Manuscrit auteur, publié dans "JFLA - Journées Francophones des Langages Applicatifs - 2012 (2012)"

hal-00665909, version 1 - 3 Feb 2012

Dérivation de l'algorithme de Schorr-Waite en Coq par une méthode algébrique

Jean-François Dufourd

Université de Strasbourg,
LSIIT (UMR CNRS-UdS 7005),
Pôle Technologique, BP10413, 67412 Illkirch, France
jfd@unistra.fr

Résumé

Nous présentons une spécification, une dérivation et une preuve de correction totale de l'algorithme de Schorr-Waite pour un marquage en profondeur d'un graphe binaire codé par des pointeurs. Notre approche est purement algébrique et fonctionnelle, depuis une spécification simple jusqu'à un programme récursif terminal simulant le langage C. Puis, nous obtenons par des transformations élémentaires un véritable programme en C, impératif et itératif.

1. Introduction

Nous relatons une nouvelle expérience de spécification formelle, dérivation et preuve de correction totale en Coq [2] de l'algorithme de Schorr-Waite basée sur une approche algébrique (au sens des spécifications algébriques, ou de l'algèbre universelle) fonctionnelle que nous pensons réutilisable pour d'autres problèmes. L'algorithme de Schorr-Waite [15] parcourt itérativement un graphe binaire codé par des pointeurs en profondeur d'abord et marque tous les nœuds visités. Le problème est classique, mais la solution de H. Schorr et W.M. Waite est économique parce qu'elle évite l'usage d'une pile auxiliaire par une gestion astucieuse des pointeurs. Un tel algorithme est utile dans un *ramasse-miettes*, pour le marquage des cellules référencées quand la mémoire disponible est réduite, ou dans un *modeleur géométrique*, pour la recherche de composantes connexes dans une *hypercarte* [8, 6].

Une littérature abondante traite de cet algorithme, dont la preuve de correction totale est considérée comme particulièrement difficile [3]. Rappelons que, pour un algorithme, *correction totale = terminaison (finie) + correction partielle*, c.-à-d. conformité par rapport à une spécification. Les premières preuves ont été faites "à la main" en utilisant la méthode des assertions [9] ou des transformations en sémantique dénotationnelle [17]. Depuis plus de dix ans, les preuves sont assistées par des outils automatiques. R. Bornat et al. ont écrit une preuve de correction partielle qui a été vérifiée à l'aide de l'éditeur de preuves Jape [3]. J.-R. Abrial a utilisé la méthode et la boîte à outils B pour raffiner et fusionner des spécifications en forme d'affectations élémentaires en sept étapes, avec des *obligations* de preuves automatisées [1].

F. Mehta et T. Nipkow ont prouvé la correction totale de l'algorithme de Schorr-Waite en logique d'ordre supérieur avec Isabelle/HOL [14]. Des chemins et des listes d'adresses sont les prémices de nos *orbites* et *pile interne* mémorisant les adresses en cours d'examen [5]. La base est la preuve de R. Bornat utilisant la logique de Hoare. A. Loginov et al. ont réalisé une preuve entièrement automatique de correction totale en utilisant une logique trivaluée, mais seulement pour des arbres ou dags binaires [13]. T. Hubert et C. Marché ont utilisé la méthode des assertions et le système Caduceus couplé à Coq pour une preuve directe d'une version en code source C de l'algorithme [10]. Ils ont prouvé

hal-00665909, version 1 - 3 Feb 2012

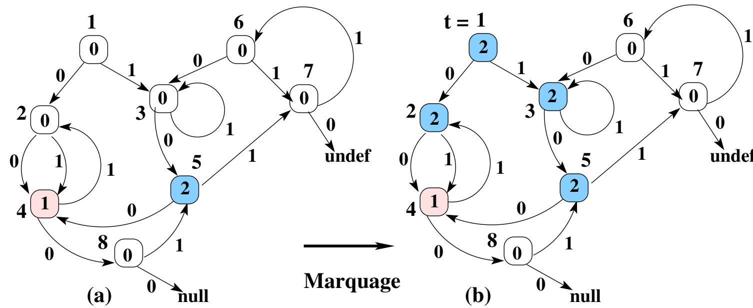

Figure 1: Graphe binaire initial (a) et son nouveau marquage à partir de t = 1 (b).

automatiquement qu'aucun déréférencement invalide de pointeur ne peut se produire. Parvenir à leur programme de départ en C a beaucoup motivé certains de nos choix de définition de fonctions.

Récemment, M. Giorgino et al. ont étudié une méthode par abstraction et raffinement pour prouver l'algorithme de Schorr-Waite en utilisant Isabelle/HOL [7]. Certaines de nos propositions sont assez proches des leurs, mais leur idée-clé est de travailler en deux temps (algorithme sur un arbre de recouvrement, puis enrichissement sur un graphe général) alors que nous traitons dès le départ un graphe général. De plus, ils raisonnent sur le parcours de composantes, un peu comme R.E. Tarjan [16], ce que nous évitons avec l'utilisation d'orbites [5]. Enfin, ils utilisent des transformateurs d'états et des monades en Isabelle pour travailler avec des programmes impératifs, quand nous allons directement vers des programmes en C.

L'article suit la méthode que nous proposons. La Section 2 présente une spécification algébrique des graphes binaires marqués. La Section 3 spécifie le marquage en profondeur dans les graphes par une fonction doublement récursive. La Section 4 définit les piles internes par la notion d'orbite. La Section 5 construit le marquage en profondeur avec une pile interne comme une fonction récursive terminale. La Section 6 spécifie inductivement cellules et mémoires avec la simulation d'un simple allocateur C. La Section 7 définit un isomorphisme entre graphes et mémoires. La Section 8 transporte le marquage avec pile interne de graphe à mémoire. La Section 9 extrait cette dernière opération en une fonction récursive terminale en OCaml, dérivée ensuite "à la main" en un programme itératif en C. Enfin, la Section 10 prouve que la spécification initiale correspond bien à la notion d'accessibilité habituelle, et la Section 11 conclut. Toutes les spécifications sont écrites en Calcul des Constructions Inductives et toutes les preuves sont guidées et vérifiées par Coq [2]. Dans la suite, le langage et le système Coq sont supposés connus. Le développement entier peut être consulté en ligne [4].

