Vérification de programmes OCaml fortement impératifs avec Why3

Jean-Christophe Filliâtre, Mário Pereira, Simão Melo de Sousa

Vérification de programmes OCaml fortement impératifs avec Why3

Jean-Christophe Filliâtre[1,2], Mário Pereira[1,2], and Simão Melo de Sousa[3,4]

[1] Lab. de Recherche en Informatique, Univ. Paris-Sud, CNRS, Orsay, F-91405
[2] INRIA Saclay – Île-de-France, Orsay, F-91893
[3] Universidade da Beira Interior, Portugal
[4] LISP - Lab. de Informática, Sistemas e Paralelismo, Portugal

Résumé

Cet article présente une méthodologie pour prouver des programmes OCaml fortement impératifs avec l'outil de vérification déductive Why3. Pour un programme OCaml donné, un modèle mémoire spécifique est construit et on vérifie un programme Why3 qui le manipule. Une fois la preuve terminée, on utilise la capacité de Why3 à traduire ses programmes vers le langage OCaml, tout en remplaçant les opérations sur le modèle mémoire par les opérations correspondantes sur des types mutables d'OCaml. Cette méthode est mise à l'épreuve sur plusieurs exemples manipulant des listes chaînées et des graphes mutables.

1 Introduction

Depuis la version 4.03 du compilateur OCaml [1], il est possible de déclarer des types algébriques avec des composantes mutables. Ainsi, il est possible de définir un type de listes simplement chaînées de la façon suivante :

```
type 'a cell =
  | Nil
  | Cons of { mutable content: 'a; mutable next: 'a cell }
```

Un tel type évite ainsi le recours déplaisant au type `option`, aux valeurs récursives ou, pire encore, à la fonction `Obj.magic` [2]. Avec un type comme le type `cell`, on obtient une représentation, et donc des performances, analogues à celles d'un code C ou Java, la valeur `Nil` jouant le rôle du pointeur `null`. Une différence notable, cependant, est que le système de type d'OCaml assure qu'on ne cherchera pas à accéder au champ `content` ou `next` d'une valeur `Nil`. Cette extension du langage OCaml ouvre des perspectives intéressantes en matière de programmation de structures fortement impératives. Comme il est notoire qu'on se trompe facilement en écrivant de tels programmes, nous proposons dans cet article une approche pour prouver des programmes OCaml avec des structures algébriques mutables.

Notre approche repose sur l'outil de vérification déductive Why3 [2], qui propose à la fois une logique, un langage de programmation et une bibliothèque adaptés à la preuve de programmes. Sa logique est une extension de la logique du premier ordre avec polymorphisme,

Ce travail a été en partie financé par la Fondation des Sciences et de la Technologie du Portugal (bourse FCT-SFRH/BD/99432/2014 et FCT-SFRH/BSAB/135039/2017), par le Laboratoire d'Informatique, Systèmes et Paralellisme (LISP FCT- UID/CEC/4668/2016) et par l'Agence Nationale de la Recherche (projet VOCAL ANR-15-CE25-008).

1. https://caml.inria.fr/pub/docs/manual-ocaml/extn.html#sec257
2. Ces différentes solutions sont discutées dans un article des JFLA 2014 [9] ; elles sont aujourd'hui devenues obsolètes pour la plupart du fait de cette extension du langage.

```
type 'a cell = Nil | Cons of { mutable content: 'a; mutable next: 'a cell; }

let get_next = function Nil -> assert false | Cons { next } -> next
let set_next l1 l2 = match l1 with Nil -> assert false | Cons c -> c.next <- l2

let concat l1 l2 =
  if l1 == Nil then l2 else
  begin
    let n = ref l1 in
    while not (get_next !n == Nil) do n := get_next !n done;
    set_next !n l2;
    l1
  end
```

FIGURE 1 – Concaténation en place de listes chaînées.

types algébriques, définitions récursives et prédicats inductifs [8]. Son langage de programmation, WhyML, est un sous-ensemble du langage OCaml auquel a été ajouté du code fantôme [11] et un contrôle statique des alias [10]. Sa bibliothèque propose à la fois des modules de spécification (entiers mathématiques, ensembles, etc.) et des modules de programmation (tableaux, entiers machine, etc.). L'outil Why3 extrait les conditions de vérification d'un programme WhyML à l'aide d'un calcul de plus faibles préconditions et les transmet à de nombreux démonstrateurs automatiques, dont la majorité des démonstrateurs SMT, et à des assistants de preuve interactifs si besoin. Une fois prouvé, un programme WhyML peut être traduit automatiquement en syntaxe OCaml par un mécanisme d'extraction analogue à celui de Coq.

Nous commençons par présenter notre approche dans la section 2, sur l'exemple simple de la concaténation en place de listes chaînées, puis sa validation expérimentale sur quatre études de cas plus complexes dans la section 3. Nous concluons en évoquant quelques perspectives.

2 Méthodologie

Nous illustrons notre approche avec la preuve d'un programme OCaml qui réalise la concaténation en place de deux listes chaînées. Ce programme est contenu dans la figure 1. Il est volontairement écrit dans un style très impératif, avec une boucle `while` et des références, mais pourrait tout aussi bien être écrit avec une fonction récursive. Les deux fonctions auxiliaires `get_next` et `set_next` sont là pour faciliter la manipulation du champ `next` d'une liste dont on sait qu'elle n'est pas égale à `Nil`. Notre objectif est de prouver la correction de la fonction `concat` avec Why3. En particulier, nous montrerons que les fonctions `get_next` et `set_next` ne sont jamais appelées avec `Nil`.

