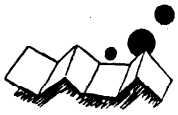
Vol. 28 No. 10　　情　報　処　理　　Oct. 1987

解　説

2. 方　式



2.2 演繹推論による自動プログラミング†

後　藤　滋　樹††

1. はじめに

自動プログラミングの各手法の中で最も原理的に明快なのは，ここで述べる演繹推論による手法であろう．この方法は定理の自動証明の技術を応用してプログラムを自動的に合成するもので，でき上がったプログラムは最初に与えた仕様 (specification) に照らして常に正しいことが保証されている．それは論理的に厳密な証明を経てプログラムが合成されるからにほかならない．

すなわち，演繹的なアプローチによれば，バグのないプログラムを作成することができる．この特徴は一見たいそう魅力的である．バグのないプログラムを作れるならば多少の手間はいとわない，という場合が非常に多いからである．

しかし現実には，演繹的な手法が実用の域に達しているとはとうてい言えない．そこには理想と現実との食い違いをみることができる．

結論を先にいえば，演繹的アプローチの完全な実現のためには，解決しなければならない研究課題が山積している．したがって，研究対象としてははなはだ興味深いものがあるが，実用に供せられるのは少し先のことになるであろう．ただし演繹的なアプローチの研究が進展するとともにプログラムの論理的な構造が現在よりもさらに明確になるから，演繹的アプローチの研究成果は単に自動プログラミングの範囲に留まらず，広くコンピュータ・サイエンス全般に良好な影響を与えるものと期待されている．

本解説では，以上に述べた現状認識にかんがみ，最初に演繹推論による自動プログラミングの原理をなるべく具体的に解説する．さらに，原理を具体的に応用する場合に遭遇する２つの大きな問題点（課題）を述べ，どのような解決法が提案されているかを概観する．

最後の章では，今日大いに普及している論理プログラミング言語（PROLOG など）と本稿の演繹推論による自動プログラミングとの比較を試みる．この両者は同じ発想に立脚しているが，利用者からみた効用は大いに異なっている．

2. 演繹的アプローチ

コンピュータのプログラムと論理式の証明とが酷似していることは昔から指摘されていた事実である[1]．ここで注意が必要なのは，プログラムと論理式とが似ているというのではなく，論理式の「証明」とプログラムとが似ているという点である．ここで証明という用語が突然登場したので，話が急に難しくなったと驚かれた読者もいるかもしれない．実は少しも難しくないのであるが，若干の用語の紹介は必要であろう．

一般に論理体系というのは公理系とも呼ばれており，その基本的な構成要素は**公理**と**推論規則**である (図-1)．ここに公理というのは，論理式の中で特に無条件に成り立つと認められた論理式の集合のことである．公理に推論規則を適用すると**定理**が得られる．定理というのは「証明可能な論理式」の別名である．

一度得られた定理に対して再度推論規則を適用してさらに次の定理を求めてもよい．このようにして定理の個数は次第に増加していく．ある程度複雑な論理式を証明するためには，公理から出発して何度も推論規則の適用を経なければならない．

話を論理とプログラムとの対応に戻そう．繰り返して指摘するが，プログラムに似ているのは証明である．そして証明は推論規則を何度も適用して得られる．これはちょうどプログラムを構成するためには命令をいくつも並べたり，あるいは関数を何段も重ねて適用するのと同じことである．

つまり，簡単にいい切ってしまえば，１個の推論規則が１つの命令あるいは１つの関数に対応するのであ

† Deductive Approach to Automatic Programming by Shigeki GOTO(NTT Software Laboratories).

†† NTTソフトウェア研究所

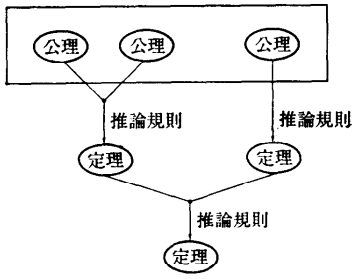


図-1 公理と推論規則

る．この対応付けの正確な説明をするためには，論理体系およびプログラムの厳密なモデルを設定しなければならないが，本解説では細部にわたる話は省略して，だいたいの様子を概観するに留めよう．

