数理解析研究所講究録
1125 巻 2000 年 65-80

MGTP:モデル生成型定理証明システム -実装と応用-

MGTP: A Model Generation Theorem Prover - Its Implementation and Application -

越村 三幸 Miyuki KOSHIMURA
藤田 博 Hiroshi FUJITA
長谷川 隆三 Ryuzo HASEGAWA

九州大学大学院システム情報科学研究科
Graduate School of Information Science and Electrical Engineering
Kyushu University

概要

本論文では，モデル生成法に基づく一階述語論理の定理証明システム MGTP(Model Generation Theorem Prover) の実装と応用に関わる最近の研究成果について報告する．まず最初に，Java 言語による MGTP の実装技術について述べる．項や述語，節などの基本データ構造を工夫することにより，記号処理言語による従来の実装を上回る性能が得られた．次に，証明分岐に伴って生ずる冗長な推論を削除する，証明濃縮と畳込み法をモデル生成法に組み込む方法について論じ，その効果の定量的評価を与える．最後に，MGTP を利用した論証支援システムについて報告する．

1 はじめに

定理証明技術は，数学の定理証明に留まらず知識処理システムを構築するのにも重要である．モデル生成法 [17] は，一階述語論理の定理証明法の一つであり，節集合の充足可能性の判定を行う．一方で，モデル生成法は，その名の示すとおり節集合のモデルを生成する．特に節集合のエルブランモデルを (原理的に) 全て生成することができるので，定理証明のみならず，プログラム検証や論理プログラム，演繹データベース，非単調推論などの分野で有効である．

MGTP [2] は，モデル生成法に基づく定理証明システムである．本論文では先ず，Java 言語 [9] による MGTP の実装について，項や述語といった基本データ構造の設計を中心に述べる．Java は，ネットワーク社会の進展とともに登場し急速に普及した．Java には，オブジェクト指向やスレッド処理といった機能もあるが，記号処理プログラミングをする上で最も魅力的なのは，ゴミ集め機能を内蔵していることであろう．いうまでもなくゴミ集めは，Lisp や Prolog に代表される記号処理言語には内蔵されており，これによりプログラマは，不要メモリ領域の解放，という仕事から解放される．

したがって，「Java は記号処理プログラミング向き言語でもあるのではないか」，という観点から，我々は，Java による MGTP の実装 [7] を開始した．それまで MGTP は，KL1 や KLIC [22] といった論理型言語で実装されていた．この従来の実装の知見を踏まえつつ，Java による試行錯誤のプログラミングを通して得られた，幾つかの有用なプログラミング技法について述べる．これらの技法は，オブジェクト指向の性質を利用している．実験評価によれば，Java 版 MGTP は，KLIC 版 MGTP を上回る性能を示している．

モデル生成法は，超導出 (hyper resolution) 法の一種であるとみなすこともできるが，証明探索において分岐が生ずることから，むしろタブロ (tableau)

法 [13] に基づく証明手法に本質的には近いといえる．命題論理を対象にした計算量の解析によると，純粋なタブロ法は導出 (resolution) 法 [12] に比べ劣っており [23]，モデル生成法もこのタブロ法の非効率性を受け継いでいる．本論文では，この非効率性を改善する畳込み法 (folding-up) [16] のモデル生成法への組み込み手法を提案する [1]．

本手法は，証明の依存関係の解析に基づいて行なわれる．この解析により，証明濃縮 (proof condensation) [19] と呼ばれる冗長証明を除去する手法も同時に実装できる事も示す．二つの手法をモデル生成法に組み込む事により，モデル生成法の効率面での弱点を補強することに成功した．

最後に，モデル生成法の応用として MGTP 上での法的推論について述べる．MGTP を用いた法的推論では，法律の条文や判例はルールとして，事実は公理として表現され，これらを元に推論を行う．ところで，数学の定理証明では，ある証明法で結論付けられた定理が，別の証明法でくつがえされることはない．一方現実の裁判では，検察側と弁護側では，たとえ事実認定が同じとしても，全く異なった結論を主張することがある．これは，どの条文や判例を適用するかが立場によって異なるからである．

このような状況に対処するために，ルール間に優先度を導入する手法が提案されている．ルール間の優先度を変えることによって，検察官の結論も弁護士の結論も導びくことが可能となる．本論文では，ルール間優先度を陽に記述できる拡張論理型言語から MGTP が操作できる節形式への変換法 [4] を中心に述べる．

2 モデル生成法

本論文を通じて，節は次のように含意式の形で表現される：$A_1 \land \ldots \land A_n \rightarrow B_1 \lor \ldots \lor B_m$．ここで，$A_i (1 \leq i \leq n)$ 及び $B_j (1 \leq j \leq m)$ は原子論理式 (アトム) である．$\rightarrow$ の左側を前件部，右側を後件部という．

$n = 0$ のとき，前件部を特に $true$ と書き，正節と呼ぶ．一方，$m = 0$ のとき，後件部を特に $false$ と書き，負節と呼ぶ．それ以外の節 $(m \neq 0, n \neq 0)$ は混合節と呼ばれる．また，節が基礎アトムの集合 M に基礎代入 σ のもとで違反 (violated) しているとは，$\forall i (1 \leq i \leq n) A_i\sigma \in M \land \forall j (1 \leq j \leq m) B_j\sigma \notin M$ であることをいう．この条件の前半の検査を連言照合，後半の検査を包摂テストという．

図 1にモデル生成法による証明手続きを示す．手続き MG は，真であると考えられる基礎アトムの集合 Mc(モデル候補) と節集合 S を受け取り，S の (部分) 証明木を返す．ここで，証明木の (根でも葉でもない) 節点は基礎アトム，葉節点は $false$ か $unsatisfiable$ でラベル付けされる．なお，根はラベル付けされない．$satisfiable$ とラベル付けされた葉は，S のモデルが獲得されたことを示す *．得られた証明木の葉が全て $false$ でラベル付けされていれば S は充足不能，そうでなければ充足可能である．この場合は，葉の少なくとも一つは $satisfiable$ とラベル付けされているか，枝の少なくとも一つは無限に延びていくかのいずれかである．

3 Java 言語によるモデル生成法の実装

マルチメディア，インターネット，モバイルコンピューティング等のキーワードを看板とする，先進的情報システムに関する技術革新が進んでいる．Java 言語 [9] は，そのような時流に乗って急速に流行しつつある最新のプログラミング言語である．

一方，知能処理システムの分野においては，従来，プログラミング言語としては，Lisp を代表とする関数型言語や，Prolog を代表とする論理型言語が用いられることが多かった．もちろん，実用化に際してプログラムの実行速度が問題とされる場合等，限られた局面では C のような手続き型言語が用いられることもある．

「Java は C や C++の亜流にすぎない」,「オブジェクト指向ゆえに意味論が貧弱である」等の批判も聞かれるが，短期間にこれほど爆発的に普及したのは，単にインターネットのおかげだけではないだろう．これは，Java が単なる “化粧直し”，あるいは “ごった煮” の言語といったものではないことを示唆しているのではないか．特に我々は，Java の設計理念に，記号処理関係の専門家によるアイデアがほどよく盛り込まれている点に注目している．

