Preuves de programmes impératifs dans le système Coq

Tri par insertion d'un tableau

Nicolas Magaud
Stage de première année du Magistère d'Informatique et Modélisation de l'E.N.S. Lyon

Responsables du stage : Christine Paulin et Jean-Christophe Filliâtre

1 Juin - 11 Juillet 1998

Table des matières

Introduction

La validation d'un logiciel telle qu'elle peut se faire en B (ou dans d'autres systèmes formels) est obligatoire dès qu'il s'agit d'applications critiques (logiciels embarqués, protocoles de communication réseaux, logiciel de contrôle d'un réacteur nucléaire), ou plus proches de nous de protocoles bancaires (cartes, transferts de fonds...). Les tests expérimentaux ne suffisent pas à assurer que le logiciel est fiable. L'utilisation d'une méthode formelle permet non seulement de valider un système, mais aussi d'assurer l'adéquation entre le besoin exprimé (les spécifications) et le produit final.
Dans le cadre des langages purement fonctionnels basés sur le $\lambda-$calcul, on peut facilement faire des preuves en utilisant l'isomorphisme de Curry-Howard qui relie les notions de preuve et de programme. Cependant, la majorité des logiciels existants sont écrits dans des langages impératifs. Il est donc nécessaire de disposer de méthodes formelles de preuves sur les programmes impératifs. La logique de Hoare ou la méthode de preuves assertionnelles à la Floyd en sont des exemples. D'autre part, il semble intéressant de disposer d'une méthode formelle associée à un outil de preuves puissant.
Le système `Coq` [2, 1] est un assistant de preuves implantant une logique d'ordre supérieur, le Calcul des Constructions Inductives. Ce formalisme étant lui-même un $\lambda-$calcul typé, il est donc bien adapté à la preuve de programmes purement fonctionels. On peut légitimement s'interroger sur l'adaptation d'un tel système à la validation de programmes impératifs vis-à-vis de leurs spécifications.
Une méthode de correction totale "à la Hoare" de programmes impératifs (écrits dans un syntaxe proche de celle de `Caml`) a été récemment implantée dans le système `Coq`. A partir de la donnée d'un programme impératif annoté (comme en logique de Hoare : préconditions, postconditions, invariants de boucle...), la tactique engendre la liste des propriétés à démontrer pour valider le programme.
Dans ce rapport, on présente le développement d'une preuve d'un programme de tri par insertion d'un tableau. A travers cette preuve, il s'agit de voir si le langage de spécifications `Gallina` et les outils de preuve du système `Coq` conviennent pour établir la correction d'un programme. Après avoir défini ce programme et ces caractéristiques, on va spécifier précisément les propriétés qu'il doit vérifier; puis finalement, on établira chacune de ses propriétés à l'aide des outils de preuve du système `Coq`, ce qui validera le programme vis-à-vis de ses spécifications.

1 Préliminaires

On se propose de réaliser la preuve formelle de correction totale d'un programme de tri par insertion d'un tableau de taille N (dont les indices valides sont compris entre 0 et $N-1$) au moyen de la tactique `Programs`.

1.1 La tactique `Programs`

Cette tactique, dont on trouvera la description complète dans [1] sera considérée ici comme une boîte noire. On se contentera d'en décrire le comportement tel qu'on l'observe

depuis l'extérieur.
A partir d'un programme impératif (annoté dans le style de la logique de Hoare), la tactique construit un arbre de preuve incomplet i.e. dans lequel certaines propriétés restent à démontrer : ce sont les obligations de preuve engendrées. De plus, la tactique assure que si l'utilisateur parvient à montrer ces différents buts, la correction totale du programme sera établie. Cependant, ce dernier reste libre, soit de conclure que tous ces buts sont trivialement démontrables, soit d'utiliser les outils de preuve du système `Coq` pour les montrer, soit même d'utiliser un autre assistant à la démonstration mieux adapté au cas considéré.

