多重ループからの脱出での goto 文の是非：Hoare 論理の観点から

金藤栄孝[†] 二木厚吉[††]

Dijkstra の **goto** 文有害説とそれに引き続く構造的プログラミングの提唱以降，**goto** 文の使用に関する問題は長く議論されてきたが，**goto** 文の使用法に関しての理論的裏付けを持つ研究としては，逐次的プログラムの制御フローは 3 基本構造（順次接続，条件分岐，反復）のみで表現可能であるから **goto** 文を用いたプログラムは 3 基本構造のみによる等価なプログラムに書き換えられる，という結果に基づいた Mills らの **goto** 文排斥論以外は皆無であり，Dijkstra 本来のプログラムの正しさを示す手段としてのプログラムの構造化という観点での **goto** 文の使用の是非は，プログラム検証論の立場から考察されなかった．本論文では，プログラムの正しさを示すという検証手段としての Hoare 論理に基づき **goto** 文の使用を再検討する．特に，多重ループの打ち切りの場合，**goto** 文を用いて脱出するプログラミングスタイルの方が，Mills 流の Boolean 型変数を追加してループを打ち切るスタイルよりも，Hoare 論理での証明アウトラインが簡単に構成でき，したがって，前者の **goto** 文を用いたプログラミングスタイルの方が，そのようなプログラムに対する Hoare 論理による検証上からは望ましいことを示す．

On the Use of goto's in Escaping from Nested Loops: from the Hoare Logic Viewpoint

Hidetaka Kondoh[†] and Kokichi Futatsugi[††]

There have been a vast amount of debates on the issue on usages of **goto** statements initiated by the famous Dijkstra's Letter to the Editor of CACM and his proposal of "Structured Programming". Except for the **goto**-less programming style by Mills based on the fact that any control flows of sequential programs can be expressed by the sequential composition, the conditional (**if-then-else**) and the indefinite loop (**while**), there have not been, however, any scientific accounts on this issue from the Dijkstra's own viewpoint of verifiability of programs. In this work, we reconsider this issue from the viewpoint of Hoare Logic, the most standard framework for proving the correctness of programs, and we see that usages of **goto**'s in escaping from nested loops can be justified from the Hoare Logic viewpoint by showing the fact that constructing the proof-outline of a program using a **goto** for this purpose is easier than constructing the proof-outline of a Mills-style program without **goto** by introducing a new Boolean variable.

1. 歴史的背景

本論文が取り扱う問題は「**goto** 文有害説」と「構造的プログラミング☆」という非常に古典的な主題に関するものであり，著者の考えでは，これらの主題は，提唱者である Dijkstra 本来の意図に従った形では，いまだ十分に検討されていない．そこで，少し長くなるが，これらの主題に関する歴史的な流れをまとめておく．

分かりやすいプログラムを書くことがソフトウェア工学的に大切であることは論をまたない．CACM の編集者への手紙の形の "Go to Statements Considered Harmful" [11] で，Dijkstra は **goto** 文がプログラムの分かりやすさに対して悪影響を与えると警告した．Dijkstra は，さらに，この手紙に引き続いて提唱した[12),13)]「構造的プログラミング」では，分かりやすいプログラムを書く目的はプログラムの正しさを示しやすくすることにあり「プログラムを構造化するのが必要なことを，プログラムの正しさを証明できる必要があることの帰結であるとして提唱しました」（訳書[13)]，p.47）と述べ，最初の手紙での「プログラムの

† 株式会社日立製作所システム開発研究所
Systems Development Laboratory, Hitachi, Ltd.
†† 北陸先端科学技術大学院大学
Japan Advanced Institute of Science and Technology (JAIST)

☆ "Structured Programming" の訳語としては，「構造的プログラミング」と「構造化プログラミング」との 2 つが広く使用されているが，本論文では「構造的プログラミング」を一貫して用いる．

分かりやすさ」という漠然とした基準を「プログラムの正しさの示しやすさ」といういっそう明確な基準へと洗練した．なお，**goto** 文有害説を述べた最初の手紙において，Böhm ら[5] の仕事に代表されるような制御フローに関する表現能力の等価性に基づいた機械的な **goto** 文の除去はプログラムの分かりやすさの向上という――彼が **goto** 文の問題を提起した本来の――趣旨からは無意味である，と Dijkstra 自身がすでに指摘していることを注意しておく．

さて，Dijkstra の手紙は，いわゆる，**goto** 文論争を引き起こし，**goto** 文の使用法や構造的プログラミングが 1970 年代を通じてプログラミング論やソフトウェア工学における重要な主題であったことは，たとえば，その問題に関する代表的な議論の記録としての ACM 主催の討論会報告[23] に見ることができるし，また，1970 年代後半から 1980 年代初頭にかけてのプログラミング論やソフトウェア工学に関する様々なアンソロジー[3),16),43)~46)] に **goto** 文に関する多数の論文が再録されていることからもうかがい知ることができる．しかし，この **goto** 文論争の中で提出された様々な意見や研究は，Dijkstra 自身の本来の意図の「プログラムの分かりやすさ」やそれの洗練としての「プログラムの正しさの証明の容易さ」とはかけ離れて，**goto** 文を許すべきだ[21]，あるいは，許すべきでない[49] といった主張に代表されるように，直感的なものや経験的なもの――**goto** 文がなくてもプログラムを書くうえで困ったことがない等――がほとんどで，**goto** 文の使用に関しては，科学的な裏付けを持つ――すなわち，プログラムに関する科学としてのプログラム理論に基づいた――説明は乏しい．

たとえば，**goto** 文を用いての構造的プログラミングの提唱として頻繁に引用されてきた Knuth[25] はプログラムの読みやすさを損ねない **goto** 文の使用に関して，いくつかの場合をあげ，また，いわゆる，$n\frac{1}{2}$ ループとその Hoare 論理での推論規則等，検証の立場からの制御構造の検討を加えている．しかし，制御構造を超えたスケールとしてのプログラミングスタイルに関しては，**goto** 文使用の是非を検証の立場から検討しているわけではなく，他の **goto** 文論争での論説と同様，直感的な説明にとどまっている．

一方，**goto** 文論争での **goto** 排斥派の与えた **goto** 文忌避の根拠は「**goto** 文なしでも十分に分かりやすいプログラムは書ける」といった経験的なものか，あるいは，Böhm ら[5] に代表される表現力に関する結果に基づくものである．さらに，後者を根拠として **goto**-less プログラミングの発想が生まれ，後にソフトウェア工学的な方法論に発展する．すなわち，クリーンルーム手法（Cleanroom Software Engineering）である．

Mills らは，Dijkstra による **goto** 文の危険性の指摘に基づき，一連の仕事[27),30),31)] において，**goto** 文を用いないプログラミングスタイルを考察し「構造的プログラミング」を，Dijkstra 本来の意味とは異なる **構造的プログラミング＝goto-less プログラミング** としてとらえ[28]，そのようなプログラミングスタイルに限れば，逐次的なプログラムの表示的な意味は単純な状態変換関数としてとらえられることに着目し，Mills の意味での構造化されたプログラム（**goto**-less プログラム）に対する検証手段としては関数等価性[33] を採用し，この検証法を技術的核として，高品質ソフトウェア開発のためのクリーンルーム手法[14),34),36),37),39),40)] を開発した．

Mills らが構造的プログラミングを **goto** 文を除いたプログラミングと同一視する根拠として用いたのは，Böhm らの仕事[5] に代表される逐次的なプログラムの表現力に関する結果である．すなわち，逐次的なプログラムにおける任意の制御フローは，順次接続・条件分岐・ループの，いわゆる，3 基本構造のみで構築することが可能であり，**goto** 文を必要としないという事実である．しかし，制御フローに関する表現力についての理論的事実は，前に注意したとおり，そもそも，Dijkstra が **分かりやすいプログラムを作る目的での goto 文除去の根拠とはならない** と当初より指摘していたものである．

ゆえに，Mills らが「構造的プログラミング」を **goto**-less プログラミングと看做してクリーンルーム手法☆を生み出した過程で，**分かりやすく正しさを容易に示せるプログラムを書くこと**，という Dijkstra 自身が「構造的プログラミング」という言葉に込めた考え方は重大な変質をこうむった，といえる．

言語機能としての **goto** 文そのものに関するプログラム理論上の研究は多数存在する．たとえば，上で Mills らが **goto**-less プログラミングの根拠とした制御フローの表現可能性とそれに基づく **goto** 文の使用の原理的な回避可能性に関する研究，すなわち，逐次的なプログラムの場合，**goto** 文を用いて構成できる任意のプログラムは原理的には，元のプログラム中に