Modélisation des listes chaînées en Why3. Why3 n'autorise pas la définition d'un type tel que `cell`. Ceci est dû au fait que Why3 utilise un système de types avec effets pour déterminer statiquement tous les alias entre pointeurs dans le programme [10] et le type `cell` sort du cadre de cette analyse statique. La solution consiste alors à modéliser le contenu du tas, sous la forme d'un ensemble de types et d'opérations pour allouer, lire et écrire dans la mémoire. Une telle idée est déjà employée dans des outils qui utilisent Why3 comme langage intermédiaire, par exemple Frama-C [7]. Contrairement à ces outils, cependant, nous construisons ici un modèle spécifique au type `cell` et nous continuons d'utiliser les autres types de Why3 lorsque c'est

possible. Nous commençons par introduire un type non interprété `cell` et une constante `nil` :

```
type cell 'a
val constant nil : cell 'a
```

La syntaxe `val constant` indique que le symbole `nil` peut être utilisé à la fois dans les spécifications et dans les programmes. Nous modélisons ensuite le contenu du tas à l'aide du type `mem` suivant :

```
type mem 'a = {
  mutable content: cell 'a -> option 'a;
  mutable    next: cell 'a -> option (cell 'a);
} invariant { forall l. content l = None <-> next l = None }
```

Les champs `content` et `next` correspondent aux deux arguments du constructeur `Cons`. Ils sont introduits comme des fonctions du type `cell` vers des valeurs optionnelles [3]. Les valeurs pour lesquelles les champs `content` et `next` sont différents de `None` sont les valeurs allouées de la forme `Cons`. L'invariant associé au type `mem` assure que ces deux champs sont toujours `None` simultanément. Le fait de modéliser les champs `content` et `next` indépendamment l'un de l'autre est une façon efficace de traduire qu'une modification de l'un n'a pas d'incidence sur l'autre. Cette idée est due à Burstall et remonte à 1972 [3].

Pour opérer sur ce modèle mémoire, nous introduisons plusieurs fonctions : `mk_cell` pour allouer, `get_content` et `get_next` pour lire, `set_content` et `set_next` pour écrire. Par exemple, la fonction `set_next` est ainsi déclarée :

```
val set_next (ghost mem: mem 'a) (l1 l2: cell 'a)  : unit
  requires { l1 <> nil }
  requires { mem.next l1 <> None }
  writes   { mem.next }
  ensures  { mem.next = Map.set (old mem.next) l1 (Some l2) }
```

Elle reçoit la mémoire en premier argument. Cet argument est fantôme, car le modèle mémoire n'est pas destiné à apparaître au final dans le code extrait. Les deux préconditions exigent que `l1` soit une valeur allouée différente de `nil`. La postcondition décrit comment la mémoire a été affectée.

Preuve de la fonction `concat`. Nous sommes maintenant en mesure d'écrire et de vérifier une version de la fonction `concat` à l'aide de ce modèle mémoire. Le code WhyML est donné dans la figure 2. Il est identique au code OCaml de la figure 1, à l'exception des éléments de spécification et de preuve (pré- et postcondition, code fantôme, invariants et variant de boucle).

Commençons par expliquer la spécification de cette fonction. En précondition, nous exigeons que `l1` et `l2` soient des listes finies (lignes 16–17), c'est-à-dire terminées par `Nil` [4]. Il y a de multiples façons de l'exprimer. Nous choisissons ici de matérialiser tous les éléments de chacune de deux listes dans des arguments fantômes `s1` et `s2` de type `seq (cell 'a)`. Le type `cell` est défini dans la bibliothèque standard de Why3, avec des opérations comme l'accès au i-ième élément (noté $s[i]$), la longueur (notée `length`), la concaténation (notée `++`), etc. Le prédicat `is_list_from_to` (lignes 1–10) exprime que la liste partant de la cellule `from` est finie, se termine par la cellule `to` et est composée exactement des cellules contenues dans la liste `s`.

3. Le type `option` est défini dans la bibliothèque de Why3. Il est identique à celui d'OCaml, c'est-à-dire `type option 'a = None | Some 'a`.

4. En toute rigueur, il n'est pas nécessaire d'exiger que `l2` soit finie. Cependant, l'exiger nous permettra de donner une spécification simple en terme de concaténation.

```
predicate is_list_from_to (mem: mem 'a) (from : cell 'a)
                          (s: seq (cell 'a)) (until: cell 'a) =
  let n = length s in
  n = 0 /\ from = until \/
  n > 0 /\ from = s[0] /\
  (forall i. 0 <= i < n -> s[i] <> nil) /\
  (forall i. 0 <= i < n -> s[i] <> until) /\
  distinct s /\
  (forall i. 0 <= i < n - 1 -> mem.next s[i] = Some s[i+1]) /\
  mem.next s[n-1] = Some until

predicate disjoint (s1 s2: seq 'a) =
  forall i j. 0 <= i < length s1 -> 0 <= j < length s2 -> s1[i] <> s2[j]

let concat
  (ghost mem: mem 'a) (l1 l2: cell 'a) (ghost s1 s2: seq (cell 'a)) : cell 'a
  requires { is_list_from_to mem l1 s1 nil }
  requires { is_list_from_to mem l2 s2 nil }
  requires { disjoint s1 s2 }
  ensures  { is_list_from_to mem result (s1 ++ s2) nil }
= if l1 == nil then
    l2
  else begin
    let n = ref l1 in
    let ghost step = ref 0 in
    let ghost tail1 = ref s1 in
    while not (get_next mem !n == nil) do
      invariant { 0 <= !step }
      invariant { !n = s1[!step] <> nil }
      invariant { is_list_from_to mem !n !tail1 nil }
      invariant { !step + length !tail1 = length s1 }
      variant   { length !tail1 }
      n := get_next mem !n;
      incr step;
      tail1 := !tail1 [ 1 .. ]
    done;
    set_next mem !n l2;
    l1
  end
```

FIGURE 2 – Code Why3 de la fonction `concat`.