1.2 Premier programme envisagé

On considère d'abord un programme naturel de tri par insertion d'un tableau comme on pourrait l'écrire en `Caml`. La fonction `insertion` prend en paramètre le tableau a, sa taille n, ainsi que l'indice i de l'élément à insérer. Les paramètres de la fonction `tri` sont quant à eux le tableau a et sa taille n.

```
let insertion a n i =
    let v = a.(i) and j= ref i in
        begin
           while ((!j > 0) && (a.(!j-1) > v)) do
                  a.(!j) <- a.(!j-1);
                  j := !j-1
           done;
           a.(!j) <- v;
           a
        end
;;

let tri a n =
  let i = ref 1 in
      begin
        while (!i < n) do
          (insertion a n !i) ;
          i := !i+1
        done;
        a
      end
;;
```

Ici, on utilise le fait que le mode habituel d'évaluation des connectives logiques (or, and) est paresseux. En `Caml` (ou `C`) par exemple, l'évaluation du second argument de la connective `and` ne s'effectue qu'après celle du premier argument et seulement si ce dernier s'est évalué à `true`. Dans le programme ci-dessus, cela permet de garantir que l'on n'accède jamais au tableau hors des bornes.
Or la tactique de correction utilisée considère les connectives comme des fonctions strictes, et ne fait aucune hypothèse sur l'ordre d'évaluation des paramètres d'une connective logique. Ainsi, il n'est pas possible de prouver la correction du programme précédent dans ce cadre. Et, en effet, il est clair que le programme proposé n'est pas correct vis-à-vis de la sémantique du langage utilisé pour décrire les programmes.

On va donc reprendre le programme précédent et le modifier légèrement afin qu'il n'utilise plus la sémantique paresseuse des connectives, mais qu'explicitement les tests concernant l'indice i et la valeur $A[i]$ soient séparés. De cette façon, on pourra garantir que tout accès au tableau se fait bien dans les bornes.

1.3 Programme final

La nouvelle version proposée introduit une nouvelle variable booléenne et conduira à une spécification et une preuve un peu plus compliquées de la validité du programme.
Le programme de tri que l'on considèrera dans la suite de cette étude est le suivant (où seule la fonction `insertion` a été modifiée).

```
let insertion a n i =
    let v = a.(i) and j=ref i and fini = ref false in
        begin
           while ((!j > 0) && (not !fini)) do
             (if (a.(!j-1) > v)
                 then begin
                      a.(!j) <- a.(!j-1);
                      j := !j-1
                      end
                 else begin fini := true end)
           done;
      a.(!j) <- v
    end
;;

let tri a n =
  let i = ref 1 in
      begin
        while (!i < n) do
          (insertion a n !i) ;
          i := !i+1
        done;
        a
      end
;;
```

On dispose maintenant d'un programme dont on veut montrer qu'il réalise bien le tri par insertion d'un tableau de taille N (indicé de 0 à $N-1$). Pour cela, on doit maintenant formaliser, i.e. spécifier quelles propriétés un tel programme doit vérifier.

2 Spécifications

Pour obtenir une preuve du programme de tri par insertion présenté dans la section précédente, on va procéder en deux temps en utilisant le *caractère modulaire* du programme. Ainsi la spécification et la preuve elle-même seront modulaires. Une fois que l'on aura établi une

preuve de correction du tri s'appuyant sur une spécification précise de la fonction `insertion` (donnée d'une pré et d'une postcondition, liste des effets), on pourra facilement construire une nouvelle preuve d'un programme de tri insertion par dichotomie par exemple. Pour cela, il suffit de faire la preuve d'une fonction d'insertion par dichotomie dans un tableau trié qui ait les mêmes pré et postconditions que la fonction `insertion` précédente.