実は，我々も当初「自動推論システムを，ネットワーク経由で容易に利用可能とする手段」という，

* $satisfiable$ とラベル付けされた葉の Mc(葉から根に至るパス上の節点にラベル付けされた基礎アトムの集合) が S のモデルとなる．

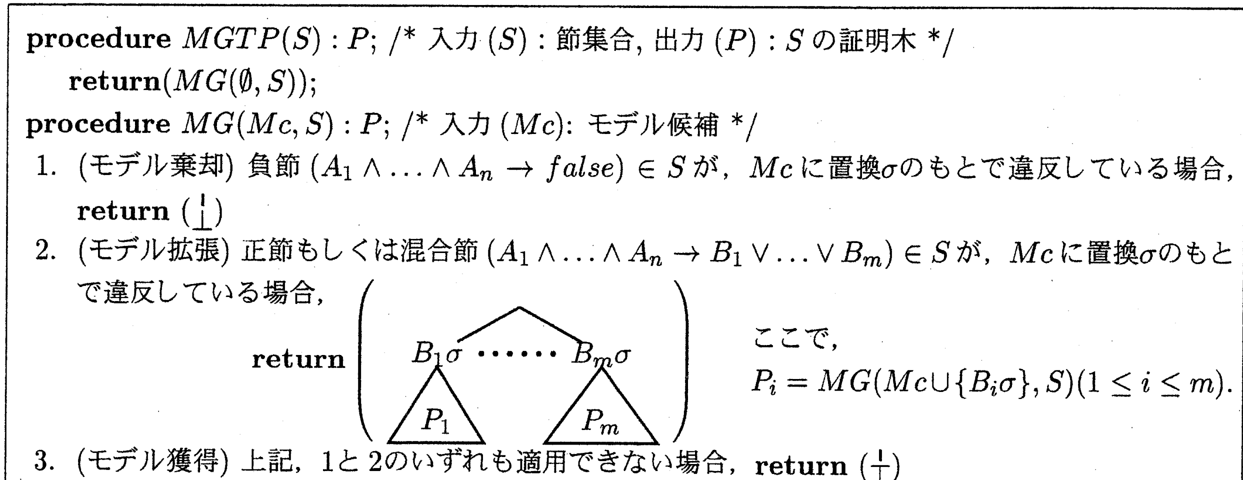


図1　モデル生成手続き

限定的かつプラグマティックな動機から Java によるシステム作りに取り掛かった．ところが，わずかの試行期間の後，我々は記号処理的なプログラムについても Java が Lisp や Prolog に比べて遜色のない，たいへんに書きやすい言語であるという印象を得た．定理証明プログラム MGTP [2] は，論理型言語によれば数ページ程度で書き下ろせるコンパクトさが特長の一つであるが，Java で書き直してみても，コード増加分はさほど大きくない．言語に不慣れであるにも関わらず，デバッグも大きな障害を感じずに捗る．

上のような次第で，我々は方針を改め，知識処理システムの核となる基本推論エンジンも含めて，すべて Java を用いて構築していくことにした．当然ながら，Java によるシステムは，従来の関数型／論理型言語による実装に対して，実行効率面でも競争に足るものであることが要求される．

本章では，定理証明系 MGTP を Java で実現する過程で開発したプログラミング技法や，実装上の工夫，改良点などについて，実験結果を踏まえて述べる．数々の改訂を経て完成した Java 版 MGTP は，従来の KLIC 版 MGTP を凌ぐ実行性能を示している．開発工数は，初期の版が Java 習得を含めて 1 人週ほど，その後の度重なる改訂が一応の収束に至るまでには 3 人月ほどを要した．

3.1 Java 言語機能の検討

Java はオブジェクト指向パラダイムに基づいているが，クラスメソッドを再帰関数とみなすことにより，関数的プログラミングが可能である．初期 MGTP の母体となった SATCHMO の Java 版を実装した淵 [8] によると，「関数プログラミング的コーディングは容易だが，実行効率は予想以上に悪い.」その後，オブジェクト指向に忠実な設計に改めることにより，良好な実行効率が得られたという．

また，論理型言語の Prolog 等を用いた知識処理プログラムの開発では，「実行可能な仕様」と言われるように，通常，プロトタイプとしての抽象仕様を次第に詳細化するという，トップダウン的手法が採られる．これに対し，“型意識の強い” オブジェクト指向言語の Java では，計算の対象が最初から具体的に定式化される必要があり，自ずからボトムアップ的手法が採られることになる．

我々は，プログラミングパラダイムの混合から生じる設計方針と言語仕様との齟齬に阻まれながら，さまざまな試行錯誤を重ねた．特に考慮すべき言語機能については，以下のようにまとめられる．

項や節などの構造データを表現するのに Java ではどのデータ型／クラスが適切であろうか．いくつかの選択肢が可能であり，記述の容易さと実行効率が必ずしも比例するとは限らない．

3.1.1 配列，ベクタ，リスト

配列は Java の組み込みデータ型で，要素の型が同一かつ長さ (要素数) が固定的である．

ベクタは，配列の要素を動的に追加／挿入／削除可能としたもので，java.util.Vector という標準パッケージの形で提供されている．

リストは再帰的クラスとしてユーザが定義することになる．例えば，

```
class List {
```

```
  Object head; List tail;
  List(Object h, List t) {
    head=h; tail=t;
  }
}
```

のように定義すると，(1) new List(Obj,Lis) で Lis の先頭に Obj を追加した新たな List インスタンスを生成する，(2) Lis.head で Lis の先頭を参照する，(3) Lis.tail で Lis の後部を参照する，という操作が可能である．

リスト，配列，ベクタの処理に要する時間の比率は，単純に要素アクセスだけで見ると約 1:1.3:10 くらいだが，複数要素処理のための for 文や while 文の制御を含めると，さらに差が開き，1:30:110 ほどにもなる．

3.1.2 データの複製

MGTP は選言に関する推論として場合分けを行うが，各場合を独立に処理するために，一般にはそれまでの推論環境の複製を作る必要がある．

一方，一般に Java のオブジェクトは，clone() と呼ばれるメソッドにより複製が作れる．ただし，リストのようなユーザ定義のオブジェクトについては，複製手続きを明示的にプログラムする必要がある．

配列，ベクタが 1 要素あたり約 70ns に対し，リストは約 4μs かかる†．約 60 倍も重いわけである．

3.1.3 再帰関数／繰り返し制御

関数型／論理型プログラミングにおいては，通常繰り返し計算を再帰関数 (述語) を用いて実現する．特に末尾再帰の形に対しては，環境の退避／復旧の操作を省いて最適化することも普通に行われている．

一方，Java では，メソッドの再帰的な呼出しが許されるが，通常末尾再帰最適化は行なわれない．これには for 文，while 文といった繰り返し文で対処できるが，再帰手続きにともなうデータの退避／復旧のため，スタックを明示的に記述する必要が生じる場合がある．

3.1.4 型の弁別

ある手続き f が異なるクラスのオブジェクト obj に対して適用される場合，クラスごとに “同名異形の” インスタンスメソッド m_fを定義しておく．メソッド呼出し $obj.m_f$によって，適切な手続き実体が起動される．これはオブジェクト指向処理系で最も肝腎な機構であり，当然ながらインデキシング等により効率よく実現されている．

一方，オブジェクトの型 (クラス) に応じて場合分け処理を行うのに，手続き本体中でinstanceof と if 文を用いることができる．しかし，メソッド弁別とinstanceof では，やはり前者が速い．以下で明らかにされるように，クラス定義の工夫次第で，instanceof の省略が可能となる場合がある．

3.1.5 キャスト

Java では，キャストと称する明示的な型指定の記述が要求される局面が多い．キャストは型の不一致による誤り回避にも効果的であるが，実行時にわずかなりともオーバヘッドを生じる．したがって，後述するような方法で，できる限りキャストを行わずに済むようなコーディングを心がけた方がよい．

3.1.6 論理式評価

たとえば，`return A && B;` と

`if (!A) return false; else return B;`

では，A の評価がfalse となる場合，後者の方がわずかながら速い！

3.2 Java による実装

知識処理システム一般，なかでも定理証明プログラムにおいては，項 (Term) が基本オブジェクトである．項を如何に適切な形で表現できるかが，効率よいシステム構築の鍵となる．

3.2.1 項

項は，定数項，変数項，複合項 (関数記号と 1 個以上の項の列 (引数) をもつ項)，に分類できる．この三つが項の具体例の雛型であり，これらを概念的に束ねる項は抽象的な型にすぎず，直接の具体例を生じない．そこで，Java では項を抽象クラス Term とし，三つの雛型を各々Term のサブクラスとして定義することができる．

3.2.1.1 2-フィールド項

まず，抽象クラス Term は全サブクラスのフィールド変数の共通集合をもつ，とするのが自然であろう．どの項も識別子をもつので，

```
abstract class Term {
  String name;
}
```

とするわけである．定数項は，識別子のみで十分であるが，複合項は引数列をもつので，

† Sun Ultra1 Model170, JDK1.1.3

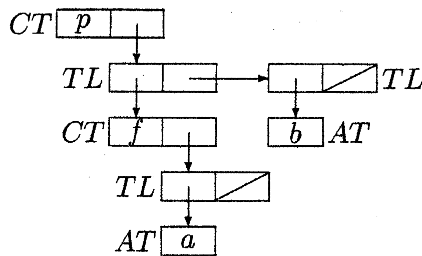


図2 2-フィールド項

```
class CTerm extends Term {
  TermList args;
}
```

のように，TermList クラス (Term を要素とするリスト) の args フィールドをもつであろう．TermList は配列 (ベクタ) で実現することもできるが，前節の検討に従えばリストで表現したほうがよい．この定義に基づいて複合項 $p(f(a),b)$ のオブジェクトを表現すると，図 2 のようになる．図中，AT, CT, TL は，それぞれ $ATerm, CTerm, TermList$ の略である．また，セル中，p, f, a, b 等は実際には文字列オブジェクトへの参照であるが，簡単のため直接値であるかのように書いている．斜線は空オブジェクト null を表している．