☆ クリーンルーム手法とは，単に **goto**-less というスタイルに基づくプログラミング方法論だけではなく，インクリメンタル開発や統計的品質管理といったいくつかのソフトウェア工学的な考え方からなる複合的な方法論である．あくまでも，ここで，議論の対象としているのは，クリーンルーム手法の中のプログラミング（とそれを支える関数的等価性に基づく検証手法）の側面のみに関してである．

存在しない変数 1 個を追加することで

(1) 順次接続—Pascal での ";" による文の連接,
(2) 条件分岐—同じく **if** 文,
(3) 反復—同じく **while** 文,

のいわゆる「3 基本構造」と呼ばれる構成のみで書き直せる，という類のプログラムの内容に関知しない制御フロー的側面のみに関する研究は，**goto** 文論争以前も以後も存在する．たとえば，すでに触れた Böhm ら[5] 以外にも文献 2)，8)，9)，26) 等をあげることができる．

また，手続き的プログラミング言語の言語機能の 1 つとしての **goto** 文の意味に関するプログラム理論的研究も，たとえば，表示的意味論[35),42)] の立場からは接続法[41)] が，また，Hoare 論理[17),18)] の立場からは **goto** 文に対する検証規則[6),7)] が与えられた．

しかし，これらの研究はあくまでも **goto** 文自身に関する研究であり，Dijkstra が問題提起し，また，本論文で議論の対象とする **goto 文の使用法**については，検討を加えているわけではない．

以上が **goto** 文論争と構造的プログラミングに関する歴史的な概観であるが，これで分かるとおり，Dijkstra が提起した **goto** 文の**使用法**に関しては，膨大な論争があったにもかかわらず，提起から 30 年以上を経ても Böhm らに代表されるような制御フローの表現力のみに関する研究を超えたプログラムの検証の容易さ，という観点からの検討は皆無といってよい．その結果，**goto** 文を全面的に排斥すべしという主張も，一定の使用法は許されるべしとする主張も，ともに，直感的・経験的・主観的な理由のみに基づいたレベルにとどまらざるをえない．そして，**goto** 文の使用法に関して科学的な裏付けを持つ検討とその結果としての **goto** 文の使用基準がいまだに与えられていないことが，近年に至ってもたとえば Plauger のエッセイ[38)] のような **goto** 文の使用法に関する感覚的な意見表明が尽きない原因と考えられる．

本論文では，多重ループで一定の条件の成立で全ループを打ち切る処理を **goto** 文による脱出で行うべきか，そのようなプログラムに対する Hoare 論理による検証が容易になるか否かの観点から検討する．多重ループの打ち切りでの **goto** 文の使用は，Dijkstra 自身が彼の手紙において——彼自身は「元々は Hoare の提案だったと思う」と断っているが——プログラムの異常時の飛び出しに関しては **goto** 文を用いるのが有効だ，と述べている場合に該当し，また，Knuth[25)] でも **goto** 文を用いるのが適切とされている場合の 1 つであるが，Mills 流の構造的プログラミング（たとえば彼と Linger らとの教科書[28)]）に従えば，変数の導入によって **goto** 文を削除すべきとなる．

本論文の構成は以下のとおりである．次章では，本論文で使用する最小限の Hoare 論理の体系を示す．次に，3 章では，具体例に関して，**goto** 文を用いて多重ループを脱出するプログラムと，Mills 流の「構造的プログラミング」に従って脱出すべき条件が成立したか否かを保持する Boolean 変数を導入して **goto** 文を消去したプログラムとのいずれの検証上の表明（ループ不変表明等）が単純になるかを示す．4 章では，多重ループからの脱出を **goto** 文によるプログラムを Boolean 変数を導入して **goto** 文を除去したプログラムに変形した場合の検証上の表明がどのように変化するかを，一般的なプログラムスキームについて示す．最後に，5 章で，なぜ，Mills が **goto** 文の削除にこだわったのかを，彼が用いた検証方法としての関数等価性との関連で述べる．

2. Hoare 論理

本論文では，プログラミング言語としては Pascal を用いる．その Hoare 論理[18)] の公理と推論規則の中で以下の議論に必要な最小限の部分を**図 1** に示す．Pascal に対する Hoare 論理の規則群の全貌に関しては，Hoare らによる Pascal の公理的定義[20)] を参照されたい．なお，図 1 中の公理や推論規則は，式の評価中に変数の値を変更する，いわゆる「式の副作用」はないという前提の下のものである．また，**goto** 文の推論規則に関しては Clint ら[7)] の付録で与えられ，de Bruin[6)] で用いられている形を採用する．以下，Hoare 論理に関する用語については林[17)] に従う．

表明を記述する論理式で用いる論理記号は次のとおり．"∧"，"∨"，"⊃" は，論理積，論理和，含意を，"⊥" は命題定数の偽を，それぞれ表す．Pascal の Boolean 型の式は論理式としても使用するが，本来の論理式なのか Pascal の Boolean 型の式由来なのかを区別するために，Boolean 型の演算子や定数は "**and**"，"true"，"false" と，論理記号とは区別した形で表す．ただし，論理演算のうえでは対応する論理記号/命題定数と同一視する．最後に，"$\Longrightarrow$" はメタ言語レベルでの含意を，また，"$\Longleftrightarrow$" はメタ言語レベルでの論理的等価性を，それぞれ表すものとする．

以下では「プログラム」とは，Pascal の文法上のプログラム——すなわち，"**program**" から "**end**." まで——ではなく，Hoare 論理の議論でよく見られる[17)] ように，いわゆる，実行文——代入文・空文・複合文・制御構造を与える文——の 1 つ以上が ";" で区切られ

$$\{A\}\,(*\ \text{空文}\ *)\,\{A\} \quad [\text{空文}]$$

$$\{A[e/x]\}\,x := e\,\{A\} \quad [\text{代入文}]$$

$$\frac{\{A\}\,S_1\,\{B\} \qquad \{B\}\,S_2\,\{C\}}{\{A\}\,S_1; S_2\,\{C\}} \quad [\text{順次接続}]$$

$$\frac{\{A \wedge b\}\,S_1\,\{B\} \qquad \{A \wedge \neg b\}\,S_2\,\{B\}}{\{A\}\,\textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\,\{B\}} \quad [\text{条件分岐：2 分岐 \textbf{if} 文}]$$

$$\frac{\{A \wedge b\}\,S\,\{B\} \qquad A \wedge \neg b \supset B}{\{A\}\,\textbf{if}\ b\ \textbf{then}\ S\,\{B\}} \quad [\text{条件分岐：単分岐 \textbf{if} 文}]$$

$$\frac{\{A \wedge b\}\,S\,\{A\}}{\{A\}\,\textbf{while}\ b\ \textbf{do}\ S\,\{A \wedge \neg b\}} \quad [\text{反復：\textbf{while} 文}]$$

$$\frac{\{A\}\,S\,\{B\}}{\{A\}\,\textbf{begin}\ S\ \textbf{end}\,\{B\}} \quad [\text{複合文}]$$

$$\frac{A' \supset A \qquad \{A\}\,S\,\{B\} \qquad B \supset B'}{\{A'\}\,S\,\{B'\}} \quad [\text{帰結規則}]$$

$$\frac{\{B\}\,\textbf{goto}\ L\,\{\bot\} \vdash \{A\}\,S_1\,\{B\} \qquad \{B\}\,\textbf{goto}\ L\,\{\bot\} \vdash \{B\}\,S_2\,\{C\}}{\{A\}\,S_1; L\!:\!S_2\,\{C\}} \quad [\textbf{goto}\ \text{文}]$$

図 1 Pascal の Hoare 論理の公理と推論規則（本論文で用いるもののみ）
Fig. 1 Axioms and inference rules of Hoare Logic for a Pascal subset.

