (min_list n y)
 -> (min_list n (merge x y)).
```

Nous aurons également besoin de ce qui fait l'intérêt de la fonction de fusion pour les algorithmes de tri, à savoir, si `x` et `y` sont deux listes triées alors leur fusion l'est également :

```
Lemma sorted_merge : (x,y:(list nat))
 (sorted_list x) -> (sorted_list y)
 -> (sorted_list (merge x y)).
```

Enfin, nous aurons besoin de savoir que le nombre d'occurences d'un objet `a` dans la fusion de deux listes `x` et `y` est égal à la somme du nombre d'occurences de `a` dans `x` et dans `y` :

```
Lemma Nocc_merge : (x,y:(list nat))(n1,n2:nat)(a:nat)
 (Nocc_list nat n1 a x) -> (Nocc_list nat n2 a y)
 -> (Nocc_list nat (add n1 n2) a (merge x y)).
```

5 Propriétés particulières des arbres

5.1 Minoration et majoration d'un arbre

De façon analogue à ce que nous avons fait pour les listes (*cf* :3.3), nous définissons la propriété, pour un entier, de minorer, *resp.* majorer, les éléments d'un arbre. Nous n'aurons en fait besoin de ces notions que pour les arbres dont les nœuds sont étiquetés (et non les feuilles). Il faut donc entendre par "éléments d'un arbre", tout ou partie de ces étiquettes.

Nous exprimons donc, dans un premier temps, le fait que tous les éléments d'un arbre satisfont une propriété `R1` donnée :

```
Definition R_mapt :=
 [t:(lnb_tree A)](a:A)(Mem_tree a t)->(R1 a).
```

Minorer et majorer, les éléments d'un arbre se définissent respectivement :

```
Definition min_tree :=
 [n0:nat](R_mapt nat (le n0)).

Definition maj_tree :=
 [n0:nat](R_mapt nat (ge n0)).
```

5.2 Qu'est-ce qu'un arbre binaire de recherche ?

Un *arbre binaire de recherche* est un arbre binaire dont les nœuds sont étiquetés par des éléments sur lesquels on possède une relation d'ordre. De plus, pour chaque nœud ν de l'arbre, si n est son étiquette, alors toutes les étiquettes du sous-arbre gauche de ν sont inférieures (ou égales) à n et toutes les étiquettes du sous-arbre droit de ν sont (strictement) supérieures à n.
L'ensemble des arbres binaires (dont les nœuds sont étiquetés par des entiers) étant défini récursivement, il est tout aussi naturel de définir récursivement la notion d'arbre binaire de recherche à l'aide des deux clauses :

(1) l'arbre vide `Lf` est un arbre binaire de recherche;

(2) si deux arbres `t1` et `t2` sont des arbres binaires de recherche et, pour tout `n`$\in$`nat`, si `n` minore `t1` et `n` majore `t2` alors l'arbre `(Br n t1 t2)` est un arbre binaire de recherche.

Ce qui se paraphrase en `Coq` par :

```
Inductive bst : (lnb_tree nat) -> Prop :=
 bst_Lf : (bst (Lf nat))
|bst_Br : (n:nat)(t1,t2:(lnb_tree nat))
 (maj_tree n t1) -> (bst t1) -> (min_tree n t2) -> (bst t2)
 -> (bst (Br nat n t1 t2)).
```

5.3 Qu'est-ce qu'un tas ?

Un *tas* est un arbre binaire équilibré[2] dont les nœuds sont étiquetés et sur les lesquels on possède une relation d'ordre. Chaque nœud ν de l'arbre est tel que l'étiquette de ν minore ses sous-arbres gauche et droit.
Nous utiliserons dans notre développement une notion plus faible de "tas" ne faisant pas intervenir la notion d'arbre équilibré. En effet, pour prouver la correction de nos fonctions de tri, la propriété essentielle des tas est qu'elle préordonne les éléments de la liste. Le fait que ce préordonnancement se fasse dans une structure d'arbre équilibré ne concerne que la complexité de la fonction et

2. Nous consacrons la section 12.2 au problème de la construction d'arbres binaires équilibrés

non sa correction. Nous définissons donc (récursivement) la notion de tas par les clauses :

(1) l'arbre vide `Lf` est un tas;

(2) si deux arbres `t1` et `t2` sont des tas et, pour tout `n`$\in$`nat`, si `n` minore `t1` et `t2` alors l'arbre `(Br n t1 t2)` est un tas.

Ce qui se paraphrase en Coq par :

```
Inductive heap : (lnb_tree nat) -> Prop :=
 heap_Lf : (heap (Lf nat))
|heap_Br : (n:nat)(a1,a2:(lnb_tree nat))
 (min_tree n a1) -> (heap a1) -> (min_tree n a2) -> (heap a2)
 -> (heap (Br nat n a1 a2)).
```

5.4 Nombre d'occurences d'un élément

Pour démontrer que les fonctions de tri qui utilisent des structures temporaires d'arbres binaires calculent bien une permutation de leur argument, nous aurons besoin de compter le nombre d'occurrences d'un élément dans un arbre. Nous devons définir cette notion pour les arbres dont les feuilles sont étiquetées et pour ceux dont les nœuds sont étiquetés. le principe de ces définitions est trés proche de celui qui a présidé à la définition du nombre d'occurences dans une liste c'est pourquoi nous en donnons directement les énoncés sans plus de commentaires.

Nombre d'occurences d'un élément dans un arbre dont les feuilles sont étiquetées :

```
Inductive Nocc_llb_tree : nat -> A -> (llb_tree A) -> Prop :=
 Nocc_Empty_lf : (a:A)(Nocc_llb_tree O a (Empty_lf A))
|Nocc_SSome_lf : (a:A)(Nocc_llb_tree (S O) a (Some_lf A a))
|Nocc_Some_lf :
 (a,b:A)~(a=b) -> (Nocc_llb_tree O a (Some_lf A b))
|Nocc_Br2 : (n1,n2:nat)(a:A)(t1,t2:(llb_tree A))
 (Nocc_llb_tree n1 a t1) -> (Nocc_llb_tree n2 a t2)
 -> (Nocc_llb_tree (add n1 n2) a (Br2 A t1 t2)).
```

Nombre d'occurences d'un élément dans un arbre dont les nœuds sont étiquetés :

```
Inductive Nocc_lnb_tree :
 nat -> A -> (lnb_tree A) -> Prop :=
 Nocc_Lf : (a:A)(Nocc_lnb_tree O a (Lf A))
|Nocc_SBr : (n1,n2:nat)(a:A)(t1,t2:(lnb_tree A))
 (Nocc_lnb_tree n1 a t1) -> (Nocc_lnb_tree n2 a t2)
```

```
 -> (Nocc_lnb_tree (S (add n1 n2)) a (Br A a t1 t2))
|Nocc_Br : (n1,n2:nat)(a1,a2:A)(t1,t2:(lnb_tree A))
 ~(a1=a2)
 -> (Nocc_lnb_tree n1 a1 t1) -> (Nocc_lnb_tree n2 a1 t2)
 -> (Nocc_lnb_tree (add n1 n2) a1 (Br A a2 t1 t2)).
```

6 Ce qu'il faudra montrer

La spécification globale d'un algorithme de tri est *"l'obtention d'une permutation ordonnée de la liste à trier"*. Cette spécification globale peut être décomposée en :

I l'obtention d'une liste triée;

II l'obtention d'une liste contenant les mêmes éléments (répétitions comprises) que la liste à trier.

Nous nous limiterons au cas des listes d'entiers. Pour démontrer la correction d'une fonction de tri, `x_sort`, nous procéderons donc en deux temps.

Correction I : la fonction `x_sort` calcule une liste triée. Ce qui s'énonce :

