PROLOGプログラム設計に関する
一考察

(1)

PROLOGプログラム変換に関する論理学的研究

Logical Studies on Prolog Program Transformation

沢村　一

Hajime Sawamura

（株）富士通研究所
国際情報社会科学研究所

Fujitsu Laboratories Ltd.
International Institute for Advanced Study of Social Information Science

要　旨

非決定的なPrologプログラムをソースレベルで最適化するための技法を論理学的見地より考案し、それに基づいてオプティマイザを構築した。特に、述語（呼出し）の決定性が、Prologのような非決定性のプログラムの最適化変換を考える上で重要な概念となることを見い出し、述語（呼出し）の決定性の概念とその数学的性質を詳しく論じた。

変換プロセスは次の三つのフェイズを含んでいる：(1)最適化に必要なプログラム情報の抽出、(2)プログラムのインライン展開、(3)局所的最適化法の適用。(1)では、各述語は、直線型、終端再帰型、一般再帰型のいずれかに分類される。さらに、述語（呼出し）が決定的であるか否かが、我々の二種類の決定性の定義に従って決められる。(2)では、不必要なバックトラックを避けるためにカットの自動挿入が行われ、次いで、述語呼出しのその定義体によるインライン展開が行われる。これらの最適化法は手続き内と手続き間の最適化を達成する。(3)では、様々な局所的最適化技法が、インライン展開後のプログラムに対して適用され、節内における最適化が図られる。局所的最適化法としては次の技法が考案された: (a)部分ユニフィケーション、(b)命題的簡単化（多重連言、選言の除去、冗長述語の除去、不実行述語の除去、因子化）、(c)冗長変数の除去（等式による代入）、(d)等式ゴール列の統合化、(e)選言節の複数節への分解。

いくつかの典型的な例によるオプティマイザの性能評価実験は、我々のオプティマイザが平均20% 位の速度効率を、明らかに冗長と思われるプログラムに対しては40% ～60% の速度効率を達成することを示した。

論文の後半では、様々な論理を扱うことのできる汎用論証支援システムEUODHILOSの諸機能とその論理型プログラム変換への応用を論じた。さらに、Prologプログラム変換に関する我々の方法論を正当化するためになされた理論的考察の結果、及び他の研究との違いなどについても述べた。

ABSTRACT

A source-level optimizer for Prolog, a nondeterministic logic programming language, is designed and implemented for the purpose of Prolog program improvement. Especially, it is emphasized that the determinacy of a predicate (call) is a crucial concept in optimizing Prolog programs.

The design principles of the optimizer include the following three phases: (1) Extracting program information, (2) Expanding programs in-line, (3) Applying local optimization techniques. In the phase (1), each predicate is classified into any one of the three types: straight-line, tail-recursive, general recursive, and is decided whether it is deterministic or not, according to the two kinds of determinacy: a-determinacy and r-determinacy. In the phase (2), automatic cut insertion is done to avoid unnecessary backtracking, and then the predicates are expanded (partially evaluated) by the inline substitutions of their defining predicates to them. In the phase (3), various local optimization techniques are applied to the resultant predicates of the inline expansions. As the local optimization techniques, are established (a) partial unification, (b) propositional simplifications (deletion of multiple conjuncts and disjuncts, deletion of redundant "true" and "fail" predicates, deletion of unexecutable parts, factoring), (c) deletion of redundant variables by equality substitution, (d) integration of equational predicates, (e) resolution of disjunctive goals, etc.

The Prolog optimizer integrating these techniques shows approximately 20% time efficiency in average for some typical Prolog programs, and 40 - 60% time efficiency for obviously redundant programs.

In the latter half of the paper, EUODHILOS: a general reasoning system for a variety of logics is introduced. This is a generic logic environment which allows us to define our own logical system and to reason about it in the defined system. Then, its applicability to logic program transformation is discussed. Some theoretical considerations to justify our methodology for optimizing Prolog programs at the source level and discussions about related works are also added.

目　次

第１章　はじめに

プログラミング言語をより非手続き的にしようという試みはこれまで主として２種類の言語クラスを生んできた。一つは関数型の言語クラスであり、他の一つは関係型の言語クラスである。これらは、出発点として、何らかの論理系の論理式をプログラムの表現として用い、計算機構として論理の推論機構の一部を採用しているという意味において、総称的に論理型言語という名で呼ばれている。これらの言語によって、プログラマはプログラムの手続き的な詳細や効率面等にあまり関心を払うことなく、むしろ計算対象となるデータとその構造およびその上のオペレーションの間の論理的関係により焦点を当ててプログラミングを行うことが可能となってきた。その結果、そのような言語は、プログラムとそれが解こうとしている問題の関係を理解するのがより容易で、したがって、信頼性の高いプログラムの作成等に役立つというように主にソフトウェア工学上の諸問題に寄与するところが大きいものと期待されている。他方、その代償として、そのような言語のインタプリタやコンパイラに対しては、言語に固有のより効率的なインタプリメント技法や最適化法等が強く求められることになった。今日、Prologは、プログラムの表現形式を一階述語論理の部分クラスであるHorn論理に、そして計算メカニズムを導出原理によって実現した関係型のプログラミング言語の代表格として認められ、知識情報処理を支えるプログラミング言語として期待されている。それは、Prologのもつ記号処理上の数々のユニークな言語特性のためであることはもちろんであるが、Prologの効率的なインプリメント技法の進歩がそこに伴っていたことが大きな理由として上げられる。本論文では、Prologプログラムをソースレベルにおいて最適化変換する問題を論理学的見地より考察し、それに対する一つの有力なアプローチを与える。

プログラムの最適化とは、プログラムが与えられたとき、実行時に計算機資源をより有効に使用するよう変換することをいう。したがって，最適化(optimization)というより改良(improvement) といった方が正確であろう〔Allen 72〕。プログラムの最適化の目標、効果はソースからオブジェクトに至る言語階層の異なったレベルでそれぞれ異なっているものである。Prologソースレベル・オプティマイザはPrologプログラムを対象とし，ソースレベルにおいて改良のための変換を行う。プログラムをソースレベルにおいて最適化することの利点は、計算が部分的に進められること、およびプログラムテキストに明示的もしくは暗に潜んでいる冗長性が記号実行と言う意味で除去されることにある。したがって、このようなソースレベルでの最適化はプログラムの知的変換法の要素技術の一つとしても有用である。さらに、ソースレベルで得られた最適化手法は言語の効率的なコンパイレーションにも多くのヒントを与えることにもなる。

本論文は、Prologプログラムをソースレベルで最適化するために考え出されたいろいろな手法を詳細に述べている。それらの最適化手法は主として空間効率よりは時間効率を改良することに関わっている。Prologはそもそもその言語的性格からいって、ソースレベルでの最適化に適した言語である。なぜなら、それは一階述語論理の手続き的解釈より導かれた言語である一方、プログラムに対して入力と出力の関係としての宣言的な読み、言い換えると論理式としての読みを部分的に可能にしているからである。

ここでは、純Prologに制限することなく現実のProlog〔Bowen 81〕の最適化方法について述べる。そして、実際的に有用なオプティマイザの構成を提案する。実際、本論文のアプローチはPrologプログラム変換に関する現在の研究現状とはかなり異なっている（これまでのプログラムの最適化、変換論については文献〔

Partsch 83〕を参照。）大局的な特徴づけを、手法の効果の高低と手法の一般性の高低の２軸の中で行うならば、本論文における最適化方法は weak but general" であり、 strong but specialized"なPrologプログラム変換 (例えば、〔Hogger 81〕、〔Komorowski 82〕、〔Sato 84〕) とは対照的であると言える。

Prologプログラムは、ゴールと呼ばれる手続きの呼出し、及びそれを定義している手続きの集まりから構成されている。したがって、プログラムのソースレベルでの最適化手法としては、最適化の対象とするスコープによって、

(i) ゴールの定義体による展開、

(ii) 論理式としての節の変形、

(iii) 述語定義体単位の変形

などが考えられる。(i)は通常のプログラム言語ではサブルーチン展開とかインライン展開と呼ばれ、複数プロシージャ間での最適化手法の一つであり、計算の部分実行に相当する。(ii)は一つの節内に限って、冗長性の除去を行う局所的最適化手法である。(iii)は一つの述語定義体内での最適化に関するものである。これらの最適化手法には、いくつかの適用条件が必要になる。中でも述語呼出しが決定的であるか否かを検出する必要がある。また、Prologはバックトラックに基づく非決定性を、言語の大きな特徴としているにもかかわらず、それに起因する言語処理系の空間的・時間的負荷はかなり大きい。述語の決定性検出は、そのような負荷を軽減する最適化手法においても有用になるものである。このような理由から、本論文では述語（呼出し）の決定性の定義とその数学的性質を特に綿密に論じている。

本論文のPrologオプティマイザは次の三つの設計原理によって構成されている。

(1) 最適化のための情報抽出、

(2) インライン展開、

(3) 局所的最適化。

先ず，Prologプログラムの最適化に当たって、あらかじめプログラムテキストより最適化に必要な情報を抽出しておくと都合がよい。我々の目的のためには、それらはプログラムの型と決定性に関する情報である。ユーザプログラムの各述語は次のいずれかの型に分類される：直線型、終端再帰型、一般再帰型。そして述語の決定性なる概念（第２章で定義されるａ－決定性とｒ－決定性）に従って述語が決定的であるか否かが検出される。

次に、述語呼出しの決定性情報を用いて、不必要なバックトラックが起こり得る場所にカット記号が挿入される。これにより、知的バックトラックの簡単なケースが達成されることになる。その後、プログラムのインライン展開が行われる。一般に、インライン展開の主要目的は次の二つからなっている：

・サブルーチンリンクのオーバヘッドの除去、

・局所的最適化手法に、より大きなプログラム単位を与え最適化の機会を増すこと。

Prologのためのインライン展開は、述語呼出し（ゴール）をその定義節の選言項で置き換えるという方法で行われる。このとき、それはPrologプログラムの実行メカニズム”最左及び深さ優先”に従がわねばならない。インライン展開は一般にはプログラムのサイズをかなり大きくし、空間効率に問題を引き起こすように思われるが、ここではそのような側面には触れない。時間と空間の双方を考慮したインライン展開の方法は今後の課題としておく。

本論文ではさまざまな局所的最適化変換手法が提案されている。それらは初

期プログラムに独立に適用されることもあり、またインライン展開後のプログラムにも適用される。そのような局所的最適化変換手法には次のようなものがある。

(a) 部分ユニフィケーション、

(b) ゴール列の簡単化（多重連言・多重選言の除去、冗長述語 'true' 、'fail'の除去、不実行ゴール列の除去、共通ゴールのくくり出し)、

(c) 等式代入による冗長変数およびゴールの除去、

(d) 等式ゴール列の統合化、

(e) 節の分解。

本論文では、Prologプログラムのソースからソースへの変換の形式的な取扱いを与えることも、また変換が特定の意味論の下でプログラムの同値性を保存することの証明を与えることも意図していない。さらに、本論文はPrologプログラムの最適化をプログラムの複雑さの理論の観点から捉えることにも関わっていない。むしろ、本論文の目的は、Prologプログラムをソースレベルで最適化するさいに、自然で直観的な最適化手法の可能性を追求し、明らかにすることにある。言い方を換えると、本論文での関心は、Prologプログラムのソースレベルではどのような種類の最適化がなされ得るかについてのアイディアを提示することにある。そして、それらを組み込んだ一つのオプティマイザを試作した上で評価してみることにある。

本論文の構成は次のようになっている。第２章は最適化図式で用いられる記法とプログラムの型、決定性の定義を述べている。特に、述語（呼出し）の決定性については、その異なった定義法及びいくつかの性質が詳しく論じられている。第３章は各種の最適化図式とその適用条件を与えている。第４章はオプティマイザのインプリメンテーションを正当化するために有用な結果、議論を述べている。

また、決定性の非可解性に関する結果も含む。第５章はPrologで書かれたPrologオプティマイザの概要を記述している。第６章はいくつかの最適化の例とその時間評価を述べている。それによって本論文の方法の有効性が示される。第７章では，様々な論理を扱うことのできる汎用論証支援システムEUODHILOSの諸機能とさまざまな論理定義，証明例を取り上げる。そしてEUODHILOSの論理型プログラム変換への応用について議論する。第５章で述べたオプティマイザは変換規則が初めから固定され，拡張性・修正可能性に乏しいものであるが，EUODHILOSのような汎用システムは，論理型プログラムのオプティマイザをインクリメンタルに構成していくことを可能にする．第８章はPrologプログラム変換の最近の研究の現状を簡単に紹介している。ここで、我々の方法との定性的な相違点が明らかにされる。最後の章において本論文の主要な寄与をまとめ、今後の課題と展望を与える。

なお、本論文の内容は筆者の論文〔Sawamura 85a〕，〔Sawamura 85b〕，〔沢村 86a〕，〔沢村 86b〕，〔沢村 86c〕，〔沢村 86d〕，〔Sawamura 86e〕，〔沢村 79〕，〔沢村 81〕，〔沢村 82〕，〔Sawamura 85c〕，〔Sawamura 86f〕をもとにして書かれた。特に，２．４節は論文〔沢村 86a〕，〔沢村 86b〕に，第３章及び４．１～４．３節は論文〔Sawamura 85a〕に，４．４節は論文〔Sawamura 85b〕，〔Sawamura 86e〕に、第５、６章は論文〔沢村 86c〕に多くを負っている。第７章は，論文〔Sawamura 90〕，〔Sawamura 91〕，〔Sawamura 92〕，〔沢村 93〕に主に基づいている。論文〔沢村 79〕，〔沢村 81〕，〔沢村 82〕，〔Sawamura 85c〕，〔Sawamura 86f〕，〔Sawamura 83〕，〔沢村 86g〕は本書の将来課題、特にPrologプログラムの意味論と変換図式の同値性証明法等に関係し、それらに有力な方法を提供する可能性をもつものと言える（終章参照）。

第２章　記法と定義

本論文では、Prolog〔Bowen 81〕の構文法とその実行的意味に関する知識を仮定する。ここでは、以下で導入する最適化変換図式を記述するのに特有の記法と定義のみを述べる。

Prologプログラムは次の形式をした順序づけられた節の有限集合である。

H :- B1,・・・,Bn.　(n ≧ 0)

ここで、H は節のヘッド、各Biはゴールあるいは述語呼出しと呼ばれる。" B1,・・・, Bn "は節の本体と呼ばれる。ゴール (述語名) が与えられたとき、そのゴール (述語名) とヘッドが同じ名前と同じアリティ（引数の数）をもつ節の順序づけられた集合は、そのゴール (述語名) の述語定義体とか、そのゴール (述語名) の定義節と呼ばれる。例えば、次のプログラムにおいて、

```
p(A,B) :- Γ.
p(C) :- Δ.
p([D¦E], F) :- Θ.
q(G,F) :- Λ.
p(a,b).
```

ゴールp(X,d)の定義体はそれと同じ述語名と同じ引数のヘッドをもつ第１、第３、第５の節からなる。

以下、Prologの構文要素の上を動くメタ変数に対して区別された記号を用いることに注意されたい。

2.1 記号の使用法

【記号規約】

(1) (添字付) 文字 P, G, H はゴールあるいはヘッドを表す。(添字付) 文字 X, Y, Z はProlog変数を、(添字付) 文字 t, s はPrologの項を、文字 p, q, rは述語名あるいは命題を表す。

(2) (添字付) ギリシァ文字 Γ、Δ、Θはゴールのコンマで区切られた列を表す(空列も含む)。Γが空列のときには、それをゴール "true" と同一視する。(添字付) 文字A、B はPrologの項の非空な列を表す。

(3) Γのいずれかのゴールに変数X が出現するとき、そのようなゴール列を特にΓ(X) と表すことがある。このとき、Γ(t) はΓ(X) の中の変数X のすべての生起に項t を代入してできたゴールの列を表す。

(4) Φ, Ψはゴールあるいは節を縦に並べた列を表す。

(5) t をPrologの項とするとき、t'はt のすべての変数を名前換えしてできた項を表す。すなわち、“ ’ ”はリネーム・オペレータを表すものとする。

2.2 最適化変換図式

ゴールあるいは節の列が適当な述語定義体を用いて最適化された列に変換されることを図示するために、論理における導出可能性に対応する水平線を用いる。

【最適化変換図式】

最適化変換図式とは、次のような形式の図式をいう。

$$\frac{\Phi}{\Psi}$$

ここで、Φ、Ψをそれぞれ最適化変換図式の上式、下式と呼ぶ。上式は最適化前の式の列を、下式は変換後の式の列を表している。

2.3 述語定義体の型

ユーザのプログラムは直線型、終端再帰型、一般再帰型のいずれかに分類される。そして、この分類にしたがってインライン展開が行われる。

【述語定義体の型】

(1) 節 "H :- Γ. " は、Γの中にヘッドH と同じ述語名およびアリティの述語呼出しがないとき、直線節であるという。

(2) ゴールの述語定義体Φは、Φのすべての節が直線節であるとき、直線型

であるという。

(3) 述語名p の述語定義体Φを、

H1 :- Γ 1, P1.
:
Hi :- Γ i, Pi.
:
Hn :- Γ n, Pn.

とする、ここで、

(i) Γ 1, ・・・, Γ n には名前がp でそれと同じアリティをもつ述語呼出しは現れない。さらに、

(ii) P1, ・・・, Pnは("G1 -> G2" のような複合的なゴールではなく) アトミックなゴールで、それらの中には名前がp でそれと同じアリティをもつ述語呼出しが少なくとも一つ存在する。

このとき、述語定義体Φは終端再帰型であるという。

(実際のオプティマイザでは、条件 (ii) は弱められている。すなわち、Piは " G1; G2 " という形式であってもよい。ただし、そのときは節" Hi :- Γ i,G1. " 及び " Hi :- Γ i,G2. " は終端再帰でなければならない。)

(4) 述語定義体が直線型でもなく終端再帰型でもなければ、(一般) 再帰型であるという。

2.4 述語（述語呼出し、ゴール）の決定性

述語 (呼出し) の決定性は非決定性プログラムを最適化するさいに重要な役割を果たす。実際、インライン展開、カットの自動挿入、およびある種の局所的最適化手法においては、述語 (呼出し) の決定性は本質的な条件となっている。本論文で述べるPrologプログラムの最適化法に必要な決定性の概念を次のように定義する。

【述語呼出しの決定性】

述語呼出しが決定的であるとは、述語呼出しがそれの述語定義体を呼んだとき、述語定義体の高々一つの節で成功し、バックトラックしたとき、述語呼出しは決して成功することがないことを言う。より単純化して言えば、述語呼出しが決定的であるとは、それが高々一回成功することを言う。

上の定義は、次のようなゴールが決定的であることを含意することに注意されたい：

(i) 最初に呼ばれたとき停止しないゴール、

(ii) 最初の実行で成功し、バックトラックしたとき停止しなくなるようなゴール。

この定義を箱による制御フローモデル〔Byrd 80〕で示してみよう。まず、一つのゴールの非決定的な計算を、一つのゴールへの制御の出入りを図1で示すような腕が四本ある箱で表現することにする。

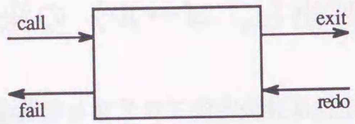


図１：制御のボックスモデル

また、Prologの質問及び定義節のボディのゴール列は、このような四本の腕をもつ箱がAND 結合で図２のように並んだものであると考えることができる。

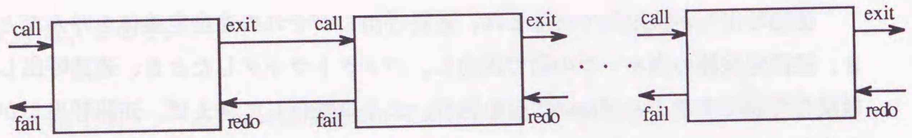


図２：ゴールのAND 結合

このとき、代替節が複数ある場合において、一つのゴールを決定的に達成するための制御の可能な流れの全体の様子は図３のように表現することができる。図３はゴールが決定的であることからその定義体は排他的にOR結合されていることを示している。ただし、同図において、ノード⊕はバックトラックしてきたコントロールは左に流し、failしたコントロールは下に流す働きをもつものとする。もちろんこれは再帰的な図を表していることに注意されたい。

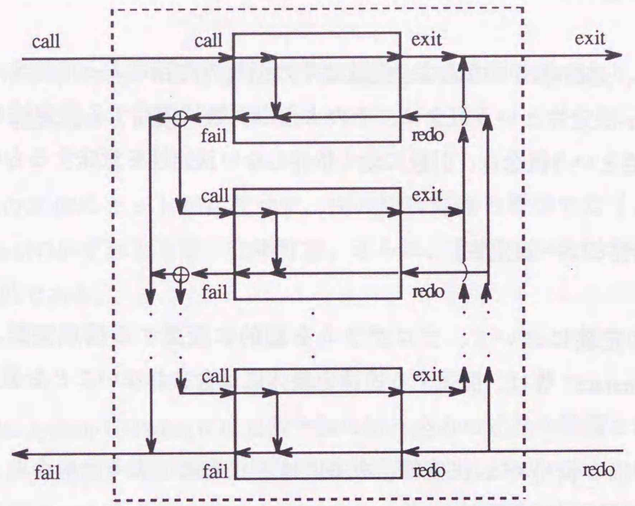


図３：決定的ゴールの制御フロー

上の我々の定義と実質的に同じものは〔Warren 77〕、〔Mellish 85〕にも見出され、Prologにおける決定性、非決定性の意味として共通に了解されているものである。そして、これは、述語（関係）が関数的である場合に決定的であるとする定義（〔Kowalski 79〕、〔Mendelzon 85〕、〔Debray 86〕）よりも、一般的である。ここで、述語が関数的であるというのは、述語のある変数と他の変数が関数関係にあることを言う。上記の我々の定義は、そのような関数性とは異なり、述語呼出しの成功、不成功という概念を基にして与えられていることに注意されたい〔Sawamura 85b〕。

しかしながら、いずれの定義においても、一般に述語呼出しが決定的であるか否かを決定することは決定不能であるので (この証明は第４章で述べる)、次節では互いに関係する二つの決定可能な述語 (呼出し) の部分クラスを定義する。

2.5 a-決定性とr-決定性

一般に、述語呼出しの成功、失敗はその引数の内容によって決まる。以下で定義されるr-決定性という概念は、そのような引数に依存する決定性を意味し、他方a-決定性という概念は、引数に全く依存しない決定性を意味するものである。

【ａ－決定性とｒ－決定性】

以下の定義において、プログラムを動的に変更する構成要素、例えば"assert"、"retract"等は、関連する述語定義体には含まれないことを仮定する。

述語呼出しp(A)がa-決定的、あるいはr-決定的であるということを、次のように相互帰納的に定義する。

(i) pが組み込みの述語名で、かつp(A)が決定的であれば、p(A)はa-決定的であり、かつr-決定的である。

(ii) 述語ｐ に関する定義体を次のものとする。

```
H1 :- Γ1.
 :
Hi :- Γi, [!,] Δi.    (ここで、ボディ部に明示されているカット記号
 :                      は、カットが複数存在するならばそれらの最右
Hn :- Γn.              出現を表わしている)
```

このとき、各Hi(1≦ｉ≦n)に対して、次の条件のうち(1)あるいは(2)が成立するならば、p(A)はa-決定的であり、p(A)と単一化可能なHiに対して(1)～(3)のいずれかが成立するならば、p(A)はr-決定的である。

(1) Hiの本体にカットが存在し、Δiはすべてa-決定的か、r-決定的である。

(2) Hiの本体にカットが存在せず、Hiは定義体の最後の節であり、Γi、Δiはすべてa-決定的か、r-決定的である。

(3) Hiの本体にカットが存在せず、Hiの節は定義の最後でなく、p(A)はHj(i+1≦ｊ≦n)のいずれとも単一化不可能、さらに、Γi、Δiはすべてa-決定的か、r-決定的である。

注意:

(a) Dec system-10 Prologには百数十個の組み込みの述語が準備されている。それらは入出力述語、制御述語、超論理的述語等からなっているが、それらを決定性という観点からも分類することができる。決定的述語の中には、‘write'、`atom'、`var'等があり、非決定的述語の中には、`clause'、`repeat'、`retract'、`;'、`->'、`call'といった述語が含まれている。

(b) プログラムを相互帰納的に定義している場合でも、決定的と判定できることがある。例えば、次のプログラム

```
q :- Γ,!,r.
r :- Θ,q, Δ,!.
```

では、qおよびrは相互帰納的に定義されているが、呼び出しq、rは共にa-決定的でもあり、r-決定的でもあることが分かる。より一般的には次のように言うことができる。上の決定性の定義における述語pの定義体

H1 :- Γ1.
:
Hi :- Γi, [!,] Δi.　（ここで、ボディ部に明示されているカット記号は、カットが複数存在するならばそれらの最右出現を表わしている）
:
Hn :- Γn.

