Une preuve formelle de l'algorithme de Tarjan-1972 pour trouver les composantes fortement connexes dans un graphe

Ran Chen, Jean-Jacques Lévy

Une preuve formelle de l'algorithme de Tarjan-1972 pour trouver les composantes fortement connexes dans un graphe

Ran Chen[1] & Jean-Jacques Lévy[2]

1: Institute of Software CAS, Pékin & Inria, Saclay
ran.chen@inria.fr
2: Inria, Paris
jean-jacques.levy@inria.fr

Résumé

Nous présentons une preuve formelle de l'algorithme de Tarjan (1972) pour trouver les composantes fortement connexes dans un graphe. Cet algorithme exécute une seule passe sur le graphe le parcourant en profondeur d'abord. Notre programme est ici écrit dans un style de programmation fonctionnelle dans le langage Why3-ML. La preuve utilise des lemmes et des assertions exprimés dans la logique de Why3. La majorité d'entre eux sont prouvés automatiquement à l'exception de 5 assertions ou lemmes prouvés manuellement en Coq. Une preuve d'une version impérative de ce programme ne figure pas dans notre article, mais nous indiquons comment y parvenir par raffinements successifs. Un point important de notre article est que nos preuves sont intuitives et lisibles par un humain.

1. Introduction

Nous présentons une preuve formelle de l'algorithme de Tarjan-1972 [22, 6, 20] pour trouver les composantes fortement connexes dans un graphe. Cet algorithme exécute une seule passe sur le graphe en effectuant un parcours en profondeur d'abord. Il maintient une pile des sommets visités et le rang du plus ancien sommet accessible par au plus un arc de retour dans l'arbre de recouvrement de chaque sommet. Dans cet article, nous adoptons un style de programmation fonctionnelle et associons à chaque sommet son rang dans la pile des sommets visités.

Chaque sommet peut avoir trois états : blanc (sommet non visité), gris (sommet en cours de visite), noir (sommet visité). Notre programme Why3-ML repose sur deux fonctions mutuellement récursives *dfs1* et *dfs'* qui prennent respectivement comme arguments un sommet x et un ensemble de sommets *roots* et qui retournent le rang m du plus ancien sommet accessible par un arc de retour. Les deux fonctions utilisent des variables d'état passées comme arguments et résultats complémentaires ; ce sont un ensemble de sommets noirs, la pile courante et l'ensemble des composantes fortement connexes déjà trouvées. Un ensemble fantôme (*ghost*) de sommets gris est aussi passé comme argument et utilisé dans la preuve de correction, mais cet ensemble n'intervient pas dans l'algorithme. Donc les deux fonctions prennent cinq arguments et retournent un quadruplet.

Les graphes sont représentés par un ensemble fini *vertices* de sommets et une fonction *successors* qui associe à chaque sommet l'ensemble fini des sommets directement accessibles depuis ce sommet. La pile courante est une liste de sommets dont le premier élément représente le sommet de la pile. Le programme utilise aussi les fonctions de la bibliothèque Why3 sur les ensembles (*add* qui ajoute un

élément à un ensemble, *mem* qui teste l'appartenance à un ensemble, *choose* qui choisit aléatoirement un élément dans un ensemble, *remove* qui supprime un élément d'un ensemble, et *is_empty*, *union*, *inter*, *diff*, *subset*, *cardinal* dont le sens est intuitif sur des ensembles finis) et sur les listes (*Nil*, *Cons*, *lmem* qui teste l'appartenance à une liste, *length* qui donne la longueur d'une liste, *elements* qui retourne l'ensemble des éléments d'une liste, *num_occ* qui donne leur nombre d'occurrences, ++ qui concatène deux listes). Nous avons aussi deux fonctions auxiliaires : *rank* qui retourne le rang d'un élément dans l'image miroir de la liste, *split* qui retourne la paire des sous-listes obtenues par décomposition autour de la première occurrence de cet élément dans la liste. Si cet élément n'est pas dans la liste, on retourne *max_int()*, encore dénoté $+\infty$ (ici ce sera le cardinal de l'ensemble de tous les sommets). Enfin *min* retourne le minimum de deux entiers.

```
let rec dfs1 x blacks (ghost grays) stack sccs =
  let m = rank x (Cons x stack) in
  let (m1, b1, s1, sccs1) =
    dfs' (successors x) blacks (add x grays) (Cons x stack) sccs in
  if m1 ≥ m then
    let (s2, s3) = split x s1 in
    (max_int(), add x b1, s3, add (elements s2) sccs1)
  else
    (m1, add x b1, s1, sccs1)

with  dfs' roots blacks (ghost grays) stack sccs =
  if is_empty roots then
    (max_int(), blacks, stack, sccs)
  else
    let x = choose roots in
    let roots' = remove x roots in
    let (m1, b1, s1, sccs1) =
      if lmem x stack then
        (rank x stack, blacks, stack, sccs)
      else if mem x blacks then
        (max_int(), blacks, stack, sccs)
      else
        dfs1 x blacks grays stack sccs in
    let (m2, b2, s2, sccs2) =
      dfs' roots' b1 grays s1 sccs1 in
    (min m1 m2, b2, s2, sccs2)

function rank (x: vertex) (s: list vertex): int =
  match s with
  | Nil → max_int()
  | Cons y s' → if x = y && not (lmem x s') then length s' else rank x s'
  end

function max_int (): int = cardinal vertices
```

