Prouvez que vos programmes fonctionnels n'ont pas de bugs avec Coq Première partie

Yves Bertot

Prouvez que vos programmes fonctionnels n'ont pas de bugs avec Coq
Première partie

Yves Bertot

25 janvier 2023

CV. de l'auteur

Yves Bertot est chercheur à l'Inria depuis 1992. Il s'est spécialisé dans les preuves en theorie des types, étudiant successivement les propriétés des langages de programmation, la géométrie algorithmique, les flux infinis coinductifs, les calculs mathématiques et les interaction entre preuve formelle et calcul formel algébrique. Il a co-écrit avec Pierre Castéran le premier livre sur le système Coq qui a été publié en 2004. Yves Bertot a participé à certains des résultats les plus remarquables obtenus avec le système Coq, comme le compilateur Compcert et la preuve vérifiée sur ordinateur du théorème de l'ordre impair.

Lien vers une photo : `https://www-sop.inria.fr/members/Yves.Bertot/Yves.Bertot-2021.jpg`.

Résumé

Dans cet article, nous donnons un exemple très simple de programmation fonctionnelle et nous montrons comment ce style de programmation se prête à des raisonnements logiques pour éviter les erreurs de programmation. Les raisonnements logiques peuvent eux-même être effectués avec le système Coq. Cet article est également une introduction à l'utilisation de Coq, en utilisant une application bancaire simplifiée comme illustration.

Une introduction à la programmation fonctionnelle

La programmation fonctionnelle est un style de programmation où les regroupements d'opérations sont essentiellement représentés par des fonctions. Ces fonctions retournent un résultat à partir d'un certain nombre d'arguments, sans modifier de variable où de "mémoire ambiante".

Ainsi une fonction d'addition et une fonction de multiplication prennent chacune en entrée deux nombres et retournent un nombre, lui-même susceptible d'être utilisé par une une opération plus tard. Une telle fonction à deux arguments sera appelée une fonction binaire. L'appel d'une telle fonction ne modifie pas le

comportement des autres fonctions et un nouvel appel avec les mêmes arguments retournera la même valeur. Bien sûr, ce genre de fonction existe aussi dans les langages de programmation conventionnels non fonctionnels.

Ce qui distingue les langages fonctionnels, c'est que les valeurs échangées par des fonctions peuvent également être des fonctions. Ainsi, une fonction F pourra prendre une fonction binaire en argument et trois nombres et appliquer répétitivement la fonction binaire sur les arguments pour les utiliser tous. Une telle fonction qui prend des fonctions en argument est appelée une fonction d'ordre supérieur.

On pourra par la suite appliquer F à la fonction d'addition pour obtenir une fonction qui calcule la somme de trois nombres ou à la fonction de multiplication pour obtenir une fonction qui calcule leur produit.

Parmi les langages fonctionnels, tous utilisent cette possibilité de passer des fonctions arguments. Ils peuvent ensuite se différencier suivant qu'ils adhèrent à une discipline de programmation fonctionnelle avec effets de bord ou de programmation fonctionnelle pure. Pour la programmation fonctionnelle avec effets de bord, on inclut également dans le langage de programmation des primitives permettant de modifier des variables existantes ou un état global. Le langage OCaml est un exemple de langage de programmation avec effets de bord. Dans la programmation fonctionnelle pure, ces primitives sont inexistantes. Haskell est un exemple de langage de programmation fonctionnelle pure. L'avantage de la programmation fonctionnelle pure est que l'on sait dès la définition d'une variable et pour toute sa durée de vie quelle est la valeur de cette variable.

La programmation fonctionnelle pure facilite le raisonnement sur les programmes. Si l'on établit à un moment donné la valeur d'une fonction pour une certaine entrée, cette valeur est établie une fois pour toutes. En programmation avec effets de bord, toute propriété faisant référence à des variables du programme a une validité limitée dans le temps.

