HAL
archives-ouvertes.fr

L'Assistant de Preuve Coq Table des matières

Sandrine Blazy, Pierre Castéran, Hugo Herbelin

▶ **To cite this version:**

Sandrine Blazy, Pierre Castéran, Hugo Herbelin. L'Assistant de Preuve Coq Table des matières. Techniques de l'Ingenieur, Techniques de l'ingénieur, 2017. <hal-01645486>

HAL Id: hal-01645486
https://hal.inria.fr/hal-01645486

Submitted on 23 Nov 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

L'Assistant de Preuve Coq

Sandrine Blazy, Professeur, Université de Rennes 1
Pierre Castéran, Univ. Bordeaux, LaBRI, CNRS, INP-Bordeaux
Hugo Herbelin, Chercheur Inria Paris

23 novembre 2017

Table des matières

1 Exemple de développement en Coq : preuve de correction d'un algorithme simple **1**

2 Applications principales **7**

3 Coq et sa place en informatique **8**

4 Évolution **9**

5 Apprendre Coq **9**

6 Conclusion **10**

Face à l'importance croissante des composants informatiques — logiciels et matériels — dans des domaines critiques *aussi divers que transports, énergie, santé, finance, etc., le besoin de composants* sûrs *se fait de plus en plus impérieux.*

Par exemple, l'exécution d'un programme doit se faire sans erreurs dans les conditions d'application prévues : absence d'erreur à l'exécution ou de bouclage non désiré, conformité à une spécification fonctionnelle.

Or nous savons, depuis Turing [6] qu'aucun algorithme ne peut automatiquement prendre en donnée un programme quelconque et rendre en un temps fini un diagnostic de correction. Par conséquent, seule une partie de la tâche de certification d'un logiciel peut être automatisée. Pour le reste, on peut recourir à des outils interactifs*, où la preuve — souvent complexe — de correction doit être guidée par l'utilisateur.*

Les assistants de preuve *[22, 10, 23, 11, 15] sont des logiciels interactifs permettant à leur utilisateur de prouver des théorèmes en le déchargeant des tâches fastidieuses et susceptibles d'être entachées d'erreurs. Outre cette fonction d'assistance à l'écriture de démonstrations, ces logiciels vérifient que toutes les preuves construites sont complètes et respectent toutes les lois de la logique.*

Par « théorème », nous entendons toute sorte d'énoncé concernant soit le domaine mathématique, soit le comportement de programmes écrits dans un langage donné. Citons deux exemples, prouvés en Coq :

> *« Toute carte planaire peut être coloriée avec au plus quatre couleurs » [14]*
>
> *Le compilateur CompCert garantit que tout code exécutable qu'il génère se comporte exactement comme le programme source C dont il est issu [19, 1].*

Coq est un logiciel puissant et complexe, en constante évolution. Dans cet article, nous proposons en première partie une introduction sur un exemple très simple de preuve d'un petit programme fonctionnel. Une deuxième partie présente quelques applications de taille réelle. Nous terminerons par des indications sur l'apprentissage de Coq et des précisions sur l'évolution de cet outil.

1 Exemple de développement en Coq : preuve de correction d'un algorithme simple

La preuve de correction d'un programme fonctionnel simple nous permet de présenter quelques aspects essentiels du logiciel Coq : énoncés mathématiques et spécification formelle de programmes, outils pour la preuve interactive, etc. L'exemple proposé consiste en la définition d'une fonction de tri sur les listes, accompagnée d'une preuve formelle de sa correction.

Conventions typographiques Nous adoptons les conventions suivantes afin que le lecteur puisse facilement reconnaître quel texte un utilisateur doit soumettre à Coq et quelles sont les réponses du système.

- Le texte soumis à Coq est écrit dans des encarts bleu clair.
- Les réponses de Coq sont présentées en italique dans des encarts gris.
- Les expressions composées placées dans le texte de l'article sont entourées de parenthèses en gris, comme par exemple (**9** + **4**), afin d'en améliorer la lisibilité. Ces parenthèses ne font pas partie du texte Coq.

1.1 Établissement d'un contexte de travail

L'assistant de preuve Coq est accompagné d'une vaste bibliothèque standard comportant entre autres la définition de constructions usuelles : types de données, notions mathématiques et logiques de base, etc. On peut la consulter depuis l'onglet `Documentation` du site du logiciel [10].

Dans notre exemple, nous utiliserons le module sur les listes et celui sur les chaînes de caractères. La commande **Require Import** permet de charger ces modules et ainsi de bénéficier d'une base de fonctions et de propriétés élémentaires en lien avec ces structures de données.

```
Require Import List String.
```

Quelques lignes de commande nous permettent de disposer d'une syntaxe simple à la OCaml [2, 20] pour écrire des listes et des chaînes de caractères dans nos programmes et nos exemples :

```
Import ListNotations.
Open Scope list_scope.
Open Scope string_scope.
```

La liste contenant les entiers naturels 3, puis 4 et 5 peut s'écrire indifféremment **[3 ; 4 ; 5]** ou (**3 :: [4 ; 5]**) ou encore (**cons 3 (cons 4 (cons 5 []))**). Son *type* s'écrit (**list nat**) en Coq. La commande **Check** *terme* permet de vérifier que *terme* est correctement typé et de donner son type.

```
Check [4; 5; 6].
```

```
[4; 5; 6] : list nat
```

```
Check ["Anne"; "Pierre"; "Pascale"; "42"].
```

```
["Anne"; "Pierre"; "Pascale"; "42"] : list string
```

Notre définition se fait dans une *section*, l'équivalent des blocs des langages de programmation usuels, dans laquelle nous déclarons un type arbitraire **A**, ainsi qu'une fonction de comparaison **leb** de type (**A → A → bool**). Cette déclaration signifie que **leb** prend deux arguments de type **A** et renvoie une valeur booléenne. L'application de **leb** à deux expressions **a** et **b** de type **A** se note en Coq (**leb a b**), de façon cohérente avec OCaml. Signalons que le nom **leb** a été constitué à partir de la locution "**l**ess or **e**qual" et de l'adjectif "**b**oolean".