2. Graphes binaires

Définitions de base Les notations qui suivent sont celles de Coq. Nous supposons que `undef` et `null`, non nécessairement distincts, codent des valeurs particulières de `nat`. Ici, un *graphe binaire (fonctionnel marqué)* `g = (E,mark,son0,son1)` est un sous-ensemble `E` de *sommets* (ou *nœuds*) dans `nat - {undef,null}`, muni de trois fonctions : `mark` retourne un entier naturel de `{0,1,2}`, `son0` et `son1` retournent des entiers naturels nommés fils *gauche* et *droit*, n'appartenant pas nécessairement à `E`. Un exemple est en Fig. 1(a), avec `E = {1,...,8}`, des marques dans les cercles associés aux sommets (en blanc, gris clair ou gris foncé selon la marque, 0, 1 ou 2), `son0` et `son1` représentés par des arcs avec les étiquettes 0 ou 1. Par convention, les trois fonctions sont étendues à tout `nat` : en dehors de `E`, appelé *support* de `g`, `mark` retourne 0, `son0` et `son1` retournent `undef`. Pour éviter

de nombreux tests élémentaires, nous définissons d'abord énumérativement en Coq le type `nat2` des marques. Alors, le type `graph` est défini inductivement par deux *constructeurs* : `vg` retourne le graphe *vide* et `iv g x m x0 x1` *insère* dans un graphe `g` un nouveau sommet `x`, avec sa marque `m`, et ses deux fils `x0` et `x1` :

```
Inductive nat2 : Type:= zero : nat2 | one : nat2 | two : nat2.
Inductive graph: Type:=
   vg : graph | iv : graph -> nat -> nat2 -> nat -> nat -> graph.
```

Observateurs et invariant de graphe Le prédicat `exv g z`, qui teste l'existence dans le graphe `g` du nœud `z`, est défini récursivement par filtrage :

```
Fixpoint exv(g:graph)(z:nat): Prop:=
 match g with vg => False | iv g0 x _ _ _ => x = z \/ exv g0 z end.
```

Il en est de même pour les fonctions `mark` et `son` (compactant `son0` et `son1`). Si l'usage de `iv` respecte la précondition `prec_iv`, le graphe obtenu est *bien formé* et respecte l'invariant `inv_graph` :

```
Definition prec_iv(g:graph)(x:nat): Prop := ~ exv g x /\ x <> null /\ x <> undef.
Fixpoint inv_graph(g: graph): Prop :=
 match g with
     vg => True |  iv g0 x m _ _ => inv_graph g0 /\ prec_iv g0 x
 end.
```

Parmi les autres *observateurs* ainsi définis, `nv` est le nombre de sommets et `marksum` est la somme des marques des sommets d'un graphe, avec bien sûr :

```
Lemma marksum_bound: forall g, marksum g <= 2 * nv g.
```

Mutateurs Des fonctions de mise à jour sont décrites de la même manière : `chm g z m` change la marque de `z` en `m`, et `cha g k z zs` change le `k`-ième fils (ou *arc*, avec `k` $\in$ {`0`,`1`}), de `z` en `zs`. Leurs propriétés, prouvées inductivement, sont essentielles pour pouvoir par la suite identifier convenablement deux graphes. Ainsi, selon leurs arguments, `chm` et `cha` sont *idempotentes*, *permutatives*, *absorbantes*, et permutent entre elles. Par exemple, nous avons :

```
Lemma chm_chm: forall g z1 m1 z2 m2,
  chm (chm g z1 m1) z2 m2 =
    if eq_nat_dec z1 z2 then chm g z2 m2
    else chm (chm g z2 m2) z1 m1.
Lemma cha_idem: forall g k x y z, k <= 1 ->
  cha (cha g k x y) k x z = cha g k x z.
Lemma cha_chm: forall g x y z k m, k <= 1 ->
  cha (chm g z m) k x y = chm (cha g k x y) z m.
```

3. Spécification du marquage en profondeur

Préliminaires Pour mieux aborder l'algorithme de Schorr-Waite, nous élargissons le problème original en travaillant avec un graphe `g` quelconque, c.-à-d. *avec n'importe quel marquage initial* (entre 0 et 2) et *avec des fils quelconques* (dans le support de `g` ou non). Le problème consiste alors à parcourir en profondeur le *sous-graphe* de `g` constitué de tous les sommets marqués par 0 et *accessibles* depuis un entier naturel `t`, et à les marquer par 2. La Fig. 1(b) donne le résultat final sur le graphe en Fig. 1(a)

hal-00665909, version 1 - 3 Feb 2012

hal-00665909, version 1 - 3 Feb 2012

en partant avec `t = 1`. Avec cet énoncé, la *condition d'arrêt* du parcours en profondeur à partir d'un entier naturel `t` est `stop`, qui est facilement prouvée *décidable*, la fonction `stop_dec` testant si `stop g t` est satisfaite ou non. Le problème entier est alors résolu par la fonction que nous appelons `df` :

```
Definition stop g t := ~ exv g t \/ mark g t <> 0.
"Definition df(g:graph)(t:nat): graph :=
  if stop_dec g t then g
  else let g0 := df (chm g t one) (son g 0 t) in
       df (chm g0 t two) (son g 1 t)."
```

Si `t` n'est pas un sommet ou a une marque différente de 0, `df` retourne `g` non modifié. Sinon, `df` marque `t` avec 1, effectue un parcours à partir du fils gauche `son g 0 t`, puis marque `t` avec 2 et effectue un parcours à partir du fils droit `son g 1 t`. En conséquence, pour chaque sommet non bloquant, la marque indique le *nombre de visites* qu'il a reçues. Il nous semble que `df` énonce le problème de marquage de manière très simple. Aussi, nous la considérons désormais comme notre *spécification.* Nous prouverons d'ailleurs à la fin (Section 10) que `df` correspond exactement à la notion d'accessibilité. Cependant, une telle définition récursive ne peut pas être directement écrite en Coq sans traiter la terminaison (d'où les guillemets dans la définition ci-dessus). De plus, la récursion *emboîtée* (*double*) ajoute à la difficulté. Il est intéressant de voir comment ces obstacles peuvent être surmontés en Coq, en suivant la voie indiquée dans le Coq'Art [2] (pages 419-420).