において、各i (1≦ i ≦n)に対して条件(1)のΔi、条件(2)及び(3)のΓi,Δiを述語pのクリティカルパートと呼ぶことにする。このとき、プログラムを構成する各述語定義体のクリティカルパートに含まれる述語呼び出しの間のクロス・レファレンス(crossreference)を考えると (これは一般に有向グラフになる)、これにループがないことが決定性判定のためには必要である。上の例では、クリィティカルパート間のクロス・レファレンスには相互呼び出しは存在していない。

(c) プログラム（手続きの集まり）が与えられたとき、その中から上の定義を用いてa-決定的述語を抽出することができる。そしてこの抽出は一意に定まる。すなわち、どの述語からa-決定性の判定を行うかに依存しない。これは(b)で述べたクリティカルパートのクロス・レファレンスの一意性による。

(d) クリティカルパートに(A ; B)、(A -> B ; C)のような形式の複合ゴールが含まれているとき、研究の現段階ではそれらを単に非決定的ゴールとして扱っている。(A ; B) が決定的と判定されるためにはA、B 共に決定的でかつそれらのゴールの成功が排他的であることが必要となるが、そのことを一般的に判定することは不可能であろう。

我々の定義の正当性は次の系によって保証される。

系　1．ゴール p(A) が a- 決定的か r- 決定的ならば、それは決定的である。

証明． 定義の構成に関する帰納法に従う。(i) のケースは明らか。(ii)のケースは3 つの条件を一つ一つ調べれば十分であるが、それらはPrologの実行意味論と帰納法の仮定から直ちに導かれる。■

a-決定性とr-決定性の定義は三つの概念に基づいていた。すなわち、カット、組み込みの決定的述語及び静的に決定される単一化可能性である。別の言い方をすると、それらは述語呼出しに現れる変数がどのような型の引数を取るかということには関わっていない。すなわち、述語の意味に関わることなく定められる決定性である。ここで、述語の意味とは、その述語が成功する項の領域をいう。

定義より、ゴールが a- 決定的ならば、それはその引数の形によらずいつも決定的であることがわかる。別の言い方をすると、a-決定的なゴールは、それがゴールの中の引数の形式に依存しないという意味で絶対的に決定的である (absolutely deterministic)。それゆえ、ゴール p(A) が a- 決定的であることがわかったとき、述語 pを a- 決定的とか、あるいは単に決定的と呼ぶことがある。他方、絶対的な決定性とは対照的に、r-決定的なゴールは、その述語がどのように呼ばれるかに依存しているので相対的に決定的である(relatively deterministic) 。ゴール p(A)がr-決定的であるとき、述語p がr-決定的であると呼ぶことがある。これらの観察を次のような系として表明しておく。

系　2．ゴールが a- 決定的ならば、それはまた r- 決定的である。

さらに、定義より、

系　3．組み込みの述語を除き、決定的な述語呼出しがa-決定的と判定されないのは、その定義体の数が 2 より大きく、かつその中にカット記号が全く出現しないときである。

もちろん、次の簡単なプログラム例から示唆されるように、系3の条件は強過ぎるように思われるかもしれない。

H :- Γ, fail.
H :- Δ.

ここでカット記号はΓ及びΔには出現せず、Δのすべての述語呼出しはa-決定的かr-決定的である。したがって、述語呼出しHは、その定義体に全くカットが含まれていないのにも拘らず決定的であると言える。このような個々の特殊な場合を上の定義に組み込むことは難しくないが、それよりもここではできるだけ一般的な決定性の定義を設定することに関心があった。

系　4．命題レベルでは、すなわち、調べようとしている述語に変数が含まれないとき、a-決定性はr-決定性に一致する。

系　5．すべての項Aに対してゴールp(A)はr-決定的である <=> 述語pはa-決定的である。

証明．(=>) Aの任意性より、ゴールp(X)を考える。ここで、Xは異なった変数の列を表す。これはr-決定性の条件(3)によってr-決定的とはなり得ない。したがって、p(A)がすべてのAに対してr-決定的ならば、それはr-決定性の条件の(1)あるいは(2)の下でr-決定的とならなければならない。ゆえに、述語pはa-決定的となる。

(<=)　系2より。■

系　6．任意の引数列Aに対してゴールp(A)がr-決定的であることと、Xを異なった変数の列とするときp(X)がr-決定的であるということは同値である。

証明．(=>) 明らか。

(<=) 系5の証明と同様の理由による。■

系　7．Xを異なった変数の列とする。ゴールp(X)はr-決定的である <=> 述語pはa-決定的である。

証明：系5と系6による。■

系5と7は決定性に関して別の同値な定義を示唆する。すなわち、最初にr-決定性を定義し、次に系4あるいは系7にしたがってa-決定性を定義するものである。正確に表現すると、次のようになる。

【r－決定性】

以下の定義において、プログラムを動的に変更する構成要素、例えば"assert"、"retract"等は、関連する述語定義体には含まれないことを仮定する。述語呼び出しp(A)がr-決定的であるということを、次のように帰納的に定義する。

(i) pが組み込みの述語名で、かつp(A)が決定的であれば、p(A)はr-決定的である。

(ii) 述語pに関する定義体を次のものとする。

H1 :- Γ1.
　:
Hi :- Γi, [!,] Δi.　（ここで、ボディ部に明示されているカット記号は、カットが複数存在するならばそれらの最右出現を表わしている）
　:
Hn :- Γn.

このとき、p(A)と単一化可能な各Hi(1≦ i ≦n)に対して、(1)～(3)のいずれかが成立するならば、p(A)はr- 決定的である。

(1) Hiの本体にカットが存在し、Δiはすべてr-決定的である。

(2) Hiの本体にカットが存在せず、Hiは定義体の最後の節であり、Γi、Δiはすべてr-決定的である。

(3) Hiの本体にカットが存在せず、Hiは定義体の最後の節でなく、p(A)はHj (i+1 ≦ j ≦n)のいずれとも単一化不可能、さらに、Γi、Δiはすべてr-決定的である。

【a-決定性】

述語p は、すべての項A に対してゴールp(A)がr-決定的であるとき、a-決定的と呼ばれる。

あるいは、単に、

【a-決定性】

述語 p は、ゴールp(X)がr-決定的なとき、a-決定的と呼ばれる。ただし、X は異なった変数の列である。

以上の定義は以前の定義より簡単なように思われるが、a-決定性の第一の定義は構成的でないことを注意しておかなければならない。これまでの議論から、決定性の判定アルゴリズムは最初のa-決定性とr-決定性の定義かあるいは後で定

義されたr-決定性とすぐ上のa-決定性の定義に基づいて与えられることがわかる。我々のPrologオプティマイザでは，前者の定義に従って決定性判定アルゴリズムが実現されている。

例１．次の述語pは a- 決定的である。

```
p(a) :- write(a1), nl, !, write(a2).
p(b) :- write(b).
```

言い換えると、ゴールp(X)はr-決定的である。なぜなら、ゴールp(X)は第一節で成功し、バックトラックしても述語write の決定性とカットにより直ちに失敗するからである。

例２．述語pの定義体を例１のものとすると、述語呼出しq[a,b]) は r- 決定的であるが、q(X)はそうではない。なぜなら、述語呼出しq[a,b]) は第二節でしか成功せず、バックトラックしても述語write とp の決定性により直ちに失敗するからである。

```
q([c]) :- write(c), p(b).
q([a|X]) :- write(a), p(X).
```

例３．次の述語transform は、述語foldがどのようなものであれ、a-決定的である。

```
transform(`end-of-file') :- !.
transform(T) :- fold(T,R),write(R),write(`.'),nl,!,fail.
```

これまでPrologプログラムの最適化に必要とされる述語 (呼出し) の決定性を

論じてきた。a-決定性とr-決定性という概念は、Prologプログラマが日常無意識のうちに用いている直観を自然に定義したものに他ならない。また、それらの定義によって捉えられる決定的なゴールのクラスもかなり大きなものと考えられる。

2.6 決定性の拡張

これまでのa-決定性及びr-決定性なる概念は完全に統語論的に与えられてきた。この章の最後に、これらの定義を意味論的に拡張する有力な二つの方法について触れておく。

(1) r-決定性の定義の条件(3)において、ゴールとその定義体のヘッドとが単一化可能であったとき、そのときの代入情報をそのボディ側のゴールの決定性判定に利用する。すなわち、以下の定義体において、ゴールp(A)がヘッドHiと単一化可能であったとき、そのときのユニフィケーション情報 (代入情報) を、Γiかつ（あるいは）Δiの決定性判定に利用する。

```
H1 :- Γ1.
  :
Hi :- Γi, [!,] Δi.   (ここで、ボディ部に明示されているカット記号
  :                    は、カットが複数存在するならばそれらの最右
Hn :- Γn.              出現を表わしている)
```

(2) 述語の型推論 (例えば、〔Mishra 84〕) によって述語引数のタイプを見出し、ゴールとその定義体ヘッドとの単一化可能性をより詳細に分析する。これはr-決定的なゴールのクラスの拡大を助けるであろう。

以上の拡張法は、一言でいえば、Prologの手続き (述語) 間のデータ／制御フローの分析を意図した意味的拡張であるといえる。(1)の拡張は簡単であるが、(2)の理論的検討と実現は今後の課題としなければならない。

第3章　PROLOGプログラムのための最適化変換手法

この章では、さまざまな最適化変換図式およびその適用条件を提案する。

3.1　部分ユニフィケーション

部分ユニフィケーションは、二つの項をユニファイすることを目的とするゴールを部分評価する最適化手法である。そのようなゴールは、Prologの組み込みの述語“＝”を用いて、$t_1 = t_2$ と表わされ、通常の等式に類似の性質をもつことから、本論文では等式ゴールと呼ぶことにする。ここで、t_1 および t_2 はPrologの項である。等式ゴールがPrologプログラム中に現れるのは次の二つの場合においてである。すなわち、最初からユーザのプログラムの中に明示的に等式ゴールが含まれている場合と、第3．3節で述べられるインライン展開の方法によってプログラムテキスト上に現れてくる場合である。

通常のプログラム言語では、いわゆるサブルーチンの呼出し機構はサブルーチンの展開機構によってうまく取り除かれる。Prologでは、一般に、それに対応する述語の呼出し機構は、ゴールとその定義体のヘッドとの単一化可能性を表す等式ゴールという形で残る。部分ユニフィケーションはこのような等式ゴールを部分評価することによって実行時のユニフィケーションのプロセスの負担を軽減する。それのみでなく、本オプティマイザにおいては、部分ユニフィケーションの結果である等式ゴール列は、引き続く最適化過程においても利用される。例えば、第3．5節で述べる等式代入による最適化においてである。そして最適化の最後において、残された等式ゴール列は達成すべきゴールの数を減らす目

的で、第3．6節で述べるゴール列の統合化によって、一つの等式ゴールへと統合化されることになる。

二つの項の単一化可能性を表現するPrologの等式ゴールは、それと同値な単一化を表現する等式ゴール列へと次の図式によって変換される。

【最適化変換図式　1】

単一化可能なとき：	単一化不可能なとき：
f(・・・) = f(・・・)	f(・・・) = f(・・・)
X1 = T1, ・・・, Xn = Tn	fail

ここで、X1，・・・，Xnは上式に現れる変数のいくつか、T1，・・・，Tnは項である。以下に、上式から下式への変換アルゴリズムをMartelliとMontanari〔Martelli 82〕に従って詳細に定義する。まず、等式の集合に対して次の二つの形式の変換を定義する：

(1)　項リダクション

$$\frac{\{f(t1,\ldots,tn) = f(r1,\ldots,rn),\ \ldots\}}{\{\ t1 = r1,\ldots,tn = rn,\ \ldots\ \}}$$

ここで、ti,ri は項を表す。また、n = 0 のとき、等式を単に除去する。

この変換は、二つの項のユニフィケーションを、それら二つの項の対応する引数のユ

ニフィケーションに置き換えることを意味する。

(2) 変数除去

$$\frac{\{t1 = r1, \ldots, x = t, \ldots, tm = rm\}}{\{t1 = r1, \ldots, tm = rm\}\ \theta \quad \cup \{x = t\}}$$

ここで、θは代入 {t/x } を、t,ti,ri は項を、x は変数を表す。

この変換は、上式にある複数の等式ゴールを同時にユニファイすることは、等式ゴールのいずれかの側に変数があるような等式ゴールx = t を選び、それを代入 {t/x } と考え他の等式ゴール列に施してできる等式ゴールの集まりと、この等式x = t との和集合からなる等式ゴールをユニァイすることに等しいことを意味する。

等式の集合が与えられたとき、それが次の条件を満たすとき解形式にあると言われる:

① その要素はxi = ti (1≦ i ≦k) の形式である。ただしxiは変数、tiは項である。

② 等式の左辺に現れる変数はそこにしか現れない。

解形式の等式の集合は自明なユニファイヤ

$$\theta = \{t1/x1, \ldots, tk/xk\}$$

をもち、もし他のユニファイヤσをもつならば、それは次のように得られる。

$$\sigma = \theta \circ \alpha = \{t1\alpha/x1, \ldots, tk\alpha/xk\} \cup \alpha$$

ただしαは変数xiを書き変えないある代入である。すなわち、任意のユニファイヤσはユニファイヤθにある適当な代入αを合成して得られるという意味において、θは最も一般的なユニファイヤ(most general unifier)と呼ばれる。

等式の集合を解形式の等価な等式の集合に変換する非決定的アルゴリズムを以上の定

義のもとに述べる。

〔アルゴリズム〕

等式の集合が与えられているものとする。次の変換のいずれかを繰り返し行う。

いかなる変換も適用できなければ、成功して停止する。

(a) 次の形式の任意の等式を選択する。

t = x 、ただし、t は変数でなく、x は変数である。

そしてこれを次のように書き変える。

x = t

(b) 次の形式の任意の等式を選択する。

x = x 、ただし、x は変数である。

そしてそれを除去する。

(c) 次の形式の任意の等式を選択する。

t = r 、ただし、t,r は変数でない項である。

これらの項の主要関数記号が異なっていれば失敗として停止する；さもなくば項リダクションを適用する。

(d) 次の形式の任意の等式を選択する。

x = t 、ただし、x は等式の集合の他のどこかで生起している変数で、

t はx と異なる項である。

このとき、x がt の中に出現しているならば失敗として停止する；さもなくば変数除去を適用する。

次の定理はこのアルゴリズムの正当性を保証する。証明は文献〔Martelli 82 〕で与えられている。

定理 (Martelli and Montanari, 1982)

等式の集合Ｓが与えられているとする。

（ｉ）アルゴリズムは、選択点がどのようなものであれ停止する。

（ii）アルゴリズムは失敗して停止するならば、Ｓはユニファイヤをもたない。またアルゴリズムは成功して停止するならば、集合Ｓは解形式の等価な集合に変換される。

例　4．最適化変換図式の具体例

```
p(X,g(Y,X,c)) = p(h(Z),g(Z,h(d),c))
```

```
X = h(d), Y = d, Z = d
```

下式は上式に上で述べたアルゴリズムを適用して次のようにして導出された。

```
p(X,g(Y,X,c)) = p(h(Z),g(Z,h(d),c))
```

((c)の適用)

```
X = h(Z),g(Y,X,c) = g(Z,h(d),c)
```

((c)の適用)

```
X = h(Z),Y = Z,X = h(d)
```

((d)の適用)

```
X = h(Z),Y = Z,h(Z) = h(d)
```

((c)の適用)

```
X = h(Z),Y = Z,Z = d
```

((d)の適用)

```
X = h(d),Y = d,Z = d
```


3.2　カットの自動挿入

Prologのような非決定的なプログラム言語では、無駄なバックトラックを避けるための最適化は本質的である。ここでは、バックトラックしたとき再び成功することがないことが分かっているとき、適切な位置にカット記号を挿入して不必要なredoを避ける方法について述べる。この方法においては述語の決定性の検出が重要な役割を果たしている。

次の最適化変換図式は、第ｉ節のボディ部のゴール列Γｉとδｉの間にカットを挿入することを可能にする。

【最適化変換図式　2】

```
H1 :- Γ1.
 ⋮
Hi :- Γi,  Δi.
 ⋮
Hn :- Γn.
```

```
H1 :- Γ1.
 ⋮
Hi :- Γi, !, Δi.
 ⋮
Hn :- Γn.
```

ただし、次の条件が満たされていなければならない。

【適用条件】

① Γi の中にカットが存在しなければ、Hiの節はこの定義節の最後の節であり、Γi は a- 決定的か、r-決定的である。

② Γi の中にカットが存在するならば、Γi の最右カットの右にある述語はすべて a- 決定的か、r-決定的である。

この条件の中の ‘ a- 決定的か、r-決定的 ’という部分は単に ‘r-決定的 ’とすれば十分である（前章の系2による）。しかしながら、ここでは我々の二つの決定性の定義を忠実に反映することを意図した述べ方を採用した。

例 5.

```
r( [c ] ) :- write(c), !, p(b).
r( [a |X ] ) :- write(a), q(X).
```

```
r( [c ] ) :- write(c), !, p(b),!.
r( [a |X ] ) :- write(a), q(X).
```

```
r( [c ] ) :- write(c), !, p(b), !.
r( [a |X ] ) :- write(a), !, q(X).
```

ここで、述語呼出しp(b)とq(X)はそれぞれ例1と例2の定義節を呼んでいる。中段の第一節の最右カットは適用条件の②によって、下段の第二節のカットは適用条件の①によって挿入されている。

3.3 インライン展開

Prologの論理変数(logical variable)は、部分的に具体化された対象(partially instantiated object) を許すという意味において、インライン展開のような部分評価に好都合な言語特性の一つである。以下ではPrologの論理変数の特性を十分利用した最適化法としてインライン展開を考える。

インライン展開の主要目的は次の二点にある。

① 述語の呼出し機構の除去（軽減）

② 他の最適化手法の適用範囲の拡大

非決定性プログラミング言語であるPrologのインライン展開は、一般に代替節が複数存在しているために，通常のプログラミング言語のサブルーチン展開よりも複雑になる。

ここではPrologのインライン展開を、『述語の呼びを，呼出し機構を表す等式を付随した代替節の選言によって置き換える』という自然な方法によって行う。

しかしながら，これが自由に正しく行えるのは呼ばれる側の述語定義体の中にカット記号が現れないときである。なぜなら、次の例に見られるように、述語呼出し側に持ち込まれた定義体側のカットは、バックトラックが起こったとき以前とは異なった制御フローを引き起こすからである。

(1) インライン展開前 :

```
p :- q, a, r.

p.

a :- b,!,c.

a :- d.
```

(2) インライン展開後 :

```
p :- q, (a=a,b,!,c ; a = a,d),r.
```

```
p.
a :- b,!,c.
a :- d.
```

ここで、“＝”は単一化可能性を表すPrologの組み込み述語である。プログラム(1)において、述語呼出しｃが失敗したと仮定してみる。そのとき、述語呼出しa は失敗し、制御はq にバックトラックする。他方、プログラム(2)では、述語呼出しｃの失敗は述語呼出しpを失敗させることになる。

このように、カットの存在はインライン展開に重大な影響を与えることになる。以下に、このようなインライン展開の問題を避けるために述語（呼出し）の決定性を用いた条件を与える。上の例で示したカットの変則的なふるまいをProlog言語のある種の“指示的不透明性”(referentially opacity)的問題として捉えることについての議論は第８．２節を見られたい。

インライン展開におけるカットの問題はDEC-10 Prolog〔Bowen 81〕における特有の現象に見えるかもしれない。確かに、C-Prolog〔Pereira 83〕、KL0〔Chikayama 83b〕ではカットの意味が異なるために、呼出しゴールが真に選言項に置き換えられるときはこのようなことは起こらないようにできる。その代わりに、本論文の第３．７節で与えられる最適化図式においては、これがちょうどインライン展開と逆の効果をもつ変換であるために以下で述べるようなインライン展開の条件と類似した条件が必要になってくる。

３．３．１　カットをもたない述語定義体によるインライン展開

ゴールG はその定義体にカットを含まないときは、次のように無条件にインライン展開される。

【最適化変換図式　３】

```
G
H1 :- Γ1.
H2 :- Γ2.
 ⋮
Hn :- Γn.
―――――――――――――――――――――――――
G =H1',Γ1' ; ...  ;G =Hn',Γn'
H1 :- Γ1.
H2 :- Γ2.
 ⋮
Hn :- Γn.
```

ここで，“Hi :-Γi”（1 ≦i ≦ n）はG の定義節とし，Γi が存在しないときは、”G = Hi', Γi” は単に ”G =Hi'” を表すものとする。

例 6.

```
transform(T) :- fold(T,R),write(R),write('.'),nl,!,fail.
fold((H :- G),(H :- R)) :- red(H,G,R).
fold(H,H).
```

```
transform(T) :- (fold(T,R) = fold((H1 :- G1),(H1 :- R1)),
                 red(H1,G1,R1) ; fold(T,R) = fold(H2,H2)),
                write(R),write('.'),nl,!,fail.
fold((H :- G),(H :- R)) :- red(H,G,R).
fold(H,H).
```

3．3．2 カットを含む述語定義体によるインライン展開

(1) カット図式（その1）

ここでは、次のような述語定義体の中にカットがちょうど一つ含まれる特殊なケースを考える（類似の図式は〔O'Keefe 85〕にも見られる）。

H1 :- Γ1,!,Δ1.
⋮
Hn :- Γn,!,Δn. あるいは Hn :- Γn. あるいは Hn.

ここで、Γi 、Δi(1 ≦ i ≦n) にはカットは含まれない。そして、Γi あるいはΔi が空のとき、節の本体 "Γi,!,Δi" は次のものを表す。

"!" : Γi およびΔi が空のとき

"!, Δi" : Γi が空のとき

"Γi,!" : Δi が空のとき

このとき、ゴールG は次の図式に従って展開される。

【最適化変換図式 4】

G
H1 :- Γ1,!,Δ1.
⋮
Hn :- Γn,!,Δn. あるいは Hn :- Γn. あるいは Hn.

G = H1',Γ1' -> Δ1' ; ・・・ ; G = Hn',Γn' -> Δn' あるいは
G = Hn',Γn' あるいは
G = Hn'

H1 :- Γ1,!,Δ1.
⋮
Hn :- Γn,!,Δn. あるいは Hn :- Γn. あるいは Hn.