演繹的な推論規則の選び方には各種の流儀があり，論理体系ごとに異なった推論規則が設定されている．ここでは代表的な推論規則として次の3種類を選ぶ．

（1） 三段論法

（2） 場合分けの証明

（3） 数学的帰納法

以下では，これらの推論規則を説明するとともに，プログラムとの対応付けを解説しよう．

三段論法 (modus ponens) は分離規則 (detachment) とも呼ばれる．三段論法は最も基本的な推論規則であり，PROLOG などは三段論法のみに頼って論理的な推論を行っている．一般に推論規則は上述のように定理（公理を含む）から新たな定理を導くものであるが，具体的な処理の内容としては，仮定（前提ともいう）から結論を導くものである．三段論法の処理を図示すると**図-2** のようになる．ここでは A と $A \supset B$ とが仮定であり，結論は B である．ここに A, B, C は論理式を表す．

$A \supset B$ という論理式は「A ならば B」と読む．その心は「もし A が成り立つならば，B も成り立つ」というものであるが，これをプログラムのように解釈すると「A という入力があれば B という出力を返す」と読める．三段論法のもう一方の仮定は A であり，これは A が成り立つという意味であるが，A が存在するとも読める．結局，A を入力として使えば，B が出力となる．これが三段論法の結論の B の意味である．

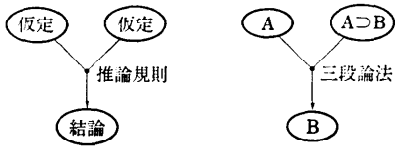


図-2 三段論法の処理

このように考えると，三段論法の処理は $b=f(a)$ という関数 f（実は $A \supset B$）に a（これは A という論理式）を与えて $b=f(a)$（結論の B）を得るのと同じことになる．これがプログラムとしての意味である．

さらに a がすでに $g(x)$ という形をしているとすれば，最終的な結果は $f(g(x))$ というふうに関数の合成になる．これが三段論法に対応するプログラムの一般形である．

場合分けの証明 (case analysis) という用語は日本語としては聞き慣れないが，推論の内容は日常的にもよく使われている．たとえば次のような陳述を考えてみよう．

ア）明日雨が降れば，映画を見に行く．

イ）明日雨が降らなければ，野球を見に行く．

ウ）映画を見に行くことは，外出することである．

エ）野球を見に行くことは，外出することである．

この4つの陳述から，「明日は（雨が降っても，降らなくても）外出する」という結論が導ける．これは，雨が降る場合と降らない場合とに場合分けをして推論したのである．論理的に厳密に議論する場合も同様である．

この場合分けの証明をプログラムとして解釈すると条件文に相当する．これは分かりやすいであろう．**図-3** にはプログラムの雰囲気を出すためにフローチャートを掲げておいた．

最後の**数学的帰納法** (mathematical induction) という推論規則は少し複雑である．誰でも高校生のころ（?）次のような推論規則のことを学んだはずである．

すべての自然数 n について論理式 $P(n)$ が成り立つことを示すためには，次の2つのことを証明すればよい．

（1） 自然数0について，$P(0)$ が成り立つことを

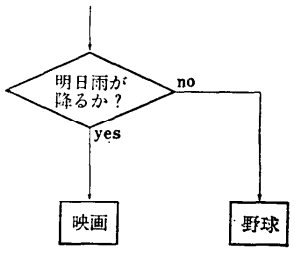


図-3 条件文

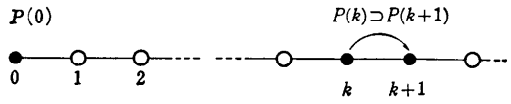


図-4 数学的帰納法

示す.

(2) 任意の自然数 k について $P(k)$ が成り立つと仮定して, $P(k+1)$ についても成り立つことを示す. この様子を図示したのが図-4 である. 具体的に $P(n)$ の形を与えてみると次の例のようになる.

[例題] すべての自然数 n について, その数よりも大きな自然数が存在することを数学的帰納法で説明せよ.

[解答] 題意を論理式で表せば $\forall n \exists z(n<z)$ となる. ここに $\forall n$ は「すべての n について以下が成り立つ」という意味をもつ論理記号である. また $\exists z$ は「以下の条件を満たす z が存在する」という意味である.

