TR-0746

遅延モデル生成法に基づく
定理証明器

長谷川　隆三、越村　三幸、藤田　博

March, 1992

ICOT

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

(03)3456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

遅延モデル生成法に基づく定理証明器

A Lazy Model Generation Based Theorem Prover

長谷川 隆三
新世代コンピュータ技術開発機構

越村 三幸
東芝情報システム

藤田 博
三菱電機中央研究所

概要

本論文では効率のよい定理証明器を実現する 遅延モデル生成法と呼ばれる新しい方法について述べる。モデル生成法に基づく定理証明の仕事はモデルの要素になるアトムの生成と充足可能性のテストであるが、一般にモデル生成過程で要する計算量とメモリ空間が膨大になる傾向がある。遅延モデル生成法は証明を得るために無関係なアトムや不必要なアトムは生成されないようにすることを目的とする。実際、どのような種類の証明法でも情報の生成や保持の際、要求駆動的制御を考慮にいれないと、限られた計算資源のもとで巨大な探索空間を持つ問題の定理証明は成功しない。遅延モデル生成法は、計算量及びメモリ空間のオーダを下げることが可能であり、かつ資源を効率よく利用するような要求駆動制御が容易にできるため、並列定理証明器の作成に非常に適している。

1 はじめに

本研究は 並列論理型言語 KL1 [UC90] のプログラミング技術を使った一階述語論理の高速な定理証明器の開発を目的とする。

Prolog のプログラミング技術を使った定理証明器には Stickel の PTTP[Sti88] や Schumann の SETHEO[Sch89] のようなモデル消去法に基づく後向き推論型のものや、Manthey と Bry [MB88] による SATCHMO のようなモデル生成法に基づく前向き推論型のものがある。

我々は KL1 のプログラミング技術を用いた定理証明器の開発にあたり、その第一歩としてモデル生成法を採用した。その理由は、以下の通りである。1) KL1 は コミッティッドチョイス型言語であり、後戻り機構に基づく「完全な」ユニフィケーションを支援していないので、ユニフィケーションが不要であるという SATCHMO の性質は証明器を KL1 で実装するにあたって非常に好ましいものである。2) Lukasiewicz 問題 [Ove90] のような深い推論（長い証明）が必要な問題解決に向いた補題化や 包摂テストや他の 削除戦略を容易に組み込める。

モデル生成法に基づく証明器を実装する場合、モデル候補に現れるアトムと問

節の前件部との 連言照合（後で定義する）を行なう際に生じる冗長性を回避することが重要になる。このために我々は RAMS 法 [FH91] と MERC 法 [Has91] を提案した。

モデル生成法に基づく証明器の効率化に関する課題でもっと重要なのは全計算量及び証明過程で必要なメモリ空間をいかに削減するかである。この問題はLukasiewicz の問題のような種類の問題を扱う時にはより深刻になる。この問題を解決するためにはモデル生成法に基づく証明を generate-and-test と捉え、検査器 (tester) から要求がきた時のみ生成器 (generator) がモデル要素を生成する要求駆動の考えが重要である。

SATCHMO の場合、モデルの拡張（生成）と モデル棄却 (テスト) は assert/retract 及び Prolog の後戻り機構を用いて、逐次的に完全に同期して行なわれているため、生成プロセスが暴走する危険は生じない。しかしながら、モデル要素を一つ生成するたびにテストを行なうという固定化された制御の中に戦略を採り入れるのは難しい。例えば複数の要素から適当なものを選んで次の要素にするという重み付けヒューリスティクスのような戦略は使えない。

これに対して、 OTTER [McC90] のような証明器は多くの戦略をともに採用できるように汎用的かつ柔軟に設計されている。しかしながら、 generate-and-test という考え方なしに hyper-resolution のような推論を行なった場合、生成される レゾルベントが組み合わせ的爆発を引き起こす危険があり、また、要求駆動の考えに基づく制御なしには証明プロセス中に無駄な計算が生じうる。さらに悪いことには、この計算は並列環境下ではメモリ空間の爆発を引き起こすことになるかもしれない。

本論文では、上記の問題の解決をはかり、'generate-only-at-test' という要求駆動的考え方に基づく遅延モデル生成 (Lazy Model Generation) という新しい方法を提案し、いくつかの効率改善手法を示す。

2 モデル生成法

本論文中では節 (clause) は以下のような 含意形式で記述する。

$$A_1, A_2, \ldots, A_n \rightarrow C_1; C_2; \ldots; C_m$$

ただし $A_i (1 \leq i \leq n)$ 及び $C_j (1 \leq j \leq m)$ はアトム、前件部 (antecedent) は $A_1, A_2, \ldots, A_n$ の 連言、後件部 (consequent) は $C_1, C_2, \ldots, C_m$ の 選言である。節はその前件部が $true (n = 0)$ のとき正節 (positive clause)、後件部が $false (m = 0)$ のとき負節 (negative clause)、それ以外 $(n \neq 0, m \neq 0)$ のときは混合節 (mixed clause) と呼ぶ。正節及び混合節を generator 節、負節を tester 節とも呼ぶ。

モデル生成法は以下の二つの規則を持つ。

- モデル拡張規則: ある generator 節 $A \rightarrow C$ に対し、ある代入 σ が存在して、モデル候補 M で $A\sigma$ が充足可能であり、かつ $C\sigma$ が充足不能ならば、$C\sigma$ を M に加えてこのモデル候補を拡張する。

- モデル棄却規則: ある tester 節 $A \rightarrow false$ に対し、ある代入 σ のもとで、前件部 $A\sigma$ がモデル候補 M で充足可能ならば、このモデル候補を棄却する。

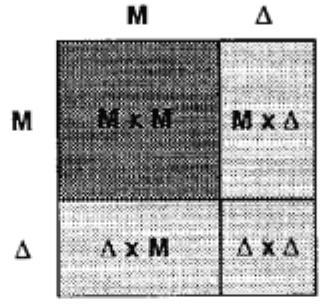
ここで、$A\sigma$ を得る過程を、モデル要素と前件部リテラルの連言照合 (conjunctive matching, CJM) と呼ぶ。正節の前件部 ($true$) は任意のモデルで充足可能であることに注意。

モデル生成法では、空なモデル候補集合から始めて、与えられた節集合に対するモデルを構築していく。もし節集合が充足可能であればモデルが存在し、そのモデルが有限であれば、この証明手続きはそれを発見して停止する。また、あらゆるモデル構築の可能性を調べた結果、モデルが構築できなかった場合は、その節集合は充足不可能である。