Noter que cette définition inclut le cas d'une liste vide, lorsque `from` est égal à `to` et que `s` est la séquence vide (lignes 3–4). L'intérêt d'avoir matérialisé ainsi `s1` et `s2` est que nous pouvons les réutiliser ensuite dans d'autres aspects du contrat, par exemple pour exiger que les deux listes sont disjointes (ligne 18) et pour exprimer la postcondition (ligne 19).

Pour prouver que la fonction `concat` respecte ce contrat, on introduit un certain nombre d'invariants de boucle. Ces invariants gardent trace du fait que la variable `n` avance dans la liste `l1`. Ici, on a choisi de l'exprimer à l'aide d'une variable fantôme `step` qui représente l'indice de `n` dans la liste `l1` et d'une variable fantôme `tail1` qui représente le suffixe de `s1` restant à parcourir. On prouve également la terminaison de la boucle `while` à l'aide d'un variant, à savoir la longueur de la séquence `tail1`. Une fois passé à l'outil Why3, le code de la figure 2 est prouvé entièrement automatiquement en quelques secondes.

Extraction de code OCaml. Une fois la preuve terminée, la dernière étape consiste à traduire le programme Why3 en un programme OCaml. On utilise pour cela un mécanisme d'extraction automatique fourni par Why3. Il consiste à effacer le code fantôme et les annotations logiques, à traduire les constructions de WhyML vers les constructions d'OCaml et faire correspondre les symboles de la bibliothèque standard de Why3 avec ceux de la bibliothèque d'OCaml. Ce dernier aspect est matérialisé par un fichier texte, appelé *driver*, que l'utilisateur peut compléter pour ses propres besoins. Dans notre cas, il s'agit de traduire vers OCaml le type `cell`, la constante `nil` et les trois opérations `==`, `get_next` et `set_next` de notre modèle.

```
syntax type cell        "%1 SinglyLL.cell"
syntax val  nil         "SinglyLL.Nil"
syntax val  (==)        "%1 == %2"
syntax val  get_next    "SinglyLL.get_next %1"
syntax val  set_next    "SinglyLL.set_next %1 %2"
```

Le module OCaml `SinglyLL` contient les trois premières lignes de la figure 1. Pour chaque symbole Why3, le code OCaml est donné sous la forme d'une chaîne de caractères, où `%`n sera remplacé par l'extraction du n-ième argument de ce symbole. Notons que `set_next` apparaît ici comme n'ayant plus que deux arguments, son premier argument ayant été supprimé de par son statut fantôme.

3 Études de cas

Cette section présente une validation expérimentale de notre approche, sous la forme de quatre études de cas. L'ensemble des fichiers Why3 de ces preuves peut être téléchargé à l'adresse http://why3.lri.fr/jfla-2018/.

3.1 Module `Queue`

Notre premier exemple est celui du module `Queue` de la bibliothèque standard d'OCaml. Il s'agit d'une structure de file réalisée par une liste simplement chaînée, de la manière suivante :

```
type 'a cell = Nil | Cons of { content: 'a; mutable next: 'a cell }
type 'a t = { mutable length: int; mutable first: 'a cell; mutable last: 'a cell }
```

À chaque instant, on conserve un pointeur vers le premier élément de la liste (`first`) et un autre vers le dernier (`last`). Les éléments sont retirés de la file du côté de `first` et ajoutés du côté de `last`. On conserve par ailleurs le nombre total d'éléments de la file (`length`), pour éviter d'avoir à le recalculer.

```
1  type t 'a = {
2    mutable       length   : OneTime.t;
3    mutable       first    : cell 'a;
4    mutable       last     : cell 'a;
5    mutable ghost view     : seq 'a;
6    mutable ghost list     : seq (cell 'a);
7    mutable ghost used_mem : mem 'a
8  } invariant { length.OneTime.valid }
9    invariant { length > 0 > first = list[0] /\ last = list[length  1] /\
10                              used_mem.next last = Some nil  }
11   invariant { length = 0 > first = last = nil }
12   invariant { forall i. 0 <= i < length > list[i] <> nil }
13   invariant { forall x: cell 'a. T.mem x list <> used_mem.next x <> None }
14   invariant { forall i. 0 <= i < length  1 >
15                 used_mem.next list[i] = Some list[i+1] }
16   invariant { forall i. 0 <= i < length >
17                 used_mem.content list[i] = Some view[i] }
18   invariant { length = Seq.length view = Seq.length list }
19   invariant { distinct list }
```

FIGURE 3 – Type des files.

Modélisation. Le type `cell` des listes chaînées est modélisé en Why3 d'une façon tout à fait analogue à celle présentée dans la section précédente, avec un type `mem` pour la mémoire, un type `cell` pour les (pointeurs vers des) valeurs de type `cell` et des opérations `get_next`, `set_next`, etc. La figure 3 contient la définition du type Why3 correspondant au type `Queue.t` d'OCaml. On retrouve les trois champs `length`, `first` et `last` du code OCaml, auxquels sont ajoutés trois champs fantômes contenant respectivement la séquence des pointeurs composants la liste chaînées (`list`), la séquence des valeurs contenues dans ces éléments (`view`) et la portion de la mémoire associée à la file (`used_mem`).

