ICOT Technical Memorandum: TM-0246

TM-0246

証明とコンピュータ
—数学する人々の"よき助手"としての自動検証系の試み—

廣瀬　健(早稲田大学)
横田一正

January, 1987

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

談話室

証明とコンピュータ

数学する人々の“よき助手”としての自動検証系の試み

廣瀬 健・横田一正

■“定理”の自動証明系について■

A. Newell, J. C. Shaw, H. A. Simon らが命題論理の定理を証明するプログラムについて発表したのが 1956 年．発表はされなかったようだが，M. Davis が数論における定理証明のプログラムを 1954 年に書いたといわれているから，定理の証明を自動的に行わせる試みは，すでに 30 年以上の歴史を持っていることになる．

1984 年に American Mathematical Society から《Contemporary Mathematics》の vol. 29 として刊行された

《Automated Theorem Proving》

にも，

〈After 25 Years〉

という副題がつけられている[1]．いわゆる人工知能（AI）の試みの最も古いものの１つといえよう．

日本でも，1950 年代の後半から 1960 年代前半にかけて，すでに高須達氏の試みや，島内剛一氏や西村敏男氏ら数学基礎論グループによる試みなどがある．島内氏らの自動証明系の方法の骨子は，現在のシステムでも，ほとんどそのままに使用されている．

さて，自動証明系（automated theorem prover）とは，“命題が与えられたとき，その命題が正しいものであれば，（いつかは必ず）その証明を作りあげるシステム”のことである．

このようなシステムの存在を原理的に保証するのは「Gödel の完全性定理」であるが，完全性定理の証明は，正しい命題に対してその証明を作りだすアルゴリズムを与えるものではないから，自動証明系を作成するには，正しい命題に対して証明を生成する有効なアルゴリズムを工夫しなくてはならない．

一般に，与えられた命題の真偽を決定することは不可能である（ことが知られている）から自動証明系は，与えられた命題が正しいか否かをチェックはできない．

与えられた命題が正しくても，適当な時間内で，証明を作成するアルゴリズムを工夫することはきわめて困難である．

事実，最近の Cook, Haken, Urquhart らの研究によれば，命題論理の範囲で形式化できる対象でも，現在知られているほとんどの体系について，“手に負えない（intractable）証明”，つまり，（計算量の理論でいうところの）多項式時間内では処理できない証明が存在することも知られている．

つまり，有用で一般的な自動証明系の作成には，本質的な困難さがともなうように思われる．

それにも関わらず，自動証明系の作成は魅力的な存在である．

１つには，上述の悲観的な見通しの根拠が，きわめて一般的な対象についてのものであること．さらには，定理の自動証明系の研究は，人工知能研究の大きな柱となるものだからである．

■“証明”の自動検証系について■

証明の自動検証系（automated proof checker）とは，“与えられた証明，すなわち，公理から定

理までの推論の有限列が正しいものであるか否かを検証するシステム”のことである.

この実現は容易そうにみえる.1つ1つの推論をチェックしさえすればよいではないか?

与えられた証明が,証明論における形式的証明,すなわち,形式化された推論の列になっている場合ならば,原理的な困難さは,たしかに何もない.

しかしながら,論文や教科書で述べられているような非形式的証明をチェックするとなれば,これは容易でない.

非形式的証明では,しばしば“証明の省略”が行われている.すなわち,証明の行間を埋めなければ厳密な意味での証明にはならないのが普通である.

専門的な論文の証明を非専門家が読めないのは,証明の行間が広いためといってよいであろう.

この行間を自動的に埋める仕事が容易であろうはずはない.極端な話,与えられた“証明”が前提と結論のみで,その間が全部省略されていたら,これは定理そのものであり,このチェックには証明系と同様の働きが必要となる.

これは,あり得ない話ではない.あまり明らかでない命題について,「明らか」としか証明が書かれていない場合も多いのだから…….

証明の行間が広いと,証明の方針,そこで用いられる数学的知識なども必要になってくる.すなわち,この場合,証明検証系はさまざまな数学的理論についての知識ベースをもっていなくてはならず,場合によっては,その知識ベースにもとづいて(演繹的推論だけでなく)帰納的推論をも行わなくてはならない.

これらの技法は計算機科学においても未だ成熟していない重要な分野に属する.

■ 実用的システムへの志向 ■

証明系や証明検証系における方法は,原理的には25年前の方法と同じで,他に,特につけ加えられた著しい方法はない.

一方において,ここ25年の間の計算環境の向上は著しいものがある.ミリ秒(1/1,000秒)単位だった演算速度はナノ秒(1/1,000,000,000秒)単位で測るようになり,キロ単位だった記憶容量も,メガ(1,000,000)単位,ギガ(1,000,000,000)単位で語られるようになっている.マン・マシン・インタフェース——入出力の状況も飛躍的に改良された.

原理的に本質的な進歩がないとしても,計算機科学および計算環境の進展の状況からして,実用的なシステムへの思索と志向があるのは当然のことであろう.

事実,LCF(当初スタンフォード大学で,現在はエディンバラ大学,INRIAで研究・開発されているシステム)とか,EKL(スタンフォード大学で作られた項書き換え方式を基にしたシステム),あるいはMizar(ワルシャワ大学で開発され,アメリカの諸大学でも試用されているシステム)など,現在,さまざまな試みが行われている.