La syntaxe extensible de Coq nous permet de relier la fonction **leb** à la notion mathématique d'inégalité. Précisons qu'il ne s'agit que d'une notation commode : du point de vue de la programmation, **leb** est une fonction booléenne *quelconque*. Nous verrons plus loin quelle propriété cette fonction doit satisfaire pour que le tri associé soit correct.

```
Section Insertion_sort.

  Variable A : Type.
  Variable leb : A → A → bool.
  Notation "a ≤ b" := (leb a b = true).
```

1.2 Définition du tri par insertion

Coq permet d'écrire des programmes purement fonctionnels dans un formalisme qui peut rappeler les langages de programmation OCaml et Haskell. Une différence très forte tient aux restrictions sur la récursion, qui interdit notamment l'écriture de programmes dont l'exécution pourrait boucler. Dans le cas des listes, un schéma de définition récursive de fonction sur les listes se fait par *filtrage*. La définition suivante distingue deux cas pour la liste **l** :

- **l** est la liste vide `[]`
- **l** se décompose en une tête **b** suivie d'une queue **l'**, sur laquelle porte l'appel récursif (**insert a l'**)

```
Function insert (a:A) (l: list A) : list A :=
    match l with
     | [] => [a]
     | b::l' => if leb a b
                   then a::l
                   else b::insert a l'
    end.
```

Notre *intention* est que (**insert a l**) renvoie la liste obtenue à partir de **l** en insérant **a** " à la bonne place ". En section 1.4, nous prouverons cette propriété, *à condition que* ***l*** *soit une liste triée.*

Il reste à définir la fonction principale de tri, par récursion sur la liste **l**. Le tri d'une liste **l** se fait par une suite d'insertions à partir de la liste vide.

```
Function sort (l: list A) : list A :=
    match l with
      | [] => []
      | a::l' => insert a (sort l')
    end.
```

1.3 Spécification de la fonction sort

Une fonction correcte de tri sur les listes de type (**list A**) doit prendre en argument une liste quelconque **l** et renvoyer une permutation de **l** telle que, si un élément **a** précède immédiatement un élément **b** dans **l**, alors **a** ≤ **b**.

Par exemple le tri de la liste **[3 ; 4 ; 1 ; 8 ; 4]** doit donner **[1 ; 3 ; 4 ; 4 ; 8]** mais non **[1 ; 3 ; 4 ; 8]** ni **[1 ; 3 ; 4 ; 8 ; 4]**.

La notion mathématique de *permutation* est déjà définie dans la bibliothèque standard de Coq. Une exploration de cette bibliothèque [12] nous donne l'adresse du module à importer. Ce module contient la définition de cette notion, ainsi que la preuve de quelques théorèmes. La commande **Search** permet par exemple de retrouver quel théorème du module **Permutation** montre que la relation « être une permutation de » est transitive. Il suffit à l'utilisateur de donner un *motif*

sous la forme de l'énoncé correspondant au théorème cherché. Ce motif contient trois "inconnues" **?x**, **?y** et **?z** qui doivent être remplacées chacune par un terme. Notons que le motif de notre exemple est "non-linéaire" : chaque occurrence d'une même inconnue doit correspondre au même terme.

```
Require Import Coq.Sorting.Permutation.
Search (Permutation ?x ?y →
            Permutation ?y ?z →
            Permutation ?x ?z).
```

```
Permutation_trans:
  ∀ (A : Type) (l l' l'' : list A),
  Permutation l l' → Permutation l' l'' →
  Permutation l l''
```

Une originalité de Coq est la cohérence entre le formalisme utilisé pour écrire d'une part les énoncés et leur preuve, d'autre part des types et les programmes.
Par exemple, le symbole → est utilisé à la fois pour représenter des types fonctionnels, comme le type de **leb***, et l'implication logique comme dans l'énoncé ci-dessus. De même, la quantification universelle* ($\forall a : A, B$) *peut servir à exprimer le polymorphisme ou la dépendance d'un programme vis-à-vis d'un paramètre.*
Le lecteur pourra trouver la raison de cette « surcharge » dans les présentations du Calcul des Constructions Inductives*, le système de types utilisé par Coq (voir par exemple [10, 7]).*
L'avantage de cette cohérence est une grande uniformité de traitement entre les programmes et leur preuve de correction. Elle facilite également l'apprentissage de Coq aux personnes possédant une expérience en programmation fonctionnelle.

La notion de liste triée est également définie dans la bibliothèque standard de Coq. Nous allons cependant la définir entièrement aux fins d'illustration des techniques utilisées en Coq.

En mathématiques, il est usuel de définir une propriété de façon *inductive*, par une énumération de conditions suffisantes pour qu'un objet satisfasse cette propriété. De telles définitions peuvent être *récursives*, c'est-à-dire s'exprimer en termes d'objets ayant déjà la propriété considérée.