ここでは次の2つのことを証明すればよい.

(1) n が0のときには, 証明すべき論理式は $\exists z(0<z)$ となる. これは0よりも大きな自然数が存在するという意味である. これを示すには, z としてたとえば1を選べばよい. 確かに0よりも大きな自然数が存在するといえる.

(2) n が k のときには証明すべき論理式が成り立つと仮定する. すなわち $\exists z(k<z)$ を証明の仮定として使ってよい. この z の具体的な値は分からないが, とにかく自然数には違いない. この自然数を仮りに a と名付けておく. すると $k<a$ という仮定が存在することになるので, $k+1$ に対しては $(k+1)<(a+1)$ とすればよい. 結局 $\exists z(k+1<z)$ という論理式が示せたことになる. (この z は $a+1$ に対応する.)

(1)と(2)とに数学的帰納法を適用すれば求める結果を得る. (証明終)

上の例題の証明をプログラムとして解釈してみよう. このプログラムの入力は n であり, 出力は z である. n を5としてプログラムとしての動作を追跡してみる. n が5のときの論理式は $\exists z(5<z)$ である. 読者はただちに z が6であることを見抜いてしまうであろうが, ここは計算機になったつもりで逐一動作を追跡する.

$\exists z(5<z)$ という論理式は確かに証明可能である. 言い換えれば定理である. その証明は数学的帰納法を用いているが, 帰納法の推論は2つの部分に分かれる. 第一の部分は $\exists z(0<z)$ の証明であり, 第二の部分は $\exists z(k<z)$ から $\exists z(k+1<z)$ を導くものである. 今5は0ではないので, 5を $k+1$ とする証明の部分が関係する. すなわち, $\exists z(5<z)$ という論理式の証明を $\exists z(4<z)$ という仮定のもとで行う. これは $\exists z(5<z)$ の証明を $\exists z(4<z)$ の証明に帰着させていることにほかならない.

同様な動作を繰り返すと, 最終的には $\exists z(0<z)$ の証明に帰着する. このとき, 元の $\exists z(5<z)$ の z に対する答を a_5 とすると $a_5=a_4+1$ となる. (その理由は上の証明の中に示されている.) ここに a_4 とは $\exists z(z<4)$ の z に対する答である. 以下同様に $a_4=a_3+1, \cdots, a_1=a_0+1$, $a_0=1$ となるから $a_5=6$ という答を得る.

このようなプログラムの動作は図-5に掲げる再帰的なプログラムの動作と全く同じものである.

結局上に説明した3つの推論規則は, それぞれ表-1のようにプログラムと対応付けられることが分かった.

```
f(n)=if n=0 then z=1
     else z=f(n-1)+1
```

図-5 再帰的なプログラム

表-1 推論規則とプログラム

推論規則	プログラム
三段論法	関数の合成 $f(g(x))$
場合分け	条件文 if
数学的帰納法	再帰プログラム

3. 論理体系の選び方

普通に「論理」という場合には古典的な一階の述語論理を指すことが多い. 本稿では論理体系の微妙な差異については立ち入らないが, プログラムと論理との対応付けは, 論理体系の選び方によって左右されるという事実には注意しておこう.

たとえば次の形の論理式は**排中律** (excluded middle) と呼ばれるもので, 古典論理においては証明できる論理式である (定理である).

$$P \vee (\neg P)$$

この論理式は P *or* (*not* P) を論理記号で表した式である. その意味は「P が成り立つか, それとも not P が成り立つか, いずれかである」というもので, 一見もっともらしい. しかし, プログラムの立場でこの論理式を眺めると, P または not P という2つの場合を想定しているのだから「場合分けの証明」に関係し

ている．事実，排中律を認めるとプログラムにおいて P に関する条件判定をしてもよいことになるのである．

if P then … else …

ところが，プログラムの動作の観点からは無意味な条件というものがある．たとえば，「明日雨が降る」という条件は，明日になってみれば確かに「明日雨が降る」(P) か，それとも「雨が降らないか」(not P) かのいずれかであることが分かる．しかし今日の時点で条件判定をするのは無理である．

このような問題は，一般化して考えることができる．すなわち，排中律でいう P or (not P) は究極的な真偽の決着を意味しているのであって，プログラムの中でただちに条件判定が可能であるとは限らない．したがって次のような条件をプログラムの中で使うのは無意味である．

