Vérification formelle du tri fonctionnel par tas - Etude opérationnelle

Pascal Manoury

Vérification formelle du tri fonctionnel par tas Étude opérationnelle

Pascal Manoury

PPS – Université Paris Diderot
UFR d'informatique – Université Pierre et Marie Curie

La programmation fonctionnelle a su produire un certain nombre de «perles» : expressions élégantes ou astucieuses d'algorithmes performants en termes de purs calculs de valeurs. Parmi celles-ci, nous nous sommes penchés sur la formulation fonctionnelle, bien connue, de l'algorithme de *tri par tas* des éléments d'une liste (voir, par exemple [Bir96]).

L'algorithme fonctionnel de tri par tas repose sur le schéma général en deux étapes des tris par arbres :

1. construire une structure de tas à partir des éléments de la liste à trier.
2. extraire la liste triée de la structure de tas.

L'étape 1 est réalisée par itération d'une *fonction d'insertion* d'un nouvel élément dans la structure arborescente.

L'étude présentée ici ne concerne pas la correction *dénotationnelle* (le résultat est bien une liste triée) mais plutôt la correction *opérationnelle* de l'implantation qui repose sur la construction d'une structure d'*arbre équilibré*. À cet égard, l'élément nodal de l'expression fonctionnelle du tri par tas est la fonction d'insertion. C'est sur celle-ci que se concentre le travail présenté ici. Toute fonction construisant une structure d'arbre dont la racine minimise (ou maximise) les valeurs contenues dans l'arbre conviendra pour remplir la première étape du processus de tri. En revanche, seule une fonction construisant effectivement une structure d'arbre équilibré donnera la *performance algorithmique* attendue d'un tri pas tas. C'est en cela que nous distinguons la correction opérationnelle qui concerne les propriétés algorithmiques du processus d'obtention d'une valeur de la correction dénotationnelle qui ne concerne que les propriétés de la valeur obtenue.

Nous avions abordé cette question dans [Man96] au détour d'une étude plus générale de l'expression fonctionnelle d'algorithmes de tri dans le système Coq (à l'époque [Coq96]) dont la dernière partie était consacrée à la preuve de correction d'arbres binaires équilibrés. Cependant, pour ce qui est de la propriété d'équilibrage, nous avions travaillé sur structure d'arbre simplifiée – dans arbres sans étiquettes – ce qui avait simplifié la conduite de la preuve. Nous proposons aujourd'hui l'étude de la version complète de la fonction d'insertion. Nous n'avons pas connaissance d'autre étude axée sur une telle correction opérationnelle.

Cette étude s'inscrit dans le domaine de la *preuve de programme*. Nous en présentons le résultat selon deux formes :

– formulation et une preuve «à la main» du résultat de correction recherché ;
– formulation et preuve «à la machine» de ce même résultat en utilisant l'outil de preuve formelle PAF ! (voir [Bar03a]).

L'intérêt de cette double présentation est, à notre sens, de mesurer la distance entre la *rédaction* d'une preuve destinée à assurer un lecteur intuitif et cultivé de la validité d'un algorithme et le *codage* enchaînant et combinant un ensemble strictement défini de traits de langage de spécification et de commandes de construction de preuves formelles destinés à une vérification mécanique.

En effet, de même qu'il est par trop pénible de suivre le code d'un programme non commenté, *on ne lit pas un script de preuve formelle*. En revanche, si l'on dispose du guide d'une preuve «à la main» et si la distance entre rédaction et codage n'est pas infranchissable, le destinataire d'une preuve de programme, sans avoir besoin de déchiffrer ou rejouer l'ensemble du script de la preuve «à la machine», pourra se convaincre de la validité du codage des grandes étapes de celle-ci : l'énoncé des propriétés et des lemmes afférants. Le déchiffrage et la validation des étapes détaillées de la preuve n'offre plus d'intérêt : la machine s'en est chargé. On peut rapprocher cette méthode d'exposition de la démarche de spécification proposée par N. Lopez dans [Lop02] :

– spécification *pré-formelle* (notre preuve «à la main») ;
– spécification formelle (notre preuve «à la machine»).

dont l'objet est d'obtenir l'accord d'un client (*a priori* non spécialiste de tel ou tel système formel) sur la pertinence de l'énoncé d'une spécification formelle. Le système se chargera de la vérification mécanique de la correction du programme dérivé.

L'exposé de la preuve «à la machine» introduira au fur et à mesure des besoins les traits pertinents du système PAF !.

1. Éléments du problème

Arbre équilibré La performance algorithmique du tri par tas est obtenue si la structure d'arbre construite est *équilibrée*. Un arbre est *parfaitement équilibré* lorsque toutes ses feuilles sont à égale distance de la racine. Un arbre binaire parfaitement équilibré posséde un nombre de nœuds fonction de sa profondeur. Les listes à trier possèdent un nombre quelconque d'éléments, la propriété d'équilibrage requise pour l'algorithme de tri par tas est donc moins restrictive : on demandera simplement que pour toute paire de feuilles, leurs distances à la racine diffèrent au maximum de 1.

On peut donner une définition équivalente de cette propriété d'équilibrage évitant l'usage explicite de la quantification universelle (*«pour toute paire de feuilles...»*). Appelons *hauteur* la distance d'une feuille à la racine. Un arbre possède une *hauteur minimale* et une *hauteur maximale*. On dit alors qu'un arbre est équilibré lorsque sa hauteur minimale et sa hauteur maximale diffèrent au plus de 1.

Soit b un arbre binaire. Notons $hmin(b)$ sa hauteur minimale et $hmax(b)$ sa hauteur maximale. La propriété d'équilibrage voulue se formalise ainsi :

$$hmax(b) = hmin(b) \vee hmax(b) = hmin(b) + 1$$

Ce que l'on notera de façon plus conscise :

$$hmin(b) \pm hmax(b)$$

Pour un ensemble A d'étiquettes, on note $B[A]$ l'ensemble des arbres binaires étiquetés par les éléments de A. L'ensemble $WB[A]$ des arbres binaires équilibrés est alors défini par

$$WB[A] = \{b \in B[A] \mid hmin(b) \pm hmax(b)\}$$

On a que $b \in WB[A]$ est équivalent à $b \in B[A]$ et $hmin(b) \pm hmax(b)$.

Construction du tas : invariant La construction du tas à partir des éléments de la liste à trier est obtenue par itération, sur les éléments de la liste, d'une fonction d'insertion d'un nouvel élément a dans un tas b. Pour assurer la performance algorithmique il faut s'assurer que cette fonction préserve la propriété d'équilibrage de l'arbre binaire qu'elle produit. Appelons ins la fonction d'insertion. Un moyen immédiat d'atteindre notre but serait d'établir la simple propriété d'invariance

$$b \in WB[A] \Rightarrow ins(a, b) \in WB[A]$$

Mais nous verrons que cette invariance ne tient pas et qu'il faut en déterminer une plus fine.