La vraie spécification en Coq D'abord, nous définissons une *mesure* `mes` des graphes qui va décroître à chaque appel récursif et nous considérons deux relations binaires sur `graph`, `ltg` and `leg` :

```
Definition mes g:= 2 * nv g - marksum g.
Definition ltg g' g := mes g' < mes g.
Definition leg g' g := mes g' <= mes g.
```

On prouve rapidement que ces deux relations sont des *préordres*, respectivement *strict* et *large*, que `ltg` est *nœthérienne* sur `graph`, et que les deux usages de `chm` dans `df` font décroître la mesure. En fait, la terminaison de `df` nécessite `ltg (chm g t one) g`, ce qui est immédiat, et `ltg (chm g0 x two) g`, qui est satisfaite si `leg g0 g`. Ceci requiert d'avoir comme résultat du parcours non seulement un graphe, mais aussi le fait que ce graphe soit inférieur ou égal au graphe argument `g`. En Coq, un tel résultat a le type *existentiel dépendant de* `g` noté `{g' : graph | leg g' g}`. Alors, une fonction `df_aux`, auxiliaire de `df` avec un résultat de ce type, a elle-même le type Coq suivant :

```
Definition df_aux_type: graph -> Set:= fun g : graph => nat->{g': graph | leg g' g}.
```

La construction de `df_aux` commence par la preuve du théorème suivant, `df_aux_F`, qui sera un paramètre essentiel de l'*opérateur de point fixe* prédéfini `Fix` [2] :

```
Theorem df_aux_F: forall g : graph, (forall g' : graph, ltg g' g -> df_aux_type g')
  -> df_aux_type g.
Proof.
 unfold df_aux_type.
 refine (fun g F t =>
    match stop_dec g t with
      left _ => exist (fun g' => leg g' g) g _
    | right Hx =>
        let cg0 := F (chm g t one) _ (son g 0 t) in
        let g0 := match cg0 with exist g _ => g end in
          match F (chm g0 t two) _ (son g0 1 t)
```

```
            with exist g1 h1 => exist _ g1 _
          end
    end).
  ...
Defined.
```

Examinons rapidement cette écriture assez complexe. D'après la correspondance de Curry-Howard, une preuve de ce théorème est une *fonction* qui transforme un graphe `g`, une fonction `F` du type `(forall g' : graph, ltg g' g -> df_aux_type g')` et un entier naturel `t` en un graphe `g'` tel que `leg g' g`. Alors, la fonction(-preuve) attendue est obtenue par application de la tactique `refine` avec le terme paramètre `(fun ... end)` donnant le *squelette* de la fonction. Des marqueurs de place `_` correspondent aux sous-preuves demandées par Coq à propos de la terminaison et effectuées à partir de nos résultats précédents :

- Quand `stop g t`, le résultat est `exist (fun g' => leg g' g) g _`, où `exist` est le constructeur du type `{g':graph | leg g' g}` avec 3 paramètres : le prédicat `(fun g' => leg g' g)`, `g` fourni comme *témoin* et une preuve (à élaborer) de `leg g' g`, ici `leg g g`.
- Quand `~ stop g t`, le résultat est obtenu par deux appels récursifs emboîtés de `F`, d'abord sur `son g 0 t` après changement de la marque de `t` en 1, puis sur `son g0 1 t`, après changement de la marque en 2. Les arguments de `F` sont aussi bien des données usuelles que des preuves.

Alors `df_aux` est construite par une induction nœthérienne sur `graph` muni de `ltg` utilisant l'opérateur appelé `Fix`. Ce dernier a trois arguments : la preuve, nommée `wf_ltg`, que `ltg` est nœthérien, le type du résultat, `df_aux_type`, et le précédent théorème, `df_aux_F`. Finalement, `df` est obtenue en extrayant le *témoin* du résultat, c.-à-d. le graphe `g'` tel que `leg g' g` :

```
Definition df_aux: forall g : graph, nat -> {g' : graph | leg g' g}:=
   Fix wf_ltg df_aux_type df_aux_F.
Definition df(g:graph)(t:nat) : graph := match df_aux g t with exist g' _ => g' end.
```

Propriétés de la spécification en Coq D'abord, la *terminaison* de `df_aux` et de `df` est automatiquement assurée par leur construction. La plupart des autres propriétés de `df` sont obtenues par induction nœthérienne sur la définition de `df_aux` en utilisant des *récurseurs* prédéfinis. Ainsi, nous prouvons que `df` préserve l'*invariant* de `graph`, l'*existence* des sommets et les *fils* du graphe initial. Un résultat important est que `df` est *idempotente*. Enfin, nous retrouvons exactement la définition espérée de `df`, en prouvant l'*équation à point fixe* suivante, ce qui nécessite l'axiome classique d'*extentionnalité*:

```
Lemma df_idem: forall g t, df (df g t) t = df g t.
Theorem df_eqpf: forall (g:graph)(t:nat),
  df g t =
    if stop_dec g t then g
    else let g0 := df (chm g t one) (son g 0 t) in
         df (chm g0 t two) (son g 1 t).
```

4. Orbites et piles internes

Orbites Nous définissons d'abord dans un graphe `g` une fonction de *succession* pour les entiers naturels qui appartiendront à la *pile interne* de l'algorithme de Schorr-Waite :

hal-00665909, version 1 - 3 Feb 2012

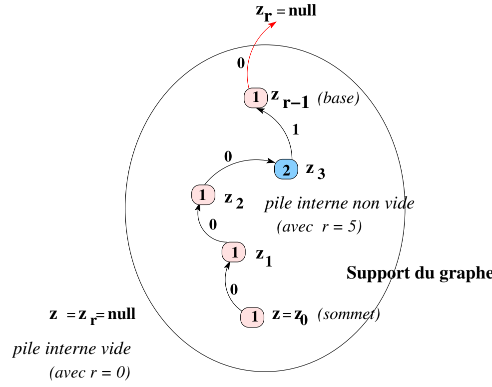

Figure 2: Aspects de piles internes.

```
Definition succ g z :=
   if eq_nat_dec z null then null
   else if eq_nat_dec (mark g z) 0 then null
        else son g (mark g z - 1) z.
```

Cette fonction (totale) peut être *itérée* depuis un entier naturel `z`, formant ce que nous appelons l'*orbite* de `z`. Cette notion a été définie et étudiée de manière générale [5]. Ici, nous rappelons seulement ce qui est utile pour la suite. Ainsi, pour tout entier naturel `k`, le `k`-ième itéré par `succ` depuis `z` est `zk := Iter (succ g) k z`, où `Iter` est la fonctionnelle classique d'itération. La suite (necessairement finie) `z0, ..., zk, ...`, tant que `zk` est dans `g` et est différent de ses prédécesseurs, est la `succ`-orbite de `z`. Sa longueur, définie par induction nœthérienne [4], est notée `lenorb g z`.

Pile interne Pour nous, la `succ`-orbite de `z` est une *pile interne* au graphe `g` si, pour `r := lenorb g z`, `zr := Iter (succ g) r z` et `zr_1 := Iter (succ g) (r - 1) z`, les deux conditions suivantes sont satisfaites : (*i*) `zr = null`; (*ii*) `0 < r -> 1 <= mark g zr_1 <= 2`. La Fig. 2 illustre cette notion : `r` est la *hauteur* de la pile interne, alors que `z (= z0)` et `zr_1` peuvent être vues comme son *sommet* et sa *base* quand elle n'est pas vide. La condition (*i*) dit qu'une orbite termine sur `null`, qui n'est pas un nœud de `g`, donc n'appartient pas à la pile. La condition (*ii*) impose que la marque de `zr_1` (si l'orbite est non vide) soit `1` ou `2`. En Coq, cette définition est captée par l'invariant appelé `inv_istack`. Une conséquence est que tous les éléments d'une pile interne sont de (véritables) nœuds de `g`, avec marque non nulle. Bien sûr, les orbites, et piles internes, peuvent être affectées par une mise à jour de marque par `chm`, ou de fils par `cha`. Les différents cas peuvent être étudiés de manière systématique, comme pour les structures algébriques linéaires [5]. Cependant, l'algorithme de Schorr-Waite utilise seulement des configurations particulières liées à trois opérations présentées maintenant.