た並びを表す．なお，変数の宣言等は省略する．

図 1 の中のほとんどの公理・推論規則に関しては説明は不要と思うが，**goto** 文に関する推論規則についてのみ簡単に説明する．この **goto** 文の推論規則は，C 等の一部の言語で許されているような複合構造を持つ文（**if** 文・**while** 文・**begin** グループ）の内部ラベルへとその文の外側から飛び込む **goto** 文を対象としていないことを注意しておく．

この推論規則の仮定部（規則の水平線の上側）にある 2 つの仮定

$$\{B\}\,\textbf{goto}\ L\,\{\bot\} \vdash \{A\}\,S_1\,\{B\},$$
$$\{B\}\,\textbf{goto}\ L\,\{\bot\} \vdash \{B\}\,S_2\,\{C\}$$

は，“⊢” の右側の S_1, S_2 各々に対する三つ組を Hoare 論理で証明する際，S_1 や S_2 中の **goto** L に関しては，事前条件として飛び先のラベル L が付いた S_2 の事前条件を（事後条件の方は **goto** 文は直後の文へ制御を渡さないので偽 “⊥” を）仮定してよい，ということを表している．そして，その仮定の下で S_1，S_2 各々の三つ組が証明できるならば，帰結部（規則の水平線の下側）にある $S_1; L\!:\!S_2$ に関する三つ組の正しさが（他の三つ組に関する何らかの仮定を置かず）証明できたと看做してよい，ということを，**goto** 文の推論規則は表している．

3. 例題：2 重ループからの脱出

本章では，**goto** 文を使うべきだといわれている[25]代表的なケースであるループからの脱出での **goto** 文の使用に関して，Hoare 論理による検証の見地から，それが望ましい——すなわち，検証が容易なプログラムを書くのに有効である——か否かを 2 重ループの最内ループからの脱出というプログラム例で検討する．具体的には次の問題を考える．

> **例題** 2 次元配列 a[1..m, 1..n] の中に値が 0 の要素が存在すれば，そのような要素の中で辞書式順序で最小の添字値を持つ要素の添字値を，各々，integer 型の変数 i と j とに設定する．値が 0 の要素が a の中に存在しない場合には，m の値より大きな値を i に設定する．

さて，この問題に対して **goto** 文による脱出を用いたプログラムを**図 2** に示す．一方，新しく変数を追加し，代わりに **goto** 文を用いないで済ませたプログラムは**図 3** である．図 3 のプログラムで新たに追加された変数は Boolean 型の変数 nYF で，値が true ならば値 0 の配列要素がいまだ見つかっていないことを表す．

図 2 に完全な表明を付けた証明アウトラインを**図 4** に，また，図 3 に対する証明アウトラインを**図 5** に，各々，示す．なお，これら 2 つの証明アウトラインに

```
 1        i := 1;
 2        while i ≤ m do
 3          begin
 4            j := 1;
 5            while j ≤ n do
 6              if a[i,j] = 0 then
 7                goto 99
 8              else
 9                j := j + 1;
10            i := i + 1
11          end;
12      99: (* 終わり *)
```

図 2 2 次元配列探索のプログラム（**goto** 文使用）
Fig. 2 The program for searching a 0 in a 2-dimensional array (with a **goto**).

```
 1        i := 1;
 2        nYF := true;
 3        while i ≤ m and nYF do
 4          begin
 5            j := 1;
 6            while j ≤ n and nYF do
 7              if a[i,j] = 0 then
 8                nYF := false
 9              else
10                j := j + 1;
11            if nYF then
12              i := i + 1
13          end
```

図 3 2 次元配列探索のプログラム（**goto** 文不使用）
Fig. 3 The program for searching a 0 in a 2-dimensional array (without **goto**).

関して細かいことをいうと，図 2，3 の各プログラムの正しさの完全な証明のためには $\mathrm{i} \geq 1$，$\mathrm{j} \geq 1$ といった条件を表明として追加する必要があり，内側ループ（変数 j を加算するループ）の終了時に $\mathrm{j} = \mathrm{n}+1$ が成立することを形式的に示すには $\mathrm{j} \leq \mathrm{n}+1$ を同ループの不変表明として加えておく必要がある．しかし，これらの条件はいずれのプログラムに対する証明アウトラインでも同じように追加されるので，図 4，5 の両者の比較という本論文の主題には影響しないので，議論を重要な点に集中するために，これらの細かい条件は省略してある．

これらの証明アウトライン中の述語記号や命題定数の意味は以下のとおり．$NYF(k,l)$ は（列優先で探索して）第 k 行目の第 l 列までの範囲では値 0 の配列要素が見つかっていない（Not Yet Found）ということを，$MI(k,l)$ は配列要素 $\mathrm{a}[k,l]$ の値は 0 でそのような要素の添字の中で (k,l) は辞書式順序で最小（Minimal Index）であること，また，SZ は配列 a が値 0 の要素を持つ（Somewhere Zero）ということを，それぞれ表す．これらは，1 階述語論理式で具体的に定義できるが，図 4，5 での帰結規則の適用の正しさを示すうえでは，

$$\begin{aligned}
MI(i,j) &\iff NYF(i,j-1) \\
&\quad \wedge \mathrm{a}[i,j] = 0 \\
&\quad \wedge 1 \leq i \leq \mathrm{m} \wedge 1 \leq j \leq \mathrm{n}, \\
NYF(i,j) &\iff NYF(i,j-1) \\
&\quad \wedge \mathrm{a}[i,j] \neq 0, \\
NYF(i,0) &\iff NYF(i-1,\mathrm{n}), \\
\neg SZ &\iff NYF(\mathrm{m},\mathrm{n}),
\end{aligned}$$

という一連の性質が与えられていれば十分なので，論理式による具体的な定義は省く．

これら 2 つの証明アウトラインを見比べてただちに感じることは，**goto** 文を用いたプログラムの証明アウトライン（図 4）の方が明らかに簡単だ，ということである．

そして，単に簡単に見えるだけでなく，**goto** 文を用いていないプログラム図 3 の正しさの証明で証明アウトラインの図 5 を組み上げるうえで必要な論理的内容は，実質的には，**goto** 文を用いたプログラム図 2 に対する証明アウトラインの図 4 を組み上げるうえで必要な論理的内容の真部分集合であることを以下で示す．

goto 文を用いたプログラムの証明アウトラインである図 4 の方が用いていない方の図 5 よりも簡単に見える理由は，たとえば，各々の外側の **while** 文のループ不変表明を比べれば分かる．最初の **goto** 文を用いたプログラムの当該ループの不変表明は $NYF(\mathrm{i}-1,\mathrm{n})$ である．一方，後の **goto** 文を用いない方のそれは $\neg\mathrm{nYF} \wedge MI(\mathrm{i},\mathrm{j}) \vee \mathrm{nYF} \wedge NYF(\mathrm{i}-1,\mathrm{n})$ となっている．すなわち，後の **goto** 文なしのプログラムの外側ループの不変表明は，

(1) 前の **goto** 文ありでの外側ループの不変表明で，それはループを継続する要件であるから，値 0 の配列要素がいまだ見つかっていないことを表している表明，
(2) 値 0 の配列要素が見つかった場合に成立する表明である $MI(\mathrm{i},\mathrm{j})$，

との各々に，**goto** 文を削除するために導入した Boolean 型の変数 nYF の各々の場合の真偽を表す論理式——$\neg$nYF と nYF——を論理積の形で標識と

```
1        {⊤}
2          {1 = 1}
3          i := 1
4          {i = 1}
5        {i = 1};
6          {NYF(i − 1, n)}
7          while i ≤ m do
8            {NYF(i − 1, n) ∧ i ≤ m}
9            begin
10             {NYF(i − 1, n) ∧ i ≤ m}
11               {NYF(i, 1 − 1) ∧ i ≤ m}
12               j := 1
13               {NYF(i, j − 1) ∧ i ≤ m}
14             {NYF(i, j − 1) ∧ i ≤ m};
15             while j ≤ n do
16               {NYF(i, j − 1) ∧ i ≤ m ∧ j ≤ n}
17               if a[i, j] = 0 then
18                 {NYF(i, j − 1) ∧ i ≤ m ∧ j ≤ n ∧ a[i, j] = 0}
19                   {MI(i, j) ∨ ¬SZ ∧ i > m}
20                   goto 99
21                   {⊥}
22                 {NYF(i, j − 1) ∧ i ≤ m}
23               else
24                 {NYF(i, j − 1) ∧ i ≤ m ∧ j ≤ n ∧ ¬(a[i, j] = 0)}
25                   {NYF(i, (j + 1) − 1) ∧ i ≤ m}
26                   j := j + 1
27                   {NYF(i, j − 1) ∧ i ≤ m}
28                 {NYF(i, j − 1) ∧ i ≤ m}
29               {NYF(i, j − 1) ∧ i ≤ m}
30             {NYF(i, j − 1) ∧ i ≤ m ∧ ¬(j ≤ n)};
31               {NYF((i + 1) − 1, n)}
32               i := i + 1
33               {NYF(i − 1, n)}
34             {NYF(i − 1, n)}
35           end
36           {NYF(i − 1, n)}
37         {NYF(i − 1, n) ∧ ¬(i ≤ m)}
38       {MI(i, j) ∨ ¬SZ ∧ i > m};
39     99: (* 終わり *)
40       {MI(i, j) ∨ ¬SZ ∧ i > m}
```

図 4 図 2 のプログラムに対する完全証明アウトライン
Fig. 4 The complete proof-outline of the program in Fig. 2.

