ICOT Technical Report: TR-552

TR-552

帰納法を用いる定理証明システム

金森　直，藤田　博(三菱)

April, 1990

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

目次

１．　はじめに

大多数の読者は、一階推論については読んだり聞いたりしたこともあり馴染みが深いであろう。数理論理学の教科書を開くと、だいたい命題論理の説明から始まるが、ハイライトは一階述語論理で、Hilbert型やGentzen型の証明システムが紹介されていて、（ちょっと意味が取りにくいところもあるけれど）だいたい至極当たり前の推論規則が並んでいる。さらにパラパラとページを繰ってみると、これを使って証明できることが“正しい”こと（健全性）だけでなく、“正しい”ことはこれを使えばとにかく何とか証明できること（完全性）まで保証されているらしい。また、人工知能の教科書を開くと、“論理による知識の表現”などという章の後にはresolutionと呼ばれる証明システムが紹介されている。これは前のHilbert型やGentzen型の証明システムとは随分印象が違うけれど、計算機での処理に適したように工夫された一階推論のシステムだということで、やはり健全で完全だと説明されている。（しかも、あのPrologの動きはこのresolutionの特別な場合だという。）これらの教

科書をもう少しちゃんと読んでみると、“正しい”というのは、正確には“すべてのモデルについて成り立つ”の意味で、一階推論はどうやら“すべてのモデルについて成り立つ論理式を証明する”ためのものらしい。（この辺りは本特集の後藤滋樹氏の解説を御覧頂きたい。）

これに対して、数学的帰納法については高校生の頃に習っており、“一階推論”などという仰々しい名前に比べると早くから馴染んでいるはずなのに、帰納法が重要だという感じはそれほど一般的でない。だいたい数理論理学の教科書には、一階述語論理の後に自然数論(Number Theory)の章があり、読んでみると“数学的帰納法”という次のような推論規則が追加されている。

“ある性質が

Base Case ：０について成り立つ

Induction Step：Ｎについて成り立つならＮ＋１についても成り立つ

ならばこの性質はすべての自然数について成り立つ”

しかし、後は至極当たり前のことばかり書いてあり、最初は一体何が問題なのかピンとこない。出てくる論理式も一階論理式で、普通の一階推論と何が違うのかも見え

にくい。この自然数論の場合は自然数という対象を扱ったが、プログラムではさらにいろいろな対象を扱うことになる。そういったプログラムの性質を証明することは自然数論の証明に対応し、そのためには数学的帰納法に対応する推論規則が必要になる。荒っぽく言えば、帰納法はそういった“特定のモデルについて成り立つ性質を証明する”ためのものであるが、一階推論のような完全な証明システムは、“一般には”不可能であることが知られている。（この辺りも本特集の後藤滋樹氏の解説を御覧頂きたい。）そういったスカッとした証明システムが不可能であることが、一階推論に比べて帰納法が一般受けしない理由の一つかもしれないが、だからといって、帰納法による推論が重要でないということではない。実は多少とも興味ある性質は帰納法なしには証明できない。（『数学的帰納法を使うな。』と言われたら、整数論を専攻する数学者は失業してしまうだろうし、『帰納法を使ってはいけない。』ということだったら、プログラム検証で示せることは高が知れている。）

本解説では定理証明システムのうち、特に帰納法を用いるものを解説する。以下、第２章ではBoyer-Moore 定

理証明システム(BMTP)を中心にLispプログラムの性質を帰納法を用いて証明するシステムについて[2],[4]、第３章では我々のArgus 検証システムを中心にPrologプログラムの性質を帰納法を用いて証明するシステムについて述べる [11],[12],[13]。特に両システムの帰納法の比較・対照のために、多少図式的かもしれないが、それぞれの章の節の構成まで対応させてある。(このため、本来のシステムの姿から見ると強調の仕方や比重の置き方に多少正確でないところも生じるかもしれない。その辺りは原著に当たって頂きたい。)

２．Boyer-Moore 定理証明システム

2.1 停止的なLispプログラム

Lispプログラムは下のPLUS* のような再帰的Lisp関数を含むことによって面白く有意義な性質をもつことになるが、これが非再帰的関数ばかりだとプログラム自体もその性質も取るに足らないものになってしまうに違いない。

(PLUS X Y)=

(IF(ZEROP X)(FIX Y)(ADD1(PLUS(SUB1 X)Y)))

Boyer-Moore の定理証明システムは、二つ以上の関数が互いに呼び合う（相互再帰する）ことがない、などの条件を満たす適切に定義されたLisp関数を対象にすることにより、必ずしも自明でないLispプログラムの性質を証明しようというものである。

適切に定義されたLisp関数とは、端的にいえば“停止的な”関数であるということである。では、再帰的Lisp関数が停止的であるということはどのようにして確かめ

*(FIX Y)はY が自然数であるときはY を、そうでないときは 0を返すLisp関数とする。

られるのてあろうか。それには、その関数の再帰呼び出しが親の呼び出しより何らかの基準によって常に“減少”していること、かつ無限に減少するということはなく、必ず“最小”のところ（それ以降は再帰呼び出しが行われないという点）に到達することが言えればよい。そのような基準のことを有礎順序（Well-founded Ordering）という。

上のPLUSでは、再帰呼び出し(PLUS(SUB1 X)Y) が親の

呼び出し(PLUS X Y)に比べて第1引き数の大きさが減少しており、いずれ 0に到達してそれ以降は再帰呼び出しは行われないということがわかる。ここで用いた有礎順序は、自然数の大小に従う普通の順序である。

2.2 有礎帰納法としての数学的帰納法

ある有礎順序をもとに停止的であることが確かめられたLisp関数を対象とすると、その性質についての命題を証明する際にも同じ有礎順序をもとにした帰納的な推論を行うことが可能となって都合が良いのである。このような有礎順序に基づく推論のことを有礎帰納法といっている。

