日本ソフトウェア科学会第 27 回大会 (2010 年度) 講演論文集

経路列挙手続きを用いたプログラム検証法

服部 哲

プログラムのホワイトボックステストで用いる実行経路の列挙手続きによって，プログラムの検証ができることを示す．プログラムにおける実行経路は，逐次的に実行される代入文と判定される条件式の列で表す．経路を評価して代入文を実行すると，プログラムの入力変数に関する条件式と，プログラムを実行後の変数への値割付けとが求まる．検証は，前者を事前条件，後者を事後条件として，仕様との整合性を調べることで行う．なお，ループを含むプログラムについては，ループ不変条件とループ終了条件とが代入文で与えられることを仮定している．この仮定の下で，経路列挙手続き中では，ホーア論理で言う検証条件の証明が必要ない．ループ不変条件については，一つの例題を取り，容易に与えられる場合を考察する．ループ内のプログラムを差分方程式に変換し，差分方程式が求解できる場合である．

1 はじめに

著者は，論文 [4] [5] において，数式処理システム Mathematica によるテストケース生成手続きの実現例を 3 つ示した．ブラックボックス法のうち，同値分割法と限界値分析のそれぞれに基づくものと，ホワイトボックス法に基づくものである．ホワイトボックス法に基づく手続きは，プログラムにおける実行経路を列挙するものである．

本論文では，プログラムにおける実行経路を列挙する手続きを，プログラムの検証に用いることができることを示す．手続きでは，プログラムにおける実行経路の候補を，逐次的に実行される代入文の式と判定される条件式の列で表す．プログラムに対する経路列挙手続きの出力は，プログラムにおける実行経路の候補の全てである．経路の候補は，条件式が充足可能であれば，実際に経路となる．

「事前条件 P を仮定してプログラム S を実行したとき，事後条件 Q が成立する」ことをプログラムの部分正当性の表明と言い，$\{P\}S\{Q\}$ と記す．ホーア論理は，プログラムの部分正当性を証明する体系である [1] [2] [6] [7]．ホーア論理の言葉で言えば，プログラムに対する経路列挙手続きは，事前条件を恒真として，プログラムを先頭から末尾へとたどり，最強事後条件を求める手続きであると言える．本論文では，そのように事前条件，事後条件を設定したとき，経路列挙手続きが，ホーア論理によるプログラムの部分正当性の証明手続きとなっていることを示す．証明のためには，何らかの帰納的な構造が必要である．経路列挙手続きはいくつかの部分手続きから構成される．これらの部分手続きの呼び出し関係が木構造を作るので，これを証明に利用する．

実際のプログラム検証は，事前条件と事後条件を経路ごとに分割したもので行う．一つの経路を評価して代入文を実行すると，プログラムの入力変数に関する条件式と，プログラムを実行後の変数への値割付けとが求まる．検証は，前者をプログラムの事前条件，後者を事後条件として，仕様の事前条件，事後条件との整合性を調べることで行う．

なお，ループを含むプログラムについては，実行経路を表すために，ループ不変条件とループ終了条件と

A Program Verification Method Using Path Testing Procedure

Satoshi Hattori, 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, Graduate School of Information Science and Engineering, Tokyo Institute of Technology.

が代入文で与えられることを仮定している．この仮定の下で，経路列挙手続き中では，ホーア論理で言う「検証条件の証明」が必要ない．このことは，経路列挙手続きによる検証のユニークな点であると言える．

上記の通り，本論文の経路列挙手続きでは，ループ不変条件とループ終了条件とを代入文で与えなければならない．ループ不変条件を与えるのは一般には難しいこととされている．本論文では，ループ不変条件が容易に与えられる特別な場合を，一つの例題を取り上げて考察する．ループの 1 ステップを「前ステップの変数値から現ステップの変数値を定めるもの」と捉える．すると，ループ内のプログラムは数学における差分方程式（再帰方程式，漸化式）の一種とみなすことができる．もちろん，任意の差分方程式が求解可能というわけではないので，ループ内のプログラムの任意の形式に対してループ不変条件が与えられるわけではない．ループ内には一般に複数の変数があることから，連立差分方程式を考える必要がある.「前ステップの変数値から現ステップの変数値を定める」という点では，一階差分方程式を考えればよい．連立一階差分方程式は，線形であれば，求解が可能である．本論文では，このように差分方程式が求解可能な場合のループ不変条件を考える．そして，ループ不変条件を代入文で表して，プログラムの実行経路を表現する．

以下，まず, 2 節において, プログラムに対する経路列挙手続きを導入する．次に，3 節で，経路列挙手続きが，ホーア論理によるプログラムの部分正当性の証明手続きとなっていることを示す．4 節では，一つの例題を取り上げ，差分方程式の求解によってループ不変条件を与えることと，ループ不変条件を用いた経路の表示について述べる．5 節で関連研究を紹介し，最後に，6 節で，今後の課題について述べる．

2 プログラムに対する経路列挙手続き

2.1 while プログラム

本論文ではプログラムのクラスを，ホーア論理で用いる while プログラムのクラスとする．while プログラムの構文定義を示す[1]．while プログラムは，以下で定義される最小の集合 P の要素である．

- (代入文) 任意の変数 x と任意の式 e に対し，$x := e \in P$.
- (複合文) $S_1, S_2 \in P$ に対し，$S_1; S_2 \in P$.
- (if 文) 任意のブール式 B と $S_1, S_2 \in P$ に対し，if B then S_1 else S_2 fi $\in P$.
- (while 文) 任意のブール式 B と $S_1 \in P$ に対し，while B do S_1 od $\in P$.

