ICOT Technical Memorandum: TM-0049

TM-0049

Prologによる
Boyer-Moore型定理証明系の実現

藤田　博　　堀内謙二
(三菱電機)

March, 1984

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

Prolog による
Boyer-Moore 型定理証明系の実現*

三菱電機株式会社　中央研究所

藤田　博,　　堀内謙二

概要

プログラムの虫取りや正しさの検証は、依然としてほとんど人の手に委ねられている。正しさの検証とは、プログラムが与えられた仕様を満たしているかどうかを確かめることであるが、それには一般に多大な手間と時間を要する。そこで、この作業を計算機によってできるだけ自動化しようとする研究が進められている。特に、検証の過程ではプログラムの満たすべき性質を、ある定理として証明することが必要となる。

本論文では、 Prolog によって実現した Boyer-Moore 型定理証明システム (以下 BMTP と略称) について述べる。 BMTP は元来、関数型の Lisp プログラムの満たすべき性質を定理として証明するものであるが、論理型の Prolog プログラムを対象とした検証システムを実現する際にも有益な technique を提供するものと考える。

Keywords

Program Verification, Program Debugging, Program Specification, Theorem Prover, Lisp, Functional Program, Prolog, Logic Program

---

* 本論文は、三菱電機技報 (Vol. 58, No. 6) に掲載予定の解説記事 "プログラム自動検証システム" に加筆修正したものである。

1. まえがき

プログラムの虫取りや正しさの検証は、依然としてほとんど人の手に委ねられている。正しさの検証とは、プログラムが与えられた仕様を満たしているかどうかを確かめることであるが、それには一般に多大な手間と時間を要する。そこで、この作業を計算機によってできるだけ自動化しようとする研究が進められている。特に、検証の過程ではプログラムの満たすべき性質を、ある定理として証明することが必要となる。

本論文では、 Prolog によって実現した Boyer-Moore 型定理証明システム (以下 BMTP と略称) について述べる。 BMTP は元来、関数型の Lisp プログラムの満たすべき性質を定理として証明するものであるが、論理型の Prolog プログラムを対象とした検証システムを実現する際にも有益な technique を提供するものと考える。

2. システムの概要

このシステムでは、次のような式の真偽を証明する。

```
[equal,[flat1,X],[flat2,X,nil]]
```

flat1 は二分木 X の葉を一列のリストに並べる関数である。flat2 は同様の事を別のアルゴリズムで行う関数である。そこで、上の式は同じ仕様に対する異なる二つの関数が、同じ値を与えるものであることを主張している。

2.1 プログラムの構成

システムのプログラムモジュール構成を図-1 に示す。

Main モジュールは、以下のサブモジュールの制御並びに、ユーザーインタフェースを行う。Definition モジュールは、データタイプと関数の定義を行う。Simplification モジュールは、等式による書換え、データタイプの計算、関数の展開等による式の簡単化を行う。Other Heuristics モジュールは、帰納法適用の前処理としていくつかの書換え規則を試みる。Induction モジュールは、帰納的に定義されたデータタイプや再帰関数に注目して帰納的証明を行う。

