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 l'indirection qui serait causée par le recours à deux types, un type somme d'une part et un type enregistrement d'autre part. D'autres solutions moins élégantes consisteraient à utiliser des valeurs récursives ou, pire encore, la fonction `Obj.magic` [2]. Avec un type comme le type `cell`, on obtient une représentation analogue à celle 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 montrerons aussi, par une systématisation de la construction des modèles mémoire, comment nous outillons notre méthodologie au dessus de Why3 pour obtenir automatiquement les modèles sur lesquels repose l'effort de spécification et de preuve.

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 4. 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` ou encore que la correction de la fonction `concat` repose sur l'hypothèse que les deux listes sont totalement disjointes. En effet, si les deux listes passées à `concat` partagent un suffixe, le résultat n'est plus une liste chaînée finie.

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 `loc` pour représenter l'adresse mémoire d'un constructeur `Cons`. Le type `cell` reste un type algébrique.

```
type loc 'a
type cell 'a =
  | Nil
  | Cons (loc 'a)
```

Nous modélisons ensuite le contenu du tas à l'aide du type `mem` suivant :

```
type mem 'a = {
  mutable  content: loc 'a -> option 'a;
  mutable     next: loc '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 `loc` 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 : `alloc_cons` pour allouer un `Cons`, `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) (c1 c2: cell 'a)  : unit
  requires { match c1 with Nil -> false | Cons l -> mem.next l <> None end }
  writes   { mem.next }
  ensures  { match c1 with
             | Nil    -> false
             | Cons l -> mem.next = (old mem.next)[l <- Some c2]
             end }
```

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. La précondition exige que `l1` soit une valeur différente de `Nil` et allouée. 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).

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`.

```
  predicate is_list_from_to (mem: mem 'a)
    (from : cell 'a) (s: view 'a) (until: cell 'a) =
    let n = length s in
    n = 0 /\ from = until
  \/
    n > 0 /\ from = Cons s[0] /\
    (forall i. 0 <= i < n -> Cons s[i] <> until) /\
    distinct s /\
    (forall i. 0 <= i < n - 1 -> mem.next s[i] = Some (Cons 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: view 'a) : cell 'a
  requires { is_list_from_to mem l1 s1 Nil }
  requires { is_list_from_to mem l2 s2 Nil }
  requires { disjoint_seq 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 = Cons s1[!step] }
      invariant { is_list_from_to mem !n !tail1 Nil }
      invariant { !step + Seq.length !tail1 = Seq.length s1 }
      variant   { Seq.length !tail1 }
      n := get_next mem !n;
      incr step;
      tail1 := !tail1 [ 1 .. ]
    done;
    set_next mem !n l2;
    l1
  end
end
```

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

Commençons par expliquer la spécification de cette fonction. En précondition, nous exigeons que `l1` et `l2` soient des listes finies (lignes 17–18), 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 des deux listes dans des arguments fantômes `s1` et `s2` de type `seq (cell 'a)`. Le type `seq` 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`, ne contient pas de répétition donc en particulier de boucle (ligne 8) et est composée exactement des cellules contenues dans la liste `s`. 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 19) et pour exprimer la postcondition (ligne 20).

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` et toutes les constantes et opérations de notre modèle.

```
syntax type cell         "%1 SinglyLL.cell"
syntax function Nil      "SinglyLL.Nil"
syntax function Cons     "SinglyLL.Cons %1"
syntax val  alloc_cons   "SinglyLL.Cons { content = %1; next = %2 }"
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 ici la définition du type `cell` et les définitions des fonctions comme `get_next`, à la manière des 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.

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.

3 Automatisation

Nous avons développé un outil qui automatise la construction du modèle mémoire et du *driver* associé. Cet outil prend en entrée la définition d'un ou plusieurs types OCaml, mélangeant enregistrements et types algébriques. Pour chaque type contenant des composantes mutables, il construit un modèle mémoire associé analogue à celui présenté dans la section précédente. Par exemple, pour un type comme