例えば、(PULS X Y)を含む命題(p X) が成り立つことを証明したいとしよう。このとき、我々は(p X) を直接証明するかわりに次の二つが成り立つことを証明すれば良いというのである。

```
base case : (IMPLIES (ZEROP X)(p X))
induction step :
 (IMPLIES (AND (NOT (ZEROP X))(p(SUB1 X)))(p X))
```

これはまさに普通の数学的帰納法そのものである。

2.3 一般の有礎帰納法

Lispプログラムは自然数以外にリストをはじめ文字列、そのほかの領域（データ型）を対象にするので、それらの領域に対しても適用できるような帰納的推論法がほしくなる。では、自然数以外の領域を対象にしたときの帰納法はどのようになるのであろう。例えば、下のAPPENDのように対象領域がリストであるようなプログラムを考えてみよう。

```
(APPEND X Y)=
  (IF(LISTP X)(CONS (CAR X)(APPEND(CDR X)Y))Y)
```

ここで、X がリストのときその“大きさ”を(CAR X)の大きさと(CDR X) の大きさの和プラス１とし、リストでないときは０としよう。このようにリストの大きさを自然数に対応づけると、リストの領域にも自然な有礎順序が導入される。

この有礎順序によれば、PLUSの例と同様、上のAPPENDの再帰呼び出し(APPEND(CDR X)Y)が親の呼び出し(APP-

END X Y)に比べて第１引き数の大きさが減少しており、いずれ葉（リストでないデータ）に到達してそれ以降は再帰呼び出しが行われないということが確かめられる。

次に、(APPEND X Y)を含む命題(p X) が成り立つことを証明したいとしよう。このとき、(p X) を直接証明するかわりに次の二つが成り立つことを証明すれば良いというのである。

```
base case : (IMPLIES(NOT (LISTP X))(p X))
induction step :
  (IMPLIES(AND(LISTP X)(p(CDR X)))(p X))
```

これは、普通の数学的帰納法を一般化したものと言うことができよう。

このように自然数以外の領域を対象にするときにも、その領域に適当な有礎順序が定義されていさえすれば有礎帰納法が適用できる。Boyer-Moore 定理証明システムでは“shell"と呼ばれる機構を用意して、有礎順序つきの任意の領域を定義できるようにしている。

shell は一種の抽象データ型と考えることができ、構成子(constructor function)、抽出子(access function)、型識別子(recognizer function) などがひと揃いと

なって定義される。リストでは、構成子はCONS、抽出子はCAR 、CDR 、型識別子はLISTP である。このとき、次のような公理が導入される。

(LISTP(CONS A B))

(EQUAL(CAR(CONS A B))A)

(EQUAL(CDR(CONS A B))B)

さらに、

(IMPLIES(LISTP X)(CDR-LESSP(CDR X)X))

が定理となるような適当な有礎順序CDR-LESSP が定められる。

さて、実際に与えられたゴールに対して有礎帰納法を適用しようとするとき、どの領域の帰納法図式を選びどの変数をターゲットに定めればよいであろうか。

そのヒントは与えられたゴールに含まれている再帰的関数にある。その再帰的関数は停止的であることが有礎順序によって確かめられているのであるが、そのような再帰的関数の定義形を見れば、適用可能な帰納法図式の形がおのずと決まってくるのである。

まず、その関数の対象領域に注目すると、その領域に関してshell で定められた有礎順序を用いた帰納法図式

がまず候補に挙がる。

しかし、shell による帰納法図式でいつでもうまくいくとは限らない。一般の有礎帰納法はむしろ関数定義から直接に導かれるより複雑な帰納法図式を用いることになる。その仕組みはおおよそ以下のようである。再帰的関数が停止的であることを確かめる段階で、実はもっと複雑な有礎順序が必要になる場合がある。例えば、再帰呼び出しが親呼び出しより“減っている”のが単純にCDR のようにshell で定められた抽出子によるのではなくLASTのような別の再帰的関数によっている場合や、ただ１箇所の引き数に注目したのではすべての再帰呼び出しの“減少”を保証できない場合である。前者の場合には、リストの大きさを計る計測関数(measurefunction)にLASTを用いた新たな有機順序を導入すればよいし、後者の場合には、幾つかの引き数の組(measured subset)に注目して直積領域を考え、辞書的順序(lexicographic ordering)によって新しい有礎順序を構成することを試みればよい。このようにして得られる有礎順序に基づいた帰納法図式は、その有礎順序によって停止的であることが確かめられた再帰的関数自身の構造を反映すること

になる。すなわち、関数の定義体のIf分岐に応じて、再帰呼び出しを含む枝の一つ一つをそれぞれ一つのinduction stepに、帰納呼び出しのない枝はまとめてbase caseに対応づけるのである。詳細は“A Computation Logical”を参照してほしい。

2.4 有礎帰納法図式の併合など

Boyer-Moore 定理証明システムでは証明するゴールに幾つもの有礎帰納法図式が適用できるときには、併合、除去、吸収、比較などの処理を施して最も良さそうな有礎帰納法図式が適用されるが、それらの処理の中でも最も効果の大きいのは有礎帰納法図式の併合である。

次のゴールを考えてみよう。

```
(IMPLIES(AND(LESSP X Y)(LESSP Y Z))(LESSP X Z))
```

このゴールには三つの項がある。ここで、例えば(LESSP X Y) に注目すると次の二つの帰納法図式が考えられる。

```
base case : (IMPLIES(ZEROP X)(p X Y))
induction step :
  (IMPLIES(AND(NOT(ZEROP X))
```

```
                (p(SUB1 X)(SUB1 Y)))
        (p X Y)                                    … (1)
base case : (IMPLIES(ZEROP Y)(p Y X))
induction step :
  (IMPLIES(AND(NOT(ZEROP Y))
               (p(SUB1 Y)(SUB1 X)))
        (p Y X)                                    … (2)
```