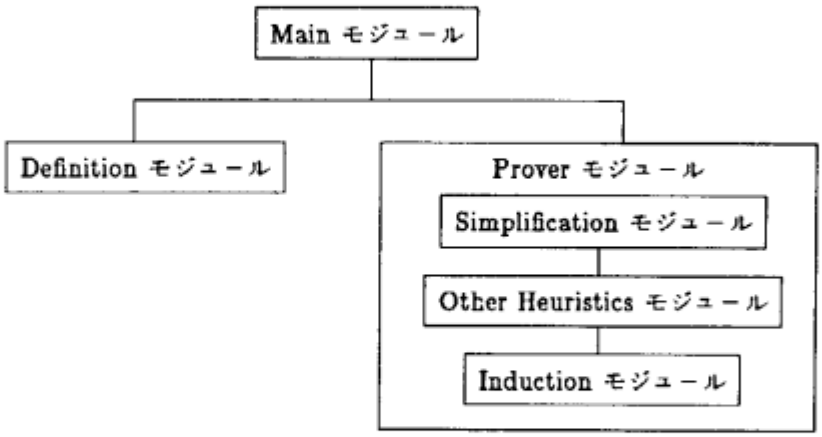

図-1 システムのプログラムモジュール構成

2. 2 システムの機能

このシステムにおける証明過程を図-2 によって説明する。以下、証明しようとする式を流体に準える。

まず、証明しようとする式を (A) に注ぐ。(B) において、式を clause 形式に変換し、(C) に流下する。(C) 内では、様々な書換えを行って clause を簡単化するが、真偽値にまで書換わった時、clause は "蒸発" して証明終了となる。(C) 内で "蒸発" しきれない (clause がこれ以上書換わらない) 状況に達した時、弁 (D) を開き、Other Heuristics 槽 (E) に流下する。(E) 内で clause が書換わった時は、再び (C) に戻される。(E) 内でもこれ以上書換わらない状況に達した時、弁 (F) を開き Induction 槽 (G) に流下する。(G) 内では、帰納法により、base case と induction step に相当するいくつかの新しい clause が生成され、これらは再び (C) に戻される。