して付けたものどうしの論理和という形になっている．

そして，図 5 では，上で述べたとおり外側のループ不変表明が見つかった場合に関する部分表明を論理和の成分として含んでいるので，そのループの内部の多くの表明も見つかった場合の部分表明を同じ形で含む羽目になっている．そして，実際に，見つからない場合に関する処理の場合には，帰結規則を用いて，見つからない場合に関する部分表明を，その処理の直接の事前/事後表明とするような表明の変形操作を必要としている．

一方，**goto** 文を用いた場合の図 4 では，上で説明した外側ループの不変表明をはじめとして，ほとんどの表明は値 0 の配列要素が見つからない場合の形となっており，唯一の例外は，実際に見つかった場合に 2 重ループから脱出するための **goto** 99 の事前条件（19 行目）のみが，見つかった場合に成立すべき表明である $MI(\mathrm{i},\mathrm{j})$ を含んだ形となっている．

つまり，図 2 のプログラムは **goto** 文を用いることによって，その Hoare 論理による検証のための表明付けでは，当該 **goto** 文の前後を除いて，見つかっていない場合のみを考えて表明を考えればよいのに対して，**goto** 文を用いていない図 3 の場合は，つねに見つかった場合と見つかっていない場合の両方の可能性に対応する表明付けを行わなければならない．

なぜ，この違いが起こったのかを考察すると，**goto** 文は次の文に制御を渡さないという性質から事後条件としては偽 “⊥” であるので，帰結規則により任意の都合のよい表明へと変更できるからである．つまり，図 4 の **goto** 文の事前条件は見つかった場合の表明の形を含んでいるにもかかわらず，事後条件の方は，見つかっていない場合のみの形に戻すことができている．これが，**goto** 文でループ脱出をすることによっ

```
 1  {⊤}
 2    {1 = 1}
 3    i := 1
 4    {i = 1}
 5  {i = 1};
 6    {true ∧ i = 1}
 7    nYF := true
 8    {nYF ∧ i = 1}
 9  {nYF ∧ i = 1};
10    {¬nYF ∧ MI(i, j) ∨ nYF ∧ NYF(i − 1, n)}
11    while i ≤ m and nYF do
12      {(¬nYF ∧ MI(i, j) ∨ nYF ∧ NYF(i − 1, n)) ∧ i ≤ m and nYF}
13        {nYF ∧ NYF(i − 1, n) ∧ i ≤ m}
14        begin
15          {nYF ∧ NYF(i − 1, n) ∧ i ≤ m}
16            {nYF ∧ NYF(i, 1 − 1) ∧ i ≤ m}
17            j := 1
18            {nYF ∧ NYF(i, j − 1) ∧ i ≤ m}
19          {nYF ∧ NYF(i, j − 1) ∧ i ≤ m};
20            {¬nYF ∧ MI(i, j) ∨ nYF ∧ NYF(i, j − 1) ∧ i ≤ m}
21            while j ≤ n and nYF do
22              {(¬nYF ∧ MI(i, j) ∨ nYF ∧ NYF(i, j − 1) ∧ i ≤ m) ∧ j ≤ n and nYF}
23                {nYF ∧ NYF(i, j − 1) ∧ i ≤ m ∧ j ≤ n}
24                if a[i, j] = 0 then
25                  {nYF ∧ NYF(i, j − 1) ∧ i ≤ m ∧ j ≤ n ∧ a[i, j] = 0}
26                    {¬false ∧ MI(i, j)}
27                    nYF := false
28                    {¬nYF ∧ MI(i, j)}
29                  {¬nYF ∧ MI(i, j) ∨ nYF ∧ NYF(i, j − 1) ∧ i ≤ m}
30                else
31                  {nYF ∧ NYF(i, j − 1) ∧ i ≤ m ∧ j ≤ n ∧ ¬(a[i, j] = 0)}
32                    {nYF ∧ NYF(i, (j + 1) − 1) ∧ i ≤ m}
33                    j := j + 1
34                    {nYF ∧ NYF(i, j − 1) ∧ i ≤ m}
35                  {¬nYF ∧ MI(i, j) ∨ nYF ∧ NYF(i, j − 1) ∧ i ≤ m}
36                (* if 文終わり *)
37                {¬nYF ∧ MI(i, j) ∨ nYF ∧ NYF(i, j − 1) ∧ i ≤ m}
38              {¬nYF ∧ MI(i, j) ∨ nYF ∧ NYF(i, j − 1) ∧ i ≤ m}
39            {(¬nYF ∧ MI(i, j) ∨ nYF ∧ NYF(i, j − 1) ∧ i ≤ m) ∧ ¬(j ≤ n and nYF)}
40          {¬nYF ∧ MI(i, j) ∨ nYF ∧ NYF(i, n)};
41          if nYF then
42            {(¬nYF ∧ MI(i, j) ∨ nYF ∧ NYF(i, n)) ∧ nYF}
43              {nYF ∧ NYF((i + 1) − 1, n)}
44              i := i + 1
45              {nYF ∧ NYF(i − 1, n)}
46            {¬nYF ∧ MI(i, j) ∨ nYF ∧ NYF(i − 1, n)}
47          {¬nYF ∧ MI(i, j) ∨ nYF ∧ NYF(i − 1, n)}
48        end
49          {¬nYF ∧ MI(i, j) ∨ nYF ∧ NYF(i − 1, n)}
50      {¬nYF ∧ MI(i, j) ∨ nYF ∧ NYF(i − 1, n)}
51    {(¬nYF ∧ MI(i, j) ∨ nYF ∧ NYF(i − 1, n)) ∧ ¬(i ≤ m and nYF)}
52  {MI(i, j) ∨ ¬SZ ∧ i > m}
```

図 5 図 3 のプログラムに対する完全証明アウトライン
Fig. 5 The proof-outline of the program in Fig. 3.

て，表明付けが簡単になった理由である．一方，図 3 のプログラムでは，その中のほとんどの部分は，当該配列要素が見つかったか見つかっていないかのいずれか一方の場合に対する処理であるので，帰結規則を多用して，それらの処理に直接に対応する事前/事後表明を両方の可能性に対応した表明と結び付けなければならなくなっている．このことが，図 5 の証明アウトラインが図 4 のよりも複雑に見える理由である．実際，前者では帰結規則の適用は 6 カ所であるのに対して，**goto** 文を用いていない後者では 10 カ所とほとんどの文に対して帰結規則を適用する結果となっている☆．

以上で見たとおり，図 2 のプログラムから図 4 の

☆ 図 5 で帰結規則の適用を免れている文は，41〜46 行目——元の図 2 での 11〜12 行目——の単分岐 **if** 文全体のみである．

証明アウトラインを導くのに必要な論理的内容（ループ不変表明や中間表明の設定，各帰結規則での論理式の含意関係の確認）のすべては図 3 から図 5 を導く過程でも利用されており，さらに，**goto** 文を除去した後者の場合には，図 4 には存在しない余分な帰結規則の適用が必要である．

ゆえに，図 2 のプログラムに対して適切なループ不変表明を考えついて図 4 の形へと表明付けできないならば，図 3 のプログラムを図 5 に示した完全証明アウトラインとへと仕上げて検証できないのは明らかである．

4. ループ脱出のための goto 文の除去の一般形に対する表明付け

本章では，前章の例題で観察した，ループ脱出で **goto** 文を使用した場合と Boolean 型変数の導入で **goto** 文を除去した場合とで Hoare 論理による検証での表明付けがどのように変化するかを，一般的に $(n+1)$ 重ループ $(n \geq 1)$ を含むプログラムスキーマの形で検討する．

図 6 は $(n+1)$ 重ループの最内ループから条件 *ifCond* が成立している場合に **goto** 文で全ループを脱出するプログラムスキーマであり，最内の $(n+1)$ 重目以外の i 重目の各ループ $(1 \leq i \leq n)$ は，

```
while whCond_i do
  begin
    S_i;
    (i+1) 重目のループ;
    T_i
  end
```