これは、(LESSP X Y) の定義が、

```
  (LESSP X Y)=
    (IF(ZEROP Y)F
      (IF(ZEROP X)T(LESSP(SUB1 X)(SUB1 Y))))
```

となっていて、X 、Y のそれぞれで個別に有礎順序が与えられるからである。この二つの帰納法図式を次のように一つに併合することにしよう。

```
base case :
  (IMPLIES(OR(ZEROP X)(ZEROP Y))(p X Y))
induction step :
    (IMPLIES(AND(NOT(ZEROP X))
                 (NOT(ZELOP Y))
                 (p(SUB1 X)(SUB1 Y)))
```

```
(p X Y))                                    …(3)
```

(3) の帰納法図式は果たして“安全”なのであろうか。(3) を適用した結果できるゴールはbase case については、

```
(IMPLIES(ZEROP X)(p X Y))                   …(3-1)
(IMPLIES(ZEROP Y)(p X Y))                   …(3-2)
```

の二つのゴール（のAND)、induction stepについては、

```
(IMPLIES(NOT(ZEROP Y))
  (IMPLIES(AND(NOT(ZEROP X))
               (p(SUB1 X)(SUB1 Y)))
           (p X Y)))                        …(3-3)
```

という形のゴールと論理的に等価である。(3-1) は(1) のbase case と同一であるし、(3-3) 式の外側の帰結部は(1) のinduction Stepと同一である。これは、(1) のinduction stepを(ZEROP Y) と(NOT(ZEROP Y))で場合分けし、前者の内側の前提（帰納法の仮定）を除いたものが(3-2) で、後者が(3-3) になっているとみることができる。そもそも(1) によって導かれる二つのゴールを証明すれば十分であるというのだから、その一部を(3-2) のように前提条件を緩めて“強めた”ゴールと(3-1) と

(3-3) の三つが証明されればやはり十分であって論理的な不都合はない。

実は、もとのゴールに現れている残りの二つの項、(LESSP Y Z) と(LESSP X Z) からも二つづつ帰納法図式が得られて、最終的にはそれらすべてが併合された次の帰納法図式が用いられる。

```
base case :
  (IMPLIES(OR(ZEROP X)(ZEROP Y)(ZEROP Z))
           (p X Y Z))
induction step :
  (IMPLIES(AND(NOT(ZEROP X))
               (NOT(ZEROP Y))
               (NOT(ZEROP Z))
               (p(SUB1 X)(SUB1 Y)(SUB1 Z)))
           (p X Y Z)
```

このような帰納法図式に対する併合やそのほかの操作とそれらの正当性についての議論の詳細は“A Computational Logic”を参照して頂きたい。

2.5 有礎帰納法とともに使われる推論

Boyer-Moore 定理証明システムは帰納法を用いることに特徴があるのだから、彼らのシステムを解説した本“A Cpmputational Logic”はほとんどが帰納法の解説であろうと思われるかもしれない。しかし、彼らの本のうち直接に帰納法を扱っている部分は案外少ない。では、それ以外は何が書いてあるかと言えば、帰納法をうまく使って証明できるように帰納法以外の推論をどう併用するかが書いてあると言っていい。“帰納法を用いる”からと言って、“帰納法だけを用いる”わけではない。最初にも言ったように帰納法は“特定のモデルで成り立つ性質を証明する”ためのものであるが、“すべてのモデルで成り立つ性質を証明する”ための推論を併用しても何も問題はない。

実際には、関数呼び出しをその定義体で展開したり、shell の機構によって導入された幾つかの公理や、既に証明済みの定理のうち帰結部が等式となっているものを書き換え規則として使い、ゴールを書き換えて簡略化するという推論ステップが併用される。

rewriting terms と simplifying claues

次のゴールREVERSE _REVERSE :

```
(IMPLIES(PLISTP X)
        (EQUAL(REVERSE(REVERSE X))X))
```

に帰納法を適用すると、次のbase case が得られる。

```
(IMPLIES(AND(NOT(LISTP X))(PLISTP X))
        (EQUAL(REVERSE(REVERSE X))X))      ……①
```

ここで、PLISTPとREVERSE は、次のように定義されている。

```
(PLISTP X)=
  (IF(LISTP X)(PLISTP(CDR X))(EQUAL X NIL))
(REVERSE X)=
  (IF(LISTP X)(APPEND(REVERSE(CDR X))
                     (CONS(CAR X)NIL))
              NIL)
```

従って、前提条件(NOT(LISTP X))のもとに(PLISTP X)は(EQUAL X NIL) に展開され、(REVERSE X) はNIL に展開される。さらに、(REVERSE NIL) はNIL に展開されて、①の帰結部は前提部に含まれる条件と等価の(EQUAL NIL X) となるので①が真であることが証明される。

そのほか、以下に示すような４つのヒューリスティックな書き換え規則が適用される。

destructorの除去

```
(IMPLIES
  (AND(LISTP X)
      (EQUAL(REVERSE(REVERSE(CDR X)))(CDR X))
      (PLISTP(CDR X)))
  (EQUAL(REVERSE(APPEND(REVERSE(CDR X))
                       (CONS(CAR X)NIL)))
        X))                                    ……②
```

②は、ゴールREVERSE _REVERSE に帰納法を適用したときのinduction stepである。(LISTP X) という前提は、X が実は(CONS A B)という形をしているということと等価である。

そこで、X のすべての出現を(CONS A B)で置き換えて、

```
(IMPLIES(AND(EQUAL(REVERSE(REVERSE B))B)
             (PLISTP B))
  (EQUAL(REVERSE(APPEND(REVERSE B)
                       (CONS A NIL)))
        (CONS A B)))                           ……③
```