Une liste l est triée si

1. l est la liste vide `[]` ,
2. ou l contient un seul élément,
3. ou l est de la forme ($a :: b :: l'$), où $a \leq b$ et la liste ($b :: l$) est triée.

Ce type de définition, annoncé en Coq par le mot-clef **Inductive** n'est pas sans rappeler les définitions de prédicats sous forme de clauses *Prolog*. L'en-tête de la définition suivante précise que **Sorted** est un *prédicat* sur le type (**list A**), c'est à dire que pour toute liste l de ce type, l'expression (**Sorted** l) est une proposition.

```
Inductive Sorted : list A → Prop :=
  | ls_0 : Sorted []
  | ls_1 : ∀ a, Sorted [a]
  | ls_2 : ∀ a b l,  a ≤ b →
                   Sorted (b::l) →
                   Sorted (a::b::l).
```

Nous sommes en mesure de spécifier formellement ce qu'est une fonction correcte de tri. Le prédicat **Sort_spec** prend en argument une fonction f opérant sur les listes et renvoie une proposition exprimant que pour toute liste l, la liste ($f\ l$) est une permutation triée de l.

```
Definition Sort_spec (f : list A → list A) :=
  ∀ l, let l' := f l in Permutation l' l ∧
                        Sorted l'.
```

1.4 Preuve interactive de correction de sort

Nous devons maintenant prouver que notre fonction de tri est correcte, ce qui revient à prouver le théorème d'énoncé (**Sort_spec sort**). Comme **sort** est définie à l'aide de la fonction auxiliaire **insert**, nous commençons par étudier les propriétés de celle-ci.

Rappelons que la fonction de comparaison **leb** a été spécifiée comme une fonction binaire *quelconque* à valeurs dans **bool**. Or, notre fonction de tri ne donnera des résultats corrects que si pour deux éléments quelconques a et b, la proposition $a \leq b \vee b \leq a$ est vraie. Nous exprimons cette contrainte de façon générique par un prédicat sur les fonctions booléennes à deux arguments :

```
Definition Total (comp : A → A → bool) : Prop :=
   ∀ a b, comp a b = false → comp b a = true.

Hypothesis leb_total : Total leb.
```

La première propriété à prouver est que l'insertion de x dans une liste l est une permutation de la liste obtenue en ajoutant x *en tête de* l. Nous annonçons à Coq le nom et l'énoncé du résultat que nous voulons prouver. Une fois cet énoncé accepté, — des points de vue de la syntaxe et de la logique de Coq, nous entamons la preuve interactive de ce lemme :

```
Lemma insert_perm :
  ∀ x l, Permutation (x :: l) (insert x l).
Proof.
```

Buts, sous-buts et tactiques

La preuve interactive d'un énoncé peut être une tâche longue et complexe.
Tout comme en programmation, il convient de structurer une démonstration en un ensemble de preuves plus simples d'énoncés auxiliaires correspondant à la notion de lemme *en logique mathématique. Cette structuration, non seulement facilite la recherche d'une démonstration, mais permet d'en comprendre la structure et de dégager certains « patrons de preuve ».*
Elle s'appuie sur les notions suivantes :

but, sous-but : *Un but correspond à une proposition à prouver. Il se décompose en un ensemble d'*hypothèses *et une* conclusion. *Les sous-buts sont créés lors de l'application de tactiques.*

tactique : *L'implantation de la stratégie « diviser pour régner » s'applique aux assistants de preuves depuis LCF [22]. Nous présentons dans un encadré une classification des tactiques utilisées dans notre exemple. Pour avoir une idée de tout le catalogue de tactiques offert par Coq, se référer à la documentation du système. Le catalogue des tactiques de Coq s'enrichit régulièrement des apports de contributeurs, sous forme de greffons et de bibliothèques. Les utilisateurs peuvent aussi définir des tactiques plus spécifiques à leur travail particulier.*

Les tactiques utilisées dans notre exemple

Nous ne pouvons pas présenter dans cet article l'imposant ensemble, — par ailleurs extensible, — de tactiques mises à la disposition de l'utilisateur de Coq. Nous nous contentons de classer les tactiques citées dans notre exemple en quelques catégories. La consultation du manuel de référence et des ouvrages sur Coq donnera une idée de la richesse et de la variété des outils permettant de mener à bien une démonstration.

- *règles élémentaires de la logique :* **split**, **apply**, **intro**, **rewrite**, etc.
- *notions mathématiques simples :* **reflexivity**, **transitivity**, **symmetry**
- *raisonnement par cas et par récurrence :* **case**, **case_eq**, **destruct**, **case**, **induction**, etc.
- *évaluation symbolique et simplification :* **cbn**, **simpl**,
- *décision et semi-décision :* **auto**, **trivial**,
- *tacticielles :* composition de tactiques : ($\boldsymbol{tac}_1$; $\boldsymbol{tac}_2$), résolution triviale des sous-buts engendrés par *tac* : (**now** ***tac***).