という形である．一方，**図 7** は，それと等価で **goto** 文を除去し代わりに Boolean 型の新しい変数 *fv* を導入してその値でループの反復を打ち切るプログラムスキーマであり，最内以外の i 重目の各ループ $(1 \leq i \leq n)$ は，

```
while whCond_i and fv do
  begin
    S_i;
    (i+1) 重目のループ;
    if fv then
      T_i
  end
```

という形である．以下，*fv* のように **goto** 文除去の目的で導入された新しい変数を「フラグ変数」と呼ぶ．なお，図 6，図 7 中の各メタ変数については，S_i，T_i $(1 \leq i \leq n+1)$ および U は 0 個以上の文の並びを，$whCond_i$ $(1 \leq i \leq n+1)$ および *ifCond* は Boolean 型の式を，また，*fv* は Boolean 型のフラグ変数を，それぞれ表す．

さて，図 6 のプログラムスキーマの事前条件を *Pre*，

```
1   while whCond_1 do
2     begin
3       S_1;
4       ...
5         while whCond_{n+1} do
6           begin
7             S_{n+1};
8             if ifCond then
9               goto L
10            else
11              T_{n+1}
12          end
13      ...;
14      T_1
15    end;
16  U;
17  L: (* 空文 *)
```

図 6 **goto** 文によるループ脱出を含むプログラムスキーマ
Fig. 6 A program-scheme with a loop escaped by a **goto**.

```
1   fv := true;
2   while whCond_1 and fv do
3     begin
4       S_1;
5       ...
6         while whCond_{n+1} and fv do
7           begin
8             S_{n+1};
9             if ifCond then
10              fv := false
11            else
12              T_{n+1}
13          end
14      ...;
15      if fv then
16        T_1
17    end;
18  if fv then
19    U
```

図 7 図 6 の **goto** 文を除去したプログラムスキーマ
Fig. 7 The program-scheme after eliminating the **goto** in Fig. 6.

```
1        {Pre}
2          {Inv_1}
3          while whCond_1 do
4            {Inv_1 ∧ whCond_1}
5            begin
6              {Inv_1 ∧ whCond_1}
7              S_1
8              {Inv_2};
9              ...
10               {Inv_{n+1}};
11               while whCond_{n+1} do
12                 {Inv_{n+1} ∧ whCond_{n+1}}
13                 begin
14                   {Inv_{n+1} ∧ whCond_{n+1}}
15                   S_{n+1}
16                   {P};
17                   if ifCond then
18                     {P ∧ ifCond}
19                       {Post}
20                       goto L
21                       {⊥}
22                     {Inv_{n+1}}
23                   else
24                     {P ∧ ¬ifCond}
25                     T_{n+1}
26                     {Inv_{n+1}}
27                   {Inv_{n+1}}
28                 end
29                 {Inv_{n+1}}
30               {Inv_{n+1} ∧ ¬whCond_{n+1}};
31             ...
32             {Inv_2 ∧ ¬whCond_2};
33             T_1
34             {Inv_1}
35           end
36           {Inv_1}
37         {Inv_1 ∧ ¬whCond_1}
38       {Inv_1 ∧ ¬whCond_1};
39       U
40       {Post};
41     L: (* 空文 *)
42       {Post}
```

図 8 図 6 に対する証明アウトライン
Fig. 8 The proof-outline for the program-scheme in Fig. 6.

事後条件を $Post$ で表す．このとき，このプログラムスキーマに対する証明アウトラインのスキーマは，各ループに対する適当なループ不変表明 Inv_i $(1 \leq i \leq n+1)$ と中間表明 P とが存在して，**図 8** となる．したがって，図 6 のプログラムスキーマを図 6 中のメタ変数を具体的な文や式に置き換えることで具体的なプログラム——たとえば図 2 ——にあてはめた場合，そのプログラムの正当性の証明を図 8 に示した証明アウトラインのスキーマを用いて行うために示さなければならない事柄（証明責務）は，各帰結規則の適用の正しさ（帰結規則の適用により 1 つの文には事前/事後両条件が各々2 個ずつネストした形となるが，その外側の事前条件から内側のものが含意されることと内側の事後条件から外側のが含意されること）の中で図 8 の証明アウトラインスキーマでの論理式の変形で導けないものとしての

(1) $Pre \supset Inv_1$,
　1 行目の表明の論理式から 2 行目のへの含意.
(2) $P \wedge ifCond \supset Post$,
　18 行目の論理式から 19 行目のへの含意.

の 2 項目と，2 章で示した公理や推論規則の適用結果でない形の三つ組としての

(3_i) $\{Inv_i \wedge whCond_i\}\, S_i\, \{Inv_{i+1}\}$
$(1 \le i \le n)$, 例：6–8 行目の三つ組 $(i = 1)$.

(4_i) $\{Inv_{i+1} \wedge \neg whCond_{i+1}\}\, T_i\, \{Inv_i\}$
$(1 \le i \le n)$, 例：32–34 行目の三つ組 $(i = 1)$.

(5) $\{Inv_{n+1} \wedge whCond_{n+1}\}\, S_{n+1}\, \{P\}$,
14–16 行目の三つ組.

(6) $\{P \wedge \neg ifCond\}\, T_{n+1}\, \{Inv_{n+1}\}$,
24–26 行目の三つ組.

(7) $\{Inv_1 \wedge \neg whCond_1\}\, U\, \{Post\}$,
38–40 行目の三つ組.

の $(2n+3)$ 項目の，計 $(2n+5)$ 項目の各メタ変数を具体的な文/式/表明に置き換えた論理式/三つ組の正しさである．

一方，フラグ変数 fv の導入により **goto** 文を除去した図 7 のプログラムスキーマに対する同一の事前/事後条件での証明アウトラインのスキーマは**図 9** である．これを具体的なプログラム——たとえば，図 3 ——の検証で使う場合に証明責務として示すべきは，

$(1')$ $Pre \supset (\neg\mathrm{true} \wedge Post \vee \mathrm{true} \wedge Inv_1)$,
1 行目の表明の論理式から 2 行目のへの含意.

$(2')$ $(fv \wedge P \wedge ifCond) \supset (\neg\mathrm{false} \wedge Post)$,
27 行目の表明の論理式から 28 行目のへの含意.

$(3'_i)$ $\{fv \wedge Inv_i \wedge whCond_i\}\, S_i\, \{fv \wedge Inv_{i+1}\}$,
$(1 \le i \le n)$, 例：12–14 行目の三つ組 $(i = 1)$.

$(4'_i)$ $\{fv \wedge Inv_{i+1} \wedge \neg whCond_{i+1}\}\, T_i\, \{fv \wedge Inv_i\}$
$(1 \le i \le n)$, 例：48–50 行目の三つ組 $(i = 1)$.

$(5')$ $\{fv \wedge Inv_{n+1} \wedge whCond_{n+1}\}\, S_{n+1}\, \{fv \wedge P\}$,
23–25 行目の三つ組.

$(6')$ $\{fv \wedge P \wedge \neg ifCond\}\, T_{n+1}\, \{fv \wedge Inv_{n+1}\}$,
34–36 行目の三つ組.

$(7')$ $\{fv \wedge Inv_1 \wedge \neg whCond_1\}\, U\, \{fv \wedge Post\}$,
60–62 行目の三つ組.

の，やはり計 $(2n+5)$ 項目である．

フラグ変数 fv の選び方より，この変数 fv は文や式を表すメタ変数 S_i，T_i，U，$whCond_i$，$ifCond$ を置換して具体的なプログラムとする際の実際の文/式の中には現れないと仮定してよい．ゆえに，fv の表す Boolean 型変数は表明のメタ変数 Pre，$Post$，Inv_i，P が表す具体的な表明中に現れないとしてよい．

さて，図 9 に対する証明責務 $(1')$〜$(7')$ の各々は，図 8 に対する証明責務 (1)〜(7) から次のように導ける．

まず，論理式に関しては，$(1) \Longleftrightarrow (1')$ なる等価性と $(2) \Longrightarrow (2')$ という含意が成立するのは明らか．対応する三つ組—— (k) と (k')，$k = 3$〜7 ——どうしに関しては，次の推論規則

$$\frac{\{A\}\, S\, \{C\} \quad \{B\}\, S\, \{D\}}{\{A \wedge B\}\, S\, \{C \wedge D\}},$$

および，公理

$\{A\}\, S\, \{A\}$
（A は S 中で代入されうる変数を含まない）

の各々[1] が Hoare 論理で許容可能（admissible）であることに注意すれば，具体的なプログラムに対する証明アウトラインでは，各三つ組 (k) の正しさから (k') の正しさを導けることは容易に分かる．すなわち，$k = 3$〜7 の各三つ組どうしについて，$(k) \Longrightarrow (k')$ である．

