This assignment will be closed on July 03, 2023 (23:59:59).

On June 25, 2023 (16:13:27), we had submissions for the following questions: 1, 2, 3 — Details (/agns/INF371/TD10/2022/uploads/:my/)

Preuve de programmes avec Why3

Julien Signoles

Introduction : l'outil Why3

Why3 (http://why3.lri.fr) est un outil de preuve de programmes. Il permet d'écrire des programmes dans un langage de programmation appelé WhyML et de les prouver corrects vis-à-vis de leurs spécifications écrites dans un langage dédié. Pour prouver un programme, Why3 génère un ensemble d'obligations de preuve (i.e., des formules logiques) qui doivent toutes être démontrées à l'aide de prouveurs, automatiques ou interactifs.

Dans ce TP, nous vous fournirons les programmes WhyML, et vous devrez les spécifier et les prouver. La version navigateur de Why3 (http://why3.lri.fr/try/) intégrée avec le prouveur Alt-Ergo (http://alt-ergo.lri.fr/) est tout à fait suffisante pour tous les exercices, mais veillez à cliquer sur la clé à molette pour régler le paramètre "First-attempt step limit" (et éventuellement les autres paramètres avec "step limit"), puis mettez une valeur suffisamment haute (e.g. 2000). Si vous le souhaitez, vous pouvez bien sûr installer la version de Why3 en tant qu'application autonome et l'essayer avec d'autres prouveurs plus puissants (comme Z3 (https://github.com/Z3Prover)).

Exercice 1 : Première preuve de programme

1. Copier-coller le programme WhyML suivant et sauvegarder le dans un fichier `loop.mlw` .

   Vous devriez pouvoir comprendre aisément les deux fonctions `loop1` et `loop2` de ce fichier. Voici néanmoins quelques remarques:

   - Les clauses `use` (éventuellement suivies de `import` ou `export`) donnent accès à des modules préalablement définis (ici, dans la bibliothèque standard de Why3).
   - En WhyML, les variables ne sont pas modifiables par défaut. Pour ce faire, il faut créer une *référence* (mot clé `ref`). L'opérateur `!` permet de lire une référence, tandis qu'une affectation introduite par `:=` permet de la modifier.
   - Les commentaires sont écrits entre `(*` et `*)` .

```
module Loops

  use int.Int
  use ref.Ref

  let loop1 (n:int)
    (* question 4:
      requires ...
      ensures { result = 0 } *)
    =
    let i = ref n in
    while !i > 0 do
      (* question 2: variant { ... } *)
      (* question 4: invariant { ... } *)
      i := !i - 1
    done;
    !i

  let loop2 (n:int) =
    let i = ref n in
    while !i < 100 do
      (* question 2: variant { ... } *)
      i := !i + 1
    done

end
```

2. Remplacer les deux commentaires étiquetés `question 2` par des variants spécifiant des quantités entières positives strictement décroissantes garantissant la terminaison de chacune des fonctions.

3. Prouver ce programme dans l'interface de Why3. Pour cela copiez-coller le contenu de votre fichier `loop.mlw` dans la version navigateur de Why3, puis cliquez sur le bouton "Verify". Une coche verte signifie que la propriété est prouvée correctement. Le but est que toutes les propriétés soient prouvées (comme indiquée dans la digression 2 plus bas, il s'agit d'une condition nécessaire, mais non toujours suffisante).

4. On souhaite maintenant démontrer que la première fonction retourne toujours 0. C'est la signification de la clause `ensures` fournie, correspondant à une postcondition. Le mot clé `result` correspond à la valeur retournée par la fonction (ici, la valeur de `i` à la dernière ligne de la fonction).

   Pour prouver des fonctions avec des boucles, il faut écrire le(s) invariant(s) de boucle, introduit(s) par le mot clé `invariant`, correspondant à l'hypothèse d'induction dans une preuve par induction. Chaque invariant de boucle doit ainsi être vrai avant le premier tour de boucle (cas de base) et à la fin de chaque tour, dernier tour inclus (cas inductif). Il peut y avoir plusieurs invariants par boucle.

   Il peut être également nécessaire d'ajouter des préconditions, introduites par le mot-clé `requires` permettant de limiter le domaine de définition des arguments.

   Ajouter le(s) invariant(s) de boucle et les éventuelles préconditions nécessaires à la preuve de correction de la fonction `loop1` vis-à-vis de sa postcondition, puis prouver ce programme. En cas de difficulté, vous pouvez lire la digression ci-dessous…

Expected files: loop.mlw

Digression 1: pourquoi une preuve échoue-t-elle ?

