Aabu

Geste de savoir

Écrire des programmes prouvés corrects avec Coq

1er août 2022

Table des matières

Introduction

Les logiciels occupent une place importante dans les objets technologiques qui nous entourent. Il est parfois impératif qu'un programme ne comporte absolument aucune erreur de programmation. On y parvient en utilisant des techniques permettant de prouver mathématiquement que le programme respecte bien sa spécification, c'est-à-dire qu'il fait ce qu'on attend de lui. Dans cet article, je dirai pour simplifier *prouvé correct* pour signifier *prouvé correct vis-à-vis de sa spécification.*[1]

Dans le présent article, nous montrerons sur un exemple à quoi ressemble l'écriture de programmes prouvés corrects avec l'assistant de preuve *Coq*. Il s'agit d'un logiciel permettant de faire des démonstrations mathématiques[2], qui a la particularité d'intégrer aussi un langage de programmation et qui permet ainsi de mêler programmation et preuves mathématiques. Il n'est pas nécessaire de connaître Coq pour suivre cet article puisque les explications permettent de suivre l'essentiel du propos.

L'exemple que nous développons dans l'article est le cœur d'un mini-compilateur. Les langages source et cible sont délibérément minimalistes pour faciliter les explications. Cet exemple, bien que simpliste, est inspiré par une des applications les plus remarquables de Coq, à savoir un

1. Il faut alors garder en tête qu'il peut rester des erreurs dans le programme, mais ce seront des erreurs de spécification et non de programmation.

compilateur vérifié pour le langage C , notamment utile pour les applications critiques où l'on veut *garantir* que la compilation n'introduit pas de bugs.

1. Un problème de compilation

Notre objectif est d'écrire un mini-compilateur et de prouver qu'il est correct. Ce compilateur a pour rôle de transformer des programmes écrits dans un langage de programmation source en des programmes écrits dans un langage de programmation cible. Pour prouver qu'il est correct, nous nous assurerons que la *signification* du programme source est bien la même que celle du programme compilé.

Par souci de simplicité, nous ne donnons pas de définition concrète sous forme de texte aux langages source et cible, ce qui évite de digresser sur l'analyse syntaxique des programmes. Ainsi, il n'y aura pas de code source à proprement parler et les programmes seront directement représentés par des structures de données.

1.1. Langage source et langage cible

Pour nos deux langages, un programme est une *liste d'instructions*, qui manipulent un entier relatif jouant le rôle de compteur. L'exécution de chaque instruction a un certain effet sur le compteur. L'exécution d'un programme consiste à exécuter chaque instruction l'une à la suite des autres, en partant d'un compteur à zéro.

Pour le langage source, les deux seules instructions sont:

— `Add n`, qui ajoute l'entier naturel n au compteur;
— `Sub n`, qui soustrait l'entier naturel n au compteur.

En utilisant la notation de Coq pour les listes (des crochets pour délimiter la liste et le point-virgule pour séparer les instructions), un programme dans le langage source est par exemple `[Add 2; Sub 3; Add 3]`. Ce programme termine son exécution avec un compteur à 2.

Le langage cible est plus simple encore, les deux seules instructions sont:

— `Incr`, qui ajoute 1 au compteur;
— `Decr`, qui soustrait 1 au compteur.

Un exemple de programme dans le langage cible est `[Incr; Incr; Decr; Decr; Incr]`, qui termine avec un compteur à 1. Un autre exemple de programme est `[Decr; Decr; Incr; Decr; Decr]`, qui termine avec un compteur à -3.

2. Pour un aperçu de l'utilisation de Coq, voir l'article Un zeste de mathématiques assistées par ordinateur .

1.2. Compilation

Compiler le langage source vers le langage cible, c'est générer un programme dans le langage cible qui a le même comportement qu'un programme donné dans le langage source. Il faut comprendre que la manière de procéder ici est totalement libre, tant que le comportement du programme compilé est le même que celui du programme source.