この構文定義から，while プログラムを実行する過程で出現する要素は，代入文と（if 文，while 文の）ブール式であることが分かる．一つの実行過程を表現するのにそのような要素の列を用いることにする．実行過程，または，実行過程に出現する要素の列を「実行経路」と呼ぶ．なお，正確には「実行経路の候補」と言うべきである．ブール式が充足不能なために実行できない場合があるからである．本節では，実行経路の候補を列挙する手続きを示す．手続きは，簡単に「経路列挙手続き」と呼ぶことにする．この手続きは，論文[4][5] で示したホワイトボックステストのための手続きと本質的に同じものである．

手続きの記述には，[4][5] と同様，Mathematica を用いる．まず，Mathematica での while プログラム表現法を，BNF 風表記を用いて下記の通り定める．

```
<Program> ::= <Statement>
<Statement> ::=
  <Assignment-statement> | <Block> |
  <If-statement> | <While-statement>
<Assignment-statement> ::=
  <Variable> "=" <Expression>
<Block> ::= "{ block," <Statement>
  { "," <Statement> } "}"
<If-statement> ::=  "{ if," <BooleanExp>
  "," <Statement> "," <Statement> "}"
<While-statement> ::=  "{ while,"
  <BooleanExp> "," <Statement> "}"
```

例えば，while プログラムにおける代入文を，Mathematica でも代入文として実行させる．そのため，while プログラムの代入文で用いる記号 := が，上記の Mathematica 表現では，= に替わっている．同様に，while プログラムにおける「等価性」の記号 = の替わりに，Mathematica では == を用いる．なお，

Mathematica に関する基本事項を付録 A に載せる．

2.2 経路列挙手続きのアルゴリズム

経路列挙手続きは，while プログラムの Mathematica 表現を入力とし，実行経路の集合をリストで出力する．一つの実行経路は，条件式（ブール式）と代入文を要素とするリストで表現する．リストにおける要素の順序は，経路に現れる順とする．実行経路がリストで表されるので，経路列挙手続きの出力はリストのリストである．なお,「経路のリスト」という表現はどちらのリストを指すのかあいまいである．要素に順序がある，条件式と代入文の集まりには「リスト」の語を用い，そのリストの集まりには「集合」の語を用いることにする．

while プログラムは，代入文，複合文，if 文，while 文から構成される．経路列挙手続きには，まず，4 種類の文を処理する手続き (`procstate`) を設ける．この手続きは，まず，文を入力したとき，それが上記 4 つのいずれであるかを判別する．代入文ならば，実行経路に代入文を付加する．それ以外の文ならば，文の種別に応じて，以下の 3 つの手続きのいずれかを呼び出す.

複合文に対する手続き (`procblock`): プログラム S に 2 個以上の文があるとき，ホーア論理の証明手続きでは，S をどこで区切って S_1, S_2 とするかは任意である．経路列挙手続きでは，最初の文を S_1 とし，残りのプログラムを S_2 とする．まず，S_1 に対して `procstate` を呼び出し，実行経路の集合 l_1 を得る．残りのプログラム S_2 が存在する場合のみ，次を行う．1 個以上の文をブロックと定義すれば，S_2 はブロックである．よって，S_2 に対して `procblock` を呼び出し，実行経路の集合 l_2 を得る．プログラム S に対する実行経路の集合を得るには，l_1 の任意の要素に l_2 の任意の要素を結合しなければならない．よって，S に対する実行経路の集合 l は次の式で与えられる．$l = \{Join(p_1, p_2) \mid p_1 \in l_1, p_2 \in l_2\}$．ここで，$Join(p_1, p_2)$ は，2 つのリスト p_1, p_2 を結合して得られるリストとする．

if 文に対する手続き (`procif`): 条件式 B が成立する場合に S_1 を実行し，成立しない場合に S_2 を実行する．S_1 に対して `procstate` を呼び出し，実行経路の集合 l_1 を得る．そして，l_1 の各リストの先頭に条件式 B を付加する．そのようにして得られる集合を l_1' とする．同様に，S_2 に対して `procstate` を呼び出し，実行経路の集合 l_2 を得る．そして，l_2 の各リストの先頭に条件式 $\neg B$ を付加する．そのようにして得られる集合を l_2' とする．if 文に対する手続きの出力は，集合 l_1' と l_2' の和集合である．

while 文に対する手続き (`procwhile`): ループ内の文 S_1 から得られるループ不変条件を inv とする．inv は 1 つの実行経路に相当するもので，リストで表現する．リスト inv の末尾に，条件 $\neg B$ を付加してリスト $path$ する．while 文に対する手続きの出力はリスト $path$ である．

2.3 経路列挙手続きの実現

経路列挙手続きの Mathematica による実現例を図 1 に示す．Mathematica における構文定義において，while プログラムの構成要素のうち，代入文だけがリスト構造を持たない．リスト長を返す関数 `Length` は，そのような式に対しては値 0 を返す．また，複合文，if 文，while 文は，それぞれ整数値 block(0), if(1), while(2) を先頭要素とし，後続要素が実質の文の要素となるリストである．これらのことから，手続き `procstate` を図 1 の通り定めている．なお，while 文を扱うための部分手続き `procwhile` については，実現の方針を 4 節で述べる．

ここで，while 文を含まないものあるが，論文 [4] [5] と同様に,「三角形問題」のプログラムを例に取り上げる．三角形問題とは，次のプログラムに対するテストケースを作成せよ，というものである [10].「このプログラムは，入力ダイアログから 3 つの整数を読む．この 3 つの値は，それぞれ三角形の 3 辺の長さを表すものとする．プログラムは，三角形が不等辺三角形，二等辺三角形，正三角形のうちのどれであるかを示すメッセージを表示する.」図 2 に，三角形問題のプログラムの while プログラムでの記述例を示す. 入力変数は `a`，`b`，`c`，出力変数は `r` である．出力について，`r = 100` は「3 つの値は三角形の 3 辺の長さでないこと」を，`r = 200` は「不等辺三角形」を，`r =`

```
block = 0; if = 1; while = 2;
procstate[l_] :=
  If[Length[l] > 1,
    Switch[First[l],
      block, procblock[Rest[l]],
      if, procif[Rest[l]],
      while, procwhile[Rest[l]]
    ],
    {{l}}
  ]
procblock[l_] :=
  If[Length[l] > 1,
    Distribute[{procstate[First[l]],
      procblock[Rest[l]]}, List],
    procstate[First[l]]
  ]
procif[l_] :=
  Join[Map[Join[{l[[1]]}, #] &,
           procstate[l[[2]]]],
       Map[Join[{! l[[1]]}, #] &,
           procstate[l[[3]]]]
  ]
```