```
Theorem sorted_x_sort :
 (x:(list nat))(sorted_list (x_sort x)).
```

Correction II : la fonction `x_sort` calcule une permutation de son argument. Ce qui s'énonce :

```
Theorem Nocc_x_sort : (x:(list nat))(n,m:nat)
 (Nocc_list nat n m x) -> (Nocc_list nat n m (x_sort x)).
```

7 Le tri par insertion séquentielle

7.1 L'algorithme

Nous formulons, grossièrement, l'algorithme de tri par insertion séquentielle de la façon suivante : soit `x` la liste à trier,

1 : trier (récursivement) la queue de `x`,

2 : insérer à sa place le premier élément de `x` dans le résultat du tri de la queue de `x`.

7.2 Le programme

Fonction d'insertion d'un élément dans une liste :

```
Recursive Definition ins [n:nat] : (list nat) -> (list nat) :=
 (Nil nat) => (Cons nat n (Nil nat))
|(Cons nat m x)
 => (ifSet (list nat)
     (less_eq_nat n m)
     (Cons nat n (Cons nat m x))
     (Cons nat m (ins n x))).
```

Fonction principale :

```
Recursive Definition ins_sort : (list nat) -> (list nat) :=
 (Nil nat) => (Nil nat)
|(Cons nat n x) => (ins n (ins_sort x)).
```

Commentaire

Le lecteur familier de la programmation fonctionnelle aura remarqué que la fonction `ins_sort` procède par itération de la fonction d'insertion `ins`. On a l'habitude en mathématiques de parler de *"l'itérée n^e"* d'une fonction sur une valeur x, que l'on note $f^n(x)$. On la définit par induction sur n : $f^0(x) = x$ et $f^{n+1}(x) = f(f^n(x))$. *Mutatis mutandis*, on peut définir une notion d'itération pour laquelle "l'exposant" utilisé dans la notation ci-dessus est une liste. Intuitivement, l'itération d'une fonction f sur une valeur x pour une liste $[a_1, a_2, .., a_n]$ a pour résultat : $f(a_1, f(a_2, .., f(a_n, x)))$.
Notons $[\,]$ la liste vide et $a :: x$ la liste dont le premier élément est a et qui se poursuit par la liste x. La fonction it_f calculant l'itérée d'une liste sur b est définie récursivement par : $it_f([\,]) = b$ et $it_f(a :: x) = f(a, it_f(x))$. En abstrayant la fonction itérée f et la valeur b, on peut définir une fonction dite *d'ordre supérieur*, ou *fonctionnelle*, qui calcule cette forme d'itération. Pour l'exprimer dans notre formalisme. Donnons nous `A` et `B` deux ensembles quelconques de valeurs. La fonctionnelle d'itération sur les listes s'écrit :

```
Recursive Definition list_it [f:A->B->B, b:B] :
 (list A) -> B :=
 (Nil A) => b
|(Cons A a x) => (f a (list_it A B f b x)).
```

Il faut lire `[f:A->B->B, b:B]` comme la déclaration de paramètres de la fonction `list_it`. Notre fonction de tri `ins_sort` pourrait alors se définir simplement comme :

```
Definition ins_sort :=
 [x:(list nat)](list_it ins (Nil nat) x).
```

Nous n'avons pas utilisé ici cette possibilité d'abstraction pour ne pas obscurcir inutilement le code des fonctions. La présente remarque a pour but de mettre en évidence le mécanisme d'itération sur lequel nous reviendrons lorsque nous commenterons les preuves de correction.

7.3 Correction I

La correction de l'algorithme du tri par insertion repose sur la propriété suivante : insérer un élément dans une liste triée produit une liste triée. Il faut donc montrer que la fonction `ins` satisfait cette propriété. Pour ce, nous prouvons le lemme suivant :

```
Lemma sorted_ins :
 (n:nat)(x:(list nat))(sorted_list x)->(sorted_list (ins n x)).
```

La preuve est faite par récurrence sur `x` en utilisant le lemme auxiliaire :

```
Lemma sorted_ins_Cons :
 (x:(list nat))(n,m:nat)(le n m)
  -> (sorted_list (Cons nat n x))
  -> (sorted_list (Cons nat n (ins m x))).
```

La preuve de ce lemme s'obtient par induction sur `x`. Lorsque `x` n'est pas la liste vide, on considère les deux cas : `m` est plus petit que le premier élément de `x` ou non.

La propriété de correction de la fonction d'insertion étant établie, il devient trivial de montrer (par récurrence sur la liste `x`) que :

```
Theorem sorted_ins_sort :
 (x:(list nat))(sorted_list (ins_sort x)).
```

Commentaire

Nous avions remarqué en 7.2 que la fonction de tri par insertion est un cas particulier de fonction (it_f) définie par itération d'une fonction plus élémentaire (f) sur les éléments d'une liste. Le schéma général de définition de telles fonctions est : $it_f([\,]) = b$; $it_f(a :: x) = f(x, it_f(x))$.
Nous voulons à présent montrer que pour toute liste x, le résultat du calcul de $it_f(x)$ satisfait une certaine propriété P, ce que nous notons $P[it_f(x)]$. Pour cela, nous procédons par récurrence sur x. Ce qui nous donne à examiner deux cas : $P[b]$ et $P[it_f(a :: x')]$ sous l'hypothèse (de récurence) que $P[it_f(x')]$. La résolution du premier cas dépend essentiellement du choix de la valeur de b. Pour traiter le second cas, nous utilisons la définition de la fonction d'itération pour nous ramener à démontrer $P[f(a, it_f(x'))]$ sous l'hypothèse $P[it_f(x')]$. Nous faisons ici un pas d'abstraction supplémentaire en montrant qu'il est conséquence

d'un résultat plus général concernant f. A savoir : pour toute liste x et tout objet a, si $P[x]$ alors $P[f(a,x)]$. En terme de preuve de programme, nous montrons que P est un *invariant* pour f. Le lemme `sorted_ins` est un exemple de tel invariant.

7.4 Correction II

Pour montrer que le nombre d'occurences d'un entier quelconque `n` est conservé par la fonction `ins_sort`, on montre qu'il revient au même d'ajouter un élément à une liste en tête de celle-ci (en utilisant le constructeur `Cons`) que de l'y ajouter en utilisant la fonction d'insertion `ins`. Cette propriété est établie par le lemme suivant :

```
Lemma Nocc_ins_Cons : (x:(list nat))(n,m,p:nat)
 (Nocc_list nat n p (Cons nat m x))
 -> (Nocc_list nat n p (ins m x)).
```

On démontre alors trivialement par récurrence sur la liste `x` que :