Le programme principal appelle *dfs'* avec l'ensemble *vertices* de tous les sommets du graphe, et des ensembles vides de sommets noirs et gris et aussi un ensemble vide *sccs* de composantes fortement connexes. Alors *dfs'* choisit aléatoirement un sommet x dans *roots*. Si ce sommet x est blanc, on appelle la fonction *dfs1* sur lui. Ce sommet x est alors empilé sur la pile courante et *dfs'* est appelé sur ses successeurs directs après avoir rangé x dans l'ensemble des sommets gris. Si la valeur retournée est plus grande ou égale (en fait égale) au rang de x, une nouvelle composante fortement connexe est ajoutée au résultat en dépilant tous les éléments jusqu'à x. Si la valeur retournée est plus petite que le rang de x, on retourne le minimum de cette valeur et des résultats à partir des autres sommets de l'ensemble *roots*. Si le sommet est non blanc et appartient à la pile courante, il est l'extrémité d'un arc

de retour dans l'arbre de recouvrement et on retourne son rang dans la pile courante. Si le sommet est non blanc et figure déjà dans l'ensemble des composantes fortement connexes déjà découvertes, on retourne $+\infty$ (donc *max_int()*).

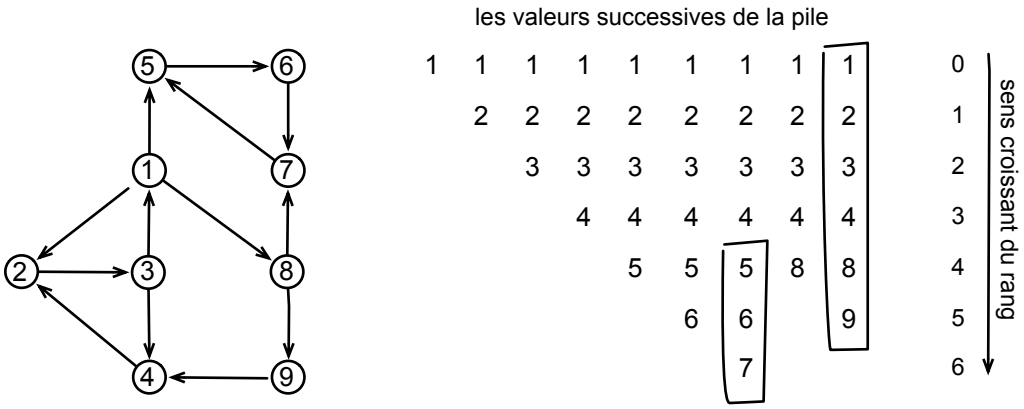

FIGURE 1 – Un exemple d'exécution : les sommets du graphe sont numérotés et empilés dans l'ordre de leur visite par *dfs1*. Lors de la découverte de la première composante fortement connexe $\{5, 6, 7\}$, ils sont désempilés et on continue les appels de *dfs1* jusqu'au retour de l'appel pour le sommet 1 où la deuxième composante $\{1, 2, 3, 4, 8, 9\}$ est découverte et désempilée. On remarque que la composante $\{5, 6, 7\}$ n'a pas d'arc de retour atteignant un sommet de rang inférieur au rang 4 du sommet 5. De même pour la deuxième composante avec le rang 0 du sommet 1.

Le programme peut paraître très inefficace puisqu'il manipule des listes et des ensembles, mais la version impérative est très efficace (cf. conclusion). La section suivante présente les pré-/post-conditions et assertions insérées dans le programme. La section 3 discute de la preuve des obligations de preuves et des lemmes. La conclusion est dans la section 4.