というゴールを得る。この書き換えには、shell で導入された公理を用いた。③では(LISTP X) の前提が不要になり、REVERSE やPLISTPの引き数の形が簡単になって後の推論が簡単になることが期待されるのである。

相互交配(Cross-fertilization)

③の帰納部の等式の右辺でB をREVERSE(REVERSE B))に置き換えると、

```
(IMPLIES(PLISTP B)
  (EQUAL(REVERSE(APPEND(REVERSE B)
                        (CONS A NIL)))
         (CONS A(REVERSE(REVERSE B)))))     ……④
```

となる。即ち、前提部の等式を書き換え規則として帰結部の等式を書き換えるに用いたのである。帰納部の等式の両辺の複雑さの差がより小さくなった分、以後の証明がやりやすくなることが期待されるのである。

一般化(Generalization)

④において(REVERSE B) を一般化してC に置き換える。

```
(IMPLIES(PLISTP B)
  (EQUAL(REVERSE(APPEND C(CONS A NIL)))
```

(CONS A(REVERSE C))))　　……⑤

⑤は④より“強い”ゴールになってしまうが、簡単な形になった分証明しやすくなることが期待されるのである。

irrelevance の除去

⑤においてその前提部(PLISTP B)は今や帰結部の真偽に直接関係しないので除去してもよいだろう。こうして、

(EQUAL(REVERSE(APPEND C(CONS A NIL)))
　　(CONS A(REVERSE C)))　　……⑥

と書き換えることができる。

⑥についてはもうこれ以上の簡単化ができないので、再び帰納法が適用される。帰納法で作られたゴールには上と同様にして書き換え規則の適用が繰り返された後、すべてが真となったとき証明が完結する。

３．　Argus 検証システム

３．１　Prologプログラム

Prologプログラムは、DEC10 Prolog[1] の記法に従えば、例えば次のような形をしている。

```
number(zero).
number(suc(N)) :- number(N).
add(zero,Y,Y).
add(suc(X),Y,suc(Z)) :- add(X,Y,Z).
```

読者もよく御存知のように、このプログラムは次のような一階述語論理式の集合に対応している。（以後、論理式についても、変数、定数記号、関数記号などについては、DEC10 Prologの記法を踏襲することにする。また、簡単のため、論理式の一番外側の∀による束縛は取り除いて書くことにする。）

```
number(zero)
number(N) ⊃ number(suc(N))
add(zero,Y,Y)
add(X,Y,Z)⊃ add(suc(X),Y,suc(Z))
```

この対応を活かして、Prologプログラムの意味を、一階述語論理の特殊なモデルである最小Herbrandモデルで定義するのが普通である。“最小”という形容詞がつくのは、trueに解釈するアトムの集合が最も小さいせいである。例えば、上のプログラムの最小Herbrandモデルは、領域は{zero, suc(zero), suc(suc(zero)), …}で普通の自然数の集合（と同型）で、numberやadd はごく常識的に解釈したものである。（一階述語論理の基本的用語やモデルについては、本特集の後藤滋樹氏の解説、あるいは[15]を御覧頂きたい。）

第２章では、Lispプログラムの性質はＳ式で表現されることにしたが、この第３章ではPrologプログラムの性質は一階述語論理式で表現されるものとしよう。すると、このPrologプログラムの意味に従って、“ある性質が与えられたPrologプログラムについて成り立つこと”を、“この論理式がそのPrologプログラムの最小Herbrandモデルの上で成り立つこと”と考えても、それほどおかしくはないだろう。

３．２　計算帰納法としての数学的帰納法

前の節に従えば、論理式が与えられたPrologプログラムの最小Herbrandモデルの上で成り立つことを示す必要がある。そこで、まず、高校生の頃にやった数学的帰納法を思い出してみよう。例えば、あまりにも簡単な例であるが、『どんな自然数に0を足しても元の自然数に等しい。』という性質がある。これまで使ってきた論理式を用いて表現すれば、これは次のように書ける。

number(N) ⊃ add(N,zero,N)

これはどうやって証明しただろうか。次の二つの場合を証明すればよかった。

『 add(zero,zero,zero)が成り立つ。』

『 add(N,zero,N)が成り立つとすると

add(suc(N),zero,suc(N))が成り立つ。』

論理式を用いて表現すれば、

add(zero,zero,zero)

add(N,zero,N) ⊃ add(suc(N),zero,suc(N))

である。一般には Q(N) を証明したい性質を表す（ Nだけが束縛されていない）論理式として

number(N) ⊃ Q(N)

を証明したければ、

Q(zero)

Q(N)⊃Q(suc(N))

を証明すればよい。図で書けば次のようになる。

$$\frac{Q(zero) \qquad\qquad Q(N) \supset Q(suc(N))}{number(N) \supset Q(N)}$$

図 3.2 数学的帰納法の図式

ここで述語numberを定義するPrologプログラムを思い出そう。対応する論理式で書けば、

number(zero)

number(N) ⊃ number(suc(N))

である。これを先ほど数学的帰納法の図式で生成した新しいゴールと比べてみていただきたい。証明したいゴール　number(N) ⊃ Q(N)

の前提部分の述語numberを定義するPrologプログラムでその述語numberを結論部分のQで置き換えたものにほかならないことに気がつかれただろう。

３．３　一般の計算帰納法

前の節であげたPrologプログラムと数学的帰納法との対応を一般化してみよう。リストを逆順に並べるrev というPrologプログラム

```
rev([ ],[ ]).
rev([X | L],M) :- rev(L,N),ap(N,[X],M).
ap([ ],M,M).
ap( [X | L],M, [X | N]) :- ap(L,M,N).
```

について性質

G_0 : rev(L,M)⊃ rev(M,L)

を証明したいとしよう。前の節との対応で言えばQ(L,M)をrev(M,L)として

rev(L,M)⊃ Q(L,M)