Opérations sur les piles internes

• La *première* opération, `ipush`, empile un nœud `t` sur une pile interne de sommet `p`, après un changement de la marque de `t` en `one`, et renvoie un nouveau graphe (Fig. 3(a1)). Sa précondition impose que `t` soit bien un nœud du graphe avec une marque nulle :

```
Definition ipush g t p := cha (chm g t one) 0 t p.
```

Après `ipush g t p`, `p` est toujours le sommet d'une pile interne, mais `t` est également le sommet

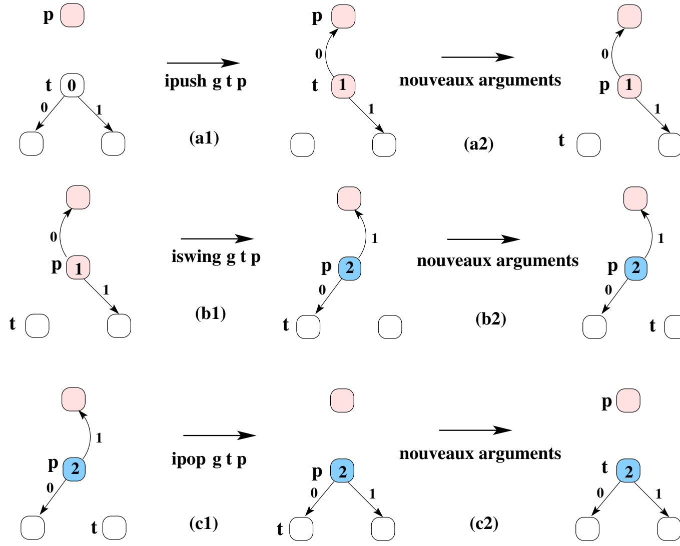

Figure 3: Opérations sur les piles internes.

d'une autre pile interne qui contient la première. L'opération utilise le fils gauche de `t` pour atteindre le successeur de `t`, c.-à-d. `p`, dans la nouvelle pile, `p` n'étant plus père de `t`. Tout ceci est prouvé formellement, notamment la préservation des invariants de graphe et de pile interne.

• La *deuxième* opération, `iswing`, est une sorte de *rotation* au sommet d'une pile interne `p` pour changer ses fils après que sa marque a muté de `one` (précondition) en `two`. Cette "opération de pile" est emblématique de l'algorithme de Schorr-Waite (Fig. 3(b1)) :

```
Definition iswing g t p := cha (cha (chm g p two) 0 p t) 1 p (succ g p).
```

En fait, elle remplace le fils gauche qui conduisait au successeur dans la pile interne par le fils droit, rétablissant le fils gauche initial de `p` en `t`, `p` n'étant plus père de son ex-fils droit. A nouveau, les invariants de graphe et de pile interne sont préservés, mais aussi la pile interne elle-même.

• La *troisième* opération, `ipop`, dépile d'une pile interne `p` son sommet (justement `p`), et rétablit son fils droit. Sa précondition impose que la marque de `p` soit `two` (Fig. 3(c1)) :

```
Definition ipop g t p := cha g 1 p t.
```

Après `ipop g t p`, `succ g p` est le sommet de la pile restante, dont la hauteur a diminué de 1 et qui peut être vide. Noter que les préconditions de `iswing` et de `ipop` entraînent `exv g p`. Enfin, il est prouvé que les trois opérations conservent les nœuds du graphe, et que `ipush` et `iswing` ajoutent 1 à la somme des marques, alors que `ipop` la laisse inchangée.

5. Marquage en profondeur utilisant une pile interne

Produit cartésien Nous nommons `graphistack` le type des couples `(g,p)` composés d'un graphe `g` et d'une pile interne `p`, muni de la conjonction des invariants des types composants :

hal-00665909, version 1 - 3 Feb 2012

hal-00665909, version 1 - 3 Feb 2012


```
Definition graphistack := (graph * nat)%type.
Definition inv_graphistack(gp:graphistack) :=
   inv_graph (fst gp) /\ inv_istack (fst gp) (snd gp).
```

Nos précédents résultats permettent de prouver rapidement que l'invariant `inv_graphistack` est satisfait pour un graphe (bien formé) et une pile interne vide, et qu'il est préservé par chacune des trois opérations, `ipush`, `iswing` et `ipop`, pour les nouveaux graphe et pile interne appropriés.

Conception de l'algorithme L'algorithme que nous visons est à *récursivité simple* et *terminale*. Pour le paramètre `gp = (g,p)`, sa terminaison sera garantie par la décroissance stricte de la *mesure* `2 * mes g + lenorb g p` à chaque appel récursif impliquant une et une seule des opérations `ipush`, `iswing` et `ipop`. Nos précédents résultats entraînent cette décroissance et un *préordre strict nœthérien* convenable dans `graphistack` est `ltgip` :

```
Definition ltgip (gp' gp:graphistack) :=
  let (g',p') := gp' in let (g,p) := gp in
    2 * mes g'  +  lenorb g' p' < 2 * mes g + lenorb g p.
```

Nous reprenons alors la méthode adoptée pour `df` afin de définir notre nouvel algorithme récursif avec pile interne, nommé `dfi`. Puisque l'algorithme est simplement récursif, il n'est pas nécessaire d'introduire un préordre large. Cependant, les preuves de décroissance de mesure nécessitent l'invariant `inv_graphistack` à chaque appel récursif. Alors, nous introduisons le *sous-type* {`gp: graphistack | inv_graphistack gp`} pour le résultat de notre fonction auxiliaire, nommée `dfi_aux`, dont le type est `dfi_aux_type`. Nous complétons aussi le prédicat d'arrêt `stop` de `df` en `stopi` :

```
Definition dfi_aux_type: graphistack -> Set :=
   fun gp:graphistack =>
    (inv_graphistack gp -> nat -> {gp':graphistack | inv_graphistack gp'}).
Definition stopi p g t := p = null /\ stop g t.
```