2.1 連言照合における冗長性の回避

前件部に二つのリテラルを持つ節 C を考える。この節に対する 連言照合を行なうためには、モデル候補 M から要素の対を選ぶ必要がある。その対が今 M から選ばれたものとする。そして M から選ばれた全ての対に対する連言照合の結果、モデル拡張候補 Δ が得られたとする。そうすると、次の段階では $M + \Delta$ から要素対を選ぶことになる。この対の数は全部で

$$(M + \Delta)^2 = M \times M + M \times \Delta + \Delta \times M + \Delta \times \Delta.$$



なる。

ここで $M \times M$ 個の対については、前の段階で既に選ばれているので、これらの対に対する 連言照合は冗長になる。つまり、Δ からの要素を少なくとも一つ持つ対のみを選べばよい。前件部のリテラル数が 3 以上の場合にも同様のことがいえる。

上述のような冗長性を除去するために OTTER がとっている方法は以下の通りである。節

$$A_1, A_2, \ldots, A_n \rightarrow C_1; C_2; \ldots; C_m.$$

に対して、まず、モデル拡張候補 Δ に含まれるアトムに対して、A_i を一つ選び照合させる。そして、残ったリテラル

$$\{A_1, A_2, \ldots, A_n\} - \{A_i\},$$

それぞれに対して $M + \Delta$ から取り出したアトムを照合させる。この操作を $A_1, \ldots, A_n$ に対して行なう。ただし M は現在の モデル候補である。

この方法は以下に述べるように、前件部リテラルが 2 以上の節に対する連言照合の際に重複計算が生じる。対 $< A_i, A_j >$ $(i \neq j)$ を考えると、これは $< \Delta, M + \Delta >$ と $< M + \Delta, \Delta >$ の双方に照合するので $\Delta * \Delta$ の 連言照合は重複してしまう。この冗長性は MERC 法 [Has91] によって除外できる。

我々が Δ-M 法と呼んでいる実際の実装では二つの前件部リテラルを持つ節に対しては

$$\{[\Delta, \Delta], [\Delta, M], [M, \Delta]\}$$

のようなパターンを用意し、三つの前件部リテラルを持つ節に対しては

$$\{[\Delta, \Delta, \Delta], [\Delta, \Delta, M], [\Delta, M, \Delta], [M, \Delta, \Delta],$$

$$[\Delta, M, M], [M, \Delta, M], [M, M, \Delta]\}$$

のようなパターンを用意する。以下同様である。このパターンに従って、与えられた節の 連言照合に必要となる可能なアトムの組み合わせを全て作り出す。

Δ-M 法は MERC 法に似ているが、MERC 法はこのパターンの代わりに、パターンの個数分の節を用意する必要がある。それぞれの節の前件部は対応するパターンに対する連言照合のみを行なう。

RAMS 法 [FH91] はこれらとは異なる冗長性除去の方法であり、同じモデル要素に対して同じリテラルを再照合させることのないように、モデル要素に対してリテラル A_i の照合に成功した場合、その成功情報 (変数の束縛情報) をすべて覚えておくというものである。このように成功情報を覚えることにより、Δ-M 法と MERC 法にある冗長な計算を回避することができる。例えば、$[M, \Delta, \Delta]$ と $[M, \Delta, M]$ のパターンに対する計算では共通の部分パターン $[M, \Delta]$ の再計算は、RAMS 法では行われない。しかし、成功情報を格納するのに多大なメモリが必要となる欠点がある。

3 モデル生成法アルゴリズム

本節と次節ではいくつかのモデル生成法のアルゴリズムを示し、計算量とメモリ領域の観点からそれらを比較する。以下の節では、議論を簡単にするために問題は Horn 節のみで与えられるものと仮定する。しかし、考え方は非 Horn 節で記述された問題にも一般化可能である。

$M := \phi$;
$D := \{A \mid (true \rightarrow A) \in$ a set of given clauses$\}$;
while $D \neq \phi$ **do begin**
$D := D - \Delta$;
if $CJM_{Tester}(\Delta, M) \ni false$ **then return**(success);
$new := CJM_{Generator}(\Delta, M)$;
$M := M + \Delta$;
$new' := subsumption(new, M + D)$;
$D := D + new'$;
end return(fail)

図 1: 基本アルゴリズム

3.1 基本アルゴリズム

図1に示す基本アルゴリズムは、モデル生成法を幅優先探索で実行する基本的なアルゴリズムである。これはOTTER[McC90]で行われるhyper-resolutionアルゴリズムと本質的に同一のアルゴリズムである。[1]

ここで、Mはモデル候補を、Dはモデル拡張候補(モデル拡張規則の適用の結果としてMに付加されるべきアトムの集合)を、ΔはDの部分集合をそれぞれ表す。MとDの初期値は空集合及び正節の後件部のアトムの集合である。このアルゴリズムを一回りする間に1) Dの部分集合Δを選び、2) ΔとMを使って棄却テスト(tester節に対する連言照合)を行なう。3) 反駁に成功するとアルゴリズムは停止。さもなければ、4) ΔとMを使ってモデル拡張(generator節に対する連言照合)を行ない、5) 新しく生成されたアトム集合newの$M + D$に対する包摂テストを行なう。このサイクルの始めにDが空ならば反駁が失敗、即ちモデルが見つかったことになり、このアルゴリズムは停止する。

連言照合及び包摂テストはそれぞれ以下のような集合の上の関数として表される。

$$
\begin{aligned}
&CJM_{Cs}(\Delta, M) = \\
&\quad \{\sigma C \mid \sigma A_1, \ldots, \sigma A_n \rightarrow \sigma C \\
&\qquad \wedge \ A_1, \ldots, A_n \rightarrow C \in Cs \\
&\qquad \wedge \ \sigma A_i = \sigma B (B \in M + \Delta)(1 \leq \forall i \leq n) \\
&\qquad \wedge \ \exists i (1 \leq i \leq n) \sigma A_i = \sigma B (B \in \Delta)\}
\end{aligned}
$$

$$
subsumption(\Delta, M) = \{C \in \Delta \mid \forall B \in M (B \text{ dosen't subsume } C)\}
$$