```
Theorem Nocc_ins_sort : (x:(list nat))(n,m:nat)
 (Nocc_list nat n m x) -> (Nocc_list nat n m (ins_sort x)).
```

REMARQUE : la preuve de `Nocc_ins_Cons` utilise les lemmes `Nocc_Cons_Cons` et `Nocc_Nocc_Cons` énoncés en 4.1

Commentaire

Nous ne pouvons pas utiliser ici le schéma de preuve faisant intervenir un invariant. En effet, la propritété P recherchée s'écrit, dans le cas présent sous la forme $P[x] \rightarrow P[it_f(x)]$, ce que l'on peut noter plus simplement $Q[x, it_f(x)]$. Au second cas de l'induction sur x, nous devons montrer $Q[n :: x', f(n, it_f(x'))]$ sous l'hypoyhèse $Q[x', it_f(x')]$. Il est inutile de généraliser $it_f(x')$ pour chercher à montrer que pour toute liste y, si $Q[x', y]$ alors $Q[n :: x', f(n, y)]$ car la validité de cet énoncé peut dépendre du rapport entre x' est y; ce que masque précisément notre généralisation.
Dans le cas du tri par insertion séquentielle, nous chercherions à montrer que si a apparaît n fois dans une liste $a' :: x'$ alors il apparaît également n fois dans la liste obtenue en insérant a' dans une liste quelconque y. Ce qui a peu de raison d'être vrai en général !

8 Le tri par fusion

8.1 L'algorithme

Nous formulons de la façon suivante de l'algorithme du tri par fusion : soit `x` la liste à trier,

1 : scinder (par le milieu) `x` en `x1` et `x2`

2 : trier (récursivement) `x1` et `x2`

3 : fusionner les résultats des tris de `x1` et de `x2`

8.2 Le programme

Pour implémenter le tri par fusion, nous utilisons une structure intermédiaire d'arbre binaire dont les feuilles sont étiquetées. Cette structure intermédiaire permet une implémentation raisonnable de l'opération *"scinder la liste par le milieu"* si les constituants de la liste originale sont dans un premier temps tranférés dans un arbre équilibré (*cf* :12.2).

Fonction d'insertion d'un élément dans un arbre binaire dont les feuilles sont étiquetées :

```
Recursive Definition ins_baltree [n:nat] :
 (llb_tree nat) -> (llb_tree nat) :=
 (Empty_lf nat) => (Some_lf nat n)
|(Some_lf nat m) => (Br2 nat (Some_lf nat n) (Some_lf nat m))
|(Br2 nat a1 a2) => (Br2 nat a2 (ins_baltree n a1)).
```

Fonction de stockage des éléments de la liste à trier dans une structure d'arbre équilibré.

```
Recursive Definition baltree_of_list :
 (list nat) -> (llb_tree nat) :=
 (Nil nat) => (Empty_lf nat)
|(Cons nat n x)
 => (ins_baltree n (baltree_of_list x)).
```

Fonction de "dépliage" de l'arbre.

```
Recursive Definition it_merge :
 (llb_tree nat) -> (list nat) :=
 (Empty_lf nat) => (Nil nat)
|(Some_lf nat n) => (Cons nat n (Nil nat))
|(Br2 nat a1 a2) => (merge (it_merge a1) (it_merge a2)).
```

Fonction principale.

```
Definition merge_sort := [x:(list nat)]
 (it_merge (baltree_of_list x)).
```

Commentaires

Nous retrouvons dans la fonction de stockage `baltree_of_list` le principe d'itération sur une liste. Mais nous pouvons remarquer que la fonction de "dépliage" `it_merge`, comme son nom le suggère, est également construite sur un principe d'itération. La donnée servant à l'itération est ici un arbre binaire. Le type des arbres binaires manipulés lors du tri par fusion est construit à partir de deux cas de base, (`Empty_Lf` et `Some_Lf`), et d'un cas récursif (`Br2`) faisant appel à deux arbres supposés déjà construits. Le schéma de définition de l'itération pour les arbres binaires a, au cas récursif, la forme $it_f(Br(x_1, x_2)) = f(it_f(x_1), it_f(x_2))$. Sa particularité, par rapport aux itérations pour les entiers et les listes, est qu'apparaissent ici deux appels récursifs de la fonction d'itération correspondant aux deux sous-arbres gauche et droit.

8.3 Correction I

La correction du tri par fusion repose essentiellement sur le fait que la fonction de dépliage calcule une liste triée, quel que soit l'arbre déplié.

```
Lemma sorted_it_merge :
 (t:(llb_tree nat))(sorted_list (it_merge t)).
```

On en déduit, par définition de `merge_sort` :

```
Theorem sorted_merge_sort :
 (x:(list nat))(sorted_list (merge_sort x)).
```

Remarque : Pour prouver la correction de la fonction de dépliage `it_merge`, on utilise le fait que la fonction itérée `merge` calcule une liste triée lorsqu'on lui donne en argument deux listes triées (*cf* :lemme `sorted_merge`, 4.3).

Commentaire

Le lemme `sorted_merge` que nous utilisons ici peut être considéré comme un cas limite d'utilisation d'un invariant de la fonction itérée.
Nous avons introduit la notion d'invariant d'une fonction f à propos de la preuve de correction du tri par insertion séquentielle sous la forme : pour toute liste s et tout x, si $P[s]$ alors $P[f(x,s)]$. Si l'on voit f comme un processus traitant une structure s, on aura tendance à formuler l'invariance de f pour P comme : si P est vérifié *"avant"* traitement f alors P est toujours vérifié *"aprés"*. Cette

vision de l'invariance en termes de *"pré-"* et *"post-"* rend difficile la formulation de l'invariance de notre fonction de fusion. En effet, si l'on peut voir la fonction d'insertion comme un processus de modification d'une structure, cette vision opératoire devient moins naturelle lorsqu'il s'agit de fabriquer une liste à partir de deux listes.
Si, en revanche, on conserve le caractère fonctionnel de la fonction d'insertion, son invariance s'exprime comme : si l'*argument* vérifie P alors le *résultat* vérifie également P. Dés lors, la seule différence entre l'expression d'un invariant pour la fonction d'insertion et pour la fonction de fusion est que là où la première n'avait besoin que d'un seul argument, la seconde en a besoin de deux et il suffit de mettre au pluriel le sujet de notre prémisse pour exprimer une notion d'invariance valant de façon plus générale : $P[x_1] \to P[x_2] \to P[f(x_1, x_2)]$. D'ailleurs le lemme `sorted_merge` que nous utilisons dans la preuve de correction du tri par fusion n'exprime pas autre chose que : si deux listes sont triées alors le résultat de leur fusion est également triée.

8.4 Correction II

La fonction de tri par fusion est définie par composition d'une fonction de stockage et d'une fonction de dépliage. Pour prouver qu'aucun élément n'est perdu lors du tri, nous procédons en deux temps : nous montrons d'abord que le fonction de stockage n'oublie personne, puis nous montrons que la fonction de dépliage n'oublie également personne.

La correction de la fonction de stockage vient de ce que la fonction d'insertion d'un élément `a` dans un arbre `t` ajoute *effectivement* `a` à `t`. En d'autres termes, insérer `a` dans `t` augmente de un le nombre d'occurences de `a` dans `t`. C'est ce qu'exprime le lemme :

```
Lemma Nocc_Sins_baltree : (t:(llb_tree nat))(n:nat)(a:nat)
 (Nocc_llb_tree nat n a t)
 -> (Nocc_llb_tree nat (S n) a (ins_baltree a t)).
```

Nous utiliserons également le fait que l'insertion d'un élément `a` dans `t` ne modifie pas le nombre d'occurrences d'éléments autres que `a`. Ce qui s'exprime :

```
Lemma Nocc_ins_baltree : (t:(llb_tree nat))(n:nat)(a,b:nat)
 ~(a=b) -> (Nocc_llb_tree nat n a t)
 -> (Nocc_llb_tree nat n a (ins_baltree b t)).
```

Ces deux lemmes permettent d'établir qu'aucun élément de la liste à trier n'est oublié lors du stockage transitoire :

```
Lemma Nocc_list_Nocc_lnb_tree : (x:(list nat))(n:nat)(a:nat)
 (Nocc_list nat n a x)
 -> (Nocc_llb_tree nat n a (baltree_of_list x)).
```

Le fait que tous les éléments de l'arbre se retrouvent dans la liste calculée par `it_merge` s'exprime :

```
Lemma Nocc_lnb_tree_Nocc_list :
 (t:(llb_tree nat))(n:nat)(a:nat)
 (Nocc_llb_tree nat n a t)
 -> (Nocc_list nat n a (it_merge t)).
```

On déduit des deux résultats précédents :