Nous allons donc nous attacher, dans la suite de cet article, à poser une propriété invariance plus précise de la fonction d'insertion et nous verrons comment elle permet d'établir la correction opérationnelle recherchée. Nous préciserons pour cela comment caractériser le sous ensemble particulier des arbres engendrés par l'itération de la fonction d'insertion (les *arbres adéquats*). Cette caractérisation repose sur la notion de similarité de structure entre arbres (*arbres isomorphes*) ainsi que, et là est l'astuce, sur l'usage de la fonction d'insertion elle-même.

2. Rédaction «à la main»

Soit A un ensemble d'étiquettes. Soient Lf et Br deux symboles, respectivement d'arité 0 et 3. L'ensemble des arbres binaires $B[A]$ est donné par l'algèbre de termes :

- $Lf \in B[A]$;
- si $a \in A$, si b_1, b_2 $B[A]$ alors $Br(a, b_1, b_2) \in B[A]$.

Soit c une fonction binaire de comparaison des éléments de A. La fonction d'insertion ins est définie par les équations récursives conditionnelles suivantes :

$$\begin{array}{lcll} ins(a, Lf) & = & Br(a, Lf, Lf) & \\ ins(a_1, Br(a_2, b_1, b_2)) & = & Br(a_1, b_2, ins(a_2, b_1)) & \text{si } c(a_1, a_2) \\ & = & Br(a_2, b_2, ins(a_1, b_1)) & \text{sinon} \end{array}$$

2.1. Affiner l'invariant

On n'arrivera pas à démontrer directement que $\forall b \in B[A].b \in WB[A] \Rightarrow ins(b) \in WB[A]$. C'est à dire, pour tout $b \in B[A]$ $hmin(b) \pm hmax(b) \Rightarrow hmin(ins(b)) \pm hmax(ins(b))$, car cet énoncé est tout simplement faux. En effet l'arbre binaire $b = Br(a, Br(a, Lf, Lf), Lf)$ vérifie bien $hmin(b) \pm hmax(b)$ mais $ins(a, b) = Br(a, Lf, Br(a, Lf, Br(a, Lf, Lf)))$ et $hmax(ins(a, b)) = hmin(ins(a, b)) + 2$.

En fait, l'itération de la fonction ins, en prenant pour premier terme l'arbre vide Lf, détermine une suite particulière d'arbres équilibrés. Si l'on fait abstraction des étiquettes, on a le schéma d'insertion suivant :

$$\begin{array}{lcl} ins_a(Lf) & = & Br(Lf, Lf) \\ ins_a(Br(b_1, b_2)) & = & Br(b_2, ins_a(b_1)) \end{array}$$

L'itérée $ins_a^n(Lf)$ engendre la suite (voir figure ci-après) :

$$\begin{array}{lcl} b_0 & = & Lf \\ b_1 & = & Br(Lf, Lf) \\ b_2 & = & Br(Lf, Br(Lf, Lf)) \\ b_3 & = & Br(Br(Lf, Lf), Br(Lf, Lf)) \\ b_4 & = & Br(Br(Lf, Lf), Br(Lf, Br(Lf, Lf))) \\ b_5 & = & Br(Br(Lf, Br(Lf, Lf)), Br(Lf, Br(Lf, Lf))) \\ b_6 & = & Br(Br(Lf, Br(Lf, Lf)), Br(Br(Lf, Lf), Br(Lf, Lf))) \\ etc... & & \end{array}$$

En mettant de côté le premier terme b_0, on remarque que les arbres obtenus ont deux formes :

- soit $Br(b_i, b_i)$
- soit $Br(b_i, ins_a(b_i))$

pour un b_i quelconque de la suite. De cette remarque, on peut tirer une définition inductive de l'ensemble WBI des arbres engendrés par la fonction ins, partant de l'arbre vide :

- $Lf \in WBI$;

FIG. 1 – Étapes b_1 à b_5

– si $b \in WBI$ alors $Br(b, b) \in WBI$;
– si $b \in WBI$ alors $Br(b, ins_a(b)) \in WBI$.

Arbres isomorphes Nous ne voulons pas ici, comme nous l'avions fait dans [Man96], raisonner sur une telle abstraction de la fonction d'insertion. Il nous faut donc expliciter comment «faire abstraction des étiquettes» en introduisant la notion d'*arbres isomorphes* (arbres de même forme).

Notons $b_1 \approx b_2$ le fait, pour deux arbres binaires, d'être isomorphes. C'est une relation d'équivalence que l'on définit par :
– $Lf \approx Lf$;
– si $b_1 \approx b_2$ et $b_3 \approx b_4$ alors, pour tout $a_1, a_2 \in A$, $Br(a_1, b_1, b_3) \approx Br(a_2, b_2, b_4)$.

Arbres adéquats On définit alors l'ensemble $WBI[A]$ d'arbres binaires à étiquettes dans A – dont nous verrons qu'ils sont équilibrés et qu'ils sont en adéquation avec la fonction d'insertion – de la façon suivante :
– $Lf \in WBI[A]$;
– si $b_1 \in WBI[A]$, et si $(b_2 \approx b_1 \vee \forall a \in A.b_2 \approx ins(a, b_1))$ alors $Br(a, b_1, b_2) \in WBI[A]$.
Nous appelons *arbres adéquats* les éléments de $WBI[A]$.

Résultats La validation de la correction opérationnelle d'un programme fonctionnel de tri par tas repose sur les deux résultats suivants :

Théorème 1 *l'ensemble $WBI[A]$ est clos par la fonction ins, c'est-à-dire*

$$\forall b \in B[A].(b \in WBI[A] \Rightarrow \forall a \in A.ins(a, b) \in WBI[A])$$

Théorème 2 *tout élément de $WBI[A]$ est équilibré, c'est-à-dire $WBI[A] \subset WB[A]$, ou encore*

$$\forall b \in B[A].(b \in WBI[A] \Rightarrow b \in WB[A])$$

En effet, la fonction de construction du tas $HeapOfList$ par itération de la fonction d'insertion sur les éléments d'une liste est définie par récurrence sur la liste :

$$\begin{array}{lcl} HeapOfList(Nil) & = & Lf \\ HeapOfList(Cons(x, xs)) & = & ins(x, HeapOfList(xs)) \end{array}$$

La correction opérationnelle finale de la fonction de construction du tas est

$$\forall xs \in L[A].HeapOfList(xs) \in WB[A]$$

où $L[A]$ est l'ensemble des listes d'éléments de A. Comme on a que $\forall xs \in L[A].HeapOfList(xs) \in B[A]$ il suffit, en vertu du théorème 2, de montrer que

$$\forall xs \in L[A].HeapOfList(xs) \in WBI[A]$$

Ce que l'on obtient par induction sur la liste xs :

- si $xs = Nil$, on a bien $Lf \in WBI[A]$
- si $xs = Cons(x, xs)$, on a, par hypothèse d'induction $HeapOfList(xs) \in WBI[A]$. Il nous faut $HeapOfList(Cons(x, xs)) \in WBI[A]$, c'est-à-dire $ins(x, HeapOfList(xs)) \in WBI[A]$. Ce que l'on a en combinant le théorème 1 à l'hypothèse d'induction.