ただし、Γi あるいはΔi が空のとき、下式の選言項の第i 番目の項は次のように構成される。

G = Hi' -> Δi' : 第i番目の節が "Hi :- !,Δi." のとき
G = Hi',Γi' -> true : 第i番目の節が "Hi :- Γi,!." のとき
G = Hi' -> true : 第i番目の節が "Hi :- !." のとき

例　7.

```
transform(T)
transform('end-of-file') :- !.
transform(T) :- fold(T),write(R),write('.'),nl,!,fail.
```

```
transform(T) = transform('end-of-file') -> true ;
  transform(T) = transform(T1),fold(T1),write(R1),write('.'),nl -> fail
transform('end-of-file') :- !.
transform(T) :- fold(T),write(R),write('.'),nl,!,fail.
```

(2)　カット図式（その２）

カットが述語定義体の中に出現する場合は、述語の決定性を用いてインライン展開を可能にすることができる。

【最適化変換図式　５】

```
H1 :- Γ1.
 ⋮
Hi :- Γi,p(A), Δi.
 ⋮
Hm :- Γm.
p(A1) :- Θ1.
 ⋮          (Θj(1 ≦j ≦n)のいずれかにカットが含まれているものと
p(An) :- Θn.    する)
```

```
H1 :- Γ1.
 ⋮
Hi :- Γi,
      ( p (A)= p(A1') , Θ1' ; ... ; p(A) = p(An') , Θn') ,
      Δi .
 ⋮
Hm :- Γm.
p(A1) :- Θ1.
 ⋮
p(An) :- Θn.
```

ただし、次の条件を満たさなければならない。

【適用条件】

① Γi の中にカットが存在しないとき ：

Γi の中のすべての述語は a- 決定的か、r-決定的であり、Hiは定義節の最後の節である。

② Γi の中にカットが存在するとき ：

Γi の中の最も右にあるカットの右にあってΓi に含まれるすべての述語は a-決定的か、r-決定的である。

この条件においてもカットの自動挿入と同じように、‘a- 決定的か、r-決定的’という部分を単に‘r- 決定的’としてよい。また、特別な場合として、カットが述語定義体の中に出現しているとしても、上の最適カット図式において条件(1)及び(2)がなくとも正しくインライン展開が行われることがある。それは、述語呼出し p(A) とカットを含む節のヘッドのユニフィケーションが失敗すると分かっている場合である。しかしながら、ここでは簡単化のためにそのような状況は考慮していない。

例 8.

```
r(a,Y,Z) :- !, q(Y), append( [a ] ,Y,Z).
r(b,Y,Z) :- p(b), append( [b ] ,Y,Z).
append( [] ,L,L) :- !.
append( [X | L1] ,L2, [X | L3] ) :- append(L1,L2,L3),!.
```

```
r(a,Y,Z) :- !, q(Y), append( [a ] ,Y,Z).
r(b,Y,Z) :- p(b),
 (append( [b ] ,Y,Z) = append( [] ,L4,L4),! ;
  append( [b ] ,Y,Z) = append( [X | L5] ,L6, [X | L7] ),
  append(L5,L6,L7),!).
```



```
append( [] ,L,L) :- !.
append( [X | L1] ,L2, [X | L3] ) :- append(L1,L2,L3),!.
```

ここで、述語呼出し q(Y)及び p(b) は例１、２の述語定義を呼んでいる。上式の第一節のゴールappend([a] ,Y,Z) はその直前のゴールq(Y)が決定的でないために展開することができない。他方、第二節目のゴール append([b] ,Y,Z) は条件より展開可能となっている。

下式の第二節は後で述べる最適化変換手法を使うと、以下の変換過程を経て最も下の式： r(b,L, [b | L]):- p(b),!. へと変換される。

```
r(b,Y,Z) :- p(b),
 (append( [b ] ,Y,Z) = append( [] ,L4,L4),! ;
  append( [b ] ,Y,Z) = append( [X | L5] ,L6, [X | L7] ),
  append(L5,L6,L7),!).
```

```
r(b,Y,Z) :- p(b),
 (fail ; X=b, L5 = [] ,Y=L6,Z= [b | L7] ),append(L5,L6,L7),!.
```

```
r(b,Y,Z) :- p(b),X=b, L5 = [] ,Y=L6,Z= [b | L7] ),append(L5,L6,L7),!.
```

```
r(b,L6, [b | L7] ) :- p(b),append( [] ,L6,L7),!.
```

```
r(b,L6, [b | L7] ) :- p(b),append( [] ,L6,L7)= append( [] ,L,L),!,!.
```

```
r(b,L6, [b | L7] ) :- p(b),L6=L,L7=L,!,!.
```

```
r(b,L, [b |L ] ) :- p(b),!,!.
```

```
r(b,L, [b |L ] ) :- p(b),!.
```

以下の節では、以上のインライン展開の最適化図式を、いろいろな型の述語定義の集団の中で利用する方法を提案する。展開の処理を分かりやすくするために、各述語定義はその型にしたがって展開されるが、我々のこの段階的展開法は本質的には直線的なプログラムを呼んでいるゴールを展開していることに相当する。ただし、終端再帰型プログラムの場合だけはその終端のゴールが一度展開されるようになっている。これは、帰納的なプログラムに対する客観的な展開基準をまだ見出し得ていないという現在の研究の状況によっている。それゆえに、以下で与えられる展開方法は我々のインライン展開の最終的な解答ではないということを注意しておかなければならない。本論文では、その代わりに理想的なインライン展開基準に向けての第一歩として、元々の述語定義の型がインライン展開によって保存される範囲内でインライン展開を行うことにした。これは結果的には、型を保存するインライン展開がオプティマイザの信頼性を保証するためにも、またデバッグにも都合がよいという利点を生むことになった。また、プログラムの型の保存によって、プログラムの構造が保持されることになるので、プログラマのプログラムに対する意図をそれほど崩さないことにもなるので利点も多いと考えられる。

以下では、インライン展開のアルゴリズムを細かく記すよりも、展開の全体像が見えるように述べる。ただし、この中では【最適化変換図式　4】の組み込みは行われていない。

3．3．3　直線型述語定義のインライン展開

さまざまな述語定義からなるプログラムが与えられるているとする。このとき、直線型述語定義を構成している各節の各ゴールはプログラムの中の対応する述語定義を用いて次のようにインライン展開される。これは、Prologの実行にしたがって上から下、左から右という順序で行われる。

述語名ｐの述語定義を選ぶ：

（ｉ）述語ｐはａ－決定的であると仮定し、その定義体を、

```
H1 :- Γ1.
 ⋮
Hi :- Γi, [! ] , Δi.（ここでカット記号はもし存在するならボディ部の最右
 ⋮                    出現とする）
Hn :- Γn.
```

とする。このとき、

Γｉ の各ゴールに対して、

(1)　もしそれが次のすべての条件を満たすならば、それを【最適化変換図式　3】によって展開する；

(a)　そのゴールの定義体にはカットは現れない、

(b)　そのゴールの定義体は直線型である、

(c)　そのゴールの展開によって生成される選言項には、述語名ｐを含めてすでに展開された述語名は現れない。

次に、その結果得られる選言項の各ゴールに対して、これを繰り返す。

(2)　もし、そのゴールの定義体にカットが含まれているならば、それを【最適化変換図式５】によって展開する。次に、生成された選言項の各ゴールを上の(1)と同様に展開す

る。

Δi の各ゴールと最後の節の各ゴールに対して、

(3) もし、それが条件(b)及び(c)を満たすならば、それを【最適化変換図式 3】によって展開し、次に結果の選言項の各ゴールを上の(1)と同様に展開する。

(ii) さもなくば、述語名 p の定義体の各ゴールを上の(1)あるいは(2)と同様に展開する。

注意:

① 条件(c)は直線型述語定義のインライン展開のための停止条件を与える。それはインライン展開が循環するのを禁止している。

② 直線型述語定義のインライン展開の結果は、条件(b)によって再び直線型となる。

例 9. 次のプログラムでインライン展開を例示する。

```
p :- p1,q,p2.
q :- q1,r,q2.
r :- r1,p,r2.
```

各述語定義は直線型である。第一の節のゴールq は第二の節によって展開され、

```
p :- p1,(q = q,q1,r,q2),p2.
```

となる。これはこれ以上展開条件によって展開されない。第二節は第三節によって展開され、

```
q :- q1,(r = r,r1,p,r2),q2.
```

これはさらに、述語p の新しい述語定義によって、

```
q :- q1,r = r,r1,(p = p,p1,q = q,q1,r,q2,p2),q2.
```

へと展開される。これは第3. 4節の局所的最適化手法を用いるならば、

```
q :- q1,r1,p1,r,q2,p2.
```

と簡約される。第三の節は展開条件によって展開されない。

3. 3. 4 終端再帰型述語定義のインライン展開

終端再帰プログラムはしばしば効率化のために繰り返しのプログラム形に変換される。実際、DEC-10 Prolog コンパイラではそれを行っている。しかしながら、Prologは繰り返しという概念をもたない再帰型の言語であるから、ソースレベルでの終端再帰型から繰り返し型への変換は不可能である。

以下では、終端再帰型の節において、繰り返し達成されるゴールをインライン展開するという方法をのべる。これは、繰り返し型プログラムにおけるループ内でのサブルーチンと同じアイディアであり、インライン展開の効果が最も期待されるものである。さらに、Prologプログラムはそのほとんどが再帰型であり、特に終端再帰型が多いことを考えると、そのようなインライン展開の効果は極めて大きいと考えられる。

直線型述語定義のインライン展開を終えた後、終端再帰型の述語定義のインライン展開が次のように行われる。

(i) 終端再帰型述語定義を構成しているすべての直線節を節3. 3. 3と同様に展開する。

(ii) 述語名 p の定義節の一つを、

```
p(A) :- Γ,p(B).
```

とする。ここで、Γには述語名 p をもつゴールは現れないものとする。このとき、Γの各ゴールG を節3. 3. 3と同様に展開する。

(iii) (i)、(ii)の後、終端再帰ゴール p(B) を自分自身によって次のように一回展開する。

(1) もし述語 p が a −決定的ならば、終端再帰ゴールを【最適化変換図式 3】

によって一回展開する。

⑵ もし述語ｐの定義体にカットが含まれているならば、終端再帰ゴールを【最適化変換図式　5】によって一回展開する。

注意:

① 直線型述語定義のインライン展開の停止条件は終端再帰型述語定義のインライン展開の停止条件ともなる。

② 述語定義の終端再帰性はインライン展開によって（節２．３で言及した意味において）保存される。

３．３．５　一般再帰型述語定義のインライン展開

一般再帰型述語定義を構成している各節の再帰的でないゴールは、節３．３．３で述べた直線型述語定義のゴール展開と同様に展開される。

注意:

① 述語定義の一般再帰性はインライン展開によって保存される。

３．４　ゴール列の簡単化

この節では、命題論理的にゴールを変形する手法や変数の除去などに伴う最適化の方法について述べる。これらはほとんどが局所的な簡単化や冗長性の除去戦略であり、インライン展開されたプログラムに対してしばしば適用される。

３．４．１　ゴールの多重生起の除去

（ｉ）連言列中の重複除去

連言列の中に起こる二つ以上の同じ述語は、次の最適化図式を繰り返し適用して、最も左の述語を残し他を除去する。

【最適化変換図式　６】

$$\frac{\Gamma,\ P,\Delta,P,\ \Theta}{\Gamma,\ P,\Delta,\ \Theta}$$

ただし、下式のゴールP は最も左の出現を示し、さらに次の条件を満たさなければならない。

【適用条件】

⑴ ゴール Pは r- 決定的である。

⑵ P は組み込みの入出力述語といったサイドイフェクトをもつ述語呼出しとか、メタ述語呼出しではない。また、それはカット、"repeat"といった特別の制御述語であってはならない。

以下に、このような条件が満たされていない最適化変換図式の反例のいくつかを示す。

$$\text{(a)}\quad \frac{p,q,p}{q,p}$$

は正しくない。なぜなら、上式の最初のp が停止せず、下式のq が停止しその下でp が停止する場合を考えると同値とはならなくなるからである。

$$\text{(b)}\quad \frac{\Gamma,var(X),p(X),var(X),\Delta}{\Gamma,var(X),p(X),\ \Delta}$$

は正しくない。ここで、“ var”はメタ述語である。

$$\text{(c)}\quad \frac{\Gamma,write(X),p(X),write(X),\Delta}{\Gamma,write(X),p(X),\ \Delta}$$

は正しくない。ここで、 "write"は出力述語である。

次の竹島卓氏による例は、除去される述語が非決定的であってはならないことを示している。

(d) 次の節が与えられているとする :

```
q(a).
q(b).
q(c).
```

このとき、次の図式は正しくない :

$$\frac{repeat,q(X),repeat,not(X = a)}{repeat,q(X),not(X = a)}$$

なぜなら、上式では q(X) がq(a)で成功すると、 "not(X = a)" を無限に繰り返すことになるが、下式では q(X) がq(b)で成功すると、下式の実行を終わらせることになるからである。

次の加藤昭彦氏による例は、解の生成の順序に意味があるとするとき問題になる例である。しかしながら、一般に非決定性プログラム言語のプログラムの同値性は、解の集合によって定められるのが普通であるので、本質的な難点とはならない。

(e) 次の節が与えられているとする :

```
p(f(Y,V)).
p(f(b,c)).
p(f(b,d)).
q(f(b,U)).
q(f(e,V)).
```

このとき、次の図式を考えてみよう :

$$\frac{p(X),q(X),p(X),not(X = f(b,c))}{p(X),q(X),not(X = f(b,c))}$$

このとき、上式の解集合は { f(b,d),f(e,V)} で、下式のそれは { f(e,V),f(b,d)} である。

(ii) 選言列中の重複除去

選言列の中に起こる二つ以上の同じ述語は最も左の述語を残し他を除去する。

【最適化変換図式　7】

$$\frac{\Gamma\,;\,P\,;\,\Delta\,;\,P\,;\,\Theta}{\Gamma\,;\,P\,;\,\Delta\,;\,\Theta}$$

ただし、下式のP は最も左の出現であり、次の条件を満たす。

【適用条件】

(1)　P はサイドイフェクトをもつ述語呼び出しではない。

(2)　バックトラックは上式の"P ;Δ ; P" の部分には入らない。

次の図式は条件(2)の意味を例示している。

$$\frac{\text{(p ; q; p),!,write(a),fail}}{\text{(p ; q),!,write(a),fail}}$$

3．4．2　冗長な "true" 、"fail"　述語の除去

ゴールの連言列に現れる冗長なProlog述語 "true"(あるいは、"X = X" など)，ゴールの選言列に現れる冗長なProlog述語 "fail" (あるいは、"not(X = X)"など) を除去する。

【最適化変換図式 8—1 】

$$\frac{\Gamma,\ \text{true},\ \Delta}{\Gamma,\ \Delta}$$

【最適化変換図式　8－2】

$$\frac{\Gamma;\ \text{fail};\ \Delta}{\Gamma;\ \Delta}$$

3．4．3　不実行部分の除去

制御が決して到達しないゴール列はプログラムテキストより除去してよい。例えば、ゴールの連言列の中にProlog述語 "fail" が現れるとき、その後ろに起こるすべてのゴールを除去する。

【最適化変換図式　9】

$$\frac{\Gamma,\ \mathrm{fail},\ \Delta}{\Gamma,\ \mathrm{fail}}$$

しかしながら、次の双対な図式は明らかに妥当ではない。

$$\frac{\Gamma;\ \mathrm{true};\ \Delta}{\Gamma;\ \mathrm{true}}$$

3．4．4　共通ゴールのくくり出し

ゴールの列はプログラムの効率と明快さのために、共通ゴールがくくり出される。これはゴール上の分配則の逆に相当する。

【最適化変換図式　10】

$$\frac{\Gamma,\ \Theta,\ \Delta\ ;\ \Gamma,\ \Lambda,\ \Delta}{\Gamma,\ (\Theta\ ;\ \Lambda),\ \Delta}$$

ただし、Γ、Δのすべての述語はサイドイフェクトをもたない。

例　10.

$$\frac{p,\ q\ ;\ p,\ q,\ r}{p,\ q,\ (\mathrm{true}\ ;\ r)}$$

3.5　等式代入による変数除去

ここでは、プログラムの変数を除去する最適化手法として、述語論理の等式代入規則から暗示される最適化手法を考える。これはインライン展開後のプログラムに対してしばしば適用され、達成すべきゴールの数を減らすのにも有効である。また、この最適化変換手法はデータ構造の（前方／後方）伝播として知られている最適化手法〔Komorowski 82〕の基本変換メカニズムに対応するものと見なすことができる。

【最適化変換図式　11－1】

$$\frac{p(\cdots,\ X,\ \cdots)\ :-\ \Gamma,\ X = t,\ \Delta.}{p(\cdots,\ t,\ \cdots)\ :-\ \Gamma(t),\ \Delta(t).}$$

【適用条件】

ここで、Γには、サイドイフェクトをもつ述語、カット記号及びメタ述語は現れない。ただし、"t" が変数のときはその左にカットが現れてもよい。

この最適化図式は次のような特別な場合を含んでいる。

```
p(Y) :- q(Y), X = t, r(Y).
```

(1) ―――――――――――――――――――――――――

```
p(Y) :- q(Y), r(Y).
```

```
P :-  Γ, X = f(t1, ・・・, tn), X = f(r1, ・・・, rn),  Δ.
```

(2) ―――――――――――――――――――――――――

```
P :-  Γ, f(t1, ・・・, tn) =  f(r1, ・・・, rn),  Δ.
```

次に上の条件が満たされない場合の反例を示そう。

```
q(X) :- !, X = t,  p(X).
q(r).
```

(a) ―――――――――――――――――――――――――

```
q(t) :- !, p(t).
q(r).
```

は正しくない。なぜなら、述語呼出し q(r) に対して、上式と下式では明らかに意味が異なっている。ただし、"t" は変数ではなく、"t" と"r" は単一化不能とする。

```
p(X) :- write(X),X = a.
```

(b) ―――――――――――――――――――――――――

```
p(a) :- write(a).
```

は正しくない。なぜなら、上式、下式ともに、例えばゴール p(b) に対して失敗するが、上式はサイドイフェクトとして "b"を書き出す。

【最適化変換図式　11－2】

```
p(・・・, X,・・・) :-  Θ; Γ, Y = t,  Δ; Λ.
―――――――――――――――――――――――――
p(・・・, X,・・・) :-  Θ; Γ(t),  Δ(t) ; Λ.
```

ただし、上と同様の条件を満たさなければならない。

3.6　ゴールの統合化

インライン展開において，あるゴールはそれを達成するために必要な述語の定義体で置き換えられるが，そのさいその呼出しの機構は一般にユニフィケーションの部分実行による等式の列によって表現された。そして，この等式列の一部はその後、等式代入による最適化に利用されることになった。ゴールの統合化とは，他の最適化に利用されることがなくなった等式の列を一つのゴールへと統合化するという最適化手法である。この最適化手法も達成すべきゴールの数を減らすのに有効である。

【最適化変換図式　12】

```
X1 = t1,  ・・・, Xn = tn
―――――――――――――――――――――――――
f(X1, ・・・, Xn) = f(t1, ・・・, tn)
```

ここで、"f" は適当な関数記号である。

例　１１．この例はユニフィケーションの部分実行と組み合わせて等式ゴールを最適化（簡単化）した例である。

```
p(X,g(Y,X,c)) = p(h(Z),g(Z,h(d),c))
-----------------------------------
X = h(d), Y = d, Z = d
-----------------------------------
f(X,Y,Z) = f(h(d),d,d)
```

3.7　節の複数節への分解

インライン展開された節は一般に次の形式をしている。

P :- Γ, (Θ1 ; ・・・ ; Θn) , Δ.

インライン展開とは逆に、このような形式の節を複数節に分解すると、局所的な最適化がさらに進行する場合がある（特に，等式代入による最適化が適用可能となる）。

【最適化変換図式　１３】

```
P :- Γ, (Θ1 ; ・・・ ; Θn ) , Δ.
-----------------------------------
P :- Γ, Θ1, Δ.
 ⋮
P :- Γ, Θn, Δ.
```

【適用条件】

ただし、Γにはサイドイフェクトをもつ述語は現れてはならない。また、Γにはカットが現れてはならない。

次の例はカットの作用により、正しくならない（A 、B が同時に成功することがなければ正しい）。

```
H :- Θ,!,(A ; B).
-----------------
H :- Θ,!,A.
H :- Θ,!,B.
```

この最適化手法は、他の最適化手法の適用を可能にするための変形規則と見られ、Γが空、もしくは下式の節の各ヘッドがその後の最適化によって互いに単一化不能とすることができたとき有効であると考えられる。

別の見方をするならば，節の複数節への分解は，非決定的な計算過程（パス）を定義節として数え上げていくことに相当する。

第４章　理論的考察と諸結果

この章では、これまで述べてきたPrologプログラムの最適化変換手法について、次の３つの側面から、基礎的かつ理論的な考察を加える。

・インライン展開によって不変な性質

・インライン展開の停止性

・最適化手法の適用順序

これらは、実際にオプティマイザを設計、開発するときに考察の必要性があったものである。次章で述べるPrologオプティマイザは、これらを確認した上で決定されたインプリメンテーションとなっている。また、ここに第２章で言及した述語呼出しの決定性に関する帰納的非可解性問題に対して、一つの証明とそれのいくつかの帰結を含める。

4.1　不変な性質

次の二つの命題は、第３章で述べたインライン展開の方法から容易にチェックされ得る。すなわち、命題１はインライン展開方法の直接的な帰結であり、またそれは述語定義体をインライン展開するために我々が設定した展開基準の一つでもあった。別の言い方をすると、命題１は、インライン展開に関する我々の意図を保証するものであると言える。命題２は、我々のインライン展開に依存する特有の性質ではなく、どのようなインライン展開によってでも成り立っていなければならない性質である。すなわち、インライン展開によって、述語が本来もっていた性質であるａ－決定性が壊されるようなことがあるならば、そのようなインライン展開は明らかに正当ではない。

命題　１．　述語定義体の型はインライン展開によって不変に保たれる。

命題　２．　ａ－決定性はインライン展開によって保存される。

4.2　インライン展開の停止性

述語呼出しが錯綜するPrologプログラムにおいて、述語呼出しの展開をできるだけ深く、そして停止性を確認できるのに十分な条件を設ける問題はそれほど自明ではない。節３．３．３で述べた直線型プログラム、終端再帰型プログラム、および一般の再帰型プログラムに対するインライン展開法は、単に展開の一つの方法を述べたのに止まらず、インライン展開の停止性のための一つの十分条件を強く意識して設定された。

ここで改めて、インライン展開の停止性を次の定理として表明しておくことにする。

定理　３．インライン展開は停止する。

証明．直線プログラムのインライン展開が停止することを言えば十分である。それは節３．３．３で与えた展開の条件(c)、「一度展開されたゴールはそれ以上展開しない」によって、一つのゴールから深さ優先で得られるインライン展開系列に同じ述語名のゴールは出現しないことが保証されている。このことによって、インライン展開の無限展開が避けられている。■

インライン展開の停止性を保証する十分条件をいかに拡げるべきかは、合理的な展開基準は何かという問題と絡んで、今後の重要な課題としなければならない。

4.3　局所的最適化手法の適用順序

複数ある最適化手法の適用順序は、それらのいくつかがプログラムに適用可能であるとき、重要になってくる〔Aho 72〕。適用順序がプログラムに対して適切でなければ十分に最適化されたプログラムが得られないことになる。

ここでは、インライン展開後のプログラムの冗長性を除去するためにいつも適用される局所的最適化変換手法である、dt1:【最適化変換図式　８-１】、dt2:【最適化変換図式　８-２】、du:【最適化変換図式　９】を取り上げ、その適用順序について論ずる。インライン展開の直後に必ずしも適用可能となるとは限らない他の最適化手法の適用順序を定める問題はまだ未解決である。これは複数個の最適化変換手法が与えられている状況下での単なる標準形の存在問題ではなく、最も効率の良い形式への変換問題であり、一般的に解決することはほとんど期待できない問題であろう。このような状況において、Prologオプティマイザを試作するに当たり我々が選択した方法は、５．２節で述べる。以下において，Γ，Δ，Πはゴールの列を表すものとする．

$$\mathrm{dt1:}\ \frac{\Gamma,\ \mathrm{true},\ \Delta}{\Gamma,\ \Delta} \qquad \mathrm{dt2:}\ \frac{\Gamma;\ \mathrm{fail};\ \Delta}{\Gamma;\ \Delta}$$

ここで、Γ、Δは非空である。

$$\mathrm{du:}\ \frac{\Gamma,\mathrm{fail},\Delta}{\Gamma,\mathrm{fail}}$$

ここで、Δは非空である。

dtを規則dt1 かdt2 のいずれかを表すものとし、下で定義される関係 dt-> は関係dt1 ∪ dt2 を表すものとする。ここで、∪ は関係の和を表す。

定義　1．　Γ dt-> Δ (Γ du-> Δ) <=> Γ、Δはそれぞれ最適化図式 dt (du)の上式、下式である。

定義　2．　関係＊は(dt -> ∪ du ->)$^+$、すなわち、関係 dt-> と du-> の和の推移的閉包を表す。

定義　3．　列Γは dt (du)に関して標準形である <=> dt (du) は列Γに適用可能ではない。列Γは dt 及び du に関して標準形である <=> dt、du のいずれもが列Γには適用可能ではない。列Γの dt 及び du に関する標準形をそれぞれΓ↓dt, Γ↓duと表す。

明らかに，

命題　4． duに関する標準形は一意に定まる。

命題　5． dtに関する標準形は一意に定まる。

命題　6． Γ↓du↓dtは標準形となる。

証明． 任意の列Γに対して、duがdtの適用の後にのみ適用可能になるということはないので，Γ↓duにある回数dtを適用すればそれ以上変形できない標準形が得られる．■

命題　7． Γ dt-> Πならば Γ↓du↓dt = Π↓du↓dt.