Alors, l'algorithme s'arrête quand la pile `p` est vide, et `t` n'existe pas dans `g` ou a une marque non nulle. Ce prédicat est décidable et la fonction de test correspondante est `stopi_dec`. Comme prélude à la définition de la fonction `dfi_aux`, on a, comme pour `df_aux`, à prouver le théorème `dfi_aux_F` :

```
Theorem dfi_aux_F:
 forall gp : graphistack, (forall gp' : graphistack, ltgip gp' gp -> dfi_aux_type gp')
  -> dfi_aux_type gp.
```

La technique de preuve est la même, avec la tactique `refine`, où l'on doit fournir une fonction-preuve avec trois appels de `F` correspondant à `ipop`, `iswing`, et `ipush`, et de nouveaux arguments donnés en Fig. 3(a2,b2,c2) et visibles dans l'équation à point fixe qui suit. Alors, comme pour `df_aux` (Section 3), `dfi_aux` est obtenue par application du récurseur `Fix`. Enfin, `dfi` est la projection de `dfi_aux` sur la composante `graph` en partant d'une pile interne vide (`inv_graphistack_null g hg` est une preuve que `inv_graphistack` est satisfaite pour le vrai graphe `g` avec une pile interne vide) :

```
Definition dfi_aux : forall gp : graphistack,
     inv_graphistack gp -> nat -> {gp' : graphistack | inv_graphistack gp'} :=
  Fix wf_ltgip dfi_aux_type dfi_aux_F.
Definition dfi (g:graph)(hg:inv_graph g)(t:nat) : graph:=
  match dfi_aux (g,null) (inv_graphistack_null g hg) t with
    exist (g',s) _ => g'
  end.
```

hal-00665909, version 1 - 3 Feb 2012

Il faut noter que `dfi` garde une preuve `hg` en argument de `inv_graph g`. La *terminaison* de `dfi_aux` et `dfi` est automatiquement assurée par ces constructions. Une équation à point fixe est prouvée pour `dfi_aux` de la même façon que pour `df` (Section 3). Pour simplifier la présentation, nous remplaçons dans les appels récursifs les arguments de preuve par des marqueurs de place. En outre, cette écriture sera optimisée plus tard, au moment d'aller vers un langage de programmation usuel :

```
Theorem  dfi_aux_eqpf :
 forall (g: graph)(p : nat)(igp:  inv_graphistack (g,p))(t:nat),
  dfi_aux (g,p) igp t =
    match stopi_dec p g t with
       left _ =>  exist _ (g,p) igp
     | right nSt =>
          match stop_dec g t with
              left Cx => let s1 := son g 1 p in
                match eq_nat_dec (mark g p) 2 with
                  left Hm   => dfi_aux (ipop g t p, s1) _ p   (* pop *)
                | right nHm => dfi_aux (iswing g t p, p) _ s1 (* swing *)
                end
            | right nCx => let s0 := son g 0 t in
                               dfi_aux (ipush g t p, t) _ s0  (* push *)
           end
     end.
```

Correction totale de l'algorithme Nous voudrions prouver que `dfi` est la "même fonction" que `df`. A défaut, puisque `dfi` a un argument de preuve supplémentaire (pour `inv_graph g`), nous serons satisfaits si, pour les mêmes `g` et `t` en entrée, elles retournent le même graphe en sortie *quel que soit l'argument de preuve effectif* pour `dfi`. L'idée de cette preuve est d'introduire une nouvelle fonction, nommée `Iteristack`, avec un argument `gp:graphistack`. La pile interne de `gp` y est parcourue du sommet à la base et mise à jour par `ipop`, pour appliquer `iswing` au nœud en cours quand sa marque est 1 (ou différente de 2, puisque les marques dans une pile interne ne sont jamais nulles), et `df`, à la place `dfi`. La terminaison de `Iteristack` est garantie par la décroissance de la même mesure que pour `dfi_aux`. Pour alléger, nous montrons juste le squelette de la fonction, avec des marqueurs de place à remplacer par des preuves de la préservation de l'invariant `graphistack` :

```
Function Iteristack (gp:graphistack)(igp: inv_graphistack gp)(t:nat)
  {measure (fun gp:graphistack =>
   2 * mes (fst gp) + lenorb (fst gp) (snd gp)) gp}: graphistack :=
 match eq_nat_dec (snd gp) null with
   left _ =>   gp
 | right nH => let s1:= son (fst gp) 1 (snd gp) in
    match eq_nat_dec (mark (fst gp) (snd gp)) 2 with
      left H2 =>   Iteristack (ipop (fst gp) t (snd gp), s1) _ (snd gp)
    | right nH2 => Iteristack (df (iswing (fst gp) t (snd gp)) s1, (snd gp)) _ s1)
    end
 end.
```

Par induction nœthérienne sur `ltgip`, et usage de l'axiome classique d'*indifférence aux preuves*, il est établi qu'appliquer `Iteristack` à `gp` donne le même résultat que projeter `dfi_aux` sur sa première composante. Alors, partant d'une pile interne vide, on obtient le *résultat fondamental*, l'*identité* de `dfi` et `df` :

```
Theorem df_dfi_aux1 : forall (gp:graphistack)(h:inv_graphistack gp)(t:nat),
  Iteristack (df (fst gp) x, snd gp) (inv_graphistack_df gp t h) t =
      match dfi_aux gp h t with exist rgp _ => rgp end.
Theorem df_dfi : forall (g:graph)(H:inv_graph g)(t:nat), dfi g H t = df g t.
```

Ce théorème achève la preuve de *correction totale* du marquage en profondeur avec pile interne. Ses conséquences sont nombreuses, puisqu'il transpose immédiatement à `dfi` les bonnes propriétés de `df`. Mais nous voulons aller jusqu'à un programme impératif réaliste avec manipulation de cellules de mémoire et de pointeurs, où, comme d'habitude, la mémoire reste *implicite*. Plutôt que de partir de `dfi` en essayant d'éliminer toute mention du graphe `g`, nous préférons nous rapprocher davantage en Coq des notations de bas niveau habituelles avant les transformations ultimes à la main. Pour cela, il nous faut d'abord un modèle de mémoire chaînée.