である。前と同様に上のプログラムで revをすべてQで置き換えると

G_1 : rev([ ],[ ])

G_2 : rev(N,L)∧ ap(N,[X],M) ⊃ rev(M,[X | L])

が新しいゴールになる。一般には次の図式が対応する。

$$\frac{Q([\,],[\,]) \qquad Q(L,N) \wedge ap(N,[X],M) \supset Q([X \mid L],M)}{rev(L,M) \supset Q(L,M)}$$

図 3.3.1 rev(L,M)を前提部分とする計算帰納法

どうしてこんなことをしていいのだろうか。Clark[3]の説明を借りよう。DをPrologプログラムに現れる定数記号と関数記号から作られる変数を含まないすべての項の集合としよう。この集合はPrologプログラムが与えられれば固定される。さてrev というのは、このDの上の二項関係で、

(1) 対([ ],[ ]) を含み、

(2) rev の関係が対 ([s | t1],t)を含み、apの関係が三つ組(t,[s],t2)を含むとき対(t1,t2) を含む

ような最小の関係であった。もしQが同じ条件を満たすなら、すなわち論理式で書くと

(1) Q([ ],[ ])

(2) Q(L,N)∧ap(N,[X],M) ⊃Q([X | L],M)

が成り立てば、QはDの上のrev を包含する関係のはず

である。論理式で書くと

rev(L,M)⊃ Q(L,M)

が最小Herbrandモデルの上で成り立つということになる。（したがって、プログラムはBoyer-Moore 定理証明システムのように停止的なものに限らなくてもよい。）

前の節の例のnumber(N) でもこの例のrev(L,M)でも前提部分のアトムの引き数は異なる変数であった。そうでない場合も計算帰納法を使えるだろうか。例えば前提部分のアトムがap(L,[Y],N) のとき計算帰納法を使えるだろうか。いま行ったばかりの理由づけをもう一度考え直してみよう。今の議論が p(X1,X2,...,Xn)という形のアトムについてうまくいったのは、このアトムの代入例で最小Herbrandモデルに含まれるすべてのアトムの集合が対応する確定節で示される条件を満たす最小の関係であったせいである。したがって、もともとのPrologプログラムに含まれていなくてもそういう確定節に対応するものがあれば計算帰納法を使っていいことになる。例えば前提部分のアトムがap(L,[Y],N) のとき、元のapのプログラムの代入例である

ap([ ],[Y],[Y]).

ap([X | L],[Y], [X | N]) :- ap(L,[Y],N).

を考えると、ap(L,[Y],N) の代入例で最小Herbrandモデルに含まれるものをすべて計算するプログラムになっている。したがって、

ap(L,[Y],N) ⊃ Q(L,Y,N)

を証明するための次のような図式が得られる。（詳細は[10]を御覧頂きたい。）

$$\frac{Q([\,],Y,[Y]) \qquad Q(L,Y,N) \supset Q([X \mid L],Y,[X \mid N])}{ap(L,[Y],N) \supset Q(L,Y,N)}$$

図 3.3.2 ap(L,[Y],N) を前提部分とする計算帰納法

３．４ 計算帰納法図式の併合

我々のArgus 検証システムでも、証明するゴールに幾つもの計算帰納法図式が適用できるときには、併合、除去、吸収、比較などの処理を施して最も良さそうな計算帰納法図式が適用されるが、それらの処理の中で最も効果の大きいのはやはり計算帰納法図式の併合である。

前の節では、前提部分のアトムがただ一つの場合だけを（意図的に）考えてきた。もし、そういうアトムが二つ以上あったらどうしたらいいだろう。例えば、

$$rev(N,L) \wedge ap(N,[X],M) \supset rev(M,[X \mid L])$$

を証明したかったとしよう。この論理式を

$$rev(N,L) \supset (ap(N,[X],M) \supset rev(M,[X \mid L]))$$

と考えるとQ_1を、

$$ap(N,[X],M) \supset rev(M,[X \mid L])$$

として計算帰納法が適用できる。この論理式を

$$ap(N,[X],M) \supset (rev(N,L) \supset rev(M,[X \mid L]))$$

と考えるとQ_2を、

$$rev(N,L) \supset rev(M,[X \mid L])$$

として計算帰納法が適用できる。どちらか一つを選んで使えばいいのだろうか。この例では、どちらの計算帰納法を使っても、変数Ｎには同じ形の項が代入されることに注意しよう。実は、この二つの計算帰納法は同時に適用できる。考え方は元のプログラムの代入例を使った前の節と同じである。もともとのPrologプログラムに含まれていなくても、何か正当化するものがあれば計算帰納法を使ってもよい。

そのために、ここで玉木－佐藤のPrologプログラム変換[19]を考えよう。まずは手っ取り早く例を見てみよう。最初に初期プログラムP_0として、

C_1 : rev([ ],[ ]).

C_2 : rev([X | L],M) :- rev(L,N),ap(N,[X],M).

C_3 : ap([ ],M,M).

C_4 : ap([X | L],M, [X | N]) :- ap(L,M,N).

C_5 : new(L,M,N,X) :- rev(N,L),ap(N,[X],M).

が与えられているとしよう。（それぞれの確定節を指すためにC_1、C_2、C_3、C_4、C_5といった名札を付けることにする。）まず、確定節C_5の本体の最初のアトムrev(N,L)に注目しよう。このアトムはrevを定義するC_1の頭部と単一化できる。C_5でアトムrev(N,L)をC_1の本体で置き換えた後、その最汎単一化代入を確定節全体に施すと一つの確定節が得られる。これをC_6としよう。同じことをC_2について行うと、やはり一つの確定節が得られる。これをC_7としよう。そこでC_5をC_6とC_7で置き換えた新しいプログラムをP_1とする。

C_6 : new([ ],M,[ ],X) :- ap([ ],[X],M).