現在の証明系あるいは証明検証系のねらいは,次の(i),(ii),(iii)に要約されよう:

(i) 数学者に対し“良き助手”となり,かつ“仕事をしやすい環境”を提供するシステム.

(ii) 教育の場における有用な補助手段を提供するシステム.

(iii) 知識情報処理システムへの応用.

(i)については,たとえば,式の変形や計算のチェック,個々については簡単だが場合別けの多い証明のチェック,援用すべき定理の探索や条件のチェック,証明記述や論文作成の支援などが挙げられよう.そして,ここでは,会話型での仕事の進行が自然なものとなろう.

(ii)については,多勢の学生の行った証明のチェックとか,証明を行う場合の相談相手を勤めるシステム,あるいはさまざまな証明例を挙げることなどが考えられる.これらについては,会話型の進行とともに,バッチ型の処理も必要であろう.

(iii) は，人工知能研究の知見の適用に他ならない．ここで述べるべき事柄は多いが，本稿では立入らないことにしよう．

以上の目的意識の下では，(i), (ii), (iii) いずれの場合でも，証明系と証明検証系の区別は稀薄になる．双方の機能が複合的に必要となるからである．

▌CAP プロジェクト ▐

現在，開発が進行中の証明検証系について，以下では，筆者の関わっている CAP プロジェクトについて述べたい．CAP (Computer Aided Proof) プロジェクトは，ICOT (Institute for new generation COmputer Technology；新世代コンピュータ技術開発機構) におけるプロジェクトである．

ICOT における研究の主目標の1つは知識情報処理であり，人間－機械系である．

上述のような証明系，証明検証系の研究は，知識情報処理のメカニズムの有力な具体例を与えるものであり，人工知能の典型的な例と手がかりを与えるものでもあるから，ICOT のプロジェクトの1つとして，研究・開発が進められている．

CAP プロジェクトでは，

(1) 線型代数 (LA；Linear Algebra)
(2) 記号算術 (SA；Symbolic Arithmetic)
(3) 綜合微分幾何 (SDG；Synthetic Differential Geometry)

の3つの分野が選ばれているが，以下では (1) の CAP-LA システムについて，その概略を紹介することにしたい．

なお，このプロジェクトでは，多くの ICOT メンバーの他，筧捷彦氏（早大），佐藤雅彦氏（東北大），永田守男氏（慶大），萩谷昌己氏，林晋氏（京大），桜井貴文氏（都立大），後藤滋樹氏（通研），玉井哲雄氏（三菱綜研），山田眞市氏（日本ユニバック）らの協力を得て進められているものである．

▌CAP-LA システム ▐

証明記述言語 PDL

証明の検証，すなわち，証明が正しいかどうかをコンピュータにチェックさせるためには，まずその証明の記述がコンピュータに理解できるものでなくてはならない．コンピュータによる自然言語理解の研究はかなり進んできてはいるが，現段階では，証明記述のために，自然言語でなく構文規則のはっきりした形式言語が必要となる．

証明は人が読むものであるから，形式言語であっても人にとって読みやすいものでなくてはならない．さいわい，数学では数式や論理式の占める割合が多く，使用される自然言語を比較的少なくすることができる．したがって設計する形式言語は，限られた自然言語を使用し，数式や論理式が表現でき，明解な構文規則をもつ，なるべく自然に読めるものにしたい．

コンピュータでは入力することのできる文字の種類がひじょうに限られている．通常の端末では，英数字，かな，そして少々の特殊記号だけである．たしかに特殊な端末を使用すれば，字種は増えるが，それではシステムを作ったとしても使用環境が限られてしまう．そこで証明の記述は，通常の端末で簡単に入力できる範囲の字種だけでできるようにする．出力については，証明を通常の読みやすいものに変えるために清書機能を付けることを考えたい．

以上のような方針の下に，CAP-LA システムでは，次のような形式的言語 PDL (Proof Description Language) が用意されている．現在，新しい機能を盛り込んだ第2版を設計中であるが，ここでは使用可能な第1版について説明することにする．ここで PDL を詳細に説明する余裕はないが，以下の証明例などから，そのおおよそはお分りいただけると思う．

使用する記号としては，定数記号，数変記号，関数記号，述語記号などがあるが，これらは教科書などでは文字の種類で区別することが多い．し

かし PDL では文字の種類が限られていることもあって，すべて英数字（列）を使用し，構文上の使われ方によってそれらを識別する．定数記号や関数記号，述語記号は，利用者によって自由に定義することが可能であるが，あらかじめ用意されている記号もいくつかある．たとえば自然数上の加算＋，等号＝などである．$\sum$, $\prod$ はぜひ用意しておきたいものであるが，PDL では sum, product（あるいは prod）とそれぞれ記し，現在は特殊な扱いにしている．添字は sum や product の引数として記述する．

PDL での論理記号は次のようなものを用いている：

通常の記法	PDL の記法	
¬	\	(not)
∧	&	(and)
∨	|	(or)
∀	all	
∃	some	

対象の“型”を考えることは重要である．計算機の世界の言葉でいえば“データ型”であるが，LA では自然数，実数，行列，ベクトルなどさまざまな型を用いる．証明の中でもっともあいまいになりがちなのがこの型で，ある変数の型が何かは，文脈から判断することが多い．それらを区別するために何種類かの型を用意しておき，すべての変数はどの型に属しているかを明確に宣言するようにした．たとえば加算＋は自然数にも行列にも使用されるが，内容的には異なったものである．これを区別するのが使用されている型というわけである．たとえば，

$$A : \mathrm{nat}$$

とすれば，これは“Aは自然数である”ことの宣言である．ある型の対象の部分集合を“sort”としてパラメータ付で定義することもできる．たとえば

```
sort seg⟨n : nat⟩  (x : nat)
  as 0 ≦ x ≦ n
end_sort
```