以上の過程で、総ての槽 (C),(E),(G) 内で clause が完全に "蒸発" してしまった時、最初の式が証明された事になる。

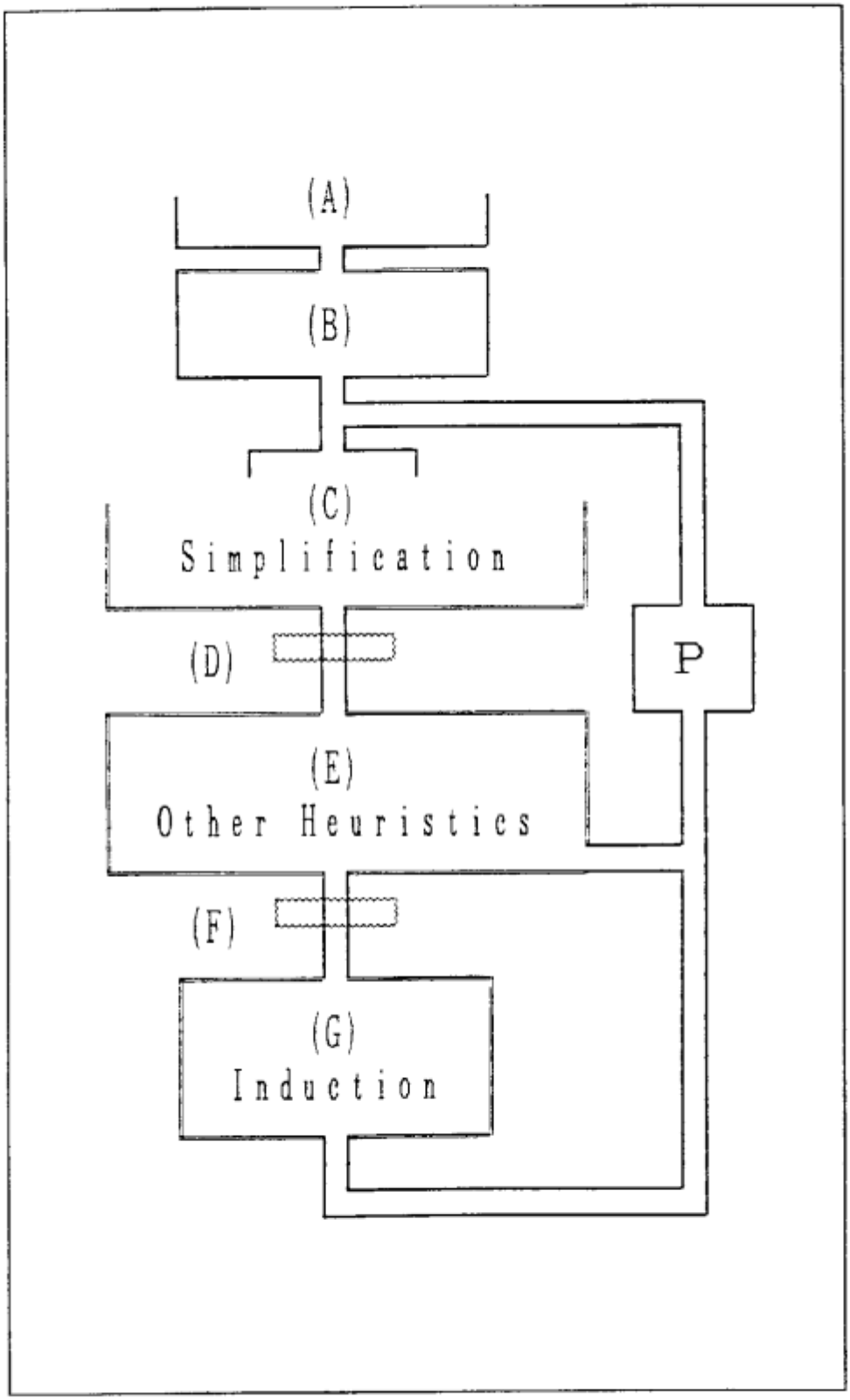

図-2 システム動作説明図

3. システムの機能詳細

3. 1 Definition モジュール

このモジュールでは、新しいデータタイプを導入したり、定められた基準に適合する関数定義を公理として受付ける。

(1) データタイプの定義

例えば次のようにして、二分木データタイプ cons が定義される。

```
Add data type "cons" of two arguments with:
    constructor: cons
    recognizer: listp
    accessors: car,cdr
```

これにより、以下のような公理群がシステムに加えられる。

(i) `[listp,[cons,X1,X2]]`

(ii) `[equal,[car,[cons,X1,X2]],X1]`

(iii) `[equal,[cons,[car,X],[cdr,X]],X]`

(iv) `[implies,[listp,X],[lessp,[cdr,X],X]]`

(2) 関数定義

再帰的に定義された関数は、再帰呼出しがある measure に関して単調減少する事が証明されるものに限り、受付けられる。

例えば、関数 append を次のように定義する。

```
[append,X,Y]
  =  [if,[listp,X],
         [cons,[car,X],[append,[cdr,X],Y]],
         Y]
```

この定義は、データタイプ cons の導入の際、システムに加えられた公理 (iv) により、

```
[append,X,Y] ≻ [append,[cdr,X],Y]        when [listp,X]
```

が示されるので、受付けることができる。

3. 2 Simplification モジュール

この定理証明システムの中核を成すモジュールである。このモジュールでは、equivalance を保存するような基本的な書換えを行い、式を可能なかぎり簡単化する。

(1) if、equal と論理演算子

このシステムは次のような真偽値 t、f、並びに関数 if、equal に関する基本公理をもつ。

```
t ≠ f
X = Y →  [equal,X,Y] = t
X ≠ Y →  [equal,X,Y] = f
X = f →  [if,X,Y,Z]  = Z
X ≠ f →  [if,X,Y,Z]  = Y
```

また、論理演算子は次のように if で定義されている。

```
[not,P]       = [if,P,f,t]
[and,P,Q]     = [if,P,[if,Q,t,f],f]
[or,P,Q]      = [if,P,t,[if,Q,t,f]]
[implies,P,Q] = [if,P,[if,Q,t,f],t]
```

一般に、等式は左辺から右辺への書換え規則として使われる。

(2) type による簡単化

まずシステムは、式中に含まれる関数に対する type prescription や証明ステップにおける仮定を用いて式の type set を計算し、それらの情報を用いて簡単化を試みる。type set とは、その項が持つ可能性のある値のデータタイプの集合である。その集合の要素は t、f またはデータタイプの recognizer であり、全体集合は universe と表現される。例えば [cons,X,Y] の type set は cons のデータタイプ定義より [listp]、変数 X の type set は universe である。これらの情報を基に項は簡単化される。

今 [equal,[append,X,nil],[add1,Y]] の簡単化を考える。append の type set は append の関数定義より listp かまたは第二引数の type set であるとわかっているので [listp,litatom] である。また [add1,Y] の type set はデータタイプ定義より [numberp] である。こうして equal の両辺の type set の積集合は空なので、結局 f に簡単化される。

(3) 公理と lemma

type set による簡単化に失敗すると、システムは公理や lemma を使った書換えを試みる。これらは、データタイプ定義時に加えられる公理、以前証明された定理、またはユーザーが与える公理や lemma である。項を書換えるのに適当な公理か lemma が見つかると、それを用いて項はただちに書換えられる。適当な公理か lemma が複数ある場合は最初に検索されたものが選ばれる。