以上から，**goto** 文を用いないプログラムスキーマの証明アウトラインのスキーマ図 9 の適用時に示さなければならない証明責務 $(1')$〜$(7')$ は，**goto** 文を用いたもの図 8 の適用時の (1)〜(7) よりも論理的に弱い．

一般に，論理的に弱い命題や三つ組を示すことは，より強いものを示すよりも容易である．

しかし，以上の証明責務の間の論理的な強弱の違いは，文/式/表明を表すメタ変数を含んだ証明アウトラインのスキーマで議論していることに由来する見かけ上ものであり，具体的なプログラムに対する証明アウトライン（たとえば，図 4 と図 5）においては，実際には両者の証明責務を示す難しさは実際上は同一である．以下，その理由を述べる．

まず，論理式に関する証明責務の (2) と $(2')$ とであるが，議論を簡単にするために，$(2')$ の代わりにそれと論理的に等価な

$(2'')$ $fv \supset ((P \wedge ifCond) \supset (\neg\mathrm{false} \wedge Post))$

という論理式と (2) の論理式との証明の難しさが同等であることを示す．前に述べたとおり，フラグ変数 fv は表明 P，$Post$ や Boolean 型の式 $ifCond$ に現れない．したがって，$(2'')$ の論理式の証明で，含意の結論側（外側の “⊃” の右側）の論理式—— (2) のとまったく同一——には fv は現れない．したがって，結論側の論理式の証明には fv を有効な仮定として使用できない．ゆえに，たとえば，自然演繹法で $(2'')$ を証明する場合には，含意の結論側の論理式（すなわち，(2) そのもの）を fv を仮定として用いずに証明し，最後に，実際には仮定として用いていない fv を含意の導入規則を用いて空の仮定として落とさねばならない．このことは，$(2'')$ の証明は (2) の証明に余分な 1 ステップを加えたものだ，ということを意味する．よって，(2) の証明の難しさは $(2'')$ の証明と同等以下である．

```
1   {Pre}
2     {¬true ∧ Post ∨ true ∧ Inv1}
3     fv := true
4     {¬fv ∧ Post ∨ fv ∧ Inv1}
5   {¬fv ∧ Post ∨ fv ∧ Inv1};
6     {¬fv ∧ Post ∨ fv ∧ Inv1}
7     while whCond1 and fv do
8       {(¬fv ∧ Post ∨ fv ∧ Inv1) ∧ (whCond1 and fv)}
9         {fv ∧ Inv1 ∧ whCond1}
10        begin
11          {fv ∧ Inv1 ∧ whCond1}
12            {fv ∧ Inv1 ∧ whCond1}
13            S1
14            {fv ∧ Inv2}
15          {¬fv ∧ Post ∨ fv ∧ Inv2};
16          ...
17            {¬fv ∧ Post ∨ fv ∧ Invn+1};
18              {¬fv ∧ Post ∨ fv ∧ Invn+1}
19              while whCondn+1 and fv do
20                {(¬fv ∧ Post ∨ fv ∧ Invn+1) ∧ (whCondn+1 and fv)}
21                  {fv ∧ Invn+1 ∧ whCondn+1}
22                  begin
23                    {fv ∧ Invn+1 ∧ whCondn+1}
24                    Sn+1
25                    {fv ∧ P};
26                    if ifCond then
27                      {fv ∧ P ∧ ifCond}
28                        {¬false ∧ Post}
29                        fv := false
30                        {¬fv ∧ Post}
31                      {¬fv ∧ Post ∨ fv ∧ Invn+1}
32                    else
33                      {fv ∧ P ∧ ¬ifCond}
34                        {fv ∧ P ∧ ¬ifCond}
35                        Tn+1
36                        {fv ∧ Invn+1}
37                      {¬fv ∧ Post ∨ fv ∧ Invn+1}
38                    {¬fv ∧ Post ∨ fv ∧ Invn+1}
39                  end
40                  {¬fv ∧ Post ∨ fv ∧ Invn+1}
41              {¬fv ∧ Post ∨ fv ∧ Invn+1}
42            {(¬fv ∧ Post ∨ fv ∧ Invn+1) ∧ ¬(whCondn+1 and fv)}
43          {¬fv ∧ Post ∨ fv ∧ Invn+1 ∧ ¬whCondn+1};
44          ...
45          {¬fv ∧ Post ∨ fv ∧ Inv2 ∧ ¬whCond2};
46          if fv then
47            {(¬fv ∧ Post ∨ fv ∧ Inv2 ∧ ¬whCond2) ∧ fv}
48              {fv ∧ Inv2 ∧ ¬whCond2}
49              T1
50              {fv ∧ Inv1}
51            {¬fv ∧ Post ∨ fv ∧ Inv1}
52          {¬fv ∧ Post ∨ fv ∧ Inv1}
53        end
54        {¬fv ∧ Post ∨ fv ∧ Inv1}
55      {¬fv ∧ Post ∨ fv ∧ Inv1}
56    {(¬fv ∧ Post ∨ fv ∧ Inv1) ∧ ¬(whCond1 and fv)}
57  {¬fv ∧ Post ∨ fv ∧ Inv1 ∧ ¬whCond1};
58  if fv then
59    {(¬fv ∧ Post ∨ fv ∧ Inv1 ∧ ¬whCond1) ∧ fv}
60      {fv ∧ Inv1 ∧ ¬whCond1}
61      U
62      {fv ∧ Post}
63    {Post}
64  {Post}
```

図 9 **goto** 文を除いた図 7 のプログラムスキーマに対する証明アウトライン
Fig. 9 The proof-outline for the **goto**-less program-scheme in Fig. 7.

次に三つ組に関する証明責務の (3_i)〜(7) の各々と $(3'_i)$〜$(7')$ の各々とが同等であることを示す．いずれも，後の三つ組は対応する前の三つ組の事前/事後両条件の論理式にフラグ変数 fv が論理積で結ばれている形であり，次の命題の前提条件を満たす形をしている．

命題 S をプログラム，A，B を論理式，v を S，A，B のいずれにも現れない Boolean 型のプログラ

ム変数とする．このとき，

$$\vdash \{v \wedge A\}\, S\, \{v \wedge B\} \implies \vdash \{A\}\, S\, \{B\}.$$

証明 $\{v \wedge A\}\, S\, \{v \wedge B\}$ が証明可能であり，v は S に現れないので，特に，$v = \mathrm{true}$ の場合の三つ組 $\{(v \wedge A)[\mathrm{true}/v]\}\, S\, \{(v \wedge B)[\mathrm{true}/v]\}$ も証明可能である．v は A，B に現れないので，$(v \wedge A)[\mathrm{true}/v] \equiv \mathrm{true} \wedge A \iff A$．同様に，$(v \wedge B)[\mathrm{true}/v] \iff B$．したがって，$\{A\}\, S\, \{B\}$ も証明可能である． ■

したがって，三つ組の形の証明責務に関しても，両者の証明責務を示すことは同等である．

以上より，**goto** 文を用いた場合の証明責務 (1)〜(7) と用いない場合の (1′)〜(7′) は同等であるといえるが，一方，**goto** 文を用いた場合の証明アウトラインスキーマの図 8 と **goto** 文を除去した場合の証明アウトラインスキーマの図 9 とを比較すると，3 章の例題での観察（図 4 と図 5 との比較）と同様，**goto** 文を除去した場合の方が，証明アウトラインの構築上，必要とされる帰結規則の適用回数も多い．また，各ループの不変表明についても，図 8 の **goto** 文を用いた場合にはループが正常に反復する場合のみを考えた論理式 Inv_i であるのに対し，図 9 の場合は，フラグ変数 fv が真でループの反復が継続する場合には正常な反復の不変表明を，また，同変数が偽となってループ脱出が起こる場合には脱出後に成り立つべき表明 $Post$ となるように，2 通りの状況に応じてフラグ変数 fv の真偽それぞれで論理積をとった形の論理和となっているのは，3 章の例題でも観察したとおりである．

ゆえに，多重ループ脱出では，**goto** 文を用いた方が，それを用いないのと比べて，Hoare 論理による検証のための証明アウトラインを同等かより容易に構築できることが示された．