2. Pré-/Post-conditions

Nos graphes sont donc représentés par le type abstrait *vertex*, par la constante *vertices*, et par la fonction *successors* qui donne pour chaque sommet l'ensemble de ses successeurs. Le prédicat *edge* indique que deux sommets sont reliés par un arc.

```
type vertex
constant vertices: set vertex
function successors vertex : set vertex
axiom successors_vertices:
  ∀x. mem x vertices → subset (successors x) vertices
predicate edge (x y: vertex) = mem x vertices ∧ mem y (successors x)
```

La preuve intuitive utilise la structure d'arbre de Noel constituées par les composantes fortements connexes dans les arbres de recouvrement. Le parcours en profondeur empile tous les sommets rencontrés dans l'ordre préordre des arbres de recouvrement, et accumule dans la variable *sccs* les composantes cc_1, cc_2, ... cc_n (toutes noires) déjà trouvées (cf. figure 2). Les sommets déjà rencontrés, mais non encore rangés dans une composante fortement connexe de *sccs* sont tous contenus dans la pile *s*. Ces sommets sont soit noirs (car complètement explorés par la fonction *dfs1*), soit gris (car en cours d'exploration par *dfs1*). Pour chaque sommet, la fonction *dfs1* minimise le rang des sommets

accessibles à partir de ses descendants dans l'arbre de recouvrement par au plus un arc de retour. Quand ce rang est égal au rang de l'argument, la fonction *dfs1* a trouvé le premier sommet d'une nouvelle composante fortement connexe dont tous les éléments ont été empilés dans la pile après x.

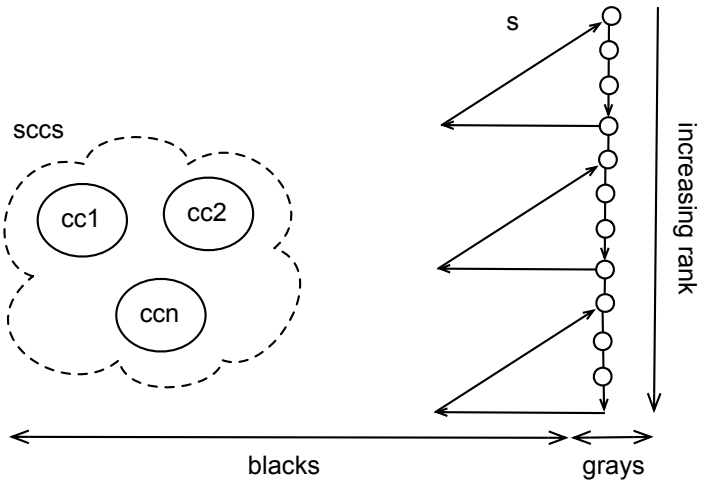

FIGURE 2 – Invariants sur les couleurs et la pile

Mais cette preuve demande à être formalisée. Dans notre article, nous n'introduisons pas la structure d'arbre de recouvrement. Nous ne considérons que les ensembles de sommets blancs, gris, ou noirs. Notre preuve formelle manipule donc des ensembles de sommets, leur accessibilité et la dynamique du programme. Notre convention pour les noms de variable est : x, y, z pour les sommets du graphe, s pour les piles, m pour les rangs, b pour les ensembles de sommets noirs, cc pour les composantes fortement connexes, *sccs* pour les ensembles de composantes fortement connexes.

La preuve formelle repose sur deux invariants et quatre post-conditions. Les deuxièmes expriment les propriétés de la valeur m du résultat en terme d'accessibilité avec au plus un arc de retour. Les premiers sont un peu plus délicats. Ils décrivent les relations entre les couleurs des sommets dans la pile courante et la couleur des composantes fortement connexes déjà trouvées. En plus ces invariants montrent l'accessibilité entre les sommets et la pile (comme illustré dans la figure 2). Dans le programme Why3-ML, le premier invariant dit que le prédicat *wff_color* est toujours vrai. Cela signifie que les ensembles de sommets gris et noirs sont disjoints, que les éléments de la pile courante sont exactement les sommets gris ou noirs (qui ne sont pas dans l'ensemble *sccs* des composantes fortement connexes — *set_of* est l'union de tous les élements de son argument). Enfin le prédicat *no_black_to_white* dit qu'il n'y a pas d'arc d'un sommet noir vers un sommet blanc (donc tout sommet noir ne peut être relié qu'à un sommet noir ou gris). En terme d'arbres de recouvrement, cela signifie qu'un nœud ne peut pointer directement vers un cousin à droite.

```
predicate no_black_to_white (blacks grays: set vertex) =
  ∀x x'. edge x x' → mem x blacks → mem x' (union blacks grays)

predicate wff_color (blacks grays: set vertex) (s: list vertex)
  (sccs: set (set vertex)) =
  inter blacks grays = empty ∧
  (elements s) == union grays (diff blacks (set_of sccs)) ∧
  (subset (set_of sccs) blacks) ∧
  no_black_to_white blacks grays
```