で“0からnまでの自然数の集合”を表わすsort, seg⟨n⟩を定義することができる．証明中に現れるすべての定数，変数は必ずこのような型（あるいは sort）のどれかに属している．

推論規則と証明記述の形式

対象の記述の次には，証明の記述形式が問題になる．証明は推論規則に沿って論理的に記述しなければならない．したがって PDL は推論規則を反映した論理的な筋道を表現できなければならない．そこで証明を記述するための形式，いわば鋳型（テンプレート）を推論規則に対応させて準備する．

数学における論理の形式化としてはさまざまな形式的体系が存在する．PDL では，Gentzen の NK（自然演繹体系）が採用されている．これは NK が通常の証明記述を表現するのにもっとも自然であると考えられたからである．NK はたとえば以下のような推論規則からなっている：

導入規則

$$\frac{\begin{array}{c}\vdots\\ A\end{array}}{A \vee B} \qquad \frac{\begin{array}{cc}\vdots & \vdots\\ A & B\end{array}}{A \wedge B} \qquad \frac{\begin{array}{c}[a:t]\\ \vdots\\ A(a)\end{array}}{\forall x:t.\, A(x)}$$

除去規則

$$\frac{\begin{array}{ccc}[A] & [B] & \\ \vdots & \vdots & \vdots\\ A \vee B & C & C\end{array}}{C} \qquad \frac{\begin{array}{c}\vdots\\ A \wedge B\end{array}}{A} \qquad \frac{\begin{array}{cc}\vdots & \vdots\\ \forall x:t.\, A(x) & a:t\end{array}}{A(a)}$$

この他の NK の規則や，その内容については，たとえば [2] を参照されたい．

上に挙げた6つの推論規則に対応する PDL での証明の形式は次のようである．

∨-導入規則

```
  ⋮
  A
hence A|B
```

∧-導入規則

```
        ⋮
        A
        ⋮
        B
        hence A & B
```

∀-導入規則

```
all x : t. A(x)
  since
    let a : t be arbitrary
    ⋮
    A(a)
  end_since
```

∨-除去規則

```
  ⋮
A|B
hence C
  since divede and conquer A|B
    case A
      ⋮
      C
    case B
      ⋮
      C
  end_since
```

∧-除去規則

```
        ⋮
      A & B
      hence A
```

∀-除去規則

```
        ⋮
    all x : t. A(x)
        ⋮
    a : t
    hence A(a)
```

この証明の形式を元の推論規則と比較していただければ，その意味と対応は明らかであろう．

数学の証明を記述するためには NK の推論規則だけでなくさらにいくつかの推論規則を用いたい．たとえば数学的帰納法で，その推論規則は

$$\dfrac{\begin{array}{cc} & [n:\mathrm{nat}]\ \ [A(n)] \\ \vdots & \quad\vdots \\ A(0) & \quad A(n+1) \end{array}}{\forall\, m:\mathrm{nat}\ A(m)}$$

であり，これに対する証明の形式は

```
all m : nat. A(m)
  since induction on m
    base
      ⋮
      A(0)
    step
      let n : nat be arbitrary
      [ind-hyp-is A(n)]
      ⋮
      A(n+1)
  end_since
```

である．

これらの例でわかるとおり，PDL では結論をまず書いて，その証明をその後の since から end_since の間に書くようになっている．

これまで『行列と行列式』（古屋茂著，培風館）中の証明を約 90% ほど記述し，第2版への経験を蓄積しているところである．

証明例

PDL での証明の記述を，1つ2つお目にかけよう．

例1は，“行列を2度転置すると，元の行列に等しい”ことの証明であり，例2は，転置行列の行列式についての定理である．例2は，教科書にある証明と PDL による記述を対比させてある．

例1

```
theorem trans_trans:
  all m, n: pos. (all A: matrix(m, n).
    trans(trans(A)) = A)
proof
  let m1, n1: pos be arbitrary
  all A: matrix(m1, n1). trans(trans(A)) = A
  since
    let X: matrix(m1, n1) be arbitrary
    col_size(trans(trans(X)))
      = row_size(trans(X))
      = col_size(X)
      = m1
    row_size(trans(trans(X)))
      = col_size(trans(X))
      = row_size(X)
      = n1
    hence col_size(trans(trans(X))) = m1;
    row_size(trans(trans(X))) = n1;
    all i: seg⟨m1⟩, j: seg⟨n1⟩.
      trans(trans(X))[i, j] = X[i, j]
    since
      let i: seg⟨m1⟩, j: seg⟨n1⟩ be arbitrary
      trans(trans(X))[i, j]
        = trans(X)[j, i]
        = X[i, j]
      hence trans(trans(X))[i, j] = X[i, j];
    end_since;
    hence trans(trans(X)) = X;
  end_since;
end_proof;
end_theorem
```

次に, S. Lang の《Linear Algebra》での定理と証明を引用する:

Theorem 6.