[1] OTTERでは、次の小節で述べる全テストアルゴリズムにあるように、newに含まれる各リテラルの生成後すぐに単位反駁テストを行うという改良が施されている。

$M := \phi$;
$D := \{A \mid (true \rightarrow A) \in$ a set of given clauses$\}$;
while $D \neq \phi$ **do begin**
$D := D - \Delta$;
$new := CJM_{Generator}(\Delta, M)$;
$M := M + \Delta$;
$new' := subsumption(new, M + D)$;
if $CJM_{Tester}(new', M + D) \ni false$ **then return**(success);
$D := D + new'$;
end return(fail)

図 2: 全テストアルゴリズム

3.2 全テストアルゴリズム

図2に基本アルゴリズムを改良した全テストアルゴリズムを示す。このアルゴリズムでは、1) D から Δ を選び、2) Δ と M を使ってモデル拡張を行い、Δ に対する次の世代のアトム集合 new を生成する。3) $M + D$ に対して new の包摂テストを行った後、4) 包摂テストを通過した new' と $M + D$ を使って棄却テストを行う。

この修正は見たところ非常に小さいものだが、次章で述べるように、これによって全計算量と空間量が大幅に削減される。重要な点は以下の通りである。Δ のかわりに新しく生成されたアトム集合 new のすべてのアトムに対して包摂テストの後、棄却テストを行なう。この結果、new の中に棄却アトム[2]（X とする）があった場合、直ちに偽が検出される。基本アルゴリズムとは異なり、棄却アトムが見過ごされ、D に付加されてしまうことはない。従って、X が見つかる前に無関係なアトムが生成 / テストされることはない。

3.3 遅延アルゴリズム

図3に基本アルゴリズムの別の改良である遅延アルゴリズムを示す。このアルゴリズムでは、generator 節に対応するプロセスと tester 節に対応するプロセスの二つが交信しながら独立並行に動作するものと考える。

tester プロセスは、1) generator プロセスに Δ を要求し、2) これまでのテスト済みのアトム集合（$M + D$）に対して Δ の包摂テストを行った後、3) 棄却テストを行う。

generator プロセスは、1) 一回のモデル拡張の結果を蓄える Buf が空のとき、これを埋める準備をするため、D からアトム e を選んで Buf にモデル拡張コードを設定(**delay** CJM) しておく。そして、2) tester プロセスからの Δ 要

[2]棄却アトム (falsifying atom) とは、X 自身のみで負節の前件部を直接満たすか、又は X と $M + D$ 中のアトムとの組で、負節の前件部を満たす、ようなアトムのことである。

process tester:

repeat forever

$request(generator, \Delta)$;

$\Delta' := subsumption(\Delta, M + D)$;

if $CJM_T(\Delta', M + D) \in \perp$ **then return**(success);

$D := D + \Delta'$.

process generator:

repeat forever

while $Buf = \phi$ **do begin**

$D := D - e$; $Buf :=$ **delay**$CJM_G(e, M)$; $M := M + e$ **end**;

$wait(tester)$;

$\Delta :=$**force**Buf;

until $D = \phi$ **and** $Buf = \phi$.

図 3: 遅延アルゴリズム

求を待って、3) Buf に Δ を要求(**force** Buf)する。

ここで、**delay** はオペレータで、被作用子の関数の実行を遅延する。従って、1) の時点で $CJM_G(e, M)$ はただちには計算されないで、Buf 上にコードとして置かれる。その後、3) で **force** オペレータが Buf に作用する時点で、**delay** された関数が起動され、必要とされるだけのデータが作られる。これにより、毎回 tester プロセスの要求する Δ 分の拡張アトムだけが実際に生成されるわけである。なお、Buf には残りの生成を継続するために再び **delay** された CJM 関数コード(continuation)が置かれる。

M、D へのアトムの登録は拡張アトムの生成順、及びこれらに対するテストの順番が基本アルゴリズムと同じになるように行われる。従って、遅延アルゴリズムは基本アルゴリズムにおけるアトム間の論理的な生成順序を保ちつつ、生成及びテストの相対的なスピードを均等化したものといえる。これにより、生成の行き過ぎによる計算量及びメモリ消費量の無駄をなくすことができるわけである。

3.4 単位負節に対する最適化

単位負節に対しては上述の三つのアルゴリズムはさらに改良できる。これには二通りの方法がある。

一つは 先読み方式と呼ばれる動的な方法である。これは、単位負節による棄却テストのためだけにモデル拡張に使用されるアトムを先回りして生成するものである。すなわち、new を生成した直後に次の段階で生成されるべき new_{next} を生成する。そして new_{next} に対して単位負節で棄却テストを行う。このテストに失敗したとき、new は記憶するが、new_{next} は廃棄する。

$$< \Delta, M > \Rightarrow generate(A_1, A_2 \rightarrow C) \Rightarrow new$$

$$< new, M + D > \Rightarrow generate(A_1, A_2 \rightarrow C) \Rightarrow new_{next}$$

$$new_{next} \Rightarrow test(A \rightarrow false)$$

new_{next} を記憶しないで良い理由は、単位負節による棄却テストが M も D も必要とせず、new_{next} 自身で行えるからである。これに対し、2 リテラル以上の負節の場合は new_{next} を廃棄してしまうと、new_{next} 同士の組み合わせに対する棄却テストができず、棄却テストが不完全になる。

new_{next} は次の段階の new として再び生成される。従ってある連言照合はモデル要素の同一の組に対して二度行なわれるかもしれない。しかしながらこれによる計算量の増加は、全計算量に比べてオーダー的に無視できる。詳細は次章で議論する。

もう一つの方式は、部分評価を使った静的な方法である。すなわち、単位負節と generator 節の後件部をユニファイし、部分評価された非単位負節を得る。

$$Generator: \ A_1, A_2 \rightarrow C.$$

$$Unit\ tester: \ A \rightarrow false.$$

$$\Downarrow$$

$$Non\text{-}unit\ tester: \ \sigma A_1, \sigma A_2 \rightarrow false.$$

$$where\ \sigma C = \sigma A$$

部分評価方式の連言照合の計算量は、先読み方式と同じである。しかし、部分評価方式では証明器そのものが修正されることはないので、先読み方式より単純といえる。さらに負節（ゴール）の持つ情報を伝播できるため、探索空間の刈り込みが可能になる場合もある。しかしながら部分計算の結果、一般には節の数が増加するので、逆に実行性能が悪くなる場合もある。