図 1　経路列挙手続き procstate

300 は「二等辺三角形」を，r = 400 は「正三角形」をそれぞれ意味する．

図 3 に，while プログラムの Mathematica による表現を示すが，見易さのため，一部は省略してある．省略のないものを付録 B に載せる．図 3 において，例えば，sen110 = HoldForm[tmp = a] のように，式 tmp = a に関数 HoldForm が適用してある．Mathematica では，式 tmp = a を入力すると即座に評価が行われ，代入 tmp = a が実行される．このままでは，複数の実行経路を求める手続きにおいて，その代入のみが有効となってしまい，不具合が生じる．評価を経路ごとに行うために，関数 HoldForm を適用して，評価を保留しておく．

さて，while プログラム prog も文であるので，prog に対して経路列挙手続きを行うには，procstate[prog] を実行すればよい．すると，56 個の要素から成る実行経路の集合が出力される．各実行経路を見ると，再帰呼びの結果，リストのネストが生じている．そこで，関数 Flatten を適用して平坦化する．56 番目の実行経路 fpath56 を例示しておく．以下では，Mathematica によるプログラムとその実行結果を示す場合，実行結果はプログラムの下に破線で区切って示す．

```
if a > b then tmp := a; a := b; b := tmp
         else skip fi;
if b > c then tmp := b; b := c; c := tmp
         else skip fi;
if a > b then tmp := a; a := b; b := tmp
         else skip fi;
if a <= 0
 then r := 100
 else
  if a + b < c
   then r := 100
   else
    if a + b = c
     then r := 100
     else
      if a = b
       then
        if b = c then r := 400
                 else r := 300 fi
       else
        if b = c then r := 300
                 else r := 200 fi
      fi
    fi
  fi
fi
```

図 2　三角形問題の while プログラム

```
con1 = HoldForm[a > b]
sen110 = HoldForm[tmp = a]
sen111 = HoldForm[a = b]
sen112 = HoldForm[b = tmp]
       :
con9 = HoldForm[b == c]
sen91 = HoldForm[r = 300]
sen90 = HoldForm[r = 200]
prog =
{block,
 {if, con1,
  {block, sen110, sen111, sen112}, {}},
 {if, con2,
  {block, sen210, sen211, sen212}, {}},
 {if, con3,
  {block, sen310, sen311, sen312}, {}},
 {if, con4,
  sen41,
  {if, con5,
   sen51,
   {if, con6,
    sen61,
    {if, con7,
     {if, con8, sen81, sen80},
     {if, con9, sen91, sen90}
    }
   }
  }
 }
}
```

図 3　while プログラムの Mathematica 表現

```
paths = procstate[prog]
fpath56 = Flatten[paths[[56]]]
----------------------------------------
{! a > b, ! b > c, ! a > b,
 ! a <= 0, ! a + b < c, ! a + b == c,
 ! a == b, ! b == c, r = 200}
```

2.4　実行経路の評価とプログラムの事後条件

このようにして得られたリストを評価することで，経路上の代入文が実行される．実行経路の評価を行う手続き evalpath を図 4 に示す．関数 HoldForm で保留した評価は，それと対になる関数 ReleaseHold で実行することができる．リストの要素が代入文であれば，代入操作が実行される．リストの要素が条件式であれば，式に代入の結果が反映される．条件式の評価後の代入操作の結果も，条件式に反映される．評価した条件式は保存しておく．

```
evalpath[l_] := Module[{lis,i,tmp,exp},
  lis = {};
  For[i = 1, i <= Length[l], i++,
    tmp = l[[i]];
    exp = ReleaseHold[l[[i]]];
    If[tmp[[1, 0]] =!= Set,
      AppendTo[lis, exp]
    ]
  ];
  Return[lis]
]
```

図 4　実行経路を評価する手続き evalpath

手続き evalpath において，リスト lis に条件式を保存している．評価後の式が代入文であるか条件式であるかは，評価前の式の先頭の記号によって判定する．Set であれば代入文，さもなくば条件文である．

実行経路の評価に関して一点補足しておく．例題のプログラムには変数のスワップをするブロックがある．例えば，tmp = a; a = b; b = tmp である (表記は Mathematica のものを用いている)．Mathematica では，a，b に値が代入されていない状況下ではスワップは行われない．変数 a が変数 b の別名である，という結果になる．この状況を回避するために，a = a0; b = b0; c = c0 という仮の値の代入を行っておく．すると，スワップが行われ，変数 a，b の値はそれぞ

れ b0，a0 という結果が得られる．上記の「仮の値の代入」は，プログラム内で更新される変数 a，b，c と，プログラムの入力変数 a0，b0，c0 とを区別するという意味も持つ．なお，この方法と，経路を評価する関数 evalpath を与えたことによって，論文 [4] [5] で挙げていた「変数の名前換えの問題」は解決した．

さて，先に得た経路 fpath56 を evalpath で評価し，条件式のリスト cond56 を結果に得る．リスト cond56 中の条件式の充足可能性を関数 FindInstance で調べる．

```
v = {a = a0, b = b0, c = c0}
cond56 = evalpath[fpath56]
FindInstance[cond56, v, Integers]
-----------------------------------------
{a0 <= b0, b0 <= c0, a0 <= b0,
 a0 > 0, a0 + b0 >= c0, a0 + b0 != c0,
 a0 != b0, b0 != c0}
{{a0 -> 4, b0 -> 6, c0 -> 8}}
```

リスト cond56 中の条件式は充足可能である．この経路はテストケース a0 = 4，b0 = 6，c0 = 8 によって実行可能である [4] [5].