・明日A社の株価が上がるか．

・幽霊は存在するか．

・計算の複雑さに関して $P=NP$ が成り立つか．

ところが，古典論理の教えるところでは排中律が常に成り立つから，いかなる P に対しても条件判定が可能なようにみえてしまう．これはプログラムとの対応付けを考える上では都合が悪い．

そこで一般には，古典論理を修正して多少の制限を加えて排中律を用いるか，あるいは抜本的に古典論理に代えて直観主義論理を採用してプログラムと論理式(証明)との対応を考えることが多い．実は PROLOG も直観主義の中で考えるのが自然であることが知られている．この話題に関してはすでに本誌に解説が掲載されているので，ここではこれ以上は立ち入らない[2]．

なお直観主義論理というのは，古典論理から排中律を取り除いた論理体系である．

いずれにしても，自動プログラミングへの演繹的アプローチにおいては，論理体系の選び方についての議論が不可欠である．

4. 自動化の問題

上の排中律の問題は場合分けの証明に関係があった．この章の問題は数学的帰納法に関係する．

そもそも演繹的な方法で「自動」プログラミングが可能であるのは，定理の自動証明が可能であるという事実に基づいている．演繹的な方法の全体の処理の流れを図-6に示す．ここにプログラムの仕様を表す論理式は多くの場合に次の形をしている．

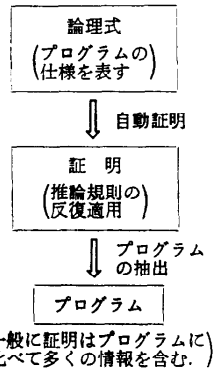


図-6 論理的な自動プログラミング

$$\forall x \exists z (P(x,z))$$

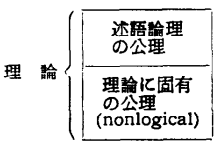
この論理式全体をプログラムとして解釈すると，x は入力を表し，z が出力を表す．$P(x,z)$ というのは入力と出力との関係を記述する論理式である．

さて，このように $\forall$ を含む論理式が与えられたときにそれを自動証明することができるか否かが問題である．答を先に言うと，現在までの自動証明の技術では上の形の論理式を完全に自動的に取り扱うのは難しい．

この答を聞いて，自動証明の技術に詳しい人は奇異に思うかもしれない．なぜなら，一階の述語論理に対しては完全 (complete) な自動証明の技法がいくつも知られているからである．そのような自動証明法を適用すれば問題はただちに解けるのではないか，というのが疑問の内容である．

そこで話の中心は，数学的帰納法の取り扱いになる．上の陳述「一階述語論理に関しては完全な自動証明が可能である」は確かに正しい．問題は数学的帰納法が一階述語論理に含まれるか否かである．この問にはきわめて明快に答えることができる．すなわち，答は NO である．この答に関しては誤解を招きやすいので少し補足説明をしておきたい．

一般に論理を用いて特定の分野の問題を記述するときには，述語論理の公理のほかにその対象となる領域の公理を追加する．このように特定の分野の公理を含



理論の例：群論，自然数論

図-7 理論と nonlogical axioms

む論理体系を理論 (theory) と呼ぶ[3),4)]. 追加された公理を proper axiom とか nonlogical axiom という. 論理の話をしているはずなのに nonlogical というのは少し変な気もするが, こういうときは logic＝述語論理のつもりである. 図-7 には論理と nonlogical axiom の関係を書いておいた.

さて数学的帰納法という推論規則は, 自然数論 (number theory) という理論に属するものである. したがって数学的帰納法が自動証明で取り扱えないからといって述語論理の自動証明が可能であることに反しない. もちろん, 自動証明の分野では数学的帰納法の研究も盛んに行われている. しかし, 理論的に考えても容易なテーマではなく, 単一の原理で簡単に解決するわけではない[5)].

このような次第で自動証明の技法に基礎を置く自動プログラミングは, 肝心の「自動」という性質がそれほど簡単には達成できないことになる. 簡単にという意味は, 強力な一般原理があって, それを適用すればもろもろの問題が解けるというような状態を指す. もちろん, いくつかの有効な原理は分かっているので, それらを組み合わせて個々の問題に対処しているのが現状である.