Les calculs que l'on peut faire en programmation fonctionnelle pure sont les mêmes que ceux que l'on peut faire en programmation avec effets de bord, mais l'efficacité est souvent moins bonne. Une opération de mise-à-jour dans un tableau a un coût constant dans un langage à effet de bord, mais dans un programme fonctionnel pur, ce coût est en moyenne logarithmique vis-à-vis de la taille du tableau. Pour cette raison, la programmation fonctionnelle pure est parfois évitée dans le développement de programmes efficaces.

Les cas d'usage de la programmation fonctionnelle sont très variés et touchent souvent des domaines où le coût des erreurs serait catastrophique : applications bancaires, blockchains, trading à haute fréquence. Dans ces domaines, un avantage souvent cité est la grande lisibilité par des experts du domaine d'application (financiers, traders) tout en permettant une programmation très sûre. D'autres exemples d'application concernent des outils de programmation comme des compilateurs ou des analyseurs pour d'autres langages de programmation, comme le langage C (compilateur CompCert) ou des outils de preuve et de vérification de programmes, comme le langage Coq ou FramaC. Pour ces applications, c'est la facilité à programmer sans erreur avec de nouveaux types de données qui est souvent appréciée.

Lorsque l'on s'intéresse à la correction des programmes, la programmation fonctionnelle pure est un outil puissant. D'une part, il existe des programmes pour lesquels l'efficacité des programmes fonctionnels pur suffit. Mais même lorsque le programme étudié est un programme à effets de bords, il est possible d'utiliser un programme fonctionnel pur comme étape intermédiaire pour vérifier que le programme étudié est correct. Dans une première étape, on montre que les calculs effectués dans le programme étudié sont correctement modélisés par le programme fonctionnel (même si les deux programmes ne prennent pas le même temps d'exécution). Dans un second temps, on montre que le programme fonctionnel pur satisfait les propriétés désirées. Cette approche permet de vérifier par exemple qu'une relation est bien satisfaite entre l'état initial et l'état final du programme étudié, ou bien qu'une certaine propriété est satisfaite dans l'état final.

Dans cette première partie, nous ne montrerons pas comment établir une correspondance entre programmes à effets de bords et programmes en programmation fonctionnelle pure. Nous nous concentrons sur la preuve de propriétés pour ces derniers.

Prouver des propriétés de programmes fonctionnels purs avec Coq

La raison d'être de Coq est de prouver que certains programmes satisfont des propriétés. Ces propriétés peuvent alors servir d'informations pour vérifier que plusieurs bibliothèques logicielles sont utilisées de façon cohérente.

Le premier niveau de cohérence est fourni par le typage : toutes les valeurs manipulées dans les programmes Coq ont un type et toutes les fonctions prenant des arguments indiquent quel est le type attendu. Vérifier que chaque fonction reçoit en entrée des valeurs du bon type est l'une des premières étapes pour éviter les erreurs de programmation.

Les différents langages fonctionnels typés se distinguent par le pouvoir expressif des types présents dans ce langage. Souvent ces types donnent des informations basiques : être un nombre entier, une chaine de caractères, ou bien une valeur de vérité. Dans Coq, le type peut contenir des informations logiques ou mathématiques beaucoup plus avancées. Par exemple, on peut y définir le type des nombres compris entre 0 et 100, le type des nombres premiers, ou le type des listes triées de nombres.

Comme illustration, nous allons imaginer un cas d'usage où l'on développe une application financière associant à une collection d'invidus (identifiés par une chaine de caractères) des sommes d'argent. Dans un premier temps, nous supposerons l'existence d'une bibliothèque fournissant une base de données avec seulement deux opérations : consulter la base de données pour un identifiant et fabriquer une nouvelle base de données à partir d'une base de données existante, où seule la valeur associée à un identifiant est modifiée.

Démarrer une session Coq

Le système Coq est disponible à l'adresse suivante : `https ://coq.inria.fr`, il est possible de télécharger des programmes binaires pour installer sur votre ordinateur, mais il est également possible de faire tourner une instance de ce programme dans votre navigateur web, grâce aux capacités des navigateurs moderne à faire tourner des logiciels écrits en javascript ou web assembly.