```
type tree = { v: int; mutable sub: tree list }
```

il va construire

- un type abstrait `loc_tree` pour modéliser les valeurs de type `tree` ;
- un type `mem_tree` pour modéliser le contenu de la mémoire, de la forme

```
type mem_tree = {
  mutable   v: loc_tree -> option int;
  mutable sub: loc_tree -> option (list loc_tree);
} invariant { forall l. v l = None <-> sub l = None }
```

- une constante `empty_mem_tree` et une fonction d'allocation `mk_tree` ;
- des fonctions d'accès en lecture et écriture aux différents champs, à savoir ici `get_v`, `get_sub` et `set_sub` ;
- enfin, un *driver* pour l'extraction, qui envoie notamment le type `loc_tree` vers le type OCaml `tree` et les fonctions comme `get_v`, `set_v`, etc., vers des définitions OCaml.

Pour un type algébrique, l'outil va construire d'une part un type algébrique Why3 avec les mêmes constructeurs et d'autre part autant de modèles mémoire qu'il y a de constructeurs avec des composantes mutables. Par exemple, sur un type OCaml comme

```
type alg = Nothing | A of { mutable a: int } | B of { mutable b: bool }
```

l'outil va construire deux modèles mémoire distincts pour les deux enregistrements, avec notamment deux types `loc_A` et `loc_B`, et deux types `mem_A` et `mem_B`, ainsi qu'un type algébrique

```
type alg = Nothing | A loc_A | B loc_B
```

On aurait pu aussi construire un seul modèle mémoire, avec un seul type `loc_alg` et un seul type `mem_alg`. Mais distinguer les deux modèles nous permet d'obtenir gratuitement le fait qu'une modification d'une valeur de la forme `A` n'a pas d'incidence sur les valeurs de la forme `B` qui existent par ailleurs. C'est toujours l'idée de *components-as-arrays* de Burstall [3].

Il reste à la charge de l'utilisateur de réunir différents modèles mémoire dans un seul type Why3, s'il le souhaite, ou encore de définir des prédicats sur la base de cette modélisation pour les besoins de la spécification, comme le prédicat `is_list_from_to` dans la section précédente. Ceci ne saurait être automatisé. Nous en donnons des exemples dans la section suivante.

4 Études de cas

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

```
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 (loc 'a);
7    mutable ghost used_mem : mem 'a
8  } invariant { length.OneTime.valid }
9    invariant { length = 0 -> first = last = Nil }
10   invariant { length > 0 -> first = Cons list[0] /\
11                             last = Cons list[length - 1] /\
12                             used_mem.next list[length - 1] = Some Nil }
13   invariant { forall x: loc '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 (Cons list[i+1]) }
16   invariant { forall i. 0 <= i < length ->
17                 used_mem.contents list[i] = Some view[i] }
18   invariant { length = Seq.length view = Seq.length list }
19   invariant { distinct list }
```

FIGURE 3 – Type des files.

4.1 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.

Modélisation. Le type `cell` des listes chaînées est modélisé en Why3 exactement comme dans la section 2. 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 longueur (le type `OneTime` représente des entiers qui ne peuvent être utilisés que linéairement) ;
- dans le cas d'une file vide, les champs `first` et `last` sont tous deux `Nil` (ligne 9) ;
- 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 10 – 12) ;
- toute adresse répertoriée dans `list` correspond à une adresse vers une cellule mémoire allouée et vice-versa (ligne 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 (ligne 19). 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, c'est-à-dire qu'elles ne partagent aucune cellule de type `cell`. En particulier, 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 valeurs modifiées par 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 6). Sinon, elle distingue le cas où q_2 est vide (lignes 7–13) du cas général (lignes 15–22). 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 par 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.

4.2 Parcours en profondeur

L'exemple suivant est celui d'un programme qui effectue un parcours en profondeur sur un graphe mutable. En OCaml les sommets du graphe sont représentés par le type suivant :