```
Theorem Nocc_merge_sort : (n:nat)(a:nat)(x:(list nat))
 (Nocc_list nat n a x) -> (Nocc_list nat n a (merge_sort x)).
```

REMARQUE : Pour montrer que rien n'est oublié lors de dépliage de l'arbre, nous utilisons le lemme `Nocc_merge` énoncé en 4.3.

Commentaire

La fonction de stockage est définie par itération d'une fonction d'insertion, mais nous avons vu en 7.4 que la propriété visée ici ne permet pas l'utilisation de la méthode de l'invariant. Nous ne pouvons pas non plus reproduire le schéma de preuve utilisé pour le tri par insertion séquentielle car l'itération se fait ici pour un arbre binaire et non pas pour une liste. La méthode de preuve mise en œuvre ici repose sur la remarque que la fonction d'insertion se comporte vis-à-vis du nombre d'occurences comme se comporte le constructeur `Some_lf`. Cette analogie de comportement est exprimée par l'analogie entre les lemmes `Nocc_ins_baltree` et `Nocc_Sins_baltree`, et les clauses `Nocc_Somme_lf` et `Nocc_SSome_lf` de la définition du prédicat `Nocc_llb_tree`.
Nous retrouverons ce schéma de preuve pour les deux autres fonctions de tri utilisant une structure intermédaire d'arbre : `bst_sort` et `heap_sort`.

9 Le tri par arbre binaire de recherche

9.1 L'algorithme

Le tri par arbre binaire de recherche est une variante fonctionnelle du tri par insertion dichotomique. Là encore, la structure des listes nous oblige à passer par une structure intermédiaire d'arbres binaires. Le caractère *"dichotomique"* de l'insertion tient à ce que l'arbre intermédiaire construit est un arbre binaire de recherche tel que nous l'avons décrit en 5.2.

L'algorithme que nous implémentons peut se formuler :

`1:` stocker tous les éléments de la liste dans un arbre binaire de recherche

`2:` extraire (récursivement) les éléments de l'arbre selon l'ordre suivant :

  `2.1:` extraire le sous-arbre gauche

2.2: extraire la racine

2.3: extraire le sous-arbre droit

9.2 Le programme

Fonction d'insertion d'un élément dans un arbre (binaire de recherche) :

```
Recursive Definition bst_ins :
 nat -> (lnb_tree nat) -> (lnb_tree nat) :=
 n (Lf nat) => (Br nat n (Lf nat) (Lf nat))
|n (Br nat m t1 t2)
 => (ifSet (lnb_tree nat)
     (less_eq_nat n m)
     (Br nat m (bst_ins n t1) t2)
     (Br nat m t1 (bst_ins n t2))).
```

Fonction de stockage des éléments de la liste à trier dans un arbre binaire de recherche :

```
Recursive Definition bst_of_list :
 (list nat) -> (lnb_tree nat) :=
 (Nil nat) => (Lf nat)
|(Cons nat n ns) => (bst_ins n (bst_of_list ns)).
```

Fonction d'extraction d'une liste à partir d'un arbre binaire de recherche :

```
Recursive Definition list_of_bst :
 (lnb_tree nat) -> (list nat) :=
 (Lf nat) => (Nil nat)
|(Br nat n t1 t2)
 => (append nat (list_of_bst t1) (Cons nat n (list_of_bst t2))).
```

Fonction principale :

```
Definition bst_sort : (list nat) -> (list nat) :=
  [ns:(list nat)] (list_of_bst (bst_of_list ns)).
```

Commentaire

La fonction de stockage est clairement définie selon le shcéma d'itération donné en 7.2.
L'apparente complexité du second cas de définition de la fonction d'extraction `list_of_bst` ne doit pas nous cacher qu'il s'agit simplement ici, encore, d'une fonction définie par itération sur les arbres. En posant $append(x_1, Cons(n, x_2)) = f(n, x_1, x_2)$, nous retrouvons un schéma d'itération proche de celui donné en 8.2 : $it_f(Br(a, x_1, x_2)) = f(a, it_f(x_1), it_f(x_2))$.

9.3 Correction I

La fonction de tri par arbre binaire de recherche est définie par composition d'une fonction de stockage et d'une fonction d'extraction. Sa correction vient essentiellement du fait que l'on sait extraire une liste triée d'un arbre binaire de recherche. Mais nous ne pourrons utiliser ce résultat que si nous montrons que la première phase du traitement produit effectivement un arbre binaire de recherche.

Nous établissons d'abord que la fonction d'insertion préserve les propriétés de minoration et de majoration :

```
Lemma min_tree_ins : (t:(lnb_tree nat))(n,m:nat)
 (le n m) -> (min_tree n t) -> (min_tree n (bst_ins m t)).

Lemma maj_tree_ins : (t:(lnb_tree nat))(n,m:nat)
 (ge n m) -> (maj_tree n t) -> (maj_tree n (bst_ins m t)).
```

On en déduit l'invariance de la fonction d'insertion pour les arbres binaires de recherche :

```
Lemma bst_bst_ins :
 (n:nat)(t:(lnb_tree nat))(bst t) -> (bst (bst_ins n t)).
```

Ce qui nous donne la correction de la fonction de construction de l'arbre binaire de recherche :

```
Lemma bst_bst_of_list : (x:(list nat))(bst (bst_of_list x)).
```

Pour démontrer la correction de la fonction d'extraction, il nous faut montrer, dans un premier temps, la propriété essentielle des tris par dichotomie, à savoir : si un èlément `n` majore une liste triée `x` et minore une liste triée `y` alors la liste obtenue en concaténant `x`, puis `n` puis `y` est une liste triée :

```
Lemma sorted_middle : (n:nat)(x,y:(list nat))
 (sorted_list x) -> (maj_list n x)
 -> (sorted_list y) -> (min_list n y)
 -> (sorted_list (append nat x (Cons nat n y))).
```

Dans un second temps, afin de pouvoir utiliser le lemme précédent, nous montrons que la fonction d'extraction transpose les propriétés de minoration (*resp.* de majoration) des arbres vers les listes :

```
Lemma min_tree_min_list : (n:nat)(t:(lnb_tree nat))
 (min_tree n t) -> (min_list n (list_of_bst t)).

Lemma maj_tree_maj_list : (n:nat)(t:(lnb_tree nat))
 (maj_tree n t) -> (maj_list n (list_of_bst t)).
```

On déduit de ce qui précède que :

```
Lemma bst_sorted : (t:(lnb_tree nat))
 (bst t) -> (sorted_list (list_of_bst t)).
```

Enfin, en combinant la correction de la fonction de stockage et de la fonction d'extraction, on obtient :