Nous dirons que le comportement de deux programmes est le même si on aboutit à la même valeur finale pour le compteur. Cette définition est assez simple, mais suffisante pour les besoins de l'article. Notez qu'il existe des définitions plus fines, souvent nécessaires pour les langages de programmation de la vraie vie.

Une manière simple de compiler en ayant bien le même comportement au départ et à l'arrivée, est de traduire directement chaque instruction:

— l'instruction `Add n` est transformée en n fois l'instruction `Incr`;
— l'instruction `Sub n` est transformée en n fois l'instruction `Decr`.

Par exemple, le programme `[Add 2; Sub 3]` sera compilé en `[Incr; Incr; Decr; Decr; Decr]`. En exécutant mentalement ces programmes, on voit que le programme source aboutit sur un compteur à -1, tout comme le programme cible et donc que la compilation semble correcte.

On pourrait aussi imaginer un compilateur optimisant, qui ferait des simplifications, pour par exemple compiler `[Add 2; Sub 3]` directement en `[Decr]`, mais on ne développera pas plus cette idée.

Il faut maintenant effectivement écrire un compilateur pour prouver ensuite qu'il est effectivement correct.

2. Écriture du cœur du compilateur

Coq nous permet d'écrire le compilateur et de le prouver correct dans un seul outil. Dans cette section, nous commentons des extraits de programme Coq au fur et à mesure, en omettant quelques détails. Le code complet et fonctionnel est listé à la fin de l'article.

2.1. Définitions de types pour les langages source et cible

Dans la section précédente, nous avons mentionné que les programmes seront directement représentés par des structures de données, à savoir une liste d'instructions. Coq est un langage fortement typé, nous définissons donc des types pour les instructions, ainsi que des types pour les listes d'instructions.

On commence par définir le langage source. Les quelques lignes ci-dessous définissent d'abord un type `instr_s` disposant de *constructeurs* pour chaque instruction de ce langage. Ensuite, on définit une notation (*i.e.* un raccourci d'écriture) qui dit simplement qu'un programme est une liste d'instructions, ce qui permet d'utiliser les listes définies dans la bibliothèque standard de Coq. Le suffixe `_s` désignera toujours le langage source dans la suite.

```
Inductive instr_s :=
  | Add (n:nat)
  | Sub (n:nat).

Notation prog_s := (list instr_s).
```

On fait de même pour le langage cible. Le suffixe `_t` désignera le langage cible dans la suite.

```
Inductive instr_t :=
  | Incr
  | Decr.

Notation prog_t := (list instr_t).
```

On peut demander d'ailleurs demander à Coq de vérifier le type des exemples de la section précédente. Pour l'exemple de programme source, on entre la commande suivante pour demander à Coq de vérifier le type de l'expression:

```
Check [Add 2; Sub 3; Add 3].
```

Coq est un logiciel interactif, il nous répond en redonnant l'expression et son type. (Dans la suite, les réponses de Coq à une commande seront toujours sur fond sombre comme ci-dessous.)

```
[Add 2; Sub 3; Add 3]
     : prog_s
```

De même pour l'exemple d'un programme cible, on peut vérifier que Coq déduit le bon type.

```
Check [Incr; Incr; Decr; Decr; Incr].
```

```
[Incr; Incr; Decr; Decr; Incr]
     : prog_t
```

2.2. La fonction de compilation

On a désormais ce qu'il faut pour écrire le compilateur. Il s'agit d'une fonction récursive nommée `compile`, qui fait appel à la fonction `compile_instr` pour la compilation d'une instruction individuelle. On applique directement le principe de compilation présenté dans la section précédente.

```
(* Compilation d'une unique instruction, fonction normale *)
Definition compile_instr (i:instr_s) : prog_t :=
  match i with (* on regarde le type de l'instruction *)
  (* Cas `Add` : on renvoie une liste avec Incr répété n fois *)
  | Add n => repeat Incr n
  (* Cas `Sub` : on renvoie une liste avec Decr répété n fois *)
  | Sub n => repeat Decr n
  end.

(* Compilation d'un programme *)
Fixpoint compile (prog:prog_s) : prog_t :=
  (* On regarde la forme du programme *)
  match prog with
  (* Programme vide, rien à faire *)
  | nil => nil
  (* On compile la première instruction et on met ensuite au bout
     le résultat de la compilation du reste du programme *)
  | i::prog0 => (compile_instr i) ++ (compile prog0)
  end.
```