以上二つの最適化は、負節が単位節のときには、計算量と記憶量の点でオーダ的に同等の改善が期待できる。

4 複雑さの解析

本節では、前節で提案されたアルゴリズムに必要な計算量とメモリ空間について議論する。

議論をできるだけ簡単にするために次のことを仮定する。1) 問題は前件部が二個、後件部が一個のリテラルのみからなる generator 節と、高々二個のリテラルからなる tester 節を含む。2) D から取り出される Δ は単一のアトムと仮定する。3) 連言照合でユニフィケーションが成功し、かつその結果が包摂テストを通過する割合 (生存率と呼ぶ) を $\rho(0 \leq \rho \leq 1)$ とする。また、4) Δ の取り出し順序及びモデル拡張に置けるアトムの生成順は、アルゴリズムを問わず一定であるとする。

以下では、アルゴリズムの複雑度の解析にあたって2.1節に示したようなマトリックスを用いる。このマトリックスは L_1, L_2 の二つの前件リテラルに対する連言照合操作を表す。このマトリックスのi行j列の要素は、L_1 にi番目、L_2 にj番目のモデル要素を照合させることを意味する。連言照合、包摂テストに成功して確定したモデル要素には、1から順に番号がふられる。例えば、下図では1をaxiomとすると 1×1 の結果を2、1×2 の結果を3、2×1 の結果を4、2×2 の結果を5、というように確定モデル要素に番号をふる。

	1	2	3	...	i
1	2	3	6		1×i
2	4	5	7	...	2×i
3	8	9	10		3×i
:		:			:
i	i×1	i×2	i×3	...	i×i

ここで、i 番目のモデル要素に対するモデル拡張 $CJM_{Generator}(i, M)$ とは、M の全要素 $1,\ldots,i-1$ に対して $i\times(1\ldots i-1),(1\ldots i-1)\times i, i\times i$ の連言照合を行い、新たなモデル拡張候補を生成することを指す。i番目のモデル要素に対するモデル拡張 $CJM_{Generator}(i, M)$ が完了した時点では、総計 i^2 回の連言照合が行われ、生存率を ρ とすると、総計 $\lfloor\rho i^2\rfloor$ 個の要素が $M+D$ に格納されている[3]。

4.1 基本アルゴリズム

図4は 基本アルゴリズムに必要な計算量とメモリ空間を示す。左の正方形の面積は generator 節で行なわれる 連言照合の回数を表す。右の正方形の面積は二つのリテラルを持つ負節の行なう 連言照合の回数を表し、この正方形の下の線の長さは単位負節の 連言照合の回数を表す。M、D は、それぞれの領域で行われる連言照合の結果得られる要素集合である。基本アルゴリズムでは、M の要素に対してはモデル拡張及び棄却テストが完了し、D の要素に対してはモデル拡張も棄却テストもまだ行われていない。

図4は、$false$ が検出された時点のものである(以後、図では $false$ を $\perp$ と表示)。ここで、m 番目のモデル要素に対する モデル拡張 $CJM_{Generator}(m, M)$ によって、棄却アトム X が生成されるものとする。このアトムは、$\lfloor\rho(m-1)^2\rfloor+1$ 番目から $\lfloor\rho m^2\rfloor$ 番目のいずれかの要素であるが、簡単のため、$\lfloor\rho m^2\rfloor$ 番目とする。

基本アルゴリズムでは、まずモデル拡張候補 D から要素 Δ を一つ取りだし、tester 節による棄却テスト $CJM_{Tester}(\Delta, M)$ を行う。全テストアルゴリズムとは異なり、Δ に対するモデル拡張 $CJM_{Generator}(\Delta, M)$ の結果である new に対

[3] i 個の要素が M に、$\lfloor\rho i^2\rfloor-i$ 個の要素が D に格納される。

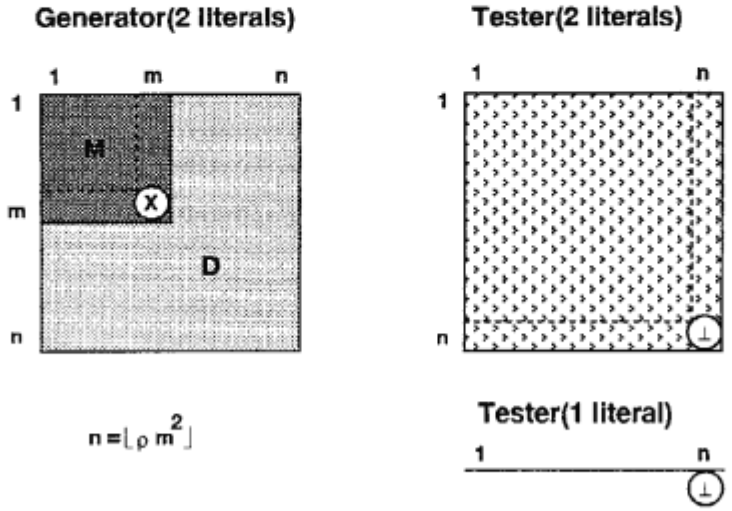


図 4: 基本アルゴリズムの複雑度

して棄却テストを行っていないため、棄却アトム X が new に含まれていてもこの時点では見過ごされてしまい、一旦 D に格納されることになる。従って、D から X が取り出されて棄却テストが行われるまで、偽であることは判明しない。

この結果、$m+1$ 番目から $\lfloor \rho m^2 \rfloor$ 番目の要素に対する無駄な連言照合が generator 節で行われてしまい、X に対する棄却テストが終了した時点では、generator 節と tester 節でそれぞれ合計 $\lfloor \rho^2 m^4 \rfloor$ 個の 連言照合が行われることになる。$M + D$ の中には、$\lfloor \rho^3 m^4 \rfloor$ 個のアトムが保持される。

4.2 全テストアルゴリズム

図 5は、全テストアルゴリズムにおいて $false$ が検出された時点の図である。X, M は 基本アルゴリズムの時と同じ意味であるが、D については、その要素に対してモデル拡張 (generator の連言照合) は開始されていないのに、棄却テストは完了しているという点で異なる。

全テストアルゴリズムでは、棄却テストは Δ に対してではなく、Δ に対するモデル拡張の結果である new に対して行なわれている。従って、m 番目のモデル要素に対するモデル拡張 $CJM_{Generator}(m, M)$ の結果として棄却アトム X が生成されると、この要素に対する 棄却テストによって直ちに 偽であることが判明する。このとき、 tester 節で行われる 連言照合の回数は、$M + D$ の個数が $\lfloor \rho(m \times m) \rfloor$ なので[4]、$(\rho(m \times m))^2 = \rho^2 m^4$ となり、これは基本アルゴリズムと同じである。