C7 : new(L,M,[Y | N],X) :-

```
rev(N,L1),ap(L1,[Y],L), ap([Y | N],[X],M).
```

一般には、確定節の本体のアトムと単一化できる頭部を持つすべての確定節について本体で置き換えた後、その最汎単一化代入を確定節全体に施して得られる確定節で元の確定節を置き換えればよい。これを“展開規則”と呼ぶ。直感的にいえば、本体のアトムの実行を前もってすべての方向に１ステップだけ進めておくことに相当する。同様に確定節C_6の本体のアトムと、C_7の本体の３番目のアトムについて展開規則を使うと、C_6をC_8、C_7をC_9で置き換えたプログラムP_3が得られる。

```
C8 : new([ ],[X],[ ],X).
C9 : new(L,[Y | M], [Y | N],X) :-
        rev(N,L1),ap(L1,[Y],L),ap(N,[X],M).
```

さてここでC_9の本体に注目しよう。１番目のアトムと３番目のアトムを合わせると、最初のプログラムP_0のnewを定義する確定節の本体と同じ形をしている。そこでC_9の本体のそれら二つのアトムを対応する頭部で置き換えた新しいプログラムをP_4とする。

```
C1 : rev([ ],[ ]).
C2 : rev([X | L],M) :- rev(L,N),ap(N,[X],M).
```

C_3 : ap([ ],M,M).

C_4 : ap([X | L],M, [X | N]) :- ap(L,M,N).

C_8 : new([ ],[X],[ ],X).

C_{10}: new(L,[Y | M], [Y | N],X) :-

new(L1,M,N,X),ap(L1,[Y],L).

一般には、確定節の本体のアトムの集合が最初のプログラムの確定節の代入例のとき、それらのアトムを対応する頭部の代入例で置き換えて得られる確定節で元の確定節を置き換えればよい。（実はもう少し細かい条件がいるのだが省略する。）これを“たたみ込み規則”と呼ぶ。直感的にいえば、一連の手続き呼び出しをそれらを呼び出す一つの手続きで代用することに相当する。

この玉木－佐藤のPrologプログラム変換の最も重要な性質は、最小Herbrandモデルを保存することである。すなわち、最初のPrologプログラムに“展開規則”と“たたみ込み規則”を任意の回数施して得られるPrologプログラムの最小Herbrandモデルは、最初のPrologプログラムの最小Herbrandモデルに等しい[19]。今の例で言えばP_4の最小HerbrandモデルとP_0の最小Herbrandモデルが等しいことになる。

さて話を元に戻すと、先ほど二つの計算帰納法の使い方があった

$$rev(N,L) \wedge ap(N,[X],M) \supset rev(M,[X \mid L])$$

という論理式は、new を使うと

$$new(L,M,N,X) \supset rev(M,[X \mid L])$$

と書けるが、前提部分のアトムnew(L,M,N,X)の代入例でP_0の最小Herbrandモデルに入っているものは、P_4の最小Herbrandモデルに入っているものに等しい。したがって、P_0ではなくP_4を用いた次の帰納法図式が使える。

$$\frac{Q([\,],[X],[\,],X) \qquad Q(L1,M,N,X) \wedge ap(L1,[Y],L) \supset Q(L,[Y \mid M],[Y \mid N],X)}{rev(N,L) \wedge ap(N,[X],M) \supset Q(L,M,N,X)}$$

図 3.4 $rev(N,L) \wedge ap(N,[X],M)$ を前提部分とする計算帰納法

この図式でQ(L,M,N,X)をrev(M,[X | L]) とすれば、次の二つの新しいゴール

G_3 : rev([X],[X])

G_4 : rev(M,[X | L1]) ∧ ap(L1,[Y],L)

⊃ rev([Y | M], [X | L])

が生成されるが、これは先ほどの二つの計算帰納法を同時に使うことに相当する [10] 。（Argus 検証システムでは、この処理が機械的にできるように、帰納法図式の併合は述語new を導入して本体のアトムを１回ずつ展開してたたみ込みができる場合に限っている。）

３．５　計算帰納法とともに使われる推論

我々のArgus 検証システムでも、計算帰納法とともに、“すべてのモデルについて成り立つ性質を証明する”ためのものである一階推論を併用することが必要である。Argus 検証システムでは、特にPrologの実行を拡張した形で一階推論を行うことに特徴がある [9]。以下、Ｇを

（アトムを∧でつないだもの）⊃

∃Y1Y2…Yn（アトムを∧でつないだもの）

の形の論理式として、⊃の左側を前提部分、⊃の右側を帰結部分と呼ぶことにする。（前と同様に一番外側の∀

による束縛は省略してある。）まず、Prologの普通の実行を一般化しよう。

帰結部分のアトムの実行

Aを Gの帰結部分の中の一つのアトムとしよう。いま

B :- B1, B2, …, Bm

をプログラムPの確定節で、∀で束縛されている変数に代入を施すことなくAが頭部Bと（例えば）最汎単一化代入σで単一化できるようなものとする。このときGの中のAをB1∧B2∧…∧Bmで置き換えてσを施した新しいゴールを生成する。（mが0のときはAをtrueで置き換える。新たに導入された変数は∃で束縛されている。）

Gの前提部分がtrueで、∀で束縛されている変数がないときは、Prologの普通の実行になっている。例えば、

G_1 : rev([ ],[ ])

に適用されるとtrueが生成される。∀で束縛されている変数があるときでも、それを具体化しないこと以外はPrologの普通の実行と同じで、例えば、

G_3 : rev([X],[X])

に適用されると、順に

G5 : ∃N (rev([ ],N) ∧ ap(N,[X],[X]))

G_6 : ap([ ],[X],[X])