Myers が言うように，テストケースには，入力に対する出力の情報も含めなければならない [10]．経路を実行したときの事後条件は，評価後の条件式と各変数への値割付けである．後者は変数値の表示で得られる．経路 fpath56 に対して，出力変数 r の値は 200 であると分かる．

```
postvalue = {a, b, c, r}
-----------------------------------------
{a0, b0, c0, 200}
```

最後に，56 個の実行経路の候補全てについて，条件式の充足可能性を調べるプログラムを記しておく．結果，実際の実行経路の数は 31 個である．

```
paths = procstate[prog]
fpaths = Map[Flatten[#] &, paths]
conds = {}
For[i = 1, i <= Length[fpaths], i++,
  v = {a = a0, b = b0, c = c0};
  AppendTo[conds, evalpath[fpaths[[i]]]]
]
tcases = Map[FindInstance[#, v, Integers]&,
  conds]
```

3 ホーア論理と経路列挙手続き

本節では，経路列挙手続きが，ホーア論理によるプログラムの正当性証明手続きとなっていることを示す．

ホーア論理については，文献 [1] [2] [6] に基づく．while プログラム S が事前条件 P と事後条件 Q に対して部分的に正当であることを，$\{P\}S\{Q\}$ と記す．この表明は，事前条件 P を満たすときにプログラム S を実行して終了した場合，事後条件 Q を満たすことを意味する．

ホーア論理は，表明 $\{P\}S\{Q\}$ を証明するための体系の 1 つである．この体系は 1 つの公理と 4 つの推論規則から成る．以下において，記法 $Q[e/x]$ は，式 Q における変数 x のすべての出現を式 e に置換えて得られる式を表す．

- (A1) 代入文の公理
$$\{Q[e/x]\}x := e\{Q\}$$
- (R1) 複合文の規則
$$\frac{\{P\}S_1\{R\}\ \{R\}S_2\{Q\}}{\{P\}S_1; S_2\{Q\}}$$
- (R2)if 文の規則
$$\frac{\{P \wedge B\}S_1\{Q\}\ \{P \wedge \neg B\}S_2\{Q\}}{\{P\}\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}\{Q\}}$$
- (R3)while 文の規則
$$\frac{\{P \wedge B\}S_1\{P\}}{\{P\}\text{while } B \text{ do } S_1 \text{ od}\{P \wedge \neg B\}}$$
- (R4) 帰結の規則
$$\frac{P \Rightarrow P_1\ \{P_1\}S\{Q_1\}\ Q_1 \Rightarrow Q}{\{P\}S\{Q\}}$$

次の補題を証明する．

補題 1 *while* プログラム S に対し，事前条件 P の成立を仮定する．その下で，S に対して経路列挙手続きを実行したときの事後条件を Q とする．このとき，表明 $\{P\}S\{Q\}$ が成立する．

証明の前に，次の点を補足しておく．前節では，「経路上にある代入文の式と条件式の生成」の手続きと「経路の評価による代入の実行」の手続きとを別々に与えた．しかし，代入文が生成された時点で代入が実

行されるとみなせば，経路上の全ての式が生成された時点でプログラムの実行が完了するとみなせる．よって，上記の補題も前者の手続き，すなわち，経路列挙手続きのみを対象としている．

さて，補題の証明のためには，帰納的な構造が必要である．ここでは，経路列挙手続きにおける部分手続きの呼び出し関係が木構造を成すことを利用する．

定義 1 経路列挙手続きにおける手続き呼び出しに関する木 T は，節点の集合と辺の集合とから成る．任意の節点は一つのラベルを持つ．ラベルは statement，assignment，block，if，while のいずれかである．任意の節点はそのラベルに応じて，0 個，または，1 個，または，2 個の子節点を持つ．

- statement: 1 個の子節点を持つ．ラベルは，assignment，block，if，while のいずれかである．
- assignment: 0 個の子節点を持つ．
- block: 1 個または 2 個の子節点を持つ．1 個の子節点の場合，ラベルは statement である．2 個の子節点の場合，ラベルは，左の子節点が statement，右の子節点が block である．
- if: 2 個の子節点を持つ．ラベルは，2 つとも statement である．
- while: 0 個の子節点を持つ．

経路列挙手続きでは，代入文の処理を行う部分手続きを設けなかった．手続き `procstate` 中で代入文の処理を行う部分を手続き `procassign` と名づけ，`procstate` から呼び出されるものとみなす．

手続き呼び出しに関する木 T の節点は，各部分手続きが呼び出されたときに生成される．手続き `procstate`，`procassign`，`procblock`，`procif`，`procwhile` が呼び出されたとき，生成される節点のラベルはそれぞれ，statement，assignment，block，if，while である．

節点が生成されたとき，呼び出した親手続きの節点との間に，木 T の枝が生成される．

補題 1 の証明を行う．

(証明) 手続き呼び出しに関する木の高さに関する帰納法で証明する．

木の高さが 0 のとき，木は子節点を持たない節点 1 つから成る．この節点のラベルは，assignment または while であるので，場合分けする．

- assignment: プログラム S は代入 $x := e$ である．事前条件 P に対し代入 $x := e$ を行うと事後条件 $Q = P[e/x]$ が得られる．Q は P のインスタンスであるので，$P \Rightarrow Q$ が成立する．よって，$\{P\}S\{Q\}$ である．
- while: プログラムが $S =$ while B do S_1 od であるとする．この while 文のループ不変条件を P とすると，$\{P \wedge B\}S_1\{B\}$ が成立する．(R3)while 文の規則から，$\{P\}$while B do S_1 od$\{P \wedge \neg B\}$ を得る．

次に，木の高さが 1 以上のとき，高さが $n-1$ までの木についての成立を仮定して，高さが n の木に対して証明する．木の根節点のラベルは，statement または block または if であるので，場合分けする．

- statement: 子節点は，assignment，block，if，while のいずれかである．子節点を根節点とする部分木の高さは $n-1$ であるから，帰納法の仮定より，部分木に対して補題が成立する．よって，statement を根とする木に対しても成立する．
- block: プログラム $S = S_1; S_2$ について，S_1 が 1 つの文，S_2 が 1 つ以上の文から成るブロックであるとする．S_2 が存在しない場合も証明は同様である．事前条件 P が成立していると仮定する．木の根節点は 2 個の子節点を持つ．ラベルは，左の子節点が statement，右の子節点が block である．左の子節点を根とする部分木について，対応するプログラムは S_1 である．根から任意の葉節点 $e_i (1 \leq i \leq n)$ への経路について，帰納法の仮定より，事後条件 R_i が存在して，$\{P\}S_1\{R_i\}$ が成立する．$R = R_1 \vee \cdots \vee R_n$ とおく．(R4) 帰結の規則により，任意の葉節点 n_i への経路について $\{P\}S_1\{R\}$ が成立する．次に，右の子節点を根とする部分木について，対応するプログラムは S_2 である．根から任意の葉節点 $n'_j (1 \leq j \leq m)$ への経路について，帰納法の仮定より，事後条件 Q_j が存在して，$\{R\}S_2\{Q_j\}$ が成立する．$Q = Q_1 \vee \cdots \vee Q_m$ とおく．(R4) 帰結の規則により，任意の葉節点 n'_j への経路について $\{R\}S_2\{Q\}$ が成立する．以上と，(R1) 複