On peut tester le bon fonctionnement de la fonction sur un exemple. On utilise `Compute` pour que Coq calcule la valeur d'une expression.

```
Compute compile [Add 2; Sub 3].
```

```
     = [Incr; Incr; Decr; Decr; Decr]
     : prog_t
```

La fonction semble fonctionner correctement, mais un exemple ne vaut pas une preuve!

3. Signification des programmes

Avant de prouver que la fonction de compilation est correcte, il est essentiel de définir ce que *signifient* nos programmes. Sans ça, il est impossible d'exprimer dans Coq le fait que la fonction

de compilation doit conserver la signification des programmes.[1]

Dans la première section, nous avons dit que l'exécution d'un programme correspond à l'évolution du compteur au fil des instructions. Il faut écrire ça formellement dans Coq.

On définit d'abord une notation `counter` qui est simplement un synonyme pour le type des entiers relatifs `Z`.

```
1 Notation counter := Z.
```

3.1. Signification d'un programme source ou cible

Maintenant qu'on dispose d'un type compteur, il faut écrire des fonctions qui traduisent les instructions en leur effet sur le compteur.

On a traduit littéralement la signification des instructions telles que décrites dans la première section de cet article. L'exécution d'une instruction consiste à prendre un état, ajouter ou soustraire un entier selon l'instruction puis renvoyer le nouvel état correspondant.

La fonction qui réalise l'exécution d'un programme source est une fonction récursive qui fait appel à une autre fonction pour traiter chaque instruction. On pourrait tout faire dans une seule fonction, mais l'utilisation d'une fonction auxiliaire simplifie légèrement l'écriture de la preuve que nous ferons après.

```
1  (* Exécution d'une instruction *)
2  Definition exec_instr_s (i:instr_s) : counter :=
3    (* Analyse de cas sur le type d'instruction *)
4    match i with
5    (* Première instruction, on ajoute n au compteur *)
6    | Add n => Z.of_nat n (* Z.of_nat fait la conversion entre
       entiers naturels et relatifs *)
7    (* Deuxième instruction, on soustrait n au compteur *)
8    | Sub n => - Z.of_nat n
9    end.
10
11 (* Exécution d'un programme *)
12 Fixpoint exec_s (prog:prog_s) : counter :=
13   (* Analyse de cas sur un programme *)
14   match prog with
15   (* programme vide, le compteur est nul *)
16   | nil => 0
17   (* on exécute l'instruction, puis le reste du programme *)
18   | i::prog0 => exec_instr_s i + exec_s prog0
19   end.
```

1. Nous utilisons le mot *signification* par souci de clarté, mais dans le jargon, on parle de *sémantique*, qui est synonyme.

On a presque exactement la même chose pour le programme cible.

```
Definition exec_instr_t (i:instr_t) : counter :=
  match i with
  | Incr => 1
  | Decr => -1
  end.

Fixpoint exec_t (prog:prog_t) : counter :=
  match prog with
  | nil => 0
  | i::prog0 => exec_instr_t i + exec_t prog0
  end.
```

3.2. Test sur des exemples

Encore une fois, on peut prouver des exemples pour tester que ça fait bien ce qui est prévu.

```
Compute exec_s [Add 2; Sub 3].
```

```
     = -1
     : counter
```

```
Compute exec_t [Incr; Incr; Decr; Decr; Decr].
```

```
     = -1
     : counter
```

Il n'est pas possible de prouver que les fonctions `exec_s` et `exec_t` définies ci-dessus sont mathématiquement correctes. Il s'agit de *spécifications*; on définit tout simplement ce que l'on veut que les programmes signifient. *Les écrire correctement est donc absolument primordial.*
Même avec toute la rigueur de Coq, il peut rester des *bugs de spec*: ce qui est formellement spécifié n'est pas vraiment ce qu'on voulait signifier initialement. On peut parer ce problème en testant intensivement la spécification avec des techniques habituelles de génie logiciel.