L'interaction entre le système et l'utilisateur prend alors la forme suivante. À chaque étape :

- Coq affiche une liste de *buts* à prouver pour terminer la démonstration. Un but correspond à un lemme présenté sous la forme d'un certain nombre d'hypothèses et d'une conclusion. Au début de l'interaction, le but initial correspond à l'énoncé du lemme à prouver, dans un contexte formé par les déclarations et hypothèses actives.
- L'utilisateur tente de *réduire* un des buts (par défaut le premier affiché) en demandant l'application d'une *tactique*. Cette opération peut ou non créer de nouveaux *sous-buts* qu'il faudra alors résoudre. Nous sommes donc en présence du paradigme « diviser pour régner » : l'application d'une tactique à un but est censée remplacer ce but par un nombre quelconque — possiblement nul —, de nouveaux sous-buts, que l'on espère plus faciles à résoudre.
- Quand il ne reste plus de buts à résoudre, Coq vérifie la preuve construite interactivement, et en cas de succès, l'enregistre dans sa base de données.

```
1 subgoal, subgoal 1

A : Type
leb : A → A → bool
leb_total : Total leb
------------------------------------------------------
∀ (x : A) (l : list A),   Permutation (x :: l) (insert x l)
```

De façon cohérente avec la structure récursive de la fonction **insert**, nous proposons d'attaquer le but courant par une tactique de *récurrence* sur *l*. Coq produit alors deux sous-buts associés respectivement au cas de base de la liste vide, et au cas d'une liste non vide. Seul le premier sous-but est affiché de

façon complète.

```
induction l.
```

```
2 subgoals, subgoal 1

A : Type
leb : A → A → bool
leb_total : Total leb
x : A
  ----------------------------------------------
Permutation [x] (insert x [] )

subgoal 2 is:
Permutation (x :: a :: l) (insert x (a :: l))
```

Le premier sous-but se résout aisément : D'une part, Coq est capable de *simplifier* l'expression (**insert x [] **) en **[x]**. D'autre part, la relation **Permutation** est enregistrée dans le module **Sorting.Permutation** comme une relation d'équivalence, dont nous pouvons exploiter la réflexivité.

```
reflexivity.
```

Le sous-but suivant comporte une *hypothèse de récurrence* appelée **IHl**. Cette hypothèse exprime que la propriété que nous voulons prouver est vérifiée par la liste **l** ; le but courant consiste à prouver que cette propriété est alors vérifiée par (**a : : l**), où **a** est un élément quelconque de type **A**.

```
A : Type
leb : A → A → bool
leb_total : Total leb
x, a : A
l : list A
IHl : Permutation (x :: l) (insert x l)
----------------------------------------
  Permutation (x :: a :: l) (insert x (a :: l))
```

La tactique **cbn** (pour *call by name*) procède à une évaluation symbolique de l'expression (**Permutation (x : : a : : l) (insert x (a : : l)**) et fait apparaître le motif (**if (le x a) then ... else ...**).

```
cbn.
```

```
------------------------------------------------------------
 Permutation (x :: a :: l)
     (if leb x a then x :: a :: l else a :: insert x l)
```

Le but affiché suggère une étude par cas du booléen (**leb x a**) ; que nous faisons suivre d'une tentative de résolution automatique des sous-buts triviaux issus de cette tactique. Seul le cas (**leb x a = false**) échappe à cette résolution automatique. Nous avons alors besoin d'utiliser quelques tactiques.

```
case (le x a); trivial.
cbn.
```

```
A : Type
leb : A → A → bool
leb_total : Total leb
x, a : A
l : list A
IHl : Permutation (x :: l) (insert x l)
------------------------------------------------
  Permutation (x :: a :: l) (a :: insert x l)
```

L'examen de l'hypothèse de récurrence **IHl** nous suggère d'utiliser la liste (**a : : x : : l**) comme intermédiaire. La tactique **transitivity** s'applique car **Permutation** est prouvée être une relation d'équivalence. Rappelons que cette preuve fait partie du module **Sorting.Permutation** de la bibliothèque standard.

```
transitivity (a::x::l).
```

Deux nouveaux sous-buts apparaissent :

```
2 focused subgoals
(unfocused: 0-0), subgoal 1

A : Type
leb : A → A → bool
leb_total : Total leb
x, a : A
l : list A
IHl : Permutation (x :: l) (insert x l)
  ============================
   Permutation (x :: a :: l) (a :: x :: l)

subgoal 2 is:
 Permutation (a :: x :: l) (a :: insert x l)
```

Pour résoudre le premier sous-but, cherchons si la bibliothèque standard contient déjà un résultat sur la permutation de deux éléments en tête d'une liste. Une fois prouvé ce résultat, nous l'appliquons immédiatement.

```
Search
  (Permutation (?x::?y::?l) (?y::?x::?l)).
```

```
perm_swap:
  ∀ (A : Type) (x y : A) (l : list A),
  Permutation (y :: x :: l) (x :: y :: l)
```

```
apply perm_swap.
```

Il ne nous reste plus qu'un sous-but à résoudre, qui nous paraît être une conséquence triviale de l'hypothèse de récurrence **IHl**. La tactique **auto** nous permet de finaliser la preuve.

```
A : Type
leb : A → A → bool
leb_total : Total leb
x, a : A
l : list A
IHl : Permutation (x :: l) (insert x l)
------------------------------------------------
  Permutation (a :: x :: l) (a :: insert x l)
```

```
auto.
```

```
No more subgoals.
```

La preuve de **insert_perm** est terminée. La commande suivante, abrégée de *Quod erat demonstrandum*, demande à Coq de vérifier que nous avons construit une preuve correcte et complète, et de l'enregistrer.

```
Qed.
```

```
insert_perm is defined
```

Voici terminée une première preuve en Coq. Nous avons pu observer quelques traits caractéristiques de cette activité d'ingénierie de la preuve : buts, tactiques, application de lemmes déjà prouvés, etc.