合文の規則により，$\{P\}S_1; S_2\{Q\}$ が成立する．

- if: プログラムが $S =$ if B then S_1 else S_2 fi であるとする．事前条件 $P = P \wedge (B \vee \neg B)$ が成立していると仮定する．木の根節点は，ラベルが statement である 2 個の子節点を持つ．左の子節点を根とする部分木について，対応するプログラムを S_1 とする．根から任意の葉節点 $e_i(1 \leq i \leq n)$ への経路について，帰納法の仮定より，事後条件 R_i が存在して，$\{P \wedge B\}S_1\{R_i\}$ が成立する．$R = R_1 \vee \cdots \vee R_n$ とおく．(R4) 帰結の規則により，任意の葉節点 n_i への経路について $\{P \wedge B\}S_1\{R\}$ が成立する．次に，右の子節点を根とする部分木について，対応するプログラムを S_2 とする．根から任意の葉節点 $n'_j(1 \leq j \leq m)$ への経路について，帰納法の仮定より，事後条件 R'_j が存在して，$\{P \wedge \neg B\}S_2\{R'_j\}$ が成立する．$R' = R'_1 \vee \cdots \vee R'_m$ とおく．(R4) 帰結の規則により，任意の葉節点 n'_j への経路について $\{P \wedge \neg B\}S_2\{R'\}$ が成立する．ここで，$Q = R \vee R'$ とおく．(R4) 帰結の規則により，$\{P \wedge B\}S_1\{Q\}$ と $\{P \wedge \neg B\}S_2\{Q\}$ を得る．(R2)if 文の規則により，$\{P\}$if B then S_1 else S_2 fi$\{Q\}$ が成立する．

□

経路列挙手続きをホワイトボックステストに用いるという前提では，プログラム全体の事前条件は「恒真」である．言い換えると，経路列挙手続きでは，プログラムに対する「全ての入力値」を考え，経路によって入力値を場合分けしているのである．補題 1 の事前条件 P として恒真 $(True)$ を与えると，次の定理が得られる．

定理 1 *while* プログラム S に対して経路列挙手続きを実行したときの事後条件を Q とする．このとき，表明 $\{True\}S\{Q\}$ が成立する．

経路列挙手続きで得られる経路の数を N として，各経路を評価した後に得られる条件式のリストをそれぞれ $C_1, \cdots, C_N$ とする．なお，各リスト中の条件式は全て入力変数によって表されている．経路は if 文によって分岐するので，入力変数への任意の値割付けに対し，$C_1, \cdots, C_N$ のうちの高々一つが充足可能である．また，$C_1 \vee \cdots \vee C_N = True$ である．一方，残りの事後条件である変数への値割付けを，各経路に対して $Q_1, \cdots, Q_N$ とする．条件 $C_1, \cdots, C_N$ は事後条件として得たが，これを事前条件とすれば，次が定理 1 の系として得られる．

系 1 *while* プログラム S に対して，経路列挙手続きで得られる経路の数を N とする．各経路を評価した後に得られる条件式のリストをそれぞれ $C_1, \cdots, C_N$ とする．また，評価後の変数への値割付けをそれぞれ $Q_1, \cdots, Q_N$ とする．このとき，任意の $i(1 \leq i \leq N)$ に対し，$\{C_i\}S\{C_i \wedge Q_i\}$ が成立する．

系 1 を用いて，前節の三角形問題の例を，プログラム `prog` の検証という観点から見てみる．56 番目の経路について，経路を評価した後の条件式のリスト `cond56` は，下記の通りであった．

```
{a0 <= b0, b0 <= c0, a0 <= b0,
 a0 > 0, a0 + b0 >= c0, a0 + b0 != c0,
 a0 != b0, b0 != c0}
```

また，出力変数に関する値割付けは，`r = 200` であった．この経路の事後条件をこれで代表させる．系 1 より，事前条件 `cond56` が成立するとき，プログラム `prog` を実行すると，事後条件は `r = 200` となる．全ての経路について，仕様の事前条件，事後条件との整合性を調べれば，プログラム `prog` の検証ができる．

4 差分方程式の解を用いたループ不変条件

本節では，経路列挙手続きの部分手続きで，while 文を扱うための `procwhile` の実現の一つの方針を示す．ただし，この方針では，任意の while ループに対してループ不変条件を与えられるわけではない．ループ内のプログラムが特定の条件を満たす場合に与えることができる．Mathematica で差分方程式の求解ができることを利用し，ループ内のプログラムを差分方程式に変換してループ不変条件を与える．

変数 $x, y, z, \cdots$ について，ループの第 k ステップ終了後の変数値を $x[k], y[k], z[k], \cdots$ とする．同様に，第 $k+1$ ステップ終了後の変数値を $x[k+1], y[k+1], z[k+1], \cdots$ とする．$x[k+1], y[k+1], z[k+1], \cdots$ のそれぞれを $x[k], y[k], z[k], \cdots$ によって表せば，連

立一階差分方程式となる．

連立一階差分方程式の一般項は，例えば，方程式が線形であり，かつ，非同次項が変数 k の多項式である場合に求めることができる [13]．また，while プログラムのループ不変条件を得るためには，一般項 $x[k], y[k], z[k], \cdots$ の式から変数 k が消去できて，$x, y, z, \cdots$ の関係式を得なければならない．この関係式がループ不変条件となる．

例題として，次の while プログラムを取り上げる (一部，文献 [1] などとは異なる記号を用いている)．

```
f := 1;
i := 1;
while i != n + 1 do
  f := 2 * f;
  i := i + 1
od
```