この図でただちに分かるように，複合項の表現に欠点がある．すなわち，複合項が入れ子になる (複合項の引数に複合項が現れる) 場合，引数への参照 (矢印) が 2 段になっていることである．たとえば，p から a までに，`p.args.head.args.head` と参照することになる．これは冗長で，単一化等，項に対する操作を遅くする．

3.2.1.2 3-フィールド項

上の欠点を回避するためには，二つのクラス CTerm と TermList を統合する必要がある．すなわち，

```
class CTerm extends Term {
  Term args; Term next;
}
```

のように定義しなおす．そうすると図 3のようになり，p から a までが`p.args.args` の 2 段参照で済む．

3.2.1.3 match メソッド

二つの項 t1,t2 の照合テストを，boolean メソッド match として定義する．t1 が定数項 ATerm の

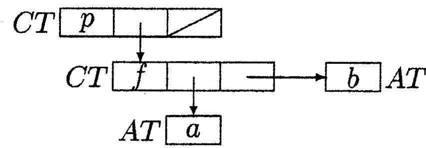


図3 3-フィールド項

場合は，

```
boolean match(Term t2) {
  if (!t2 instanceof ATerm)
    return false;
  return name==t2.name;
}
```

t1 が複合項 CTerm の場合は，

```
boolean match(Term t2) {
  if (!t2 instanceof CTerm) return false;
  if (name!=t2.name) return false;
  if (args.match(((CTerm)t2).args))
    return next.match(((CTerm)t2).next);
  return false;
}
```

のようになる．ここで，t1 のクラスに対応する match の弁別は処理系に任せることができる．

一方，t2 に対するクラス弁別は`instanceof` を用いて行なわれている．実は，この`instanceof` は次のようにして除去することができる．すなわち，抽象クラス Term を

```
abstract class Term {
  String name; Term args; Term next;
}
```

のように改める．サブクラスで“後から”追加されるはずのフィールド変数のすべての和集合を，初めから親クラスにもたせてしまうのである．ここでは，args, next フィールドは本来複合項のみに必要で，定数項にとっては基本的に無意味で冗長である．いささか不自然なクラス定義に見えるが，ここでは徹底的に無駄な計算 (クラス弁別とキャスト) を省くことを優先している．

こうして，定数項の match は，

```
boolean match(Term x) {
  if (name!=x.name) return false;
  return next.match(x.next);
}
```

となり，複合項の match は，

```
boolean match(Term x) {
  if (name!=x.name) return false;
  if (args.match(x.args))
```

```
    return next.match(x.next);
  return false;
}
```

となる．まず，当該項と比較項の識別子が比較される．不一致の場合ただちに false が返され，一致の場合 (複合項ではさらに args の match も一致した場合)，次の引数 next の照合に従う．

すなわち，当該項はある複合項の引数列の一部として出現していることを想定している．その場合，この match に至る以前に，複合項の識別子 (関数記号と引数の個数) に関する match が成功しているはずである．それゆえ，当該項が引数列の最後である場合には比較項も引数列の最後のはずであり，next は双方とも null のはずであるから，最後の match は省略可能である．この無駄な match を省くため，

```
class ATermL extends ATerm {
  boolean match(Term x){
    return name!=x.name;
  }
}
```

のように引数列の最後の項のための特別なサブクラスを設ける．(他のサブクラスについても同様．) ちなみに，命題リテラルも ATermL で表現できる．

3.2.1.4 変数項

変数項については，束縛値の扱いが問題である．すなわち，二つの項の照合の結果，変数がある値に束縛される場合，変数とその束縛値の対応を如何に内部表現するか，また，同じ変数名の別の出現に対し，束縛情報を如何に伝達するか，という問題である．

Lisp の "A-list" のように，変数の出現とその束縛値とを切り離し，$\langle$ 変数，束縛値 $\rangle$ の対を別の領域にもつ方式が考えられる．これによると，項は常にいわゆる環境 (変数束縛情報) との対として扱われなければならず，記述が繁雑で実行も遅くなる．記述を簡単にし，実行を速くするためには，変数オブジェクトが直接その束縛値を参照できる機構にしておくのが望ましい．

MGTP の場合，連言照合過程において，節の前件中の変数にモデル候補中のアトムに含まれる基礎項が束縛されるが，変数の最初 (最左) の出現のみが未束縛 (以後，$?X$ のように表示) で，同名の変数の後 (右) の出現はすべて既束縛 ($!X$ のように表示) である．そこで，以下のような二つの変数クラスを定める．

まず，非束縛変数の match を，

```
boolean match(Term x) {
  args=x; return next.match(x.next);
}
```

のように定義する．任意の項 x との照合が常に成功し，この変数項の束縛値が定まるが，これを本来は複合項の引数のために抽象クラス Term 中に設けておいた args フィールドにおく．型が同じなので代用するのである．

次に，束縛変数の match は，

```
boolean match(Term x) {
  if (args.args.vmatch(x))
    return next.match(x.next);
  return false;
}
```

と定義する．ここでも args フィールドを借用し，同名の非束縛変数項オブジェクトへの参照を書き込んでおくことにする．この束縛変数項オブジェクトに match メソッドが発行された時点では，対応する非束縛変数の match が実行済みで，束縛値が決定しているはずであるから，それを`args.args`で参照し，その束縛項オブジェクトに対してあらためて match を発行すればよい．ただし，この場合，束縛値の項の next フィールドに書き込まれた情報は無関係なので，無視しなければならない．next フィールドを無視する照合メソッドを`vmatch`とし，各項クラスごとに定義しておくことにする．(詳細は省略)

3.2.2 節の連言照合

節の前件がモデル候補によって充足されるか否か，をテストする連言照合は，MERC 法 [2](の簡易版) に基づいて行われる．MERC 法自体，冗長計算を極力避けるべく考案されたものであるが，実装にあたって冗長性の再混入に注意が必要である．

モデル候補を $\mathbf{M}$，モデル拡張アトムをΔ，拡張後のモデル候補を $\mathbf{M}_\Delta = \mathbf{M} \cup \Delta$とするとき，簡易 MERC 法では以下のように照合を行う．

たとえば，節 C1: $s(X), s(Y), s(Z) \to \ldots$ において，

(1) $s(?X) :: \Delta \;\&\&\; s(?Y) :: \mathbf{M}_\Delta \;\&\&\; s(?Z) :: \mathbf{M}_\Delta$
(2) $s(?Y) :: \Delta \;\&\&\; s(?X) :: \mathbf{M}_\Delta \;\&\&\; s(?Z) :: \mathbf{M}_\Delta$
(3) $s(?Z) :: \Delta \;\&\&\; s(?X) :: \mathbf{M}_\Delta \;\&\&\; s(?Y) :: \mathbf{M}_\Delta$

の三通りの連言照合を行う．ここで，"::" は照合操作，"&&" は連言演算を表す．

同様にして，節 C2: $p(X, Y), q(Y, Z) \to \ldots$ の

照合で，前件のリテラル評価順序を単純に入れ換えると，

(4) $p(?X,?Y) :: \Delta$ && $q(!Y,?Z) :: \mathbf{M}_\Delta$

(5) $q(!Y,?Z) :: \Delta$ && $p(?X,?Y) :: \mathbf{M}_\Delta$

となる．(5) で，束縛変数$!Y$の照合が非束縛変数$?Y$の照合に先行する，という不都合が生じている．

節 C2 のように，複数のリテラルに出現する変数 (共有変数) がある場合には，予め

$$p(?X,?Y) \mid q(!Y,?Z) \rightarrow \ldots$$
$$q(?Y,?Z) \mid p(?X,!Y) \rightarrow \ldots$$

のような 2 本の節を (内部表現として) 用意しておくことにする．ここで，“|” は論理的には “,” と同じ連言を意味するが，連言照合の際，“|” の左のリテラルはΔとのみ照合し，“|” の右は$\mathbf{M}_\Delta$と照合する，という操作的意味をもつ．

節 C1 のように前件リテラル間に共有変数がない場合は，リテラル順序の入れ換えで変数の束縛／非束縛の変化がないので，上のような節の多重化は不要である．

以上のような MERC 法に特有の処理も含め，MGTP の入力節読み込み，構文解析，内部表現への変換には，Java 用の「パーサジェネレータ」が便利に利用できる．

3.2.3 場合分け推論

MGTP では，ノンホーン節[‡]，$A_1 \wedge \ldots \wedge A_n \rightarrow B_1 \vee \ldots \vee B_m$ に関して場合分けによる推論を行なう．すなわち，現モデル候補 $\mathbf{M}$ を $\mathbf{M} \cup \{B_1\}, \ldots, \mathbf{M} \cup \{B_m\}$ の m 個のモデル候補に拡張し，各々のモデル候補ごとに推論が進められる．