特に我が国では, これまでの自動プログラミングの研究が論理体系の選び方とプログラムの抽出の部分に重点を置いていたように思われる[6)~11)]. いわば「自動」という視点が弱かったようである. 今後は自動化, すなわち帰納法の自動証明の研究も盛んになることが期待される.

なお, 上の説明では数学的帰納法は自然数のみを扱うように書いたが, 自然数に限らずリストなどのデータ構造も適切な帰納法で特徴付けられる[12)]. したがって自動化と帰納法との関係は一般的に論じることができる.

5. 論理プログラミングとの比較

自動プログラミングの解説をしたはずなのに, 肝心の自動化の部分がいささか弱体であった. それでも演繹的なアプローチ自身の意味はあるので, 相変わらず研究は盛んに行われている.

もっとも, 読者の中には次のような意見を抱いた人もいるであろう. すなわち, 本稿で解説したアイデアは PROLOG などの「論理プログラミング」と寸分違わない. 論理とプログラムとの対応は, すでに論理プログラミングにおいて実用に供されているというべきである. しかも PROLOG においては, プログラムが「自動的」に走っているではないか. いったい, 上の解説の中で数学的帰納法がうんぬんと言ったのは何事であるか. やさしい話をわざと難しく言っただけではないのか.

このような疑問に答えるためには, PROLOG と上述の自動プログラミングとの比較を推論規則のレベルで論じなければならない. すでに本文中で PROLOG が使う推論規則は三段論法だけである, と指摘した. それが本当ならば, すべての計算が三段論法の推論規則で実現できることになりはしないか. 本当に PROLOG では場合分けの証明も数学的帰納法も要らないのであろうか.

話が際立つのは数学的帰納法のほうである. 比較のために先の例題を PROLOG で記述して比較してみよう. 先の証明では 5<6 などと書いたが, "<" は PROLOG の中ですでに特殊な意味をもっているので, 以下のプログラムの中では great (5, 6) などと表す.

```
great (0, 1).
great (K, Z) :- K1 is K-1, great (K1, Z1),
                Z is Z1+1.
```

上のプログラムは2行 (印刷の都合で3行) からなっており, 1行目が 0<1 に相当し, 2行目が k から $k+1$ を導くところである. 先に説明した例題を PROLOG で走らせてみると, 次のようになる.

```
?-great(5, Z).
Z=6
yes
```

つまり答は6というわけである. 5以外の数字を入力してもかまわない.

```
?-great(0, Z).
Z=1
yes
?-great(10, Z).
Z=11
yes
```

確かに, このような答は「自動的」に出力される. ただし, ここが肝要であるが, great (n, Z) の n のところに具体的な数を入れれば, 自動的に答が得られるのである. PROLOG のプログラムを使って $\forall x \exists z(x<z)$ という論理式を証明しているのではなく, 具体的な数, たとえば5とか0とか 10, に対して z の値を返しているに過ぎない. この点を強調して描いたの

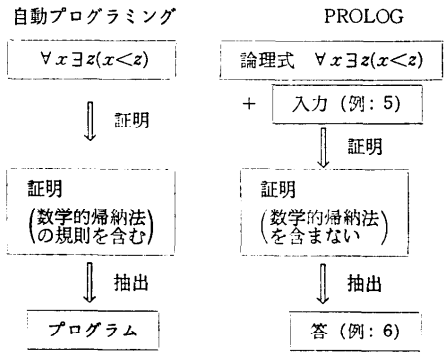


図-8 自動プログラミングと PROLOG

が図-8 である.

すなわち，自動プログラミングにおいては一般の n について証明を済ませておいてからプログラムを抽出する．これに対して PROLOG では入力が与えられてから証明をスタートするので，数学的帰納法を用いる必要がない．$P(0)$ と $P(k)\supset P(k+1)$ を $k=0, 1, 2, 3, 4$ に対して用いるだけで $P(5)$ は示すことができるから，入力が5と分かっていれば数学的帰納法を使うまでもない．