6. Modèle de mémoire

Cellules et mémoire Notre modèle de mémoire a en ligne de mire l'algorithme de Schorr-Waite original. Les *cellules* de mémoire sont d'un unique type inductif, dont `mkcell` est le constructeur, et `val, s0, s1` des sélecteurs de champs, pour marque, fils gauche et droit. Alors, plutôt que de donner une axiomatique complexe de la gestion mémoire, qui peut s'avérer dangereuse (du point de vue de la consistance), nous avons trouvé sain et élégant de définir algébriquement un type de mémoire `Mem` :

```
Record cell:Type:= mkcell {val : nat2; s0 : nat; s1 : nat}.
Inductive Mem:Type:= init : Mem  | alloc : Mem -> nat -> cell -> Mem.
```

Les adresses sont simulées par des entiers naturels. Le constructeur `init` renvoie la mémoire vide, et le constructeur `alloc` une nouvelle mémoire obtenue à partir d'une ancienne en insérant, et initialisant, une cellule à une nouvelle adresse, durant une *allocation*. Bien sûr, un modèle de mémoire plus réaliste pourrait capturer les subtilités d'un allocateur/désallocateur, particulièrement pour *séparer* des cellules de différents types [11]. De telles fonctionnalités sont importantes pour prouver la correction de compilateurs ou de programmes astucieusement intriqués [12]. Elles le sont moins pour notre objectif *dérivation de programmes bien structurés* manipulant des types de données standard, ici résoudre le problème de marquage sur un unique type de graphes.

Opérations en mémoire Dans notre modèle, il est facile de définir un prédicat pour tester si une adresse est *valide*, c.-à-d. correspond à une cellule allouée. Alors, les fonctions usuelles, `load`, `free` et `mut`, respectivement pour récupérer depuis une adresse le contenu d'une cellule, pour libérer une cellule (et son adresse) et pour changer le contenu d'une cellule d'après son adresse, sont facilement définies par filtrage, par ex. :

```
Fixpoint load(M:Mem)(z:nat):cell :=
  match M with
      init => initcell
    | alloc M0 a c => if eq_nat_dec a z then c else load M0 z
  end.
```

Les allocations sont consistantes seulement sous la précondition suivante, ce qui conduit à un *invariant* sur `Mem` pour les mémoires *bien formées* :

```
Definition prec_alloc M a := ~exm M a /\ a <> undef /\ a <> null.
Fixpoint inv_Mem(M:Mem): Prop :=
  match M with init => True | alloc M0 a c => inv_Mem M0 /\ prec_alloc M0 a
end.
```


hal-00665909, version 1 - 3 Feb 2012

Plutôt que de fournir soi-même l'adresse, on peut utiliser cette allocation pour simuler mieux le comportement d'un allocateur C qui renvoie une adresse fraîche [4]. Beaucoup de propriétés décrivant le comportement de ces opérations sont prouvées par induction sur `Mem`. Maintenant, nous examinons comment transposer les mémoires en graphes et inversement.

7. Mémoire vers graphe, graphe vers mémoire

Abstraction et représentation Les structures de graphe et de mémoire sont très proches. Pour sauter de l'une à l'autre, nous définissons deux opérations, `Abs`, pour *abstraction*, et `Rep`, pour *représentation*, dont la *réversibilité* est confirmée par deux théorèmes :

```
Fixpoint Abs(M:Mem): graph :=
  match M with
     init => vg | alloc M0 a c => iv (Abs M0) a (val c) (s0 c) (s1 c)
  end.
Fixpoint Rep (g:graph) : Mem :=
  match g with
     vg => init | iv g0 x m x0 x1 => alloc (Rep g0) x (mkcell m x0 x1)
  end.
Theorem Rep_Abs : forall M, Rep (Abs M) = M.
Theorem Abs_Rep : forall g, Abs (Rep g) = g.
```

Transposition des opérations et propriétés On peut dire que `graph` et `Mem` sont des structures algébriques *isomorphes* par `Abs` et `Rep`. Alors, toutes les opérations de `graph` peuvent être définies pour `Mem` et leurs propriétés transportées automatiquement par `Abs` et `Rep`. Dans la suite, les noms des opérations et propriétés dans `Mem` sont ceux de `graph` précédés par "R", p.ex. pour `chm` et `Rchm` :

```
Lemma Rchm_chm : forall M x m, Rchm M x m = Rep (chm (Abs M) x m).
Lemma chm_Rchm : forall g x m, chm g x m = Abs (Rchm (Rep g) x m).
```

Pour les piles internes, nous définissons `Rsucc`, `Rlenorb`, `inv_Ristack` et récupérons immédiatement leur comportement dans `Mem`, similaire à celui dans `graph`. Ceci conduit aux différentes versions de marquage en profondeur dans une mémoire.

8. Marquage en profondeur dans une mémoire

Spécification du marquage dans une mémoire La condition d'arrêt `stop` et le préordre `ltg` deviennent `Rstop` et `Rltg` pour les mémoires. Tous les lemmes que nous avions pour définir `df` sont transposés pour spécifier le marquage récursif (emboîté) `Rdf` dans une mémoire, avec :

```
Theorem df_Rdf : forall g t, df g t = Abs (Rdf (Rep g) t).
Theorem Rdf_df : forall M t, Rdf M t = Rep (df (Abs M) t).
```

En conséquence, toutes les propriétés de `df` dans `graph` sont transposables à `Rdf` dans `Mem`, d'où l'équation à point fixe, sorte de *spécification* pour l'algorithme de Schorr-Waite "dans une mémoire" :

```
Theorem Rdf_eqpf : forall (M:Mem)(t:nat),
  Rdf M t =
      if Rstop_dec M t then M
      else let M0 := Rdf (Rchm M t one) (Rson M 0 t) in
           Rdf (Rchm M0 t two) (Rson M 1 t).
```


hal-00665909, version 1 - 3 Feb 2012

hal-00665909, version 1 - 3 Feb 2012

Marquage en profondeur avec pile interne Les opérations `ipush`, `iswing` et `ipop` sont facilement transposées pour `Mem` en `Ripush`, `Riswing` and `Ripop` avec les mêmes propriétés. Alors, la contrepartie de `graphistack` est `Memistack` avec l'invariant `inv_Memistack` :

```
Definition Memistack := (Mem * nat) %type.
Definition inv_Memistack(Mp:Memistack) :=
   inv_Mem (fst Mp) /\ inv_Ristack (fst Mp) (snd Mp).
```

A la condition `stopi` et à `ltgip` correspondent `Rstopi` et `Rltgip`, qui est nœthérienne. Alors, la définition des fonctions `Rdfi_aux` et `Rdfi` avec le théorème `Rdfi_aux_F` suit la même construction que pour `dfi_aux` et `dfi`, dont elles sont les transpositions exactes dans `Mem`.