各モデル候補の推論は独立，平行に進めてよい．一般に並列実行を行なう場合，場合分け直前のモデル候補 $\mathbf{M}$ とモデル拡張候補 $\mathbf{D}$[§]は，場合分け後に各場合で異なる変化を辿るので，ここにいわゆる「環境の複製」の必要性が生じる．しかし，2 節で述べたとおり，一般に構造の複製は高価である．そこで，元来共用できるデータに対しては複製を極力避けることが大切である．

‡ 節 $A_1 \wedge \ldots \wedge A_n \rightarrow B_1 \vee \ldots \vee B_m$が $m \leq 1$ を満たすときホーン節，$m > 1$ を満たすときノンホーン節と呼ぶ．ホーン節のみを含む節集合をホーン問題，ノンホーン節を含む節集合をノンホーン問題と呼ぶ．

§ モデル拡張候補とは，前件部がモデル候補で充足している節の後件部の集合である．実際の MGTP では，図 1 のモデル拡張のための違反節は，モデル拡張候補から選ばれる．

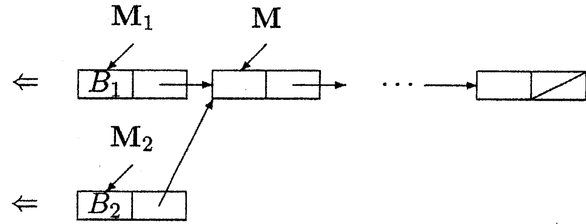


図 4 モデル候補の分岐

KLIC 版 MGTP では，モデル候補 $\mathbf{M}$ は，図 4 に示すような Last-In-First-Out 構造で表されている．すなわち，右端のセルから始めてモデル拡張とともに左に延びるリストである．$\mathbf{M}$ が，$B_1, B_2, \ldots$ による場合分けにより $\mathbf{M}_1, \mathbf{M}_2, \ldots$ に拡張されるとしよう．$\mathbf{M}_1$に属するすべてのアトムは，$\mathbf{M}_1$の tail ポインタを辿って参照できる．$\mathbf{M}_2$についても同様である．$\mathbf{M}$ から右は各 $\mathbf{M}_i$で共有され，複製の必要がない．また，ポインタの書き換えもない．

一方，モデル拡張候補 $\mathbf{D}$ は，通常 First-In-First-Out 構造の必要がある．ここでは，図 5に示すようなリスト構造を用いる．左端のセルから始めてモデル拡張候補の登録／削除に従って，登録ポインタ $\mathbf{D}^I$ならびに削除ポインタ $\mathbf{D}^O$が右に進むようなリストである．今，登録／削除ポインタが $\mathbf{D}^I_{sp}, \mathbf{D}^O_{sp}$ のとき，場合分け推論が適用されたとする．ケース 1 で，登録ポインタが $\mathbf{D}^I_{sp}$から $\mathbf{D}^I_1$へ，削除ポインタが $\mathbf{D}^O_{sp}$から $\mathbf{D}^O_1$へ進んだとしよう．ケース 1 の探索が完了すると，登録／削除ポインタが一旦 $\mathbf{D}^I_{sp}, \mathbf{D}^O_{sp}$に復帰する．そこから今度はケース 2 の探索が始まり，登録ポインタが $\mathbf{D}^I_{sp}$から $\mathbf{D}^I_2$へ，削除ポインタが $\mathbf{D}^O_{sp}$から $\mathbf{D}^O_2$へ進むことになる．このとき，登録ポインタの $\mathbf{D}^I_{sp}$の tail を書き換えるだけで，それより左のセルはケース 1 とケース 2 でやはり共有できる[¶]．

「諸悪の根源」と言われる “破壊的代入” も，経験あるプログラマにとっては，C 言語等 (Java も含めて) 手続き型言語に許された，確かに最も便利な言語機能の一つである．他方，関数型／論理型言語の “単一代入規則” は，効率を第一義とするプログラマにとっては，不自由な手枷となることがあるのも事実である．

すでに，並列計算環境下では台数効果により効

¶ KLIC 版 MGTP では，処理系による暗黙の複製を利用せざるをえない．

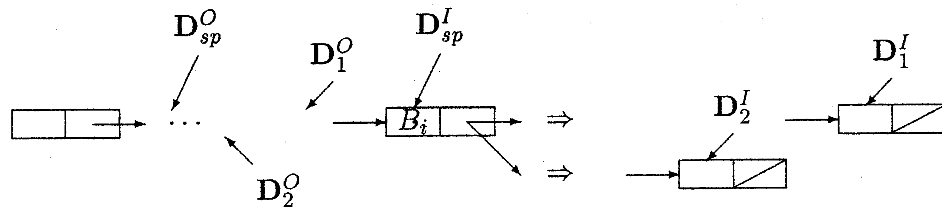


図 5　モデル拡張候補の分岐

率的な並列実行が確認されている [3]．本論文では，並列実行に関する問題には深く立ち入らない．むしろ，逐次実行を前提として如何に効率よい実装が可能かを追求する．場合分け (分岐) 推論を逐次化するために，スタックを導入する．ケース $\mathbf{M} \cup \{B_i\}$ の実行前にスタックに積むべき情報は，(1) 残りのケース $(B_{i+1}, \ldots, B_m)$，(2) 分岐前のモデル候補 $\mathbf{M}$，(3) 同モデル拡張候補 $\mathbf{D}(\mathbf{D}^I_{sp}, \mathbf{D}^O_{sp})$ である．

3.3　性能評価と改善

定理証明のベンチマーク集 TPTP [20] からいくつか問題を選び，Java 版 MGTP とオリジナルの KLIC 版 MGTP(項メモリ不使用) の実行性能を比較評価した．結果を表 1に示す．表中 jsat は Java 版 MGTP を SATCHMO [17] 制御としたものである．JIT(Just In Time compiler) つき Java 版 MGTP では，走行環境が若干有利であることにもよるが，KLIC 版に比べ実質的に最大 10 倍強の高速化を達成している．同じ走行環境でも，Java 版 MGTP(JIT なし) は，ほとんどの問題で概ね KLIC 版 MGTP と同等以上の速度で解くことができている．

各プログラム部分の実行比率は扱う問題によってかなり大幅に異なる．とりわけ，連言照合と包摂テストの比率は問題ごとの差異が大きい．SYN006-1 や SYN009-1 は 1 本のノンホーン節のみでモデル拡張が行われ，分岐ばかりが生じる極端な問題であると同時に，他の問題に比べ，包摂テストの占める比率が大きい．Java 版では包摂テストでもユーザ定義の match が実行されるのに対し，KLIC 版では処理系に組み込み (C コード) の "頭部単一化" が利用できる．こうしたことから，この 2 例では Java 版 (JIT なし) が KLIC 版より 2〜2.5 倍遅くなっている．

SATCHMO 制御ではモデルを拡張するアトムの選択順序が異なるため，一般に得られる証明も異なる．この順序を変えることが証明の探索制御に相当するが，SATCHMO では原理的にこの順序が固定で，探索制御が事実上不可能である．そのような場合，節の順番を変えてみるくらいの手立てしかない．MERC 法に基づく MGTP ではこの問題がなく，探索制御の後付けが容易で，さまざまな戦略／戦術の適用にも柔軟に対処できるのが一つの特長となっている．

Java 版 MGTP の改善を進めるにあたり，

- 実行回数カウンタや計時用タイマーをプログラム中に挿入して各部の実行状況を観測し，
- 最も時間を要する箇所を特定して，改良を施す，

ことを徹底した．アムダールの法則に基づいて，処理コストの最大比率を占める箇所を改善することが最大の効果をもたらすからである．ただし，ある設計変更が注目箇所の最適化に有効でも，他の場所には逆効果をもたらす場合があり，そのような場合には，適切なトレードオフが必要となる．

4　証明の依存性解析による冗長探索の削除

モデル生成法の証明は，公理 (正節) から始まり，推論規則 (混合節) を適用して次々と定理 (アトム) を生成していく過程を，目標とする定理 (負節) が得られるまで続ける，ボトムアップ実行に基づいているとみなすことができる．しかし純粋なモデル生成法は，証明には無関連な推論を行ったり，証明の分岐後に同じ証明を重複して行ってしまう，という効率上の問題点を原理的に抱えている．

本章ではこの問題点を解決するために，証明に無関連な推論の除去と補題による重複証明を回避するための機構をモデル生成法に組込む手法を提案する．前者は証明濃縮 (proof condensation) [19]，後者は畳込み (folding-up) 法 [16] という名でタブロ