4. Preuve de correction de la compilation

Comme nous l'avons évoqué plusieurs fois dans l'article, prouver que la fonction de compilation est correcte, c'est prouver que le programme source et le programme cible associé ont la même signification.

Dans Coq, nous avons choisi de l'exprimer comme suit:

```
Theorem compile_correct:
  forall (prog:prog_s), exec_s prog = exec_t (compile prog).
```

Il faut le comprendre comme suit: *«Quelque que soit le programme source, l'exécution de celui-ci aboutit au même compteur que l'exécution du programme issu de sa compilation.»* Cela sera vrai pour *tous* les programmes sources possibles, qui sont en nombre infini, et qu'on ne saurait tester exhaustivement.

Bien que ce soit possible, il n'est pas forcément très lisible de démontrer ce théorème de but en blanc. Il est plus clair de démontrer auparavant des propriétés utiles pour la démonstration du théorème principal. Ces propriétés utiles ne se devinent pas forcément *a priori*, mais s'identifient au fur et à mesure de la démonstration principale, quand on voit les éléments importants de la démonstration.

4.1. Propriétés utiles

La première propriété énonce qu'exécuter deux programmes bout à bout, c'est comme additionner le compteur à la fin de l'exécution du premier avec celui à la fin du deuxième. Son utilité vient du fait que la fonction `compile` contient l'opérateur `++` pour concaténer deux listes, et que cette propriété permet de transférer cette opération au monde des entiers, c'est-à-dire au domaine de la signification des programmes.

```
Lemma exec_t_app:
  forall (prog1 prog2:prog_t), exec_t (prog1 ++ prog2) = exec_t
      prog1 + exec_t prog2.
```

La deuxième propriété sera vraiment le cœur de la démonstration du théorème de compilation, puisqu'elle met en relation la fonction de compilation, l'exécution d'une instruction source et celle de sa version compilée. Elle énonce qu'une instruction source et le résultat de sa compilation ont bien la même signification.

```
Lemma compile_instr_correct:
  forall (i:instr_s), exec_instr_s i = exec_t (compile_instr i).
```

Nous passons les démonstrations de ces propriétés pour commenter en détail celle du théorème principal. Sachez juste qu'à chaque fois, la clé de la preuve repose sur les propriétés des entiers relatifs qui permettent de montrer l'équivalence entre l'exécution des programmes. Le code de cet article est listé à la fin de celui-ci.

4.2. Démonstration du théorème de correction

Coq est un prouveur interactif. On construit la preuve au fur et à mesure en saisissant des tactiques auxquelles Coq répond en indiquant les hypothèses courantes, suivies de ce qu'il reste à démontrer.

On énonce le théorème et on démarre la preuve.

```
1 Theorem compile_correct:
2   forall (prog:prog_s), exec_s prog = exec_t (compile prog).
3 Proof.
```

Coq nous répond en disant qu'on a un sous-objectif, l'énoncé du théorème lui-même. Les hypothèses sont au-dessus de la barre (il n'y en a aucune présentement) et ce qu'il faut démontrer est en dessous.

```
1 1 subgoal
2 ____________________________________(1/1)
3 forall prog : prog_s, exec_s prog = exec_t (compile prog)
```

Une technique habituelle quand on travaille avec des listes est de procéder par récurrence. Dans Coq, on utilise pour cela la tactique `induction` en lui indiquant sur quel objet on souhaite procéder par récurrence.

```
1 induction prog.
```

Coq remplace alors le sous-objectif actuel par deux sous-objectifs. Cela signifie que pour prouver notre objectif initial, il suffit de prouver deux propriétés, à savoir l'initialisation et la propriété de récurrence. En interne, Coq a gardé en mémoire le lien logique entre ces deux propriétés et le théorème initial et saura reconstruire la preuve complète à la fin.

```
1 2 subgoals
2 ____________________________________(1/2)
3 exec_s [] = exec_t (compile [])
4 ____________________________________(2/2)
5 exec_s (a :: prog) = exec_t (compile (a :: prog))
```