証明． Γ dt-> Πならば Γ↓du (dt->)* Π↓du という事実による。■

命題　8． Γ du-> Πならば Γ↓du↓dt = Π↓du↓dt.

証明． 明らか。■

定理　9．　任意の列Γに対して，Γ↓du↓dtはΓの＊に関するユニークな標準形を与える．

証明：　命題7，8より，Γ＊Πならば Γ↓du↓dt = Π↓du↓dt。特に，Γの任意の標準形Δについて，

$$\Delta = \Delta\downarrow du\downarrow dt = \Gamma\downarrow du\downarrow dt$$

となる。■

定理9は列を簡約化するとき、duを数回、次にdtを数回適用すれば十分であることを示している。

これらの結果に基づいて、冗長な述語 "true"、"fail" を除去するプログラム、及び不実行ゴール列の除去プログラムはそれぞれ(dt)^n (n ≧ 0)、(du)^m (m ≧ 0) として実現されている。そして、任意の列に対して、(du)^m (m ≧ 0) が最初に適用され、その次に(dt)^n (n ≧ 0) が適用されている。

4.4　決定性の決定不能性

「決定性」とか「非決定性」という言葉は、オートマトン・形式言語理論、

計算理論、プログラム言語理論及び言語の意味論・検証論などの計算機科学の広範囲において、他の科学の分野と同様しばしば現れる。これらの言葉の意味はそれぞれの分野毎に異なっているようであるが、決定性自身を判定する必要のある問題は少ない。

これまで、本質的に非決定的な性格をもつ手続きを表現することを可能にする非決定性プログラム言語は、いく人かの人によって研究されてきた。その中でも、Floyd〔Floyd 67〕、Dijkstra〔Dijkstra 76〕による言語、人工知能向き言語であるMicro-planner〔Sussman 71〕は、最近のProlog、Concurrent Prolog〔Shapiro 83〕に加えてよく知られた非決定性プログラム言語である。

Kowalskiは非決定性の概念を分類し、論じている〔Kowalski 79〕。それによると、非決定性には、一つのゴールに複数個の手続きがマッチする場合に係わる非決定性と、複数個のゴールが与えられた場合、達成すべき順序に係わる非決定性がある。上記の言語において現れる非決定性は前者の非決定性であり、それらはまた二つの主要な意味において非決定的である。すなわち、don't care(DC)とdon't know (DK) である〔Kowalski 79〕。PrologとMicro-planner はDK非決定性を実現し、Concurent PrologとDijkstraのガード付命令による言語はDC非決定性を実現している。Floyd の言語は両方の解釈をもち得る。

以下では、DK非決定性の決定問題が非可解であることを示すが、他方、DC非決定性の決定問題も決定不能であることが分かる。ここで、DC非決定性とは、非決定的な選択点において選択点が一意か否かということであり、例えば、Dijkstraのガードに基づく非決定性言語の非決定的if文において、どのガードが真となり選択されるかという問題は容易に一階論理の妥当性問題に帰着させることができるので、それは一般的には非可解である〔Church 36〕。

決定不能性問題は、ある適当な符号化とか、上のように他の決定不能な問題に帰着させることによって解かれることが多い〔Davis 65〕、〔Hopcroft 69〕。以下で与えられる決定性の帰納的非可解性の証明は、Russell によるCantorの素朴集合論に関するパラドックスのように、Prologプログラムによって二律背反命題を作ることによって与える。証明の前に、計算可能な関数は、ホーン節によって、またPrologによって計算可能であることを述べておく〔Tarnlund 77〕、〔Sebelik 82〕。

ここで、本論文における決定性の定義を思い起こしておこう。

定義 4．述語呼出し（ゴール）が決定的であるとは、述語呼出しがそれの述語定義体を呼んだとき、述語定義体の高々一つの節で成功し、バックトラックしたとき、述語呼出しは決して成功することがないことをいう。より単純化して言えば、述語呼出しか決定的であるとは、それが高々一回成功することを言う。

注意.

(i) この定義の下では、最初の実行において停止しない述語呼出しは決定的である。また、最初の実行では停止するが、バックトラックされたとき停止しなくなるような述語呼出しは決定的である。

(ii) Kowalskiは決定性の判定は原理的に決定不能であろうと述べている〔Kowalski 79〕。彼の決定性の定義は述語呼出しの関数性に関するものである。例えば、関係F(x,y)は、変数y が入力変数x の関数となっているとき決定的であると定義される。この定義の下では、決定性の決定問題は次のような一階論理の

論理式の妥当性判定問題に帰着させられる。

$$\text{Prog} \vdash \forall x,y,z\ (F(x,y) \wedge F(x,z) \supset y = z)$$

ここで、Progは述語F を規定するホーン節の集合である。一般に、一階論理の論理式の妥当性判定問題が決定不能であることが、Kowalskiの決定性の非可解性の根拠であるように考えられる。またより一般には、プログラムで扱うデータタイプには、自然数、リスト等が含まれ、そのようなデータ領域上の帰納法などを考慮に入れると、決定性を判定すべき論理式は二階以上になってしまう。我々の決定性の定義は、述語が関数的であるとする決定性の定義よりも一般的である。すなわち、述語のどの引数が入力でどの引数が出力であるかなどには関わりなく、述語呼出しの成功、失敗にのみ関わる定義である。そして、以下では、この一般的定義の下での決定性の非可解性を考えることにする。したがって、次の定理１０はPrologという一般的な枠組みでのKowalskiの表明の形式的正当化ともなる。

定理　１０．　任意の述語呼出しに対して、それが決定的であるか否かを判定するアルゴリズムは存在しない。

証明．次のような述語det のアルゴリズムが存在すると仮定する:
任意の述語呼出しP に対して、

det(P) = success,　if P : 決定的
　　fail , otherwise.

ここで、次のように定義されるプログラムを考える。

```
q :- det(q).
q.
```

すると、

(i) det(q) = success と仮定すると、q はバックトラックしたときdet(q)で再び

成功するか、さもなくば第二節で再び成功するので、決定的ではない。

(ii) det(q) = fail と仮定すると、q は第二節で成功するのみで、決定的である。

いずれにしても、矛盾となるので、detなるアルゴリズムは存在しない。 ■

注意.

(i) q のプログラムで第一節を "q :- det(q),!." とすると、上の(i)は単にdet(q) = successとなるだけである。 しかしながら、証明では矛盾を引き起こす例が一つでも作れてしまうので、detなるアルゴリズムの存在仮定は誤りであることを述べている。

(ii) 証明の方法は述語det をdet(P,Defs) のように二項述語として明示的に書いた場合でも同じである。ここで、Defsは述語呼出しP の決定性を調べるのに必要な定義体の集合である。証明では、det を明快さのために単項にして論じた。

(iii) この証明では，detのような自己参照をするメタ述語が用いられた．付録Ｊに，この証明より形式的な証明として対角線論法による別証明を与えた．

系　１１．　任意の述語呼出しに対して、それが非決定的であるか否かを判定するアルゴリズムは存在しない。

証明．　そのようなアルゴリズムが存在したとすれば、決定的述語呼出しの判定アルゴリズムも存在することになり定理１０に反する。 ■

系　１２．　任意の述語呼出しの解の個数を答えるアルゴリズムは存在しない。

証明．　そのようなアルゴリズムは特別な場合として決定的な述語呼出しの

解の個数を答えることになる。それは定理１０によって不可能である。■

系　１３．　任意の命題（変数を含まない）に対して、それが決定的であるか否かを判定するアルゴリズムは存在しない。

証明．定理１０の証明の中で述語呼出しpを命題pで置き換えればよい。■

系１３は命題のレベルにおいても述語呼出しの決定性の検出は原理的に不可能であることを述べている。

系　１４．　次のような述語det%のアルゴリズムが存在するものと仮定する：任意の停止する述語呼出しｐに対して、

det%(P) = success, if P : 決定的

fail, otherwise.

このとき、ある停止しない述語ｒが存在してdet%(r) は停止しない。

証明．次のプログラムを考える。

```
r :- det%(r).
r.
```

すると、プログラムｒは停止するか否かのいずれかである。停止すると仮定する。このとき、

(i) det%(r) = success と仮定すると、バックトラックしたときdet%(r) で再び成功するか、さもなくば第二節で再び成功するので、決定的でない。

(ii) det%(r) = failと仮定すると、ｒは第二節で成功するのみで、決定的である。いずれにしても、矛盾となるので、ｒは停止しない。このことは、ｒの定義より、det%(r) が停止しないことを含意する。■

第５章　PROLOGオプティマイザの試作

本章では、第３章で提案された各種最適化変換手法から構成され、また第４章で得られた理論的結果を考慮したPrologソースレベル・オプティマイザプログラムの基本構成と試作について述べる。

5.1　オプティマイザの基本構成

オプティマイザへの入力はDec system-10 Prolog〔Bowen 81〕で書かれたソースプログラムであり、入出力機能に加えてその主要機能は大きく次の三つからなっている。

① 最適化に必要な情報をプログラムテキストから抽出する機能

② 各種局所的最適化手法に対応する機能

③ インライン展開機能

①については第２章で述べた。②及び③は第３章で最適化変換の条件と共に述べた最適化変換図式を実現したものである。

オプティマイザの基本構成は図４のようになっており、ユーザからの指示に従って，直線，終端再帰，一般の再帰的プログラムの各インライン展開の前後に、各種局所的最適化手法を適用するというように進行する。これは大局的に見れば、まずPrologプログラムを展開することによって計算の無駄、冗長性等を浮き彫りにし、それを除去することによって最適化を図ろうとする方策の現れであり、代数式の簡約化のさいに取られる方策と類似するものである。

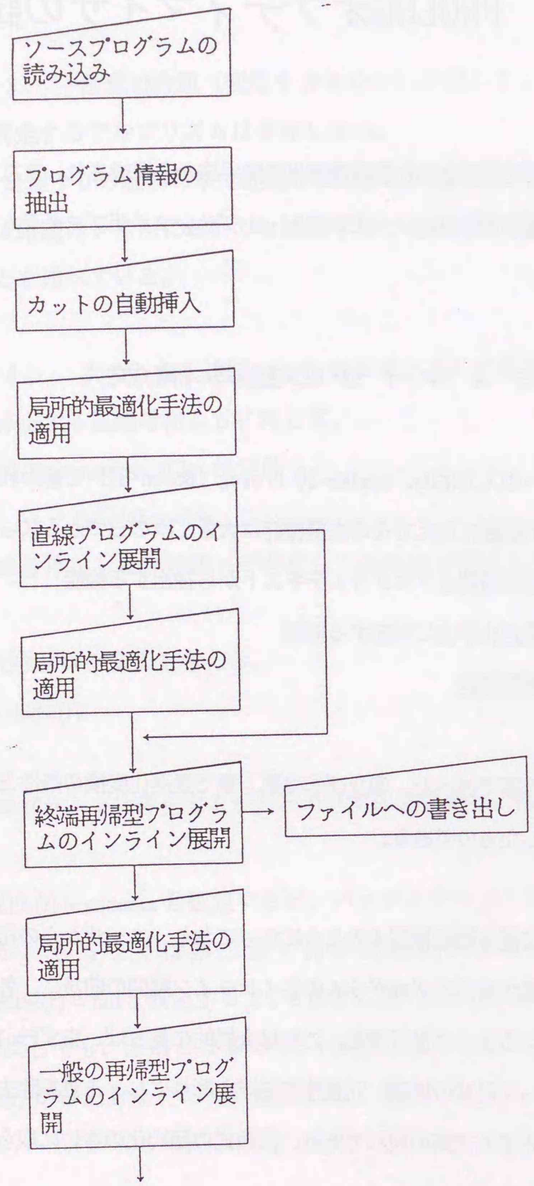


図4：Prologソースレベルオプティマイザの基本構成（次ページにつづく）

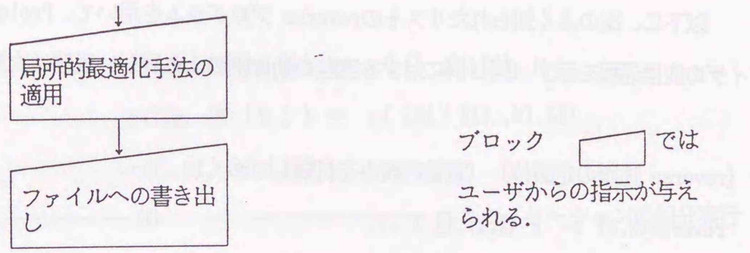


図4：Prologソースレベルオプティマイザの基本構成

以下に、次のよく知られたリストのreverse プログラムを用いて、Prologオプティマイザの変換過程を示す（同じ例に対する実際の動作例のより詳しい説明は次節で述べる）。

〔reverse 述語の定義体〕（宣言的読みを付随しておく）

```
reverse(X,Y) :-  r (X,Y, [] ) ,!.                    (1)
```

（Xを反転したものをY とするためには、X を反転したものとリスト [] を結合すればよい）

```
r( [] ,Z,Z) .                                        (2)
```

（空リストを反転し、Z と結合したものはZ である）

```
r( [H |T ] ,W,Z) :-  r (T,W, [H |Z ] ) .             (3)
```

（リスト [H |T] を反転し、Z と結合したものをW とするためには、T を反転しリスト [H |Z] に結合すればよい）

〔変換プロセス〕

① (1)は直線プログラムであるので展開の対象となるが,r (X,Y, []) の展開のときその定義節である(2), (3)は直線プログラムでないので展開されない。

② (2), (3)は終端再帰型プログラムであるので展開の対象となる。

最適化処理の流れは次のとおりである。


```
r( [] ,Z,Z) .
r( [H |T ] ,W,Z) :- r(T,W, [H |Z ] ) =  r( [] ,Z1,Z1) ;
                    r(T,W, [H |Z ] ) =  r( [H1 | T1] ,W1,Z1) ,
                    r(T1,W1, [H1 | Z1] ) .
```

——————————————————— ユニフィケーションの部分実行

```
r( [] ,Z,Z) .
r( [H |T ] ,W,Z) :- T = [] ,W=Z1,Z1 = [H |Z ] ;
                    T = [H1 | T1] ,W=W1,Z1 = [H |Z ] ,
                    r  (T1,W1, [H1 | Z1] ) .
```

——————————————————— 節の複数節への分解

```
r( [] ,Z,Z) .
r( [H |T ] ,W,Z) :- T = [] ,W=Z1,Z1 = [H |Z ] .
r( [H |T ] ,W,Z) :- T = [H1 | T1] ,W=Z1,Z1 = [H |Z ] ,
                    r  (T1,W1, [H1 | Z1  ) .
```

——————————————————— 等式代入

```
r( [] ,Z,Z) .                                        (4)
r( [H ] ,  [H |Z ] ,Z).                              (5)
r( [H | [H1 | T1] ] ,W1,Z) :- r(T1,W1, [H1 | [H |Z ] ] ) .   (6)
```

(1)～(3)から, (4)～(6)に至る一連の最適化において, インライン展開後の節に「節の複数節への分解」, 「等式代入」を適用して生成される節に注意されたい。すなわち、節の複数節への分解によって等式代入が可能になり、それによってボディ側のデータ構造が節のヘッド側に伝播された節が生成されている。結果として、節(2)が入力リストが単一の要素からなる場合（(5)）と二個以上の場合（(6)）へと分解されたために、与えられた入力リスト

の走査に必要な時間が節約されている。例えば、与えられたリストに(6)の定義節を適用するとリストの先頭から要素が二つずつ取られ、それらを反転したものがスタックに積まれていくことになる。

一般にインライン展開は前向きのデータ構造の伝播（forward data structure propagation 〔Komorowski 82〕）を促すが、等式代入規則は逆向きのデータ構造の伝播（backward data structure propagation 〔Komorowski 82〕）を引き起こす。

5.2 オプティマイザの試作

これまで述べてきた各種の最適化手法を組み込んだオプティマイザを試作した。各種最適化手法の大まかな適用順序は図４に示されているとおりである。適用順序が適切に定まらない局所的最適化手法については、ユーザがその都度指定するような一つのインプリメンテーションを試みた。将来、より適切な最適化順序が、多くの最適化プロセスと最適化例を通して定められたとき、完全自動化によるオプティマイザへの道が開かれていくかもしれない（unfold/fold 法に基づくプログラム変換の自動化に向けての研究については文献〔中川 85〕で行われている）

オプティマイザプログラムはVAX11/780/UNIX上のCProlog 及びDEC-system-10 Prologで書かれている。それは約60,000バイトの大きさである（付録Ａ～Ｉに、Prologオプティマイザプログラムの一部を掲げた）。

5．2．1 入力

オプティマイザへの入力はDec system-10 Prologで書かれたプログラムで、それは適

当なファイルに入っていなければならない。さらに、最適化の対象となるプログラムはすべて一つのファイルにまとめられていなければならない。すなわち、プログラムの中に実行時にファイルをコンサルト（ロード）するような述語が含まれていてはならない。また、このような述語以外のdirectives(“?-” で始まる質問、あるいは“:-”で始まるコマンド)及びコメントには何の作用も加えられない。

5．2．2 起動方法

Prologオプティマイザの起動は次のように行われる。

① Prologの起動

```
% Prolog
```

② オプティマイザ関係ファイルのコンサルト

```
?-  [Prolog-optimizer].
```

③ オプティマイザプログラムの実行

```
?-  Prolog-optimizer.
```

この後、オプティマイザは最適化変換のために必要とする指示を要求してくるので(プロンプト‘:’が出される)、５．３節で述べる指示を入力する。オプティマイザの処理が一段落するとオプティマイザプログラムは成功し、Prologは‘yes’を出力する。この後、Prologを終えるか、または上の③のように入力するならば再度最適化を続けることができる。

5．2．3　最適化変換処理手順

Prologソースレベルオプティマイザのおおよその処理の流れは図４に示した。オプティマイザは処理の進行にしたがって、最適化手法の適用を要求してくるのでその都度ユーザは以下の定められた記号で入力を行う。最適化変換過程の途中で得られるプログラムはディスプレイ上に書き出されるので、ユーザはそれを見てさらに最適化を進める、あるいは止めるかの判断を行う。また、それは指定されたファイルに書き出すことができる。このとき、ソースプログラム中のコメントは無視される。

オプティマイザは正しく書かれたPrologプログラムならば、すべて読み込み、最適化処理を施すが、ユーザプログラムに構文上の誤りがあった場合は、入力ファイルからの読み込み段階でProlog処理系からのエラーメッセージが出される。このときは、入力プログラムを訂正した後、再びオプティマイザを呼ぶことができる。