Régulièrement, les prouveurs automatiques ne parviennent pas à prouver toutes les propriétés que l'on souhaiterait. Une des difficultés de la preuve de programme consiste à comprendre pourquoi. Il y a principalement quatre raisons possibles:

1. le programme est faux;
2. une spécification est fausse;
3. il manque des annotations indispensables pour effectuer la preuve, typiquement un ou plusieurs invariant(s) de boucle;
4. le prouveur automatique n'est pas assez puissant pour prouver la propriété souhaitée (prouver des formules de logique du premier ordre est en effet un problème indécidable).

Dans ce TP, vous serez très certainement soit dans le cas 2, soit dans le cas 3, car vos enseignants, personnes éminemment sympathiques, ne vous donnent que des programmes corrects pouvant être vérifiés avec les prouveurs automatiques fournis.

L'interface de Why3 met à disposition de ses utilisateurs plusieurs outils pour faciliter les preuves et mieux comprendre les propriétés à prouver.

- L'onglet `Task`, une fois une propriété sélectionnée, permet de voir le but à prouver et les hypothèses à disposition: cela peut aussi permettre de comprendre pourquoi le prouveur ne parvient pas à prouver le but choisi (par exemple, s'il est trivialement faux).
- Le bouton `split` permet de découper une formule (typiquement conjonctive) en plusieurs sous-buts, permettant ainsi d'une part d'aider les prouveurs et d'autre part de distinguer les sous-buts prouvés de ceux qui ne le sont pas.

Digression 2: pourquoi une preuve réussit-elle ?

Cette question peut paraître saugrenue. En effet, une preuve réussit parce que la propriété souhaitée a pu être démontrée. Notamment, lorsque toutes les obligations de preuves sont vérifiées, vous avez réussi à prouver que le programme satisfait bien sa spécification.

Néanmoins, il convient de se convaincre que les spécifications sont bien celles attendues. En effet, deux cas malencontreux, et de la faute de l'utilisateur, peuvent notamment survenir.

1. Vous avez prouvé une propriété trop faible, le cas extrême étant la preuve de la postcondition `true`.
2. Vous avez utilisé une précondition trop forte, le cas extrême étant la précondition `false` (ou, plus généralement, une incohérence dans vos préconditions et/ou axiomes). Lorsqu'une telle extrémité survient, toutes les obligations de preuve sont en général prouvées instantanément quelque soit leur complexité (et leur validité), ce qui doit mettre la puce à l'oreille de l'utilisateur. Pour être certain que ce n'est pas le cas, un bon test est d'essayer de prouver une postcondition invalide. Si le but est prouvé, votre contexte est incohérent (ou bien soit le prouveur, soit Why3 est incorrect, mais c'est moins probable).

Exercice 2 : Tableaux

On considère à présent une fonction prenant en entrée un tableau d'entiers et déterminant si tous ses éléments sont nuls. Le code et sa spécification sont donnés ci-après.

```
module All_zeros

  use int.Int
  use array.Array

  exception Found

  let all_zeros (a: array int)
    ensures { result <-> forall i: int. 0 <= i < length a -> a[i] = 0 }
    =
    try
      for i = 0 to length a - 1 do
        (* à compléter: invariant { ... } *)
        if a[i] <> 0 then raise Found
      done;
      true
    with Found ->
      false
    end

end
```

On peut remarquer plusieurs choses.

- Dans le programme, l'usage d'une inégalité `<>` et d'une exception `Found`. Cette dernière est levée grâce au mot-clé `raise` et rattrapée par la construction `try ... with Found -> ... end`.
- Dans la spécification, l'usage d'une formule logique universellement quantifiée (introduite par le mot-clé `forall`) et celles d'une équivalence `<->` et d'une implication `->`.

Donner le(s) invariant(s) de boucle nécessaire(s) pour prouver ce programme.

Note: Il est possible de spécifier plusieurs invariants pour une même boucle en écrivant plusieurs clauses `invariant` ou en utilisant une formule conjonctive (l'opérateur de conjonction étant noté `/\` en Why3).

Last submission: June 20, 2023 (17:40:16)

Choose | Choose one or more files... | Submit

Reuse files from previous submissions

Expected files: all_zeros.mlw

Exercice 3 : Recherche dichotomique dans un tableau trié

Nous considérons ici le problème de la recherche dichotomique dans un tableau trié (binary search) donné par le programme suivant. Étant donné un tableau `a` d'entiers, trié par ordre croissant, et une valeur `v`, on chercher à déterminer si `v` apparaît dans `a`. Le tableau étant trié, on peut procéder par dichotomie, en divisant par deux l'espace de recherche à chaque étape. Le programme suivant effectue cette recherche, les variables `l` et `u` délimitant l'espace de recherche.

```
module Binary_search

  use int.Int
  use int.ComputerDivision
  use ref.Ref
  use array.Array

  exception Break int

  let binary_search (a: array int) (v: int)
    =
    let l = ref 0 in
    let u = ref (length a - 1) in
    try
      while !l <= !u do
        let m = div (!l + !u) 2 in
        if a[m] < v then l := m + 1
        else if a[m] > v then u := m - 1
        else raise (Break m)
      done;
      -1
    with Break m ->
      m
    end

end
```