Let A be a square matrix. Then

$$\mathrm{Det}(A) = \mathrm{Det}({}^tA).$$

Proof. In Theorem 5, we had

$$(*)\quad \mathrm{Det}(A) = \sum_{\sigma} \varepsilon(\sigma) a_{\sigma(1),1} \cdots a_{\sigma(n),n}.$$

Let σ be a permutation of $\{1, \cdots, n\}$.

If

$$\sigma(j) = k,$$

then

$$\sigma^{-1}(k) = j.$$

We can therefore write

$$a_{\sigma(j),j} = a_{k,\sigma^{-1}(k)}.$$

In a product

$$a_{\sigma(1),1} \cdots a_{\sigma(n),n}$$

each integer k from 1 to n occurs precisely once among the integers

$$\sigma(1), \cdots, \sigma(n).$$

Hence this product can be written

$$a_{1,\sigma^{-1}(1)} \cdots a_{n,\sigma^{-1}(n)}$$

and our sum $(*)$ is equal to

$$\sum_{\sigma} \varepsilon(\sigma^{-1}) a_{1,\sigma^{-1}(1)} \cdots a_{n,\sigma^{-1}(n)},$$

because

$$\varepsilon(\sigma) = \varepsilon(\sigma^{-1}).$$

In this sum, each term corresponds to a permutation σ. However, as σ ranges over all permutations, so does σ^{-1} bacause a permutation determines its inverse uniquely. Hence our sum is equal to

$$(**)\quad \sum_{\sigma} \varepsilon(\sigma) a_{1,\sigma(1)} \cdots a_{n,\sigma(n)}.$$

The sum $(**)$ is precisely the sum giving the expanded form of the determinant of the transpose of A.

これを PDL で書くと以下のようになる:

概念

```
theory determinant

det(A : square)
  :=sum P : perm⟨col_size(A)⟩.
    sgn(P)
      * prod I : seg⟨col_size(A)⟩. A[P[I], I]

theorem determinant_of_transpose
  all A : square.    det(A) = det(trans(A))

proof
  let a : square be arbitrary
  n :=col_size(a)
  then n = col_size(trans(a))
  det(a)
  = sum P : perm⟨n⟩.
    sgn(P) * prod I : seg⟨n⟩. a[P[I], I]
                                  by definition
  = sum P : perm⟨n⟩.
    sgn(inv(P))
    * prod I : seg⟨n⟩. a[inv(P)[I], I]
  = sum P : perm⟨n⟩.
    sgn(P) * prod I : seg⟨n⟩. trans(a)[P[I], I]
    since
      let p : perm⟨n⟩ be arbitrary
      prod I : seg⟨n⟩. a[inv(p)[I], I]
      = prod I : seg⟨n⟩.
          a[inv(p)[p[I]], p[I]]
      = prod I : seg⟨n⟩. trans(a)[p[I], I]
        since
          let i : seg⟨n⟩ be arbitrary
          a[inv(p)[p[i]], p[i]]
          = a[i, p[i]]
          = trans(a)[p[i], i]]
        end_since
      sgn(inv(p)) = sgn(p)
    end_since
    = det(trans(a))              by definition
end_proof
end_theorem

end_theory
```

両者を見較べていただければ，現在の PDL がどのようなものかおわかりいただけると思う．文字の種類の制限もさることながら，$\sum$ を sum と書かなくてはならないことや，添字が引数として書かれていることなど，平板になって分りにくい．

それで，出力として清書機能を用意し，2 次元化や日本語化表示（入力された英語風の言葉を内部で自動的に変換する）を可能にする機能を追加することを考えている．現在，試験的に用いている日本語化表示による証明は次のようになる：

```
理論 determinant

det(A : square)
  := sum P : perm⟨col_size(A)⟩.
    sgn(P)
      * prod I : seg⟨col_size(A)⟩. A[P[I], I]

定理 determinant_of_transpose
  すべての正方行列 A にたいして
    det(A) = det(trans(A))

証明
  任意に正方行列 a を固定する
  n := col_size(a)
  すると
  n = col_size(trans(a))
  det(a)
  = sum P : perm⟨n⟩.
    sgn(P) * prod I : seg⟨n⟩. a[P[I], I]
                                  定義より
  = sum P : perm⟨n⟩.
```

sgn(inv(P))
* prod I : seg⟨n⟩. a[inv(P)[I], I]
= sum P : perm⟨n⟩.
sgn(P) * prod I : seg⟨n⟩. trans(a)[P[I], I]
なぜならば
任意に perm⟨n⟩ p を固定する
prod I : seg⟨n⟩. a[inv(p)[I], I]
= prod I : seg⟨n⟩.
a[inv(p)[p[I]], p[I]]
= prod I : seg⟨n⟩. trans(a)[p[I], I]
なぜならば
任意に seg⟨n⟩ i を固定する
a[inv(p)[p[i]], p[i]]
= a[i, p[i]]
= trans(a)[p[i], i]
sgn(inv(p)) = sgn(p)
= det(trans(a)) 定義より
Q. E. D

まだ，たどたどしいが，一部を日本語に簡単に自動変換するだけで，ずいぶんわかりやすくなる．さらに数式の2次元化を追加すれば，かなり見やすくなるのではないかと考えている．

証明の検証

次に PDL で記述した証明の検証について考えよう．証明の内容は大きく2つの部分に分類できる．すなわち，上記の推論規則を適用して論理的な推論を行っている部分と，数式を変形して等号で次々に結んでゆく部分である．この2つの部分の検証をどのように調和させるかが重要である．