Notons, tout d'abord, que le champ `length` est modélisé par un type `OneTime` (ligne 2) d'entiers machines qui obéissent à des restrictions fortes pour éviter tout débordement arithmétique [5]. Les deux champs `list` et `view` dupliquent par convenance des informations déjà présentes dans `used_mem`. En effet, il suffit d'extraire les informations contenues dans la mémoire à partir du pointeur `first` pour retrouver l'intégralité des listes `view` et `list`. Des invariants sont là pour exprimer la cohérence entre ces trois champs. D'une manière générale, notre type `t` est équipé d'un certain nombre d'invariants. Ces invariants sont supposés vérifiés à l'entrée de toute fonction manipulant une valeur de type `t` et doivent être établis à la sortie de toute fonction modifiant ou renvoyant une valeur de type `t`. Ils se décomposent ainsi :

- l'invariant (ligne 8) indique la validité de l'entier représentant la longeur étant données les contraintes imposées aux éléments du type `OneTime` ;
- dans le cas d'une file non vide, l'adresse mémoire contenue dans le champ `first` (resp. `last`) est bien la première adresse contenue dans la séquence `list` (resp. la dernière) et le dernier élément de la file (pointé par `last`) termine la séquence (son champ `next` est `nil`) (lignes 9 – 10) ;
- dans le cas d'une file vide, les champs `first` et `last` sont tous deux `nil` (ligne 11) ;
- toute adresse répertoriée dans `list` correspond à une adresse vers une cellule mémoire allouée (et vice-versa). De même, à toute adresse dans `list` correspond une cellule de la

```
1  let transfer (q1 q2: t 'a) : unit
2    requires { disjoint_mem q1.used_mem q2.used_mem }
3    writes   {q1,q2}
4    ensures  { q2.view == (old q2.view) ++ (old q1.view) }
5    ensures  { q1.view == empty }
6  = if not (is_empty q1) then
7      if is_empty q2 then begin
8        q2.length, q1.length <- q1.length, OneTime.zero ();
9        q2.first, q2.last <- q1.first, q1.last;
10       q2.list <- q1.list;
11       q2.view <- q1.view;
12       q2.used_mem, q1.used_mem <- q1.used_mem, empty_memory;
13       q1.first, q1.last, q1.list, q1.view <- nil, nil, Seq.empty, Seq.empty;
14     end else begin
15       let len = OneTime.add q2.length q1.length in
16       q2.length, q1.length <- len, OneTime.zero ();
17       set_next q2.used_mem q2.last q1.first;
18       q2.last, q2.list, q2.view <-
19         q1.last, q2.list ++ q1.list, q2.view ++ q1.view;
20       q2.used_mem, q1.used_mem <-
21         mem_union q2.used_mem q1.used_mem, empty_memory;
22       q1.first, q1.last, q1.list, q1.view <- nil, nil, Seq.empty, Seq.empty
23     end
```

FIGURE 4 – Code Why3 de la fonction `transfer`.

file dont le champ `next` contient l'adresse vers une cellule allouée ou alors l'adresse `nil` (ligne 12 – 13) ;
— le contenu des champs `list` et `view` est cohérent avec la file en mémoire (`used_mem`) et ont même taille (lignes 14 – 18) ;
— enfin, toutes les adresses impliquées dans la liste chaînée sont distinctes deux à deux. En particulier, ceci implique qu'il n'y a pas de cycle dans la liste (ligne 19).

Opérations sur les files. Le type des files étant donné, nous pouvons maintenant définir et prouver les opérations attendues sur les files. Nous illustrons ici la preuve de l'opération `transfer`, qui reçoit deux files q_1 et q_2 en paramètre, concatène tous les éléments de q_1 à la fin de q_2, puis vide q_1. Son code spécifié et annoté est donné figure 4. La précondition (ligne 2) exige que les deux files soient distinctes en mémoire. Ainsi, il n'est pas autorisé d'appeler `transfer` en lui passant deux fois la même file en argument. Le contrat ligne 3 indique que les seules valeures modifiées para la fonction seront celle de q_1 et q_2. La postcondition (lignes 4 et 5) exprime la modification des contenus des deux files.

La fonction `transfer` n'a strictement rien à faire lorsque q_1 est vide (ligne 5). Sinon, elle distingue le cas où q_2 est vide (lignes 6–12) du cas général (lignes 14–21). Dans ce dernier cas, elle fait pointer le champ `next` du dernier élément de q_2 vers le premier de q_1 et réalise l'union des modèles mémoire de q_1 et q_2. La précondition de disjonction a ici son importance si on veut pouvoir prouver les invariants de type de la file une fois modifiée. Le champ `length` est mis à jour en faisant la somme des deux longueurs (ligne 15). Cela a pour effet d'annuler la validité de ces deux longueurs, mais l'une est remplacée par la somme (qui est valide) et l'autre part zéro

(qui est valide également). Ceci est expliqué en détail dans l'article décrivant l'intérêt du module `OneTime` au regard des débordements de capacité arithmétique [5]. La preuve de `transfer` est réalisée entièrement automatiquement par un ensemble de plusieurs démonstrateurs de type SMT. Une fois le code Why3 extrait vers OCaml, en utilisant la même technique que celle présentée dans la section 2, on obtient un code très proche de celui de la bibliothèque standard d'OCaml, avec des performances en tout point identiques.

3.2 Tri fusion

Notre second exemple est un tri fusion en place de listes simplement chaînées. Nous reprenons à cet effet le modèle mémoire qui nous est maintenant familier et nous introduisons les raccourcis suivants :

```
type list 'a = cell 'a
type view 'a = seq (cell 'a)
predicate is_list (mem: mem 'a) (from: cell 'a) (s: view 'a) =
  is_list_from_to mem from s nil
```

Pour *encadrer* les effets des différentes fonctions, nous introduisons les prédicats `frame_mem` et `frame` suivants qui expriment que deux mémoires coïncident sur les adresses d'une vue donnée ou alors sur toute les vues distinctes d'une vue donnée.

```
predicate frame_mem (m1 m2: mem 'a) (s: view 'a) =
  forall i. 0 <= i < length s ->
    m1.next s[i] = m2.next s[i] /\ m1.content s[i] = m2.content s[i]
predicate frame (m1 m2: mem 'a) (s: view 'a) =
  forall ss: view 'a. disjoint s ss -> frame_mem m1 m2 ss
```

Notre preuve s'appuie sur un certain nombre de lemmes concernant ces deux prédicats, tels que `forall m1 m2 s l. is_list m1 l s -> frame_mem m1 m2 s -> is_list m2 l s`. Ces lemmes peuvent être consultés dans les fichiers en téléchargement à http://why3.lri.fr/jfla-2018/.