2.1 Propriétés générales sur les tableaux

Le langage de description des programmes impératifs n'implémente pas physiquement les tableaux, ces derniers sont représentés dans `Coq` par un type abstrait et axiomatisés par les opérations de lecture (`access`), écriture (`store`) et de création (`new`). Nous reviendrons sur la manière dont les tableaux sont gérés dans la section 3.1.1 consacrée à la preuve proprement dite.
On distingue deux types de propriétés sur les tableaux :

- la notion d'échange de deux éléments et celle de permutation;
- la notion d'ordre et d'éléments triés dans le tableau.

2.1.1 La bibliothèque sur les échanges et les permutations

Les propriétés des échanges et des permutations y sont définies. La relation de permutation est définie comme la plus petite relation réflexive, symétrique et transitive qui contienne tous les échanges.

2.1.2 La bibliothèque `Sorted.v`

Pour faire des preuves sur les tableaux triés, il fallait disposer d'un bibliothèque de propriétés simples. On a tout d'abord défini la propriété `sorted_array` qui traduit le fait pour une partie d'un tableau d'être triée par ordre croissant, puis à partir de cette définition, plusieurs lemmes simples sur les tableaux triés ont été établis.
Le lemme `sorted_elements` donne des propriétés d'ordre entre deux éléments d'un même sous-tableau trié. On dispose d'autre part de lemmes permettant de passer d'un tableau trié des indices i à j à un tableau trié des indices i à $j+1$ sous réserve dans $A[j+1]$ soit plus grand que $A[j]$.

2.2 Spécifications des programmes

Informellement, ce que l'on désire obtenir est la propriété suivante : "A partir d'un tableau quelconque de taille $N > 0$, on veut obtenir un tableau trié par ordre croissant et qui soit une permutation du tableau initial". Il s'agit maintenant de formaliser tout cela dans le langage de spécification de `Coq`.

2.2.1 Le programme de tri

La fonction `insertion` insère le $n^{\text{ième}}$ élément d'un tableau dans le sous-tableau trié $[0 \ldots n-1]$. Elle a pour postcondition que le tableau est maintenant trié de 0 à n et que celui-ci est bien une permutation du tableau de départ.
On peut facilement écrire un "prototype" de la fonction insertion sur lequel on s'appuyera pour montrer la correction du programme de tri, puis nous reviendrons plus tard sur la spécification précise interne de la fonction `insertion`. Pour l'instant, nous nous contentons de l'observer de l'extérieur.

```
Global Variable insertion : (n:Z)
                            returns u:unit
                            reads A
                            writes A
                            pre (sorted_array N A `0` `n-1`)/\(`1<=n<N`)
                            post ((sorted_array N A `0` n)/\(permut A A@))
                            end.
```

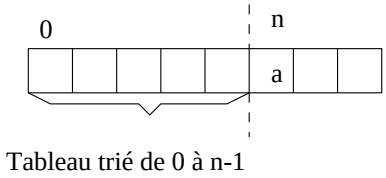

FIG. 1 - *Insertion du $n^{ième}$ élément d'un tableau dans le sous-tableau trié* $0 \ldots n-1$

On peut remarquer la notation `A@` dans `(permut A A@)`, celle-ci permet de faire référence au tableau `A` avant l'exécution de `insertion`. La notation particulière `A@0` permet de faire référence au tableau `A` initial. Cela permet de parler facilement du tableau à n'importe quel étape de l'exécution du programme sans avoir à utiliser un grand nombre de variables temporaires.
A partir de cette représentation et du code de `tri`, on peut spécifier l'invariant et le variant utilisés dans la preuve. Pour le variant, il est clair que $N-i$ décroit strictement à chaque itération et qu'il reste toujours supérieur ou égal à zéro. L'invariant, quant à lui, se divise en deux parties, l'une traduit le fait que le tableau se trie un peu plus à chaque étape et l'autre assure qu'à chaque itération de la boucle, le tableau courant est bien une permutation du tableau initial.
Enfin, on annote le programme par une postcondition représentant exactement le résultat attendu : le tableau a est trié par ordre croissant et c'est une permutation du tableau de départ.

```
Correctness tri
  {`N>0`}
  let i = ref 1 in
    begin
```

```
  while (!i < N) do
    { invariant ((sorted_array N A `0` `i-1`)/\(permut A A@0)/\(`1<=i`)) as I
      variant `N-i` }
    (insertion !i) { (sorted_array N A `0` i)/\(permut A A@) };
    i := (Zs !i)
  done
end
{ (sorted_array N A `0` `N-1`)/\(permut A A@0) }.
```

