Vol.2011-SE-171 No.26
2011/3/15

証明支援系を用いた Morris の二分木走査アルゴリズムの検証

山田一宏[†1]　渡部卓雄[†1]
森口草介[†1]　西崎真也[†1]

我々は Morris のアルゴリズムの C による実装の検証を試みた．このアルゴリズムは再帰やスタックを用いずに二分木の走査を行うものであり，節点内部に余分なタグも不要であるという特徴がある．本研究はポインタ操作を伴う C プログラムへの検証支援ツール適用のケーススタディであり，今回は検証支援系 Caduceus および Why，自動証明系 Simplify および対話的証明支援系 Coq を使用した．本稿では，これらによる検証手法とその結果の概要を述べる．

Verifying C Source Code using Proof Assistant Tools — A Case Study with Morris' Tree Traversal Algorithm —

KAZUHIRO YAMADA ,[†1] TAKUO WATANABE ,[†1]
SOSUKE MORIGUCHI [†1] and SHIN-YA NISHIZAKI[†1]

We proved the partial correctness of a C implementation of Morris's tree traversal algorithm. The algorithm is known as a recursion-free, stack-free and tag-free binary tree traversal. This work is intended to be a case study of verifying C programs with pointer manipulation using existing verification tools such as Caduceus, Why, Simplify and Coq. In this report, we describe the outline of the verification methodology and results.

1. はじめに

ポインタ操作を伴うプログラムの検証例題として，Deutche-Schorr-Waite の二分木（グラフ）走査アルゴリズム（DSW アルゴリズム）がよく用いられている．このアルゴリズムはポインタ反転をおこなうことで再帰やスタックによらずに木の走査を行うものであり，その検証には適切なメモリモデルを必要とする．例えば Hubert らは component-as-array と呼ばれる手法を用いてメモリモデルを構成した[3]．一方湯浅らはポインタによる構造を様相論理のモデルと見なしている[7]．また Loginov らは 3 値論理にもとづくモデル化を行っている[4]，

しかしこれら各種の手法を他のプログラムに適用した事例はそれほど多くない．我々は，Hubert らの手法[3] を Morris の二分木走査アルゴリズムの C による実装に適用し，その正当性の機械的検証を行った．以下，検証対象，検証手法および結果の概要を述べる．

†1 東京工業大学・大学院情報理工学研究科・計算工学専攻
Department of Computer Science, Tokyo Institute of Technology

```
 1 typedef struct bintree_node {
 2     val_t val;
 3     struct bintree_node *tl, *tr;
 4 } *bintree;
 5
 6 void visit_node(bintree node);
 7
 8 void traverse(bintree t) {
 9     while (t != NULL) {
10         if (t->tl == NULL) {
11             visit_node(t);
12             t = t->tr;
13         }
14         else {
15             bintree rm = t->tl;
16             while (rm->tr != NULL && rm->tr != t)
17                 rm = rm->tr;
18             // rm is the rightmost node of the left subtree of t
19             if (rm->tr == NULL) { // rm does not have a thread
20                 rm->tr = t;       // link thread
21                 t = t->tl;
22             }
23             else {                // rm has a thread to t
24                 visit_node(t);
25                 rm->tr = NULL;    // unlink thread
26                 t = t->tr;
27             }
28         }
29     }
30 }
```

図 1　Morris の二分木走査アルゴリズム

2. 検証対象: Morris の二分木走査アルゴリズム

検証の対象は，Morris による二分木走査アルゴリズム[6] の C による実装 (図 1) である．こ

Vol.2011-SE-171 No.26
2011/3/15

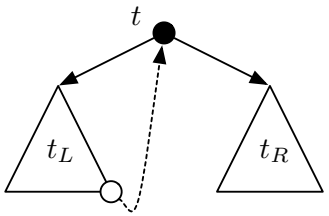


図 2 糸 (thread)