現在稼動しているシステムは，まだ主として厳密な証明のみを対象としているが，論理的な推論部分は，結論からそれを導くのに必要な前提を捜す後向き推論によっている．

等号の部分，たとえば上の証明の中では，

det(a)
= sum P : perm⟨n⟩.
sgn(P) * prod I : seg⟨n⟩. a[P[I], I]
= sum P : perm⟨n⟩.
sgn(inv(P))
* prod I : seg⟨n⟩. a[inv(P)[I], I]
= sum P : perm⟨n⟩.
sgn(P)
* prod I : seg⟨n⟩. trans(a)[P[I], I]

などがその例であるが，この等式変形の部分は上の推論規則のみでなく，パタン・マッチングと項書き換えを用いる．最初の等号は det の定義の展開で項書き換えを行い，次の等号は P を inv(P) で置換したものでパタン・マッチングを行う．3番目の等式は複雑であるから，以下の since から end_since までその証明を行っている．項書き換えの能力は $A \rightarrow B$ (A を B に書き換える）といった書き換え規則をどれだけもっているかに依存している．この書き換え規則は，自然数や実数などの各型に対して与えた公理から自動生成されるものと，証明中の仮定などから動的に生成，消滅するものがある．システムは証明中の環境を判断しながら可能性のある書き換え規則を取り出しては数式の変形を試み，等式の検証を行う．公理は必要に応じてシステムに与えることができるようになっているが，いくつかの基本的な公理は現在組込みになっている．

このシステムの特徴を1つ挙げると，複雑な証明ステップをスキップすることができるということである．つまり定理の証明をさぼったり，証明中の行間を空けたままにしておくことができる．命題の後に "proved" と書きさえすればその命題は無条件に正しいものとして処理されるのである．したがって証明のあらましを "proved" つきの証明として記述しておき，その全体が検証された後でその細部を記述する（すなわち "proved" を外す）ことができる．いわば証明の段階的詳細化が可能なのである．

■ CAP-LA の使用環境 ■

PDL で記述された証明を検証するためにはそ

の環境が整備されていなければならない．まず証明をコンピュータに入力したり，修正するためには証明エディタが必要となる．そして証明中の誤まりを除去するためには，エディタで検出できる構文エラーや，検証できない行間を表示するだけではなく，検証情報をチェックすることも必要である．また1つの証明は，他の多くの定理や理論を前提にしている．前述のように，このために検証ずみの定理や証明を蓄積し，必要に応じて取り出すための知識ベースも必要である．

証明エディタは PDL についての構文知識をもったエディタで，構文上のエラー・チェックのほかに構文のガイダンスも行う．したがって，PDL にそれほど詳しくない人も利用可能である．さらにエディタの使用中に他の定理を参照したくなったら，それを自在に引用することもできる．上で述べた清書機能もこの証明エディタに付加されているもので，自然言語風の証明で内容をチェックすることもできる．この証明エディタは，このシステムを使う人との中心的なインタフェースであるから，設計でもっとも気を使うところである．

誤まりを見つける作業で，簡単なものは証明中の位置とエラーの内容（構文エラー，推論不能など）の指定ですぐわかるが，大きな範囲にわたる複雑に錯綜した誤まりはなかなか発見できないものである．そこで証明の検証中にどのような仮定がいつ使用されているかなどを表示し，仮定の間違いや型のミスの修正を助ける機能もついている．

数学の知識をどのようにコンピュータに格納するかを考えると，数学者のもっている知識の量と質に驚かざるをえない．現在は，証明そのもの，証明から派生した推論規則，そしてそれらの間の依存関係をシステムにもたせている．もし証明が修正され，まだ検証が終了していない状態であれば，それに依存している定理も自動的に未検証の状態としなければならない．このような一貫性保持の機構も必要であるが，大量の知識を格納するための本格的な知識ベースについては，まだ準備を進めている段階である．

このような自動検証系の環境は，検証の能力そのものとは直接関係しないが，実質上きわめて重要である．今後も，このシステムを使用した経験を基に徐々にその拡充を図りたいと考えている．

さて，システムが現在どのような環境で動いているかを簡単に説明しよう．稼動しているコンピュータは，ICOT が第5世代コンピュータのためのツールとして開発した逐次型推論マシン PSI である．この PSI は論理型プログラミング言語 ESP が高速に動くように設計されたワークステーションである．マン・マシン・インタフェースとしては図1のように多くのウインドウが操作できるようになっている．

それでは実際に証明を検証している様子を紹介しよう．図2は例1と同じ内容で，“行列を2度転置すると元の行列に戻る”ことの証明である．これは証明エディタのウインドウで，もし構文中にエラーがあれば，たとえば図3のようなエラーが表示される．

```
**** here ****
```

の下を見ると，theorem が theoremn になっているのが分る．

また，図2の内容を日本語化表示すれば図4のようなウインドウが現れる．証明の検証中は図5のようなウインドウが表示され，現在証明のどの部分が検証されており，そのためにはどのような仮定が使用されているかが表示される．検証が終了すると，指示にもとづきその定理や理論は知識ベースに保管され，定理や理論間の関係もたくわえられる．

■ 今後にむけて ■

記述する証明をいかに省略できるかは，現在かかえている大きな問題の1つである．これには，証明の行間を広くするということと，類似の証明の記述を省略することの2つが考えられる．行間を広くするということは，明らかと考えられる部