2.2.2 La fonction `insertion`

Le corps de cette fonction se réduit à une boucle `while`, il s'agit donc de trouver un invariant permettant de prouver la correction partielle et un variant permettant d'établir la terminaison, ce qui nous donnera une preuve de correction totale de la fonction `insertion`. La difficulté réside dans le fait que l'invariant n'est pas très général. Avant d'écrire l'invariant relatif aux propriétés de tri du sous-tableau considéré, interessons-nous au problème de la permutation.
Ici, le tableau n'est pas modifié par des échanges, mais à la première itération, un élément du tableau (celui que l'on veut insérer dans la partie triée) est remplacé par son voisin de gauche et le tableau n'est donc plus une permutation du tableau initial. Cependant, il est clair que tout au long de la boucle, un des éléments du tableau est présent dans deux cases successives, on peut donc en substituter un par la valeur de l'élément manquant dans le tableau (stocké dans v) et avoir ainsi une permutation du tableau de départ.
Pour ce qui concerne l'invariant sur les propriétés de tri, il est un peu plus compliqué, à l'itération où j vaut $j0$, on doit avoir des informations et pouvoir accéder aux cases d'indices $j0-1$ et $j0+1$. De ce fait, on ne peut pas définir un invariant simple qui soit vrai quelle que soit la valeur de j; cependant les problèmes se trouvent aux bornes du domaine de j. On a donc deux cas particuliers: $j = n$ (la première itération de la boucle), ce qui correspond à `inv_n` et $j = 0$ (l'invariant en fin de corps de boucle lorsque l'on avait $j = 1$ et $A[j-1] > v$ à l'entrée de la boucle), ce qui correspond à `inv_0`. Finalement, on a un invariant général `global_inv` lorsque j est compris strictement entre 0 et n ($0 < j < n$). Voici les définitions de ces 3 invariants :

- l'invariant "initial"

```
Definition inv_n := [N:Z][A:(array N Z)][j:Z][n:Z][v:Z][b:bool]
   (`j=n` ->
      (  (sorted_array N A `0` `j-1`)
      /\ (Zle v A#[j])
      /\ ((b=true) -> (Zle A#[`j-1`] v))
      )
   ).
```

- l'invariant "final"

```
Definition inv_0 := [N:Z][A:(array N Z)][j:Z][n:Z][v:Z]
   (`j=0` ->
```

```
        (  (sorted_array N A j `n`)
        /\ (Zle v A#[j])
        )
     ).
```

– l'invariant général pour $0 < j < n$

```
  Definition global_inv := [N:Z][A:(array N Z)][j:Z][n:Z][v:Z][b:bool]
    (`0<j<n` ->
        (  (sorted_array N A `0` `j-1`)
        /\ (sorted_array N A `j` `n`)
        /\ (Zle A#[`j-1`] A#[`j+1`])
        /\ (Zle v A#[j])
        /\ ((b=true) -> (Zle A#[`j-1`] v))
        )
    ).
```