Le reste de l'invariant *wff_stack* exprime que les sommets gris peuvent accéder à tous les sommets

de rang plus élevé dans la pile, et que réciproquement tout sommet de la pile peut atteindre un sommet gris de rang plus petit. L'accessibilité est définie en termes de chemins dans le graphe par un prédicat inductif comme dans la bibliothèque standard de Why3. Le prédicat supplémentaire *simplelist* exprime que la pile ne contient aucune répétition. Donc la longueur de la pile est toujours moindre que le nombre total de sommets *max_int()*.

```
inductive path vertex (list vertex) vertex =
  | Path_empty:
      ∀x: vertex. path x Nil x
  | Path_cons:
      ∀x y z: vertex, l: list vertex.
      edge x y → path y l z → path x (Cons x l) z
predicate reachable (x z: vertex) =
  ∃l. path x l z

predicate wff_stack (blacks grays: set vertex) (s: list vertex)
  (sccs: set (set vertex)) =
  wff_color blacks grays s sccs ∧ simplelist s ∧ subset (elements s) vertices ∧
  (∀x y. mem x grays → lmem y s → rank x s ≤ rank y s → reachable x y) ∧
  (∀y. lmem y s → ∃x. mem x grays ∧ rank x s ≤ rank y s ∧ reachable y x)
```

Un second invariant caractérise la valeur courante de *sccs* en exprimant que c'est un ensemble de composantes fortement connexes toutes noires. Les composantes fortement connexes sont définies comme des ensembles maximaux de sommets connectés par deux chemins de sens opposés.

```
predicate in_same_scc (x z: vertex)  =  reachable x z ∧ reachable z x
predicate is_subscc (s: set vertex) =  ∀x z. mem x s → mem z s → in_same_scc x z
predicate is_scc (s: set vertex) =
  is_subscc s ∧ (∀s'. subset s s' → is_subscc s' → s == s')
```

La fonction *dfs1* a trois pré-conditions exprimant que le paramètre x est un sommet blanc atteignable par tous les sommets gris. Cette fonction a les quatre post-conditions mentionnées plus haut. Une première post-condition *E1* indique que x est retourné noir. Trois autres décrivent les propriétés du résultat m. La post-condition *E2* dit que la valeur de m est moindre que le rang du paramètre x ; la post-condition *E3* dit que m est le rang d'un sommet accessible depuis x ; la post-condition *E4* dit que s'il existe un arc de la nouvelle partie de la pile vers un sommet y dans la partie ancienne de la pile, la valeur de m retournée est inférieure ou égale au rang de y. Ces trois post-conditions disent que le résultat m est soit le rang de x ou celui d'un sommet y de rang minimal dans la pile au début de *dfs1*. Enfin, trois autres post-conditions dites de "monotonie" indiquent que la partie finale de la pile est une extension noire de la pile initiale, que les ensembles des sommets noirs et des composantes fortement connexes sont augmentés en fin de *dfs1*. La fonction *dfs'* a les mêmes invariants et pré-/post-conditions, modulo l'extension naturelle à l'ensemble des racines *roots*. En fait, la fonction *dfs1* pourrait être dépliée dans *dfs'*, mais nous trouvons plus naturel de garder ces deux fonctions séparées. (Les définitions des prédicats utilisés figure en page 8)

```
let rec dfs1 x blacks (ghost grays) stack sccs =
requires{mem x vertices} (* R1 *)
requires{access_to grays x} (* R2 *)
requires{not mem x (union blacks grays)} (* R3 *)
(* invariants *)
requires{wff_stack blacks grays stack sccs} (* I1a *)
requires{∀cc. mem cc sccs ↔ subset cc blacks ∧ is_scc cc} (* I2a *)
returns{(_, b, s, sccs_n) → wff_stack b grays s sccs_n} (* I1b *)
returns{(_, b, _, sccs_n) → ∀cc. mem cc sccs_n ↔ subset cc b ∧ is_scc cc} (* I2b *)
(* post conditions *)
returns{(_, b, _, _) → mem x b} (* E1 *)
```

```
returns{(m, _, s, _) → m ≤ rank x s} (* E2 *)
returns{(m, _, s, _) → m = max_int() ∨ rank_of_reachable m x s} (* E3 *)
returns{(m, _, s, _) → ∀y. crossedgeto s y stack → m ≤ rank y stack} (* E4 *)
(* monotony *)
returns{(_, b, s, _) → ∃s'. s = s' ++ stack ∧ subset (elements s') b} (* M1 *)
returns{(_, b, _, _) → subset blacks b} (* M2 *)
returns{(_, _, _, sccs_n) → subset sccs sccs_n} (* M3 *)

  let m = rank x (Cons x stack) in
  let (m1, b1, s1, sccs1) =
  dfs' (successors x) blacks (add x grays) (Cons x stack) sccs in
  assert{inter grays (add x b1) = empty};  (* for I1b *)
  if m1 ≥ m then begin
    let (s2, s3) = split x s1 in
    assert{s3 = stack};
    assert{subset (elements s2) (add x b1)};
    assert{is_subscc (elements s2) ∧ mem x (elements s2)};
    assert{∀y. in_same_scc y x → mem y (elements s2)};
    assert{is_scc (elements s2)};
    assert{inter grays (elements s2) = empty};  (* help provers *)
    (max_int(), add x b1, s3, add (elements s2) sccs1) end
  else  begin
    assert{∃y. mem y grays ∧ rank y s1 < rank x s1 ∧ reachable x y};
    (* for I1b and I2b *)
    (m1, add x b1, s1, sccs1) end
```