一方、generator 節に関しては、1 から m 番目の要素に対してのみ 連言照合が行われ、$m+1$ 番目から $\lfloor \rho m^2 \rfloor$ 番目の要素に対する無駄な連言照合は行われな

[4] M の個数は $\lfloor (m/\rho)^{1/2} \rfloor$ 、D の個数は $\lfloor \rho m^2 \rfloor - \lfloor (m/\rho)^{1/2} \rfloor$

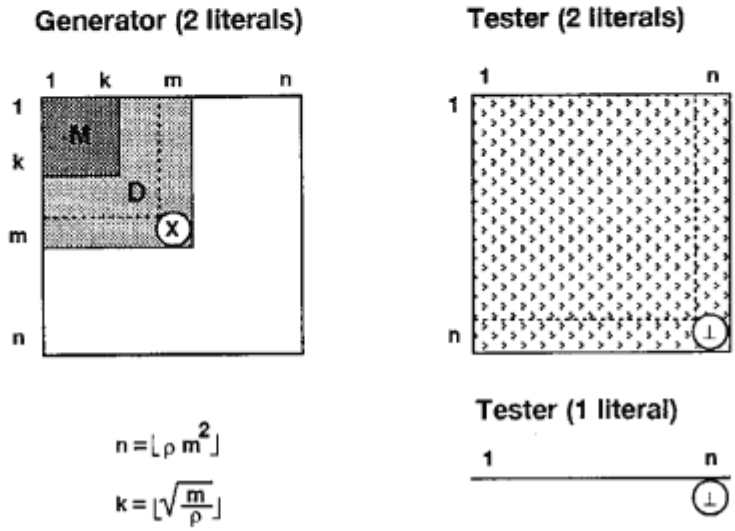


図 5: 全テストアルゴリズムの複雑度

い[5]。従って、 generator 節で行なわれる 連言照合、及び生成されたアトムを格納するメモリ空間は、それぞれ基本アルゴリズムにおいて $O(\rho^2 m^4)$ 、 $O(\rho^3 m^4)$ であったものが $O(m^2)$ 、 $O(\rho m^2)$ に削減される。

4.3 遅延アルゴリズム

遅延アルゴリズムでは、 tester 節からの要求があるごとに毎回一個ずつモデル要素 Δ が generator 節によって生成される。この新しく生成されたモデル要素は、これまでのテスト済みの要素集合（全テストアルゴリズムにおける $M + D$ と等価）に対して棄却テスト及び 包摂テストが行なわれる。

遅延アルゴリズムの複雑さは、オーダ的には、図5で示される全テストアルゴリズムと同じである。というのは、遅延アルゴリズムでは、モデル要素に対する棄却テストは生成後すぐに行なわれるため、棄却アトム X が生成された時点で偽が検出されるからである。

遅延アルゴリズムは要求に応じてモデル要素をつくっていくため、基本アルゴリズムのような作りすぎによる無駄はない。また、生成したモデル要素については直ちに棄却テストを行っているため、全テストアルゴリズムと同様の効果が生じる。

遅延アルゴリズムと 全テストアルゴリズムの複雑さが同じであるのは、全テストアルゴリズムにおいては、モデル拡張結果 new に対するテストの完了を待って次のモデル拡張を開始する、という逐次性を前提においているからである。しかし、並列環境下で全テストアルゴリズムを実行すると、適切なプロセス制御をしないと、棄却アトム X が生成 / テストされる前に generator プロセスが暴走

[5]左の正方形の逆 L 字形の空白部分

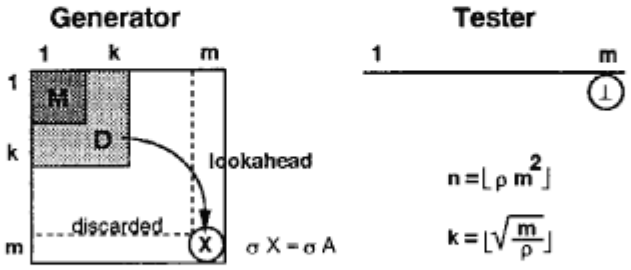


図 6: 全テスト / 遅延アルゴリズムにおける先読み最適化の効果

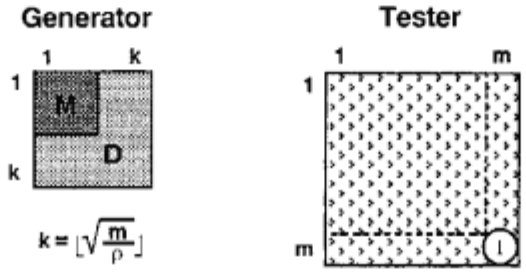


図 7: 全テスト / 遅延アルゴリズムにおける部分評価の効果

して不要なアトムを多量に生成し、無駄な包摂テストが行われてしまう可能性がある。遅延アルゴリズムは、このような並列環境下においても、 generator プロセスの暴走を自然に防ぐことができ、 generator 節における計算量とメモリ空間のオーダを $O(m^2)$、 $O(\rho m^2)$ というしかるべき値に保つことが可能である。

4.4 単位負節に対する最適化

先読みと部分評価による最適化は、いずれも各アルゴリズムにおいてテストされるアトム集合より一世代先のアトム集合を調べるという先行テスト効果を持っている。

generator 節の前件部リテラル数が 2 という仮定のもとでは、モデル拡張候補の要素数はモデル候補の要素数の自乗オーダになる。

基本アルゴリズムでは、モデル拡張候補に対する棄却テストは行われておらずモデル候補に対してのみ行われる。これに先行テストを施すとモデル拡張候補に対する棄却テストが可能になり、 generator 節の連言照合並びにメモリ空間のオーダを平方根分下げることができる。この結果、全テスト及び遅延アルゴリズムと同じ効果が得られる。この最適化は、とりもなおさず OTTER でとられているアルゴリズムそのものである。

一方、全テストと遅延アルゴリズムではモデル拡張候補に対するテストは完全に行われている。これに先行テストを施すとモデル拡張候補からさらに次の世代のアトム集合に対する棄却テストが可能になり、やはりこれらのアルゴリズムにおける generator 節の連言照合並びにメモリ空間のオーダを平方根分下げることができる。

基本アルゴリズムを示す図4において、D の領域のアトムは最適化を施すことによって、作られなくなるので generator 節の計算量及びメモリ空間のオーダはそれぞれ $O(m^2)$、$O(\rho m^2)$ まで減少する。