ループ内のプログラムを差分方程式 {f[k+1] == 2 f[k], i[k+1] == i[k] + 1} とみなす．Mathematica の関数 RSolve によって差分方程式を解く．

```
ans = RSolve[
  {f[k+1] == 2 f[k], i[k+1] == i[k] + 1},
  {f[k], i[k]}, k]
-------------------------------------------
{{f[k] -> 2^(-1+k) C[1], i[k] -> k + C[2]}}
```

任意定数 C[1]，C[2] を含む形で差分方程式の一般解が示された．Mathematica で任意定数の表示に用いられる C[1]，C[2]，C[3], ... は変数として用いることができないので，便宜上，C[1]，C[2] を変数 a，b に改める．次に，変数 k を消去してループ不変条件を求める．方程式の解は複数存在する場合があるので，上記の出力も「解のリスト」で表現されている (ただし，この場合は解は 1 個である)．リストから要素を取り出して，さらに，変数 f，i，k に関する方程式の形に改める．そして，変数 k を消去する．すると，ループ不変条件 $f == 2^{-1-b+i}a$ が得られる．

```
ansab = ans /. {C[1] -> a, C[2] -> b}
ans1 = ansab[[1]]
anseqn = {f == f[k], i == i[k]} /. ans1
inv = Reduce[anseqn, k]
-------------------------------------------
{{f[k] -> 2^(-1+k) a, i[k] -> b + k}}
{f[k] -> 2^(-1+k) a, i[k] -> b + k}
{f == 2^(-1+k) a, i == b + k}
f == 2^(-1-b+i) a && k == -b + i
```

次は，ループ不変条件を用いて実行経路の表示を行う．ループの前に成立している f == 1，i == 1 は，ループ不変条件のインスタンスでなければならない．このことと，ループの前のステップ数 k == 0 とから，得られたループ不変条件の任意定数 a，b の値が求まる．a，b の値を求めるために，a，b それぞれを f，i，k の式で表しておく．すると，代入 f = 1，i = 1，k = 0 によって，a，b の値が求まる．

```
inva = Reduce[anseqn, a]
invb = Reduce[anseqn, b]
-------------------------------------------
2^k != 0 && i == b + k && a == 2^(1-k) f
f == 2^(-1+k) a && b == i - k
```

ここまでに，ループの前に成立している式と，ループ不変条件とを考慮した．残りは，ループ終了条件である．式 i != n + 1 の否定 i == n + 1 を，代入文 i = n + 1 に変えて追加する．以上より，例の while プログラムにおける経路 path が次の通り得られる．経路中の式 f = .，i = . は，変数 f，i についてループ前に成立している式 (ループ不変条件のインスタンス) をクリアして，ループ不変条件を設定するために用いている．

```
path = {HoldForm[f = 1], HoldForm[i = 1],
HoldForm[k = 0],
HoldForm[a = 2^(1-k) f], HoldForm[b = i-k],
HoldForm[f = .], HoldForm[i = .],
HoldForm[f = 2^(-1-b+i) a],
HoldForm[i = n + 1]}
```

この経路 path について，本論文で示した経路を評価する手続き evalpath を適用する．すると，$f == 2^n$ が得られる．

```
evalpath[path]
```

```
postvalue = {f, i}
------------------------------------------
{2^n, 1 + n}
```

本節の例題に対して行った特徴的なことは，ループ不変条件を容易に与えられる場合を考えたことと，ループ不変条件，および，ループ終了条件を代入文で表したことである．前節の三角形問題の例で見た通り，while 文を含まなければ，プログラムを先頭から末尾へとたどり，最強事後条件を求めていくことで，プログラムの正当性証明が可能である．この際，ホーア論理で言う「検証条件の証明」は必要ない．while 文を含む場合にも，同様に証明をするにはどうすればよいかを考察したのが本節である．

通常は，ホーア論理の証明手続き中で，与えたループ不変条件の正しさを証明する必要がある．多くの場合，それは検証条件の証明によって行われる．ホーア論理の証明手続きの例で，検証条件のうちの 2 つにおいて，ループ不変条件の帰納法による証明の基底と帰納段階が行われているものをよく見かける．

ループ内のプログラムを差分方程式とみなし，差分方程式の解をループ不変条件とする．このとき，任意定数を含む一般解を得ることは，ループ不変条件の帰納法による証明の帰納段階を行うことと対応する．ループ前に成立している条件は，ループ不変条件のインスタンスである．この事実に基づき，一般解の任意定数を決定する作業は，帰納法による証明の基底と対応する．

また，本節の例では，ループのブール式を，`i <= n` とするのが自然であるものを，`i != n + 1` とした．これは，ループ終了条件 (ブール式の否定) が等式となるように意図したものである．ループ終了条件が等式であれば，それを代入文に変えて，実行経路の表示に用いることができる．

5 関連研究

ホーア論理によるプログラムの正当性証明の自動化に関しては，多くの研究が行われている．ホーア論理による証明では，導出された検証条件を数学的に証明することが必要になる．その用途に Mathematica を用いることは自然である．例えば，Mathematica をベースにして構築された定理証明系 Theorema [14] によるプログラム正当性証明に関する研究が行われている [9].

論文 [9] においても，ループ不変条件の導出には差分方程式の求解を用いている．ただし，証明の方法は最弱事前条件に基づくもので，プログラムを後向きにたどって，ホーア論理の推論規則を適用していく方法である．そして，導出される検証条件を Theorema で証明している．

本論文では，プログラムを前向きにたどるアプローチを取った．ループ不変条件の正しさの帰納段階の証明を差分方程式の一般解の求解に置き換えるとともに，ループ不変条件に関する基底の証明を一般解の任意定数の決定に置き換えた．このことは，本論文と論文 [9] の相違点である．

論文 [11] [12] なども，ループ不変条件の数学的な自動生成について述べている．

ESC/Java2 は簡易形式手法のツールの一つであり，Java プログラムに対して最弱事前条件に基づく検証ができる [3]．論文 [8] は，ESC/Java2 による検証で用いるループ不変条件の，ユーザが記述するアサーションからの自動生成について述べている．