Correction totale Bien sûr, `Rdfi_aux` et `Rdfi` sont à *terminaison finie*, et, par notre isomorphisme, nous transposons dans `Mem` la preuve de correction de `dfi` par rapport à `df` en une preuve de correction de `Rdfi` par rapport à `dfi`. Mieux, toujours grâce à notre isomorphisme, nous avons le théorème qui établit la *correction* de `Rdfi` en mémoire par rapport à la spécification `df` dans un graphe :

```
Theorem Rdfi_dfi : forall (M : Mem) (hM : inv_Mem M) (t : nat),
   Rdfi M hM t = Rep (dfi (Abs M) (inv_graph_Abs M hM) t).
Theorem Rdfi_df : forall (M : Mem) (hM : inv_Mem M) (t : nat),
   Rdfi M hM t = Rep (df (Abs M) t).
```

9. Vers la programmation concrète

Extraction en OCaml L'outil d'*extraction de programme* fonctionnel de Coq appliqué à `Rdfi` conduit à la version OCaml de types et de fonctions auxiliaires, que nous ne montrons pas ici, et, après quelques manipulations à la main élémentaires, au programme suivant pour `rdfi_aux` et `rdfi` (en OCaml, `rstopi_dec` et `rstop_dec` sont traduites en fonctions booléennes, et "R" et "M" sont en minuscules) :

```
let rec rdfi_aux m p t =
  if rstopi_dec p m t then (m, p)
  else if rstop_dec m t
       then if eq_nat_dec (rmark m p) (S (S O))
            then rdfi_aux (ripop m t p) (rson m (S O) p) p
            else rdfi_aux (riswing m t p) p (rson m (S O) p)
       else rdfi_aux (ripush m t p) x (rson m O t)
let rdfi m t = fst (rdfi_aux m null t)
```

Comme d'habitude, l'extraction supprime tous les termes de preuve et ne retient que les données usuelles. C'est la forme fonctionnelle de l'algorithme de Schorr-Waite. Comme `rdfi_aux` est *récursive terminale*, elle sera facilement rendue *itérative* ("dérécursivée") sans utiliser de pile (autre qu'interne).

Programme de Schorr-Waite en C D'abord, nous devons définir en C les types des cellules et des adresses, jusqu'ici simulées par des entiers (nous supprimons le type `nat2` pour simplifier) :

```
typedef struct scell
  { nat val; struct scell * s0; struct scell * s1;} cell, * address;
```

Dans la suite, `null` est implanté en C par `NULL`. On pourrait faire de même pour `undef`, mais on verra qu'il n'est en général pas nécessaire de s'en inquiéter. Comme d'habitude, dans toutes nos fonctions C, la mémoire est *implicite*. D'abord, le paramètre `m` est supprimé de tous les *observateurs* de mémoire.

hal-00665909, version 1 - 3 Feb 2012

Par exemple, `rson m 1 t` devient `rson 1 t`, et même `t->s1` en dépliant l'implantation de `rson`, et le prédicat `rstop m t` devient la fonction booléenne `rstop t`.

L'argument mémoire est également supprimé des *mutateurs*, qui doivent modifier la mémoire implicite, ce qui peut être exprimé par des affectations. Dans `riswing m g t p` par ex., `rchm m t two`, `rcha m 0 p t` et `rcha m 1 p (rson m 0 p)` sont implantés en C respectivement `t->val = 2;`, `p->s0 = t;` et `p->s1 = rson 0 p;` (ou même `p->s1 = p->s0;`). Cependant, toujours pour `riswing`, quand on change le triplet de paramètres `(m, p, t)` en `(riswing m t p = cha (cha (chm m t two) p 0 t) p 1 (rson m 0 p), p, rson m 1 p)` (Fig. 3(a2,b2,c2)), on doit faire attention à la *sérialisation* correcte des instructions correspondantes pour éviter l'usage de valeurs écrasées.

Cette situation est habituellement formalisée par un *graphe de dépendance* entre variables ou affectations. Alors, la solution générale est d'introduire des *variables auxiliaires* pour éviter les circuits et rendre possible un *tri topologique*. Dans l'exemple, poser juste `q = p->s0;` conduit à la séquence C convenablement ordonnée :

```
   p->val = 2; q = p->s0; p->s0 = t; t = p->s1; p->s1 = q;
```

Le même type de raisonnement s'applique à `ripush` et `ripop`. Enfin, une question importante est la traduction de `exm m t` cachée dans `rstop m t`. Deux situations sont envisageables : *Cas (i)* : le langage-cible offre la possibilité (peu commune) de tester si, oui ou non, une adresse `t` est *valide*, c.-à-d. correspond à une cellule allouée. Alors, `exm t` (avec `m` implicite) peut être conservée comme dans le programme OCaml ci-dessus. C'est une généralisation intéressante de l'algorithme de Schorr-Waite ordinaire ; *Cas (ii)* : le langage-cible ne permet pas le test de validité. Alors, une hypothèse doit être faite pour éviter les *erreurs de segmentation* : "*Toute adresse atteinte par la fonction de marquage est valide ou* `NULL`." En fait, c'est la situation ordinaire quand on implante l'algorithme de Schorr-Waite : le programme échoue en cas d'erreur de segmentation.

Maintenant, nous supposons être dans la situation *(ii)*. Alors, nous pouvons, comme d'habitude, remplacer le test `exm t` par `t != NULL`, et la traduction de `undef` est indifférente. Finalement, en rassemblant tous les éléments ci-dessus, la procédure itérative C de Schorr-Waite s'écrit (après dépliage des tests booléens et de `rmark`) :

```
void rdfi(address t) {
   address p = NULL, q;
   while (!(p == NULL && (t == NULL || t->val != 0))) {
     if(t == NULL || t->val != 0) {
       if(p->val == 2) {q = p->s1; p->s1 = t; t = p; p = q;}          /* pop */
       else {p->val = 2; q = p->s0; p->s0 = t; t = p->s1; p->s1 = q;} /* swing */
     }
     else {t->val = 1; q = t->s0; t->s0 = p; p = t; t = q;}           /* push */
   }
 }
```

On doit noter que ce programme fonctionne correctement *quel que soit le marquage initial*, la situation usuelle, où toutes les marques sont 0, étant juste un cas particulier. C'est une variante de la version de [10], où tout nœud est étiqueté par deux booléens : la marque et l'indication du nœud en cours de traitement, gauche ou droit. Nous aurions pu adopter ici la même convention de codage.