2.2. Lemmes

Dans la preuve de nos deux théorèmes nous ferons usage explicite de certaines propriétés concernant les rapports entre calcul de hauteur, fonction d'insertion, arbres isomorphes et arbres adéquats qu'énoncent les quatre lemmes que voici.

Lemme 1 *La fonction d'insertion fait croître la hauteur maximale d'au plus 1.*

$$\forall a \in A. \forall b \in B[A]. hmax(b) \pm hmax(ins(a, b))$$

Par induction structurelle sur b, on montre $\forall a \in A. hmax(b) \pm hmax(ins(a, b))$.

Si $b = Lf$, on a bien 0 ± 1.

Si $b = Br(a_1, b_1, b_2)$, on a, par hypothèse d'induction que $hmax(b_1) \pm hmax(ins(a, b_1))$, pour tout $a \in A$. Soit $a_0 \in A$ quelconque. Par définition de ins et de $hmax$[1], il faut montrer que

$$max(hmax(b_1), hmax(b_2)) \pm max(hmax(b_2), hmax(ins(\alpha, b_1))) \text{ avec } \alpha \in \{a_0, a_1\}$$

Suivant notre hypothèse d'induction, on peut donc raisonner par cas selon que $hmax(ins(\alpha, b_1)) = hmax(b_1)$ ou $hmax(ins(\alpha, b_1)) = hmax(b1) + 1$.

Si $hmax(ins(\alpha, b_1)) = hmax(b_1)$, il faut avoir

$$max(hmax(ins(\alpha, b_1)), hmax(b_2)) \pm max(hmax(b_2), hmax(ins(\alpha, b_1)))$$

ce qui est à l'évidence vrai.

Si $hmax(ins(\alpha, b_1)) = hmax(b_1) + 1$, il faut avoir

$$max(hmax(b_1), hmax(b_2)) \pm max(hmax(b_2), hmax(b_1) + 1)$$

On raisonne selon que $hmax(b_1) + 1 \leq hmax(b_2)$ ou non.

Si $hmax(b_1) + 1 \leq hmax(b_2)$, on a qu'alors $hmax(b_1) \leq hmax(b_2)$. Il faut donc $hmax(b_2) \pm hmax(b_2)$, ce qui est à l'évidence vrai.

Si $\neg(hmax(b_1) + 1 \leq hmax(b_2))$, un peu d'arithmétique nous donne que $hmax(b_2) \leq hmax(b_1)$. Il faut donc $hmax(b_1) \pm hmax(b_1) + 1$, ce qui est à l'évidence vrai et achève la démonstration.

Lemme 2 *Les arbres isomorphes ont bien même hauteur minimale et même hauteur maximale*

$$\forall b_1, b_2 \in B[A]. (b_1 \approx b_2 \Rightarrow hmin(b_1) = hmin(b_2))$$
$$\forall b_1, b_2 \in B[A]. (b_1 \approx b_2 \Rightarrow hmax(b_1) = hmax(b_2))$$

On obtient facilement ces deux énoncés par induction sur b_1, puis par cas sur b_2, dans le cas inductif.

Lemme 3 *La relation $\approx$ est invariante pour la fonction d'insertion*

$$\forall b_1, b_2 \in B[A]. (b_1 \approx b_2 \Rightarrow \forall a_1, a_2 \in A. (ins(a_1, b_1) \approx ins(a_2, b_2)))$$

[1] $hmax(Br(a, b_1, b_2) = 1 + max(hmax(b_1), hmax(b_2))$

Ici également, le résultat s'obtient directement par induction sur b_1, par cas sur b_2, puis analyse des cas introduits par la définition de *ins*.

Lemme 4 *L'ensemble* $WBI[A]$ *est clos par* $\approx$

$$\forall b_1, b_2 \in B[A].(b_1 \approx b_2, b_1 \in WBI[A] \Rightarrow b_2 \in WBI[A])$$

Ici encore, on raisonne par induction sur b_1, puis par cas sur b_2. Regardons un peu les étapes de cette preuve. On applique l'induction sur la formule $\forall b_2 \in B[A].(b_1 \approx b_2, b_1 \in WBI[A] \Rightarrow b_2 \in WBI[A])$.

Si $b_1 = Lf$ et si $b_2 = Lf$, le résultat est trivial.

Si $b_1 = Lf$ et $b_2 = Br(a, b_3, b_4)$, le résultat est trivialement vrai par fausseté de l'hypothèse $Lf \approx Br(a, b_3, b_4)$.

Si $b_1 = Br(a, b_3, b_4)$ et $b_2 = Lf$, le résultat est ici aussi trivialement vrai par fausseté de l'hypothèse $Br(a, b_3, b_4) \approx Lf$.

Si $b_1 = Br(a_1, b_3, b_4)$ et $b_2 = Br(a_2, b_5, b_6)$. On a, par hypothèse $b_3 \approx b_5$ et $b_4 \approx b_6$ ainsi que $b_3 \in WBI[A]$ et $b_4 \approx b_3 \vee \forall a \in A.b_4 \approx ins(a, b_3)$. On a, par hypothèse d'induction que $\forall b_2 \in B[A].(b_3 \approx b_2 \Rightarrow b_2 \in WBI[A])$. On veut (i) $b_5 \in WBI[A]$ et $(ii) b_6 \approx b_5$ ou $b_6 \approx ins(a, b_5)$, pour tout $a \in A$.

(i) est une conséquence de l'hypothèse d'induction.

(ii) s'obtient en raisonnant par cas d'après l'hypothèse $b_4 \approx b_3$ ou $b_4 \approx ins(a, b_3)$, pour tout $a \in A$.

Si $b_4 \approx b_3$, on a $b_6 \approx b_4 \approx b_3 \approx b_5$ en utilisant la transitivité et la symétrie de $\approx$.

Si $b_4 \approx ins(a, b_3)$, on a par lemme 3 que $ins(a, b_3) \approx ins(a, b_5)$; on a donc $b_6 \approx b_4 \approx ins(a, b_3) \approx ins(a, b_5)$.