表 1 性能評価

問題	節数 (+doms)	証明木 枝数 (jsat)	証明木 節点数 (jsat)	証明時間 (msec) KLIC†	jsat†*	java†*	java‡
PUZ012-1	18 (+6)	165 (165)	832 (1,060)	240	2,560	269	79
PUZ025-1	24 (+9)	90 (138)	310 (395)	370	2,067	323	79
PUZ030-1	43 (+5)	67 (80)	189 (259)	60	205	73	47
PUZ030-2	63 (+0)	153 (87)	314 (189)	130	94	116	32
SYN001-1.006	64 (+0)	720 (720)	1,956 (1,956)	2,090	4,532	1,872	266
SYN002-1.010	2 (+2)	354 (354)	1,234 (1,234)	390	2,525	363	63
SYN006-1	7 (+3)	96 (−)	532 (−)	**70**	時間切	191	63
SYN009-1	7 (+0)	19,683 (19,683)	29,526 (36,088)	**850**	11,527	2,298	344
SYN015-2	26 (+6)	196 (−)	2,557 (−)	8,530	時間切	5,362	797
SYN036-3	36 (+8)	516 (−)	8,323 (−)	6,580	時間切	7,427	1,094

† Sun Ultra1 Model170, * JDK1.1.3 without JIT　　制限時間：10 分

‡ PentiumPro 200MHz, JDK1.1.4 with JIT

法に基づく証明法の冗長性削除手法として知られる機構である．

これら二つの削除手法を実現するためにモデル生成法に導入する計算機構は,「証明の依存関係の解析」である．解析は，証明が完了した部分 (部分証明) に対して行い，これを基に，証明が未了の部分の枝刈りを行う．直観的にいえば，依存関係の解析により，(1) 各推論過程が証明に寄与したかどうかの判定と (2) 部分証明からの補題の抽出，が行える．そして，(1) により証明に無関連な推論を，(2) により重複証明を削除する．

C1: $true \rightarrow r.$　C5: $p \rightarrow false.$
C2: $r \rightarrow a \vee b.$　C6: $q \rightarrow false.$
C3: $r \rightarrow p \vee c \vee d.$
C4: $r \rightarrow p \vee q.$

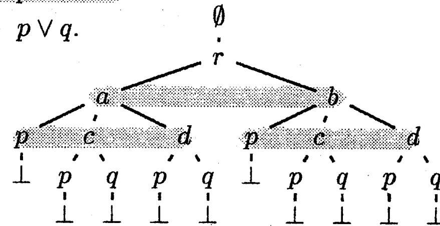


図 6 証明に無関連なモデル拡張

4.1 冗長な証明探索

本章で考察の対象とする冗長な証明探索は次の二つである．

4.1.1 証明に無関連なモデル拡張

証明に無関連なモデル拡張の例を図 6に示す．上が入力節集合で，下がその一つの証明木である．$C1$ でモデル拡張を行った時点で，$C2$，$C3$，$C4$ の三つの節が違反節となる．違反節をこの順に用いてモデル拡張をなうと右側のような証明木が選られる．この証明では，$C2$ と $C3$ によるモデル拡張 (図中網掛け部分) は証明に寄与していない．というのは，$C1$ の次の $C2$ と $C3$ によるモデル拡張は不要で，$C1$ の次に $C4$ でモデル拡張を行えば，直ちに証明が終了するからである．

モデル生成法では，モデル拡張は任意の違反節を用いて行うことができる．複数の違反節があった場合の選択基準はないので，この例のように運悪くゴールに関連しない違反節を選択してしまうと，証明を得るのに遠回りしてしまったり，場合によっては証明すら得られないこともある．

C1: $true \rightarrow t \vee p.$
C2: $p \rightarrow q \vee s.$
C3: $q \rightarrow r.$
C4: $s \rightarrow r.$
C5: $t \rightarrow p.$
C6: $p \wedge r \rightarrow false.$

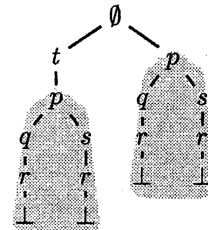


図 7 証明分岐後の重複証明

4.1.2 証明分岐による重複証明

重複証明の例を図 7に示す．左側が入力節集合で，右側がその証明木である．$C1$ でのモデル拡張により証明が分岐するが，左側と右側の部分証明の網掛け部分は全く同じ証明である．このように証明が分岐すると，それぞれの分枝の証明は全く独立に行われるため，各分枝で同一の証明を重複して行ってしまうことがある．

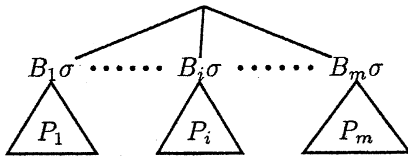


図 8　部分証明木

4.2　依存性解析による冗長性の削除

証明の依存性とは直観的には,「この定理は，この定理とこの定理を用いて証明された」といった定理の因果関係のことである．証明中にこの因果関係を書き留めておけば，証明が終了した後,「実は，この場合分けは証明に不要であった」とか「これだけの条件がそろえば，この定理が成り立つ」といったことを明らかにできる．

前者の場合，不要と判明するのが，すべての場合を尽くした後であれば，後の祭りであるが，そうではなく，一部分の証明の後に判明すれば，残りの証明を削除できる．後者の場合は，補題を生成し，補題の適用によって重複した証明を取り除くことができる．

以上のことをモデル生成法に適用するために，証明に寄与したアトム (関連アトム) の集合の計算を行う．ここで，関連アトムの集合は次のように定義される．

定義 1 (関連アトムの集合)　証明木 P に対して，関連アトムの集合 $Rel(P)$ は，以下のように定義される．

1. $P = \perp$ であり，$A_1\sigma \wedge \ldots \wedge A_n\sigma \rightarrow false$ でモデル棄却している場合 : $Rel(P) = \{A_1\sigma, \ldots, A_n\sigma\}$
2. $P = \top$である場合，$Rel(P) = \emptyset$.
3. P が図 8で示すような部分証明木であり，$A_1\sigma \wedge \ldots \wedge A_n\sigma \rightarrow B_1\sigma \vee \ldots \vee B_m\sigma$ でモデル拡張している場合 :
   (a) $\exists i(1 \leq i \leq m)B_i\sigma \notin Rel(P_i)$ の場合 : $Rel(P) = Rel(P_i)$
   (但し i_0は，$B_{i_0}\sigma \notin Rel(P_{i_0})$ を満たす．)
   (b) $\forall i(1 \leq i \leq m)B_i\sigma \in Rel(P_i)$ の場合，
   $Rel(P) = \cup_{i=1}^{m}(Rel(P_i)\setminus\{B_i\sigma\})\cup \{A_1\sigma, \ldots, A_n\sigma\}$

□

部分証明 P が$\top$を含まなければ，$Rel(P)$ は P を作るのに寄与したリテラルの内，証明木中 P より上方に現れるリテラルの集合となる．一方 P がモデルを見つけたことを示す$\top$を含む場合には，$Rel(P)$ は空集合$\emptyset$ になる．逆に $Rel(P) = \emptyset$ であれば，P は$\top$を含む．つまり，Pが$\top$を含むことと，$Rel(P) = \emptyset$ であることは同値である．

証明に関連するモデル拡張は，関連リテラルを用いて次のように定義される．

4.2.1　証明濃縮 : 証明に関連しないモデル拡張の削除

関連アトムが求まれば，モデル拡張が証明に関連しているかどうかの判定が直ちに行える．

定義 2 (証明に関連したモデル拡張)　違反節 $A_1\sigma \wedge \ldots \wedge A_n\sigma \rightarrow B_1\sigma \vee \ldots \vee B_m\sigma$ によるモデル拡張が証明に関連しているとは，このモデル拡張を根とする図 8で示す部分証明木において，$\forall i(1 \leq i \leq m)B_i\sigma \in Rel(P_i)$ であることをいう．□

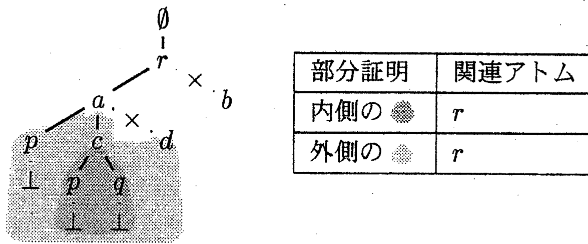
証明に関連していないモデル拡張は，削除することができる．この削除が健全であるのは次の理由による．いま，違反節 $A_1\sigma \wedge \ldots \wedge A_n\sigma \rightarrow B_1\sigma \vee \ldots \vee B_m\sigma$によるモデル拡張が証明に関連していないものとする．そうするとこのモデル拡張を根とする部分証明木 (図 8) を P_{i_0}(i_0は，$B_{i_0}\sigma \notin Rel(P_{i_0})$ を満たす) で置き換えた図形も，部分証明木となる．