のアルゴリズムは二分木を通りがけ順 (in-order) で走査する．特徴として，再帰やスタックを必要としないこと，および節点の構造体内にポインタ変更を表すフラグ等の余分なフィールドが不要であることが挙げられる．また，このアルゴリズムは実行中にポインタを変更するため，実行中は一時的に木が崩れるが，走査終了後には元の木が復元される．

二分木 t について，その根の左 (右) の子を根とする部分木を t の左 (右) 部分木と呼び，t_L (t_R) と表記する．木 t を通りがけ順で走査するためには t_L の走査後に t の根に戻る必要がある．そのために，本アルゴリズムでは t_L の最も右にある節点 (t_L を通りがけ順で走査したときに最後に訪れる節点) から t の根に戻るための辺 (図 2 の点線) を一時的に作成する．この辺は木を構成する本来の辺（木の枝）と区別するために**糸 (thread)** と呼ばれる．

糸付木 (threaded tree) と呼ばれるデータ構造の場合，木の枝とは区別できる形で糸が常時存在しており，糸をたどることで再帰やスタックを用いずに走査できる．一方，本アルゴリズムにおける糸はアルゴリズムの実行中に一時的に作られ，実行後は全て消去される．また，糸が出る節点は右の子を持たない という性質があるので，本アルゴリズムでは節点を表す構造体 (`bintree_node`) に糸のためのフィールドを追加せず，右の子のフィールド (`tr`) で代用している．このようにした場合，右の子への辺が木の枝か糸かを区別する必要が生じる．本アルゴリズムでは，t_L をその根から右方向に辿っていって t の根に到達すれば，t の根に入る辺は糸であるという性質を利用する (図 1 の 16〜17 行めのループおよび 23〜26 行目)．

本アルゴリズムにおける糸については以下のような性質が成り立つ．

(1) ある節点に入る糸は高々一つ．
(2) ある節点から出る糸は高々一つ．
(3) ある節点に糸が入っているとき，その右部分木内に糸が入る節点はない．
(4) ある節点に糸が入っていないとき，その左部分木内に糸が入る節点はない．