Avant de démarrer notre expérience, nous devons demander à Coq de charger quelques bibliothèques qui fourniront des structures de données basiques.

```
Require Import String List ZArith Bool.

Open Scope Z_scope.
```

La première ligne est utilisée pour indiquer que nous allons utiliser des bibliothèques existantes de coq pour manipuler des chaines de caractères, des listes, des valeurs entières, et des valeurs booléennes.

La deuxième ligne indique que les opération arithmétiques feront référence à des opérations entre nombres entiers par défaut.

Comment transférer une somme d'argent d'un compte vers un autre ?

Pour transférer de l'argent d'un compte vers un autre, nous voulons effectuer les opérations suivantes :

1. Trouver la somme d'argent courante du compte émetteur.
2. Retirer de cette somme d'argent la somme transférée.
3. Trouver la somme d'argent courante du compte receveur.
4. Ajouter à cette somme la somme transférée.
5. Modifier les deux comptes pour qu'ils enregistrent les nouvelles sommes.

Le diable est dans les détails, suivant l'ordre dans lequel ces opérations sont effectuées il est possible que le programme représente un risque de perte d'argent pour l'émetteur ou pour l'organisme gestionnaire des comptes.

Travailler avec des programmes inconnus

L'une des particularités de Coq est de permettre d'écrire des programmes et de raisonner sur ces programmes même si certaines parties ne sont pas encore définies.

Dans notre exemple, nous voulons commencer à écrire notre programme de transfert, même si nous ne disposons pas des programmes pour manipuler la base de données. Nous le faisons de la façon suivante :

```
Section abstract_program.
```

```
Variable accounts : Type.

Variable empty : accounts.

Variable update : string -> Z -> accounts -> accounts.

Variable get : string -> accounts -> Z.
```

La première ligne indique que nous entrons dans un espace où nous allons *supposer l'existence* de nouveaux objets.

Dans la deuxième ligne, nous supposons qu'il existe un type `accounts` pour les bases de données associant des clés qui sont des chaines de caractères et des valeurs qui sont des entiers. Puis nous supposons qu'il existe une base de données vide `empty`, et qu'il existe deux fonctions, la fonction `update` pour construire une nouvelle base de données à partir d'une base existante en modifiant seulement la valeur pour une clef, et la fonction `get` pour connaitre la données associée à une clef dans une base de données.

Lorsque l'on suppose l'existence d'une fonction, on doit donner son type. Ici, nous indiquons que la fonction `update` a le type suivant :

```
string -> Z -> accounts -> accounts
```

Ceci exprime que cette fonction prend trois arguments, le premier doit être de type `string` (un type pour représenter les chaines de caractères), le second argument argument doit être de type `Z` (un type pour représenter les entiers relatifs non bornés), et le troisième doit être de type `accounts`. Le deuxième mot `accounts` qui apparait sur cette ligne indique le type de la valeur retournée. Prise dans son ensemble, cette ligne exprime que la valeur retournée par la fonction `update` appliquée à trois arguments est de type `accounts`.

Avec ces premières lignes, rien n'indique quel doit être le comportement des fonctions `update` et `get`. En fait, nous n'avons pas besoin de savoir grand chose sur ces fonctions. En premier, nous avons besoin de savoir qu'interroger une base de données obtenue par la fonction `update` k v m pour la même clef k retourne la valeur v. Nous exprimons ceci par l'hypothèse suivante, qui s'écrit également dans le langage de Coq :

```
Hypothesis get_after_update_same_key :
  forall m k v, get k (update k v m) = v.
```

Ces deux lignes montrent que l'on utilise le texte "`update k v m`" pour représenter l'appel de la fonction `update` avec les trois arguments `k`, `v`, `m`. Le texte

```
  get k (update k v m)
```

est donc l'application de la fonction `get` à deux arguments, le deuxième étant le résultat de l'appel de la fonction `update` aux trois arguments `k`, `v`, et `m`.