全テストアルゴリズムを示す図5において、M の領域のアトムが生成された段階で D の領域のアトムに対して棄却テストが完了するので、全テスト / 遅延アルゴリズムにおける generator 節の計算量及びメモリ空間のオーダは、それぞれ $O(m/\rho)$ $O(m)$ まで減少する (図6参照)。

一方、部分評価による最適化 (図7参照) では、部分評価によって得られる 2 リテラル負節を用いた 棄却テストは先読み最適化における new_{next} の生成及び単位負節とのテストに相当する。また、図6及び7において、M で示された領域はどちらの最適化手法をとっても二度計算される。従って、先読み方式と部分評価方式は計算量とメモリ空間のオーダという点では等しい最適化手法である。

4.5 複雑さの解析結果の要約

以上述べてきた計算量の議論をまとめると表1のようになる。T、S、G はそれぞれ棄却テスト、包摂テスト、モデル拡張に要する計算量を、M は必要となるメモリ空間を表す。表中 α は、$1 \leq \alpha \leq 2$ の範囲の値をとり、包摂テストの効率を表す。$\alpha = 1$ のときは、ハッシング効果が高く定数オーダで包摂テストが行えることを意味し、$\alpha = 2$ のときは、線形探索で包摂テストを行う時のように、包摂対象要素数に比例する手間で包摂テストを行うことを意味する。condensed detachment 問題の場合、ハッシング効果は薄く、α はほぼ 2 に近い値をとる。

基本、全テスト / 遅延、遅延先読みといくにしたがって、必要メモリ量は着実に平方根ずつ減少している。これは、生成された要素数が減ることを意味しており、その結果、包摂対象要素も減るから、包摂テストの計算量も同様に削減される。$\alpha = 2$ の場合を考えると、最もコストのかかる計算は包摂テストであるから、包摂テストの計算量の削減は、全体の計算量の削減になる。一方 $\alpha = 1$ の場合には、2 リテラル負節の棄却テストが最も高価な計算となるが、これはいずれの計算法でも一定であり、遅延計算による効果は定数倍のスピードアップに止まる。いずれにしても、遅延計算により全計算量は棄却テストの計算量で抑えられるようになることが分かる。

5 KL1 による 遅延連言照合の実装

本節では 遅延モデル生成の核部分である 遅延連言照合プログラムの KL1 による実装を示す。遅延連言照合プログラムは 先行連言照合 (eager CJM) から導かれる。

問題の単純化のために、二つの前件部リテラルを持つ節 $A_1, A_2 \rightarrow C_1$ の連言

表 1: 各アルゴリズムの計算量

	単位負節			
	T	S	G	M
基本	$O(\rho m^2)$	$O(\mu\rho^2 m^{4\alpha})$	$O(\rho^2 m^4)$	$O(\rho^3 m^4)$
全テスト / 遅延	$O(\rho m^2)$	$O(\mu m^{2\alpha})$	$O(\rho m^2)$	$O(\rho m^2)$
遅延先読み	$O(m^2)$	$O(\mu/\rho m^{\alpha})$	$O(m/\rho)$	$O(m)$

	2 リテラル負節			
	T	S	G	M
基本	$O(\rho^2 m^4)$	$O(\mu\rho^2 m^{4\alpha})$	$O(\rho^2 m^4)$	$O(\rho^3 m^4)$
全テスト / 遅延	$O(\rho^2 m^4)$	$O(\mu m^{2\alpha})$	$O(m^2)$	$O(\rho m^2)$

† m は 基本アルゴリズムにおいて false が検出された時点での モデル候補数
‡ ρ は生成するアトムの生存率、μ は連言照合成功率 ($\rho \leq \mu$)、α は包摂テストの効率

照合を考える。この場合、必要な組合せとして $\Delta \times M$, $M \times \Delta$, $\Delta \times \Delta$ だけを考えればよい。

5.1 先行連言照合

まず、図 8に 先行連言照合のプログラムを示す。述語 clause/5 は前件部リテラルと M(M) 及び Δ(DM) の要素との 連言照合を行ない、成功したすべての組合せについてその束縛情報を代入した後件部を差分リスト (Ui,Uo) の形で返す。

述語 ante/4 は第 1 引数に従って M または Δ とあるリテラル Lis のユニフィケーションを行なう。

5.2 遅延連言照合

二つの 遅延連言照合プログラムを示す。一つは図 9に示すような 継続法に基づくプログラムであり、もう一つは並列実行に向いたプロセス指向のプログラムである。

5.2.1 継続法に基づく実装

図 8及び図 9からわかるように、先行連言照合と継続法に基づく 遅延連言照合の相違は、後者が 継続スタック S を持ち、L で生成すべき要素数を与えている点である。遅延連言照合ではこれらによって 遅延機構を実現する。

継続スタックは実行が遅延されているゴールのスタックである。例えば clause/7 の場合、二つのボディゴールがスタックに積まれて実行が遅延させられる。これらのゴールは現在のゴール計算が終了すると、一つずつスタックから取り出されて実行される。

このプログラムは 継続スタックを計算する関数と見なすことができる。従って、呼び出し側が 継続スタックを管理する必要があり、モデル拡張候補の要素 Δ が呼び出し側で必要になれば、 継続スタックを引数にして clause/7 を呼び出せ

```
clause(M,DM, C, Ui,Uo) :-
  ante([{DM,A1},{M,A2}], C, Um1,Uo),  % Δ * M
  ante([{DM,A2},{M,A1}], C, Um2,Um1), % M * Δ
  ante([{DM,A1},{DM,A2}], C, Ui,Um2). % Δ * Δ

ante(R, C, Ui,Uo) :-
  new_env(C, Env), % create variable environment
  ante1(R, Env, C, Ui,Uo).

ante1([], Env, C, Ui,Uo) :-
  assignValueToVariable(C1, Env), Uo = [C1|Ui].
ante1([{M,Lis}|R], Env, C, Ui,Uo) :-
  literal(M,Lis, R, Env, C, Ui,Uo).

literal(M,Lis, R, Env, C, Ui,Uo) :- getNext(M, E, M1),
  (/* M is empty */ -> Uo = Ui;
   /* E is a element of M */ -> unify(Lis,E, Env,NEnv),
     (/* unification fail */ ->
       literal(M1,Lis, R, Env, C, Ui,Uo);
      /* unification success */ ->
       ante1(R, NEnv, C, Um,Uo),
       literal(M1,Lis, R, Env, C, Ui,Um))).
```