オプティマイザへの指示において、定められた記号以外の記号を投入した場合、エラーメッセージは出されないが、その記号は無視されて最適化が進行する。

〔最適化変換手法名〕

pu：　部分ユニフィケーション（3．1節）

dc：　連言列中の多重ゴールの除去（3．4．1節）

dd：　選言列中の多重ゴールの除去（3．4．1節）

dt：　冗長な‘true’, ‘false’ 述語の除去（3．4．2節）

du：　不実行ゴール列の除去（3．4．3節）

fa：　共通ゴール列のくくり出し（3．4．4節）

es：　等式代入による冗長変数の除去（3．5節）

ig：　ゴールの統合化（3．6節）

rd：　節の複数節への分解（3．7節）

Prologオプティマイザの処理の流れを次ページの図５に示す。

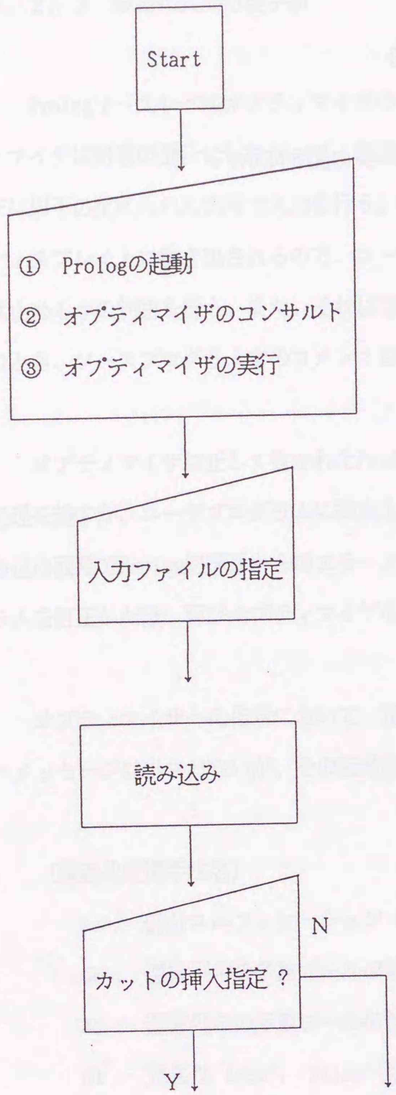


図5 : Prologオプティマイザの処理の流れ（つづく）

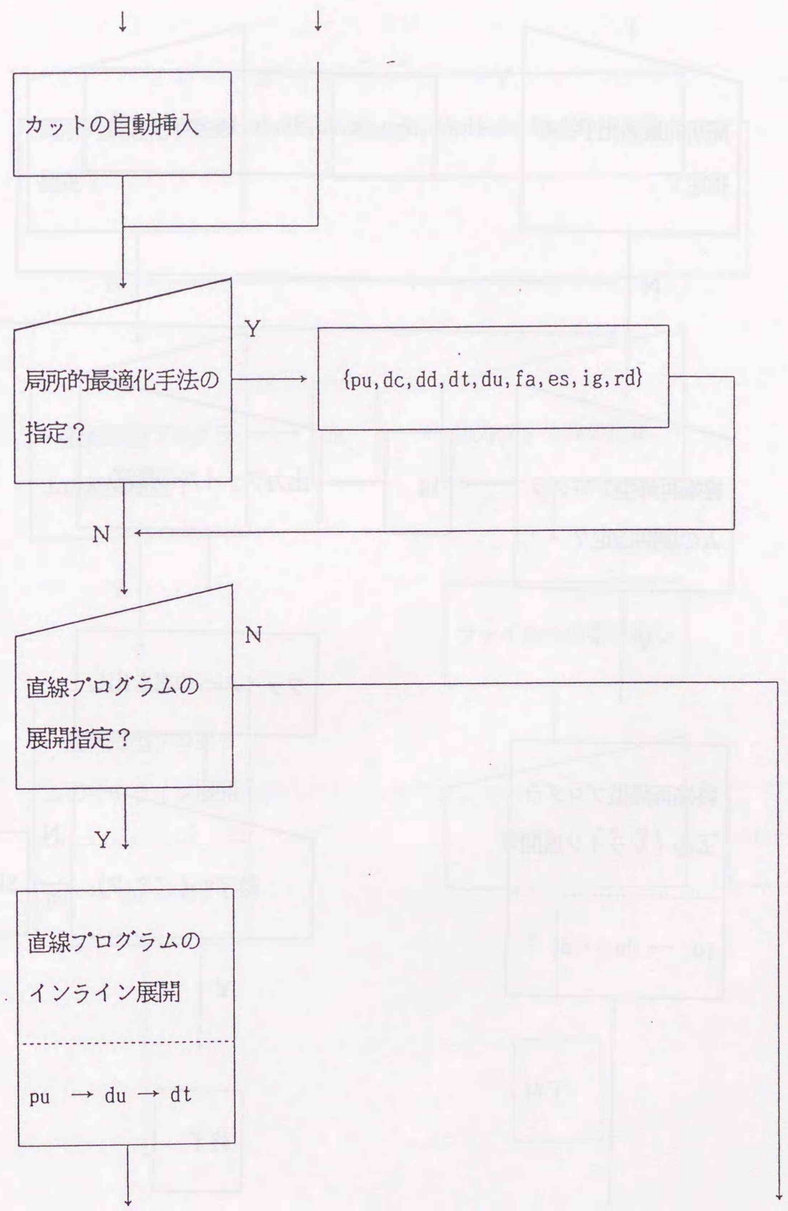


図5 : Prologオプティマイザの処理の流れ（つづく）

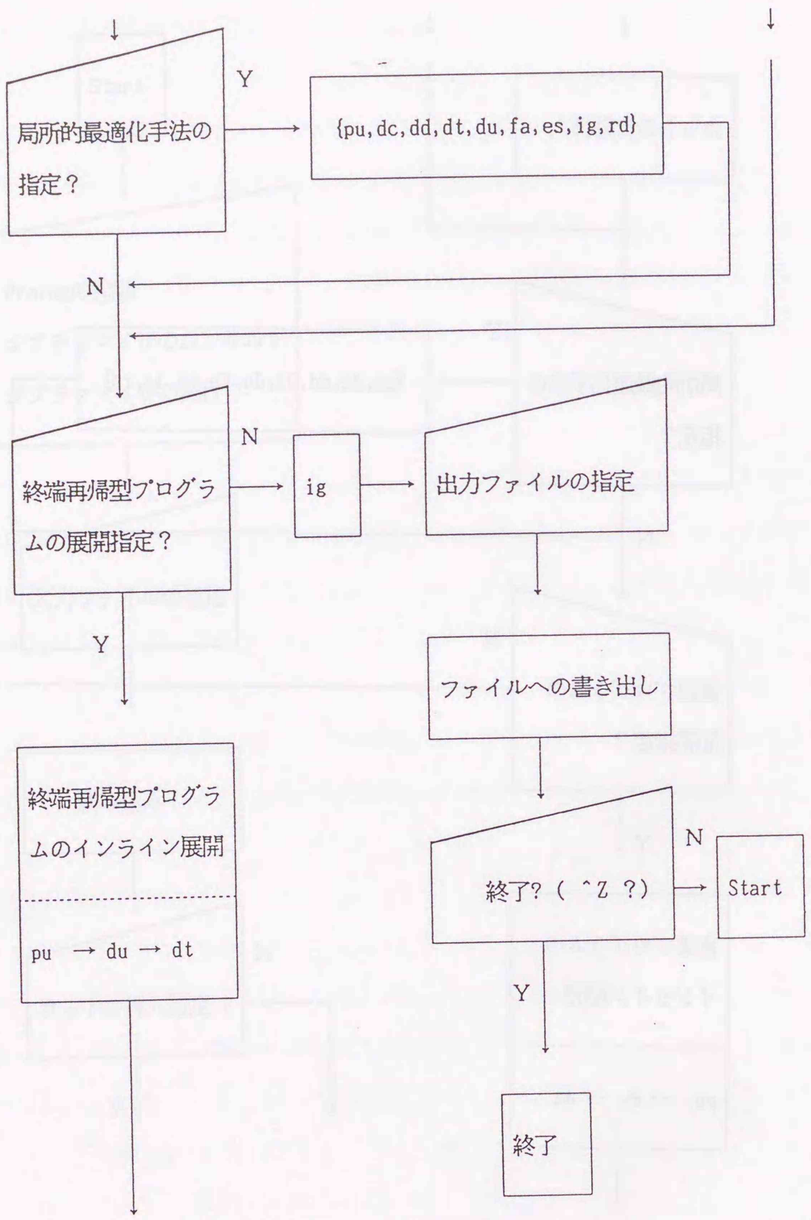


図5：Prologオプティマイザの処理の流れ（つづく）

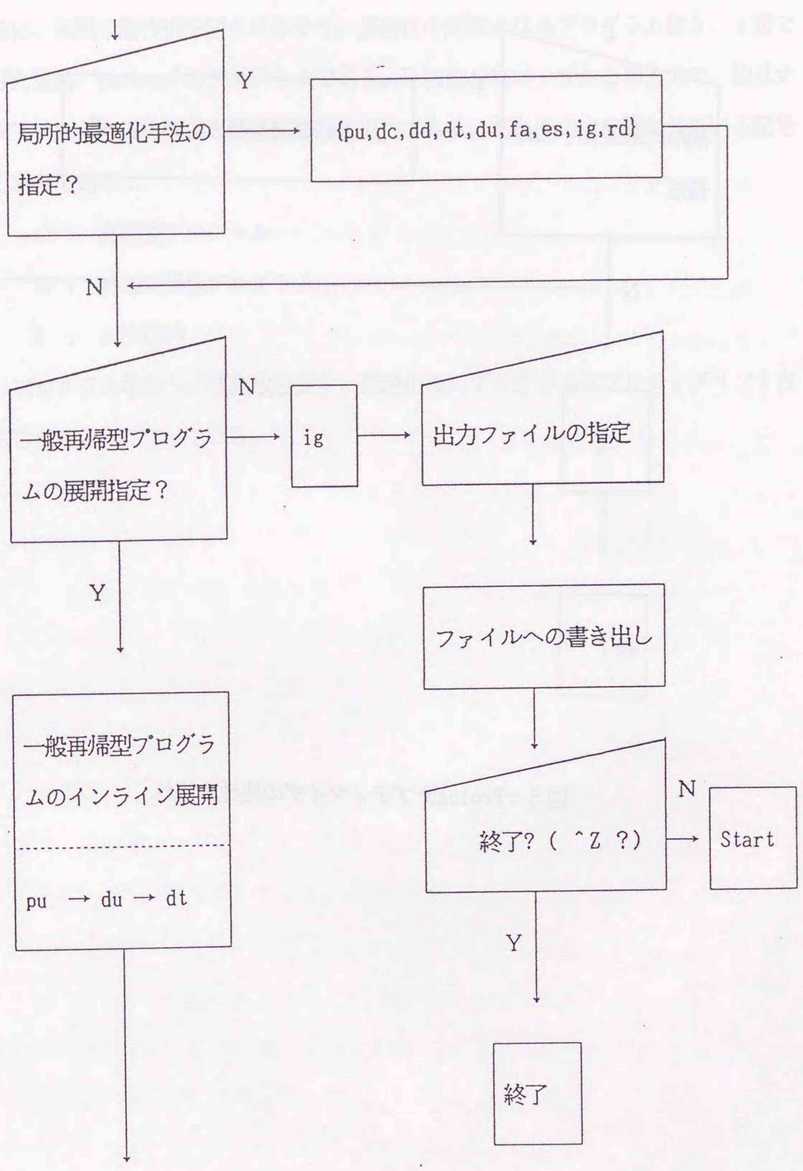


図5：Prologオプティマイザの処理の流れ（つづく）

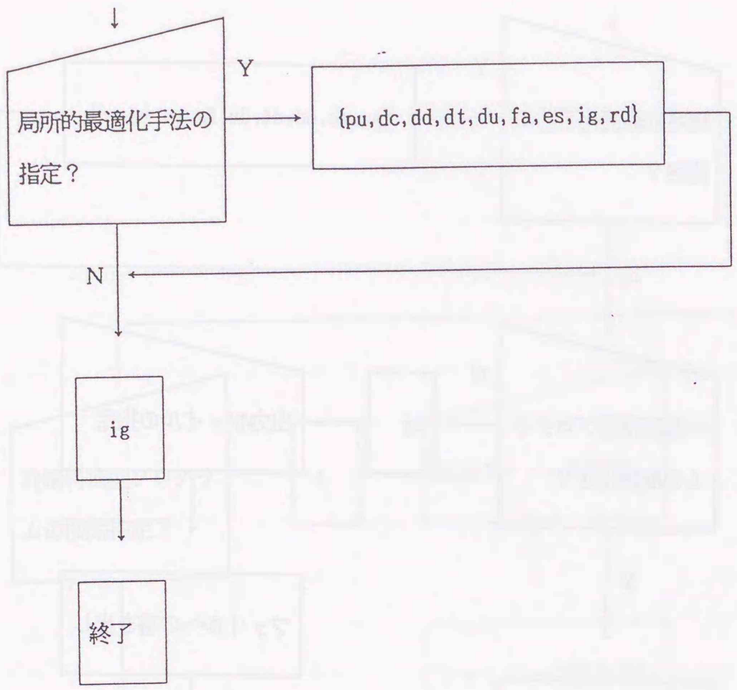


図5：Prologオプティマイザの処理の流れ

最後に、実際の動作例を図6に示そう。最適化の対象となるプログラムは5．1節で取り上げた述語‘reverse’のプログラムである。下線部分はユーザからの入力で、他はオプティマイザからの入力要求と最適化結果の出力である。またその中で使われている記号は次のとおりである：

sl ： 直線型プログラム

tr ： 終端再帰型プログラム

d ： a-決定的

さらに、プログラム中の‘-n’は変数を表す。最適化後のプログラムはプリティプリントされて出力されるようになっている。

```
| ?- prolog-optimizer.
------- PROLOG OPTIMIZER (version 1) starts -------
Source program file name to be optimized : reverse
Automatic cut-insertion ? (y or n) : n
Local optimizations ? (y or n) : n
Straight-line expansion ? (y or n) : n
Tail-recursive expansion ? (y or n) : y

 [reverse/2/(s1/d),r/3/(tr/-432)]


reverse/2/(s1/d)
    reverse(-200,-201) :-
            (r(-200,-201,  [] ),!).


r/3/(tr/-432)
    r( [] ,-243,-243).
    r( [-273 | —274 ] ,-279,-280) :-
            ((-664=  [-273 | -280] ,
              -274=  [] ,
              -279=  [-273 | -280] );
             (-984=  [-273 | -280] ,
              -279=-983,
              -274=  [-981 | -982] ,
              r(-982,-983-,  [-981 | -984] ))).
```

図6 ：Prologオプティマイザの動作例（次ページに続く）


```
Tail-recursive inline expansion completed.
Local optimizations to the resulting program ? (y or n) : y
Type in one of the following : rd., es., or [rd,es ] . :  [rd,es ] .
 [rd,es ] applied.
General recursive inline expansion ? (y or n) : n
Your Prolog program has been optimized as follows :

 [reverse/2/(s1/d),r/3/(tr/-432)]


reverse/2/(s1/d)
    reverse(-200,-201) :-
            (r(-200,-201,  [] ),!).


r/3/(tr/-432)
    r( [] ,-243,-243).
    r( [-273] ,  [-273 | -280] ,-280).
    r( [-273,-981  | -982] ,-983,-280) :-
            (r(-982,-983,  [-981,-273  | -280] )).

Output file name : reversel
Optimized programs written to the file : reversel
PROLOG OPTIMIZER ends.


yes
```

図6 ：Prologオプティマイザの動作例

第6章　最適化例とその評価

ここでは、これまで提案してきた各種の最適化手法を一つに統合化した、トータルシステムとしてのPrologオプティマイザの評価についてその検討結果を述べる。

各種最適化手法の詳細な適用順序を図5に示した。適用順序が適切に定まらない局所的最適化手法については、ユーザがその都度指定するような一つのインプリメンテーションであることはすでに述べたとおりである。

本論文の最適化手法は、大局的には、インライン展開されたプログラムテキストに対して局所的最適化を図るというものであった。他方、期待される効果という観点から一つ一つの最適化手法をみると、それらは次のような効果をもつものとして特徴づけられる：

（i）プログラムテキスト上で計算をまえもって数歩進めることによって計算の手間を省く最適化手法。
（ii）冗長性の除去によって計算の手間を省く最適化手法。
（iii）達成すべきゴールの数を減らすことによって計算の手間を省く最適化手法。
（iv）他の最適化の手法の適用を可能とするための最適化手法。

これらは全て実行マシンとか、Prologの処理系を意識しない、素朴な最適化手法ばかりであった。

以下では、現在世の中で最も多く使われ、Prolog言語のインプリメント法として定着

しているDec system-10 Prologのインタプリタ〔Bowen 81〕とCProlog インタプリタの下で行われた、最適化後のPrologプログラムの時間評価について述べる。時間計測に用いたプログラムは、Dec system-10 Prologの場合は次のPrologプログラムである〔国藤 83〕。CProlog の場合も類似のプログラムである。

```
open-timer :- statistics(runtime,-).
close-timer(T) :- statistics(runtime,  [-,T ] ).
do(1,G) :- G,!.
do(N,G) :- N1 is N-1,not(not(G)),do(N1,G).
average-time(N,G) :-
        open-timer, do(N,G),close-timer(T),
        display(T),display(' ms.'),nl,
        D is T/N,display(D),display(' ms.').
```

〔1〕リストの要素を反転させる述語'reverse'の最適化

このプログラムの最適化変換については第5章で詳しく述べた。最適化後のプログラムに対して、次の呼出しを500 回続けて実行した時間計測値を以下に表す。

```
reverse( [a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z ] ,Result).
```

これから、最適化後のプログラムに対しては、約20% の時間効率が得られることがわかる。

(単位は秒)

	1	2	3	4	5	6	7	8
最適化前	10.28	10.75	10.37	9.97	9.88	10.03	10.30	10.08
最適化後	8.07	8.42	8.22	8.27	8.07	8.08	8.17	8.07

9	10	平均	時間比（最適化後／最適化前）
9.90	9.77	10.13	0.80
8.07	7.70	8.11	

これは主として、最適化後のプログラムが、結果的にはリストの先頭から二つの要素を取りそれを反転させるプログラムとなったことによるものと考えられる。このようなプログラムが生成されることを我々は意図していなかったが、展開によってプログラムの型を不変に保つ限り、終端再帰プログラムの終端ゴールを一回展開するという我々のインライン展開の方法によって、結果的にそのようなプログラムが生成されることになった。これは、インライン展開が本来計算を一歩進める働きをもつものであるから当然の結果であるともいえる。

〔2〕局所的最適化手法プログラム自身の最適化

次のような最適化図式〔Chikayama 83〕を実現するプログラムを最適化する。

```
p(A) :- Γ,X = f(t1,...,tn),X = f(r1,...,rn), Δ.
```

```
p(A) :- Γ,f(t1,...,tn) = f(r1,...,rn), Δ.
```

ただし、変数X はp(A)、Γ、Δの部分に出現してはならない。

次のプログラムは、入力ファイルより節を読み込み、上記の最適化を施した結果の節を出力ファイルへ書き出すというものである。

```
optimize-1(IF,OF) :-
  see(IF),tell(OF),repeat,read(T),transform(T),seen,told.
transform('end-of-file') :- !.
transform(T) :-
  fold(T,R),write(R),write('.'),nl,!,fail.
fold((H:-G),(H:-R)) :- red(H,G,R).
red(H,(X=T1,Y=T2),(T1=T2)) :-
  var(X),X==Y,notoccur(X, [H ] ).
red(H,((X=T1),((Y=T2),T)),R) :-
  var(X),X==Y,notoccur(X, [H,T ] ),red(H,((T1=T2),T),R).
red(H,(A,T),(A,Z)) :- red( [H,A ] ,T,Z).
red(H,A,A).
```

```
notoccur(X,T) :- var(T),!,not(X==T).
notoccur(X,T) :- T=.. [F |As] ,mapnc(X,As).
mapnc(X, [] ).
mapnc(X, [H |T ] ) :- notoccur(X,H),mapnc(X,T).
```

述語 "transform " の中のゴール "fold(T,R)"は直線プログラムであるのでインライン展開される。また、述語定義 "red " は終端再帰型プログラムであるので、その終端ゴールは一回展開される。

このような最適化後のプログラムに対して、次の呼出しを続けて30回実行すると以下の時間計測値が得られた。

```
optimize-1(data,user).
```

ただし、ファイル'data'には次の節が入っている:

```
p(X) :- Y=f(Z),Y=g(c).
p(X,Y) :- q(Z,a),S=f(X,b),S=g(Y),q(Y,c).
p(X) :- q(Z,a),S=f(X,b),S=g(Y).
p(X) :- q(Z,a),X=f(X,b),X=g(Y,a).
p(Y) :- q(Z,a),X=f(Y,b,c),X=g(a),X=h(Z,Y),r(Z).
p(Y,Z) :- q(Y),X=f(Z),X=f(a).
```

このような最適化後のプログラムに対して、約20% の時間効率を得ている。この結果も、インライン展開の効果によるものである。

(単位は秒)

	1	2	3	4	5	6	7	8	9	10	平均	時間比 (最適化後／最適化前)
最適化前	12.05	12.13	11.85	12.37	12.57	10.03	13.08	12.72	12.17	11.70	12.35	
最適化後	9.82	10.02	9.81	10.00	8.07	9.80	10.18	10.05	10.22	9.98	9.99	0.81

〔3〕ESP コードの最適化

ESP プログラム〔Chikayama 84〕はKLO 〔Chikayama 83〕 (あるいは、Prolog) へ翻訳される。ここでは、簡単なESP プログラムのPrologコードを最適化し、その評価結果を記す。

次のような簡単なESP プログラムを取り上げる。

```
class lh has

    :append(-,X,Y,Z) :- append(X,Y,Z) ;

  local

    append( [] ,X,X) ;

    append( [W |X ] ,Y, [W |Z ] ) :- append(X,Y,Z) ;

end.
```

このプログラムの中間コードは次のとおりである。


```
:- public 'lh$c$p$append$4'/4.
:- mode 'lh$c$p$append$4'(+,?,?,?).
'lh$c$p$append$4'(A,B,C,D) :- 'lh$l$$append$3'(B,C,D).

'lh$l$$append$3'( [] A,A) :- true.
'lh$l$$append$3'( [A |B ] ,C, [A |D ] ) :- 'lh$l$$append$3'(B,C,D).
:- public 'lh$i$t$$3'/3.
:- mode 'lh$i$t$$3'(+,+,+).
:- public 'lh$c$t$$3'/3.
:- mode 'lh$c$t$$3'(+,+,+).
:- mode 'lh$c$m$new$2'(+,?).
'lh$$c$m$new$2'(A,B) :- 'lh$c$p$new$2'(A,B).

'lh$c$t$$3'(new,A,args(B)) :- !,'lh$c$m$new$2'(A,B).
'lh$c$p$new$2'(A,B) :- new-object(B,1,'lh$i$t$$3').

:- mode 'lh$c$m$append$4'(+,?,?,?).
'lh$c$m$append$4'(A,B,C,D) :- 'lh$c$p$append$4'(A,B,C,D).

'lh$c$t$$3'(append,A,args(B,C,D)) :- !,'lh$c$m$append$4'(A,B,C,D).

'lh$c$t$$3'(append,A,args(B,C,D)) :- !,'lh$c$m$append$4'(A,B,C,D).
'lh$i$t$$3'(is-class,A,args) :- !,fail.
'lh$i$t$$3'(class-object,A,args(B)) :- !,get-class-object(lh,B).
'lh$i$t$$3'(slots,A,args( [] )) :-!.
'lh$i$t$$3'(A,B,C) :- !,udm-error(A,B,C).
'lh$c$t$$3'(is-class,A,args) :- !.
'lh$c$t$$3'(class-name,A,args(lh)) :-!.
'lh$c$t$$3'(slots,A,args( [] )) :-!.
'lh$c$t$$3'(A,B,C) :- !,udm-error(A,B,C).
```

このプログラムの最適化を行うと、線で結ばれた部分の直線プログラムがインライン展開される。この結果のプログラムに対して、次の呼出しを続けて500 回実行すると以下の時間計測値が得られる。これから、最適化後のプログラムに対して、約36% の時間効率が得られている。

'lh$c$t$$3'(append,lh,args([a,b,c,d,e,f,g,h,i] , [j,k,l,m,n,o,p,q] ,R)).

(単位は秒)

	1	2	3	4	5	6	7	8
最適化前	6.07	5.93	6.15	6.05	6.28	6.20	6.53	6.25
最適化後	4.52	4.48	4.55	4.82	4.57	4.57	4.70	4.80

9	10	平均	時間比 (最適化後／最適化前)
6.65	6.30	6.24	0.74
4.62	4.55	4.62	

〔4〕 Prologプログラム変換へのオプティマイザの利用例とその評価

ここでは、オプティマイザがPrologプログラムの変換にどのように利用され得るかを、簡単な例を用いて示すことにする。Prologプログラムの変換については〔Sato 84 〕等の論文で試みられているが、ここでは、いままで提案してきたオプティマイザのさまざまな機能の下で、いかなるプログラム変換が可能であるかを示すのが目的である。

次のようにリストが抽象データ型として定義されているとする（このプログラム例は文献〔Venken 84 〕より取られた）。

```
null( [] ).
cons(-e,-list, [-e | -list ] ).
append(-in,-out,-out) :- null(-in).
append(-in,-part,-out) :-
        cons(-e,-list1,-in),
        append(-list1,-part,-list2),
        cons(-e,-list2,-out).
```

このとき、このappendを用いて二つのリストを結合するのは余り効率的ではないが、これをオプティマイザに与えると、

```
null( [] ).
cons(-e,-list, [-e | -list ] ).
append( [] ,-out,-out).
append( [-e | -in ] ,-part, [-e | -out] ) :- append(-in,-part,-out).
```

を得る。これは我々が通常抽象データ型を考えないとき書くappendの定義であり、これは上のプログラムと同値である。

この二つのappendに対して、次の呼出しを500 回続けて実行すると、以下の時間計測値が得られた。

append([a,b,c,d,e,f,g,h,i,j,k,l,m] , [n,o,p,q,r,s,t,u,v,w,x,y,z] ,R).