一般に公理や lemma は次の形をしている。

```
[implies,Hyp,[equal,LHS,RHS]]
```

これは、 Hyp が証明できるときに LHS と match する項を RHS に置き換えるという書換え規則として用いられる。今書換えようとする項を

```
[append,[cons,A,B],nil]
```

とし、

```
[implies,[plistp,X],[equal,[append,X,nil],X]]
```

なる lemma があるとする。ここで関数 plistp はリストの最後の要素が nil の時 t を返す関数である。

この時、システムは、項

```
[append,[cons,A,B],nil]
```

と lemma の LHS である [append,X,nil] との match を試み、成功するとその環境 X=[cons,A,B] で Hyp : [plistp,X] を証明しようとする。他の条件より* これが証明された時には、

```
[append,[cons,A,B],nil]
```

は [cons,A,B] に書換えられる。

Hyp が証明できなければ、別の lemma が試される。

(4) 関数定義

すべての lemma が試されてもなお項が書換えられない場合、その項に現われる関数呼出しを定義の body に展開して簡単化を試みる。今、展開しようとする n 項の関数を $[F_n, T_1, \ldots, T_n]$ とする。まず、各引数 T_i は、 (2)-(4) の一連の書換え規則に従い書換えられる。これらを T'_i とする。 F_n が非再帰関数であるならば、無条件にbodyで展開される。そして、その body に対して (2)-(4) の一連の処理が再帰的に行われる。

---

* ここでは、B=nil または、 [plistp,B]=t の時、証明できる。

次に F_n が再帰関数の場合を考える。関数 F_n の measured position[*1] の引数が explicit value[*2] ならば無条件に body で展開される。そうでないならば、まず body が書換えられる。その書換えられた body (body' とする) が下の条件のいずれかを満たせば、$[F_n, T_1, \ldots, T_n]$ は body' に書換えられる。さもなければ、$[F_n, T'_1, \ldots, T'_n]$ が返される。

その条件とは、次の 3 つである。

(イ) body'の再帰呼出しの measured position の項が今簡単化しようとしている式に存在する、則ち、展開することにより measured position に新しい項が現れるわけではない場合。

(ロ) body' の再帰呼び出しの引数の方が、$[F_n, T_1, \ldots, T_n]$ より多くの explicit value を含んでいる場合。

(ハ) body'の再帰呼び出しの measured position の項の方が、$[F_n, T_1, \ldots, T_n]$ の対応する measured position の引数より単純な項である場合。ここでいう "単純" とは、項に含まれる関数記号の数が少ないことである。

Simplification モジュールでは、式に含まれる各項に対して以上のような一連の操作を行い式の簡単化を試みる。簡単化できなかった場合は、induction を用いて証明を行うことになる。このシステムでは、induction を "上手に" 適用するために式の変形を行う heuristics を持っている。次にその heuristics について述べる。

3. 3 Other Heuristics モジュール

このモジュールは、次の Induction モジュールにおいて帰納法を適用するための前処理として、いくつかの書換え規則を試みる。

(1) Destructor の消去

ここでは、データタイプの導入の際に定義された accessor から成る項と constructor から成る項との "トレード" を行う。

例えば下の式において仮定から X がリストであるとわかるので、X を [cons,X1,X2] に [car,X] と [cdr,X] を各々 X1、X2 に書換える。

この変換によって次の induction step において X1 や X2 に関する帰納法を適用することが可能となる。

---

*1 measured position : 関数呼び出しのある引数よりその body の再帰呼び出しの対応する引数の方がある measure に関して減少しているような引数の位置。

*2 explicit value : t か f かデータタイプの bottom object または、データタイプ constructor 項でその引数が explicit value であるもの。

```
[implies,[and,[listp,X],
              [equal,[flat2,[car,X],
                            [flat2,[cdr,X],Y]],
                     [append,[flat1,[car,X]],
                             [flat2,[cdr,X],Y]]],
              [equal,[flat2,[cdr,X],Y],
                     [append,[flat1,[cdr,X]],Y]]],
         [equal,[flat2,[car,X],
                       [flat2,[cdr,X],Y]],
                [append,[append,[flat1,[car,X]],
                                [flat1,[cdr,X]]],
                        Y]]]
```