図 8: 先行連言照合のプログラム

```
clause(L, M,DM, C, S,NS, Uo) :-
  S1 = [ante([{DM,A2},{M,A1}], C),
        ante([{DM,A1},{DM,A2}], C)|S],
  ante(L, [{DM,A1},{M,A2}], C, S1,NS, Uo).

ante(L, R, C, S,NS, Uo) :-
  new_env(C, Env), ante1(L, R, Env, C, S,NS, Uo).

ante1(L, [], Env, C, S,NS, Uo) :-
  assignValueToVariable(C1, Env),
  Uo = [C1|Ui], L1 := L - 1,
  (L1 = 0 -> NS = S, Ui = [];
   L1 > 0 -> clauseCont(L1, S,NS, Ui)).
ante1(L, [{M,Lis}|R], Env, C, S,NS, Uo) :-
  literal(L, M,Lis, R, Env, C, S,NS, Uo).

literal(L, M,Lis, R, Env, C, S,NS, Uo) :- getNext(M, E, M1),
  (/* M is empty */ -> clauseCont(L, S,NS, Uo);
   /* E is an element of M */ -> unify(Lis,E, Env,NEnv),
     (/* unification fail */ ->
       literal(L, M1,Lis, R, Env, C, S,NS, Uo);
      /* unification success */ ->
       S1 = [literal(M1,Lis, R, Env, C)|S],
       ante1(L, R, NEnv, C, S1,NS, Uo))).

clauseCont(L, [ante(R,C)|S],NS, Uo) :-
  ante(L, R, C, S,NS, Uo).
clauseCont(L, [literal(M,Lis, R, Env, C)|S],NS, Uo) :-
  literal(L, M,Lis, R, Env, C, S,NS, Uo).
```

図 9: 継続法に基づくプログラム

ばよい。

5.2.2 プロセス指向型の実装

前節 5.2.1では、clause は一つの関数として定義された。これに対して、プロセス指向型のプログラムは並列実行環境に適したものである。プロセス指向型のプログラムでは generator プロセスである clause と tester プロセスが通信チャネルを介して協調的に働く。

プロセス指向型のプログラムは 継続法に基づくプログラムに引数を一つ追加することによって容易につくることができる。新たに加える引数 NL は generator と tester の間の通信チャネルを表す。

generator プロセスは要求された数だけアトムを生成すると tester プロセスからの次の要求を待つ。

```
ante1(L,NL, [], Env, C, S,NS, Uo) :-
  assignValueToVariable(C1, Env), Uo = [C1|Ui], L1 := L - 1,
  (L1 = 0 -> Ui = {Uii},  % mark Lth element
    (NL = {LL,NLL}  ->    % wait for the next L and NL
       clausesCont(LL,NLL, S, Uii));
   ...
```

tester プロセスはアトムがさらに必要になると generator プロセスに次の要求を出す。

```
...,NL,..., Uo,...) :- Uo = {Uoo} |     % reach Lth element
        NL = {LL,NLL},  % send the next L and NL
        ...
```

6 実験結果

6.1 広い探索空間を持つ Horn 節で記述された問題

これまでに提案したいくつかのアルゴリズムに関する実験結果を述べる前に、我々が解こうとしている問題を簡単に説明する。

扱う問題はすべて以下のように Horn 節のみの集合で記述される。

定理 4 (XGK [Ove90])

$$
\begin{aligned}
p(X),\ p(e(X,Y)) &\rightarrow p(Y).\\
true &\rightarrow p(e(X,e(e(Y,e(Z,X)),e(Z,Y)))).\\
p(e(e(e(a,e(b,c)),c),e(b,a))) &\rightarrow false.
\end{aligned}
$$

表 2: 性能に関する結果 (定理 4)

		基本	全テスト	遅延	遅延先読み	OTTER
time(sec)		>14000 (463.86)	409.17 (82.40)	407.58 (81.82)	210.45 (81.69)	409.16 (462.13)
unify		— (43981+74254)	1656+74800 (43981+4158)	1656+74737 (43981+4158)	81956+4095 (43981+4095)	1656+74800 (43981+74254)
subsumption test		— (5674)	5736 (596)	5736 (596)	593 (593)	5736 (5674)
Memory	M	— (272)	272 (63)	272 (63)	63 (63)	272 (272)
	D	- (1375)	1384 (209)	1384 (209)	209 (209)	1384 (1375)

定理 **6** (Lukasiewicz)

$$
\begin{aligned}
p(X),\ p(i(X,Y)) &\rightarrow p(Y).\\
true &\rightarrow p(i(X,i(Y,X))).\\
true &\rightarrow p(i(i(X,Y),i(i(Y,Z),i(X,Z)))).\\
true &\rightarrow p(i(i(i(X,Y),Y),i(i(Y,X),X))).\\
true &\rightarrow p(i(i(n(X),n(Y)),i(Y,X))).\\
p(i(i(a,b),i(i(c,a),i(c,b)))) &\rightarrow false.
\end{aligned}
$$

問題を解くには、オカーチェック付きの完全なユニフィクーションの他にも、いくつかの複雑なレゾルベントに対する 重み付けや 削除のような様々なヒューリスティクスが必要であり、また幅優先と深さ優先との間で探索戦略を変えることのできるような制御機構も必要である。定理 6 は証明が非常に短く、幅優先の探索戦略のみで簡単に解けるが、定理 4 は探索空間も広く、証明も長い。

これらの問題は以下のような性質を持つ。

- 連言照合及び 包摂テストの数は膨大である。
- 計算が進むにつれてモデル候補及びモデル拡張候補のサイズは非常に大きくなる。
  e.g. 10000 個のアトムが生成されると 連言照合の数は 1 億にものぼる。

6.2 性能の測定

実験結果を表 2、3に示す。ここでは、証明戦略としてソーティングなどの重み付けヒューリスティクスは用いず、項の大きさによる足切り及びトートロジー除去のみを用いた。

各アルゴリズムは全て KL1 で実装し、逐次推論マシン PSI-II[NN87] 上の擬似 Multi-PSI 上で測定した。ここで OTTER というのは、単位負節に対する最適化を施した基本アルゴリズムを KL1 で実装したものである。括弧の中の数字は単位負節の代わりに、これを部分評価して得られる 2 リテラル負節を用いた時

表 3: 性能に関する結果 (定理 6)