これから、変換後の'append'プログラムは約60% の時間効率を達成していることが分かる。

(単位は秒)

	1	2	3	4	5	6	7	8	9	10	平均	時間比（最適化後／最適化前）
最適化前	10.52	10.63	10.65	10.57	10.62	11.23	11.21	11.55	10.18	10.31	10.75	0.41
最適化後	4.63	4.58	3.93	4.12	4.23	4.35	4.90	4.72	4.35	4.15	4.40	

以上、いくつかの小規模プログラムに対してオプティマイザの評価を行った。この段階で、これまで取り上げたようなプログラムに対しては、少なくとも次のようにまとめることができる。

『Prologプログラマが通常書くPrologプログラムに対しては平均的に約20% の時間効率が達成され、明らかに冗長なプログラムでは40～60% の時間効率が得られる』

Prolog プログラムは、一般にはこれまで取り上げたような小規模プログラムの集合体であるので、このような小規模プログラムの時間評価でも、大規模プログラムに対して我々のオプティマイザの有効性を確信させることができるであろう。しかしながら、この結果が、任意のPrologプログラムに対して、上と同程度までの時間効率に結びつくというわけではないであろう。ベンチマークテストはあくまで我々のオプティマイザの基本性能を見る一つの目安を与えるものであると考えるのが妥当であろう。

本論文では、Prologオプティマイザの評価を、実際のマシン上の言語処理系の下で定量的に論じてきた。これは評価の客観性という面では少し弱い評価であると思われるかもしれない。実行マシン上の言語処理系に依存しない定量的評価が望まれるところであろう。しかしながら、そのためには共通に認められる、抽象的なPrologインタプリタの定義が必要となる。現在のところ、我々はそのような形式的な定義をもつに至っていない。したがって、これは今後の課題として残される。

他方、我々の最適化手法を計算の手間（複雑さ）という点から観察し、導出規則(resolution rule)を１ステップと数えるならば、明らかにそれらの最適化手法は数ステップの計算（推論）ステップを節約するのに役立っている。しかしながら、この評価方法では、ユニフィケーションとか節の探索に要する時間などは無視されていることはいうまでもない。

これまで、プログラムの時間効率のみを考察してきたが、最後に我々のインライン展開の空間効率に与える影響について触れておこう。一般に、Prologプログラムのような述

語（手続き）の並びからなるような言語では、インライン展開はテキストの脹らみを膨大にしてしまうように予想されるが、実際には我々の述語（呼び出し）の決定性条件がプログラムの展開を適度に制限し、結果として空間効率に悪い影響を与えることはなかった。むしろこの意味で、述語の決定性は空間効率のための有効な基準となり得る条件かも知れない。

第７章　汎用論証支援システム EUODHILOS

第５章で，Prologプログラムオプティアイザの構成について述べた．それはPrologプログラムの変換規則が固定されたものであり，変換規則の追加／修正に当たっては，オプティマイザプログラムの大きな変更が要求され，柔軟なシステムとは言えない．そこで本章では拡張可能なPrologプログラムオプティマイザに向けて，汎用論証支援システムEUODHILOSのプログラム変換への利用可能性を検討する．７．１節で，EUODHILOSの機能的特徴を述べ，７．２節で，EUODHILOSの論理表現能力を確かめるため，様々な論理へEUODHILOSを適用した結果を示す．７．３節では，プログラム変換のような記号操作にもEUODHILOSの汎用性が有用であることを述べる．なお，本章の内容は，論文〔Sawamura 90, Sawamura 91, Sawamura 92, 沢村 93〕およびそこで引用された論文に基づいている．

7.1　汎用論証支援システムEUODHILOSの機能的特徴

対象を論理学的に認識，表現し，対象の論理操作によって問題解決を計るといった論理学的方法論は，今日，計算機科学や人工知能の研究における一つの有力なパラダイムとなっている．これまでこのような論理学的方法論を支援するために，論理系が固定されている特定目的のための証明支援システムについて多くの研究がなされてきた．

このような研究動向とは対照的に，我々はそれぞれの問題領域毎により適した論理系を定義することを許し，さらに定義された論理系の下で論証による問題解決を可能にする汎用論証支援システムEUODHILOS (Every Universe Of Discourse Has Its LOgical Structure)を提案してきた〔Sawamura 90, 91〕．別の言い方をするならば，これは論理に依存しない証明エディタである．

EUODHILOSは，全ての対象領域はそれ固有の論理系をもつという認識に基づき，論理系が固定されている証明支援システムの場合に起こる次の問題を克服することを目的としている．

(1) 論理系を固定しているシステムは，他の論理系を必要とする対象領域には一般には使えない．使えたとしても，対象世界をそのシステムが扱える表現に変換しなければならない．そのさい，その表現は元の表現より不自然であったり長く複雑になることが多い．

(2) 各対象領域毎にその都度論証システムを構築することになれば，そのために多大の労力を必要とするのみならず，類似の作業を繰り返し行うことになり無駄も労力も多くなる．

実際これまで，EUODHILOSの汎用性によってさまざまな論理系を扱うことができた〔Sawamura 93〕．

汎用性をもつ論証支援システムを構築するには，さまざまな論理に共通するもの，および各論理がもつ固有の特異性という相反する側面を捉えるための枠組みが必要になる．しかしながら，これに正面から答えるには，「そもそも論理とは何か？」という大変基本的な問題に行き当たってしまうことになる．そこで，EUODHILOSでは，これまでよく知られた論理に

共通する基本的な特徴を集め論理記述の枠組みを考えるという実用的なアプローチを取っている．

この節では，我々が取った汎用論証支援システムEUODHILOSの代表的な諸機能を概観する．EUODHILOSを設計するに当たっては，次の三つのポイントが重視された：

(i) 記述性が高く，扱いやすい論理記述のための枠組み，
(ii) 強力で柔軟性のある証明構築支援機能，
(iii) 使いやすく，論証に向いたインタフェース．

(1)と(2)は汎用システムにとっては本質的な部分である．(3)はこれまでの定理証明機や論証支援システムでは軽視されてきたが，対話型の論証支援システムの証明構成機能とも絡む重要な部分として設計の初期段階から考慮に入れられてきた．

論理系は一般に，言語系（記号，式，論理式など）と導出系（公理，推論規則・書き換え規則・派生規則など）からなる．以下これらを記述する方法について説明する．

（1）言語系の記述

演算子優先順位付き確定節文法

EUODHILOSでは，ユーザの論理の言語は，論理に固有の概念（変数の束縛・スコープ，代入，スキーマ変数）の記述を可能にするために拡張された確定節文法(DCG)で記述される（具体的な定義例は次節で例示される）．

このさい，特種な論理記号などは，フォントエディタで設計し，各論理

毎に準備される仮想キーボードに割り付けることができる．以下に，直観主義的型の理論の言語定義の例を掲げておく〔Sawamura 90〕．

```
SYNTAX : Intuitionistic_type_theory
save  make  test  structure  print  reshape  exit

% Meta_language
meta_term --> meta_term1;
meta_type -> "A" | "B";
meta_term1 --> "F" | meta-const | meta_variable;
meta_const --> "a" | "b";
meta_variable -> "X" | "f";

% Object_language
judgement --> term,"∈ ",type;

term --> bind_op,variable,".",term|
         term,"•",term|
         "(",term,")"|
         "~",term|
         'inl","(",term,")"|"inr","(",term,")"|
         variable|constant|
         meta_term1,"(",term,")"|meta_term;

type --> type,"⊃",type|
         type,"∨",type|
         "~",type|
         "(",type,")"|
          basic_type;

variable --> "x"|"f";
constant --> "c"|"d";
basic_type --> "p"|"⊥";
bind_op --> "λ";

% Interface between meta and object languages
type --> meta_type;
variable --> meta_variable;

operator
  "~"; "∨":left; "⊃":left; "•":left; "λ"; "∈ ";
```


パーザ・アンパーザの自動生成

このような文法に対して，式の内部構造を自動的に生成する機構，パーザ・アンパーザの自動生成アルゴリズムが考案され，EUODHILOSの言語定義部で有効に使われている〔Sawamura 90, 91〕．

構文チェッカ・論理式エディタ

またユーザの意図した言語を効率よく定義することを可能にするため，定義した構文をテストできる構文チェッカや，演算子の強さなどが正しく定義されたかなどを視覚的にチェックできる論理式エディタも準備されている．これらは言語定義のさいのユーザ負担を軽減するのに役立っている．

(2) 導出系の記述

推論規則の記述に対しては次の二つを考慮に入れることが必要となる：規則の適用条件の自動チェックと結論の仮定に対する依存性の管理．

公理

推論の出発点となる公理は，論理式のリストとして（名前付きで）与えられる．

推論規則

推論規則は次のように，仮定，前提，結論，適用条件の四つから自然演繹法スタイルで記述される．

$$\frac{\begin{array}{cccc}[\text{仮定}1] & [\text{仮定}2] & \cdots & [\text{仮定}n] \\ \vdots & \vdots & & \vdots \\ \text{前提}1 & \text{前提}2 & \cdots & \text{前提}n\end{array}}{\text{結論}}\ \text{適用条件}$$

適用条件とその自動チェック

EUODHILOSでは，適用条件は多くの論理にしばしば現われる次の三つの形式の基本条件とその組み合わせによって与えられる：

(1) t is free for x in P（代入条件），

(2) x is not free in P（変数自由出現条件），

(3) a is an eigenvariable（固有変数条件）．

そして，これらの条件は証明の過程で自動的にチェックされることになる．この方法で扱うことのできない他の条件に対しては，ユーザが書いた適用条件チェッカを組み込むためのインタフェースが用意されている．以下に，直観主義的型の理論の推論規則定義の例を上げる〔Sawamura 90〕．

INFERENCE RULE: Intuitionistic_type_theory
name: λ–I
$\dfrac{\begin{array}{c}[X \in A] \\ \vdots \\ F(X) \in B\end{array}}{\lambda X.F(X) \in A \supset B}$
** side condition** X is not free in B

INFERENCE RULE: Intuitionistic_type_theory
name: λ–E
$\dfrac{a \in A \quad F \in A \supset B}{F \bullet a \in B}$
** side condition**

INFERENCE RULE: Intuitionistic_type_theory
name: inl-I
$\dfrac{a \in A}{inl(a) \in A \vee B}$
** side condition**

INFERENCE RULE: Intuitionistic_type_theory
name: inr-I
$\dfrac{b \in B}{inr(b) \in A \vee B}$
** side condition**

依存性(dependency)

EUODHILOSでは，通常の自然演繹法における結論の仮定に対する依存性は，仮定に対して与えられた自然数をもとに，集合計算で自動的に管理されている．集合計算でなく他の管理方法が必要とされる論理に対しては，EUODHILOSではタグ（あるいはラベル）付き論理式と，タグを操作するための書き換え規則を定義することによって扱うことができる．このアイディアは元々Meyer, Gabbayらによって適切論理などの非標準論理を形式化するさいに導入されたものであるが，汎用システムにおいて依存性を一般的に定義するのにも大変有用である〔沢村 93〕．例えば，次の適切論理の∧-I規則

$$\frac{P^{\alpha} \quad Q^{\alpha}}{(P \wedge Q)^{\alpha}}$$

は，P, Qが依存する仮定が同一の時にのみ（直観的には，同じ根拠の下でP, Qが得られたとき）推論が可能となるものであり，このような依存性は通常の自然演繹法では取り扱うことができない．EUODHILOSでは，論理式をタグ付きのメタ論理式として表し，さらにαの部分の計算を書き換え規則で与えるものとして，次のように表現して取り扱い可能とした．

$$\frac{\alpha \Rightarrow P \quad \alpha \Rightarrow Q}{\alpha \Rightarrow P \wedge Q}$$

書き換え規則

書き換え規則は，書き換え前と後の式で定義される．EUODHILOSでは書き換え規則を自動的に何度も適用でき，その回数の上限をユーザが指定できるようになっている．以下の例は，直観主義的型の理論の書き換え規則の一つである〔Sawamura 90〕．

REWRITING_RULE: Intuitionistic_type_theory
name: def
$A \supset \bot$ ———   ~A

派生規則

派生規則はそれが以下で述べる思考シート上でその証明が与えられたとき定義可能となる．

EUODHILOSの以上の論理記述枠組みの下では，Hilbert型，Gentzen型

(NDおよびLK, LJ)，等式型の公理系の記述が可能であることは明らかである．また，いくらか不自然になることを除けば，Tableau型の定義も可能であることが分かる．その他のよく知られている公理系の中で，Fitch型の公理系については，我々の枠組では今のところ定義不可能である．

まとめると，これまで述べてきたEUODHILOSの論理記述法は次のように特徴づけられる：

(i) 対象論理は，その証明論的性格を直接反映するような形で記述される．
(ii) 対象論理は，通常の論理の教科書に現われるような形で記述される．

その結果，EUODHILOSの方法は広範囲のユーザにも自然で取り扱いやすいのが大きな特徴であると言える．

(3) 証明構築支援環境と機能

これまでの汎用論証システムでは，定義された論理の下での証明を支援する証明構築方法には未だほどんど関心が払われておらず，トップダウンあるいは後ろ向きに証明を構成していく方法が趨勢であった．しかしながら，EUODHILOSでは論理の形式的証明を自然にかつ柔軟に，さらに効率よく支援する環境と方法が用意されている．EUODHILOSの証明は以下の証明構築機能／環境を用いて対話的に進行する．もちろん，自動定理証明機とは異なり証明の主体は，ユーザである．

思考シート

思考シートとは，証明断片からのパターンマッチング／ユニフィケーシ

ョンによる証明の結合，分離などの操作や，下記の証明方法での証明が行なえる場，あるいは環境である．

証明方法

EUODHILOSでは次のようなさまざまなスタイルでの証明方法が可能である：(i)前向／後ろ向き証明，(i)証明の結合によるこれらの組み合わせ，(iii)すでに得られたレンマ，派生規則を利用する証明，(iv)メタ変数を用いる図式証明．

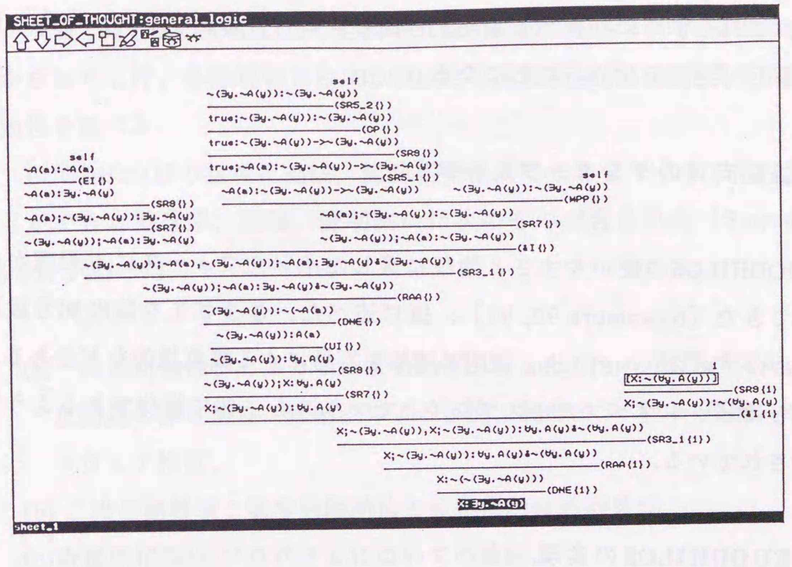
証明木の形式

証明は思考シートと呼ばれるウインドウの上で木形式の証明木を保持しながら行なわれる．

以下に，一般論理における証明のスナップショットを掲げる．思考シート中の証明木は，一般論理[36]における派生規則，

$$X|\sim \forall y.A(y) \Rightarrow X|- \exists y.\sim A(y)$$

の証明を表わしている，ただし証明木の中では，記号"|-"の代わりに"："が用いられている．証明木の葉にはselfと名付けられた一般論理の公理が現われている．白枠で囲まれ，番号１をもちカッコで括られた論理式"$[X|\sim \forall y.A(y)]^1$"はこの証明の仮定を表す．黒塗の枠で囲まれた式はこの証明の結論である．



自動証明機とのインタフェース

既存の定理証明機や項書き換えシステムの結果を有効利用し，さらに証明の効率化を計るためにEUODHILOSには，外部自動証明機を付加するためのインタフェースが用意されている．

データベース

証明された定理，派生規則はそれぞれのデータベースにストアでき，他の証明で自由に使われる．また証明の途中結果や，思考シート上の証明部品などはワークと呼ばれる領域に（必要ならば名前を付けて）ストアしておき，後で証明を再開することができる．

EUODHILOSで構築された論理／理論は論理データベースに蓄えられ，論理メニューで見ることができる．ユーザは，新しく論理／理論を考えるとき，これらすでに存在する論理／理論をコピー，修正するか，あるいは全く新しく定義を始めることになる．

(4) 論証向けのインターフェース

EUODHILOSの使いやすさと論証に適したインタフェースにも配慮がなされてきた〔Sawamura 90, 91〕．既に述べた，さまざまな論理記号設計のためのフォントエディタ，証明の構築を容易にする編集機能などである．これらはビットマップディスプレイ上でマウスを主体に操作できるように設計されている．

(5) EUODHILOSの実現

EUODHILOSはPSI上のオブジェクト指向PROLOGであるESPで実現された．PROLOGが，EUODHILOSのような記号処理システムの実現に適していることは言うまでもないが，ESPのオブジェクト機構（特に，クラス，インスタンス，インヘリタンス）はEUODHILOSのようなメタシステムの実現には特に有効であった．例えば，各ユーザ定義の論理は論理クラスのインスタンスとして生成，管理できるからである．実際，EUODHILOSは二つの基本的なクラス：assistant_systemとtheory，をもつ．assistant_systemは各論理に共通の機能：定義機能，証明編集機能など，の支援機能を司るクラスであり，theoryは論理データ：記号，言語，推論規則，定理など，を保持し管理することを司るクラスである．

7.2 EUODHILOSのさまざまな論理への応用

本節では，EUODHILOSで定義され証明実験が行なわれた論理のいくつかを取り上げ，各論理毎にEUODHILOSの論理記述法，証明構築機能の有効性を調べる．

EUODHILOSの論理記述法，証明構築機能でこれまでうまく取り扱うことができた論理系，理論，証明例には次のものが含まれる〔Sawamura 90, 91, 沢村 93〕．

(a)一階述語論理：さまざまな妥当論理式，Smullyanの論理パズル，停止問題の非可解性，帰納法による証明，初等カテゴリー理論，ハードウェア検証．

(b) 二階述語論理：数学的帰納法と完全帰納法の同値性．

(c)命題様相論理:プログラムについての論証．

(d)内包論理：メタ定理のリフレクティブ証明，Montagueの自然言語の意味論．

(e) Martin-Löfの直観主義的型の理論： 数々の直観主義的命題，選択公理など．

(f) Hoare論理，動的論理：プログラムの性質の証明．

(g) 一般論理：述語論理，適切論理とその弱い体系の論理式．

(h) 適切論理：いくつかの典型的な適切論理式．

(i) 知識の論理：論理パズル．

【一階述語論理】

(1) 停止問題の非可解性

拡張されたDCGで一階言語を定義することは容易である．例えば，限量化された論理式は次のように定義することができる．

```
formula --> bind_op, variable, formula;
bind_op --> "∀" | "∃".
```

ここで，"bind_op"は変数束縛の概念を扱うための特別な構成要素であり，このすぐ右には束縛変数が現われ，変数の右隣には変数束縛のスコープとなる式が続くことを表す．

停止問題の非可解性は次のいくつかの述語：

A(x): xはアルゴリズムである，

C(x): xはあるプログラミング言語で書かれたプログラムである，

D(x,y,z): xはyが与えられた入力zで停止するか否かを決定できる，

を用いると，

$$\vdash \neg\exists x(A(x)\wedge\forall y(C(y)\supset\forall zD(x,y,z)))$$

と表現され，定義された自然演繹法を基礎論理系としChurchの提唱などを公理とする論理式から証明された．証明のステップは推論・派生規則の適用回数で約７０ステップ程度であった．しかしながら，論理式が長いことによる証明木の横への拡がり過ぎによって，画面のスクロールを頻繁に行なわなければならなかったという欠点が見られた．このような大きな証明木に対する表示上の欠点は，他のシステムで主流の線形に並べられた証明とかある種の証明の短縮・簡約手法との併用によって克服できるものと考

えている．

(2) 初等的圏論

圏論の言語は，射と対象を基本オブジェクトとする一階言語であるので，我々の拡張された確定節文法で容易に定めることができる．圏論の推論規則は数十にのぼるため，ここで全てを述べることはしない．それらのいくつかを示す．射Fの定義域をdom(F)，値域をcod(F)と表すと，射FとGが合成可能なのは，cod(F)とdom(G)が一致するときであり，逆にその時は合成可能である．これらのことは次のように二つの推論規則によって表すことができる．

$$\frac{\mathrm{cod}(F)=\mathrm{dom}(G)}{F\cdot G}\ (\text{dot-I}) \qquad \frac{F\cdot G}{\mathrm{cod}(F)=\mathrm{dom}(G)}\ (\text{dot-E})$$

射Fの定義域がXで値域がYであることは，F: X -> Yと表されるが，このような定義は次のような三つの推論規則で表わされる．

$$\frac{\mathrm{dom}(F)=X\quad \mathrm{cod}(F)=Y}{F:X\to Y}\ (\to\text{I})$$

$$\frac{F:X\to Y}{\mathrm{dom}(F)=X}\ (\text{dom-I}) \qquad \frac{F:X\to Y}{\mathrm{cod}(F)=Y}\ (\text{cod-I})$$

このような例からもわかるように，これらの推論規則はEUODHILOSの規則記述法で直接的に記述することができる．

証明実験は，これらの理論の定義を加えた自然演繹法の下で，初等的

圏論の教科書に現われるいくつかの定理を証明することによって行なわれた．その結果，この理論の証明では類似の証明パターンが適用できるケースが多く見られ，こららはすでに証明された派生規則を用いて証明を簡単化することができた．また含意形の定理を定理データベースに蓄えて他の証明で用いるより，それらを仮定からの派生規則の形式にして保存し利用することの方が便利であることが多かった．さらに，このような理論では規則数が膨大になるため，導出のさい規則メニューから規則を選び出すより，EUODHILSOの適用可能な規則を捜し出す機能が特に有効であった．

【内包論理】

内包論理は，単純型の理論に基づく高階の様相論理である．この論理の言語は，型に依存した形成規則，すなわち文脈依存性をもつ形成規則によって定義されている．しかしながらこのような文脈依存性は拡張されたDCGで容易にかつ自然に記述できる．例えば，内包論理の形成規則の一つ，「Tが型(A, B)の項でSが型Aの項であれば，T・Sは型Bの項である」は，我々の内部構造自動生成機構付きDCG記法では，

```
term(B) --> term((A,B)), " · ", term(A).
```

と表現することができる．他方，本来のDCGでは処理の対象となる項の内部構造を含めて，

```
term(T · S, B) --> term(T, (A,B)), " · ", term(S,A).
```


と書かなければならない．この例からもわかるように我々の記法の方が，内部構造の組立というユーザにとって本質的でない作業を避けることができ，さらに誤りの混入を防ぐことができるという点で，明らかに優れていると言える．

内包論理のGallinによる公理系はHilbert型であるから，前節で述べた枠組みで容易に記述される．

(1) メタ定理の証明

証明実験の一つとして，内包論理のメタ定理である一般化規則，

$$\vdash P:t \Rightarrow \vdash \forall x:a.P:t$$

の証明を試みた．この定理を証明するためには，先ずメタ表現である$\vdash P:t$を仮定し，次に"P:t"をオブジェクト論理としての内包論理の世界に持ち込み，そこで証明を進め結論"$\forall x:a.P:t$"を得た後でそれを再びメタの世界に引き戻し$\vdash \forall x:a.P:t$を得るということを行なわなければならない．そのためには必要な限りのメタ言語を定義し，さらに次のメタ，オブジェクト間の相互行き来を許すリフレクションの規則，

$$\frac{\text{provable(A)}}{\text{A}}\ \text{(reflection)} \qquad \frac{\text{A}}{\text{provable(A)}}\ \text{(reflection)}$$

を定義する必要があった．ただし，このような場合，メタとオブジェクト論理の区別は現バージョンのEUODHILOSによって自動的にサポートされているわけではないので，どの部分がメタでどの部分がオブジェクトであ

るかといった点には注意を払わなければならない．

(2) Montagueの意味論

Montagueの言語理論では，自然言語文は初めに内包論理の式に変換され，次いでそれが多世界意味論の下で分析されることになる．ここではこの理論の前半の変換の部分に対するEUODHILOSの応用を考える．例えば，次の自然言語文，

John believes that a fish walks.