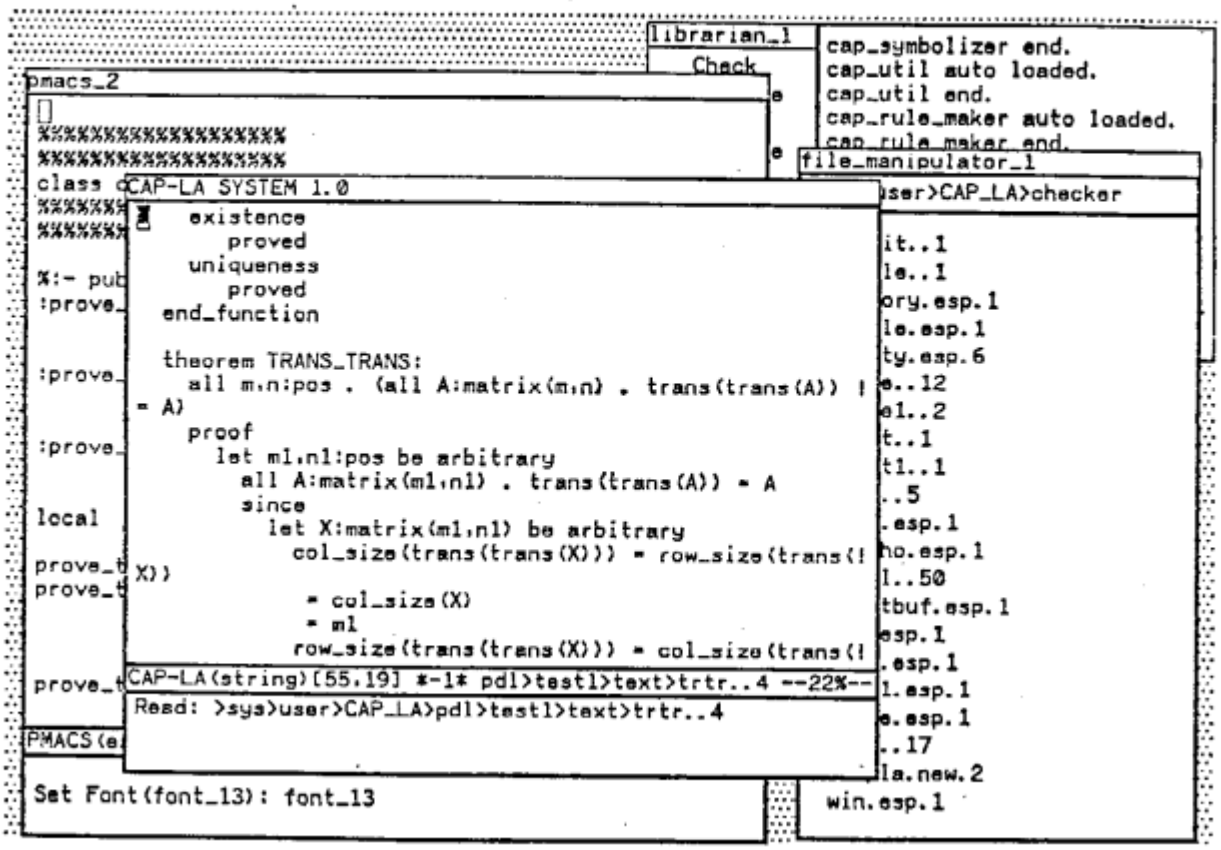

図 1 PSI のマルチ・ウインドウ

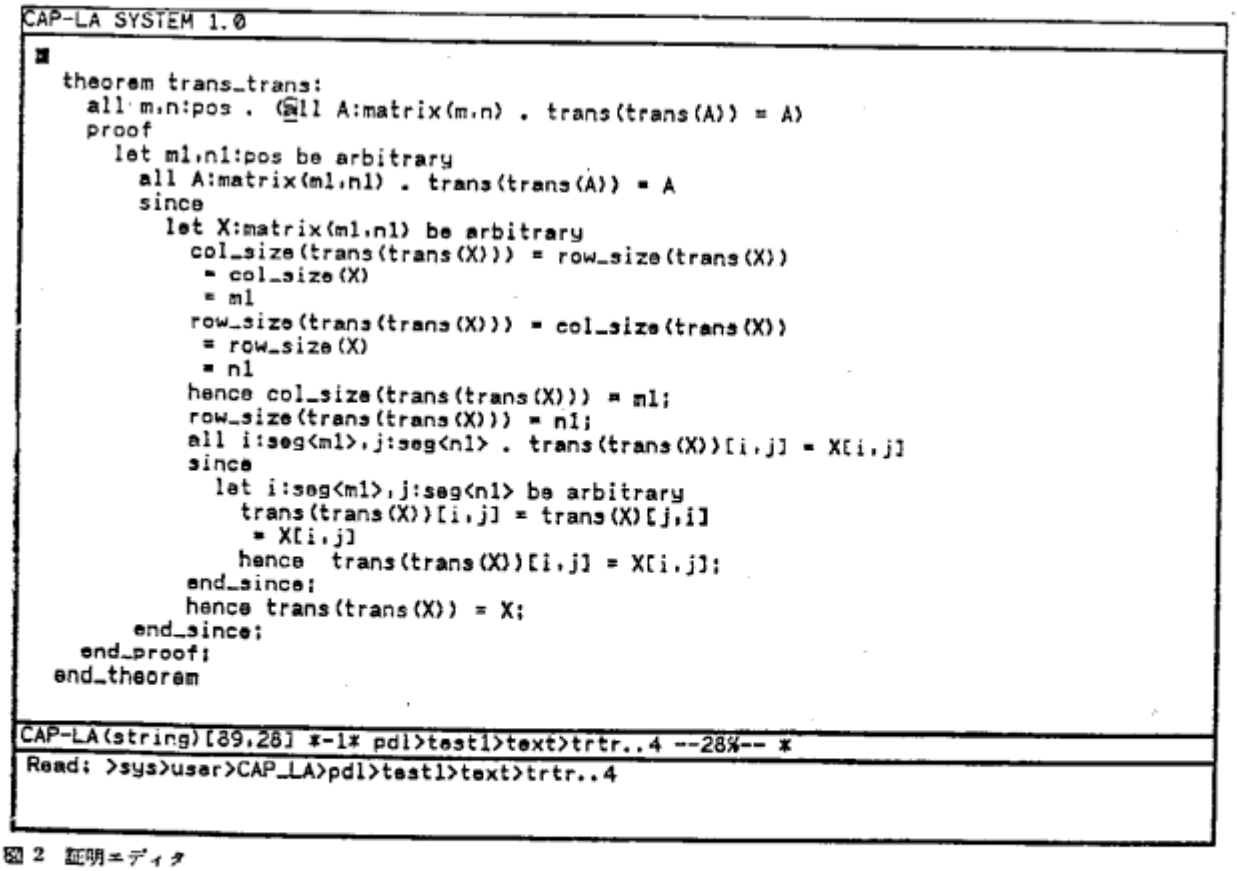

図 2 証明エディタ

```
CAP-LA SYSTEM 1.0
      &
      row_size(trans(X)) = col_size(X)
      &
      (all i:seg<trans(X)>,j:seg<trans(X)> . trans(X)[i,j] = X[j,i])
    existence
       proved
    uniqueness
       proved
  end_function

  theoremn trans_trans:
    all m,n:pos . (all A:matrix(m,n) . trans(trans(A)) = A)
    proof
      let m1,n1:pos be arbitrary
------------------------------------------------------------------------
      ############  Error Message List for trtr ############

Syntax Error ****

theory Example : sort seg < m : pos > ( x : pos ) as 1 =< x & x =< m end_sort function tr!
ans ( X : matrix ) : matrix attain col_size ( trans ( X ) ) = row_size ( X ) & row_size (!
 trans ( X ) ) = col_size ( X ) & ( all i : seg < trans ( X ) > , j : seg < trans ( X ) >!
 . trans ( X ) [ i , j ] = X [ j , i ] ) existence proved uniqueness proved end_function
**** here ****
theoremn trans_trans : all m , n : pos . ( all A : matrix ( m , n ) . trans ( trans ( A )!
 ) = A ) proof let m1 , n1 : pos be arbitrary all A : matrix ( m1 , n1 ) . trans ( trans !
( A ) ) = A since let X : matrix ( m1 , n1 ) be arbitrary col_size ( trans ( trans ( X ) !
) ) = row_size ( trans ( X ) ) = col_size ( X ) = m1 row_size ( trans ( trans ( X ) ) ) =!