さて，モデル拡張の後の各分枝の部分証明 $P_i(1 \leq i \leq m)$ が左から順 (昇順) に行われるとする．部分証明が終わるたびに $B_i\sigma \in Rel(P_i)$ かどうかの検査を行い，もし，$B_i\sigma \notin Rel(P_i)$ であれば，残りの部分証明を行う必要はなくなり，冗長な探索を削除できる．このような冗長性削除手法を**証明濃縮**と呼ぶ．

図 9に，図 6で示した証明から冗長な探索を削除した例を示す．内側の網掛け部分の証明に関連しているアトム集合は $\{r\}$ であり，これに c は含まれていない．これより，$C3$ によるモデル拡張はゴールに関連していないことがわかり，d の下の証明を削除できる．外側の網掛け部分でも同様のことがいえて，b の下の証明を削除できる．

4.2.2　畳込み法 : 補題生成による重複証明の回避

いま，ある部分証明 P から関連アトムの集合 $Rel(P)$ が計算されたものとする．そして別の部分の証明途中で，モデル候補 Mc に対して $Rel(P) \subset Mc$ であることが判明すれば，その部分の証明は，それ以上行なう必要はなく，完了してよ



部分証明	関連アトム
内側の	r
外側の	r

図 9　証明に無関連なモデル拡張の削除

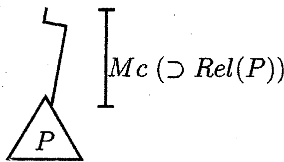


図 10　部分証明木の接木

い．なぜなら，Mc の下に P をつなげた図形が証明木となるからである (図 10).

つまり，入力節集合を S とすれば，$S \cup Rel(Ps)$ は，(S の充足可能性に関わらず) 充足不能である．このことを $S \cup Rel(P) \vdash \bot$ もしくは，$Rel(P) \vdash_S \bot$ と表記することにする．補題は，$Rel(P)$ から次のようにして作られる．

定義 3 (単位補題) $Rel(P) = \{R_1, \ldots, R_n\}$ であるとき，$Rel(P) \setminus \{R_i\} \vdash_S \neg R_i (1 \leq i \leq n)$ を $Rel(P)$ から得られる単位補題 *(unit lemma)* という．また，$Rel(P) \setminus \{R_i\}$ をこの補題が適用できる文脈と呼ぶ． □

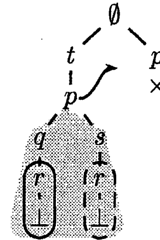
単位補題は，証明手続き (図 1) において次のように利用される．まず，何らかの方法でモデル候補 Mc が補題の文脈を満たす ($Mc \supset Rel(P) \setminus \{R_i\}$) ことが分かっているものとする．そして，モデル拡張時のある後件アトム $B_j\sigma$ について，$B_j\sigma = R_i$ であるとき，$B_j\sigma$ による部分証明 $MG(Mc \cup \{B_j\sigma\}, S)$ を行なわない．

上述のように補題の適用にあたっては，モデル候補が補題の前条件を満たすかどうかの判定が必要となる．これには集合の包含関係の判定が必要となり，計算コストがかかる．そこで，本論文では，$Rel(P)$ から得られる n 個の単位補題の内，次のような高々一つの単位補題を利用する．

定義 4 (抽出される単位補題) 図 8で示すような証明木において，部分証明木 $P_i (1 \leq i \leq m)$ から抽出される単位補題は，

$$\begin{cases} Rel(P_i) \setminus \{B_i\sigma\} \vdash_S \neg B_i\sigma \\ \qquad\qquad (B_i\sigma \in Rel(P_i) \text{ のとき}) \\ \text{抽出しない} \ (B_i\sigma \notin Rel(P_i) \text{ のとき}) \end{cases}$$

である． □



証明	関連アトム	単位補題
	p, q	$\{p\} \vdash_S \neg q$
	p, s	$\{p\} \vdash_S \neg s$
	p	$\emptyset \vdash_S \neg p$

図 11　補題生成による重複証明の削除

この補題の文脈は，P_i の兄弟証明のモデル候補に含まれる．つまり，この補題は P_i の兄弟証明で利用することが (文脈を検査しなくても) できる．さらにこの補題は，前条件が成り立たなくなる祖先まで持ち上げることができ，その祖先の子証明 (叔父や従叔父など) でも利用できる．このような単位補題を用いた冗長探索の削除手法を**畳込み法**と呼ぶ．

図 11に，図 7で示した重複証明を削除した例を示す．図中の表は各部分証明の関連アトムと抽出される補題を示す．外側の枠で囲んである部分証明 (網掛け部分) からは単位補題 $\emptyset \vdash \neg p$ が抽出される．そして，右側の p にこの補題を適用し p から下の探索を刈り込むことができる．

さて，容易に推察できるように，$Rel(P)$ からは単位補題以外にも多くの補題を得ることができる．例えば，$Rel(P) \setminus \{R_i, R_j\} \vdash_S \neg R_i \vee \neg R_j (1 \leq i, j \leq n \wedge i \neq j)$ という補題も考えられる．このような補題は，$Rel(P) \vdash_S \bot$ も補題と考えれば，2^n 個ある．本論文では，単位補題以外の補題は扱わない．というのは，これらの補題によって逆に探索空間が広がる恐れもあるからである [16].

4.3　実験結果

証明濃縮と畳込み法を取り入れたモデル生成法の評価を行うため，TPTP 問題ライブラリ (第 2.2.1 版) [21] からノンホーン問題を選び，実験を行った

||. 証明濃縮と畳込み法のいずれも，証明に分岐を生ずる問題にのみ有効な手法であり，ホーン問題では証明に分岐が生じないので，ホーン問題は実験の対象からはずした．計測には Sun Ultra-10/333 (メモリ 128MB) を用い，実行の制限時間は 10 分とした．

表 2にノンホーン問題 (全 1984 題) の内，制限時間内に解けた問題数を示す．表中，+は対応する手法を用いたことを，-は用いなかったことを表す．したがって，第 (1) 列は，何の手法も用いない通常のモデル生成法で解けた問題数を表している．括弧の中の数値は，通常のモデル生成法で解けた問題数を 100 としたときの各列の問題数を表している．

いずれの手法でも冗長探索の削除効果が現れている．畳込み法のみ (第 (2) 列) では，3%ほど解ける問題数が増加しているのに対し，これに先行棄却を加えた第 (3) 列では，23%増加している．また，畳込み法に証明濃縮を加えた第 (5) 列では，22%増加しており，畳込み法と証明濃縮の相乗効果が現れている．

一方，第 (3) 列と第 (6) 列 (もしくは，第 (5) 列と第 (6) 列) を比べてみるとそれらの差はそれほどではない．つまり，畳込み法+先行棄却に証明濃縮 (もしくは，証明濃縮+畳込み法に先行棄却) を加えても効果はあまり見られない．実際，個々の計測結果をみても，畳込み法に先行棄却 (もしくは，証明濃縮) を加えると効果が現れるが，畳込み法+先行棄却に証明濃縮 (もしくは，証明濃縮+畳込み法に先行棄却) を加えても効果はそれほどでもない，という傾向がみてとれる．

これは，証明濃縮と先行棄却の相乗効果は現れにくいことを示している．この理由は，先行棄却が証明濃縮の機能，つまり証明に無関連なモデル拡張を回避する機能を有しているためと考えられる．というのは，先行棄却が適用できる違反節によるモデル拡張は，証明に関連しているとみなすことができるからである．

表 3に，いくつかの問題に対する計測結果を示す．全て充足不可能な問題である．第 (7) 列に，定理証明システム Otter [18] による計測結果も示す．Otter は導出法の支持集合戦略に基づく定理証明システムで，世界的にもよく知られており，ここでは定理証明システムの性能の基準として示した．上段は証明時間，下段は証明木の節点数 (モデル生成法の場合) もしくは証明終了時に保持していた節数 (Otter の場合) である．下段の数値は探索空間の大きさの目安となる．

GRP124-8.004，PRV009-1，PUZ010-1，SET002-1 のように，通常のモデル生成法で証明できる問題でも，証明濃縮と畳込み法の効果が顕著である．実際，多くの問題がこのような傾向を示した．このことは，通常のモデル生成法による証明の多くは，冗長性を含んでいることを示している．

COM002-2，PRV009-1，TOP004-2 では，証明濃縮と畳込み法によって Otter 並みの証明性能が得られるようになった．また，CIV008-1.002，KRS013-1，SET002-1，SYN447-1，SYN458-1，SYN479-1 では Otter を凌駕する性能を得ることができた．