Nous avons aussi besoin d'exprimer que les autres valeurs ne sont pas modifiées par la fonction `update`. En d'autres termes, si nous interrogeons une base de

données obtenue par la fonction `update` k v m pour une clef différente, nous obtenons la même valeur que pour la base de données m. Nous exprimons ceci par l'hypothèse suivante :

```
Hypothesis get_after_update_different_key :
  forall m k1 k2 v, k1 <> k2 -> get k1 (update k2 v m) = get k1 m.
```

Dans les énoncés de ces deux hypothèses, nous utilisons le mot-clef `forall` pour exprimer que la propriété logique est satisfaite pour tous les choix possibles de `m` `k`, ou `v` pour le premier énoncé, et pour tous les choix possible de `m`, `k1`, `k2`, et `v` pour le second énoncé. La notation `k1 <> k2` est utilisée pour représenter l'énoncé logique "`k1` est différent de `k2`" et la flèche est utilisée pour représenter l'implication.

Dans ce contexte abstrait où nous ne savons pas comment les fonctions `update` et `get` sont codées, nous pouvons quand même décrire l'algorithme de transfert. Les deux hypothèses nous servent à documenter le comportement attendu pour ces deux fonctions.

Nous pouvons décrire l'opération de transfert de la façon suivante :

```
Definition abstract_transfer
  (key1 key2 : string) (amount : Z) (m : accounts) :=
  let key1_balance := get key1 m in
  let intermediate := update key1 (key1_balance - amount) m in
  let key2_balance := get key2 intermediate in
  update key2 (key2_balance  + amount) intermediate.
```

La première lignes indique que nous définissons une fonction appelée `abstract_transfer`, la deuxième ligne indique que cette fonction reçoit quatre arguments : les deux premiers sont des clefs que nous appellerons `key1` et `key2`, de type `string`, le troisième argument est un entier et le quatrième argument est une base de données `m`. Dans cette définition, nous n'indiquons pas quel est le type de retour, mais cette information pourra être déduite de la forme du code : le type de retour est le même que celui de la fonction `update` lorsque cette dernière est appliquée à trois arguments, `accounts`.

Les différentes étapes que nous avons décrites plus tôt dans un texte en français sont rendues explicites dans le code : en ligne 3 on récupère la valeur associée à la première clef dans la base de données, en ligne 4 on fait la soustraction du montant transféré et on fabrique une nouvelle base de données associant cette nouvelle valeur à la première clef, en ligne 5, on récupère la valeur associée à la deuxième clef dans la base de données et en ligne 6 on fait l'addition du montant transféré et la mise à jour pour produire la base de données finale.

Tester notre programme pour un cas simple

Parce que nous disposons d'hypothèses sur les fonctions `get` et `update`, nous pouvons Faire des tests symboliques de notre programme.

Par exemple, nous pouvons imaginer un cas d'usage où Alice a 10 pièces de monnaies dans son compte, ceci est décrit par un état initial que nous appelons

accounts1. Dans une première transaction, Alice donne 2 pièces à Bernard, et nous obtenons un nouvel état que nous appelons accounts2, plus tard Bernard donne 3 pièces de monnaies à Alice, et ceci donne encore un nouvel état que nous appelons accounts3. Ces différents états peuvent être décrits de la manière suivante :

```
Variable accounts1 : accounts.
Hypothesis balance_1_Alice : get "Alice" accounts1 = 10.

Let accounts2 := abstract_transfer "Alice" "Bernard" 2 accounts1.
Let accounts3 := abstract_transfer "Bernard" "Alice" 3 accounts2.
```

Maintenant, nous pouvons utiliser Coq pour vérifier que le solde pour Alice dans le dernier état est de 11.

```
Lemma balance_3_Alice : get "Alice" accounts3 = 11.
Proof.
unfold accounts3, accounts2, abstract_transfer.
rewrite get_after_update_same_key.
rewrite get_after_update_different_key.
rewrite get_after_update_different_key.
rewrite get_after_update_same_key.
rewrite balance_1_Alice.
all:easy.
Qed.
```

La première ligne décrit le test que nous voudrions vérifier. La deuxième ligne contient un mot-clef indiquant que nous voyons ce test comme une preuve que nous allons faire pas-à-pas. La troisième ligne indique à Coq d'expanser la définition de accounts3, accounts2, et abstract_transfer. Les autres fonctions et valeurs ne peuvent pas être expansées, parce qu'elles sont en fait inconnues, mais nous pouvons utiliser les hypothèses faites jusqu'à maintenant pour continuer notre test.