3. La preuve formelle

Il y a quelques assertions dans le corps de la fonction *dfs1*. La plupart d'entre elles figurent après la décomposition de la pile par rapport au sommet x. C'est la partie critique du programme puisqu'on doit montrer qu'on a alors trouvé une nouvelle composante fortement connexe dans la pile empilée après x. Les deux premières assertions sont prouvées facilement automatiquement. En effet la partie basse de la pile *s3* est l'ancienne pile à l'entrée de *dfs1* (puisqu'il n'y a pas de répétitions dans la pile). La deuxième assertion dit que la nouvelle partie *s2* de la pile (qui commence par x) est toute noire à la fin de la fonction *dfs1*. Les deux assertions qui suivent sont plus délicates.

Dans la troisième assertion, nous prouvons que les éléments de *s2* constituent un sous-ensemble d'une composante fortement connexe. En effet, soient deux sommets y et z dans *s2*, ils sont tous deux fortement connectés à x, puisque x est gris dans *dfs'*, et puisque nous savons que tous deux peuvent accéder à un sommet gris de la pile de rang inférieur à celui de x puisqu'après x il n'y a que des sommets noirs dans cette pile.

La quatrième assertion sert à montrer la maximalité de l'ensemble des éléments de *s2* comme sous-ensemble d'une composante fortement connexe. Donc considérons un sommet y dans la même composante fortement connexe que x. Nous voulons montrer que y est dans *s2*. C'est une preuve manuelle interactive exécutée en Coq ; nous n'avons pu l'automatiser. Pourtant deux lemmes (prouvés automatiquement) disent que si y n'est pas dans *s2*, il y a un arc de x' à y' tous les deux dans la composante fortement connexe de x tels que x' est dans *s2* et y' n'est pas dans *s2*. De plus, il n'y a alors que trois cas. Cas 1 : y' dans *stack*. Soit x' est x, mais nous savons alors que le résultat de *dfs'* est plus petit que le rang de tous les successeurs de x, et donc plus petit que le rang de y' qui est strictement plus petit que le rang de x. Soit x' est dans le reste de *s2*, et alors il existe un arc de retour vers y', la post-condition de *dfs'* contredit que le résultat de *dfs'* soit plus grand ou égal au rang de x. Cas 2 : y' est dans *sccs*. Alors il ne peut pas exister un chemin de y' vers x' puis que tous

les chemins de *sccs* restent dans *sccs* et la pile est disjointe de *sccs* à cause de l'invariant *wff_color*. Cas 3 : *y'* est blanc. C'est impossible puisque *x'* est dans *s2* et donc soit égal à x et alors *y'* dans les successeurs de x se retrouve noir ou gris en fin de *dfs'*, soit différent de x et donc dans la partie restante de *s2* toute noire, et alors il n'existe pas d'arc des noirs vers les blancs.

Les autres assertions sont montrées par les prouveurs automatiques (Alt-Ergo, E-prover, CVC4, Z3, Spass), tous nécessaires. La cinquième assertion exprime que *s2* est une composante fortement connexe. La sixième assertion est une propriété de l'algèbre booléenne sur les ensembles que nos prouveurs ne peuvent découvrir. Cette assertion est utile pour prouver l'invariant sur les couleurs. Dans le deuxième cas de l'instruction conditionnelle, il n'y a qu'une seule assertion. A ce point du programme, comme mentionné plus haut, nous savons que les éléments de *s2* forment un sous-ensemble d'une composante fortement connexe. Mais l'assertion nous dit que ce sous-ensemble est strict puisqu'il y a un élément gris à l'intérieur, et donc en dehors de *s2*.