(2) Cross-fertilization

仮定内の equality を結論部に適用する。

例えば下の式において

```
[append,[flat1,X1],[flat2,X2,Y]] を [flat2,X1,[flat2,X2,Y]]
```

に置換え、その equal 項を取り去る。

```
[implies,[and,[equal,[flat2,X1,[flat2,X2,Y]],
                      [append,[flat1,X1],
                              [flat2,X2,Y]]],
              [equal,[flat2,X2,Y],
                     [append,[flat1,X2],Y]]],
         [equal,[flat2,X1,[flat2,X2,Y]],
                [append,[append,[flat1,X1],
                                [flat1,X2]],
                        Y]]]
```

一般に証明しようとする式の仮定に項

```
[equal,X,Y]
```

があったとする。ここで、X も Y も explicit value でなく、また

```
[p,<任意の項>,<Yを含む任意の項>]
```

の形の別の項があるとする。その時、p のアーギュメントの右側だけ X を Y で置き換えその等式を元の式から取り除く。この書換え操作を Cross-fertilization と呼んでいる。

(3) 一般化

ここでは、式の中に 2 箇所以上で現れている項、または equal の両側に現れている項を変数に置換えて一般化と呼ばれる操作を行う。

例えば下の式で [flat1,X1], [flat1,X2] を各々 X15、X16 で置換える。

```
[equal,[append,[flat1,X1],
               [append,[flat1,X2],Y]],
       [append,[append,[flat1,X1],[flat1,X2]],
               Y]]
```

その結果、次のような式が得られるが、これは append の associativity を主張しているものに他ならない。

```
[equal,[append,X15,[append,X16,Y]],
       [append,[append,X15,X16],Y]]
```

ここで flat1、flat2 は、常にリストを生成するということがそれぞれの type prescription よりわかるので、仮定として

```
[and,[listp,[flat1,X15]],[listp,[flat2,X16]]]
```

が付け加えられる。

(4) 不要項の消去

以上の書換えを行った結果、式中に結論部と無関係な仮定となっている項が残る場合がある。ここでは、これらの項を取り除く。ここでいう無関係とは、その項と共通の変数を持つ他の項が式中に存在しない場合をいう。

例えば下の式

```
[implies,[plistp,B],
         [equal,[reverse,[append,Z,
                                 [cons,A,nil]]],
                [cons,A,[reverse,Z]]]]
```

において仮定の [plistp,B] は結論部の equality に含まれない変数 B に関するものなので、取り除く。

3. 4 Induction

このモジュールでは、帰納的に定義されたデータオブジェクト並びに再帰関数について帰納法による証明を行う。

ここで用いられるヒューリスティックスは、このシステムの中核となるものである。帰納法の適用は、再帰関数の定義時において有用な情報を生成する処理と、証明時においてこれらの情報を収集、選択して帰納法の形を定める処理とに分けられる。以下各々の処理について述べる。

(1) 再帰関数の定義時の処理

再帰関数に対して、 induction の形を示唆する induction template を生成する。

下に前出の関数 flat1、flat2 を例として示す。

— flat1 の定義とそれに対する induction template :

```
[flat1,X]
 =   [if,[listp,X],
         [append,[flat1,[car,X]],
                 [flat1,[cdr,X]]],
         [cons,X,nil]]

   induction template for the term: [flat1,X]

     measured subset: <X>
     case: [listp,X]
      (1) X → [car,X]
      (2) X → [cdr,X]
```

— flat2 の定義とそれに対する induction template :

```
[flat2,X,Y]
 =   [if,[listp,X],
         [flat2,[car,X],[flat2,[cdr,X],Y]],
         [cons,X,Y]]

   induction template for the term: [flat2,X,Y]

     measured subset: <X>
     case: [listp,X]
      (1) X → [car,X],  Y → [flat2,[cdr,X],Y]
      (2) X → [cdr,X],  Y → Y
```

(2) Inductionの実行

証明しようとする式の中に現れている総ての再帰関数について、生成されている induction template を収集する。

それらの induction template をいくつかの基準により取捨選択して、最終的に 1 つの induction scheme を構成する。

```
scheme for the terms: [flat1,X], [flat2,X,Y]
   case: [listp,X]
  (1)  X → [car,X],   Y → [flat2,[cdr,X],Y]
  (2)  X → [cdr,X],   Y → Y
```

こうして得られた induction scheme を適用することにより、元の式は、 base case と induction step から成る clause の積に書換わる。それらは再び Simplifier に戻される。