6 まとめと今後の課題

本論文では，プログラムに対する経路列挙手続きによって，プログラムの検証ができることを示した．ループ不変条件とループ終了条件とが代入文で与えられるという仮定の下で，経路列挙手続き中では，ホーア論理で言う検証条件の証明が必要ない．このことは，経路列挙手続きによる検証のユニークな点であると言える．ループ不変条件の与え方については，ループ内のプログラムを差分方程式に変換し，差分方程式が求解できる場合を考察した．

次の 2 点が今後の課題である．差分方程式を解いてループ不変条件を与える手続きの実現と，扱える while プログラムのクラスの定式化である．手続きの実現については，例題で行った手順を自動化する予定である．ループのステップ数を表す差分方程式の変数 `k` がうまく消去できるかどうか，などが問題になりそ

うである．この件については，同じく差分方程式の求解を用いている論文 [9] を参考にしたい．

while プログラムのクラスの定式化については，まず，解ける差分方程式のクラスを明らかにしなければならない．本論文では解けるクラスとして，連立一階線形差分方程式を取り上げた．Mathematica の関数 `RSolve` は，線形でない差分方程式も扱える [15]．論文 [9] [11] [12] なども参考にして，解ける差分方程式のクラスの調査を行いたい．もう一つ，本論文では，ループ終了条件を代入文で与えるために，ループのブール式の形を「等式の否定」に限定した．これは自然とは言えないので,「等式および不等式」を扱えるようにしなければならない．その場合，ループ終了条件を代入文で与えるにはどうすればよいかが課題となる．

参 考 文 献

[1] Apt, K.R.: Ten Years of Hoare's Logic: A Survey - Part I, ACM Transactions on Programming Languages and Systems, Vol.3, No.4, (1981), pp.431–483.

[2] 荒木 啓二郎, 張 漢明: プログラム仕様記述論, オーム社 (2002).

[3] Cok, D.R. and Kiniry, J.R.: ESC/Java2: Uniting ESC/Java and JML, International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS 2004) (2004), pp.108–128.

[4] 服部 哲: ソフトウェアテストケース生成手続きの数式処理システム上での実現, 日本ソフトウェア科学会第 26 回大会論文集, 6D-3 (2009), pp.1–20.

[5] Hattori, S.: Computer Algebra System as Test Generation System, The IEICE Transactions on Information and Systems, Special Section on Formal Approach, Vol.93, No.5 (2010), pp.1006–1017.

[6] 林 晋: プログラム検証論, 情報数学講座, Vol.8, 共立出版 (1995).

[7] Hoare, C.A.R.: An Axiomatic Basis for Computer Programming, Communications of the ACM, Vol.12 No.10 (1969), pp.576–580.

[8] Janota, M.: Assertion-based Loop Invariant Generation, Proc. of the 1st International Workshop on Invariant Generation (WING 2007) (2007), pp.15–26.

[9] Kovacs, L. and Jebelean, T.: Automated Generation of Loop Invariants by Recurrence Solving in Theorema, Proc. of the 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2004) (2004), pp.451–464.

[10] Myers, G.J., Badgett, T., Thomas, T.M. and Sandler, C.: ソフトウェア・テストの技法 第 2 版, 長尾 真監訳, 松尾 正信訳, 近代科学社 (2006).

[11] Rodriguez-Carbonell, E. and Kapur, D.: Automatic Generation of Polynomial Loop Invariants: Algebraic Foundations, Proc. of the 2004 International Symposium on Symbolic and Algebraic Computation (ISSAC 2004) (2004), pp.266–273.

[12] Sankaranarayanan, S., Sipma, H.B. and Manna, Z.: Non-linear Loop Invariant Generation using Gröbner Bases, Proc. of the 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2004) (2004), pp.318–329.

[13] 高橋 健人: 差分方程式, 培風館 (1961).

[14] The Theorema Project. http://www.theorema.org

[15] Wolfram Resarch: Wolfram Mathematica 7 ドキュメントセンター. http://reference.wolfram.com/mathematica/guide/Mathematica.html

[16] Wolfram, S.: *The Mathematica Book, Fifth Edition*, Wolfram Media/Cambridge University Press (2003).

A Mathematica に関する準備

Mathematica に関する基本事項をいくつか述べておく．以下は著者の使用する Mathematica Ver.5.2 [16] に基づいた記述である．現在の最新版 Ver.7 を用いる場合は，開発元の Wolfram Research が Web でも公開しているマニュアル [15] を参照されたい．Mathematica で基本となるオブジェクトは「式」である．以下で，式を表す変数として例えば `expr` を，複数の箇所で用いたり，異なる型で用いるたりするので注意されたい．Mathematica は基本的に対話式のインタプリタとして用いる．入力した式は，Shift+Enter で評価する．

A.1 Mathematica における関数定義

まず，Mathematica における関数定義について説明する．Mathematica の組込み関数の名前は，英語の大文字から始まる．逆に，ユーザ定義の関数の名前は，小文字から始めなければならない．ユーザ定義関数の一例を示す．`x[a_, b_, c_] = a + b > c && b + c > a && c + a > b`．Mathematica では，関数を定義する際，引数や戻り値の型を指定する必要はない．上記では，`a`，`b`，`c` の実引数として整数を与えても実数を与えてもよい．関数 `x[a_, b_, c_]` の戻り値は自動的にブール値 `True`，`False` となる．また，関数の引数はブラケット [,] で囲む．関数左辺の

引数が a_ のようになっているが，通常のプログラミング言語同様，実引数に任意の式を与えるためにはこのようにする必要がある．関数左辺の引数を単に a とした場合は，有効な実引数は変数 a のみとなる．

複数の式の逐次評価を行いたいときは，各式をセミコロンで区切って入力する．関数の右辺で複数の式を記述する場合は，セミコロンで区切られた式の全体を括弧 (,) で囲む．関数の右辺で局所変数 x，y，... を用いたいときには，Module[{x, y, ...}, expr] のようにする．この場合も，expr は複数の式でもよく，各式はセミコロンで区切られる．関数の戻り値は，Return[expr] で設定する．関数を遅延評価型で定義したいときは，左辺と右辺を結ぶ記号として = の代わりに := を用いる．なお，= や := は「割当て」の意味で使うのに対し，== は「等価性の判定」の意味で使う．