Lemme technique : *si la fonction d'insertion fait croître strictement la hauteur maximale d'un arbre de* $WBI[A]$*, c'est que l'arbre était parfaitement équilibré.* Ce fait est la clé de la correction algorithmique de la fonction d'insertion. Il manifeste comment les étiquettes sont disposées de façon à «remplir» la structure de tas avant d'en accroître la hauteur. Il est donc intéressant d'en voir le détail.

Lemme 5

$$\forall a \in A. \forall b \in WBI[A].(hmax(ins(a, b)) = hmax(b) + 1 \Rightarrow hmin(b) = hmax(b))$$

Par induction sur b.

Le cas de base est trivial.

Si $b = Br(a', b_1, b_2)$, il faut montrer $min(hmin(b_1), hmin(b_2)) = max(hmax(b_1), hmax(b_2))$. On a, par hypothèse que $Br(a, b_1, b_2) \in WBI[A]$, ce qui nous donne en particulier que $b_1 \in WBI[A]$ et $b_2 \approx b_1$ ou $b_2 \approx ins(\alpha, b_1)$ avec $\alpha \in \{a, a'\}$. On raisonne par cas sur cette disjonction.

Si $b_2 \approx b_1$, le lemme 2 nous donne que $hmin(b_1) = hmin(b_2)$ et $hmax(b_1) = hmax(b_2)$. On peut donc se ramener à montrer que $hmin(b_1) = hmax(b_1)$. Ce que l'on aura par hypothèse d'induction si l'on montre que (i) $b_1 \in WBI[A]$ et (ii) $hmax(ins(a, b_1)) = hmax(b_1) + 1$.

(i) nous est donné par hypothèse.

(ii) par hypothèse, on a que $hmax(ins(a, Br(a', b_1, b_2))) = hmax(Br(a, b_1, b_2)) + 1$, c'est-à-dire $max(hmax(b_2), hmax(ins(\alpha, b_1))) = max(hmax(b_1), hmax(b_2)) + 1$. En utilisant le fait que l'on a ici $hmax(b_1) = hmax(b_2)$, on a en fait que $max(hmax(b_1), hmax(ins(\alpha, b_1))) = hmax(b_1) + 1$. Or, il est clair que $hmax(b_1) \leq hmax(ins(\alpha, b_1))$. On a donc $hmax(ins(\alpha, b_1)) = hmax(b_1) + 1$ avec $\alpha \in \{a, a'\}$.

Si $b_2 \approx ins(\alpha, b_1)$. On a par hypothèse que $hmax(ins(a, Br(a', b_1, b_2))) = hmax(Br(a, b_1, b_2)) + 1$, c'est-à-dire $max(hmax(b_2), hmax(ins(\alpha, b_1))) = max(hmax(b_1), hmax(b_2)) + 1$. Le lemme 2 nous donne que $hmax(b_2) = hmax(ins(\alpha, b_1))$ et l'on sait que $hmax(b_1) \leq hmax(ins(\alpha, b_1))$. On tire donc de notre hypothèse l'absurdité $hmax(ins(\alpha, b_1)) = hmax(ins(\alpha, b_1)) + 1$. Ce qui résoud trivialement l'examen de ce second cas et achève la démonstration.

2.3. $WBI[A]$ est clos par ins

Soit à montrer (théorème 1)

$$\forall b \in B[A].(b \in WBI[A] \Rightarrow \forall a \in A.ins(a,b) \in WBI[A])$$

Par induction structurelle sur b :

Si $b = Lf$, on veut, pour tout $a \in A$, $Br(a, Lf, Lf) \in WBI[A]$; c'est-à-dire, (i) $Lf \in WBI[A]$ et (ii) $Lf \approx Lf \vee Lf \approx ins(a, Lf)$. On a (i) par définition de $WBI[A]$ et (ii) car $Lf \approx Lf$ par définition de $\approx$.

Si $b = Br(a_0, b_1, b_2)$, on suppose $Br(a_0, b_1, b_2) \in WBI[A]$, c'est-à-dire

$$(H1)\ b_1 \in WBI[A] \text{ et } (H2)\ (b_2 \approx b_1 \vee \forall a \in A.b_2 \approx ins(a, b_1))$$

On a, par hypothèse d'induction (et $(H1)$) que $ins(a, b_1) \in WBI[A])$, pour tout $a \in A$.

Il faut montrer que $ins(a, Br(a_0, b_1, b_2)) \in WBI[A]$, c'est-à-dire $Br(\alpha_1, b_2, ins(\alpha_2, b_1)) \in WBI[A]$ avec $(\alpha_1, \alpha_2) = (a, a_0)$, si $c(a, a_0)$ ou $(\alpha_1, \alpha_2) = (a_0, a)$, sinon. On veut donc, par définition de $WBI[A]$

$$(i)\ b_2 \in WBI[A] \text{ et } (ii)\ ins(\alpha_2, b_1) \approx b_2 \vee \forall a \in A.ins(\alpha_2, b_1) \approx ins(a, b_2)$$

Ce que l'on montre en raisonnant par cas selon $(H2)$.

Si $b_2 \approx b_1$, (i) nous est donné par le lemme 4 et $(H1)$; (ii) nous est donné par le lemme 3.

Si $b_2 \approx ins(a, b_1)$ pour tout $a \in A$, (i) nous est donné par le lemme 4 ; (ii) est donné comme cas particulier de notre dernière hypothèse.

2.4. $WBI[A] \subset WB[A]$

Soit à montrer (théorème 2)

$$\forall b \in B[A].(b \in WBI[A] \Rightarrow b \in WB[A])$$

Par induction sur b.

Si $b = Lf$, on a $Lf \in WB[A]$ car $hmin(Lf) = hmax(Lf) = 0$.

Si $b = Br(a_0, b_1, b_2)$, supposons $Br(a_0, b_1, b_2) \in WBI[A]$, c'est-à-dire

$$(H1)\ b_1 \in WBI[A] \text{ et } (H2)\ b_2 \approx b_1 \vee \forall a \in A.b_2 \approx ins(a, b_1)$$

On a alors par hypothèse d'induction que $b_1 \in WB[A]$.

Montrons $Br(a_0, b_1, b_2) \in WB[A]$, c'est-à-dire $hmin(Br(a_0, b_1, b_2)) \pm hmax(Br(a_0, b_1, b_2))$ et, plus précisément

$$min(hmin(b_1), hmin(b_2)) \pm max(hmax(b_1), hmax(b_2))$$

On raisonne par cas selon $(H2)$:

Si $b_2 \approx b_1$, on a, par le lemme 2, que $hmin(b_1) = hmin(b_2)$ et $hmax(b_1) = hmax(b_2)$. On veut donc $hmin(b_1) \pm hmax(b_1)$, ce qui revient à $b_1 \in WB[A]$ qui nous est donné par hypothèse d'induction.