は，自然言語から内包論理への変換規則をEUODHILOSの書き換え規則で表現し，それらを適用すると，

(λp:(s,(e,t)).∃x:e.(fish:(e,t)•x:e ∧ p:(s,(e,t)){x:e})) •
^λy:e.(believe:((s,t),(e,t)) • ^(walk:(e,t) • y:e) • j:e)

のようなかなり複雑な論理式が得られる．これに対して，内包論理の推論規則を適用し同値変形していくと，最終的に次のより簡単な論理式を得ることができる，

∃x:e.(fish:(e,t)•x:e∧believe:((s,t),(e,t))•
^(walk:(e,t)•x:e)•j:e).

このように，自然言語分析における複雑な論理式操作を容易にかつ正確に行なうためのツールとしてEUODHILOSを有効に用いることができる．

【Martin-Löfの直観主義的型の理論】

この論理の基本論理式(judgement)は"a∈p"という形式である，ここでaはラムダ式，pは型として解釈された論理式を表す．したがってこれらをEUODHILOSで定義することは容易である．また推論規則も通常の自然演繹型であるので，EUODHILOSの記述法に最も適合する．例えば，λ-導入規則は次のようにMartin-Löfの記法そのままに表現される．

$$\frac{\begin{array}{c}X \in A \\ \vdots \\ F(X) \in B\end{array}}{\lambda X.F(X) \in A \supset B}$$

ただし，ここで適用条件"X is not free in B"が必要になるが，これは前節で述べたEUODHILOSの基本適用条件の一つであるから問題なく記述でき，証明の過程でこの条件は自動的にチェックされることになる．

証明実験では，直観主義論理における５０個程度の代表的な論理式，さらに選択公理などが選ばれ証明された．この中で例えば，背中律の二重否定の証明過程では，前向き証明で得られた結論，

$$\lambda f.f\bullet inr(\lambda x.f\bullet inl(x)) \in (p \vee (p \supset \perp) \supset \perp) \supset \perp$$

と，別の思考シートに置かれた証明のゴール，$F \in \neg\neg(p \vee \neg p)$に対して，書き換え規則，$\neg p \Rightarrow (p \supset \perp)$，を逆向きに適用して得られる証明断片，

$$\frac{F \in (p \vee (p \supset \perp) \supset \perp) \supset \perp}{F \in \neg\neg(p \vee \neg p)}$$

の上式がユニファイされ，次の基本論理式，

$$\lambda f.f\bullet inr(\lambda x.f\bullet inl(x)) \in \neg\neg(p \vee \neg p)$$

が最終的に得られた，ここでFはメタ変数である．この簡単な例は，証明の過程で何らかの情報を求めるような論理計算のとき，それらをいくつかの証明断片から得る方法として前節で述べたユニフィケーションによる証明の結合機能が大変有効であることを示している．

【Hoare論理】

Hoare論理は，プログラムの部分正当性を証明するための論理，あるいはプログラムの公理的意味論の一手法としてよく知られている．Hoare論理の基本論理式は，"P{S}Q"でPとQは述語論理式，Sはあるプログラミング言語で書かれたプログラムである．プログラムの定義も拡張されたDCGで容易に表現でき，また公理系もHilbert型であるので，Hoare論理はEUODHILOSの論理記述枠組みで容易に捉えられる．特に，代入文の公理を次のように直接的に表現することが可能である：

$$P(T/X)\{X := T\}P.$$

ただし，P(T/X)はPの中のXのすべての自由生起にTを代入して得られる式を表す．

これまで，いくつかの基本的なプログラムに対してその正当性の証明

を行なった．プログラムに関する証明木の葉には，トートロジー，算術式，データ構造に関する命題などがいつも現われる．これらの証明に対しては，EUODHILOSの外部定理証明機インタフェースを利用して，既に証明済みの定理データベースから定理を検索する外部定理検索機を呼び出すことによって証明が効率よく行なわれた．

Hoare論理の基本論理式P{S}Qに現われるSはプログラムであるため，一般に極めて長い式となる．実際，この場合も証明木の横への拡がりは縦より飛躍的に増加していく．それを救う一方法として，プログラムセグメントに名前付けを行なう機構が望まれた．

【一般論理】

一般論理とは，広範囲の論理に対して統一的な説明を与えることのできる論理体系である．ここで取り上げた一般論理は，Slaneyによって形式化された論理で，Gentzenのシーケント型の形式的体系となっている．しかしながら，この論理のシーケントは通常のLKなどとは異なり，二つの区切り記号，","と";"によって区切られた論理式の有限列を扱い，それらの区切り記号に与えられる性質によっていろいろな論理が表示される．例えば，この論理のモーダスポネンスは，次のような形式をもつ，

$$\frac{X \vdash A \to B \quad Y \vdash A}{X;Y \vdash B}$$

ここで，X, Yは上の意味での論理式の列を表す．このような一般論理の言語および推論規則はEUODHILOSの枠組みで容易に記述される．一般論理

の枠組みでは，例えば古典述語論理は，すでに定義された適切論理に次のような";"に関するThinningの規則を追加することによって得られる．

X|−A => X;Y|−A.

明らかに，このような";"の性質はEUODHILOSでは書き換え規則，X => X;Y，として記述できる．

EUODHILOSによる一般論理の証明 は，区切り記号が2種類あることを除き，基本的にはLKと同じように進めることができる．行なわれた証明は次のような基本的な論理式，派生規則に対してであった．

p,q∨r |− p∧q∨r（分配則），

X;B |− A∧~A => X |− ~B (Reductio ad absurdum),

|− ∃y.(g(y) -> ∀x.g(x)) (Baffling formula),

X|−~∀y.A(y) => X|−∃y.~A(y).

これらの証明に対しては，前向き，後ろ向き，あるいはそれらの組み合わせによる試行錯誤的証明スタイルを柔軟に適用することができ，EUODHILOSの証明スタイルを強制しない自由度の高い証明方法の便利さ，有利さが確認された．特に，一般論理のようなあまり知られていない論理の証明，あるいは新しい論理を形式化したときなどのように，導かれる定理が予想しにくいときなどでは，EUODHILOSの自由度の高い証明スタイルは効果的であった．

【適切論理】

適切論理（Relevant logic)では，含意命題"A -> B"が真となるのは前提Aが帰結を適切に含意するとき，そしてそのときに限るとされる．このような命題の導出を可能にする適切論理の形式化では，推論規則に古典論理とは異なったかなり複雑な条件が付加されることが多い．論理実験では，適切な推論を結合子の計算によって行なうように形式化された適切論理の含意部分体系$R_{\to}$を取り上げた．

$R_{\to}$の言語は論理記号として"->"のみをもつ命題言語である．ただし，論理式は推論を適切に制御するためのTagをもった"Tag => Formula"の形式である．Tagは論理式の正当化（根拠）を表す記号とそれらを二項演算子（適用オペレータ）で結合してできる結合項である．したがって，これらをEUODHILOSの文法で記述することは容易である．$R_{\to}$の推論規則は自然演繹型の，

$$\frac{\alpha \Rightarrow P \to Q \quad \beta \Rightarrow P}{\alpha\beta \Rightarrow Q}\ (\to E) \qquad \frac{\begin{array}{c}\alpha \Rightarrow P \\ \vdots \\ \beta\alpha \Rightarrow Q\end{array}}{\beta \Rightarrow P \to Q}\ (\to I)$$

であり，Tagの計算は，例えば結合子Cに対応するものは，EUODHILOSでは書き換え規則として，

$$\frac{\alpha\beta\gamma}{\alpha\gamma\beta}$$

のように定義することができる．Tagによって推論規則の仮定の帰結に対する依存性を表現するという方法は，他の論理にも通用する有効な方法であり，通常の集合計算をTagとして表せば自然演繹法でさえ表現できることになる．

証明実験は，$R_{>}$のHilbert型の公理系における次の四つの公理の証明に対して行なわれた：

(1) |- (P -> (Q -> R)) - > (Q -> (P -> R)) (Permutation),

(2) |- (P -> Q) -> ((R -> P) -> (R -> Q)) (Prefixing),

(3) |- (P -> (P -> Q)) -> (P -> Q) (Contraction),

(4) |- P -> P (Self-implication).

この論理では，その言語が単純で，さらにTagによって推論が簡単化されたことにより，論理の定義および上記の証明が極めて短時間で遂行された．

ここで取り上げた論理は，証明スタイルは自然演繹型であるが，依存性の計算は通常のものとは異なる論理体系であった．EUODHILOSは，新しい論理を形式化したとき，その言語の修正，導かれる定理の確認といった論理構築作業にも都合がいい機能と柔軟性を備えており，論理構築ツールとしての有用性も期待できる．

7.3 プログラム変換におけるEUODHILOSの利用

これまで，さまざまな論理による論理定義，および定義された論理の下での証明を通して，EUODHILOSの使用経験を積み評価を行なってきた．これらにはその取り扱いにおいてかなり複雑な記号系，推論系からなる論理が含まれていた．従って，論理型言語の変換に対してもEUODHILOSの

汎用性を利用できることが期待できる．特にEUODHILOS では，プログラム変換の規則を新たに追加したり，修正したりすることはいつでも可能であるので，変換規則の設計とかその効果を確かめるのに有効となろう．

EUODHILOSを論理型言語の変換に適用するためには，次のことが先ず必要になる：

(i) 論理型言語の言語定義ができること

(ii) 変換規則およびその適用条件の定義ができること．

(iii)プログラムの変換で必要な記号操作が可能であること．

論理型言語は一階言語のサブセットであるから，(i)についてはEUODHILOSの言語定義で容易に表現できることは明らかである．第３章で述べたさまざまな変換図式を定義するためには，記号の代入，置き換えといった記号操作，および適用条件を記述することが必要になる．いくつかの適用条件を除き，これらはEUODHILOSの推論／書き換え規則定義法で記述可能である．直接表現できない適用条件については，ユーザ定義の条件チェッカーをインタフェースを通して呼び出すことになる．

変換規則定義例．

(1)直線プログラムの展開

$$\frac{\text{A :- G, f(x), H.} \qquad \text{f(a) :- J.}}{\text{A :- G, f(x)=f(a), J, H.}}$$

(2) 等式代入

$$\frac{\text{A :- G, x = a, J, H.}}{\text{A(a/x) :- J(a/x), H(a/x).}}$$

変換例.

A(x) :- G, p(x), H(x).　　p(a) :- b, c, d.

A(x) :- G, p(x)=p(a), b, c, d, H(x).

A(x) :- G, x = a, b, c, d, H(x).

A(a) :- G, b, c, d, H(a).

この変換の中で，p(x)=p(a)は通常の一方向代入によるパターンマッチでなく，二つの項の単一化を表す．このようなプログラム変換に対しては，EUODHILOSの現在バージョンでは，ユニフィケーションをユーザが間接的に特別の代入操作として与えなければならない．EUODHILOSの新バージョンには，ユニフィケーションオペレーションが基本記号操作機能として組み込まれる予定である．

第８章　他の研究との関係

この章では、本論文の最適化変換において中心的な役割を果たした述語（呼び出し）の決定性と密接に関連する研究を取り上げ、二つの側面より議論する。

8.1 論理プログラミングと制御

一般にプログラムの最適化に関する研究には多くの研究の積み重ねが知られている(例えば、〔Allen 72〕、〔Arsac 79〕、〔Loveman 77〕。これらはAlgol系のプログラム言語に対するものであり、Prolog言語に対してはほとんど参考にすることはできない。

他方、Prologのための効率の良いインタプリタやコンパイラの開発にはこれまで多くの努力がなされてきているが(例えば、〔Warren 77〕、〔Mellish 85〕)、Prologプログラムのソースレベルでの最適化及び変換論に関する研究は、他の非決定性プログラム言語に対しても同様であるが、多く行われてきたとは言えない(一般のプログラム変換技術とシステムについては〔Partsch 83〕を参照)。また、述語（呼び出し）の決定性はプログラムの制御構造と密接に関連する問題である。これまで、論理プログラミングのための制御プリミティブにはいくつかの研究が知られている。そのいくつかはPrologの悪名高いカットに代わるものの追求によって動機づけられていたり、論理プログラムを積極的に制御するためのものであった。むしろ、論理プログラミングにおいて制御をどう考え、かつ記述するかによって各種のPrologの変種が研究されてきたといってもよい。

ここでは述語(呼び出し)の決定性と密接に関係するそのような研究のいくつかと、Prologプログラムの最適化や変換技法に関する研究の現状に簡単に触れ、

我々の方法との類似点、あるいは相違点を明らかにする。

IC-Prolog〔Clark 79〕は、直接的にそして明示的に実行の決定性を規定する言語構成要素はもたないが、その豊富な制御機構によりPrologプログラムの細かな実行制御を可能にしている。

玉木〔玉木 84〕は論理プログラムの関数サブセットを定義した。そこでは、論理プログラミングの中で入出力の定まった決定的な述語、即ち関数を表現する方法を提案している。例えば、リストを分割するプログラムpartition は、

```
function partition(l,x)=m,n
    partition ([x.l ],y) = [x.m ], n   <-  x≦y / m,n = partition (l,y).
    partition ( [x.l],y) = m,[x.n ]   <-  x＞y / m,n = partition (l,y).
    partition ( [],y) = [], [].
```

と書かれる。

近山〔Chikayama 83c〕は、Prologをソースレベルで最適化するための簡単なスケッチを与えている。そこでは我々の方法と同様、インライン展開がPrologの最適化には重要であることが述べられている。我々のインライン展開に関する主要な寄与は、その適用条件を詳細に調べたことにあるといえる。

佐藤と玉木〔Sato 84〕は、展開 (unfold) とたたみ込み (fold) 技法〔Burstall 77〕に基づいて純Prologのプログラムを変換する方法を与えている。BurstallとDarlingtonは、元々一階の再帰的等式で書かれたプログラムの効率の改善を目標として、六つの変換規則: 定義 (definition)、具体化 (instantiation)、展開 (unfolding)、たたみ込み (folding)、抽象化 (abstraction)、法則 (law)を提案した。このアイディアは単純かつ自然であるが、それにもかかわらず強力である。これらの変換技法はPrologプログラムを変換するのに固有の条件が必要になるけれども、Prologの世界にも適用可能となった。その意味でBurstallとDarlingtonの変換技法は普遍的であるといえる。実際、我々のインライン展開はある種のunfolding に対応していることはいうまでもない。

Bloch〔Bloch 84〕は、純Prologプログラムに対してやはり展開/たたみ込み型の変換法、非終端再帰型プログラムの終端再帰型プログラムへの変換法、及びConcurrent PrologのFlat Concurrent Prologプログラムへの変換法等を提案している。これらの一部は変換図式を用いるものである。例えば、階乗やリバースの非終端再帰プログラムを終端再帰に変換する変換図式が考えられている。また、Debray〔Debray 85〕では、同じ問題に対して異なった手法を提案している。非終端再帰プログラムの終端再帰プログラムへの変換は上で述べたUnfold/Fold法でも可能であるが、これらの方法に比べてかなり複雑なプロセスを必要とする。他方、横森〔Yokomori 85〕はかなり一般的なプログラムの図式を、そこに現れる記号の解釈を通してプログラムを具体化し導出するという方法を提案している。

Venken〔Venken 84〕は、プログラムの部分評価 (partial evaluation) という立場から、現実のPrologプログラムを変換する方法を提案している。インライン展開の方法は我々の方法と似た形式であるが、述語の決定性情報を用いる我々の方法とは異なり、変換後のプログラムを特別のメタインタプリタで実行する方法を取っている。

Mellish〔Mellish 85〕は、Prologコンパイラのための最適化技法を提案している。そこではコンパイラにとってプログラムの決定性情報が重要であることが議

論され、我々の決定性述語のクラスよりもかなり小さなクラスが定義されている。その代わり、彼は効率的なコンパイリングのために、Prolog述語のモード宣言を決定性の検出に利用することを提案している。以下では、述語のモード情報が決定性の判定にどのように役立つかを次の例によって見てみよう。

```
:- mode programming-language(+,-).
computer-language(P) :- programming-language(P,Name),human(Name).
programming-language(lisp,mcCarthy).
programming-language(prolog,kowalski).
programming-langauge(pascal,wirth).
human(mcCarthy).
human(kowalski).
human(wirth).
```

述語呼び出しprogramming-language(P,Name)はモード宣言により(r-)決定的であることがわかる。なぜなら、それの述語定義体のいずれの節も互いに両立しないからである。このように述語のモード宣言は決定性判定に強力な情報を提供する。

Komorowski〔Komorowski 82〕は、対話による成長的プログラミング(incremental programming)の理論の部分としてPrologプログラムの部分評価を研究し、その中で三つの部分評価変換法を提案している:枝刈り(pruning)、前方データ構造伝播(forward data structure propagation)、展開(opening) (この展開では後方データ構造伝播 (backward data structure propagation)も行われる)。そして、抽象Prologマシンの下でこれらの変換法が正当であることを示している。展開は我々の方が一般的である。というのは、Komorowskiの展開は、手続き呼び出しの定義体が唯一つの節からなっているときしか行われないからである。また枝刈り、前方データ構造伝播は質問(ゴール)が与えられたとき働くが、我々の方法では

質問が与えられることは想定していない。すなわち、プログラムを構成している述語定義間のみの最適化を行っている。

Smolka〔Smolka 84〕はPrologの素朴なバックトラッキングに基づく制御及びカットに対する反省から、制御を明示的に書くための言語要素をもつSProlog(Self-Documented Prolog)を提案した。そこでは、決定性に関連して、手続きを関数手続きとして宣言したり、手続き呼び出しを関数呼び出しとして表明する方法を提案している。例えば、自然数nの階乗fを計算する関数手続きfactは

fproc fact: ↓integer × ↑integer

と書かれ、述語pを関数呼び出しとするときはfc pと書かれる。我々は本論文で、プログラマに決定性情報の宣言、表明を要求することなく、プログラムテキストより直接決定性情報を抽出する方法を提案した。大まかには、a-決定性の検出は手続きを関数手続きであるとして宣言することに相当し、r-決定性の検出は手続き呼び出しが関数呼び出しであることを表明することに相当する。

Hogger〔Hogger 81〕は、一階論理式で書かれた仕様からホーンプログラムへの演繹的な導出法を研究している。その方法には、これまで言及してきたようなプログラムの変換手法のいくつかが含まれている。ここでは、プログラムが純Prologで表現されるが、正しさを保存する変換論が展開されている。

Warren〔Warren 81〕は、演繹的関係データベースにおける質問文の最適化問題を論じている。Prologプログラムの最適化は、このような質問文の最適化問題と密接に関係する。Warrenは、素朴なバックトラッキングのもとで、質問を構成している部分ゴールの評価順序をどのようにするのが最も効率的かを検討している。ゴールの評価順序を換える目的は、Prologの実行の間に考慮されなければ

ならない代替節の数を最小にすることにある。すなわち、非決定性の縮少である。例えば、インスタンシエイトされることが期待されるもので、かつ代替節の少ないゴールを先に並べることは非決定性の縮少を助ける。しかしながら、我々の最適化法には、このようなゴールを並べ換えることによる手法はない。というのは、現実のPrologでは、一般にゴールの順序に意味があるので、それを構文情報のみで入れ換えることはできないからである。Warrenの場合は関係データベースの検索という特殊な目的があるが故にそれが可能になっているのである。

Bruynooghe〔Bruynooghe 84〕は、素朴なバックトラッキングによる非効率性への反省から、知的バックトラッキングを研究している。知的バックトラッキングとは、失敗に直接関係のあった場所に制御を直接戻すことである。上のWarrenの質問の最適化問題は一種の知的バックトラッキングの一つでもあり、また我々のカットの自動挿入も知的バックトラッキングの一手法とも考えられる。

Debray〔Debray 86〕は、我々の決定性の定義を包含する述語の 関数性(functionality)"を定義し、それに基づいたPrologプログラムの最適化の問題を論じている。述語が関数性をもつとは、述語のモードが与えられたとき、その述語の可能な代替による実行が同じ結果をもたらすことを言う。我々の最適化法の条件もこのより広い概念のもとに定義可能であるが、述語の関数性の概念がより広い概念であるためにそれを自動的に検出することがかなり難しい問題になってしまう。したがって、述語の決定性を我々の定義のように制限した方が決定性の検出には都合がよく、また、実際的にも述語が関数的になっているよりも、決定的になっていることの方が多いように思われるので、我々の定義で十分であると考えられる。

Mendelzon〔Mendelzon 85〕は、論理データベースにおいて、いわゆる関数

依存性(functional dependency) の情報を用いることによる質問処理の効率化の問題を論じている。関数依存性とは上で述べた述語の関数性の定義と同じものであり、それは無駄なバックトラックを防ぐために利用されるデータベースの性質となる。

8.2 Prologプログラミング方法論

本論文ではプログラムの最適化変換に必要とされる述語 (呼び出し) の決定性を論じてきた。a-決定性とr-決定性という概念は、Prologプログラマが日常無意識のうちに用いている直観を自然に定義したものに他ならなかった。また、それらの定義によって捉えられる決定的なゴールのクラスもかなり大きなものと考えられた。

実際のPrologプログラミングでは、プログラマはプログラムを決定的に書くか、もしくは、書いているつもりでいることが多い。また、Prologプログラマはプログラムのデバッグや信頼性の高いプログラムの作成に向けて述語 (呼び出し) の決定性をしばしば問題にする〔近山 83a〕。すなわち、述語 (呼び出し) の決定性とPrologプログラムの信頼性の間には密接な関係があり、プログラマにとっては述語 (呼び出し) の決定性は貴重な信頼性因子と考えられている。逐次型プログラムではプログラムの正当性、停止性、同値性、並行型プログラムではさらにさまざまな究極性(eventuality property) や不変性(invariant property)等が問題にされるように、非決定性プログラム言語ではこの述語 (呼び出し) の決定性という性質が、特に述語間の呼び出し関係が複雑になればなるほど重要になってくる。我々の決定性に関する二つのクラスは、信頼性の高いPrologプログラムの作成時に、またプログラムのデバッグのさいにも有力な指針として使うことができる。このように述語 (呼び出し) の決定性はプログラムの効率性だけでなく信頼性に大きな寄与を与える。またこの意味では、しばしば論争の的になるPrologのカッ