1. Montrer que ce programme n'accède pas en dehors des bornes du tableau `a`.

2. Montrer que ce programme termine.

3. Montrer la propriété de correction suivante: *si la valeur renvoyée est positive ou nulle alors c'est l'indice d'une cellule de `a` qui contient `v`.*

4. Exprimer le fait que le tableau `a` en entrée est trié par ordre croissant. *Note :* il existe plusieurs manières de spécifier qu'un tableau est trié. On préférera la forme à deux variables « pour tout `i` et tout `j`, si `i <= j` alors … » (à la forme utilisant une quantification sur une seule variable), car elle produit des obligations de preuve plus simples.

5. Montrer la propriété de complétude suivante : *si la valeur renvoyée est positive ou nulle, alors c'est un indice auquel `a` contient `v`. Si la valeur renvoyée vaut -1, alors `v` n'apparaît pas dans `a`. Ce sont les deux seuls cas possibles.*

   Note: Si l'on utilise une conjonction d'implications, on prendra soin de parenthéser correctement (l'opérateur "conjonction" a priorité sur l'opérateur "implication").

Last submission: June 20, 2023 (20:14:31)

Choose | Choose one or more files... | Submit

Reuse files from previous submissions

Expected files: bsearch.mlw

Exercice 4 : Tri par sélection

On se propose maintenant de vérifier la correction d'un algorithme de tri. Il s'agit ici d'un algorithme de tri par sélection d'un tableau d'entiers. Le code WhyML est le suivant. Le module `Swap_predicate` facilitera la lisibilité des spécifications et l'automatisation des preuves, tandis que les étiquettes `'L`, `'L1` et `'L2` seront utiles pour la toute

dernière question.

```
module Swap_predicate

  use int.Int
  use array.Array
  use export array.ArrayPermut
  use array.ArrayExchange as E

  predicate logical_swap (a1 a2: array int) (i j: int) =
    true (* à remplacer *)

  (* lemmes techniques permettant l'automatisation des preuves :
     comme les enseignants sont éminemment sympathiques, ils sont fournis. *)

  lemma swap_exchange :
    forall a1 a2: array int, i j: int.
    logical_swap a1 a2 i j -> E.exchange a1 a2 i j

  lemma swap_permut_all:
    forall a1 a2: array int, i j: int.
    logical_swap a1 a2 i j -> permut_all a1 a2

end

module Selection_sort

  use int.Int
  use ref.Ref
  use array.Array
  use Swap_predicate

  let swap (a: array int) (i j: int)
    =
    label L in
    let tmp = a[i] in
    a[i] <- a[j];
    a[j] <- tmp

  let sort (a: array int)
    =
    label L in
    for i = 0 to length a - 1 do
      let mv = ref a[i] in
      let mi = ref i in
      for j = i + 1 to length a - 1 do
        if a[j] < !mv then begin
          mi := j;
          mv := a[j]
        end
      done;
      swap a i !mi

    done

end
```

1. Définir le prédicat `logical_swap` comme une formule spécifiant que les tableaux `a1` et `a2` sont de même longueur, et que seuls les contenus de leurs cases `i` et `j` sont échangés, les autres éléments des deux tableaux étant égaux. Le module `Swap_predicate` doit apparaître comme prouvé quand vous lancez une vérification.

2. Montrer que ce programme n’accède pas en dehors des bornes du tableau `a` en ajoutant la (les) précondition(s) nécessaire(s) et suffisante(s) à la fonction `swap` ainsi que le(s) invariant(s) de boucle requis dans la fonction `sort`. Tout doit apparaître comme prouvé à ce stade.

3. À l’aide de postconditions, spécifier et prouver le comportement complet de la fonction `swap` indiquant que les contenus des cases `i` et `j` ont été échangés et que celui des autres cases demeure inchangé. *Note:* on pourra utiliser la construction `old` qui prend une expression WhyML `e` en argument et qui fait référence à la valeur de `e` avant l’exécution de la fonction.

4. On souhaite maintenant démontrer que le tableau `a` est trié après exécution de la fonction `sort`. Ajouter la postcondition et les invariants de boucle nécessaires à la preuve.

5. Pour implémenter un algorithme de tri, il convient aussi de s’assurer que le tableau résultant contient bien exactement les mêmes éléments que le tableau `a` (quoique réordonnés). Spécifier et prouver cette propriété additionnelle. Vous pourrez éventuellement utiliser la construction `e at L` faisant référence à la valeur de l’expression `e` à l’étiquette `L`.

   Indice : Why3 dispose d’un prédicat prédéfini `permut_all` prenant deux tableaux en arguments et qui est valide si et seulement s’ils sont une permutation l’un de l’autre.