10. Retour à la spécification, marquage avec pile externe

On peut trouver que la fonction initiale de spécification `df` est de trop haut niveau par rapport à la relation d'*accessibilité* évoquée au début et souvent utilisée dans la littérature, sous une forme moins

hal-00665909, version 1 - 3 Feb 2012

générale. L'accessibilité, au sens de la Section 3, peut être captée par la définition récursive suivante, où `reachable g t z` signifie que, dans le graphe `g`, `z` peut être atteint depuis `t` :

```
Fixpoint reachable(g:graph)(t z:nat): Prop : match g with
       vg => False
    |  iv g0 x m x0 x1 => reachable g0 t z \/ nat2_to_nat m = 0 /\
           (x = t /\ x = z \/ (x = t \/ reachable g0 t x)
             /\ (x0 = z \/ reachable g0 x0 z \/ x1 = z \/ reachable g0 x1 z))
  end.
```

Sous certaines conditions simples sur les paramètres, on prouve que la relation binaire `reachable g` est *réflexive* et *transitive*. Il faudrait mettre en rapport `reachable` et `df`. Mais les preuves directes que nous avons tentées n'ont pas abouti à cause de la récursivité double de `df`. Alors, nous sommes passés par une version du marquage simplement récursif utilisant une *pile externe* de nœuds, qui a l'avantage de laisser intacts les arcs du graphe pendant tout le processus. La fonction correspondante, appelée `dfs`, est construite en tous points comme `dfi`, avec les opérations `push`, `swing` et `pop`, et jouit des mêmes propriétés, prouvées de manière identique. En réalité, c'est par elle que nous avons commencé, pour mieux comprendre `dfi`. Par ex., nous avons l'*identité* avec `df` :

```
Theorem df_dfs : forall (g:graph)(hg: inv_graph g)(t:nat), df g t = dfs g hg t.
```

Après un assez long développement mettant en jeu l'*accessibilité à droite d'une pile*, le théorème suivant *caractérise complètement* l'effet de `dfs`, et donc de `df`, sur tous les nœuds du graphe `g`, et entraîne aussi la *correction* de `dfi`, et `Rdfi`, version "mémoire" du marquage, *vis-à-vis de l'accessibilité*:

```
Theorem reachable_dfs : forall g t z (hg: inv_graph g),
  mark (dfs g t hg) z =
    if reachable_dec g t z then if stop_dec g z then mark g z else 2 else mark g z.
```

11. Conclusion

Nous avons dérivé l'algorithme de Schorr-Waite et prouvé sa correction totale en Coq par une approche basée sur les "spécifications algébriques" plutôt que sur la "logique de Hoare". Nous transformons successivement des fonctions à partir d'une spécification sur un type abstrait. Nous réutilisons la notion d'orbite pour y caractériser une structure interne. Nous mimons en Coq la mémoire, et prouvons un isomorphisme entre structure abstraite et mémoire, pour obtenir une fonction récursive terminale. Finalement, nous transformons celle-ci "à la main" en une procédure impérative itérative en C. Nous utilisons exclusivement le système Coq, avec comme seuls ajouts d'axiomes l'*indifférence aux preuves* et l'*extentionnalité*. Le développement total en Coq représente 11 000 lignes, avec 500 définitions, lemmes ou théorèmes, le problème de marquage proprement dit faisant environ les 3/4 de ce volume. Aujourd'hui, il est difficile de voir ce qui pourrait être complètement automatique dans ces preuves.

Malgré les difficultés bien connues du problème, la gestion mémoire y est encore simple puisque sans allocation ni désallocation. Nos travaux futurs considéreront notre domaine usuel : la spécification formelle des *modeleurs géométriques* interactifs. Des structures astucieuses y décrivent la topologie des objets, par ex. des hypercartes combinatoires [8]. Prouver des propriétés ou la correction d'algorithmes fonctionnels géométriques a été notre grande préoccupation, comme dans la triangulation de Delaunay [6]. Cependant, deux questions lancinantes sont : (*i*) Comment combler la distance entre nos descriptions formelles (en Coq) et de vrais programmes (en C, C++) dans ces modeleurs ? (*ii*) Comment considérer formellement les résultats approchés, dus aux réels flottants, à la place des calculs exacts que nous utilisons jusqu'ici ? Une réponse globale aux deux questions conduira à un cadre sain pour spécifier, prouver et développer des programmes corrects en modélisation géométrique.

hal-00665909, version 1 - 3 Feb 2012

Remerciements

Nous remercions tous les membres du projet ANR blanc Galapagos (2008-2011), notamment Yves Bertot et Nicolas Magaud, pour toutes nos discussions sur les spécifications et preuves en Coq, et les rapporteurs, pour leurs nombreuses corrections et remarques constructives.

Bibliographie

[1] J.-R. Abrial. Event Based Sequential Program Development: Application to Constructing a Pointer Program. In *13th Symp. FME'2003, LNCS 2805, Springer*, pages 51-74, 2003

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

[3] R. Bornat. Proving Pointer Programs in Hoare Logic. In *5th Conf. on Mathematics of Program Construction, MPC'00, LNCS 1837, Springer*, pages 102–126, 2000.

[4] J.-F. Dufourd. *Schorr-Waite Coq Development In-line Documentation*. `http://dpt-info.u-strasbg.fr/~jfd/SW.html`, 2011.

[5] J.-F. Dufourd. Verifying Linked Representations of Linear Algebraic Data Types in Coq. *Submitted*, 43 pages, 2011.

[6] J.-F. Dufourd and Y. Bertot. Formal Study of Plane Delaunay Triangulation. In *Interactive Theorem Proving, ITP'2010, LNCS 6172, Springer*, pages 211–226, 2010.

[7] M. Giorgino et al. Verification of the Schorr-Waite algorithm - From trees to graphs. In *20th Logic Based Prog. Synth. and Transf., LOPSTR'2010, LNCS 5464, Springer*, pages 67–83, 2010.

[8] G. Gonthier. Formal Proof - The Four-Color Theorem. *Notices of the American Mathematical Society*, 55(11):1382–1393, 2008.

[9] D. Gries. The Schorr-Waite Graph Marking Algorithm. *Acta Informatica*, 11:223–232, 1979.

[10] T. Hubert and C. Marché. A case study of C source code verification; the Schorr-Waite algorithm. In *3rd IEEE Int. Conf. on Software Eng. and Formal Methods, SEFM'05*, pages 190–199, 2005.

[11] B.W. Kernighan and D.M. Ritchie. *The C Programming Language (1st ed.)*. Prentice-Hall, 1978.

[12] X. Leroy and S. Blazy. Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations. *Journal of Automated Reasoning*, 41(1):1–31, 2008.

[13] A. Loginov, T. Reps, and M. Sagiv. Automatic Verification of the Deutsch-Schorr-Waite Tree Traversal Algorithm. In *13th SAS, 2006, LNCS 4134, Springer*, pages 261–274, 2006.

[14] F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 199(1-2):200–227, 2005.

[15] H. Schorr and W.R. Waite. An efficient machine independent procedure for garbage collection in various list structures. *Comm. of the ACM*, 10(8):501–506, 1967.

[16] R.E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Comp. J.*, 1(2):146–160, 1972.

[17] M. Ward. Derivation of Data Intensive Algorithms by Formal Transformation. *IEEE Trans. on Software Eng.*, 22(9):665–686, 1996.

hal-00665909, version 1 - 3 Feb 2012