```
type node =
  | Null | Node of { mutable m: bool; mutable left: node; mutable right: node }
```

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

```
let transfer (q1 q2: t 'a) : unit
  requires { disjoint_mem q1.used_mem q2.used_mem }
  writes   { q1, q2 }
  ensures  { q2.view = (old q2.view) ++ (old q1.view) }
  ensures  { q1.view = empty }
= if not (is_empty q1) then
    if is_empty q2 then begin
      q2.length, q1.length <- q1.length, OneTime.zero ();
      q2.first, q2.last <- q1.first, q1.last;
      q2.list <- q1.list;
      q2.view <- q1.view;
      q2.used_mem, q1.used_mem <- q1.used_mem, empty_memory;
      q1.first, q1.last, q1.list, q1.view <- Nil, Nil, Seq.empty, Seq.empty;
    end else begin
      let len = OneTime.add q2.length q1.length in
      q2.length, q1.length <- len, OneTime.zero ();
      set_next q2.used_mem q2.last q1.first;
      q2.last, q2.list, q2.view <-
        q1.last, q2.list ++ q1.list, q2.view ++ q1.view;
      q2.used_mem, q1.used_mem <-
        mem_union q2.used_mem q1.used_mem, empty_memory;
      q1.first, q1.last, q1.list, q1.view <- Nil, Nil, Seq.empty, Seq.empty
    end
```

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

pendant le parcours. Notre outil (décrit dans la section 3) introduit deux types `loc` et `node`, de la manière suivante,

```
type loc
type node = | Null | Node loc
```

ainsi qu'un type `mem` regroupant les valeurs des trois champs

```
type mem = {
  mutable     m: loc -> option bool;
  mutable  left: loc -> option node;
  mutable right: loc -> option node;
} invariant { forall l. m l <> None <-> left l <> None <-> right l <> None }
```

et des fonctions `get` et `set` pour chacun des trois champs.

On en vient maintenant à la preuve du parcours en profondeur. Nous introduisons une table globale `busy` pour garder trace de tous les sommets dont l'exploration est en cours :

```
val ghost busy: ref (loc -> bool)
```

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: loc -> option node) (x y: node) =
  match x with
  | Null   -> false
  | Node l -> left l = Some y \/ right l = Some y
  end

inductive path (left right: loc -> option node) (x y: node) =
| path_nil:  forall x l r. path l r x x
| path_cons: forall x y z l r. 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 un 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 mesure d'implémenter et spécifier une fonction `dfs` qui réalise le parcours en profondeur. Le profil de cette fonction est le suivant :

```
let rec dfs (ghost mem: mem) (c: node) (ghost root: node) : unit
  if not (c = Null) && not (get_m mem c) then begin
    let ghost l = match c with Null -> absurd | Node l -> l end in
    ghost set busy l True;
    set_m mem c True;
    dfs mem (get_left mem c) root;
    dfs mem (get_right mem c) root;
    ghost set busy l False;
    ()
  end
```

L'argument `c` représente le sommet courant et l'argument fantôme `root` le 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 mem.left mem.right mem.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 mem.left mem.right mem.m }
```

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

```
  requires { path mem.left mem.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 mem.left mem.right mem.m !busy }
  ensures  { only_descendants_are_marked root mem.left mem.right mem.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 l. (old mem).m[l] = Some True -> mem.m[l] = Some 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 l. !busy[l] = True -> (old !busy)[l] = True }
```

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

```
  ensures  { match c with Null -> true | Node l -> mem.m[l] = Some True end }
```

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`. Si le graphe était fini, et cette information donnée en précondition de `dfs`, nous pourrions alors prouver la terminaison en ajoutant un variant adéquat. C'est ce qui a été fait, par exemple, dans la preuve de correction d'une implémentation de l'algorithme de Shorr-Waite dont il est question dans la section suivante.