La première partie est résolue avec `trivial`, puisqu'il s'agit de quelque chose que Coq peut simplifier et ensuite constater sans aide.

```
1 trivial.
```

La deuxième partie est plus complexe. Coq nous présente la propriété à prouver. On constate aussi la présence de la propriété `IHprog`, l'hypothèse de récurrence.

```
1 1 subgoal
2 a : instr_s
3 prog : prog_s
4 IHprog : exec_s prog = exec_t (compile prog)
5 ______________________________________(1/1)
6 exec_s (a :: prog) = exec_t (compile (a :: prog))
```

Il est possible de simplifier beaucoup, car on a une entrée de la forme `a::prog` qui correspondra à une seule branche du `match` dans les différentes fonctions en présence. On effectue cela avec la tactique `simpl` qui sait faire tout un tas de simplifications:

```
1 simpl.
```

L'objectif devient:

```
1 1 subgoal
2 a : instr_s
3 prog : prog_s
4 IHprog : exec_s prog = exec_t (compile prog)
5 ______________________________________(1/1)
6 exec_instr_s a + exec_s prog = exec_t (compile_instr a ++ compile
      prog)
```

On est embêté par la concaténation de listes, mais on peut l'éliminer grâce à la première propriété utile démontrée auparavant. On veut remplacer le terme de la forme `exec_t (_ + _)` par l'autre membre de l'égalité.

L'expression `exec_instr_s a` peut également être remplacée grâce à la deuxième propriété.

On effectue ces remplacements en utilisant une tactique de réécriture:

```
1 rewrite exec_t_app.
2 rewrite compile_instr_correct.
```

On arrive alors à ce stade-là:

```
1 subgoal
a : instr_s
prog : prog_s
IHprog : exec_s prog = exec_t (compile prog)
______________________________________(1/1)
exec_s prog + exec_t (compile_instr a) = exec_t (compile_instr a)
    + exec_t (compile prog)
```

C'est le moment d'utiliser l'hypothèse de récurrence pour se débarrasser de `exec_s prog`.

```
rewrite IHprog.
```

Il suffit alors de prouver:

```
1 subgoal
a : instr_s
prog : prog_s
IHprog : exec_s prog = exec_t (compile prog)
______________________________________(1/1)
exec_t (compile_instr a) + exec_t (compile prog) = exec_t
    (compile_instr a) + exec_t (compile prog)
```

On voit que les deux termes sont égaux, mais Coq n'est pas assez malin pour conclure seul, il faut lui dire. Il existe une tactique nommée `reflexivity` permettant de prouver un objectif sous forme d'égalité entre deux termes ayant la même forme.

```
reflexivity.
```

Coq nous dit alors qu'il ne reste plus rien à prouver.

```
No more subgoals.
```

On clôt la preuve.

```
Qed.
```

C'est fini. Notre fonction de compilation est désormais prouvée correcte vis-à-vis de sa spécification!

Conclusion

Voilà! Vous avez désormais un aperçu de comment on écrit des programmes prouvés corrects avec Coq, sous réserve que la spécification elle-même corresponde bien au besoin. Cet exemple est tout simple, mais ce genre de techniques est également utilisé sur des projets de grande envergure. En particulier, le compilateur CompCert est essentiellement écrit et prouvé avec Coq, puis extrait automatiquement vers OCaml et bénéficie ainsi d'un bon niveau de performances et de très bonnes garanties sur sa correction.

Coq n'est pas le seul outil dans le domaine des preuves de programmes. Il existe de nombreux autres outils sur le même créneau, avec des capacités diverses et parfois complémentaires. On peut citer par exemple Isabelle, Frama-C et le greffon WP, Why3 et bien d'autres. On note d'ailleurs l'excellence de la recherche française dans ce domaine puisque Coq et Frama-C en sont issus!