Lorsque nous envoyons ces commandes au système Coq, il nous répond en donnant un but, dans lequel l'état en cours du calcul apparait. Ici, nous voyons une égalité dont le membre droit est 11 et dont le membre gauche est une expression composée d'appels à get et update. Nous pouvons réduire cette expression avec les 5 lignes suivantes. A la fin, il ne reste que trois faits à vérifier. Le premier fait est une égalité entre nombres entiers, les deux autres faits sont le même fait dupliqué : il s'agit de vérifier que les deux chaines de caractères "Alice" et "Bernard" sont distinctes. Ces faits sont vérifiés par la commande easy. Après la commande easy tout a été vérifié et le test est concluant, il faut envoyer la commande Qed pour indiquer que la preuve est terminée, sinon le système ne nous laissera pas commencer une nouvelle preuve.

Nous n'avons pas besoin de savoir comment les fonctions get et update sont codées pour effectuer ce test. Le code de abstract_transfer est donc symboliquement exécutable avant d'avoir développé la librairie sur laquelle il

repose. De même, nous n'avons pas besoin de savoir combien d'argent est disponible sur le compte de Bernard.

Prouver des propriétés de notre programme

Pour être sûr que cette fonction fait ce que nous attendons, il est important de prouver quelques propriétés, qui pourront également servir de documentation pour les utiliseurs de ce code. La première propriété importante est que les valeurs associées à toutes les clefs différentes des clefs concernées restent inchangées.

La deuxième propriété est que le montant transféré est bien retranché du compte émetteur, du moins lorsque ce compte émetteur est différent du compte destinataire. La troisième propriété est que le montant transféré est bien ajouté au compte destinataire, également lorsque le compte émetteur est différent. Enfin, une quatrième propriété doit exprimer que le compte émetteur n'est pas modifié s'il est identique au compte destinataire.

Observons maintenant comment la première propriété est vérifiée à l'aide du système de preuve. La première étape est d'énoncer la propriété que nous voulons prouver :

```
 Lemma get_after_abstract_transfer_at_emitter :
  forall (key1 key2 : string) (m : accounts) (amount : Z),
  key1 <> key2 ->
   get key1 (abstract_transfer key1 key2 amount m) =
   get key1 m - amount.
Proof.
```

La première ligne indique seulement que nous voulons commencer une preuve et elle donne le nom donné à cette propriété pour un usage ultérieur. La deuxième ligne indique que nous voulons établir une propriété logique pour toutes les valeurs possibles des clefs `key1` et `key2`, toute base de données `m` et tout montant `amount`. La troisième ligne contient la partie gauche d'une implication, donc le début d'une phrase "si `key1` est différent de `key2`". Les deux dernières lignes contiennent le reste de l'implication exprimant que la valeur associée à `key1` après le transfert est modifiée de l'opposé du montant par rapport à la valeur dans `m`.

Lorsque cette commande est envoyée au système de preuve, celui-ci répond en affichant le texte suivant :

```
1 goal (ID 15)

  accounts : Type
  empty : accounts
  update : string -> Z -> accounts -> accounts
  get : string -> accounts -> Z
  get_after_update_same_key :
     forall (m : accounts) (k : string) (v : Z),
        get k (update k v m) = v
```

```
  get_after_update_different_key :
     forall (m : accounts) (k1 k2 : string) (v : Z),
        k1 <> k2 ->
        get k1 (update k2 v m) = get k1 m
  ============================
  forall (key1 key2 : string) (m : accounts) (amount : Z),
  key1 <> key2 ->
  get key1 (abstract_transfer key1 key2 amount m) =
  get key1 m - amount
```

Cette réponse se décompose en deux parties, séparée par la barre horizontale `============================`.