図 3 は本アルゴリズムの実行中に現れ得るデータ構造の例である．糸を点線で，糸が入って

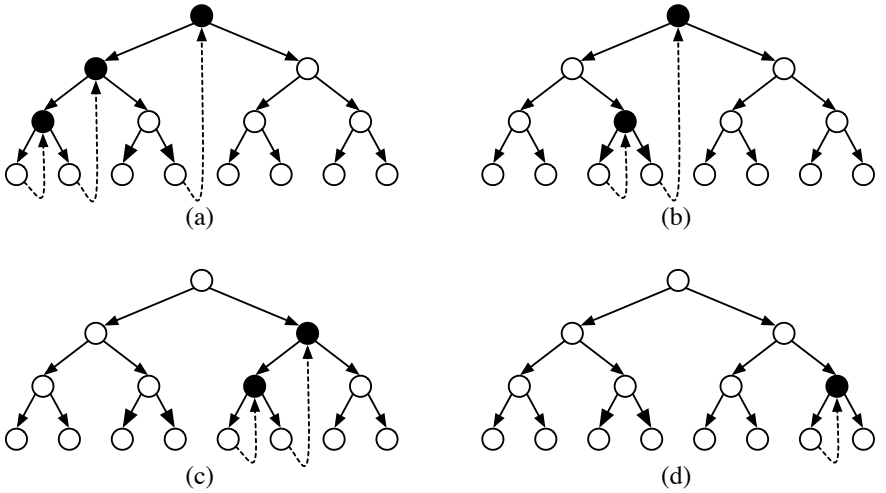


図 3 Morris のアルゴリズムの実行中に表れる構造の例

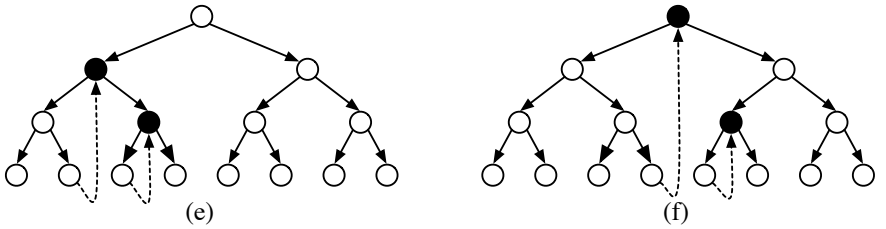


図 4 Morris のアルゴリズムの実行中には表れない構造の例

いる節点を黒丸で表している．一方図 4 は上記性質の (3) および (4) を満たしていないケースであり，本アルゴリズムの実行中にこのような構造は現れない．

3. 検証方針

プログラム 1 の部分正当性を，Hubert らによる DSW アルゴリズムの実装の検証[3] と同様の方針に従って検証した．まず検証すべき性質を事前・事後条件および不変条件として形式化し，ソースコード中にアノテーションとして記述する．ソースコードは C プログラム用の検証支援ツール Caduceus で処理される．Caduceus は Why プラットフォーム[2] に含まれるツールの一つであり，アノテーション付のソースコードから Why への入力を生成する．Why はその結果を用いてバックエンドとなる自動証明系および証明支援系用の検証条件を生成する．生成された検証条件の証明には，Why プラットフォームに含まれる自動証明系 Simplify と，対話的証明支援系 Coq[1] を使用した．

実際の検証にあたり，節点の構造体 `bintree_node` に走査済みを示すフラグ `m` と，`tr` が糸であることを示すフラグ `th` を追加し，これらのフラグを適切に設定するよう関数 `traverse` を修正した．証明すべき性質は Hubert らによるもの[3] と同様のものとした．具体的には，入力が正しい二分木であり，各節点のフラグが 0 で初期化されていることが事前条件であり，事後条件は，任意の節点についてその右の子は実行前と等しく，根から到達可能な全ての節点について `m` が 1 かつ `th` が 0 であること等である．

実際にソースコード中のアノテーションとして記述した事前・事後条件の一部を図 5 に，検証に利用したループ不変条件 (内側ループのみ) を図 6 に示す．実際に全ての事前・事後条件および不変条件を記述したソースコードは約 100 行になる．ここで用いられている `reachable` および `reachable_to_right` などの述語 (図 7) は，生成された検証条件を証明する際に証明支援系において定義する必要がある．

4. 検証結果

アノテーション付ソースコードから生成された検証条件は全部で 11 個であった．そのうち 2 個は自動証明系 Simplify によって証明できたので，残り 9 個を Coq で証明している．証明を行うにあたって，3 個の公理と 6 個の述語を定義した．

ポインタ操作を伴うプログラムを扱っているので，適切なメモリモデルを仮定する必要がある．ここでは Hubert らによる DSW アルゴリズムの検証[3] と同様に，component-as-array と呼ばれるモデルを採用した．これは図 8 に示すように構造体の配列を各フィールドを要素

```
/*@ requires
  @ (\forall bintree x;
  @      x != \null && reachable(t,x) => \valid(x))
  @ &&
  @ (\forall bintree x;
  @      x != \null && reachable(t,x) => !x->m && !x->th)
  @ &&
  @ (\forall bintree x;
  @      x != \null => reachable only(t,x))
  @ &&
  @ (\forall bintree x;
  @      x != \null => !reachable(x->lt,x) && !reachable(x->rt,x))
  @ &&
  @ (\forall bintree x;
  @      x != t => !reachable(x,t))
  @ &&
  @ (\forall bintree x;
  @      x != \null => (x->rt != x) && (x->lt != x))
  @ &&
  @ (\forall bintree x;
  @      rightmost_child_is_null(x))
  @ ensures
  @ (\forall bintree x; \old(x->rt) == x->rt)
  @ &&
  @ (\forall bintree x; x != \null &&
  @      reachable(\old(t),x) => x->m && !x->th)
  @ &&
  @ (\forall bintree x; !reachable(\old(t),x) =>
  @      x->m == \old(x->m) && x->th == \old(x->th))
  @*/
void traverse(bintree t) { ... }
```