L'invariant total, du moins pour la partie concernant la propriété de tri sera formé de la conjonction des 3 invariants précédents.
La conjonction de cet invariant, de l'invariant de permutation décrit précédemment et d'une dernière propriété `$0 <= j <= n$` nécessaire pour prouver que les accès au tableau se font bien dans les bornes nous donne notre invariant définitif tel qu'on peut l'observer ci-dessous dans le programme annoté.

```
Correctness insertion
   fun (n:Z) ->
    {((sorted_array N A `0` `n-1`) /\ `1<=n<N`)}
    let v = ref A[n] in
    let j = ref n in
    let fini = ref false in
    begin
      while ((!j > 0) and (not (sumbool_of_bool !fini))) do
        { invariant ((  (global_inv N A j n v fini)
                        /\ (inv_0 N A j n v)
                        /\ (inv_n N A j n v fini)
                     )
                     /\ (permut (store A j v) A@0)
                     /\ (`0<=j<=n`)
                     ) as I
          variant (fini,j) for R }
        (if (A[(Zpred !j)] > !v)
           then begin
                  A[!j] := A[(Zpred !j)];
                  j := (Zpred !j)
                end
           else begin fini := true end)
                  {((R (fini,j) (fini@,j@)) /\
                     ((  (global_inv N A j n v fini)
                        /\ (inv_0 N A j n v)
                        /\ (inv_n N A j n v fini)
                     )
```

```
                    /\ (permut (store A j v) A@0)
                    /\ (`0<=j<=n`)
                    ))}
  done;
     A[!j]:=!v
   end
 { (sorted_array N A `0` n) /\ (permut A A@)}.
```

Il ne reste plus qu'à définir le variant et la relation d'ordre bien fondé associée. Le `while` a deux possibilités de terminaison, soit $j <= 0$ soit $fini = true$, le variant doit donc traduire ces deux cas. On commence par définir un ordre bien fondé sur les booléens: $True < False$. Puis une relation R qui représente l'ordre lexicographique sur le type *booléen* $\times$ *entier relatif positif*. Cette relation est définie de manière suivante :

$$\forall (b,j)(b',j') \in bool * \mathbf{Z}^+ \quad (R\ (b,j)\ (b',j')) \iff (b <_{bool} b') \ \vee \ ((b = b') \wedge (Zwf\ \text{`0`}\ j\ j'))$$

où Zwf '0' est la relation d'ordre bien fondé usuelle sur les entiers relatifs positifs.
Rappel : soit un ensemble E et une relation d'ordre R sur E; on dit que R est une relation d'ordre bien fondé sur E si et seulement s'il n'existe pas de suite infinie strictement décroissante d'éléments de E.

2.3 Spécifications et `Coq`

La spécification du problème est largement indépendante du formalisme logique utilisé. Ici, la quasi-totalité des propriétés et des axiomes sont exprimés dans la logique du premier ordre. La puissance du Calcul des Constructions Inductives n'est pas nécessaire pour pouvoir spécifier, un formalisme logique basé sur la théorie de ensembles de Zermelo-Frankel serait suffisant. Cependant, le calcul des constructions inductives s'avère très utile lorsqu'il s'agit de définir inductivement les permutations à partir des échanges. De plus, la tactique `Correctness` [1] exploite également tous les avantages d'un formalisme logique d'ordre supérieur pour analyser le programme annoté et générer automatiquement les obligations de preuve.

3 Résolution des obligations de preuve engendrées

Une fois le programme annoté, on le soumet à la tactique `Correctness` qui va engendrer l'arbre de preuve et les obligations de preuves à montrer pour valider le programme. Dans cette section, nous allons nous intéresser à la forme des obligations de preuve ainsi qu'à la manière de les résoudre au moyen de l'assistant de preuve `Coq`.

3.1 Forme des obligations de preuves

Les obligations de preuve sont assez naturels et proches de celles que l'on exigerait d'une preuve "à la main" ou d'une preuve plus formelle en Logique de Hoare.