5. まとめに代えて——検証方法に基づいたプログラミングスタイルの重要性

1 章で述べたように，**goto** 文有害説や引き続く構造的プログラミングの提唱で Dijkstra 自身が持っていた意図は，容易に正しさを示せるようなプログラムを書くことであり，彼自身の構造的プログラミングの論説[13] では **goto** 文の有無の問題は直接には語られていない．すなわち，木村ら[24]（pp.56–57）が指摘しているとおり，Dijkstra 自身による構造的プログラミングの提唱での意図は，段階的詳細化[47),48)] によって内容が理解しやすく正しさの証明が容易なプログラムを書こうというものであった．

goto 文を用いないでプログラムを書くことを表す標語として，Dijkstra の提唱した “Structured Programming” の語を流用したのは Mills ら[29] であり，その **goto** 文除去の理論的根拠として用いているのは Böhm らの仕事[5] である．そこでは明確に「構造的プログラムの必要条件の 1 つは **goto** 文を用いていないことである」という主張が述べられている．そして，Mills と彼の仲間たちの一連の仕事[27),28),30),31),33)] は，最終的にクリーンルーム手法[14),34),36),37),39),40)] という開発方法論の核心部分として結実するが，その過程で，**構造的プログラミング＝goto-less プログラミング**の類の標語で表現される，連接と条件分岐と反復の 3 基本構造だけでプログラムを書くことが構造的プログラミングだ，との認識が，一般のソフトウェア技術者やソフトウェア産業に広まったと考えられる．

さて，Mills 流の構造的プログラミングが，なぜ，構造的プログラミングと **goto** 文不使用とを結び付けたかの理由を考えると，技術的な理由の 1 つとしては，彼がプログラムの検証方法として関数等価性を用いていること[28),30),31),33)] をあげることが可能である．

関数等価性に基づく検証とは，プログラムの仕様を状態変換関数（ここで「状態」とは，変数からそれが保持する値への写像）として与え，実際のプログラムが表す状態変換関数（プログラム関数）が仕様で与えられた状態変換関数（意図関数）と等しいか否かを調べる方法で，実用的な高信頼性プログラム開発技術としてのクリーンルーム手法での検証方法として採用されている．

以上のプログラムの意味を変数から値への写像としての状態変換関数と見なすのは手続き的言語に対する表示的意味論[35),42)] で**直接法（direct semantics）**と呼ばれる意味の与え方であるが，手続き的言語の表示的意味論ではよく知られているとおり，一般の **goto** 文だけでなくループ等の脱出に限定した **exit** 文も含め，何らかの飛び越しを行う言語機能を含む場合には，直接法によって意味を与えることはできず，高階関数の概念を含んだ接続[41] を用いた**接続法（continuation semantics）**と呼ばれる意味の与え方を用いる必要がある．

すなわち，Mills が彼の構造的プログラミングで **goto**-less に拘泥したのは，直接法に於けるプログラムの表示としての状態変換関数に基づく関数等価性をプログラムの正しさの証明方法に選んだため，つまり，検証方法の選択がプログラムの構造に科す制約の反映だと考えられる．むろん，Mills が接続法での表示としての接続ではなくて直接法での状態変換関数に基づいて関数等価性を検証方法として選んだのは，直感的

な理解しやすさと，それがもたらす実用的な適用可能性への配慮からである．

しかし，逐次的で決定性の簡単なプログラムの場合でさえ，その意味を状態変換関数としてとらえることは必ずしも適切でない．次の例（林[17] の p.21 の最大公約数のプログラムを Pascal に書き直したもの）を考える．

```
1        a := x; b := y;
2        while b <> 0 do
3            begin
4                a := a mod b;
5                c := a; a := b; b := c
6            end;
7        if a < 0 then
8            a := −a
```

これは，x，y は一般の（負も許した）整数を値とし，ただし，y は非零とするときに，両者の最大公約数 “$gcd(\mathrm{x},\mathrm{y})$” を変数 a に設定するプログラムである．

このプログラムに対する Mills 流の関数等価性での検証のためには，2〜6 行目の **while** 文全体の意味を状態変換関数として表す必要がある．今，変数 v_i の値を式 e_i の値とするという状態変換関数の構成単位を “$[v_1 \mapsto e_1, \ldots]$” として表現し，状態変換関数の条件ごとの重ね合わせを “[条件 $\Rightarrow$ 状態変換関数, ...]” と書くことにすれば，**while** 文全体の意味を表す状態変換関数は

$$\big[\mathit{SomeCond} \Rightarrow [\mathrm{a} \mapsto \mathit{gcd}(\mathrm{x},\mathrm{y})],\\ \neg\mathit{SomeCond} \Rightarrow [\mathrm{a} \mapsto -\mathit{gcd}(\mathrm{x},\mathrm{y})]\big]$$

という形で表される．ここで，“*SomeCond*” は，以上のループが終わった場合に a が非負になるための入力値 x，y に関する条件であるが，以上のプログラムを書く場合には，この条件 *SomeCond* を詳細に明確化する必要はない．一方，関数等価性に基づく検証では，この種の条件の明確化がつねに不要とは限らない．すなわち，プログラムの検証方法として関数等価性を用いる場合，本来はプログラム関数を厳密な形で求めなければならない．そのためには，ここでの *SomeCond* のように，プログラミングにおいてさえ必要とならない条件も具体的に求めなければならず，余計な労力が必要となってしまう．

一方，Hoare 論理の場合，少なくとも手続き呼び出しを超えない範囲での **goto** 文に関しては，図 1 の **goto** 文の推論規則で分かるとおり，大した困難をともなうことなく扱うことができる．また，Hoare 論理での検証で必要な事前/事後条件を構成する数学的概念は，入出力と状態に関する関係（述語）であり，Mills が彼のクリーンルーム手法の検証の基礎概念として選んだ状態変換関数と同じく 1 階の概念であって，接続のような高階の概念ではなく，状態変換関数という状態の間の関数を状態に関する関係へと一般化したにすぎない．この点は，Mills 流の関数等価性による検証を，**goto** 文を扱うために拡張した場合に必要となる接続法での「接続」という高階の概念（接続とは状態変換関数のなす集合上の写像なので必然的に高階関数となる）とは決定的に異なる．しかも，Hoare 論理では非決定性プログラム等の検証も無理なく扱えるのに対し，関数等価性では「関数」という制約のために非決定性等の扱いが面倒になるのは明らかである．

以上に述べてきたとおり，どのようなプログラミングスタイルが「望ましい」のかということは，どのような検証法を採用するか，ということに依存する．構造的プログラミングとは何かを研究するうえでプログラムの正当性や検証のことも考察すべきだ，ということは Gries[15] も指摘しているが，検証方法を背景としたプログラミングスタイルの検討が正しさを示しやすく信頼性の高いプログラムを書くうえで重要であり，そのような検討を積み重ねることこそが構造的プログラミングの原点としての Dijkstra 本来の意図だと著者は考える．本論文はその方向への非常に小さな一歩を目指し，Dijkstra の手紙での異常時の飛び出しは **goto** 文で対処すべし，というスタイルに対して Hoare 論理という科学的立場からの根拠を与えた．

謝辞 大槻繁氏（現エクイティ・リサーチ取締役．元（株）日立製作所システム開発研究所主任研究員）は，書籍の執筆等の機会を通じ，著者にクリーンルーム手法・ホーア論理・プログラム検証について考察する契機を与え，また励まし続けてくださいました．野木兼六氏（現神奈川工科大学教授．元（株）日立製作所基礎研究所主任研究員）は Hoare 論理のみならずソフトウェア工学やプログラミング理論全般に関し，様々な議論を通じ忍耐強く導き続けてくださいました．谷津弘一氏（（株）日本フィッツ）は草稿に対し様々なコメントをくださり本論文の改善にきわめて有益でした．山崎利治氏は，“Structured Programming” の 2 つの訳語と両者のニュアンスの違いに関しご教示くださいました．これらの方々に心より感謝いたします．

参 考 文 献

1) Apt,K.R. and Olderog, E.-R.: *Verification of Sequential and Concurrent Programs*, 2nd edition, Springer-Verlag (1997).
2) Ashcroft, E. and Manna, Z.: The Translation of ‘go to’ Programs to ‘while’ Programs, *Proc.*