Si $b_2 \approx ins(a, b_1)$, pour tout $a \in A$, on a que $hmin(b_2) = hmin(ins(a, b_1))$ et $hmax(b_2) = hmax(ins(a, b_1))$. On veut donc, pour tout $a \in A$

$$min(hmin(b_1), hmin(ins(a, b_1))) \pm max(hmax(b_1), hmax(ins(a, b_1)))$$

Or on sait que $hmin(b_1) \leq hmin(ins(a, b_1))$ et que $hmax(b_1) \leq hmax(ins(a, b_1))$, pour tout $a \in A$. On veut donc

$$hmin(b_1) \pm hmax(ins(a, b_1))$$

D'aprés l'hypothèse d'induction $b_1 \in WB[A]$, on peut raisonner par cas, selon que $hmax(b_1) = hmin(b_1)$ ou $hmax(b_1) = hmin(b_1) + 1$.

Si $hmax(b_1) = hmin(b_1)$, on veut en fait $hmax(b_1) \pm hmax(ins(a, b_1))$, ce que l'on a par le lemme 1.

Si $hmax(b_1) = hmin(b_1) + 1$, suivant le lemme 1, on a qu'ou bien $hmax(ins(a, b_1)) = hmax(b_1)$, ou bien $hmax(ins(a, b_1)) = hmab(b_1) + 1$. Mais, si $hmax(ins(a, b_1)) = hmab(b_1) + 1$, notre lemme technique (lemme 5) nous donne que $hmin(b_1) = hmax(b_1)$ ce qui contredit l'hypothèse $hmax(b_1) = hmin(b_1)+1$. On a donc que $hmax(ins(a, b_1)) = hmax(b_1)$ et il faut montrer $hmin(b_1) \pm hmin(b_1)+1$ qui est vrai par définition de $\pm$.

3. Formalisation «à la machine»

Tournons nous à présent vers la transcription formalisée de la preuve des théorèmes 1 et 2 telle que nous avons pu la réaliser dans notre système. Bien entendu, il a fallu pour cela adapter la lettre de la formulation adoptée pour la preuve «à la main» pour en garder l'esprit. Par exemple, notre système n'est pas basé sur une théorie des ensembles, alors que nous avions employé ce formalisme. Nous avons également eu recours à des définitions inductives de relation ($\approx$) ou d'ensemble ($WBI[A]$), ce que nous ne pouvons réaliser dans notre système. Mais nous verrons comment contourner cette difficulté en mélant définition de fonctions récursives et *promotion booléenne*.

Note : nous entrelarderons la présentation qui suit de quelques «notes» décrivant les principaux traits du système PAF!

3.1. Type et fonction

Note : le système PAF! est dédié à la preuve de programmes ML. Son langage de spécification reprend la syntaxe ML des définitions de type de données[2] *et de fonctions. Notre système intègre donc (du moins, en partie) le* langage de programmation *ML. Un langage de programmation est une syntaxe, mais aussi une sémantique. Notre système intègre celle de ML sous la forme de sa* sémantique naturelle *([Kan87]) qui permet la réduction ou évaluation des expressions ML. Nous nous référerons au mécanisme de réduction comme à celui d'une* évaluation symbolique.

Structure d'arbre binaire Le type ML des arbres binaires qui nous servira est donné par :

```
type 'a btree = Lf  | Br of 'a * ('a btree) * ('a btree)
```

Note : les types de données, et plus généralement les informations de type, sont utilisés par le système à deux niveaux :
- *au niveau* syntaxique *comme des sortes. Le langage des termes et des formules est un langage* multi-sorté *;*
- *au niveau* logique *comme des prédicats. L'appartenance de la dénotation d'un terme* `t` *au type de données est notée comme une* assignation *de type (par exemple* `(t :'a btree)`*) mais il faut la lire comme la satisfaction d'un prédicat (*`t` *satisfait le prédicat «être un* `'a btree`*»). On parlera dans ce cas d'assignation* forte.

Les constructeurs sont introduits comme symboles fonctionnels libres, avec leur assignation de type forte. Les principes d'induction structurelle et par cas de construction sont inférés (voir [Kri90] et [Par92] pour les attendus fondamentaux et [Bar03b] pour la notion de typage «fort»).

[2] Les types enregistrements ne sont pas encore implantés.

Fonction d'insertion La formulation de la fonction d'ajout d'un élément dans un tas que nous utiliserons est celle-ci :

```
let rec ins_heap c a h =
  match h with
      Lf -> Br(a, Lf, Lf)
    | Br(a0, h1, h2) ->
        if (c a a0) then
          Br(a, h2, (ins_heap c a0 h1))
        else
          Br(a0, h2, (ins_heap c a h1))
```

où `c` est un paramètre fonctionnel, de type `'a -> 'a -> bool`, sensé donner l'ordre utilisé sur les étiquettes.

> *Note : le système infère le type attendu des fonctions à la manière de ML. Ici, par exemple,* `ins_heap : ('a -> 'a -> bool) -> 'a -> 'a btree -> 'a btree`*. Mais le système ne vérifie pas de lui même la validité de l'assignation forte de type, c'est-à-dire la totalité ou terminaison de la fonction. Si l'on ne fait rien de plus, le système retiendra cette information comme simple contrainte* syntaxique de sorte *et interdira l'application de la fonction à autre chose qu'un terme de la sorte attendue.*
>
> *Pour que l'assignation de type prenne son sens logique fort,* il faut prouver *que la fonction est totale, c'est-à-dire que, dans notre exemple :*
>
> ```
> Forall 'a:type. Forall c:'a -> 'a -> bool. Forall a:'a. Forall h:'a btree.
>  ((ins_heap c a h):'a btree)
> ```
>
> *Les quantificateurs de cet énoncé appellent quelques commentaires.*
> - `Forall 'a :type` *n'a pas d'autre sens que celui d'une indication de sorte. Il ne faut pas y voir un sens logique réel. La variable liée peut être instanciée par n'importe quelle expression syntaxiquement identifiée comme expression de type.*
> - `Forall c :'a -> 'a -> bool` *est une quantification du second ordre. L'assignation de type doit être lue au sens fort :* `c` *ne pourra être instancié que par des fonctions réputées totales.*
> - `Forall a :'a` *et* `Forall h :'a btree` *sont des quantification du premier ordre. Et ici également, l'assignation de type doit être lue au sens fort :* `a` *et* `h` *ne pourront être instanciées que par des expressions (disons* `e1` *et* `e2`*) dont on saura exhiber une* preuve *que* `(e1 : 'a)` *et* `(e2 :'a btree)`*.*

La preuve «à la main», de caractère algébrique, a recours aux équations conditionnelles qui ont servi à définir la fonction d'insertion. Dans la preuve «à la machine», le raisonnement équationnel est remplacé par l'invocation du mécanisme d'évaluation symbolique qui implante les régles de réductions de la sémantique naturelle. L'alternative `if-then-else` du deuxième cas de la définition ML de `ins_heap` est le pendant des équations conditionnelles de la définition algébrique. Les équations de la fonction *ins* données en 2 sont vérifiées par l'évaluation symbolique de la définition ML de `ins_heap`.