3.1.1 La gestion des tableaux

Les tableaux sont considérés de manière globale, la modification d'une valeur dans le tableau est donc considérée comme une modification du tableau tout entier. A cause du problème d'aliasing des indices accédés, on ne dispose d'aucune règle sur l'affectation dans un tableau. Il s'agit d'un problème commun avec d'autres formalismes, en Logique de Hoare par exemple.
Exemple : Considérons le tableau suivant indicé de 1 à n :

2	2	$\cdots\cdots$

Si l'on disposait de la règle d'affectation usuelle, $\{p[x \leftarrow e]\}\ x := e\ \{p\}$, on pourrait écrire

$$\{1 = 1\}\ T[T[2]] := 1\ \{T[T[2]] = 1\}$$

et comme

$$(T[1] = 2\ \wedge\ T[2] = 2)\ \rightarrow\ 1 = 1,$$

on pourrait déduire par affaiblissement de la précondition que

$$\{T[1] = 2\ \wedge\ T[2] = 2\}\ T[T[2]] := 1\ \{T[T[2]] = 1\},$$

ce qui est évidemment faux.

3.1.2 Les obligations liées à une boucle `while`

$$while\ B\ do\ \{\ invariant\ I\ \ variant\ j\ for\ R\}\ S\ done$$

Les obligations de preuve relatives à ce segment sont :

- l'ordre R est bien fondé;
- l'invariant est vérifié à l'entrée de la boucle;
- l'invariant est conservé par une itération de la boucle et le variant a diminué strictement;

3.2 Validation du programme de tri par insertion

3.2.1 La fonction insertion

Les obligations de preuve résultant de l'application de la tactique `Correctness` à la fonction d'insertion sont les suivantes :

- Prouver que l'ordre R sur le type *booléen* $\times$ *entier positif* est bien fondé;
- Conservation de l'invariant avec le résultat du test du `if` valant true;
- Affectation dans la "$i^{\text{ème}}$" du tableau A;

- Conservation de l'invariant avec le resultat du test du `if` valant false;
- Accès à la case $j-1$ du tableau A;
- Validité de l'invariant à l'entrée de boucle;
- Déduction de la postcondition de `insertion` sachant que l'invariant est vrai en sortie de boucle et que le test de boucle s'évalue à faux;
- Accès à la case j du tableau A (après la fin de boucle);
- Définition de la variable locale v : accès à la case n du tableau A.

Ainsi, la correction totale (au sens de Hoare) de la fonction `insertion` est établie. A partir de cette preuve, on prouve le programme de tri par insertion en remplaçant le prototype de `insertion` par sa preuve dans `Coq`.

3.2.2 Le programme de tri

Pour ce qui concerne le programme de tri, batî sur le prototype de la fonction `insertion`, les 4 obligations de preuves engendrées sont les suivantes :

- Etablissement de la précondition de `insertion` à partir des annotations du programme `tri`;
- Décroissance du variant et conservation de l'invariant à chaque itération;
- Validité de l'invariant à l'entrée de boucle;
- Etablissement de la postcondition du programme de tri après la sortie de boucle.

On peut remarquer que dans le cas de l'ordre sur les entiers relatifs positifs utilisé ici, il n'est pas nécessaire d'établir soi-même qu'il est bien-fondé (l'énoncé et la démonstration de ce résultat classique figurant dans la bibliothèque de `Coq`). C'est la tactique qui résoud ce but de façon automatique.

3.3 Utilisation de `Coq` pour la résolution des obligations de preuve

On a vu que les obligations de preuve à montrer sont relativement proches de celles d'une preuve informelle; en effet celles-ci correspondent aux propriétés que l'on aurait dû démontrer si l'on avait fait la preuve à la main en logique de Hoare.
Toutes ces obligations de preuves comprennent un grand nombre de buts à caractère arithmétique (inégalité sur des entiers relatifs...). Ces obligations proviennent principalement des vérifications de bornes pour les accès aux tableaux ainsi que de la décroissance du variant. Le système `Coq` dispose d'une tactique particulièrement adaptée à la résolution de ce type de buts : **Omega** [1, Chap 17].
Les preuves sont longues mais ne sont pas très difficiles. La longueur des preuves n'est pas seulement liée à l'alourdissement induit par la formalisation, mais aussi à la spécification assez compliquée à laquelle on a dû recourir dans la fonction `insertion`. La définition utilisée