 col_size ( trans ( X ) ) = row_size ( X ) = n1 hence col_size ( trans ( trans ( X ) ) ) !
------------------------------------------------------------------------
CAP-LA(string)[39,14] error_trtr    --Top-- *
```

図 3　構文エラーの表示

```
CAP-LA SYSTEM 1.0
------------------------------------------------------------------------
    定理 trans_trans
      すべての 正の整数 m, n に対して
        すべての m n 行列 A に対して
          trans(trans(A)) = A
      証明
        任意に 正の整数 m1, n1 を 固定する
          すべての m1 n1 行列 A に対して
            trans(trans(A)) = A
          なぜならば
            任意に m1 n1 行列 X を 固定する
              col_size(trans(trans(X))) = row_size(trans(X))
                = col_size(X)
                = m1
              row_size(trans(trans(X))) = col_size(trans(X))
                = row_size(X)
                = n1
              ゆえに col_size(trans(trans(X))) = m1
              row_size(trans(trans(X))) = n1
              すべての i seg<m1>, j seg<n1> に対して
                trans(trans(X))[i, j] = X[i, j]
              なぜならば
                任意に i seg<m1>, j seg<n1> を 固定する
                  trans(trans(X))[i, j] = trans(X)[j, i]
                    = X[i, j]
                  従って trans(trans(X))[i, j] = X[i, j]
                ゆえに trans(trans(X)) = X
                                                    Q. E. D.
------------------------------------------------------------------------
CAP-LA(string)[72,28] $$n1_trtr    --27%-- *
```