```
Theorem sorted_list_bst : (x:(list nat))(sorted_list (bst_sort x)).
```

REMARQUE : les preuves de `min_tree_min_list` et `maj_tree_maj_list` utilisent le lemme général `R_map_append` énoncé en 4.2.

Commentaire

Comme nous l'avons remarqué en 9.2, la fonction de stockage est définie par intération d'une fonction d'insertion. Pour démontrer qu'elle produit bien la structure d'arbre attendue, nous utiliserons l'invariance de la fonction d'insertion pour la structure d'arbre binaire de recherche.
Afin de montrer cette invariance, par définition des arbres binaires de recherche, nous avons besoin de montrer que la fonction d'insertion satisfait en fait deux invariances plus fines qu'expriment les lemmes `min_tree_ins` et `maj_tree_ins` et qui permettent de mener à bien la preuve d'invariance de la fonction d'insertion dans le cas où l'arbre n'est pas vide. Ces propriétés d'invariance pour la minimisation et la majoration sont requises par la définition des arbres binaires de recherche.

Nous avons également noté en 9.2 que la définition de la fonction d'extraction suit le schéma d'itération sur les arbres. En première analyse, la démonstration de la correction de la fonction d'extraction (lemme `bst_sorted`) ne semble pas suivre un schéma de preuve par invariant. Cependant, le lemme clef de la validité des tris dichotomiques (`sorted_middle`) exprime une propriété d'invariance.
Reposons donc $append(x, Cons(n, y)) = f(n, x, y)$, nous pouvons alors écrire le lemme `sorted_middle` sous la forme $Q[n, x, y] \to P[x] \to P[y] \to P[f(n, x, y)]$, où P est le prédicat `sorted_list` et Q représente la conjonction des propriétés de minoration et majoration supposées, respectivement, entre n et x et entre n et y. Si l'on sait établir $Q[n, x, y]$ alors nous retrouvons une propriété d'invariance analogue à celle introduite en 8.3. Dans le cas présent, nous dirions que nous avons affaire à une *invariance conditionnelle* de la fonction itérée.

9.4 Correction II

La preuve que tous les éléments de la liste originale se retrouvent bien dans la liste triée procède selon le même schéma, *mutatis mutandis* que celui utilisé en 8.4.

Nous montrons d'abord que la fonction d'insertion reflète fidèlement les clauses `Nocc_SBr` et `Nocc_Br` de la définition du nombre d'occurences d'un élément dans un arbre binaire (dont les nœuds sont étiquetés) :

```
Lemma Nocc_Sbst_ins : (t:(lnb_tree nat))(n:nat)(a0:nat)
 (Nocc_lnb_tree nat n a0 t)
 -> (Nocc_lnb_tree nat (S n) a0 (bst_ins a0 t)).

Lemma Nocc_bst_ins : (t:(lnb_tree nat))(n:nat)(a1,a2:nat)
 ~(a1=a2) -> (Nocc_lnb_tree nat n a1 t)
 -> (Nocc_lnb_tree nat n a1 (bst_ins a2 t)).
```

Nous en déduisons qu'aucun élément n'est perdu lors du transfert dans l'arbre binaire de recherche :

```
Lemma Nocc_list_Nocc_lnb_tree : (x:(list nat))(n:nat)(a:nat)
 (Nocc_list nat n a x)
 -> (Nocc_lnb_tree nat n a (bst_of_list x)).
```

Nous montrons qu'aucun élément n'est perdu lors de l'extraction de la liste à partir de l'arbre :

```
Lemma Nocc_lnb_tree_Nocc_list :
 (t:(lnb_tree nat))(n:nat)(a:nat)
 (Nocc_lnb_tree nat n a t)
 -> (Nocc_list nat n a (list_of_bst t)).
```

Et nous en déduisons de ces deux lemmes que :

```
Theorem Nocc_bst_sort : (x:(list nat))(n:nat)(a:nat)
 (Nocc_list nat n a x) -> (Nocc_list nat n a (bst_sort x)).
```

REMARQUE : pour démontrer que personne n'est oublié lors de l'extraction, nous utilisons le lemme `Nocc_append` énoncé en 4.2.

10 Le tri par tas

10.1 L'algorithme

La structure de l'algorithme du tri par tas est la même que celle du tri par arbre binaire. Sa particularité est que la fonction d'insertion doit maintenir un arbre équilibré (*cf* :12.2).

L'algorithme que nous implémentons peut se formuler ainsi :

`1`: stocker tous les éléments de la liste dans un tas;

`2`: extraire (récursivement) les éléments de l'arbre selon le processus suivant :

  `2.1`: extraire le sous-arbre gauche;

2.2: extraire le sous-arbre droit;

2.3: fusionner les résultats de 2.1 et 2.2;

2.4: concaténer la racine avec le résultat de 2.3.

REMARQUE : nous avons choisi d'implémenter la fonction d'extraction de la liste non pas en maintenant une structure de tas par fusion des sous arbres, mais en destructurant (récursivement) le tas puis en fusionnant les listes triées ainsi obtenues.

10.2 Le programme

Fonction d'insertion dans un tas :

```
Recursive Definition heap_ins :
 nat -> (lnb_tree nat) -> (lnb_tree nat) :=
 n (Lf nat) => (Br nat n (Lf nat) (Lf nat))
|n (Br nat m a1 a2)
 => (ifSet (lnb_tree nat)
     (less_eq_nat n m)
     (Br nat n a2 (heap_ins m a1))
     (Br nat m a2 (heap_ins n a1))).
```

Fonction de stockage des éléments de la liste dans un tas (par itération de la fonction d'insertion) :

```
Recursive Definition heap_of_list :
 (list nat) -> (lnb_tree nat) :=
 (Nil nat) => (Lf nat)
|(Cons nat n ns) => (heap_ins n (heap_of_list ns)).
```

Fonction d'extraction de la liste triée à partir d'un tas :

```
Recursive Definition list_of_heap :
 (lnb_tree nat) -> (list nat) :=
 (Lf nat) => (Nil nat)
|(Br nat n a1 a2)
 => (Cons nat n (merge (list_of_heap a1) (list_of_heap a2))).
```

Fonction principale (composition de la fonction de stockage et de la fonction d'extraction) :

```
Definition heap_sort :=
 [x:(list nat)](list_of_heap (heap_of_list x)).
```

10.3 Correction I

Le schéma général de preuve du fait que la fonction de tri par tas calcule une liste triée suit celui de la preuve du tri par arbre binaire de recherche : correction de la fonction de stockage; correction de la fonction d'extraction.

Pour montrer la correction de la fonction de stockage, suivant la définition des tas, nous montrons d'abord l'invariance de la fonction d'insertion pour la propriété de minoration :

```
Lemma min_tree_ins : (t:(lnb_tree nat))(n,m:nat)
 (le n m) -> (min_tree n t) -> (min_tree n (heap_ins m t)).
```

Nous utilisons également une conséquence de la transitivité de l'ordre des entiers sur la propriété de minoration :

```
Lemma min_tree_trans : (n,m:nat)(t:(lnb_tree nat))
 (le n m) -> (min_tree m t) -> (min_tree n t).
```

Nous pouvons alors montrer l'invariance de la fonction d'insertion :

```
Lemma heap_heap_ins : (t:(lnb_tree nat))(n:nat)
 (heap t) -> (heap (heap_ins n t)).
```

Et nous en déduisons la correction de la fonction de construction des tas :

```
Lemma heap_heap_of_list : (x:(list nat))
 (heap (heap_of_list x)).
```

Afin de démontrer la correction de la fonction d'extraction, nous établissons d'abord, comme nous l'avons fait dans le cas du tri par arbre binaire de recherche, que la fonction d'extraction transpose la propriété de minoration des arbres vers les listes :

```
Lemma min_tree_min_list : (n:nat)(t:(lnb_tree nat))
 (min_tree n t) -> (min_list n (list_of_heap t)).
```

Puis nous montrons que la fonction d'extraction calcule une liste triée :

```
Lemma sorted_list_of_heap : (t:(lnb_tree nat))
 (heap t) -> (sorted_list (list_of_heap t)).
```

Des deux lemmes de correction partielle, nous déduisons que la fonction de tri par tas calcule une liste triée :

```
Theorem sorted_heap : (x:(list nat))
 (sorted_list (heap_sort x)).
```

Commentaire

Du point de vue de leur structure aussi bien que du point de vue de leur preuve de correction, les fonctions de tri par arbre binaire et de tri par tas sont extrêmement proches. Le lemme `sorted_merge` énoncé en 4.3 joue dans la preuve du tri par tas, un rôle analogue à la propriété essentielle des tris dichotomique (`sorted_middle`).

10.4 Correction II

Le shéma de la preuve que tous les éléments de la liste à trier se retrouvent dans la liste calculée par la fonction de tri par tas suit celui utilisé pour la fonction de tri par arbre binaire de recherche ainsi que celui du tri par fusion qui utilise également une structure transitoire d'arbre binaire. La seule différence est que la définition de la fonction d'insertion dans un tas nous force à montrer la conjonction des propriétés de conservation ou d'incrémentation du nombre d'occurences d'un élément dans un arbre.

Nous établissons donc dans une premier temps que la fonction d'insertion reflète fidèlement les clause `Nocc_Br` et `Nocc_SBr` de la définition du nombre d'occurences d'un élément dans un arbre :

```
Lemma Nocc_Sins_and_Nocc_ins :
 (t:(lnb_tree nat))(n:nat)(a0:nat)
 (Nocc_lnb_tree nat n a0 t)
 -> ((a:nat)
      ~(a0=a) -> (Nocc_lnb_tree nat n a0 (heap_ins a t)))
    /\ (Nocc_lnb_tree nat (S n) a0 (heap_ins a0 t)).