> *Note : la tactique* `SymEval` *implante, dans notre système, l'utilisation de l'évaluation symbolique pour construire les preuves. Elle prend un premier argument qui est un symbole fonctionnel. Elle calcule une* forme normale de tête *de l'expression déterminée par ce symbole. La tactique prend, optionnellement, un second argument (un entier) qui indique l'occurence du symbole à considérer. Sa valeur par défaut est 1.*
>
> *Si le symbole fonctionnel est celui d'une fonction définie, la tactique procède à l'expansion de la définition avant d'appliquer la réduction.*

3.2. Arbres équilibrés

Notre définition d'arbre équilibré repose sur les notions de *hauteur maximale* et *hauteur minimale*. Ces valeurs sont calculées respectivement par les deux fonctions

```
let rec hmax h =
  match h with
     Lf -> 0
  |  Br(a, h1, h2) -> S(max (hmax h1) (hmax h2))

let rec hmin b =
  match h with
     Lf -> 0
  |  Br(a, b1, b2) -> S(min (hmin b1) (hmin b2))
```

où `S` est la fonction successeur.

Des booléens aux propositions Pour définir la relation $\pm$, nous commençons par poser la *fonction booléenne*

```
let pred_or_eq n m = ((n = m) || ((S n)=m))
```

dont nous démontrons qu'elle est totalement définie. Ce que nous écrivions $n \pm m$, pour deux entiers n et m devient `‘(pred_or_eq n m)` dans notre système.

> *Note : remarquez la présence de l'accent grave (ou back quote) devant l'application de la fonction.*
> *La constante de prédicat discrètement notée* `‘` *est un symbole prédéfini du système qui promeut toute valeur booléenne au rang de valeur de vérité : si* `t` *est un terme de sorte booléenne (type* `bool`*), on peut former la formule atomique* `‘t` *dont la sémantique est*
> – `‘t` *est vrai si* `t` *s'évalue à la valeur* `true`
> – `Not ‘t` *est vrai si* `t` *s'évalue à la valeur* `false`
> *Notons que pour tout* `t` *de type* `bool`*, on n'a pas nécessairement que* `‘(t)` *est vrai ou faux. Il faut, pour pouvoir assigner une valeur de vérité à* `‘(t)`*, que l'on ait démontré que* `t : bool` *(au sens fort).*

Notre formulation en terme de fonction booléenne a l'avantage sur une formulation utilisant la disjonction logique (`‘(n = m) Or ‘((S n) = m)`) de pouvoir utiliser l'évaluation symbolique pour résoudre un certain nombre de cas triviaux : typiquement les cas de base des récurrences. La calcul propositionnel est ainsi, certe de façon limitée, réellement un calcul.

Le lemme 1 : *«la fonction d'insertion fait croître la hauteur maximale d'au plus 1»*, s'énonce alors

```
Theorem pred_or_eq_hmax_ins_heap :
  Forall 'a:type. Forall c:'a -> 'a -> bool. Forall x:'a. Forall b:'a btree.
    ‘(pred_or_eq (hmax b) (hmax (ins_heap c x b)))
```

Lorsque que nous avons donné la preuve «à la main» du lemme 1, nous avons utilisé le racourci suivant (souligné) :

> *Par définition de ins et de hmax, il faut montrer que*
>
> $$max(hmax(b_1), hmax(b_2)) \pm max(hmax(b_2), hmax(ins(\alpha, b_1))) \text{ avec } \alpha \in \{a_0, a_1\}$$

Dans un système formel tel que le notre, un tel racourci n'en est en fait pas un. Ce qui est en fait sous-jacent à l'utilisation de cette tournure est l'indépendance du calcul de la hauteur maximale vis-à-vis de la valeur des étiquettes présentes dans l'arbre. Nous avons, dans la preuve formalisée, utilisé explicitement ce fait en démontrant l'équation :

```
Theorem hmax_ins_heap_x :
  Forall 'a:type. Forall c:'a -> 'a -> bool. Forall b:'a btree.
  Forall x1:'a. Forall x2:'a.
    `((hmax (ins_heap c x1 b)) = (hmax (ins_heap c x2 b)))
```

> *Note : notre langage logique n'a pas de symbole d'égalité. Il ne connaît que l'égalité polymorphe prédéfinie de ML. Il n'y a donc pas à proprement parler d'équation dans notre système, mais des énoncés utilisant le mécanisme de promotion des booléens appliqué à l'expression du calcul de l'égalité de deux valeurs. La validité de tels «énoncés équationnels» peut être obtenue soit par simple évaluation symbolique, soit par preuve, en général, par induction.*
>
> *Le système de preuve offre ainsi deux formes correspondant au raisonnement équationnel : l'évaluation symbolique (implanté par la tactique `SymEval`) et l'utilisation d'énoncés équationnels (tactique `Rewrite`).*

Le résultat d'indépendance ci-dessus, nous permet d'établir, concernant la valeur de la hauteur maximale du résultat d'une insertion, l'énoncé équationnel suivant :

```
Theorem hmax_ins_heap_eq :
  Forall 'a:type. Forall c:'a -> 'a -> bool. Forall x:'a.
  Forall b1:'a btree. Forall b2:'a btree. Forall y:'a.
    `((hmax (ins_heap c y (Br(x,b1,b2))))
     = (S(max (hmax b2) (hmax (ins_heap c x b1)))))
```

qui joue le rôle, dans la preuve «à la machine» du lemme 1, de l'astuce du α dans notre preuve «à la main» de ce lemme.

Ensemble, fonction caractéristique, prédicat Nous avions défini l'ensemble des arbres équilibrés $WB[A]$ par schéma de compréhension. Ce qui n'est pas réalisable dans notre système. Cependant, on peut dire que la fonction booléenne `pred_or_eq` combinée aux fonctions `hmax` et `hmin` donne la *fonction caractéristique* de l'ensemble $WB[A]$. Posons cette fonction :

```
let is_wb b = (pred_or_eq (hmin b) (hmax b))
```

Le prédicat $hmin(b) \pm hmax(b)$ utilisé dans la définition ensembliste de $WB[A]$ devient, selon notre usage, `(is_wb b)`. Il suffit à caractériser l'appartenance d'une valeur `b` de type `'a btree` à l'ensemble $WB[A]$ si le paramètre A est lu comme le type paramètre `'a` et l'ensemble $B[A]$ comme le type paramétré `'a btree`. La formule `(is_wb b)` correspond donc à l'énoncé d'appartenance $b \in WB[A]$.