とゴールが生成されてtrueになる。Gの前提部分がtrue

でないときは、例えば、

G_4 : rev(M,[X | L1]) ∧ ap(L1,[Y],L)

⊃ rev([Y | M], [X | L])

に適用されると、順に

G_7 : rev(M,[X | L1]) ∧ ap(L1,[Y],L)

⊃ ∃N (rev(M,N) ∧ ap(N,[Y],[X | L]))

G_8 : rev(M,[X | L1]) ∧ ap(L1,[Y],L)

⊃ ∃N1 (rev(M,[X | N1]) ∧ ap(N1,[Y],L))

とゴールが生成される。

次に、Prologの“Negation as Failure”規則を一般化しよう。（前提部分のアトムには、∃で束縛されている変数は現れないことに注意されたい。）

<u>前提部分のアトムの実行</u>

AをGの前提部分の中の一つのアトムとしよう。いま

B :-B1, B2, …, Bm

をプログラムPの確定節で、Aが頭部Bと（例えば）最汎単一化代入τで単一化できるようなものとする。このときそのようなすべての確定節について、Gの中のAを

B1∧B2∧…∧Bmで置き換えてτを施した新しいゴールを生成する。（mが0のときはAをtrueで置き換える。新たに導入された変数は∀で束縛されている。）

特に、Gの帰結部分がfalseのときは、Prologの普通の“Negation as failure”規則になっている。

ここで考えたゴールが、Prologの実行ゴールと最も大きく違う点は⊃の両側にアトムが現れることがあることである。したがって、⊃の両側に現れるアトムを関係づける推論規則がないと、例えば、

$$\forall X,Y,Z\ (add(X,Y,Z) \supset add(X,Y,Z))$$

といった簡単なことも示せないことになってしまう。

帰結部分のアトムと前提部分のアトムの相殺

$A_1, A_2, \ldots, A_k$をGの帰結部分の中のアトム、$A_{k+1}, A_{k+2}, \ldots, A_{k+l}$をGの前提部分の中のアトムで∀で、束縛されている変数に代入を施すことなく（例えば）最汎単一化代入σでアトムAに単一化できるようなものとする（k,l>0）。このとき、Gにσを施した後に中のAをすべてtrueで置き換えた新しいゴールと、すべてfalseで置き換えた新しいゴールを生成する。

例えば、ゴール

G_8 : rev(M,[X | L1]) ∧ ap(L1,[Y],L)

⊃ ∃N1 (rev(M,[X | N1]) ∧ ap(N1,[Y],L))

の中のrev(M,[X | L1])とrev(M,[X | N1])に注目すると、

G_9 : ap(L1,[Y],L) ⊃ ap(L1,[Y],L)

とtrueが生成され、さらに、この新しいゴールの中で ap(L1,[Y],L)に注目すると、二つのtrueが生成される。（実は、第2章で述べた相互交配や一般化は、この推論規則に対応している。[9],[14]参照。）結局 rev(L,M) ⊃ rev(M,L) の証明全体は下図のようになる。

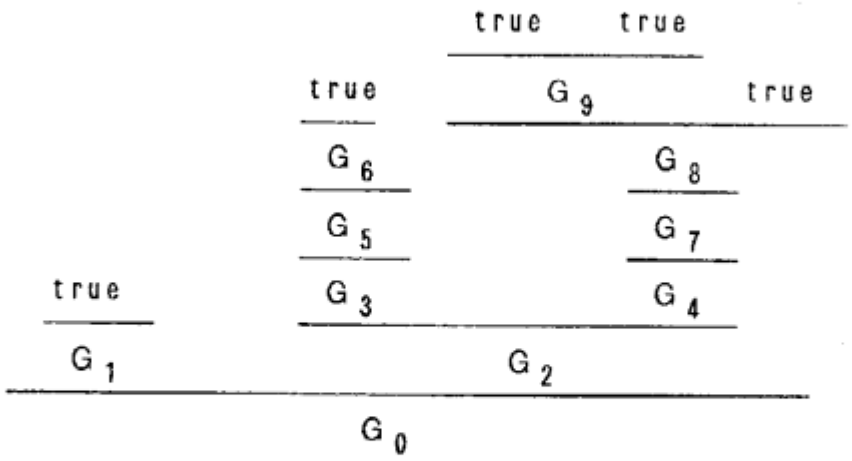

図 3.5 ∀L,M (rev(L,M) ⊃ rev(M,L)) の証明

少し理論的な結果について補足すると、我々の三つの推論規則（拡張実行）には、

（アトムを∧でつないだもの）⊃

∃Y1Y2…Yn（アトムを∧でつないだもの）

という形の論理式のクラスに対して次のような健全性と完全性が成り立つ。（Prologプログラムは“if”に対応する部分しか述べていないが、“only if”に相当する部分を補い、形の違う項は違うものを指すとか、無限項が指すものはないといった“＝”に関する公理をつけ足したものを、そのプログラムの完備化と呼ぶ。プログラムの最小Herbrandモデルは完備化のモデルでもある。）

拡張実行の健全性と完全性

この節で述べた拡張実行規則を用いて証明されるなら、それはPrologプログラムの完備化の論理的帰結である。Prologプログラムの完備化の論理的帰結であるなら、それはこの節で述べた拡張実行規則を用いて証明される。