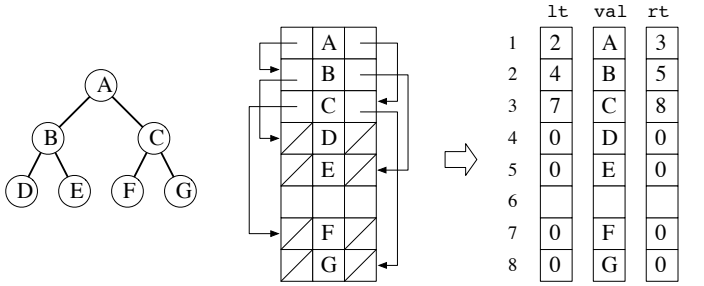
図 5 事前・事後条件

```
/*@ invariant
  @ (InvI_1 :: \forall bintree x;
  @  reachable_to_right(rm, x)
  @  => reachable_to_right(\old(rm), x))
  @ &&
  @ (InvI_2 :: rm != \null)
 */
while (rm->rt != NULL && rm->rt != t)
```

図 6 内側ループの不変条件

```
/*@ predicate reachable(bintree x1, bintree x2)
  @   eads x1->lt, x1->rt
  @
  @ predicate reachable_to_right(bintree x1, bintree x2)
  @   reads x1->rt
  @
  @ predicate rightmost_child_is_null(bintree x)
  @   reads x->th, x－>rt
  @
  @ predicate reachable_only(bintree x1, bintree x2)
  @   reads x1->lt, x1->rt
  @
  @ axiom reachable_from_itself:
  @   \forall bintree x; reachable(x,x)
  @*/
```

図 7 述語の定義



(a) 二分木とそのメモリ内での表現例　　(b) 3個の配列による表現

図 8 Component-as-Array モデル

とする配列で代用するものである．ポインタ値は配列のインデックス (自然数) になり，また各要素がスカラ値であるような一次元配列になるため扱いが容易になる．この方法では共用体や構造体内部でのポインタ演算等は扱えず，また動的なメモリ確保についても制限を受けるが，今回の目的には十分である．

Coq によって生成された検証条件を証明するにあたり，64 個の補題とその証明を行った．生成された検証条件とそれらの補題，およびその証明は全部で 1800 行ほどであり，証明には約 1ヶ月を要している．本稿執筆時点では外側のループ不変条件の維持に関する検証条件に一部未完のものがあるが，他は証明済みである．

5. 考　　察

実際の検証作業では，アノテーション (特にループ不変条件) を修正しつつ行うことが多く，修正後に再度検証条件を生成し，それらを証明支援系で証明するといった作業の繰り返しになる．一般に修正前の証明は再利用できないため，作業にかかる時間は大きくなる．例えば Mateti らが示したような諸性質[5] を最初から用いることで，停止性を含めたより完全な検証結果を得られることが期待できる．

参 考 文 献

1) Coq. http://coq.inria.fr/.
2) Why. http://why.lri.fr.
3) T.Hubert and C.Marché. A case study of C source code verification: the Schorr-Waite algorithm. In *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)*, pp. 190–199. IEEE Computer Society, 2005.
4) A.Loginov, T.Reps, and M.Sagiv. Automated verification of the Deutsch-Schorr-Waite tree-traversal algorithm. In *Static Analysis (SAS 2006)*, Vol. 4134 of *Lecture Notes in Computer Science*, pp. 261—279. Springer-Verlag, 2006.
5) P.Mateti and R.Manghirmalani. Morris' tree traversal algorithm reconsidered. *Science of Computer Programming*, 11(1):29–43, 1988.
6) J.M. Morris. Traversing binary trees simply and cheaply. *Information Processing Letters*, 9(5):197–200, 1979.
7) Y.Yuasa, Y.Tanabe, T.Sekizawa, and K.Takahashi. Verification of the Deutsch-Schorr-Waite marking algorithm with modal logic. In *Verified Software: Theories, Tools, Experiments (VSTTE 2008)*, Vol. 5295 of *Lecture Notes in Computer Science*, pp. 115—129, 2008.