3.3. Arbres isomorphes

La définition inductive de la relation d'équivalence $\approx$ se transpose directement en terme de fonction booléenne.

```
let rec biso b1 b2 =
  match b1, b2 with
    (Lf, Lf) -> true
```

```
  | ((Br(_, b11, b12)), (Br(_, b21, b22))) -> (biso b11 b21) && (biso b12 b22)
  | _ -> false
```

Selon notre usage, le prédicat correspondant à la relation $\approx$ s'obtient par promotion booléenne.

Notez que la quantification universelle sur les étiquettes qui était nécessaire dans la définition de $\approx$ est ici masquée sous la non utilisation des étiquettes dans la définition de la fonction `biso` (motif universel `_` de la deuxième clause). Nous verrons tout à l'heure que ce miracle ne se produit pas toujours.

Le double lemme 2 : *«les arbres isomorphes ont bien même hauteur minimale et même hauteur maximale»* devient dans notre formalisation les deux lemmes

```
Theorem biso_hmax :
  Forall 'a: type. Forall b1: 'a btree. Forall b2: 'a btree.
    `(biso b1 b2) -> `((hmax b1) = (hmax b2))

Theorem biso_hmin :
  Forall 'a: type. Forall b1: 'a btree. Forall b2: 'a btree.
    `(biso b1 b2) -> `((hmin b1) = (hmin b2))
```

Le lemme 3 : *«la relation $\approx$ est invariante pour la fonction d'insertion»* devient

```
Theorem biso_ins_heap :
  Forall 'a:type. Forall c: 'a -> 'a -> bool.
  Forall x1:'a. Forall x2:'a.
  Forall b1: 'a btree. Forall b2: 'a btree.
   `(biso b1 b2) -> `(biso (ins_heap c x1 b1) (ins_heap c x2 b2))
```

La présence des deux étiquettes `x1` et `x2` nous donne le corrolaire

```
Theorem biso_ins_heap_x :
  Forall 'a:type. Forall c: 'a -> 'a -> bool. Forall x1:'a. Forall x2:'a.
  Forall b: 'a btree.
   `(biso (ins_heap c x1 b) (ins_heap c x2 b))
```

dont nous verrons l'utilité tout à l'heure.

> *Note : l'étude des cas induits par l'utilisation de l'alternative* `if-then-else` *dans la définition de la fonction d'insertion est réalisée, dans notre système, par l'application du théorème suivant*
>
> ```
> Forall 'a : type. Forall X : ('a -> Prop).
> Forall b : bool. Forall x : 'a. Forall y : 'a.
>   ((`(b) -> X(x)) And ((Not `(b)) -> X(y))) -> X(if b then x else y)
> ```
>
> *prouvable dans notre système. Son usage étant fréquent, nous avons codé la tactique dédiée* `IntroIf` *qui réalise cette application.*
>
> *La quantification* `Forall X : ('a -> Prop)` *est une quantification du second ordre. La constante* `Prop`*, et, par extension le type* `'a -> Prop`*, est, à l'instar de la constante* `type` *une indication syntaxique de sorte.*

3.4. Arbres adéquats

On ne peut transposer directement la définition inductive de l'ensemble $WBI[A]$ en posant, selon notre usage, une fonction caractéristique récursive équivalente. En effet, la définition que nous avons donné de $WBI[A]$ fait usage, dans sa clause inductive, d'une quantification universelle (*«pour tout* $a \in A$*»*).

Il faut donc trouver une *astuce* pour obtenir un effet sémantiquement similaire à l'usage de ce *«pour tout»*. L'effet de ce *«pour tout»* est d'*abstraire* l'étiquette a. Du point de vue des fonctions, l'effet d'abstraction est produit par l'introduction de *paramètres*. Lorsque l'on passe aux fonctions caractéristiques, pour obtenir un effet d'abstraction des étiquettes, il suffit de passer l'une d'elle en paramètre de la fonction. D'où la définition :

```
let rec ins_heap_wb c x b =
  match b with
    Lf -> true
  | Br(y,b1,b2) ->
     (ins_heap_wb c x b1) &&
     ((biso b2 b1) || (biso b2 (ins_heap c x b1)))
```

On pourrait s'assurer de l'indépendance de notre définition vis-à-vis du paramètre `x` en prouvant que

```
Forall x1 :'a. Forall x2 :'a. (ins_heap_wb c x1 b) = (ins_heap_wb c x2 b)
```

mais nous n'avons pas explicitement eu besoin de ce résultat dans la pratique où le lemme `biso_ins_heap_x` s'est avéré suffisant.

Le lemme 4 : *«l'ensemble* $WBI[A]$ *est clos par* $\approx$*»* devient

```
Theorem biso_ins_heap_wb :
  Forall 'a:type. Forall c: 'a -> 'a -> bool. Forall x:'a.
  Forall b1: 'a btree. Forall b2: 'a btree.
    '(biso b1 b2) -> '(ins_heap_wb c x b1) -> '(ins_heap_wb c x b2)
```

Note : la preuve de ce lemme, dans les cas où `b1` est vide et `b2` non, `b1` n'est pas vide et `b2` si, fait appel au principe que ex falsum quod libet (régle d'absurdité intuitioniste). Notre système, dont le langage logique n'a pas de symbole pour l'absurdité, ne connaît qu'une version restreinte de ce principe : la régle

```
Gamma |- '(false)
-----------------
    Gamma |- A
```

Dans notre cas, par exemple, une hypothèse telle que `'(Lf = Br(a1, b3, b4))` se réduit, par évaluation symbolique, à `'(false)`. Ce qui permet de conclure.

Le lemme technique 5 : *«si la fonction d'insertion fait croître strictement la hauteur maximale d'un arbre de* $WBI[A]$*, c'est que l'arbre était parfaitement équilibré»* qui s'énonce

```
Theorem hmax_ins_heap_eq_S_hmax__hmin_eq_hmax :
  Forall 'a:type. Forall c: 'a -> 'a -> bool. Forall x:'a. Forall b: 'a btree.
    '(ins_heap_wb c x b) -> '((hmax (ins_heap  c x b)) = (S(hmax b)))
     -> '((hmin b) = (hmax b))
```

est maintenant abordable. Sa preuve suit globalement la structure de celle donnée «à la main», si ce n'est le recours au lemme d'indépendance `hmax_ins_heap_x` éludant l'utilisation du racourçi

$\alpha \in \{a, a'\}$. Également, l'usage du principe *ex falsum* de la résolution du dernier cas de la preuve «à la main» de ce lemme est implantée en utilisant explicitement la régle d'élimination de la négation de la déduction naturelle, en fait, la régle de négation droite de la *déduction libre*.