SYN458-1 では第 (5) 列では証明が 1 分半程で終了しているのに，先行棄却を加えた第 (6) 列では，10 分たっても証明が得られていないのは，先行棄却のねらいに反し奇異に感じられるかもしれない．このような結果は，この他にもいくつかあり，表中では PUZ010-1 が先行棄却を加えると証明木の大きさが倍近くになっている．先行棄却を行うと，行わない場合より早めに部分証明は完了する．しかし，一般にこの部分証明の関連アトムの集合は，先行棄却を行う場合と行わない場合では異なってくる．したがって，あるモデル拡張が証明に関連しているかどうかの判定も異なってくる可能性がある．また，関連アトム集合から作られる単位補題も異なるものが得られうる．このような違いは，どちらが良いとは一般的にはいえないが，SYN458-1 や PUZ010-1 では，先行棄却なしの方にうまく働き，証明が得られたものと思われる．

|| 図 1の手続きに畳込み法を適用する場合，

(あ) モデル候補 Mc に違反する節でモデル拡張を行なった後，各モデル分岐が補題によって棄却されるかどうかを判定する．

(い) モデル候補 Mc に違反する節の内，全ての分岐が補題によって棄却される節を優先的に選択する．

の二通りで評価を行なった．表中，先行棄却の欄が-となっているのは (あ)，+ となっているのは (い) の方法を適用したことを表している．

表 2　証明できた問題数 (1984 題中)

	(1)	(2)	(3)	(4)	(5)	(6)
証明濃縮	-	-	-	+	+	+
畳込み法	-	+	+	-	+	+
先行棄却	-	-	+	-	-	+
	310 (100)	320 (103)	380 (123)	339 (109)	379 (122)	381 (123)

表 3　各手法の比較

	(1)	(2)	(3)	(4)	(5)	(6)	(7)
証明濃縮 畳込み法 先行棄却	- - -	- + -	- + +	+ - -	+ + -	+ + +	Ot- ter
CIV008-1 .002	時間切	1310 2147	190 216	180 196	180 196	180 196	42720 3214
COM002-2	時間切	時間切	310 116	290 123	270 104	300 104	190 57
GRP124-8 .004	236500 851448	40300 72768	1410 2983	3770 9949	940 1738	1190 2442	446890 6219
KRS013-1	時間切	時間切	30 121	30 101	20 76	20 76	196140 5431
PRV009-1	11950 136289	50 407	10 53	10 41	10 39	10 39	110 40
PUZ010-1	98140 310722	13460 23173	10030 16512	32670 70466	5500 9916	9300 15022	時間切
PUZ018-1	時間切	時間切	7020 690	31770 3277	2060 233	3030 263	660 164
SET002-1	140850 315188	590 1046	0 30	50 124	10 27	0 19	9780 948
SYN447-1	時間切	時間切	111040 21854	時間切	102240 27474	106580 19069	時間切
SYN458-1	時間切	時間切	時間切	時間切	95300 9614	時間切	時間切
SYN479-1	時間切	時間切	410200 15646	時間切	時間切	373740 11042	時間切
TOP004-2	時間切	時間切	190 38	190 32	190 32	190 32	90 31

上段：証明時間 (ミリ秒)
下段：
証明木の節点数
(モデル生成法)
最終的に保持した節数
(Otter)

5　論証支援システムへの応用

本章では，ルール間優先機能や仮説推論といった法的推論において必要とされる推論機構の MGTP 上での実現法について述べる．

5.1　拡張論理型言語

本節では，法的な知識の表現言語について述べる．

表現言語としては，拡張論理型言語を採用しており，法律知識，判例等の知識は以下のようなルール ID，ヘッド，ボディからなるルール形式として表現される．

例 1

$r1(X) :: fly(agt = X) \Leftarrow$
$\quad bird(agt = X), not\ baby(agt = X).$
$r2(X) :: -fly(agt = X) \Leftarrow$
$\quad penguin(agt = X).$
$f1 :: bird(agt = pingu).$
$f2 :: penguin(agt = pingu).$
$f3 :: baby(agt = pingu).$ □

本システムで採用するルールは形式上，以下の三つに分かれる．

(1) $R :: L_0 \leftarrow L_1, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n.$
(2) $R :: L_0 \Leftarrow L_1, .., L_m, not\ L_{m+1}, \ldots, not\ L_n.$
(3) $R :: \leftarrow L_1, L_2.$

R はルール識別子で，ルール中に現れる全変数を引数として持つ関数で関数名は各ルールに固有である．$L_i (0 \leq i \leq n)$ はリテラル (正負) を表し，*not* は失敗による否定を表している．(1) は exact ルールと呼ばれ，ルールボディが全て真のときルールヘッドも必ず真となるルールである．(2) はデフォルトルールで，ルールボディが全て真のとき，矛盾を引き起こしたり一貫性制約を破ることがないならばルールヘッドも真となるルールである．また (3) は一貫性制約と呼ばれ，L_1と L_2が同時に真となることはないという制約を表す．

本言語の特徴は，このように通常の否定の他に，失敗による否定が表現できること，ならびにデフォルトルールが記述できることである．デフォルトルールの解釈は一意ではないが，直観的には，ヘッド (帰結) と矛盾する情報が陽に証明されていないことを前提としている．本システムにおけるデフォルトルールの解釈は，5.2節にて説明する．

5.2 論証の自動生成

法的推論を実現するにあたって必要とされる推論機構の導入について述べる．

5.2.1 ルール優先の形式化

法律解釈は，一意ではなく，しばしば相矛盾するルールが記述される．こうした不確実な知識を取り扱うために，ルール間の優先関係が必要となる．

5.2.1.1 意味論

優先関係を持つルール集合に適用可能な意味論は，サーカムスクリプションに述語優先関係を導入したもの，デフォルト理論にルール優先関係 を導入したもの，論理プログラムにルール優先関係またはリテラル優先関係を導入したもの ([10] ほか)，ルール優先関係を持つデフォルトルール集合に対して独自の意味論を与えるもの ([15] ほか) など数多くのものが提案されているが，本システムでは，実装が容易で法的推論に適しているものとして，[15] の方法を選択し，そこで提案されているルール集合の拡張論理プログラムへの変換方法を応用した．

5.2.1.2 拡張論理プログラムへの変換方法

デフォルトルール

$R1 :: L_0^1 \Leftarrow L_1^1, \ldots, L_m^1, not\ L_{m+1}^1, \ldots, not\ L_n^1.$

に対して条件 1を満たすようなデフォルトルール

$R2 :: L_0^2 \Leftarrow L_1^2, \ldots, L_k^2, not\ L_{k+1}^2, \ldots, not\ L_q^2.$

が存在するとき，$R1$ を

$$L_0^1 \leftarrow L_1^1, \ldots, L_m^1, not\ L_{m+1}^1, \ldots, not\ L_n^1,\ not\ defeated(R1).$$

に変換し，

$$defeated(R2\theta) \leftarrow L_1^1\theta, \ldots, L_m^1\theta, not\ L_{m+1}^1\theta, \ldots, not\ L_n^1\theta,\ not\ defeated(R1\theta), L_1^2\theta, \ldots, L_k^2\theta,\ not\ L_{k+1}^2\theta, \ldots, not\ L_q^2\theta, not\ R1\theta < R2\theta.$$

を追加する．ただし，θは次の条件 1を満たす最大単一化子とする．

条件 1 $L_0^1\theta = \neg L_0^2\theta$ となるような単一化子θが存在するか，またはある一貫性制約 $\leftarrow L_1, L_2$ に対して，$L_1\theta = L_0^1\theta$ かつ $L_2\theta = L_0^2\theta$ または $L_2\theta = L_0^1\theta$ かつ $L_1\theta = L_0^2\theta$ となるような単一化子θが存在する． □

上記により導入されるルール間の優先関係は，法令・法解釈の強弱関係や例外を表現するのに用いられる．

例 2 (法令の強弱関係) 法令の強弱関係とは，後法/先法，上位法/下位法，特別法/一般法を基準としたものであり，例えば，隔地者間の承諾の効力発生時期を規定した民法 *526* 条 *1* 項と隔地者間の意思表示の効力発生時期を規定した民法 *97* 条 *1* 項とは，特別法/ 一般法の関係にある． □

例 3 (法解釈の強弱関係の例) 民法 *541* 条によれば，継続的賃貸契約において賃借人が履行遅滞し，賃貸人への信頼関係破壊が著しい場合に賃貸人はその契約を解除することができるが，転借がある場合には転借人への催告を必要とする説と不要とする説とがあり，判例は不要説を支持している． □