Contentons-nous de donner la suite de lemmes qui aboutit à la preuve formelle de correction de notre fonction de tri. Nous encourageons le lecteur à télécharger le fichier complet à l'adresse **À définir par l'éditeur** et à observer le dialogue entre l'outil et l'utilisateur étape par étape, en utilisant CoqIde ou Proof-General.

1.4.1 Correction de la fonction insert

Remarquons que la fonction **insert** est une fonction auxiliaire de la fonction principale **sort**. Une propriété essentielle de notre couple de fonctions est que insert n'est appelée que sur une liste triée et renvoie une liste triée comme résultat.

Il importe donc de prouver le lemme suivant.

```
Lemma insert_sorted : ∀  x l ,
  Sorted l → Sorted (insert x l).
```

1.4.2 Correction de la fonction sort

La fonction principale **sort** hérite ses propriétés de la fonction auxiliaire **insert**. Nous pouvons alors prouver la correction de **sort** de façon très concise, en appliquant les lemmes précédemment démontrés.

```
Lemma sort_perm : ∀ l, Permutation (sort l) l.
Proof.
 induction l.
 - reflexivity.
 - transitivity (a:: sort l); auto;
   symmetry; apply insert_perm.
Qed.

Lemma sort_sorted : ∀ l, Sorted (sort l).
Proof.
  induction l;auto.
  now apply insert_sorted.
Qed.

Theorem sort_correct : Sort_spec sort.
Proof.
 split.
 - apply sort_perm.
 - apply sort_sorted.
Qed.
```

1.5 Après la preuve

Rappelons-nous que nous avons défini et prouvé la fonction **sort** dans un contexte déterminé par un type quelconque A, une fonction de comparaison **leb**, et une hypothèse sur cette fonction.

À la sortie de la section **Insertion_sort** nos constructions sont automatiquement généralisées. La fonction **sort** devient une *fonctionnelle polymorphe* dont le premier argument est le type des éléments de la liste et le second la fonction de comparaison booléenne utilisée.

```
Check sort.
```

```
sort
     : ∀ A : Type, (A → A → bool) → list A → list A
```

Le théorème de correction de **sort** est conditionné par l'hypothèse que la fonction booléenne **leb** est bien associée à une relation totale. En dehors de cette hypothèse, l'utilisation de la fonction de tri **sort** ne bénéficie d'aucune garantie de correction.

```
Check sort_correct.
```

```
sort_correct
     : ∀ (A : Type) (leb : A → A → bool),
       Total leb → Sort_spec leb (sort leb)
```

La commande **Compute** *terme* permet de calculer la valeur d'un terme. Nous pouvons alors procéder à des tests simples. Le tri d'une liste d'entiers naturels se fait grâce à la fonction de comparaison **leb** du module **Nat** de la bibliothèque standard.

```
Compute sort Nat.leb [8;6;9;3;7;8].
```

```
= [3; 6; 7; 8; 8; 9]
  : list nat
```

De même, trions une liste de chaînes de caractères selon le préordre associé à leur longueur.

```
Compute
 let string_leb s s' :=
     Nat.leb (length s) (length s')
 in  sort string_leb ["cb"; "ab"; "cd"; "abcd"].
```

```
= ["cb"; "ab"; "cd"; "abcd"]
  : list string
```

Terminons cet exemple par la preuve que la fonction de tri **sort**, appliquée à une liste de chaînes de caractères, renvoie bien un résultat correct. Il suffit d'appliquer le théorème générique **sort_correct** après avoir montré que la fonction de comparaison de chaînes est bien totale. Le dialogue suivant se termine par un *certificat* de correction de cette fonction de tri.

```
Lemma  nat_leb_total :  Total Nat.leb.
Proof.
  red; induction a; destruct b;
  simpl; try discriminate; auto.
Qed.

Lemma  string_leb_total :
   Total string_leb.
Proof.
  intros s s'; apply nat_leb_ok.
Qed.

Definition  sort_correct_string :=
    sort_correct  string_leb_total.

Check sort_correct_string.
```

```
sort_correct_string
    : Sort_spec string_leb (sort string_leb)
```

1.6 Extraction de programmes

Une fois prouvée la correction d'une fonction ou d'une bibliothèque de fonctions, Coq nous permet d'en *extraire* un code pour un langage de programmation fonctionnel comme OCaml, Haskell ou Scheme. Cette opération consiste en l'effacement des preuves qui peuvent apparaître dans les fonctions écrites avec Coq, et l'adaptation à la syntaxe et la sémantique propres au langage cible de cette extraction. Nous pouvons donc obtenir des programmes fonctionnels corrects avec un très grand niveau de sûreté.

```
Extraction Language Ocaml.
Recursive Extraction sort.
```

```
type bool =
| True
| False

type 'a list =
| Nil
| Cons of 'a * 'a list

(** val insert :
 ('a1 → 'a1 → bool) → 'a1 → 'a1 list → 'a1 list **)

let rec insert leb a l = match l with
| Nil → Cons (a, Nil)
| Cons (b, l') →
  (match leb a b with
   | True → Cons (a, l)
   | False → Cons (b, (insert leb a l')))

(** val sort :
 ('a1 → 'a1 → bool) → 'a1 list → 'a1 list **)

let rec sort leb = function
| Nil → Nil
| Cons (a, l') → insert leb a (sort leb l')
```

2 Applications principales

L'exemple précédent pourrait laisser penser que Coq sert surtout à valider des programmes fonctionnels simples. Il n'en est rien. Des outils de vérification pour les langages dominants comme C, Java, Ada s'appuient totalement ou plus indirectement sur des bibliothèques écrites en Coq. Ces outils s'appuient sur la formalisation mathématique de la sémantique du langage de programmation considéré. Les théorèmes prouvés une fois pour toutes en Coq garantissent, soit la conformité de l'exécutable au source, soit des propriétés telles que l'absence d'erreurs à l'exécution, telles que le déréférencement de pointeurs nuls, ou autres erreurs arithmétiques.