Les post-conditions de *dfs1* sont également démontrées automatiquement grâce à quelques lemmes. On peut par exemple noter que la post-condition sur les composantes fortement connexes utilise un lemme d'unicité de ces composantes, lemme qui indique qu'une seule composante fortement connexe contient le sommet x. Une seule post-condition de *dfs1* a besoin d'une courte preuve en Coq. Il s'agit de la monotonie de la pile qui exprime que la pile est étendue par une partie noire. Cette condition contient un quantificateur existentiel que nous avons eu du mal à traiter automatiquement. Une autre preuve Coq est nécessaire pour la pré-condition de *dfs1* sur l'accessibilité des sommets gris à partir de tout sommet.

Les obligations de preuves pour *dfs'* sont plus simples. Aucune assertion n'est nécessaire dans le corps de la fonction. Seules deux preuves Coq sont nécessaires. La première pour démontrer que le résultat m est inférieur au rang de toutes les sommets de *roots*. Il y a alors de nombreux cas avec la fonction *min* et les résultats possibles $+\infty$. (Il faut faire deux fois cette preuve selon les deux alternatives de *dfs1*)

En dehors des deux fonctions *dfs1* et *dfs'*, on utilise un certain nombre de lemmes : 5 lemmes sur la fonction *rank*, 6 lemmes sur les listes, 17 lemmes sur les ensembles(!), 8 lemmes sur les composantes fortement connexes et 7 lemmes spécialisés à notre programme.

Toute la preuve détaillée se trouve en `http:jeanjacqueslevy.net/why3/`.

```
with  dfs' roots blacks (ghost grays) stack sccs =
requires{subset roots vertices} (* R1 *)
requires{∀x. mem x roots → access_to grays x} (* R2 *)
(* invariants *)
requires{wff_stack blacks grays stack sccs} (* I1a *)
requires{∀cc. mem cc sccs ↔ subset cc blacks ∧ is_scc cc}  (* I2a *)
returns{(_, b, s, sccs_n) → wff_stack b grays s sccs_n} (* I1b *)
returns{(_, b, _, sccs_n) → ∀cc. mem cc sccs_n ↔ subset cc b ∧ is_scc cc} (* I2b *)
(* post conditions *)
returns{(_, b, _, _) → subset roots (union b grays)} (* E1 *)
returns{(m, _, s, _) → ∀x. mem x roots → m ≤ rank x s} (* E2 *)
returns{(m, _, s, _) → m = max_int() ∨ ∃x. mem x roots ∧ rank_of_reachable m x s} (* E3 *)
returns{(m, _, s, _) → ∀y. crossedgeto s y stack → m ≤ rank y stack} (* E4 *)
(* monotony *)
returns{(_, b, s, _) → ∃s'. s = s' ++ stack ∧ subset (elements s') b} (* M1 *)
returns{(_, b, _, _) → subset blacks b} (* M2 *)
returns{(_, _, _, sccs_n) → subset sccs sccs_n} (* M3 *)

  if is_empty roots then
    (max_int(), blacks, stack, sccs)
  else
    let x = choose roots in
```

```
    let roots' = remove x roots in
    let (m1, b1, s1, sccs1) =
    if lmem x stack then
      (rank x stack, blacks, stack, sccs)
    else if mem x blacks then
      (max_int(), blacks, stack, sccs)
    else
      dfs1 x blacks grays stack sccs in
    let (m2, b2, s2, sccs2) =
      dfs' roots' b1 grays s1 sccs1 in
    (min m1 m2, b2, s2, sccs2)

let rec split (x : α) (s: list α) : (list α, list α) =
returns{(s1, s2) → s1 ++ s2 = s}
returns{(s1, _) → lmem x s → is_last_of x s1}
  match s with
  | Nil → (Nil, Nil)
  | Cons y s' → if x = y then (Cons x Nil, s') else
      let (s1', s2) = split x s' in
       ((Cons y s1'), s2)
  end
```

Enfin, voici en complément la définition exacte de trois prédicats utilisés dans les assertions de nos programmes, bien que leurs noms soient très explicites. Rappelons également que *mem* est la fonction d'appartenance à un ensemble, *lmem* est la fonction d'appartenance à une liste, et ++ est l'opération de concaténation de deux listes. Ces trois fonctions sont dans la bibliothèque standard de Why3.

```
predicate rank_of_reachable (m: int) (x: vertex) (s: list vertex) =
  ∃y. lmem y s ∧ m = rank y s ∧ reachable x y

predicate access_to (s: set vertex) (y: vertex) =
  ∀x. mem x s → reachable x y

predicate crossedgeto (s1: list vertex) (y: vertex) (s3: list vertex) =
  lmem y s3 ∧ ∃s2. (s1 = s2 ++ s3 ∧ ∃x. lmem x s2 ∧ edge x y)

predicate simplelist (s: list α) = ∀x. num_occ x s ≤ 1
```