pour l'invariant s'appuie sur un grand nombre de raisonnements par cas qui alourdissent la preuve.
Globalement, les outils de preuve de `Coq` sont assez bien adaptés à la résolution des obligations de preuve; cependant les preuves ne se font que très peu automatiquement. De nombreux buts relativement simples, voire triviaux ne sont pas montrés par la tactique `Auto`. Les preuves seraient grandement simplifiées si cette tactique utilisait l'ensemble des lemmes simples de la bibliothèque de `Coq`. Ainsi, les buts évidents à montrer seraient résolus quasi-automatiquement et l'on pourrait se consacrer aux phases plus délicates de la démonstration (telle que la preuve que l'ordre lexicographique issu des ordres Z_{bool} et Zwf '0' est bien fondé).
En conclusion, voici une comparaison de l'importance des différentes parties de la preuve :

- Le programme lui-même s'écrit en une vingtaine de lignes;
- Les spécifications des propriétés du programme et des invariants s'étendent sur une centaine de lignes environ;
- Les preuves liés au programme considéré (i.e. sans les bibliothèques sur les permutations de tableaux, ni les tableaux triés) représentent approximativement entre 500 à 600 lignes de preuve ("retravaillées", les premières versions tournant autour de 800 à cause de maladresses de débutant en `Coq`).

On obtient donc un rapport de 25 entre la longueur du programme et sa preuve. La spécification et la vérification d'un programme dans un système formel comme `Coq` reste donc relativement lourde, mais possible en un temps raisonnable pour un utilisateur expérimenté disposant de bonnes bibliothèques de base.

Conclusion

Ce travail s'inscrit dans une contribution à `Coq` en fournissant une bibliothèque de lemmes sur les tableaux triés `Sorted.v`, ainsi qu'un exemple non trivial de preuve d'un programme impératif communement utilisé. Il s'agit d'un des premiers exemples de preuves de programmes impératifs réalisés en `Coq`. L'intégralité de la preuve est disponible à l'adresse suivante `http://pauillac.inria.fr/coq/contribs/summary.html` avec l'ensemble des contributions des utilisateurs du système `Coq`.
Cette preuve du tri par insertion montre qu'il est possible de traiter des exemples de preuves de programmes non triviaux et qu'il est envisageable (et envisagé : une preuve de Quicksort a été réalisé il y a peu en `Coq`) de faire des preuves de programmes conséquents. L'apparition et l'amélioration de bibliothèques telles que `Sorted.v` permettent de simplifier et surtout d'accélérer l'écriture de nouvelles preuves. Le nouvel utilisateur évite en effet de redémontrer un grand nombre de résultats simples nécessaires à la validation du programme qu'il étudie. De plus, une fois la preuve terminée, l'extraction effective d'un programme ne pose aucune difficulté. En effet, celle-ci peut être réalisée sans difficulté vers des langages comme `Pascal` ou `Caml` à partir de la preuve, il suffit en effet de "pretty-printer" le code donné par l'utiliseur vers le langage de réalisation (en étant attentif aux différences sémantiques entre ces deux

langages). Le résultat de cette extraction est donc un programme zéro-défaut pour lequel on a une garantie de fiabilité totale.

Références

[1] B. Barras, S. Boutin, C. Cornes, J. Courant, D. Delahaye, D. de Rauglaudre, J.-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, P. Loiseleur, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, and B. Werner. *The Coq Proof Assistant Reference Manual Version 6.2*. INRIA-Rocquencourt-CNRS-Université Paris Sud- ENS Lyon, May 1998.

[2] G. Kahn G. Huet and C. Paulin-Mohring. *The Coq Proof Assistant A Tutorial Version 6.2*. INRIA-Rocquencourt-CNRS-Université Paris Sud- ENS Lyon, May 1998.