実は、我々の拡張実行は、“仕様式”と呼ばれるもっと広いクラスの論理式に対して定義されており、（場合分けのための簡単な規則を三つ補うと）やはり仕様式に

ついて健全で完全であることが知られている [14] 。

4. むすび

本稿では、Boyer-Moore 定理証明システムとArgus 検証システムを中心に解説した。（Argus 検証システムの拡張・改良については [5],[6]を御覧頂きたい。）これ以外に、ここ数年、あからさまに帰納法を用いない方法も注目されている。もともとは、等式集合の始代数モデルの上で与えられた（別の）等式が成り立つ（したがって、その等式を加えてもこれまでに証明されなかった変数を含まない等式が新たに成り立つことがない）ことを示すのに、その等式に適当に向きをつけて（等式集合に対応する）項書き換え規則集合に加えた上でKnuth-Bendixの完備化アルゴリズムを用いて無矛盾性を示すもので、HuetとHullot [7]によって考案され、その適用条件も次第に緩められている。（例えば外山[20]。）この方法に直接には対応しないが、Prologについても、櫻井と元田[16],[17],[18]、HsiangとSrivas [8]によって、あからさまに帰納法を用いない方法が提案されている。

以上、プログラム検証への適用を中心に、帰納法を用いる定理証明システムについて解説を試みた。（帰納法を含めて）特定のモデルを問題にした推論は、今後、人工知能の分野でも必要になると思われ、さらに研究が進展することが期待される。

謝辞

本稿で解説したArgus 検証システムは第五世代コンピュータ計画の一環として開発された。研究開発の機会を頂いた新世代コンピュータ技術開発機構(ICOT)の渕一博所長、助言と激励を頂いた古川康一同次長、横井俊夫元同次長（現日本電子化辞書研究所所長）に感謝致します。また、原稿作製に協力して頂いたICOT第一研究室の堀内譲二氏、原稿にいろいろコメントを頂いた三菱電機中央研究所の前地真知氏に感謝致します。

参考文献

[1] Bowen, D.L., Byrd, L., Pereira, F.C.N. and Warren,

D.H.D.: DECsystem-10 Prolog User's Mannual, Department of Artificial Intelligence, University of Edinburgh, 1983.

[2] Boyer,R. and Moore,J S: A Computational Logic Academic Press,1978.

[3] Clark,K.: Predicate Logic as A Computational Formalism,Research Monograph 79/59,TOC,Imperial College, 1979.

[4] Cohen,P.R. and Feigenbaum,E.A. Eds.,: The Handbook of Artificial Intelligence,Vol.3, XII-D,Pitman Books Ltd.,(1982),pp.102-113.

[5] Elkan,R. and McAllester,D.: Automated Inductive Reasoning about Logic Programs, Proc. of 5th International Logic Programming Conference, Seatle, August 1988,pp.876-908.

[6] Fribourg,L.: Equivalence-Preserving Transformation of Inductive Propertiesof Prolog Programs, Proc. of 5th International Logic Programming Conference, Seatle,August 1988, pp.876-908.

[7] Huet,G. and Hullot,J.M.: Proof by Induction in Equational Theories with Constructors, J. of Computer and Systems Science, Vol.25,No.2 (1982) pp.239-266.

[8] Hsiang,J. and Srivas,M.: Automatic Inductive Theorem Proving Using Prolog,Theoretical Computer Science Vol.54 (1987),pp.3-28.

[9] Kanamori,T. and Seki,H.: Verification of Prolog Programs Using An Extension of Execution, Proc. of 3rd International Conference on Logic Programming Conference, London,July 1986,pp.475-489. あるいは preliminary version ICOT Technical Report TR-096, ICOT,Tokyo,December 1984.

[10] Kanamori,T. and Fujita,H.: Formulation of Induction Formulas in Verification of Prolog Programs, Proc. of 8th International Conference on Automated Deduction,Oxford,July 1986, pp.281-299.あるいは preliminary version ICOT Technical Report TR-094, ICOT,Tokyo,December

1984.

[11] 金森　直：Prologプログラムの検証システムについて，昭和59年度第五世代コンピュータプロジェクト成果報告会発表資料，May 1985.

[12] Kanamori,T.,Fujita,H.,Horiuchi,K.,Seki,H. and Maeji,M.: Argus/V : A System for Verification of Prolog Programs, Proc. of Fall Joint Computer Conference 86, Dallas,October 1986, pp.994-999.あるいは preliminary version ICOT Technical Report TR-176, ICOT,Tokyo,May 1986.

[13] 金森　直、藤田　博、世木博久、堀内譲二、前地真知：プログラム検証システム，最新ＡＩ事情（渕一博、中野幸紀監修）、通産政策広報社，1987,pp.128-135.あるいは機械振興　昭和61年度２月号,1986,pp.73-77.

[14] Kanamori,T.: Soundness and Completeness of Extended Execution for ProvingProperties of Prolog Programs, in Programming of Future Generation Computers (Fuchi,K. & Nivat,M. Eds.), North-Holland,1988,pp.259-281. あるいは Proc.

of 1st France-Japan Artificial Intelligence and Computer Science Symposium, Tokyo,October 1986,pp.219-238. あるいは preliminary version ICOT Technical Report TR-175, ICOT,Tokyo,May 1986.

[15] Manna,Z. and Waldinger,R.: The Logical Basis for Computer Programming;Volume 1:Deductive Reasoning, Addisson Wesley,1985.

[16] Sakurai,A. and Motoda,H.: Inductionless-Induction Method in Prolog, 日本ソフトウェア科学会第４回全国大会論文集, 1987年11月,pp.231-234.

[17] 櫻井彰人、元田浩: 数学的帰納法を明示的に使用しない確定節の証明法, Proc. of the Logic Programming Conference '88,1988年 4月,pp.29-38.

[18] 櫻井彰人、元田浩: Prolog節の証明法－その後－日本ソフトウェア科学会第５回全国大会論文集, 1988年 9月,pp.9-16.

[19] Tamaki,H and Sato,T.: Unfold/Fold Transformation of Logic Programs, Proc. of 2nd Inter-

national Logic Programming Conference, Uppsala July 1984, pp. 127-138.

[20] Toyama, Y.: How to Prove Equivalence of Term Rewriting Systems, Proc. of 8th International Conference on Automated Deduction, Oxford, July 1986, pp. 118-127.