Nous ne considérons pas ici le programme principal qui appelle la fonction *dfs'* avec les sommets tous blancs. Il reste alors à démontrer que les composantes fortement connexes trouvées sont les composantes accessibles à partir de l'ensemble *roots*, argument de *dfs'*. On retrouve alors la preuve des parcours en profondeur d'abord qui montre que tous les sommets noirs sont en fin de la fonction exactement l'ensemble des sommets accessibles depuis les sommets de *roots*. Remarquons toutefois que dans le cas où l'ensemble *vertices* des sommets est vide, l'ensemble vide est alors une composante fortement connexe, ce qui invaliderait l'invariant sur *sccs*. Fort heureusement, la fonction *dfs1* n'est alors pas appelée et on est libre de donner toute valeur au résultat du programme principal.

4. Vers les programmes impératifs

Pour obtenir les programmes des livres d'algorithmes, il faut supprimer les tests d'appartenance à la pile, les calculs de rang de sommets dans la pile et les structures d'ensembles. La technique usuelle consiste à considérer des numéros de série pour les sommets, en leur donnant leur numéro *num[x]* dans l'ordre pré-ordre des arbres de recouvrement, et en initialisant ces numéros à une valeur exceptionnelle

-1 pour indiquer leur couleur blanche ou en leur donnant la valeur $+\infty$ quand ils sont rangés dans l'ensemble *sccs* des composantes fortement connexes déjà trouvées.

On peut arriver à ces programmes par raffinements successifs. En fait nous visons la version fonctionnelle de ces programmes impératifs. On augmente l'état en ajoutant une fonction *num* qui donne un numéro à chaque sommet et un entier *sn* qui donne le prochain numéro disponible pour la numérotation des futurs sommets rencontrés.

```
let rec dfs1 x blacks (ghost grays) stack sccs sn num =
requires{sn = cardinal (union grays blacks) ∧ subset (union grays blacks) vertices}
(* invariants *)
requires{wff_num sn num stack} (* I3a *)
returns{(_, _, _, s, _, sn_n, num_n) → wff_num sn_n num_n s} (* I3b *)
(* post conditions *)
returns{(sn_n, m, _, s, _, _, num_n) → sn_n = m = max_int() ∨
                    ∃y. lmem y s ∧ sn_n = num_n[y] ∧ m = rank y s} (* E5 *)

  let m = rank x (Cons x stack) in
  let (n1, m1, b1, s1, sccs1, sn1, num1) =
    dfs' (successors x) blacks (add x grays) (Cons x stack) sccs (sn + 1) num[x ← sn] in
  if n1 ≥ sn then begin
    let (s2, s3) = split x s1 in
    (max_int(), max_int(), add x b1, s3, add (elements s2) sccs1, sn1, num1) end
  else
    (n1, m1, add x b1, s1, sccs1, sn1, num1)
```

On retourne donc deux résultats, m et n respectivement le rang et le numéro du plus ancien sommet visité avec un arc de retour à partir des descendants de x. Le flot de contrôle du résultat est identique grace à l'invariant suivant qui relie l'ordre préordre sur les arbres de recouvrement et les rangs dans la pile.

```
predicate wff_num (sn: int) (num: map vertex int) (s: list vertex) =
  (∀x. num[x] < sn ≤ max_int())  ∧
  (∀x y. lmem x s → lmem y s → num[x] ≤ num[y] ↔ rank x s ≤ rank y s)
```

En fait il est commode de faire ce raffinement en deux étapes. Une pour introduire les numérotation des sommets et la deuxième pour modifier la numérotation dans la fonction *split* qui alors rend trois valeurs, la pile décomposée et la nouvelle numérotation où les sommets sortis de la pile prennent tous un numéro $+\infty$. On peut alors tester cette valeur pour voir si le sommet n'est pas dans la pile. Le traitement complet se trouve en `http:jeanjacqueslevy.net/why3/`.