Sans être exhaustifs, nous pouvons citer quelques domaines d'application de Coq.

- Preuves de protocoles cryptographiques.
- Analyse d'algorithmes distribués.
- etc.

Outre la vérification de programmes, la possibilité offerte par Coq de construire et de vérifier des démonstrations complexes s'applique à des domaines tels que l'informatique fondamentale et les mathématiques.

2.1 Utilité pour la programmation impérative

Les méthodes formelles sont utilisées pour développer du logiciel embarqué critique, pour lequel des garanties de sûreté sont exigées, en particulier par des autorités de certification. Dans ce domaine, les standards encouragent l'utilisation de méthodes formelles afin de démontrer qu'un logiciel respecte un certain niveau de confiance. Différentes activités de vérification sont effectuées : utilisation de model-checkers opérant sur des modèles issus de la conception, d'analyseurs statiques et de vérification déductive opérant sur des programmes C. Elles sont complémentaires aux activités de test et aux revues de

code manuelles effectuées sur le code exécutable produit par le compilateur.

2.1.1 CompCert

Le compilateur CompCert est le premier compilateur optimisant qui soit formellement vérifié et commercialisé [19, 18, 1]. CompCert compile des programmes écrits en langage C et génère du code assembleur (Power PC, ARM ou x86) raisonnablement efficace, grâce à un ensemble d'optimisations. Sa particularité est d'être formellement vérifié en Coq, et donc de garantir que la compilation d'un programme n'introduit pas d'erreur dans le code généré.

En effet, de façon similaire à la fonction de tri par insertion présentée dans cet article, CompCert a été programmé et prouvé correct en Coq. Le théorème de correction est une propriété de préservation sémantique, qui exprime qu'un programme compilé se comporte exactement comme spécifié par le programme C dont il est issu. CompCert étant programmé en Coq dans un style fonctionnel (comme le tri par insertion), le mécanisme d'extraction de Coq fournit automatiquement le code OCaml correspondant, qui peut donc être exécuté indépendamment de Coq. La spécification de CompCert comprend la sémantique des langages source et cible du compilateur. Ces sémantiques, ainsi que celles des langages intermédiaires de CompCert sont également définies en Coq. Une sémantique associe à chaque programme l'ensemble de ses comportements observables. Un comportement indique si un programme termine (par une exécution normale ou bien par une exécution anormale) ou bien s'exécute indéfiniment ; il fournit de plus la trace des opérations d'entrées-sorties effectuées par le programme.

CompCert accomplit une succession de 20 passes de compilation, opérant sur les 11 langages intermédiaires de CompCert. La correction de CompCert résulte de la correction de chacune de ces passes, ainsi que d'un théorème général de correction de la composition de passes successives.

Au-delà de la compilation vérifiée du langage C, CompCert fournit à la communauté de chercheurs une méthodologie et un ensemble de composants pouvant être réutilisés pour vérifier d'autres outils, qui participent à la production et à la validation de logiciels critiques. Parmi les exemples de réutilisation, figurent la logique de programmes VST [4], ainsi que l'analyseur statique Verasco opérant par interprétation abstraite sur des programmes C [16].

2.1.2 Vérification de programmes annotés

L'outil Why3 [8] est un *générateur de conditions de vérifications*. À tout programme impératif annoté par des spécifications : pré-conditions, post-conditions, invariants de boucles, invariants de types, on associe un ensemble d'énoncés à prouver, tels que, si chacun de ces énoncés est vrai, alors toutes les annotations sont vérifiées lors de chaque exécution. Dans l'état actuel de Why3, toutes ces obligations de preuves sont envoyées aux prouveurs installés sur la machine de l'utilisateur. Les plus simples seront prouvées automatiquement, les plus complexes devront faire l'objet de preuve interactive, à l'aide d'assistants de preuve, parmi lesquels figure Coq. Why3 s'utilise comme greffon pour prouver la correction de programmes écrits en C, Ada, ou Java [21, 17], ou des primitives cryptographiques [5].

La cohérence des modèles des théories axiomatiques au cœur de Why3 est établie à l'aide de Coq. On peut donc en inférer qu'un utilisateur de Why3 et de ses greffons est un double utilisateur de Coq : directement, pour prouver les conditions de vérifications difficiles, et indirectement pour accorder sa confiance aux diagnostics de correction que fournit cet outil.

2.2 Informatique fondamentale et mathématiques

La double capacité de Coq à aider son utilisateur à construire des preuves de complexité très variée, mais aussi à vérifier la correction de ces preuves comporte des applications non restreintes à la vérification de programmes. Citons en quelques exemples :

La publication de nouveaux algorithmes dans des domaines très divers : algorithmique distribuée, probabiliste, etc., peut désormais s'accompagner d'une preuve formelle de leurs propriétés, ce qui renforce leur acceptation par la communauté scientifique. La correction de ces algorithmes est rendue accessible sous la forme de développements que le lecteur peut télécharger et vérifier par une simple commande de compilation.