La première fonction auxiliaire du tri fusion est une fonction `split` qui coupe une liste en deux moitiés égales (si la liste est de longueur impaire, nous faisons le choix de couper après l'élément central). Voici sa spécification :

```
let split (ghost mem: mem 'a) (l1: list 'a) (ghost s: view 'a) :
 (ghost view 'a, list 'a, ghost view 'a)
 requires { is_list mem l1 s }
 requires { length s >= 2 }
 returns  { s1, l2, s2 ->
            is_list mem l1 s1 /\ is_list mem l2 s2 /\ s == s1 ++ s2 /\
            (length s1 = length s2 \/ length s1 = length s2 + 1) /\
            frame (old mem) mem s }
```

Les préconditions requièrent que l'adresse `l1` pointe vers une liste compatible avec la vue `s` dans le modèle mémoire `mem`. Elles requièrent également que celle-ci ait au moins deux éléments. La postcondition exprime que la liste a bien été coupée et que la valeur renvoyée par `split` est bien la tête de la seconde moitié. La valeur renvoyée, `l2`, est ici accompagnée de deux valeurs fantômes `s1` et `s2`, qui sont les modèles des deux listes obtenues par le découpage. On note l'utilisation du prédicat `frame` a la dernière ligne pour exprimer que celles les adresses de `s` ont pu être modifiées par cette fonction.

La fonction `split` peut être définie de plusieurs façons [5]. Sachant quelle est la longueur `len` de la liste chaînée en entrée, nous pouvons, par exemple, calculer l'indice `len_div` de la cellule en milieu de liste. Dès lors, il suffit de parcourir la liste jusqu'à cet indice, tout en construisant les vues `s1` et `s2`. Si `i` est l'indice de la cellule actuellement visitée, la boucle `while` est spécifiée de la façon suivante :

```
while Peano.lt !i len_div do
   variant   { len - !i }
   invariant { s == !s1 ++ !s2 }
   invariant { length !s1 = !i <= len_div }
   invariant { length !s2 > 0 }
   invariant { !n <> nil }
   invariant { disjoint !s1 !s2 }
   invariant { s[!i ..] == !s2 }
   invariant { is_list mem !n !s2 }
   invariant { is_list_from_to mem l1 !s1 !n }
```

De façon succincte, l'invariant de cette boucle assure que `s1` et `s2` sont deux vues qui forment une coupure correcte de `s`, que la taille de `s1` est égale à l'indice `i` et que les vues `s1` et `s2` forment une liste dans la mémoire. Notons l'utilisation de `is_list_from_to mem l1 !s1 !n` qui indique que la structure de liste est garantie en prenant compte du cas spécial de la dernière cellule (celle à l'adresse `n`, qui n'est pas `nil`). La preuve de correction de la fonction `split` est automatique.

La fonction auxiliaire centrale au tri fusion est la fonction `merge`, dont la spécification est la suivante :

```
let merge (ghost mem: mem 'a) (cmp: cmp 'a)
          (l1 l2: list 'a) (ghost s1 s2: view 'a) : (list 'a, ghost view 'a)
  requires { is_list mem l1 s1 /\ sorted mem cmp s1 }
  requires { is_list mem l2 s2 /\ sorted mem cmp s2 }
  requires { is_pre_order cmp }
  requires { disjoint s1 s2 }
  returns  { l, s -> is_list mem l s /\ frame (old mem) mem s /\
                     permut_all (s1 ++ s2) s /\ sorted mem cmp s }
```

Elle reçoit en argument, outre la mémoire de travail, une fonction de comparaison `cmp` et deux listes `l1` et `l2` à fusionner, supposées triées pour `cmp`. La valeur renvoyée est la tête de la liste résultant de la fusion (qui sera de fait soit la tête de `l1`, soit celle de `l2`).

Enfin, le tri est réalisé par une fonction principale qui appelle `split` sur les listes ayant au moins deux éléments, trie les deux moitiés récursivement et fusionne les résultats avec la fonction `merge`. La preuve ne pose aucune difficulté.

3.3 Parcours en profondeur

Le troisième exemple est celui d'un programme qui effectue un parcours en profondeur sur un graphe mutable. Pour la preuve et mise en œuvre en Why3, nous commençons par introduire le type `loc` pour les sommets du graphe, ainsi que la constante `null` :

```
type loc
```

5. Dans le cadre de ce module, nous avons une définition et une spécification d'une version alternative à celle que nous présentons dans cet article.

```
val constant null: loc
```

Nous choisissons ici une structure de graphe où sont associés à chaque sommet deux successeurs et une marque booléenne. Cette dernière est utilisée pour marquer les sommets déjà visités pendant le parcours. Notre modèle mémoire est ici matérialisé par trois références globales :

```
val m     : ref (map loc bool)
val left  : ref (map loc loc)
val right : ref (map loc loc)
```

Pour opérer sur ce modèle de mémoire, nous introduisons plusieurs fonctions de lecture et d'écriture, à savoir `get_m`, `get_left`, `get_right`, `set_m`, `set_left` et `set_right`. Pour des besoins de la preuve, nous ajoutons à ce modèle un quatrième élément pour garder trace de tous les sommets déjà trouvés par l'algorithme mais pas encore marqués :

```
val ghost busy: ref (map loc bool)
val ghost set_busy (m: ref (map loc bool)) (l: loc) (b: bool) : unit
```

Dans la littérature, ces sommets sont désignés comme des sommets *gris*.