Si l'expression «prouvé correct mathématiquement» donne une impression d'infaillibilité, gardez en tête qu'il reste des maillons faibles dans la réalisation des programmes, tels que de potentiels bugs dans la spécification, dans Coq lui-même, dans le compilateur d'OCaml, voire dans le processeur chargé d'exécuter le programme. L'enjeu est tel que de nombreux chercheurs et ingénieurs travaillent à la réalisation de chaînes qui soient complètement vérifiées d'un bout à l'autre, afin d'avoir les garanties élevées qui sont de plus en plus recherchées pour les applications critiques.

Vous trouverez le code intégral de l'article caché ci-dessous, écrit pour Coq 8.13.

Contenu masqué n°1

Miniature du tutoriel: le logo de Coq, tel que distribué avec le logiciel.

Contenu masqué

Contenu masqué n°1

```
(* Require and Imports *)

Require Import List.
Import ListNotations.

Require Import ZArith.
Open Scope Z_scope.
```

Contenu masqué

```
8
9  Require Import Extraction.
10
11
12 (* Source language *)
13
14 Inductive instr_s :=
15   | Add (n:nat)
16   | Sub (n:nat).
17
18 Notation prog_s := (list instr_s).
19
20
21 (* Target language *)
22
23 Inductive instr_t :=
24   | Incr
25   | Decr.
26
27 Notation prog_t := (list instr_t).
28
29
30 (* Compile *)
31
32 Definition compile_instr (i:instr_s) : prog_t :=
33   match i with
34   | Add n => repeat Incr n
35   | Sub n => repeat Decr n
36   end.
37
38 Fixpoint compile (prog:prog_s) : prog_t :=
39   match prog with
40   | nil => nil
41   | i::prog0 => (compile_instr i) ++ (compile prog0)
42   end.
43
44 Compute compile [Add 2; Sub 3].
45
46
47 (* Semantics *)
48
49 Notation counter := Z.
50
51 Definition exec_instr_s (i:instr_s): counter :=
52   match i with
53   | Add n => Z.of_nat n
54   | Sub n => -Z.of_nat n
55   end.
56
57 Fixpoint exec_s (prog:prog_s) : counter :=
```

Contenu masqué

```
58    match prog with
59    | nil => 0
60    | i::prog0 => exec_instr_s i + exec_s prog0
61    end.
62
63  Compute exec_s [Add 2; Sub 3].
64
65  Definition exec_instr_t (i:instr_t) : counter :=
66    match i with
67    | Incr => 1
68    | Decr => -1
69    end.
70
71  Fixpoint exec_t (prog:prog_t) : counter :=
72    match prog with
73    | nil => 0
74    | i::prog0 => exec_instr_t i + exec_t prog0
75    end.
76
77  Compute exec_t [Incr; Incr; Decr; Decr; Decr].
78
79
80  (* Proof of correctness *)
81
82  Lemma exec_t_app:
83    forall (prog1 prog2:prog_t), exec_t (prog1 ++ prog2) = exec_t
          prog1 + exec_t prog2.
84  Proof.
85    induction prog1.
86    - trivial.
87    - intros ; simpl ; destruct a ; rewrite IHprog1 ; ring.
88  Qed.
89
90  Lemma compile_instr_correct:
91    forall (i:instr_s), exec_instr_s i = exec_t (compile_instr i).
92  Proof.
93    (* La démonstration ci-dessous fonctionne, mais est un peu
          brouillon. *)
94    destruct i ; induction n ;
95    trivial ; simpl in IHn ;  unfold compile_instr ; simpl repeat ;
96    unfold exec_t ; fold exec_t ;  rewrite <- IHn ; unfold
          exec_instr_s ;
97    unfold exec_instr_t ; rewrite Nat2Z.inj_succ ; ring.
98  Qed.
99
100 Theorem compile_correct:
101   forall (prog:prog_s), exec_s prog = exec_t (compile prog).
102 Proof.
103   induction prog.
104   - trivial.
```

Contenu masqué

```
105   - simpl.
106     rewrite exec_t_app.
107     rewrite compile_instr_correct.
108     rewrite IHprog.
109     reflexivity.
110 Qed.
```

Retourner au texte.