A.2 制御構造

次に，制御構造について述べる．If 文は，If[condition, t, f] の形式を取る．condition を評価した結果が True のとき式 t を，False のとき式 f を，それぞれ評価して返す．第 3 引数はなくても構わない．Switch 文は，Switch[expr, form1, value1, form2, value2, ...] の形式を取る．expr を評価した結果について，左から順に調べて最初にマッチする formi の次の引数 valuei を評価して返す．For 文は，For[start, test, incr, body] の形式を取る．start は初期設定，test は終了条件，incr は 1 ループ実行後の動作，body は 1 ループの処理内容である．

A.3 リストとその操作

Mathematica ではデータ構造としてリストを主に用いる．リストでは要素をカンマで区切って列挙し，それらをブレース {, } で囲む．リスト {e1, e2, ...} は，List[e1, e2, ...] と表現してもよい．AppendTo[expr, elem] は，リスト expr へ要素 elem を追加し，Join[e1, e2] は，2 つのリスト e1, e2 を結合する．Flatten[expr] は，リスト expr におけるネストを平坦化する．

リスト expr の第 i 要素は，expr[[i]] で示される．expr の要素数は，Length[expr] で与えられる．通常のプログラミング言語での配列の添字は 0 から始まるが，Mathematica のリストの要素番号は 1 から始まることに注意する．そのため，For 文でリスト expr を処理する際は，要素番号を i として，For[i = 1, i <= Length[expr], i++, ...] とするのが典型的である．

リスト中の各要素に対して何らかの操作を行うとき，上記のように For 文を用いてもよいが，関数型プログラミングの代表的な関数である Map を用いたほうが簡潔に記述できる．Map[f, expr] によって，リスト expr の各要素に関数 f を適用したリストが得られる．リストを要素とするリスト expr について，Distibute[expr, List] とすると，各リストから要素を 1 つずつ取り出して作られるリストの全ての組合せが得られる．例えば，Distribute[{{1, 2}, {3, 4}}, List] = {{1, 3}, {1, 4}, {2, 3}, {2, 4}} である．

リスト操作では，純関数と呼ばれる機能を多用する．純関数とは名前を与えない関数定義であり，ラムダ記法の Mathematica における表現法である．関数定義が body のとき，body & と表記し，body の中で形式パラメータ # を用いる．例えば，Map[# + 1 &, {1, 2, 3}] = {2, 3, 4} のように使われる．

A.4 方程式・不等式の求解

FindInstance[expr, vars, dom, n] は，方程式および不等式の連立系 expr を与えたとき，それを満たす変数のインスタンスを求める．そのような例が存在しない場合は，空集合を返す．この場合，方程式・不等式系には解が存在しない．方程式系は不定方程式系 (方程式の数が変数の数より少ない) でもよい．第 2 引数 vars はインスタンスを求める変数のリストである．第 3 引数 dom はオプションで，インスタンスの型 (Booleans，Integers，Reals，デフォルトは Complexes) を指定する．第 4 引数 n もオプションで，求めるインスタンスの個数 (デフォルトは 1) を指定する．Reduce[expr, vars] は，変数のリスト vars について方程式あるいは不等式 expr を解

く．RSolve[eqn, a[n], n] は，a[n] についての差分方程式 eqn を解く．

A.5 式の前置・中置記法，その他

And[e1, e2]，Or[e1, e2] は，2 式 e1，e2 のそれぞれ論理積，論理和を取る関数である．Not[expr] は expr の否定を返す．以上 3 つの関数は，e1 && e2, e1 || e2, !expr と表現してもよい．同様に，方程式・不等式 e1 == e2, e1 != e2, e1 >= e2, e1 > e2, e1 <= e2, e1 < e2 は，それぞれ，Equal[e1, e2]，Unequal[e1, e2]，GreaterEqual[e1, e2]，Greater[e1, e2]，LessEqual[e1, e2]，Less[e1, e2] と表現してもよい．式 e1，e2 の構文的な一致の検査には e1 === e2 を用いる．同様に，一致しないことの検査には e1 =!= e2 を用いる．

代入文 lhs = rhs，値割付けのクリア lhs = . は，それぞれ Set[lhs, rhs]，Unset[lhs] と表現できる．なお，本論文の 3 節までと 4 節で，同じ変数 a, b, i を用いている．実行して試される場合は 3 節の例の後，a=.; b=.; i=. を実行して変数値をクリアされたい．

lhs -> rhs は，式 lhs から式 rhs への変換規則である．expr /. rules は，式 expr に変換規則のリスト rules を適用する．

B 三角形問題の while プログラムの Mathematica による表現

```
con1 = HoldForm[a > b]
sen110 = HoldForm[tmp = a]
sen111 = HoldForm[a = b]
sen112 = HoldForm[b = tmp]
con2 = HoldForm[b > c]
sen210 = HoldForm[tmp = b]
sen211 = HoldForm[b = c]
sen212 = HoldForm[c = tmp]
con3 = HoldForm[a > b]
sen310 = HoldForm[tmp = a]
sen311 = HoldForm[a = b]
sen312 = HoldForm[b = tmp]
con4 = HoldForm[a <= 0]
sen41 = HoldForm[r = 100]
con5 = HoldForm[a + b < c]
sen51 = HoldForm[r = 100]
con6 = HoldForm[a + b == c]
sen61 = HoldForm[r = 100]
con7 = HoldForm[a == b]
con8 = HoldForm[b == c]
sen81 = HoldForm[r = 400]
sen80 = HoldForm[r = 300]
con9 = HoldForm[b == c]
sen91 = HoldForm[r = 300]
sen90 = HoldForm[r = 200]
prog =
{block,
 {if, con1,
  {block, sen110, sen111, sen112}, {}},
 {if, con2,
  {block, sen210, sen211, sen212}, {}},
 {if, con3,
  {block, sen310, sen311, sen312}, {}},
 {if, con4,
  sen41,
  {if, con5,
   sen51,
   {if, con6,
    sen61,
    {if, con7,
     {if, con8, sen81, sen80},
     {if, con9, sen91, sen90}
    }
   }
  }
 }
}
```