Lemma Nocc_Sins : (t:(lnb_tree nat))(n:nat)(a0:nat)
 (Nocc_lnb_tree nat n a0 t)
 -> (Nocc_lnb_tree nat (S n) a0 (heap_ins a0 t)).

Lemma Nocc_ins : (t:(lnb_tree nat))(n:nat)(a1,a2:nat)
 ~(a1=a2) -> (Nocc_lnb_tree nat n a1 t)
 -> (Nocc_lnb_tree nat n a1 (heap_ins a2 t)).
```

Ce qui nous permet de montrer que tous les éléments de la liste à trier sont bien stockés dans le tas ;

```
Lemma Nocc_list_Nocc_lnb_tree : (x:(list nat))(n:nat)(a:nat)
 (Nocc_list nat n a x)
 -> (Nocc_lnb_tree nat n a (heap_of_list x)).
```

Nous montrons ensuite que rien n'est perdu lors de l'extraction :

```
Lemma Nocc_lnb_tree_Nocc_list :
```

```
 (t:(lnb_tree nat))(n:nat)(a:nat)
 (Nocc_lnb_tree nat n a t)
 -> (Nocc_list nat n a (list_of_heap t)).
```

Enfin, nous déduisons de ce qui précède la correction de la fonction de tri par tas :

```
Theorem Nocc_heap_sort : (x:(list nat))(n:nat)(a:nat)
 (Nocc_list nat n a x)
 -> (Nocc_list nat n a (heap_sort x)).
```

11 Le tri bulle

11.1 L'algorithme

Appelons *échange* l'opération qui consiste à permuter deux éléments consécutifs d'une liste, notons `x[i]` le `i`^{e} élément d'une liste. L'algorithme du tri bulle peut se décrire ainsi : soit `x` une liste de longueur `n`,

> Pour `i` variant de `n` à `0` faire :
>
> > Pour `j` variant de `1` à `i` faire :
> >
> > > échanger `x[j]` et `x[j+1]`, si `x[j]` est plus grand que `x[j+1]`.

Cet algorithme utilise deux itérations imbriquées. La plus interne est l'itération du processus d'échange de deux éléments consécutifs. La second est l'itération de la première sur un segment initial décroissant de la liste à trier.

11.2 Le programme

Fonction d'itération de l'échange :

```
Recursive Definition bubble :
 nat -> (list nat) -> (list nat) :=
 O  x => x
|(S p) (Cons nat n (Cons nat m x))
 => (ifSet (list nat)
       (less_eq_nat n m)
       (Cons nat n (bubble p (Cons nat m x)))
       (Cons nat m (bubble p (Cons nat n x))))
|(S p) (Nil nat) => (Nil nat)
|(S p) (Cons nat n (Nil nat)) => (Cons nat n (Nil nat)).
```

Fonction d'itération de l'itération de l'échange :

```
Recursive Definition it_bubble :
```

```
 nat -> (list nat) -> (list nat) :=
 O x => x
|(S n) x => (it_bubble n (bubble n x)).
```

Fonction principale :

```
Definition bubble_sort : (list nat) -> (list nat) :=
 [x:(list nat)](it_bubble (length nat x) x).
```

11.3 Correction I

Le lemme clef de la preuve est que `(it_bubble n)` trie les `n` premiers éléments d'une liste. Pour établir ce lemme clef, nous avons besoin de deux sortes de résultats. La première permet d'établir qu'une fois qu'un élément est placé, il n'est plus affecté par les itérations suivantes. La seconde série de lemmes permet d'établir qu'un élément placé majore ceux qui le précèdent.

Le premier lemme de la première série nous dit qu'en procédant au plus à `n` échanges sur une liste `x`, le `n+2`^{e} élément n'est pas modifié.

```
Lemma bubble_Nth : (n,m:nat)(a:nat)(x:(list nat))
 (le m n) -> (Nth nat (S(S n)) a (bubble m x))
 -> (Nth nat (S(S n)) a x).
```

On en déduit qu'en itérant au plus `n` fois la fonction d'échange (c'est ce que fait `(it_bubble n)`) on n'affecte pas le `n+1`^{e} élément.

```
Lemma it_bubble_Nth : (m,n:nat)(a:nat)(x:(list nat))
 (le m n) -> (Nth nat (S n) a (it_bubble m x))
 -> (Nth nat (S n) a x).
```

Et l'on obtient, en substance, qu'une fois que `bubble` a mis en place un élément, il ne bougera plus.

```
Lemma Nth_it_bubble_Nth_bubble : (n:nat)(a:nat)(x:(list nat))
 (Nth nat (S n) a (it_bubble (S n) x))
 -> (Nth nat (S n) a (bubble n x)).
```

Ici commence la seconde série de lemmes qui nous mène au résultat de majoration annoncé *supra*.
Nous commençons par montrer que si `a` majore les `n` premiers éléments d'une liste `x` et si l'on itère l'échange jusqu'à `n+1` au maximum alors `a` majore toujours les `n` premiers éléments de la liste obtenue.

```
Lemma maj_nfirst_bubble1 :
 (m,n:nat)(a:nat)(x:(list nat))
 (le m (S n)) -> (maj_list a (nfirst nat n x))
 -> (maj_list a (nfirst nat n (bubble m x))).
```

Nous montrons ensuite que, pour une liste non vide, le `n+1`^{e} élément obtenu après `n` échanges est supérieur ou égal au premier élément de la liste de départ.

```
Lemma Nth_bubble_le : (n:nat)(a,b:nat)(x:(list nat))
 (Nth nat (S n) a (bubble n (Cons nat b x)))
  -> (le b a).
```

Ce qui nous sert à montrer que si on itère `n` fois l'échange, alors le `n+1`^{e} élément de la liste obtenue majore ses `n` premiers éléments.

```
Lemma maj_nfirst_bubble2 :
 (n:nat)(a:nat)(x:(list nat))
 (Nth nat (S n) a (bubble n x))
 -> (maj_list a (nfirst nat n (bubble n x))).
```

Nous établissons ensuite que si `a` majore les `n` premiers éléments d'une liste `x` alors, après au plus `n+1` itérations de la fonction d'échange, `a` majore toujours les `n` premiers éléments de la liste obtenue.

```
Lemma maj_nfirst_it_bubble1 : (m,n,a:nat)(x:(list nat))
 (le m (S n)) -> (maj_list a (nfirst nat n x))
 -> (maj_list a (nfirst nat n (it_bubble m x))).
```

On déduit de ce qui précède que le `n+1`^{e} élément de la liste obtenue après `n+1` itérations de la fonction d'échange majore ses `n` premiers éléments.