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` qui encapsule le parcours en profondeur à partir d'une racine donnée :

```
let mark (ghost mem: mem) (root: node) : unit
  requires { forall l. mem.m[l] = Some False /\ !busy[l] = False }
  diverges
  ensures  { only_descendants_are_marked root mem.left mem.right mem.m }
  ensures  { all_descendants_are_marked root mem.left mem.right mem.m }
  ensures  { forall l. !busy[l] = False }
=
  dfs mem 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 les 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 Alt-Ergo, CVC4 et Z3.

4.3 Autres exemples

Nous avons validé notre méthodologie sur plusieurs autres exemples que l'espace ne nous laisse pas ici décrire convenablement en détail. Nous présentons ici brièvement le tri fusion en place de listes simplement chaînées et l'algorithme de Schorr-Waite [18]. Ces développements peuvent être trouvés dans l'archive accompagnant la publication.

Tri fusion. Dans cet exemple, on prouve un tri fusion en place sur des listes simplement chaînées. Nous reprenons donc naturellement le modèle mémoire qui nous est maintenant familier (sections 2 et 4.1) et en particulier le prédicat `is_list_from_to` de la figure 2. Pour *encadrer* les effets des différentes fonctions, nous avons aussi introduit des prédicats tels que le prédicat `frame_mem` suivant qui exprime que deux mémoires coïncident sur un ensemble d'adresses :

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

On montre alors que certaines propriétés sont préservées lorsque la mémoire n'est pas modifiée aux adresses concernées. Par exemple, la préservation du prédicat `is_list_from_to` est exprimée par le lemme suivant :

```
lemma frame_is_list: forall m1 m2: mem 'a, s: view 'a, l: loc 'a.
  is_list_from_to m1 l s Nil -> frame_mem m1 m2 s ->
  is_list_from_to m2 l s Nil
```

Pendant la preuve d'un programme, un tel lemme est typiquement utilisé avec deux états de la mémoire correspondants à deux moments de l'exécution.

Algorithme de Schorr-Waite. À l'instar de l'algorithme de parcours en profondeur, il s'agit d'un algorithme de marquage des nœuds d'un graphe, mais cette fois-ci sans utiliser de mémoire auxiliaire. Il effectue un parcours en profondeur en utilisant le graphe lui-même comme une pile, le modifiant et le restaurant au fur et à mesure du parcours.

Nous avons prouvé une version de l'algorithme de Schorr-Waite sur un graphe où chaque sommet possède exactement deux pointeurs fils et une marque mutable, comme dans le parcours en profondeur vu plus haut section 4.2, mais aussi une seconde marque booléenne.

```
type node =
  | Null
  | Node of { mutable m: bool; mutable c: bool;
              mutable left: node; mutable right: node }
```

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 déclarons un dictionnaire global qui associe à chaque nœud le chemin du graphe de la racine jusqu'à ce nœud :

```
val ghost path_from_root: ref (loc -> list loc)
```

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: loc -> loc) (x y: loc) (p: list loc) = ...
```

La liste `p` garde trace des sommets du chemin entre `x` et `y`, en excluant `y`.

Comme pour le parcours en profondeur, on montre que l'algorithme de Schorr-Waite marque, à partir d'un graphe où toutes les marques sont `false`, exactement tous les nœuds atteignables. On montre également qu'il restaure bien la structure de graphe à l'identique. Enfin, on prouve la terminaison de cet algorithme, sous l'hypothèse que le graphe est fini.

5 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 4.3.

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].

6 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 de 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 développé un prototype qui automatise la construction de ce modèle mémoire.

Nous avons utilisé cette méthode avec succès sur de 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 du tri fusion en place 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 de séparation en Why3 afin 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_mem` utilisé dans la preuve du tri fusion.

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)*, volume 9593 of *Lecture Notes in Computer Science*, pages 94–109, 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.