La partie supérieure contient les choses dont nous supposons l'existence et les propriété supposées. Cette partie supérieure est appelée le *contexte* du but. Ici, nous retrouvons les différents objets introduits depuis l'ouverture de la section et les deux hypothèses. La partie inférieure contient une formule logique que nous devons prouver. Cette partie inférieure est appelée la *conclusion* du but. A cette étape, c'est exactement l'énoncé que nous avons donné.

Cette énoncé commence par une quantification universelle sur quatre nouveaux objets. Le mode de raisonnement usuel est simplement de fixer quatre constantes pour ces quatres quantification universelle. Si nous devons montrer qu'une formule est vraie pour tous les choix possibles des quatres variables, nous pouvons choisir une valeur arbitraire pour chacune de ces variables et vérifier que nous savons montrer la formule pour ce choix arbitraire. La commande pour effectuer cette étape de raisonnement est la suivante :

```
intros key1 key2 m1 amount.
```

La réponse de Coq est la suivante :

```
1 goal (ID 19)

  accounts : Type
  empty : accounts
  update : string -> Z -> accounts -> accounts
  get : string -> accounts -> Z
  get_after_update_same_key :
     forall (m : accounts) (k : string) (v : Z),
        get k (update k v m) = v
  get_after_update_different_key :
     forall (m : accounts) (k1 k2 : string) (v : Z),
        k1 <> k2 ->
        get k1 (update k2 v m) = get k1 m
  key1, key2 : string
  m1 : accounts
  amount : Z
  ============================
```

```
key1 <> key2 ->
get key1 (abstract_transfer key1 key2 amount m1) =
get key1 m1 - amount
```

Nous voyons que trois nouvelles lignes ont été ajoutées dans le contexte, et la conclusion est devenue une formule plus simple qui ne contient plus de quantification universelle.

La formule que devons prouver maintenant est une implication. La méthode usuelle pour prouver une implication est de montrer que le membre droit de l'implication peut être prouvé si le membre gauche est satisfait. Ici pour exprimer que le membre gauche est satisfait, il suffit de l'ajouter dans le contexte, ce que nous faisons avec la commande suivante.

```
intros key1_is_not_key2.
```

Le but a maintenant la forme suivante (les premières lignes sont éludées) :

```
key1_is_not_key2 : key1 <> key2
============================
get key1 (abstract_transfer key1 key2 amount m1) =
get key1 m1 - amount
```

Nous devons maintenant montrer l'égalité entre deux valeurs numériques. Celle de gauche mentionne la fonction `abstract_transfer`. Nous allons demander au système de remplacer cette occurrence de la fonction par sa définition.

```
unfold abstract_transfer.
```

La réponse est :

```
============================
get key1
  (update key2
     (get key2
          (update key1
             (get key1 m1 - amount)
             m1)
          + amount)
     (update key1 (get key1 m1 - amount) m1)) =
get key1 m1 - amount
```

Le membre gauche de l'équation que nous voulons prouver est une expression qui commence par l'interrogation (par la fonction `get`) pour la clef `key1` d'une base de données obtenue par la mise à jour pour la clef `key2`. Le comportement dans ce cas de figure est prévu par l'hypothèse `get_after_update_different_key`.

```
rewrite get_after_update_different_key.
```

Nous avons maintenant deux buts : dans le premier le comportement prévu permet de réduire le membre gauche de l'équation. Dans le deuxième but, nous devons prouver la condition nécessaire pour que cette hypothèse soit utilisée : les deux clefs doivent effectivement être différentes. Ce deuxième but est conservé pour plus tard, nous allons continuer à travailler sur le premier but.

```
 ============================
  get key1 (update key1 (get key1 m1 - amount) m1) =
  get key1 m1 - amount

goal 2 (ID 23) is:
 key1 <> key2
```

Dans le premier but, le membre droit de l'équation décrit l'interrogation d'une base de données obtenue par mise-à-jour à la même adresse. Ce cas de figure est prévu l'hypothèse `get_after_update_same_key`.

```
rewrite get_after_update_same_key.
```

Le premier but devient donc de plus en plus simple.

```
 ============================
  get key1 m1 - amount = get key1 m1 - amount
```

Ce but est maintenant une évidence logique, parce qu'il s'agit de prouver une égalité où les deux membres sont identiques. et il ne reste plus qu'à le faire accepter par le système de preuve.