Base case:

```
[implies,[not,[listp,X]],
         [equal,[flat2,X,Y],
                [append,[flat1,X],Y]]]
```

Induction Step:

```
[implies,[and,[listp,X],
              [equal,[flat2,[car,X],
                            [flat2,[cdr,X],Y]],
                     [append,[flat1,[car,X]],
                             [flat2,[cdr,X],Y]]],
              [equal,[flat2,[cdr,X],Y],
                     [append,[flat1,[cdr,X]],Y]]],
         [equal,[flat2,X,Y],
                [append,[flat1,X],Y]]]
```

4. システムの評価

システムの評価基準としては、検証可能なプログラムのクラス、検証に要する時間、メモリ容量等があげられる。

このシステムでは、再帰関数については 3.1 の (1) で述べたような measure と、well-founded な減少関数とが与えられるものに限られる。証明に要する時間は式中に含まれる書換え可能な項の数、適用される書換え規則の総数、 induction がかかる回数、

simplification や induction によって分割される clause の個数等によって決まる。メモリ容量については、証明された式が後の証明のための書換え規則としてシステムに追加される分と、一回の証明に要する作業領域分とに大別される。

このシステムでは、初等代数における加算、乗算の交換、結合則等が、またリスト処理においては append、flatten、intersection 等 Lisp 基本関数に関する性質の証明が CPU 時間で数分の order で実行されている。

インプリメンテーション言語としては、DEC-10 Prolog を採用した。プログラムサイズは、表-1 に示すようになった。Prolog のソースコードは Lisp 等他の言語に比べセマンティカルな密度が高く、その分プログラム開発工数も少ないのが特長と言える。

module name	source (lines)	interpreted code (words)	compiled code (words)
Main	480	5,222	5,693
Definition	778	8,462	7,408
Simplification	1,860	20,400	20,703
Other Heuristics	743	7,438	7,214
Induction	791	8,812	8,638
Utilities	589	5,370	6,256
total	5,242	55,004	55,912

表-1. プログラムサイズ

5. むすび

このシステムでは、関数型の Lisp プログラムを対象としたが、Prolog 等、論理型プログラムに対しても同様の検証システムを構成することを検討中である。

謝辞

本論文は ICOT 外注作業（発仕 3401-03 号）の成果を踏まえたものである。ICOT・第二研究室の古川室長を初め、関係の方々には貴重な御指導並びに御助言を頂き、深く感謝の意を表します。

参考文献

1. Boyer, R.S. and Moore, J.S. : A Computational Logic, Academic Press (1979).
2. Cohen, P.R. and Feigenbaum, E.A. Eds., : The Handbook of Artificial Intelligence, Volume 3, XII-D., Pitman Books Ltd. (1982) pp.102-113.

付録

以下、このシステムで証明された定理のいくつかと、その証明に要したCPU時間を付す。

Theorem COMMUTATIVITY.OF.PLUS

```
[equal,[plus,X,Y],[plus,Y,X]]
```

CPU time : 33.0 seconds

Theorem ASSOCIATIVITY.OF.TIMES

```
[equal,[times,[times,X,Y],Z],
       [times,X,[times,Y,Z]]]
```

CPU time : 137.0 seconds

Theorem LESSP.PLUS.SUB1

```
[not,[lessp,[plus,X,Y],[sub1,Y]]]
```

CPU time : 48.4 seconds

Theorem EQUAL.TIMES.ZERO

```
[equal,[equal,[times,X,Y],0],
       [or,[zerop,X],[zerop,Y]]]
```

CPU time : 33.5 seconds

Theorem REVERSE.REVERSE

```
[implies,[plistp,X],
         [equal,[reverse,[reverse,X]],
                X]]
```

CPU time : 69.3 seconds

Theorem FLAT1.FLAT2

```
[equal,[flat2,X,Y],
       [append,[flat1,X],Y]]
```

CPU time : 52.1 seconds

Theorem MEMBER.REVERSE

```
[implies,[member,X,[reverse,Y]],
         [member,X,Y]]
```

CPU time : 138.8 seconds

Theorem MEMBER.INTERSECT

```
[implies,[and,[member,A,B],
              [member,A,C]],
         [member,A,[intersect,B,C]]]
```

CPU time : 23.6 seconds