		基本	全テスト	遅延	遅延先読み	OTTER
time(sec)		1647.38	12.82	12.28	3.51	12.73
		(13.22)	(2.23)	(2.29)	(2.22)	(12.60)
unify		220+48618	220+598	220+556	947+68	220+598
		(358+550)	(358+70)	(358+68)	(358+68)	(358+550)
subsumption test		18804	331	327	28	331
		(328)	(32)	(28)	(28)	(328)
Memory	M	220	23	23	7	23
		(23)	(7)	(7)	(7)	(23)
	D	5369	197	197	16	197
		(195)	(16)	(16)	(16)	(195)

の結果である。 unify の欄で、プラス記号の左側は tester 節が行なった連言照合の回数、右側は generator 節が行なった連言照合の回数を表す。

この実験結果は、表 1 で示されている計算量を如実に反映したものになっている。例えば、定理 4 を部分評価なし(単位負節のまま)に解こうとすると、基本アルゴリズムでは、 14000 秒の間にゴールに到達できないが、全テスト及び 遅延アルゴリズムは約 400 秒でこの問題を解くことができる。基本、全テスト / 遅延共に最もコストのかかる計算は、棄却テストに要する計算でその差は、$O(\mu\rho^2 m^{4\alpha})/O(\mu m^{2\alpha}) = O(\rho^2 m^{2\alpha})$ となる。この差が上述の実行時間差となって現れている。

基本アルゴリズムと全テスト / 遅延アルゴリズムを比較すると、 tester 節の行なうユニフィケーションの数は等しい。しかし、 generator 節の行なうユニフィケーション及び包摂テストの数は、全テストと遅延アルゴリズムで減少する。その割合は、部分評価による最適化を施さないときは約 100 分の 1、施したときは約 10 分の 1 である。

先読み最適化によって、遅延アルゴリズムの性能は更に改善される。オーダ的には先読み最適化と部分評価による最適化は同じであるが、実験結果では差異が現れている。定理 4 では、遅延 + 部分評価の場合 81.82 秒、遅延先読みの場合 210.45 秒であった。この差は、 tester 節で行われるユニフィケーション回数の差である。この理由は、遅延先読みアルゴリズムでは、 generator 節 $p(X), p(e(X,Y)) \quad p(Y)$ によってアトムが作られた後、単位負節 $p(A) \rightarrow false$ との照合が行われるのに対し、部分評価による最適化では A の具体化情報が 2 リテラル負節 $p(X), p(e(X,A)) \rightarrow false$ の前件部に伝播し、照合の失敗が早めに検出されるためである。

部分評価は OTTER 以外のアルゴリズムには有効であるが、 OTTER には無効である。これは OTTER では単位負節に対しては既に先読み最適化が入っているが、非単位負節に関しては基本アルゴリズムに止まるためである。

これらの例題で特徴的なのは、第一に、生存率 (ρ) はほぼ一定であり全体の計算量は m で支配されること、第二に、包摂テストの効率 α が 2 に近い値をと

り、包摂テストがモデル要素の数に応じてコストが増大し、全体の計算量の中で包摂テストの占める計算量が支配的となることである。こうして、以上の結果は4.5節で行った複雑性の解析を極めてよく反映したものとなっている。

7 結論

深い推論を要する定理を証明する際には計算量及びメモリ空間の爆発を防ぐことが肝要であり、その解決法として我々は 遅延モデル生成法ならびにいくつかの性能改善手法を提案した。

逐次マシン上で実行する場合の計算量及びメモリ空間のオーダに関しては、本論文で述べた全テストアルゴリズムは遅延アルゴリズムと同様の効果があるので、全テストアルゴリズムを採用すれば充分である。しかし並列環境下では全テストアルゴリズムは無駄なアトム生成並びに包摂テストを行ってしまう危険がある。一方、遅延アルゴリズムは要求駆動的の概念に基づいており、並列マシン上で実行する場合に最も効果を発揮する。

実験結果から、全テスト / 遅延アルゴリズムの採用及び単位負節に対する最適化によって計算量及びメモリ空間が著しく削減できることが実証された。

遅延モデル生成法は Horn 節集合から非 Horn 節集合に容易に拡張できる。また、この考え方は hyper-resolution や set-of-support 戦略を使った他の一般の証明器にも適用可能である。

遅延モデル生成法は 'generate-only-at-testing' を実現したが、実はモデル生成においては 'generate-only-for-testing' という考え方の方がさらに重要である。

この言葉は *magic sets* や *relevancy testing* [WL89] のような探索空間の刈り込み戦略の採用を意味する。これらの戦略の制御構造は非常に単純であり、遅延モデル生成法にうまく組み込めると思われる。

我々は現在 遅延モデル生成法に基づく並列定理証明器を開発中であり、この結果は次の論文で報告する予定である。

謝辞

本研究を進めるにあたり、有益な助言を頂いた ICOT 第五研究室の藤田正幸氏、御討論頂いた三菱電機中央研究所の高橋和子氏に感謝します。

参考文献

[FH91] H. Fujita and R. Hasegawa, A Model-Generation Theorem Prover in KL1 Using Ramified Stack Algorithm, In *Proc. of the Eighth International Conference on Logic Programming*, The MIT Press, 1991.

[Has91] R. Hasegawa, A Parallel Model Generation Theorem Prover: MGTP and Further Research Plan, In Proc. of the Joint American-Japanese Workshop on Theorem Proving, Argonne, Illinois, 1991.

[MB88] R. Manthey and F. Bry, SATCHMO: a theorem prover implemented in Prolog, In *Proc. of CADE 88, Argonne, Illinois,* 1988.

[McC90] W. W. McCune, OTTER 2.0 Users Guide, Argonne National Laboratory, 1990.

[NN87] H. Nakashima and K. Nakajima, Hardware architecture of the sequential inference machine PSI-II, In *Proceedings of 1987 Symposium on Logic Programming,* Computer Society Press of the IEEE,1987.

[Ove90] R. Overbeek, Challenge Problems, (private communication) 1990.

[SL91] J. K. Slaney and E. L. Lusk, Parallelizing the Closure Computation in Automated Deduction, In *Proc. of CADE 90,* 1990.

[Sch89] J. Schumann, SETHEO: User's Manual, Technische Universität München, 1989.

[Sti88] M. E. Stickel, A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler, In *Journal of Automated Reasoning,* 4:353-380, 1988.

[UC90] K. Ueda and T. Chikayama, Design of the Kernel Language for the Parallel Inference Machine, In *Computer J.,* Dec. 1990.

[WL89] D. S. Wilson and D. W. Loveland, Incorporating Relevancy Testing in SATCHMO, CS-1989-24, Department of Computer Science, Duke University, Durham, North Carolina, 1989.