Dans l'enseignement des mathématiques ou de l'informatique, un sous-ensemble réduit de Coq peut s'utiliser pour demander de vraies démonstrations simples sur machine. Rappelons que Coq signale toute démonstration fausse ou incomplète, et fournit ainsi un dispositif d'auto-correction immédiate à l'étudiant ou l'élève.

Le langage de spécifications de Coq s'adapte autant aux activités de nature mathématique : logique, mathématiques discrètes et continues, étude des fondements, que de l'informatique : spécification et vérification de composants et de bibliothèques logiciels. Cette double compétence a été planifiée lors de la création du cœur de Coq il y a plus de trente ans. Les applications actuelles de cet outil montrent que le passage à l'échelle a été pleinement réussi.

3 Coq et sa place en informatique

En 2013, Coq a reçu deux prix de l'Association for Computing Machinery (ACM), le SIGPLAN « Programming Languages Software Award » et le prestigieux « Software System Award ». Le texte accompagnant ce second prix situe Coq dans le cadre de la production de logiciel correct. Le texte complet en anglais se trouve sur le site de l'ACM [3] :

> Coq, utilisé à la fois pour énoncer des théorèmes mathématiques et des spécifications logicielles, est une technologie clef pour la production de logiciel certifié.

Parmi les résultats significatifs obtenus avec Coq, on peut citer la preuve du théorème des quatre couleurs, le développement de CompCert (un compilateur entièrement vérifié pour C), le développement à Harvard d'une version vérifiée du logiciel d'isolation de fautes de Google, et, plus récemment, le noyau du système hyperviseur CertiKOS, entièrement spécifié et vérifié.

Le cœur de Coq, c'est un vérificateur de preuves pour un formalisme qui a la particularité d'être en même temps une logique de force comparable à la théorie des ensembles et un langage de programmation fonctionnel richement typé, plus expressif encore que ne le sont les langages de programmation fonctionnels standard tels que Haskell, OCaml, F#, à la seule différence que la récursion n'est pas libre : les fonctions récursives doivent terminer, au pire en les contrôlant avec une mesure du temps.

Ce qui permet à un tel formalisme d'être la fois une logique et un langage de programmation, c'est la prise de conscience par les chercheurs à partir de la fin des années 1960 que les preuves et les programmes avaient exactement la même structure. Les preuves, comme les programmes, sont des fonctions manipulant des valeurs dans des types de données. Les formules, c'est comme des types dans un langage de programmation, avec juste un peu plus de structure pour représenter les quantificateurs, ce qu'on appelle des *types dépendants*. Il y a d'ailleurs un nom pour ces systèmes qui sont à la fois des logiques et des langages de programmation : on dit que ce sont des *théories des types*.

4 Évolution

Le système Coq, dans son état actuel, est le résultat de plus de trente ans de recherches et d'attention aux besoins de ses utilisateurs. Une connaissance très légère de cette évolution permet de comprendre des caractéristiques qui pourraient paraître complexes ou étranges.

4.1 Un outil issu de la recherche fondamentale

Issu des travaux de Thierry Coquand et Gérard Huet sur le Calcul des Constructions (en anglais CoC), eux-mêmes s'appuyant sur plusieurs décennies de recherche en logique et en informatique fondamentale et appliquée, la première version du système Coq, alors appelée CONSTR, date de décembre 1984. Quelques versions expérimentales plus tard, ce sera cependant le nouveau Calcul des Constructions Inductives (en anglais CIC) de Thierry Coquand et Christine Paulin-Mohring qui servira dès 1989 de fondation. Le système CONSTR, basé sur le CoC, devient alors Coq, basé sur le CIC.

4.2 L'outil actuel

Coq vit s'ajouter au fil du temps de nombreuses extensions, motivées par des besoins concrets, étudiées par des chercheurs qui en isolent la substantifique moelle scientifique. Ce fut par exemple l'ajout, dès 1988, d'un mécanisme d'extraction automatique des preuves et des programmes Coq vers les langages compilés OCaml, Haskell et Scheme, ou l'ajout en 1999, d'une notion de type coinductif, duale de celle de type inductif et fort utile pour représenter les flux de données infinis. Sans chercher à l'exhaustivité, on pourra aussi citer pêle-mêle divers points qui font de Coq un système généraliste utilisé au quotidien : des bibliothèques, une large panoplie de méthodes de preuves programmables, que l'on appelle des tactiques, des interfaces graphiques, des mécanismes de notations et d'inférence automatique de l'information implicite, ce qui permet d'écrire des spécifications "comme sur le papier", sans parler de nombreuses extensions contribuées par les utilisateurs.

4.3 Le futur de Coq

Trente-deux ans après le premier prototype, la recherche autour de Coq est toujours aussi active, soutenue par une large communauté de logiciens et de spécialistes de la programmation. Certains mathématiciens s'intéressent à Coq, comme le médaillé Fields Vladimir Voevodsky qui utilise Coq dans le cadre d'un projet de refondation des mathématiques au-dessus de la théorie des types, en replacement de la théorie des ensembles dont elle est un raffinement typé.