5. Conclusion

Nous avons montré une preuve formelle de l'algorithme de Tarjan-1972 pour trouver les composantes fortement connexes dans un graphe dirigé. Cette preuve a été réalisée avec le système Why3. La majeure partie est prouvée automatiquement, le reste étant réalisé en Coq. Nous avons essayé de simplifier cette preuve formelle pour la rendre présentable dans ses détails dans cet article. Il n'a pas été nécessaire d'introduire la structure d'arbres de recouvrement, et les nombreuses propriétés correspondantes. Nous nous sommes contentés de la structure de pile introduite par Tarjan pour cet algorithme, pile qui a l'avantage de ne représenter que la partie utile de l'arbre de recouvrement.

Nous avons presque montré la version impérative de cet algorithme, mais il serait intéressant de disposer des outils nécessaires pour passer par extraction à la version impérative, ce que nous n'avons pas réussi à faire pour le moment.

Enfin, il serait intéressant de comparer notre preuve (son style, sa longueur, sa lisibilité) à d'autres preuves formelles de cet algorithme, notamment avec d'autres systèmes de preuves formelles et avec d'autres logiques. Nous avons eu accès aux prouveurs automatiques grâce à Why3, mais l'utilisation du système nous a demandé une certaine agilité.

Remerciements

Merci à toute l'équipe Why3 pour quelques judicieuses remarques.

Références

[1] C. Barrett and C. Tinelli. CVC4, the smt solver. New-York University - University of Iowa, 2011. `http://cvc4.cs.nyu.edu`.

[2] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art : The Calculus of Inductive Constructions*. Springer, 2004.

[3] F. Bobot, S. Conchon, E. Contejean, M. Iguernelala, S. Lescuyer, and A. Mebsout. The alt-ergo automated theorem prover, 2008. `http://alt-ergo.lri.fr/`.

[4] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3 : Shepherd your herd of provers. In *Boogie 2011 : First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011.

[5] Coq Development Team. The coq 8.5 standard library. Technical report, Inria, 2015. `http://coq.inria.fr/distrib/current/stdlib`.

[6] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[7] L. de Moura and N. Björner. Z3, an efficient smt solver. Microsoft Research, 2008. `http://z3.codeplex.com`.

[8] G. Dowek et al. *Informatique et sciences du numérique- Spécialité ISN en terminale S*. Eyrolles, 2012.

[9] J.-C. Filliâtre et al. The why3 gallery of verified programs. Technical report, CNRS, Inria, U. Paris-Sud, 2015. `http://toccata.lri.fr/gallery/why3.en.html`.

[10] J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In M. Felleisen and P. Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, Mar. 2013.

[11] G. Gonthier et al. Finite graphs in mathematical components, 2012. Available at `http://ssr.msr-inria.inria.fr/~jenkins/current/Ssreflect.fingraph.html`, The full library is available at `http://www.msr-inria.fr/projects/mathematical-components-2/`.

[12] G. Gonthier, A. Mahboubi, L. Rideau, E. Tassi, and L. Théry. A modular formalisation of finite group theory. In *Theorem Proving in Higher-Order Logics (TPHOLS'07)*, volume 4732 of *Lecture Notes in Computer Science*, pages 86–101, 2007.

[13] G. Gonthier, A. Mahboubi, and E. Tassi. A Small Scale Reflection Extension for the Coq system. Rapport de recherche RR-6455, INRIA, 2008.

[14] P. Lammich and R. Neumann. A framework for verifying depth-first search algorithms. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, CPP '15, pages 137–146, New York, NY, USA, 2015. ACM.

[15] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 2009. `http://compcert.inria.fr`.

[16] J.-J. Lévy. *Essays for the Luca Cardelli Fest*, chapter Simple proofs of simple programs in Why3. Microsoft Research Cambridge, MSR-TR-2014-104, 2014.

[17] B. Meyer. Lampsort. Technical report, ETHZ, 2014. `http://bertrandmeyer.com/2014/12/07/lampsort`.

[18] F. Pottier. Depth-first search and strong connectivity in Coq. In *Journées Francophones des Langages Applicatifs (JFLA 2015)*, Jan. 2015.

[19] S. Schulz. System Description : E 1.8. In K. McMillan, A. Middeldorp, and A. Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*. Springer, 2013.

[20] R. Sedgewick and K. Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011.

[21] A. Tafat and C. Marché. Binary heaps formally verified in Why3. Research Report 7780, INRIA, Oct. 2011. `http://hal.inria.fr/inria-00636083/en/`.

[22] R. Tarjan. Depth-first search and linear graph algorithms. *Siam Journal on Computing*, 1(2) :146–160, 1972.

[23] L. Théry. Formally-proven Kosaraju's algorithm. Inria report, Hal-01095533.

[24] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. Spass version 3.5. In *22nd International Conference on Automated Deduction, CADE 2009*, number 5663 in LNCS, pages pp. 140–145, 2009.