En fait le deuxième but est également une évidence, parce qu'il s'agit d'une expression qui apparait directement parmi les choses que nous avons supposées (c'est une des hypothèses du contexte). Nous pouvons donc demander au système de preuve de résoudre tous les cas restants.

```
all: easy.
```

Le système nous répond alors qu'il n'y a plus de but. Il faut faire enregistrer ce succès, ce que l'on fait en appelant la commande suivante :

```
Qed.
```

A partir de là, la propriété prouvée peut être utilisée pour des preuves à venir, par exemple en l'utilisant dans une commande `rewrite` comme nous l'avons fait pour les hypothèses `get_after_update...`

Pour compléter notre connaissance de la fonction `abstract_transfer`, nous convaincre de sa correction, et en fournir une documentation complète, nous pouvons ajouter trois autres lemmes qui auront les énoncés et les preuves suivants :

```
Lemma get_after_abstract_transfer_when_emitter_is_receiver :
  forall (key1 key2 : string) (m1 m2: accounts) (amount : Z),
  m2 = abstract_transfer key1 key2 amount m1 ->
  (key1 = key2 -> get key1 m2 = get key1 m1).
```

```
Proof.
intros key1 key2 m1 m2 amount m2_value.
intros key1_is_key2.
rewrite m2_value.
unfold abstract_transfer.
rewrite <- key1_is_key2.
rewrite get_after_update_same_key.
rewrite get_after_update_same_key.
ring.
Qed.

Lemma get_after_abstract_transfer_at_receiver :
  forall (key1 key2 : string) (m1 m2: accounts) (amount : Z),
  m2 = abstract_transfer key1 key2 amount m1 ->
  key1 <> key2 ->
  get key2 m2 = get key2 m1 + amount.
Proof.
intros key1 key2 m1 m2 amount m2_value key1_is_not_key2.
rewrite m2_value.
unfold abstract_transfer.
rewrite get_after_update_same_key.
rewrite get_after_update_different_key.
easy.
now auto.
Qed.

Lemma get_after_abstract_transfer_at_different_key :
  forall (key1 key2 : string) (m1 m2: accounts) (amount : Z),
  m2 = abstract_transfer key1 key2 amount m1 ->
  forall key, key <> key1 /\ key <> key2 ->
  get key m2 = get key m1.
Proof.
intros key1 key2 m1 m2 amount m2_value key
  [key_is_not_key1 key_is_not_key2].
rewrite m2_value.
unfold abstract_transfer.
rewrite !get_after_update_different_key; easy.
Qed.
```

Conclusion

La programmation fonctionnelle repose sur quelques ingrédients supplémentaires, notamment les types algébriques, la programmation à base de règles de filtrage, et la récursion. Pour les activités de preuves, ces ingrédients supplémentaires sont couverts par des techniques de raisonnement par cas et de raisonnement par récurrence. En mettant en œuvre ces différents ingrédients, il est possible

de définir une structure de données pour des ensembles finis de comptes, les opérations `get` et `update`, et de démontrer que ces opérations satisfont les propriétés exprimées dans les hypothèses `get_after_update_same_key` et `get_after_update_different_key`. Il est également possible de fournir plusieurs structures de données différentes, avec des caractéristiques d'éfficacité différentes. Ceci fera l'objet d'articles à venir, mais une ébauche d'implémentation est déjà visible à l'adresse suivante :

`https://gitlab.inria.fr/bertot/progcoq/`

Références

`https://coq.inria.fr`
`https://gitlab.inria.fr/bertot/progcoq/account.v`
`https://cel.archives-ouvertes.fr/inria-00001173`
`https://www-sop.inria.fr/members/Yves.Bertot/coqartF.pdf`
`https://github.com/coq-community/coq-art`