例 4 (例外の例) 統一売買法 *19* 条 *1* 項では反対申込みを規定し，統一売買法 *19* 条 *2* 項ではその例外

を規定している. □

5.2.2 メタ推論技法

前節の変換処理により，ルール優先情報を含むデフォールトルールは，失敗による否定 (NAF) を含んだルール形式へ変換された．NAF のモデル生成型推論系での処理は，[14] に基づいて行われる．

5.2.2.1 様相オペレータ導入

ルール内のNAF リテラルに対して様相オペレータである K を導入する．K オペレータの導入は，$A_l \leftarrow A_{l+1}, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n$ なる問題節を，以下のように変換する処理である．

$$A_{l+1} \wedge \ldots \wedge A_m \rightarrow (-KA_{m+1} \wedge \ldots \wedge -KA_n \wedge A_l) \vee KA_{m+1} \vee \ldots \vee KA_n$$

加えて，K に対する一貫性制約を追加することにより，NAF を含む問題節の安定モデルを導出することが可能である．一貫性制約は，たとえば P と $-KP$ が両立できないことを表すルールとして $P, -KP \rightarrow$. のように表される．

5.2.2.2 安定モデルの抽出

定理証明系により出力されたモデルは，導出可能なすべての安定モデルを含んでいるが，仮説のみから構成されるようなモデルも含んでいる．安定モデルのみを抽出するためには，以下の条件をチェックする必要がある．

条件 2 (T-Condition) モデル M において，$KP \in M$ なるすべての P に対して，$P \in M$ が成り立つ． □

この処理により，得られるモデルが安定モデルであることが保証される．生成された論証として出力するのは，得られた安定モデルに含まれるゴールを導く証明過程である．このとき，ゴールを導く証明過程がすべてのモデルに対して共通して現れているとき，その論証を justified といい，そうでない論証を plausible という．ここで，justified, plausible はそれぞれ，必ず勝てる論証，攻撃され得る論証という意味である．攻撃され得る論証とは，今後，新しい知識が導入されることで前提が覆される可能性のある論証を意味する．

6 おわりに

Java 言語によるモデル生成型定理証明系 MGTP の実装とモデル生成法における冗長な推論の削除法，MGTP を利用した法的推論について述べた．

Java 版 MGTP の項の設計では，抽象クラス Term に定数項，変数項，複合項の各サブクラスのフィールド変数の (共通集合ではなく) 和集合をもたせた．クラス階層として奇異に見える反面，実行効率の向上が確実に得られて有益であった．ノンホーン節に対して実行を逐次化し，情報の複製を一切避けるようにした点も高効率化に貢献している．

Java では，記憶の動的割り付けと解放の機能が組み込まれており，プログラマが明示的に書く必要がない．これにより，記号処理プログラムに必須のリスト処理が自在に記述できる．また，Java はいわゆる "厳格な型づけ" に従っている．コンパイラで検出される型誤りを正していくだけで，プログラムのバグの大半が除去される，という印象があり，これは "型づけの弱い" Lisp や Prolog に優る特長かもしれない．

以上のような経験から，Java はネットワーク応用等のみならず，記号処理を基本とする知識処理プログラミングを行う上でも効果的な言語であると考えられる．

証明濃縮は証明に無関連なモデル拡張を削除し，畳込み法は証明分岐後の重複証明を回避する手法である．二つの手法で削除される冗長性は異なるが，どちらも「関連アトムを計算する機構」をモデル生成法に組込むことによって実現される．そして，これらの効果を実験により確かめた．この実現法の原理は簡潔であるが，その効果は大きい．また，本実現法はタブロ法に基づく定理証明システムにも適用可能である．

法的推論推論の知識表現言語として採用した優先度付き拡張論理型言語は，法律という領域に関していえば，不完全な知識や矛盾する知識を記述することができるため記述能力は極めて高いことが実証されている．

Java 版 MGTP の機能として，(1) 整数，実数，その他のデータ型の導入，(2) 入力節から Java コードを呼び出す機構の導入，がすでに実装済みである．また最近，弁別木による高速なインデクシング機構や構造体の共有化を導入する事により，さらなる高速化に成功している [5].

最近，仮説推論や故障診断といった応用の面から極小モデルの効率的な生成法 [11] が注目されている．Java 版 MGTP を極小モデル生成用に改良す

る試みも始まっている [6]．また，証明濃縮と畳込み法は，モデル棄却の観点からの枝刈り手法なので，極小モデル生成にはそのままでは適用できない．今後，これらの手法と類似の手法が極小モデル生成に適用できないか考察していく予定である．

参考文献

[1] 舘林 剛史，越村 三幸，長谷川 隆三：CMGTPへの事後関連性検査と畳込み機構の組込み，九州大学大学院システム情報科学研究科報告，Vol.4，No.2 (1999)，pp.151–158.

[2] 長谷川 隆三，藤田 博：MGTP: 並列論理型言語 KL1 によるモデル生成型定理証明系，情報処理学会論文誌，Vol.37，No.1 (1996)，pp.1–12.

[3] 長谷川隆三，藤田 博，中田健浩，力 規晃：並列プログラムのＮ逐次実行方式と MGTP への適用，九州大学大学院システム情報科学研究科報告，Vol.2，No.2 (1997)，pp.241–246.

[4] 長谷川隆三，新田 克己，白井 康之：定理証明技術を応用した高度論証支援システムの開発，論文集「高度情報化支援ソフトウェア育成事業編」，pp.59–66, 論文集情報処理振興事業協会 成果発表展示会 (1999)

[5] 長谷川 隆三，藤田 博：制約問題を解くためのモデル生成型定理証明系の新実装，九州大学大学院システム情報科学研究科報告，Vol.4，No.1 (1999)，pp.57–62.

[6] 長谷川 隆三，藤田 博，越村 三幸：分岐補題の導入による極小モデルの効率的生成法，九州大学大学院システム情報科学研究科報告，Vol.4，No.2 (1999)，pp.145–150.

[7] 藤田 博，長谷川 隆三：Java 言語によるモデル生成型定理証明系 MGTP の実装，九州大学大学院システム情報科学研究科報告，Vol.3，No.1 (1998)，pp.63–68.

[8] 淵 一博：MGTP(SATCHMO) の Java による実現，(1997) (私信).

[9] Ken Arnold and James Gosling: "The Java Programming Language," Addison-Wesley, (1996).

[10] G. Brewka: Well-founded semantics for extended logic programs with dynamic preference, *Journal of Artificial Intelligence Research*, 4, pp.19–36 (1996).

[11] F. Bry and A. Yahya: Minimal Model Generation with Positive Unit Hyper-Resolution Tableaux, *Proc. Fifth Int. Workshop, TABLEAUX'96*, LNAI 1071, pp.143–159 (1996).

[12] C.-L. Chang and C.-T. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press (1973).

[13] M. Fitting: *First-Order Logic and Automated Theorem Proving 2nd edition*, Springer-Verlag (1996).

[14] K. Inoue, M. Koshimura, R. Hasegawa: Embedding Negation as Failure into a Model Generation Theorem Prover, *Proc. of CADE-11*, LNAI 607, pp.400–415 (1992).

[15] R. Kowalski and F. Toni: Abstract Argumentation, *Artificial Intelligence and Law Journal*, 4(3-4), pp 275–296,1996.

[16] R. Letz, K. Mayr and C. Goller: Controlled Integration of the Cut Rule into Connection Tableau Calculi, *J. of Automated Reasoning*, Vol.13, pp.297–337 (1994).

[17] R. Manthey and F. Bry: SATCHMO: a theorem prover implemented in Prolog. *Proc. of the 9th Int. Conf. on Automated Deduction*, LNCS 310, pp.415–434 (1988).

[18] W. W. McCune: OTTER 3.0 Reference Manual and Guide, ANL-94/6, Argonne National Laboratory, USA (1994).

[19] F. Oppacher and E. Suen: HARP: A Tableau-Based Theorem Prover, *J. of Automated Reasoning*, Vol.4, pp.69–100 (1988).

[20] G. Sutcliffe, C. Suttner and T. Yemenis: "The TPTP Problem Library," *Proc. CADE-12*, pp.252–266 (1994).

[21] C. B. Suttner and G. Sutcliffe: The TPTP Problem Library(TPTP v2.2.0), *Technical Report 99/02*, Department of Computer Science, James Cook University, Australia, (1999).

[22] K. Ueda and T. Chikayama: Design of the Kernel Language for the Parallel Inference Machine, *The Computer Journal*, Vol.33, No.6, pp.494-500 (1990).

[23] A. Urquhart: The Complexity of Propositional Proofs, *The Bulletin of Symbolic Logic*, Vol.1, pp.425–467 (1995).