```
Lemma maj_nfirst_it_bubble2 : (n:nat)(a:nat)(x:(list nat))
 (Nth nat (S n) a (it_bubble (S n) x))
 -> (maj_list a (nfirst nat n (it_bubble (S n) x))).
```

Avant d'établir notre lemme clef, nous montrons que les fonctions `bubble` et `it_bubble` n'affectent pas la longueur de la liste traitée.

```
Lemma length_bubble : (n:nat)(x:(list nat))
 (length nat x)=(length nat (bubble n x)).

Lemma length_it_bubble : (n:nat)(x:(list nat))
 (length nat x)=(length nat (it_bubble n x)).
```

Nous pouvons maintenant établir notre lemme clef : pour tout `n` inférieur ou égal à la longueur d'une liste `x`, les `n` premiers éléments de la liste obtenue après `n` itérations de la fonction d'échange sont triés.

```
Lemma sorted_nfirst_it_bubble : (n:nat)(x:(list nat))
 (le n (length nat x))
 -> (sorted_list (nfirst nat n (it_bubble n x))).
```

On en déduit que `bubble_sort` calcule une liste triée.

```
Theorem sorted_bubble_sort :
 (x:(list nat))(sorted_list (bubble_sort x)).
```

Commentaires

Cette preuve est la plus compliquée de la série des algorithmes de tris présentée ici. Sa difficulté vient de la façon peut naturelle dont l'algorithme traite les listes. D'une part, la liste triée est construite *"par la fin"*, d'autre part, la fonction `bubble` n'est itérée que sur un segment initial de la liste.
Plus finement, la difficulté de la preuve tient à ce que l'itération de la fonction `bubble` est faite selon un entier et non selon la liste traitée. C'est ce que manifeste, en particulier, les deux derniers cas *pathologiques* de la définition de `bubble` : le premier argument est de la forme `(S p)` et le second, de la forme `(Nil nat)` ou `(Cons nat n (Nil nat))`. La difficulté de la preuve tient alors à ce qu'il faut mettre explicitement en rapport l'entier qui sert à l'itération et la structure de la liste traitée.

11.4 Correction II

La preuve que la fonction `bubble_sort` calcule une permutation de son argument s'obtient en trois étapes simples.

On montre d'abord que la fonction d'échange calcule une permutation de son argument.

```
Lemma Nocc_bubble : (m:nat)(x:(list nat))(n,a:nat)
 (Nocc_list nat n a x) -> (Nocc_list nat n a (bubble m x)).
```

On en déduit qu'il en va de même en itérant la fonction d'échange.

```
Lemma nocc_it_bubble : (m:nat)(x:(list nat))(n,a:nat)
 (Nocc_list nat n a x)
 -> (Nocc_list nat n a (it_bubble m x)).
```

Ce qui nous donne le résultat escompté.

```
Theorem Nocc_bubble_sort : (x:(list nat))(n,a:nat)
 (Nocc_list nat n a x)
 -> (Nocc_list nat n a (bubble_sort x)).
```

12 Les arbres binaires équilibrés

Nous utilisons dans nos programmes de tri par tas et de tri par fusion une structure intermédiaire d'arbres que nous avons qualifiés d'*équilibrés*. C'est cette propriété qui nous permet d'affirmer que les algorithmes qu'implémentent nos fonctions se comportent, du point de vue de leur complexité en particulier, comme le prétend le nom que nous leur avons donné. Nous n'étudierons pas ici cette prétention, mais voulons plutôt nous attacher à la preuve que nous fabriquons effectivement des arbres équilibrés. Nous ne donnerons pas de preuve établissant directement la correction des fonctions `heap_of_list` et `baltree_of_list`. Nous définirons plutôt une fonction de construction d'arbre binaire abstrait, *ie* une structure d'arbre binaire dont on ignorera les étiquettes. Nous présentons cette preuve comme un bel exemple de cas où l'invention de l'invariance de la fonction itérée demande un peu d'imagination (*cf infra :* 12.3).

12.1 La donnée et le programme

Le type d'arbre binaire *"abstrait"* sur lequel nous travaillons est le suivant :

```
Inductive btree : Set :=
 Lf : btree
|Br : btree -> btree -> btree.
```

La fonction d'insertion d'un élément dans un arbre devient simplement une fonction replaçant un feuille de l'arbre par l'arbre *"élémentaire"* `(BR Lf Lf)` :

```
Recursive Definition grow1 : btree -> btree :=
 Lf => (Br Lf Lf)
|(Br t1 t2)
 => (Br t2 (grow1 t1)).
```

Cette fonction est itérée (`n` fois) pour obtenir un arbre contenant `n` nœuds :

```
Recursive Definition grow : nat -> btree :=
 O => Lf
|(S n) => (grow1 (grow n)).
```

12.2 Qu'est-ce qu'un arbre équilibré ?

Un arbre équilibré est un arbre dont toute les feuilles sont situées sur deux niveaux au maximum. Plus précisément, la différence entre la *profondeur* de deux feuilles quelconques est au plus 1.
Plutôt que d'utiliser la notion de profondeur des feuilles, nous considèrerons la notion analogue de *hauteur d'une branche*. De plus, nous n'avons pas besoin de considérer l'ensemble des branches de l'arbre, mais uniquement celles de hauteur maximale et minimale. Pour pousser plus avant l'abstraction, nous considèrerons *la* hauteur maximale ainsi que *la* hauteur minimale ce qui nous évitera, dans notre définition d'arbre équilibré d'avoir à quantifier sur l'ensemble des branches. Notre définition d'arbre équilibré devient donc : *soient t un arbre binaire, H sa hauteur maximale et h sa hauteur minimale, t est équilibré ssi $H = h$ ou $H = h + 1$.*

Supposons connues les fonctions `max` et `min` sur les entiers, notre définition des arbres équilibrés se formalise de la façon suivante.

Fonction de calcul de la hauteur maximale d'un arbre.

```
Recursive Definition hmax : btree -> nat :=
 Lf => O
|(Br t1 t2)
 => (S (max (hmax t1) (hmax t2))).
```

Fonction de calcul de la hauteur minimale d'un arbre.

```
Recursive Definition hmin : btree -> nat :=
 Lf => O
|(Br t1 t2)
 => (S (min (hmin t1) (hmin t2))).
```

Définition de la propriété d'équilibrage.

```
Definition wb_btree :=
 [t:btree](hmax t)=(hmin t) \/ (hmax t)=(S(hmin t)).
```

12.3 L'invariant

Tentons une preuve *"directe"* de la correction de la fonction `grow` et voyons pourquoi une méthode simple de preuve par invariant n'est pas possible ici.
Soit donc à prouver : `Theorem wb_grow : (n:nat)(wb_btree (grow n))`

Raisonnons par induction sur `n`.
Le cas où `n=O` est trivial.
Dans le cas où `n=(S n0)`, nous avons comme hypothèse de récurrence

```
(wb_btree (grow n0))
```

et il nous faut montrer que

```
(wb_btree (grow (S n0)))
```

c'est-à-dire que

```
(wb_btree (grow1 (grow n0)))
```

La méthode de l'invariant simple nous dit que pour avoir notre résultat, il faut montrer que la propriété d'être équilibré est conservée par la fonction itérée `grow1`. Ce qui s'écrirait :