もう1つの推論規則，場合分けの証明，についても簡単に述べておく．PROLOG のプログラムは文字どおりに書いてある順番に上から実行するのではない．PROLOG のプログラムはユニフィケーションと呼ばれる一種のパターン・マッチングによって制御される．このときにマッチングに成功するのは，ある種の条件を満たしている行である．たとえば先の great の例題では，2行のプログラムのうちのどちらかが選ばれる．great(0, Z) ならば1行目，great (5, Z) ならば2行目が選ばれるのは，このユニフィケーションの作用である．

したがって，PROLOG の中で条件文に相当した動作を行わせることは容易にできる．ただし，PROLOG のプログラムというものは，原則として答を1つ見つければ正常に停止する．そこで，正常に停止した場合の痕跡には，特に条件を判定した証拠が残らない．つまり，目の前には二股の分岐があっても，選択した後には一筋の道だけを記憶しておけばよいのと同様である．これが一般に PROLOG は三段論法だけしか使わない，と表現される理由になっている．

6. おわりに

演繹推論による自動プログラミングについて解説した．プログラムと演繹的な推論規則との対応関係は明快である．ただし，論理体系の種類によっては自然な対応関係が損なわれることもあるから，十分な注意が肝要である．

数学的帰納法は再帰的なプログラムに対応するものであるが，自動証明の技法で完全に扱うわけにはいかない．したがって，「自動」プログラミングとしては多くの研究課題が残っている．

ここに述べたプログラムと論理式との対応関係は，いわゆる論理プログラミング (PROLOG が代表) と同じ土俵に乗っている．しかし両者の違いは明確であって，自動プログラミングでは証明を済ませてからプログラムを抽出するのに対して，PROLOG では証明をしながら計算を同時に実行している．

参考文献

1) Manna, Z.: Mathematical Theory of Computation, McGraw-Hill (1974).
いわゆるプログラムの理論の代表的な教科書である．

2) 後藤滋樹，古川康一：論理型計算モデル，情報処理，Vol. 24, No. 2, pp. 123-132 (Feb. 1983).
この文献の中では直観主義という言葉は使っていないが Harrop formula の説明がそれに相当する．

3) Shoenfield, J. R.: Mathematical Logic, Addison-Wesley (1967).
コンピュータ科学の分野で好んで引用される論理学の本である．特に theory の記述は優れている．

4) 小野 寛：プログラムの基礎理論，サイエンス社 (1975).
コンパクトな本であるが theory のことも数学的理論として記述されている．

5) Boyer, R. S. and Moore, J. S.: A Computational Logic, Academic Press (1979).
数学的帰納法を計算機で実行するための方法を述べたものである．これを読むと帰納法の取扱いが難しいことがよく分かる．

6) Goto, S.: Program Synthesis from Natural Deduction Proofs, International Joint Conference on Artificial Intelligence, pp. 339-341 (1979).
ゲーデル解釈と呼ばれる手法で論理式の証明図からプログラムを抽出する方法を述べている．

7) Sato, M.: Toward a Mathematical Theory of Program Synthesis, International Joint Conference on Artificial Intelligence, pp. 757-762 (1979).
標題どおりに理論的な基礎づけを目指したものである．

8) Hagiya, M: A Proof Description Languages and Its Reduction System, Department of Information Science, Tech. Report 82-03, University of Tokyo (Feb. 1982).
証明図をリダクションすることにより，あたかもプログラムのように実行することができる．そのリダクションの戦略の研究である．
9) Goto, S: Concurrency in Proof Normalization, International Joint Conference on Artificial Intelligence, Los Angeles, pp. 726-729 (1985).
証明図のリダクションにおいて無限の要素を取り扱う方法を述べたものである．
10) Hayashi, S. and Nakano, H.: PX a Computational Logic, RIMS-573, Research Institute for Mathematical Sciences, Kyoto University (Apr. 1987).
証明図から realizer と呼ばれる手法でプログラムを抽出する研究である．
11) Sato, M.: Quty: A Concurrent Language Based on Logic and Function, Logic Programming: Fourth International Conference, pp. 1034-1056 (1987).
プログラム言語でもあり，また論理体系でもあるような計算機言語の提案．
12) Manna, Z. and Waldinger, R.: The Logical Basis for Computer Programming, Vol. 1: Deductive Reasoning, Addison-Wesley (1985).
帰納法とデータ構造との関係がよく記述されている．

(昭和 62 年 7 月 27 日受付)