La notion de chemin entre deux sommets joue un rôle crucial dans la spécification d'un parcours en profondeur. Nous la définissons de la manière suivante :

```
predicate edge (left right: map loc loc) (x y: loc) =
  x <> null /\ (left x = y \/ right x = y)

inductive path (left right: map loc loc) (x y: loc) =
| path_nil : forall x: loc, l r: map loc loc. path l r x x
| path_cons: forall x y z: loc, l r: map loc loc.
    path l r x y -> edge l r y z -> path l r x z
```

Le prédicat `path left right x y` signifie qu'il y a chemin entre les sommets `x` et `y`, les valeurs des successeurs gauches et droits étant déterminées par les dictionnaires `left` et `right`. Nous sommes maintenant en position d'implémenter et spécifier une fonction `dfs` qui réalise le parcours en profondeur. Le profil de cette fonction est le suivant :

```
let rec dfs (c: loc) (ghost root: loc) : unit
```

L'argument `c` représente le sommet courant et l'argument fantôme `root` au sommet racine d'où est parti le parcours en profondeur. La précondition de la fonction `dfs` est divisée en trois parties : (i) tous les sommets dans `busy` sont bien coloriés

```
  requires { well_colored !left !right !m !busy }
```

c'est-à-dire, pour tout arc $x \to y$ avec y différent de `null`, soit x a un statut `busy`, soit s'il est marqué alors y est aussi marqué ; (ii) seuls des sommets atteignables à partir de la racine sont marqués

```
  requires { only_descendants_are_marked root !left !right !m }
```

(iii) enfin, nous exigeons qu'il existe un chemin entre la racine et le sommet `c`

```
  requires { path !left !right root c }
```

Par manque de place, nous ne montrons pas ici les définitions des prédicats utilisés dans la spécification de la fonction `dfs`. Le lecteur intéressé peut consulter l'appendice en ligne de cet article. Quant à la postcondition de cette fonction, les propriétés `well_colored` et `only_descendants_are_marked` doivent être préservées après l'exécution du parcours

```
  ensures  { well_colored !left !right !m !busy }
  ensures  { only_descendants_are_marked root !left !right !m }
```

Ce sont donc des propriétés invariantes de l'exécution récursive de la fonction `dfs`. À la fin de l'exécution de `dfs` nous devons aussi montrer que si un sommet était déjà marqué avant l'exécution de `dfs`, alors il reste marqué après le parcours

```
  ensures  { forall x: loc. (old !m)[x] = True -> !m[x] = True }
```

et que si un sommet a un statut `busy` à la fin, c'est parce que son statut était `busy` avant le parcours :

```
  ensures  { forall x: loc. !busy[x] = True -> (old !busy)[x] = True }
```

La postcondition s'achève en disant que si `c` est différent de `null`, alors il sera bien marqué

```
  ensures  { c <> null -> !m[c] = True }
```

Notons que si jamais `root` pointe vers la racine d'un graphe infini, cette fonction ne terminera pas. En Why3, on marque une fonction comme potentiellement divergente en ajoutant à sa spécification la clause `diverges`.

Le code de `dfs` est tout à fait classique : si `c` n'est pas `null` et n'est pas marqué, le programme le marque et s'appelle récursivement sur les deux successeurs de `c`. Pour bien respecter la postcondition de `dfs`, nous prenons soin d'affecter à `c` un statut `busy` avant les appels récursifs sur les successeurs et de l'effacer ensuite. Nous terminons cet exemple par une fonction `mark` suivante, qui encapsule le parcours en profondeur à partir d'une racine donnée :

```
let mark (root: loc) : unit
  requires { forall x: loc. x <> null ->
             !m[x] = False /\ !busy[x] = False }
  ensures  { only_descendants_are_marked root !left !right !m }
  ensures  { all_descendants_are_marked root !left !right !m }
  ensures  { forall x: loc. x <> null -> !busy[x] = False }
  diverges
= dfs root root
```

Cette fonction exige qu'au début de l'exécution aucun sommet ne soit marqué (ni même avec un statut `busy`). Au final, seuls des sommets atteignables à partir de la racine sont marqués (correction) et tous les sommets atteignables sont marqués (complétude). Par ailleurs, il ne subsiste aucun sommet avec un statut `busy`. Toutes les conditions de vérifications générées pour les fonctions `dfs` et `mark`, ainsi que pour quelques lemmes auxiliaires, sont automatiquement prouvées en utilisant une combinaison des démonstrateurs Atl-Ergo, CVC4 et Z3.

3.4 Algorithme de Schorr-Waite

Le dernier exemple que nous présentons est celui de l'algorithme de Schorr-Waite [18]. Il s'agit là encore d'un algorithme de marquage des nœuds d'un graphe, mais cette fois sans utiliser de mémoire auxiliaire. Il effectue un parcours en profondeur en utilisant la structure du graphe elle-même comme une pile, la modifiant et la restaurant au fur et à mesure du parcours. Nous prouvons ici une version de l'algorithme de Schorr-Waite où chaque sommet possède exactement deux pointeurs fils. Nous pouvons ainsi réutiliser le modèle mémoire de la section précédente, en ajoutant une quatrième composante :

```
val c: ref (map loc bool)
val get_c (p: loc) : bool
val set_c (p: loc) (v: bool) : unit
```

Ce champ `c` est utilisé pendant le parcours pour savoir lequel des deux fils est en train d'être visité. Pour les besoins de la preuve, nous ajoutons encore un autre élément à notre modèle mémoire, à savoir un dictionnaire qui associe à chaque nœud le chemin du graphe de la racine jusqu'à ce nœud :

```
val ghost path_from_root : ref (map loc (list loc))
val ghost get_path_from_root (p : loc) : list loc
val ghost set_path_from_root (p: loc) (l : list loc) : unit
```