```
Lemma wb_grow1 : (t:btree)(wb_btree t) -> (wb_btree (grow1 t))
```

Or cet énoncé n'est pas vrai. En effet, l'arbre `(Br (Br Lf Lf) Lf)` est équilibré, mais si on lui applique `grow1` nous obtenons un arbre dont la hauteur maximale est 3 alors que sa hauteur minimale est 1. Il nous faut donc avoir recours à une invariance plus forte que la simple propriété d'être équilibré.

La fonction d'ajout d'un nœud dans un arbre ne conserve donc pas la propriété d'équilibrage pour tous les arbres équilibrés, mais uniquement pour un *sous-ensemble* des arbres équilibrés. De plus, il faut que ce sous-ensemble soit *clos* par la fonction `grow1`. Cet ensemble, nous l'appelons ensemble des *arbres ayant bien poussé* (`well_grown`). Pour le définir, et là est toute l'astuce, puisque nous voulons un ensemble clos par la fonction `grow1`, nous utiliserons cette fonction dans notre définition :

```
Inductive well_grown : btree -> Prop :=
 well_grown_Lf : (well_grown Lf)
|well_grown_Br1 : (t:btree)
 (well_grown t) -> (well_grown (Br t t))
|well_grown_Br2 : (t:btree)
 (well_grown t) -> (well_grown (Br t (grow1 t))).
```

REMARQUE : il nous a fallu longtemps tâtonner avant d'aboutir à une définition aussi compacte. Nous avons obtenue cette formulation en remarquant la périodicité dans la suite engendrée par `grow1` (à partir de l'arbre vide) d'arbres symétriques (clause `well_grown_Br1`) puis en observant que la dissymétrie ne venait que de l'ajout d'un nœud au sous-arbre droit (clause `well_grown_Br2`).

12.4 Correction

Le schéma général de la preuve est à peu près le suivant :

- *"tous les arbres calculés par la fonction* `grow` *sont des arbres ayant bien poussé"*;
- *"or tous les arbres ayant bien poussé sont des arbres équilibrés"*;
- *"donc les arbres calculés par la fonction* `grow` *sont des arbres équilibrés"*.

Dans les faits, la seconde prémisse de notre syllogisme sera moins générale. Elle affirme plus précisément que si l'on applique `grow1` à un arbre ayant bien poussé, on obtient un arbre équilibré. Ce qui nous suffira pour conclure.

La première prémisse se démontre extrèment facilement; la définition est faite pour cela. En revanche, la démonstration de la seconde prémisse (dans sa formulation plus restrictive) est beaucoups plus longue et nécessite, en particulier, une étude précise du comportement de la fonction `grow1` vis-à-vis des hauteurs maximale et minimale des arbres qu'elle traite.

On démontre par la méthode de l'invariant simple que tous les arbres obtenus par `grow` ont bien poussé:

```
Lemma well_grown_grow1 : (t:btree)
 (well_grown t) -> (well_grown (grow1 t)).

Lemma well_grown_grow : (n:nat)(well_grown (grow n)).
```

Nous étudions ensuite le comportement de `grow1` en montrant, dans un premier temps que la fonction fait croître la hauteur maximale de son argument de 1 ou la laisse inchangée.

```
Lemma hmax_grow1 : (t:btree)
 (hmax (grow1 t))=(hmax t)
 \/ (hmax (grow1 t)) = (S (hmax t)).
```

Nous montrons ensuite qu'il en va de même pour la hauteur minimale.

```
Lemma hmin_grow1 : (t:btree)
 (hmin (grow1 t))=(hmin t)
 \/ (hmin (grow1 t)) = (S (hmin t)).
```

Il nous faut maintenant examiner deux cas particuliers du comportement de `grow1` lorsque son argument est un arbre ayant bien poussé. Ces deux lemmes nous serviront essentiellement à éliminer des cas impossibles dans la preuve de la seconde prémisse du syllogisme.

Premièrement : si `grow1` a modifié la hauteur maximale de son argument, c'est que celui-ci était parfaitement équilibré.

```
Lemma hmax_grow1_Shmax : (t:btree)
 (well_grown t)
 -> (hmax (grow1 t))=(S(hmax t))
 -> (hmax t)=(hmin t).
```

Deuxièmement : si `grow1` a modifié la hauteur minimale de son argument alors l'arbre obtenu est parfaitement équilibré.

```
Lemma hmin_grow1_Shmin : (t:btree)
 (well_grown t)
 -> (hmin (grow1 t))=(S(hmin t))
 -> (hmax (grow1 t))=(hmin (grow1 t)).
```

Avant de conclure, nous établissons deux petits lemmes auxiliaires. Le premier nous dit que la hauteur minimale d'un arbre est inférieure ou égale à sa hauteur maximale.

```
Lemma le_hmin_hmax : (t:btree)
 (le (hmin t) (hmax t)).
```

Le second nous dit qu'un arbre symétrique est équilibré lorsque ses sous-arbres sont équilibrés.

```
Lemma wb_Br : (t:btree)
 (wb_btree t) -> (wb_btree (Br t t)).
```

Nous pouvons établir, en utilisant ce qui précède, la validité de notre seconde prémisse.

```
Lemma well_grown_wb_grow1 : (t:btree)
 (well_grown t) -> (wb_btree (grow1 t)).
```

Nous en tirons la conclusion qui s'impose.

```
Theorem wb_grow : (n:nat)(wb_btree (grow n)).
```

Conclusion

La valeur du travail présenté dans ce document ne doit pas se mesurer en terme d'originalité, mais bien plutôt, en terme d'effectivité. En effet, les algorithmes étudiés sont connus depuis belle lurette et leur correction n'est plus à mettre en doute : les ouvrages d'algorithmique où ces algorithmes sont présentés et étudiés abondent : voir, par exemple [2]. Paradoxalement, l'intérêt du travail présenté dans ce document réside dans tout ce qu'il ne contient pas : les quelques 2000 lignes de "code" contenant les "sources" des preuves vérifiées par le système Coq.

Un autre intérêt de ce travail est que les fonctions présentées sont de vrais programmes. En effet, le système Coq offre la possibilité d'extraire du code CAML à partir de preuves. En particulier, les λ-termes engendrés par la clause `Recursive Definition` peuvent être "compilés" vers CAML. Nous avons comparé les performances du code obtenu à partir de notre définition du tri par fusion et celles de la fonction de tri du module `sort` de CAML qui implémente également un tri par fusion. Cette dernière et plus rapide que notre fonction, mais seulement dans un rapport d'environ 4 à 5.

Enfin, nous pensons que ce document peut être une bonne introduction pédagogique au développement certifié de programmes fonctionnels.

Références

[1] R.S. Boyer and J S. Moore. *A Computational Logic.* ACM Monograph. Academic Press, Inc., 1979.

[2] M-C. Gaudel C. Froidevaux and M. Soria. *Types de données et algorithmes.* McGraw-Hil Paris, 1990.

[3] T. Coquand. *Une théorie des constructions.* PhD thesis, Université Paris VII, 1985.

[4] E.W. Dijkstra and W.H.J. Feijen. *A Method of Programming.* Addison-Wesley, 1988.

[5] C. Cornes *et al.* The coq proof assistant reference manual, version 5.10. Technical report, INRIA, 1996.

[6] C.A.R. Hoare. An axiomatic basis for computer programming. In *Communications of ACM*, 1969.

[7] W.A. Howard. The formulæ-as-types notion of construction. In *To H.B. Curry: Essays on combinatory logic, λ-calculus and formalism.* Academic Press, 1980.

[8] P. Manoury and M. Simonot. Automatizing termination proof of recursively defined functions. *TCS, 135*, 1995.

[9] R. Milner. A proposal for standard ml. Technical Report CSR-157-83, University of Edimburgh, 1983.

[10] C. Paulin-Mohring. Inductive definitions in the system coq - rules and properties. In *Proceedings of the conference Typed Lambda Calculi and Applications, LNCS 664*, 1993.

[11] D. Prawitz. Ideas and results in proof theory. In *Proceedings of the second Scandinavian logic symposium, North-Holland*, 1971.