> *Note : le système de déduction de PAF ! est la déduction libre introduite par M. Parigot dans le cadre de l'interprétation algorithmique de la logique classique ([Par90]). Ce système formel a l'avantage de contenir, comme règles dérivées, aussi bien la déduction naturelle que le calcul des séquents. Il facilite l'implantation de nombre de* tournures de raisonnement *usuelles ; en particulier, le raisonnement sur les hypothèses qui permet de s'approcher du raisonnement* en avant *(top-down).*

3.5. Les théorèmes

Le théorème 1 : *«$WBI[A]$ est clos par la fonction d'insertion»* s'énonce

```
Theorem ins_heap_wb_ins_heap :
  Forall 'a:type. Forall c: 'a -> 'a -> bool. Forall x:'a. Forall b: 'a btree.
    '(ins_heap_wb c x b) -> '(ins_heap_wb c x (ins_heap c x b))
```

Si l'esprit de la preuve «à la machine» suit celui de la preuve «à la main», la lettre s'en éloigne en ceci : à l'instar de ce que nous avions fait pour le lemme 1, nous avons utilisé une astuce de notation :

> *Il faut montrer que* $ins(a, Br(a_0, b_1, b_2)) \in WBI[A]$*, c'est-à-dire* $Br(\alpha_1, b_2, ins(\alpha_2, b_1)) \in WBI[A]$ *avec* $(\alpha_1, \alpha_2) = (a, a_0)$*, si* $c(a, a_0)$ *ou* $(\alpha_1, \alpha_2) = (a_0, a)$*, sinon.*

Le recours au couple (α_1, α_2) a permis de factoriser l'examen des cas introduits par l'alternative `if-then-else` de la définition de la fonction d'insertion. L'usage d'un tel de biais dans la preuve «à la machine» doit être explicitement justifié, alors que l'on peut rester implicite dans la rédaction «à la main». La preuve «à la machine» procéde donc plutôt à l'examen assez redondant des deux cas de l'alternative. Cependant, et c'est l'avantage des preuves «à la machine», les resources du «copier/coller» et du «chercher/remplacer» allégent le poids d'une fastidieuse répétition.

Le théorème 2 : *«$WBI[A] \subset WB[A]$»* se formule simplement

```
Theorem ins_heap_wb_is_wb :
  Forall 'a:type. Forall c: 'a -> 'a -> bool. Forall x:'a. Forall b: 'a btree.
    '(ins_heap_wb c x b) -> '(is_wb b)
```

Ici encore, la preuve sur machine, outre qu'elle donne plus de détails, est proche, dans sa structure de la preuve «à la main». L'ultime cas de la preuve, comme dans le cas du lemme 5 est résolu par utilisation explicite de la régle d'élimination de la négation.

Conclusion

Nous avons dans cet article établi la *correction opérationelle* de la fonction nodale d'un programme fonctionnel de tri par tas. Nous avons donné deux formes de cette preuve de correction

- une usuelle pour les algorithmiciens, rédigée «à la main» et qui fait appel à l'étendue et la souplesse de la tradition mathématique appliquée aux problèmes d'informatique ;
- une entièrement formalisée à l'aide d'un outil de preuve implanté sur machine qui s'inscrit dans le cadre inauguré il y a 3 décennies des *logiques pour les fonctions calculables* ([Mil73]).
  Soulignons, qu'à l'instar de ce qui est fait pour le langage de programmation LISP par Boyer-Moore ([Boy79]), nous avons donné une preuve formelle et mécaniquement vérifiée de la correction d'un *code source* du langage de programmation ML (dans son dialecte OCAML).

On peut donc à juste titre parler de *heap sort* à propos de ce programme fonctionnel :
- sa correction dénotationnelle («c'est un tri...») a formellement été établie dans nombre de systèmes, pour ne citer qu'un proche : [Ano00] et [Fil04], ainsi que notre précédent [Man96].
- sa correction opérationnelle («...par arbre équilibré») a fait l'objet de ce papier[3].

Note : l'auteur remercie Pierre Letouzey *pour sa relecture...à l'issue de laquelle, il s'est empressé de transcrire notre preuve en Coq :)*

Références

[Ano00] (Anonyme) *Sorting : Axiomatizations of sorts Heap* The Coq Standard Library `http ://coq.inria.fr/library/Coq.Sorting.Heap.html`

[Bar03a] Baro S. (2003) *Conception et implémentation d'un système d'aide à la spécification et à la preuve de programmes ML* Thèse de l'université PARIS 7.

[Bar03b] Baro S. et Manoury P. (2003) *Un système X, Raisonner formellement sur les programmes ML* In Proc. JFLA `http ://jfla.inria.fr/2003/actes`.

[Bir96] Bird R. S. (1996) *Functional algorithm design* Science of computer programming, vol. 26.

[Boy79] Boyer R. et Moore J S. (1979) *Computational logic* Academic Press, New York.

[Coq96] Cornes C. *et al.* (1996) *The coq proof assistant reference manual, version 5.10* Technical report, INRIA.

[Fil04] Filliâtre J.-C. et Letouzey P. (2004) *Functors for Proofs and Programs* Proc. The European Symp. on Programming, LNCS 2986.

[Kan87] Kahn G. (1987) *Natural Semantics* Proc. of Symp. on theoretical aspects of computer science, Passau, Allemagne, LNCS 247.

[Kri90] Krivine J.-L. Parigot M. (1990) *Programming with proofs* Journal of Information Processing and Cybernetics, 26 :3.

[Lop02] Lopez N. (2002) *Spécification Formelle de Systèmes Complexes Méthodes et Techniques* Thèse de doctorat CNAM.

[Man96] Manoury P. (1996) *Preuves et Programmes. Un cas d'école : preuve de correction de programmes fonctionnels de tris dans le système Coq* Rapport IBP 96/22 – disponible en `http ://www.pps.jussieu.fr/~eleph/Recherche/raplitp1.ps.gz`

[Mil73] Milner R. (1973) *Logic for Computable Functions : description of a machine implementation* Technical Report, Stanford University.

[Par90] Parigot M. (1990) *Free Deduction : An Analysis of "Computations" in Classical Logic* in Proc. First Russian Conference on Logic Programming, LNCS 592.

[Par92] Parigot M. (1992) *Recursive programming with proofs* Theoretical Computer Science, 94 :2.

[3] Le script complet est disponible en `http ://www.pps.jussieu.fr/~eleph/Recherche`