Pour raisonner sur les chemins trouvés pendant l'algorithme, nous adaptons le prédicat `path` présenté précédemment en lui ajoutant la liste des sommets intermédiaires :

```
inductive path (left right: map loc loc) (x y: loc) (p: list loc) = ...
```

La liste `p` garde trace des sommets du chemin entre `x` et `y`, en excluant `y`. Passons à la définition et spécification de la fonction `mark` qui implémente l'algorithme de Schorr-Waite :

```
let mark (root: loc) (ghost graph: set loc) : unit
```

Le profil de cette fonction est identique à celui du parcours en profondeur, si ce n'est l'argument fantôme `graph`. Cet argument supplémentaire représente l'ensemble des sommets du graphe à parcourir. Pour spécifier la fonction `mark`, nous commençons par exiger que la racine soit différente de `null` et qu'elle appartienne au modèle :

```
  requires { root <> null /\ S.mem root graph }
```

Ensuite, le modèle doit être clos par rapport aux fils de chaque nœud :

```
  requires { forall n. S.mem n graph -> n <> null /\
             (!left[n] = null  \/ S.mem !left[n] graph) /\
             (!right[n] = null \/ S.mem !right[n] graph) }
```

La dernière précondition établit qu'avant l'exécution de la fonction `mark` aucun nœud ne doit être marqué ni ne doit être signalé comme étant en cours de traitement :

```
  requires { forall x. S.mem x graph -> not !m[x] /\ not !c[x] }
```

La propriété de correction fonctionnelle de cet algorithme est introduite en ajoutant des post-conditions à la fonction `mark`. Commençons par la structure des pointeurs du graphe. À la fin de l'exécution, la structure initiale du graphe doit être rétablie :

```
  ensures  { forall n : loc. S.mem n graph ->
               (old !left)[n] = !left[n] /\ (old !right)[n] = !right[n] }
```

Quant aux valeurs des marques dans les nœuds du graphe, seuls les nœuds atteignables sont marqués (correction)

```
  ensures  { forall n : loc. S.mem n graph -> !m[n] ->
               reachable (old !left) (old !right) root n }
```

et tous les nœuds atteignables à partir de la racine sont marqués (complétude)

```
  ensures  { !m[root] }
  ensures  { forall n ch. S.mem n graph -> !m[n] ->
               edge n ch !left !right -> ch <> null -> !m[ch] }
```

Le code de `mark` commence par affecter les variables utilisées pendant le parcours, ainsi que des variables fantômes utiles pour la preuve de cette implémentation :

```
= let t = ref root in let p = ref null in
  let ghost s_nodes = ref empty in let ghost pth = ref Nil in
  ghost set_path_from_root !t !pth;
  let ghost um_nodes = ref graph in let ghost c_nodes = ref graph in
```

Les variables `s_nodes` et `pth` représentent, respectivement, la pile des sommets du parcours et le chemin entre la racine et le sommet courant. Même si l'algorithme lui-même n'utilise pas explicitement une pile de sommets pour le parcours en profondeur, elle nous est utile pour les besoins de la preuve. Quant aux variables `um_nodes` et `c_nodes`, elles représentent respectivement les ensembles des sommets non encore marqués et des sommets dont le champ `c` reste à faux. Le cœur de l'algorithme lui-même est écrit comme une boucle qui s'exécute tant que la variable `p` n'est pas `null` ou que la variable `t` n'est ni `null` ni marquée :

```
while !p <> null || (!t <> null && not get_m !t) do
  invariant { ... }
  variant   { cardinal !um_nodes, cardinal !c_nodes, length !s_nodes }
  if !t == null || get_m !t then begin
    if get_c !p then begin (* pop *) ... end
    else begin (* swing *) ... end
  end else begin (* push *) ... end done
```

Trois cas de figure sont distingués à l'intérieur du code : le premier, `pop`, correspond au cas où les deux successeurs d'un nœud ont été traités et le parcours doit alors remonter au nœud précédent (on fait ici une opération `pop` sur la pile `s_nodes`) ; le deuxième, `swing`, correspond à un parcours terminé pour le premier fils qui dont se poursuivre avec le second fils ; le troisième, `push`, correspond au cas où un nouveau nœud vient d'être découvert (on fait ici une opération `push` sur la pile `s_nodes`). L'invariant de cette boucle est long (environ 65 lignes de code) et assez complexe ; pour cette raison, il est omis ici. La terminaison de cette fonction est justifiée par un triplet ordonné lexicographiquement. Les conditions de vérification engendrées par Why3 vérifient que, à chaque tour de boucle, la valeur du triplet décroît et que ces composantes entières sont toujours positives ou nulles.

L'utilisation d'un modèle mémoire explicite, allié à la spécificité de l'invariant de la boucle du parcours, conduit à un grand nombre de conditions de vérification, dont certaines ne peuvent être prouvées immédiatement par les démonstrateurs automatiques. On parvient néanmoins à une preuve complètement automatique en ajoutant quelques lemmes auxiliaires et quelques assertions dans le code.

4 Travaux connexes

L'idée d'utiliser un modèle mémoire explicite pour vérifier des programmes impératifs n'est pas nouvelle. Le greffon *Jessie* de l'analyseur Frama-C, par exemple, traduit un programme C et sa spécification écrite en ACSL [1] vers le langage de Why3. À la différence de notre approche, *Jessie* construit un modèle mémoire généraliste, indépendamment du programme à vérifier. Un ancêtre de *Jessie*, l'outil Caduceus [12], avait été utilisé pour faire une preuve formelle de l'algorithme de Schorr-Waite [13], dont nous nous sommes fortement inspirés pour faire la preuve présentée dans la section 3.4.