Depuis plusieurs années déjà, Coq est développé dans le cadre d'un écosystème logiciel, avec le soutien d'une communauté internationale d'utilisateurs et de contributeurs d'extensions, de bibliothèques, de résultats mathématiques ou informatiques finaux tels que la fameuse formalisation du théorème des groupes d'ordre impair, ou la vérification formelle d'un compilateur du langage C. Le développement de Coq est assuré par une petite équipe de chercheurs entreprenants, accueillant aussi des contributions extérieures. Les discussions ont lieu en ligne sur la plate-forme GitHub ainsi que lors de réunions de travail bimestrielles. Le développement a successivement été coordonné par Gérard Huet, Christine Paulin et Hugo Herbelin, avec Matthieu Sozeau comme coordinateur depuis 2016.

5 Apprendre Coq

Les plus de cinq-cents pages du manuel de référence de Coq [10] ne doivent pas effrayer le futur utilisateur. Coq se doit d'être un logiciel très complet, où de nombreux outils assistent l'utilisateur dans le développement de démonstrations correctes. Quelques jours suffisent pour être capable de prouver la correction de petits programmes fonctionnels simples comme la fonction de tri donnée en exemple dans cet article. Cet apprentissage peut se faire en travaillant sur les nombreux tutoriaux ou ouvrages plus complets [7, 13, 9]. Des écoles d'été

animées par les spécialistes sont également organisées pour faciliter cette initiation, combinant cours et séances d'exercices durant quelques jours.

L'approfondissement de cet outil se fait naturellement dès que son utilisateur a un projet à mener à bien. Il est alors très utile de partager interrogations et solutions de problèmes avec les autres utilisateurs, par l'intermédiaire de la liste de diffusion `coq-club` et le wiki `Cocorico!`, accessibles depuis la page du logiciel [10]. De cette façon, l'utilisateur se constitue un catalogue de "patrons de preuve" qui, à l'instar des patrons de conception en programmation, peut aider à la résolution de problèmes parfois délicats.

6 Conclusion

Les assistants de preuve tels que Coq rendent possible la production de logiciels sûrs malgré leur complexité. Il est bien clair que tout programme ne doit pas être formellement vérifié, mais le nombre d'applications devant être sûres justifie l'émergence d'un métier que nous pouvons appeler *ingénierie de la preuve*, qui nécessite des compétences à la fois en logique et en génie logiciel.

Parmi les divers assistants de preuve, Coq se distingue par sa fiabilité, une bibliothèque standard très fournie, et un grand nombre de tactiques de preuve interactive.

Sa complexité pourrait effrayer l'utilisateur novice, mais on peut remarquer que ce n'est pas Coq qui est trop compliqué, mais la preuve rigoureuse, notamment d'algorithmes et de programmes, qui nécessite des outils suffisamment complets.

Références

[1] The CompCert compiler. http://compcert.inria.fr.

[2] OCaml page. http://www.ocaml.org/.

[3] Page of the ACM Software System Award. awards.acm.org/software_system.

[4] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program logics for certified compilers*. Cambridge University Press, 2014.

[5] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella-Béguelin. Computer-aided security proofs for the working cryptographer. In *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90. Springer, 2011.

[6] Jean-Pierre Belna. *Histoire de la logique*. Ellipses, 2014.

[7] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art : The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[8] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Let's Verify This with Why3. *Software Tools for Technology Transfer (STTT)*, 17(6) :709–727, 2015.

[9] Adam Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2011. `http://adam.chlipala.net/cpdt/`.

[10] The Coq Development Team. Coq page. http://coq.inria.fr.

[11] Agda Development Team. Agda. http://wiki.portal.chalmers.se/agda/pmwiki.php.

[12] Coq development team. Coq standard library. https://coq.inria.fr/distrib/current/stdlib/.

[13] B. Pierce et al. Software foundations. https://www.cis.upenn.edu/∼bcpierce/sf.

[14] Georges Gonthier. Formal proof — the four-color theorem. *Notices of the American Mathematical Society*, 55(11), December 2008.

[15] John Harrison. HOL-light : An overview. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 60–66, Berlin, Heidelberg, 2009. Springer-Verlag.

[16] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In *42nd ACM symposium on Principles of Programming Languages*, 2015.

[17] Nikolai Kosmatov, Claude Marché, Yannick Moy, and Julien Signoles. Static versus Dynamic Verification in Why3, Frama-C and SPARK 2014. In *7th International Symposium on Leveraging Applications*, 7th International Symposium on Leveraging Applications, page 16, Corfu, Greece, October 2016. Springer.

[18] Daniel Kästner, Xavier Leroy, Sandrine Blazy, Bernhard Schommer, Michael Schmidt, and Christian Ferdinand. Closing the gap - the formally verified optimizing compiler CompCert. In *Safety-critical Systems Symposium 2017 (SSS 2017)*, February 2017.

[19] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7) :107–115, 2009.

[20] Michel Mauny. Le langage Caml. *Techniques pour l'ingénieur*, 2017.

[21] John W. McCormick and Peter C. Chapin. *Building High Integrity Applications with SPARK*. Cambridge University Press, 2015.

[22] Robin Milner. *LCF : A way of doing proofs with a machine*, volume 74 of *Lecture Notes in Computer Science*, pages 146–159. Springer Berlin Heidelberg, 1979.

[23] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL : A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

À lire également dans nos bases

Le langage CAML de Guy Cousineau [H3018V1], dans Technologies logicielles Architectures des systèmes (1998)