1971 IFIP Congress, North-Holland, Vol.1, pp.250–255 (1972); reprinted in 44), pp.51–61.
3) Basili, V.R. and Baker, F.T.: *Tutorial on Structured Programming: Integrated Practices*, IEEE Computer Society Press (1981).
4) Becker, S.A. and Whittaker, J.A.: *Cleanroom Software Engineering Practices*, Idea Group Publishing (1997).
5) Böhm, C. and Jacopini, G.: Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules, *Comm. ACM*, Vol.9, No.5, pp.366–371 (1966); reprinted in 44), pp.13–25.
6) de Bruin, A.: **Goto** Statements: Semantics and Deduction Systems, *Acta Inf.*, Vol.15, pp.385–424 (1981).
7) Clint, M. and Hoare, C.A.R.: Program Proving: Jumps and Functions, *Acta Inf.*, Vol.1, pp.214–224 (1972).
8) Cooper, D.C.: Böhm and Jacopini's Reduction of Flow Charts, *Comm.ACM*, Vol.10, No.8, p.463 and 467 (1967); reprinted in 45), pp.205–206.
9) Cooper, D.C.: Some Transformations and Standard Forms of Graphs, with Applications to Computer Programs, *Machine Intelligence*, Dale, E. and Michie, D. (Eds.), Vol.2, pp.21–32, Edinburgh at the University Press (1967).
10) Dahl, O.-J., Dijkstra, E.W.D. and Hoare, C.A.R.: *Structured Programming*, Academic Press (1972). 野下，川合，武市（訳）：構造化プログラミング，サイエンス社 (1975).
11) Dijkstra, E.W.D.: Goto Statement Considered Harmful, *Comm. ACM*, Vol.11, No.3, pp.147–148 (1968). 木村（訳）：**go to** 文有害説，22), 第 1 部，pp.346–348.
12) Dijkstra, E.W.D.: Notes on Structured Programming, in 10), Chapter 1, pp.1–82 (1972).
13) Dijkstra, E.W.D.: Structured Programming, *Software Engineering: Concepts and Techniques*, Naur, P., et al. (Eds.), pp.222–226, Petrocelli/Charter (1976); reprinted 3), pp.38–41.
14) Dyer, M.: *The Cleanroom Approach to Quality Software Development*, John Wiley & Sons (1992).
15) Gries, D.: On Structured Programming, *Comm. ACM*, Vol.17, No.11, pp.655–657 (1974); reprinted in 16), pp.70–74.
16) Gries, D. (Ed.): *Programming Methodology: A Collection of Articles by Members of IFIP WG2.3*, Springer-Verlag (1978).
17) 林 晋：プログラム検証論，共立出版 (1995).
18) Hoare, C.A.R.: An Axiomatic Basis for Computer Programming, *Comm. ACM*, Vol.12, No.10, pp.576–580, 583 (1969); reprinted in 19), pp.45–58; in 16), pp.89–100; and in 46), pp.500–505.
19) Hoare, C.A.R.: *Essays in Computing Science*, Jones, C. (Ed.), Prentice-Hall (1989).
20) Hoare, C.A.R. and Wirth, N.: An Axiomatic Definition of Programming Language PASCAL, *Acta Inf.*, Vol.2, No.4, pp.335–355 (1973); reprinted in 19), pp.153–169; also in 46), pp.506–526.
21) Hopkins, M.E.: A Case for the goto, *Proc. 25th National ACM Conf.*, pp.787–790 (1972); reprinted in 44), pp.101–109.
22) 木村 泉（編）：特集 GO TO 論争，*bit*, Vol.7, No.5, pp.346–379 (1975).
23) 木村 泉（抄訳）：GO TO 論争，22), 第 2 部，pp.350–366. （原典：Leavenworth, B. (Ed.): Control Structures in Programming Languages —"The GO TO Controversy"—Debate, *ACM SIGPLAN Notices*, Vol.7, No.11, pp.53–91 (1972).）
24) 木村 泉，米澤明憲：算法表現論，岩波書店 (1982).
25) Knuth, D.E.: Structured Programming with go to Statements, *Comput. Surv.*, Vol.6, No.4, pp.260–301 (1974); reprinted in 43), pp.140–194; in 44), pp.259–321; and in 46), pp.104–144.
26) Knuth, D.E. and Floyd, R.W.: Notes on Avoiding 'Go to' Statements, *Inf. Process. Lett.*, Vol.1, No.1, pp.23–31 (1971); reprinted in 45), pp.153–162.
27) Linger, R.C. and Mills, H.D.: On the Development of Large Reliable Programs, in 43), pp.120–139 (1977).
28) Linger, R.C., Mills, H.D. and Witt, B.I.: *Structured Programming: Theory and Practice*, Addison-Wesley (1979).
29) Mills, H.D.: Top down Programming in Large Systems, *Debugging Techniques in Large Systems*, Rustin, R. (Ed.), pp.41–55, Prentice Hall (1971); reprinted in 32), pp.91–101.
30) Mills, H.D.: Mathematical Foundations of Structured Programming, IBM Report, FSC 72-6012, pp.18–83 (1972); reprinted in 3), pp.42–107; in 32), pp.115–179; and in 45), pp.220–262.
31) Mills, H.D.: The New Math of Computer Programming, *Comm. ACM*, Vol.18, No.1, pp.43–48 (1975); reprinted in 32), pp.215–230.
32) Mills, H.D.: *Software Productivity*, Dorset House (1988).
33) Mills, H.D., Basili, V.R., Gannon, J.D. and

Hamlet, R.G.: *Principles of Computer Programming: A Mathematical Approach*, Wm. C. Brown Publishers (1986).
34) Mills, H.D., Linger, R.C. and Hevner, A.R.: *Principles of Information Systems Analysis and Design*, Academic Press (1986).
35) 中島玲二：数理情報学入門—スコット・プログラム理論，朝倉書店 (1982).
36) Poore, J.H. and Trammell, C.J. (Eds.): *Cleanroom Software Engineering: A Reader*, NCC Blackwell (1996).
37) Prowell, S.J., Trammell, C.J., Linger, R.C. and Poore, J.H.: *Cleanroom Software Engineering: Technology and Process*, Addison-Wesley (1999).
38) Plauger, P.J.: *Programming on Purpose: Essays on Software Design*, Essay 4, Prentice Hall (1993). 安藤（訳）：プログラミングの壷 I —ソフトウェア設計編，pp.42–50, 共立出版 (1995).
39) 佐藤，大槻，金藤：ソフトウェアクリーンルーム手法—高品質ソフトウェア開発パラダイム，日科技連 (1997).
40) Stavely, A.M.: *Toward Zero-Defect Programming*, Addison-Wesley (1999).
41) Strachey, C. and Wadsworth, C.P.: *Continuations: A Mathematical Semantics for Handling Full Jumps*, Technical Monograph PRG-11, Oxford University Computing Laboratory (1974).
42) Tennent, R.D.: *Principles of Programming Languages*, Prentice-Hall International (1982).
43) Yeh, R.T. (Ed.): *Current Trends in Programming Methodology, Vol.I: Software Specification and Design*, Prentice-Hall (1977).
44) Yourdon, E. (Ed.): *Classics in Software Engineering*, Yourdon Press (1979).
45) Yourdon, E. (Ed.): *Writings of the Revolution: Selected Readings on Software Engineering*, Yourdon Press (1982).
46) Wasserman, A.I. (Ed.): *Tutorial: Programming Language Design*, IEEE Computer Society Press (1980).
47) Wirth, N.: Program Development by Stepwise Refinement, *Comm. ACM*, Vol.14, No.4, pp.221–227 (1971); reprinted in 16), pp.321–335; also in 45), pp.99–111.
48) Wirth, N.: *Systematic Programming — An Introduction*, Prentice-Hall (1973). 野下，筧，武市訳：系統的プログラミング／入門，近代科学社 (1975).
49) Wulf, W.A.: A Case against the goto, *Proc. 25th National ACM Conf.*, pp.791–797 (1972); reprinted in 44), pp.85–98.

(平成 15 年 5 月 27 日受付)
(平成 16 年 1 月 6 日採録)

金藤　栄孝（正会員）

1981 年東京大学大学院理学系研究科博士課程中退．同年（株）日立製作所入社．プログラミング言語の意味論・型理論・形式手法等に関心を持つ．EATCS，ACM，IEEE-CS，ソフトウェア科学会各会員．

二木　厚吉（正会員）

1975 年東北大学大学院工学研究科博士課程修了．工学博士．同年電子技術総合研究所（電総研）入所．1983 年〜1984 年 SRI International 客員研究員．1992 年電総研主席研究官．1993 年北陸先端科学技術大学院大学情報科学研究科教授．主な研究分野：フォーマルメソッド，プログラミング方法論，言語設計学，ソフトウェア工学．ACM，IEEE-CS，ソフトウェア科学会各会員．