Une alternative pour prouver des programmes manipulant le tas consiste à utiliser des outils possédant leur propre modèle mémoire interne. À cet égard, nous pouvons citer Dafny [15], VeriFast [14] ou encore VCC [6]. Dans ces outils, la modélisation et la manipulation de la mémoire restent invisibles pour l'utilisateur, contrairement à notre approche.

La vérification de programmes avec pointeurs est devenue populaire avec l'introduction de la logique de séparation [17]. Cette logique de programme propose un formalisme simple pour raisonner sur des structures de données mutables, en étendant la logique de Hoare classique avec de nouveaux constructeurs logiques pour établir que différentes formules sont vérifiées par des zones disjointes de la mémoire. Ces dernières années, plusieurs outils utilisant la logique de séparation ont été développés et le formalisme a gagné en maturité. Récemment, Charguéraud et Pottier ont démontré comment la logique de séparation peut être utilisée non seulement pour vérifier des propriétés fonctionnelles d'un code, mais aussi des propriétés sur son temps d'exécution [4].

5 Conclusion et perspectives

Nous avons présenté dans ce papier une méthode pour implémenter, spécifier et vérifier des programmes OCaml fortement impératifs à l'aide de l'outil de preuve Why3. Par fortement impératifs, nous entendons des programmes qui effectuent des modification complexes du tas et introduisent par conséquent des alias arbitraires entre pointeurs. Ces programmes étant hors de portée du système contrôle statique des alias de Why3, notre méthode s'appuie sur la construction d'un modèle mémoire spécifique à chaque exemple que nous voulons prouver.

Nous avons utilisé cette méthode avec succès sur des nombreux exemples non triviaux, dont certains sont présentés dans ce papier. Il reste néanmoins difficile en pratique d'utiliser et de raisonner sur un modèle mémoire explicite en Why3. D'une part, le nombre de conditions de vérification générées explose rapidement, ce qui peut rendre la preuve difficile à gérer. D'autre part, des preuves plus complexes, comme celles de *mergesort* ou de l'algorithme de Schorr-Waite, exigent toujours un grand effort, soit en terme de temps de travail, soit en terme d'expertise. Un meilleur support de la part de Why3 est clairement souhaitable. Nous envisageons la construction d'une bibliothèque de logique séparation en Why3 à fin de rendre plus maniable la preuve de programmes qui manipulent le tas, à la manière de ce qui est fait dans l'outil KIV [16]. En pratique, nous avons déjà rencontré certains éléments classiques de la logique de séparation, tels que le prédicat `frame` utilisé dans la preuve de *mergesort*.

Les modèles mémoire et les *drivers* sont, pour l'instant, écrits à la main pour chaque exemple. Une indéniable amélioration de notre approche consisterait à étendre Why3 pour accepter en entrée un type mutable OCaml et construire automatiquement d'une part le modèle mémoire et d'autre part le *driver* pour l'extraction.

Références

[1] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL : ANSI/ISO C Specification Language, version 1.4*, 2009. http://frama-c.cea.fr/acsl.html.

[2] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Let's verify this with Why3. *International Journal on Software Tools for Technology Transfer (STTT)*, 17(6) :709–727, 2015. See also http://toccata.lri.fr/gallery/fm2012comp.en.html.

[3] Rod Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7 :23–50, 1972.

[4] Arthur Charguéraud and François Pottier. Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. *Journal of Automated Reasoning*, September 2017.

[5] Martin Clochard, Jean-Christophe Filliâtre, and Andrei Paskevich. How to avoid proving the absence of integer overflows. In Arie Gurfinkel and Sanjit A. Seshia, editors, *7th Working Conference on Verified Software : Theories, Tools and Experiments (VSTTE)*, Lecture Notes in Computer Science, San Francisco, California, USA, July 2015. Springer.

[6] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC : A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *Lecture Notes in Computer Science*. Springer, 2009.

[7] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C : A software analysis perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, number 7504 in Lecture Notes in Computer Science, pages 233–247. Springer, 2012.

[8] Jean-Christophe Filliâtre. One logic to use them all. In *24th International Conference on Automated Deduction (CADE-24)*, volume 7898 of *Lecture Notes in Artificial Intelligence*, pages 1–20, Lake Placid, USA, June 2013. Springer.

[9] Jean-Christophe Filliâtre. Le petit guide du bouturage, ou comment réaliser des arbres mutables en OCaml. In *Vingt-cinquièmes Journées Francophones des Langages Applicatifs*, Fréjus, France, January 2014. https://www.lri.fr/~filliatr/publis/bouturage.pdf.

[10] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. A pragmatic type system for deductive verification. Research report, Université Paris Sud, 2016. https://hal.archives-ouvertes.fr/hal-01256434v3.

[11] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. *Formal Methods in System Design*, 48(3) :152–174, 2016.

[12] Jean-Christophe Filliâtre and Claude Marché. Multi-Prover Verification of C Programs. In *Sixth International Conference on Formal Engineering Methods*, pages 15–29. Springer, 2004.

[13] Thierry Hubert and Claude Marché. A case study of C source code verification : the Schorr-Waite algorithm. 2005.

[14] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast : A powerful, sound, predictable, fast verifier for C and Java. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011.

[15] K. Rustan M. Leino. Dafny : An automatic program verifier for functional correctness. In *LPAR-16*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[16] W. Reif, G. Schnellhorn, and K. Stenzel. Proving system correctness with KIV 3.0. In William McCune, editor, *14th International Conference on Automated Deduction*, Lecture Notes in Computer Science, pages 69–72, Townsville, North Queensland, Australia, july 1997. Springer.

[17] J. C. Reynolds. Separation logic : a logic for shared mutable data structures. In *17th Annual IEEE Symposium on Logic in Computer Science*. IEEE Comp. Soc. Press, 2002.

[18] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10 :501–506, 1967.