図 4　証明の日本語化表示

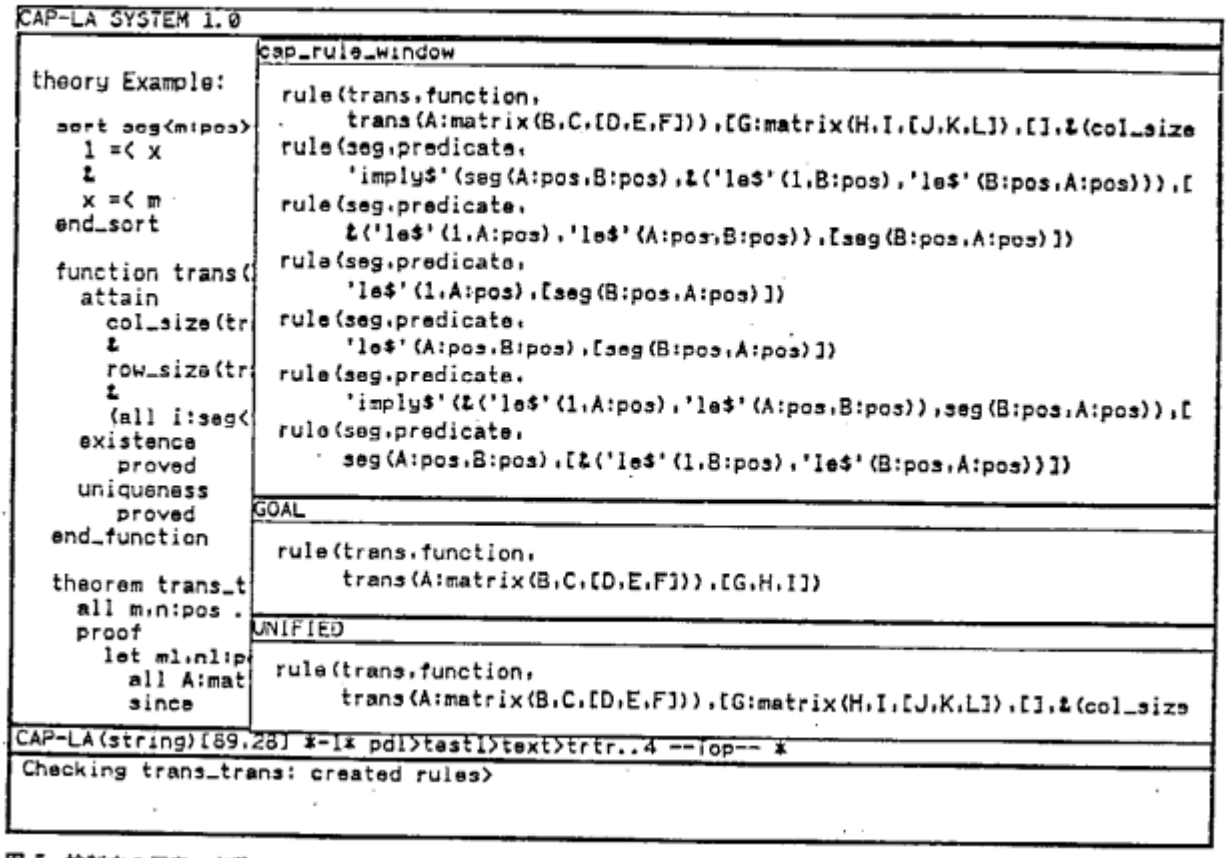

図 5 検証中の仮定の表示

分の省略であるが，これは実行の効率とも大きく関係する．もちろん，行間を広くすればそれだけ計算機における探索空間が大きくなり効率は悪くなる．類似の証明を省略することについては，どこまでを類似と見るかが難しい．

また，現在のシステムは，証明を与えてそれを検証するものであるが，場合によっては，証明を少し追加するたびにチェックしてくれる方が都合がよい．また，数式を入力すれば，システムがいろいろな規準での変形をし，システムの使用者がその中から必要な選択をして等号でそれらを結ぶ，といった作業が進められれば，証明するという作業はずっと楽になる．現在，すでにこのようなシステムの具体的な検討を始めており，まず独立にこのシステムを作成し，次に現在のシステムとの統合化を行いたいと思っている．

改良点は山ほどある．実行速度の改善を始めとして，ユーザ・インタフェースの改良，より高度な知識をもつエディタの開発，清書能力の大幅機能アップ，などである．現在のシステムは最初のバージョンということもあり，かなり機能を限定したものになっている．今後少しずつ機能を増加してゆきたいと考えている．

最後に，コンピュータの定理証明に興味をもたれた読者のために，いくつか教科書を紹介しておこう．参考文献の [3] は論理プログラミング言語への入門書としてよく書かれている．[4] と [5] は定理証明について定評のある教科書である．また最初に述べた [1] には 25 年間にたわる定理証明の歴史と最近のシステムが紹介されている．最後におもしろい話題として，[6] に自動証明系のための問題集(75 問) が掲載されていることをお知らせしたい．これは命題論理や述語論理など，自動証明系の歴史の中で話題になった問題を中心に集めたもので，各問題には点数が割り振られて

いて，自分の自動証明系の能力を採点できるようになっている．

●参考文献

[1] "Automated Theorem Proving: After 25 Years", Comtemporary Mathematics, vol. 29, American Mathematical Society, 1984.

[2] 島内剛一，『数学の基礎』(日本評論社)，1971.

[3] 長尾真，淵一博，『論理と意味』，岩波講座情報科学 7，(岩波書店)，1983.

[4] C.-L. Chang and R. C.-T. Lee, "Symbolic Logic and Mechanical Theorem Proving", Academic Press, 1973.

[5] D. W. Loveland, "Automated Theorem Proving: a Logical Basis", North-Holland, 1978.

[6] F. J. Pelletier, "Seventy-Five problems for Testing Automatic Theorem Prover", Journal of Automated Reasoning, vol. 2, pp 191-216, 1986.

(ひろせけん／早稲田大学)

(よこたかずまさ／新世代コンピュータ技術開発機構)