トは、効率性の面からだけでなく、信頼性の高いPrologプログラムの開発という観点からも議論されるべきであろう。我々の決定性の定義によって、決定的と判定される述語の定義体の中に現れるカットの使われ方は、極めて節度のある秩序だった使われ方として参考になるであろう。カットの乱用はPrologプログラムの可読性に大きな影響を与えるが、我々の決定性判定基準は適度なカットの使用法のための判断基準の一つとも見ることができる。

第9章　まとめと今後の課題

9.1　まとめ

本論文で、我々はPrologプログラムのソースレベルにおいて可能な様々な最適化変換手法を提案した。特に、述語（呼出し）の決定性がPrologのような非決定性プログラム言語の最適化には重要な役割を果たすことを示した。一般に、述語(呼出し)の決定性は非決定性プログラム言語の最適化にのみ必要となるプログラム情報ではなく、それはPrologプログラマにも自身のプログラムの振る舞いが決定的であることを伝えたり、確認する手掛りを与える。すなわち、決定性情報は、非決定性プログラム言語によるプログラム開発において信頼性を高める一つの因子であると言える。

また本論文では、Dec system-10 Prologを対象とし、プログラムの統語的側面から最適化を行うPrologソースレベルオプティマイザの試作と評価についても述べた。我々のオプティマイザを一言で特徴づけるならば、 weak but general"なオプティマイザと言える(strong but specialized" なプログラムの変換的アプローチとは対照的である）。本論文の最適化変換手法を個々に見た限りにおいては、大きな最適化効果を期待することは無理なように思われるかもしれない。しかしながら、それらを一つの最適化システムへと統合化したとき、対象が現実のプログラムにもかかわらず、第6章で述べたような効果が得られることが実証されたことは本論文の寄与の一つである。

論文の後半では，汎用論証支援システムEUODHILOS潜在的な有用性を述べ

た．EUODHILOSは元々様々な論理体系を扱えるシステムとして実現された．プログラム変換の体系も一つの論理体系と見ることができるので，EUODHILOSをプログラム変換のツールとして利用できることが期待できる．

9.2 今後の課題

前節で我々のオプティマイザを weak but general" と特徴づけたが、strong and general"なオプティマイザにしていくには、プログラムの動的および意味的解析が本質的に必要になってくるであろう。これによって決定性のクラスの拡大も期待される。今後、述語の型推論、知的バックトラック法、Prologプログラムのデータ/制御フロー解析等の理論的及び実際的な研究が必要とされるところである。

Prolog言語においては、プログラムの最適化問題はプログラムの変換問題と密接に関連し、共通課題が多く見られる。本論文のオプティマイザの機能を、プログラムの仕様からプログラムの作成までに必要とされる変換技法の一部として展開すること、及び知的プログラミング〔玉木　84〕の要素技術の一つとしていくことはオプティマイザの発展的重要テーマである。実際、我々はプログラムの最適化という立場からこれまでいろいろな手法を提案してきたが、これらのほとんどはプログラムの知的変換技術の要素としても利用可能な技法である。

本論文で提案された最適化変換手法がPrologプログラムの同値性を保存することはオペレーショナルには確認されてきた。しかしながら、形式的なPrologプログラムの同値性証明法が望まれるところである。Pure Prolog のプログラムの性質の証明法についてはこれまでいくらか研究が行われてきているが（例えば、〔Hogger 81〕〔Sato 84〕）、現実のPrologプログラムに対してはほとんど知られ

ていない。プログラムの同値性の証明には当然のことながらプログラムの意味論が必要になる。すなわち、どのような意味で同値なのかを定義する必要があるからである。本論文を締め括るに当たり、以下に二つの意味論の可能性を述べておく。一つは言語の意味を意味がよく定まった他の言語で記述しようとする立場に立つものであり、他の一つは意味を公理的に捉えようとするものである。

プログラム言語の意味を内包論理を用いて与える方法が知られている〔沢村79〕、〔沢村 81〕、〔沢村 82〕、〔Sawamura 83〕、〔Sawamura 85c〕、〔Sawamura 86f〕、〔沢村 86g〕、〔南 86〕。ここでは、Prolog言語に対しても、その意味記述に内包論理がかなり有望であることを示唆する観察を示しておく。それはPrologのカットの意味と内包論理が扱う言語の指示的透明性(referential transparency)／不透明性(referential opacity) の問題の類似性に基づく。次の変換を考えてみよう:

```
p :- q,r,s.
r :- t,!,u.
. . .
-------------------------------------- (1)
p :- q, (t,!,u),s.
. . .
```

このような図式が同値でないことは第３章で詳しく論じ、そこでは述語の決定性を用いる解決法が示された。他方、上の変則的事態は、指示的に不透明な文脈への等式代入による変則的事態とも見ることができる。すなわち、内包論理においては、古典論理において成立する次の式は一般的には成立しない。

$$\alpha = \beta \supset (\phi \supset [\alpha / \beta]\ \phi) \qquad (2)$$

ただし、［α／β］ φはφの中に起こる式αを式βで置き換えてできた式であり、βの非束縛変数が代入によって束縛されてはならない。

しかしながら、この式の制限された次の式は成立する。

$$\alpha = \beta \supset (\phi = [\alpha / \beta]\ \phi) \tag{3}$$

ただし、αは^(内包化オペレータ)、□(必然性オペレータ)のスコープには起こらない。

また、

$$\wedge\alpha = \wedge\beta \supset (\phi = [\alpha / \beta]\ \phi) \tag{4}$$

も成立する。

ところで、(1)の不成立は次のようにいうことができるであろう。

(i) 非決定的な文脈への非決定的計算を制限するカットのもち込みに起因する変則性。

(ii) カットという言語表現がもつ、呼び出しｒを失敗させるという非局所的作用に起 因する変則性。

また、(2)の不成立は、「必然」、「信ずる」といった言語表現によって作られる不透明な文脈へ、外延的に等価な表現を代入することによって引き起こされる変則的現象と解釈される。他方、(3)式における置き換えの妥当性は、Prologプログラムの観点から解釈すると、バックトラックが起こらない決定的文脈においては(1)式が成立すること、さらに(4)式の置き換えは置き換える式の方にカットが含まれないこと、あるいはその式においてはバックトラックが起こらないこと等に相当するものと言える。このようにして、これら二つの言語現象は極めて類似した構造をもっていることがわかる。したがって、Prolog言語の内包論理による意味付け、及びそれに基づく最適化変換図式の同値性の形式的証明法の展開が期待される。

言語の意味論の他の方法として、言語の構成要素の各々に、それを特徴づける公理や推論規則を付随させて意味の定義とする公理的あるいは証明論的意味論がある。例えば、Hoare の公理的意味論は、言語 (例えば、ALGOL)の構成要素である代入文や様々な制御文に、それぞれ公理、推論規則を対応させる。また、Gentzen の一階論理の公理系NKは、各論理記号の意味を論理記号の導入及び除去という推論規則によって規定しているものと見ることができる。Prolog言語に対しても、このような意味定義法は可能と思われる。実際、カットの意味をカットの導入、除去規則というスタイルで与えることができる。本論文で与えたカットの自動挿入のための最適化変換図式２とインライン展開のための最適化変換図式５はカットの導入規則であるし、カットの自動挿入のための変換図式の上式と下式を入れ換えてできる変換図式はカットの除去規則の一つであると考えられる（下図参照）。このような立場に基づいてPrologを完全に公理化することができれば、それに従ったPrologプログラムの同値性の形式的証明が可能になってくる。しかしながら，下記の【カットの導入規則】と【カットの除去規則】でcutの意味が完全に捉えられているか否か，即ちcutの完全な公理化を与えているかどうかは未知である．

【カットの導入規則】

(1) 最適化変換図式２によるカットの導入規則（詳しい条件は第３章を参照）

H1 :- Γ1.
:
Hi :- Γi, Δi.
:
Hn :- Γn.

H1 :- Γ1.
:
Hi :- Γi, !, Δi.
:
Hn :- Γn.

(2)最適化変換図式５によるカットの導入規則（詳しい条件は第３章を参照）

H1 :- Γ1.
:
Hi：- Γi,p(A), Δi.
:
Hm :- Γm.
p(A1）：- Θ1.
:
p(An）：- Θn.

H1 :- Γ1.

:
Hi：- Γi, (p(A)＝ p(A1')，Θ1'；... ；p(A）＝ p(An')，Θn'), Δi.
:
Hm :- Γm.

p(A1）：- Θ1.
:
p(An）：- Θn.

【カットの除去規則】最適化変換図式２の下式と上式を入れ替えたカットの除去規則（詳しい条件は第３章を参照）

H1 :- Γ1.
:
Hi :- Γi, !, Δi.
:
Hn :- Γn.

H1 :- Γ1.
:
Hi :- Γi, Δi.
:
Hn :- Γn.

謝　辞

本研究の遂行を温かくそして忍耐強く見守って下さった元富士通（株）国際情報社会科学研究所北川敏男会長に深く感謝する。

北海道大学情報工学科宮本衛市教授には、本論文の字句の訂正から技術的詳細にわたるまでたくさんの御指摘と貴重なご忠告をいただき、また本論文の完成に絶えず励ましをいただいた。ここに感謝の意を表したい。

いくつかの最適化変換手法に関して議論し、有益な示唆を与えていただいた竹島卓研究員に感謝します。また，Prologオプティマイザの作成に当たっては、富士通ソーシャアルサイエンスラボラトリの黒川伊保子さんに終始協力していただいた。

その他、本研究を進めるに当たり、多数の方々から直接的、間接的なお力添えをいただいた。特に、次の人達に感謝し、謝辞を述べたい。

岸本光弘氏 (富士通研・AI研) からは、αPrologの開発者の立場から、本論文の最適化手法の有効性について貴重なコメントをいただいた。

近山隆氏(ICOT)にはProlog最適化全般に関して討論していただき、その結果は本研究を進める上で多くの有益な示唆をもたらした。また、第6章のESPの例を示唆していただいた。

R. Venken（ベルギー・マネジメント研究所）は、筆者らの研究報告〔Sawamura 85〕を詳細に検討し、最適化手法の一つ一つについて彼の最適化手法と対比したコメントを送って下さった。その一部は本論文をまとめるのに参考とさせていただいた。

汎用論証支援システムの研究に当たって，熱心な議論をしていただき，また推論の依存性の一般的扱いを適切論理のアイディアで解決する方法を示唆していただいたBob MeyerおよびMike McRobbie（オーストラリア国立大学）に感謝したい．

最後に，本研究の完成をいつも温かく見守っていてくれた亡き妻，文子には感謝の念に絶えない．この論文を心から彼女に捧ぐ．

参考文献

〔Aho 72〕

A. V. Aho, R. Sethi and J. D. Ullman : Code optimization and finite Church-Rosser systems, in R. Rustin (ed.),Design and optimization of compilers, Prentice-Hall Inc., pp. 89-105, 1972.

〔Allen 72〕

F. E. Allen and J. Cocke : A catalogue of optimizing transformations , in R. Rustin (editor), Design and optimization of compilers, Prentice-Hall Inc., pp. 1-30, 1972.

〔Arsac 79〕

J. J. Arsac : Syntactic source to source transforms and program manipulation, CACM, Vol. 22, No. 1, pp. 43-54, 1979.

〔Bloch 84〕

C. Bloch : Source-to source transformations of logic programs, Master thesis, Dept. of Applied Mathematics, Weizmann Institute of Science, 1984.

〔Bowen 81〕

D. L. Bowen, Dec system-10 Prolog user's manual, version 3.43, Dept. of Artificial Intelligence, Univ. of Edingburgh, 1981.

〔Bruynooghe 84 〕

M. Bruynooghe and L. M. Pereira : Deduction revision by intelligent backtracking, in J. A. Campbell (ed.) : Implementation of Prolog, Ellis Horwood, pp. 194-215, 1984.

〔Burstall 77 〕

R.M. Burstall and J. Darlington : A transformation system for developing recursive programs, JACM, Vol. 24, No. 1, pp. 46-67, 1977.

〔Byrd 80 〕

L. Byrd : Understanding the control flow of PROLOG programs, Research Paper 151, Dept. of Artificial Intelligence, Univ. of Edinburgh, 1980.

〔近山 83a〕

近山 隆 : Prologプログラミングスタイル、メモ、1983.

〔Chikayama 83b 〕

T. Chikayama, M. Yokota and T. Hattori : Fifth generation kernel language, Version 0, ICOT-TR, 1983.

〔Chikayama 83c 〕

T. Chikayama : Source-level optimization in logic programming languages, draft, 1983.

〔Chikayama 84〕

T. Chikayama : ESP reference manual, ICOT-TR-044, 1984.

〔Church 36 〕

A. Church : A note on the Entscheidungsproblem, The Journal of Symbolic Logic, Vol. 1,

No. 1, pp. 40-41, 1936 and Correction to a note on the Entscheidungsproblem, ibid., Vol.1, No. 3, pp. 101-102, 1936.

〔Clark 79〕

K. L. Clark and F. G. McCabe : The control facilities of IC-Prolog, 1979, in P. Michie (ed.) : Expert Systems in Micro Electronic Age, Edinburgh Univ. Press, pp. 122-149, 1979.

〔Davis 65〕

M. Davis (ed.) : The undecidable, Raven Press, 1965.

〔Debray 85 〕

S. K. Debray : Optimizing almost-tail-recursive Prolog programs, LNCS, Vol. 201, pp. 204-219, 1985.

〔Debray 86 〕

S. K. Debray and D. S. Warren : Detection and optimization of functional computations in Prolog, LNCS 225, pp. 490-504, 1986.

〔Dijkstra 76 〕

E. W. Dijkstra : A discipline of programming, Prentice-Hall, 1976.

〔Floyd 67〕

R. Floyd : Non-deterministic algorithms, JACM, Vol. 14, No. 4, pp. 636- 644, 1967.

〔Hogger 81 〕

C. J. Hogger : Derivation of logic programs, JACM, Vol. 27, No. 4, pp. 372-392, 1981.

〔Hopcroft 69 〕

J. E. Hopcroft and J. D. Ullman : Formal languages and their relation to automata, Addison-Wesley, 1969.

〔Komorowski 82 〕

H. J. Komorowski : Partial evaluation as a means for inferencing data structures in an applicative language : A theory and implementation in case of Prolog, Conf. record of the 9th ACM Symp. on Principles of Programming Languages, ACM, pp. 255-267, 1982.

〔Kowalski 79 〕

R. Kowalski : Logic for problem solving, N-Holland, 1979.

〔国藤 83 〕

国藤：私信、1983.

〔Loveman 77〕

D. B. Loveman : Program improvement by source-to-source transformation, JACM, Vol. 24, No. 1, pp. 121-145, 1977.

〔Martelli 82 〕

A. Martelli and U. Montanari : An efficient unification algorithm, ACM TOPLAS, Vol. 4, No. 2, pp. 258-282, 1982.

〔Mellish 85〕

C. S. Mellish : Some global optimizations for a Prolog compiler, J. of Logic Programming, Vol. 2, No. 1, pp. 43-66, 1985.

〔Mendelzon 85〕

A. O. Mendelzon : Functional dependences in logic programs, Proc. of 11th ACM Int. Conf. on Very Large Data Bases, pp. 324-330, 1985.

〔Mishra 84〕

P. Mishra : Towards a theory of types in Prolog, Proc. of the 1984 Int. Symp. on Logic Programming, IEEE Computer Society, pp. 289-298, 1984.

〔中川 85〕

中川、中村 : Prolog 等価変換エディタと変換戦略、情報処理学会論文誌、第23巻、第5 号、pp. 905-912, 1985.

〔O'Keefe 85〕

R. A. O'Keefe : On the treatment of cuts in Prolog source-level tools, Proc. of the Symposium on Logic Programming, IEEE Computer Society, Boston, Ma., pp. 68-72, 1985.

〔Partsch 83〕

H. Partsch and R. Steinbruggen : Program transformation systems, Computing Surveys, Vol. 15, No. 3, pp. 199-236, 1983.

〔Pereira 83〕

F. Pereira (ed.) : C-Prolog user's manual, version 1.4a, Nihon DEC, 1983.

〔Sato 84〕

T. Sato and H. Tamaki : Unfold/fold transformation of logic programs, Proc. of 2nd Int. Logic Programming Conference, Uppsala, pp. 127-138, 1984.

〔Sebelik 82〕

J. Sebelik and P. Stepanek : Horn clause programs for recursive functions, in K. L. Clark and S-A. Tarnlund (eds.) : Logic programming, Academic Press, pp. 325-340, 1982.

〔Shapiro 83〕

E. Y. Shapiro : A subset of concurrent Prolog and its interpreter, ICOT, TR-003, 1983.

〔沢村 79〕

沢村　一：算法論理、情報処理学会誌、第20巻、第 8 号、pp. 725-734, 1979.

〔沢村 81〕

沢村　一：内包論理に基づく逐次型プログラムの論理、情報処理学会論文誌、第22巻、第 3 号、pp. 216-224, 1981.

〔沢村 82〕

沢村　一：プログラムの論理　- 内包論理によるアプローチ -、富士通（株）国際研研究報告　No. 7 号、43p., 1982.

〔Sawamura 83〕

H. Sawamura, T. Takeshima and A. Kato : Towards a descriptive language based on many-sorted equational logic, IIAS-RR No. 42, 1983.

〔Sawamura 85a〕

H. Sawamura, T. Takeshima and A. Kato, Source-level optimization techniques for Prolog, IIAS R.R. No. 52, ICOT-TR-0091, 1985, to be submitted.

〔Sawamura 85b〕

H. Sawamura and T. Takeshima: Recursive unsolvability of determinacy, solvable cases of determinacy and their applications to Prolog optimization, Proc. of the Symposium on Logic Programming, IEEE Computer Society, Boston, Ma., pp. 200-207, 1985.

〔Sawamura 85c〕

H. Sawamura : Axiomatization of computer-oriented modal logic and decision procedure, Bulletin of Informatics and Cybernetics, Vol. 21, No.3-4, pp. 57-66, 1985.

〔沢村 86a〕

沢村 一：PROLOG 述語（呼び出し）の決定性、ソフトウェア科学会論文誌、コンピュータソフトウェア，VOl. 4, No. 4, pp. 76-82, 1987.

〔沢村 86b〕

沢村 一：PROLOG 述語（呼び出し）の決定性について、情報処理学会ソフトウェア基礎論研究会報告　No. 18, pp. 1-8, 1986.

〔沢村 86c〕

沢村　一：Prologソースレベルオプティマイザの試作とその性能評価、情報処理学会論文誌、第28巻、第４号、pp. 339-348, 1987.

〔沢村 86d〕

沢村　一：Prolog プログラムの最適化、富士通（株）国際研研究報告　No. 19, ICOT-TR, 1986, 48p.

〔Sawamuta 86e〕

H. Sawamura : Undecidability of determinacy, two decidable cases of determinacy and their applications to source-to-source optimization of Prolog programs, to be submitted.

〔Sawamura 86f〕

H. Sawamura : A proof constructor for intensional logic - with S5 decision procedure -, IIAS-RR No. 65, also ICOT-TR, 1986.

〔沢村 86g〕

沢村・南・佐藤・小野・小野：論理式エディタ、構造エディタに関するワークショップ（ＷＳＥ），ソフトウェア科学会、相模原、昭和56年５月、1986.

〔沢村 87〕

沢村 一：汎用の論証支援システムの構想とその実現法，情報処理学会研究会ソフトウェア基礎論, No. 22, 1987.

〔沢村 89〕
沢村 一：適切さの論理，情報処理，第30巻， 第6号，pp. 665-673, 1989.

〔Sawamura 90〕
H. Sawamura, T. Minami, K. Yokota and K. ohashi: A Logic Programming Approach to Specifying Logics and Constructing proofs, Australian National University, Automated Reasoning Project Technical Report, ARP-TR-5/90, 1990, 20p. ; and 7th International Conference on Logic Programming, Jerusalem, Israel, Jun. 18-20, 1990; Proceedings: D. H. D. Warren and P. Szeredi, eds., pp. 405-424, 1990, The MIT Press.

〔Sawamura 90a〕
H. Sawamura: Notes on Unsolvability Proofs of Determinacy in Prolog, Australian National University, Automated Reasoning Project Technical Report, ARP-TR-4/90, 1990, 11p.

〔Sawamura 91〕
H. Sawamura, T. Minami, K. Yokota and K. ohashi: Potential of General-Purpose Reasoning Assistant System EUODHILOS, Software Science and Engineering: Selected Papers from the Kyoto Symposia, World Scientific, pp. 164-188, 1991.

〔Sawamura 92〕
H. Sawamura, T. Minami and K. ohashi: System Description of EUODHILOS: A General Reasoning System for a Variety of Logics, International Conference on Logic Programming and Automated Reasoning, St. Petersburg, Russia, July 12 - 17, 1992; Lecture Notes in Artificial Intelligence, Vol. 624, pp. 501-503, 1992.

〔Sawamura 92〕
H. Sawamura, T. Minami, Y. Ohtani and K. ohashi: EUODHILOS: A General Reasoning System for a Variety of Logics, The First International Conf. on the Practical Application of Prolog, London, April 2 - 3, Abstracts of the Poster Session, pp. 27-32, 1992.

〔沢村 93〕
沢村，南，大谷：汎用論証支援システムEUODHILOSの応用と評価，情報処理学会論文誌，掲載予定，1993.

〔Smorka 84 〕
G. Smorka : Making control and data flow in logic programs explicit, Proc. of the 1984 Lisp and Functional Programming Language Conf., ACM, pp. 311- 322, 1984.

〔Sussman 71〕
G. J. Sussman : Micro-planner reference manual, MIT AI-Memo 203A, 1971.

〔玉木 83 〕
玉木久夫：Prologの関数サブセットP とそれ自身による処理系記述，Proc. of the Logic Programming Conf., Tsukuba, 1983.

〔玉木 84 〕
玉木久夫・佐藤泰介：Prologの知的プログラミング環境, 情報処理, Vol. 25, No. 12, pp . 1360-1367, 1984.

〔Tarnlund 77〕

S-A. Tarnlund : Horn clause computability, BIT, Vol. 17, pp. 215-226, 1977.

〔Venken 84〕

R. Venken : A prolog meta-interpreter for partial evaluation and its application to source-to-source transformation and query-optimisation, ECAI 84 : Advances in Artificial Intelligence, T.O'Shea (editor), N-Holland, pp. 91-100, 1984.

〔Warren 77〕

D. H. D. Warren : Implementing Prolog - compiling predicate logic programs, D.A.I. Research Report, No. 39 and No. 40, Dept. of Artificial Intelligence, Univ. of Edinburgh, 1977.

〔Warren 81〕

D. H. D. Warren : Efficient processing of interactive relational database queries expressed in logic, VLDB '81, pp. 272-281, 1981.

〔Yokomori 85〕

T. Yokomori : A logic program schema and its applications, Proc. of the 9th IJCAI, pp. 723-725, 1985.

付録　A（Prologオプティマイザプログラムのコンサルト述語）

```
% ------------------------------------------------------------------------------
% |                                                                            |
% |                        CONSULTING PROLOG_OPTIMIZER                         |
% |                                                                            |
% ------------------------------------------------------------------------------

% This is stored in the file 'prolog_optimizer', and it consults all the
% files needed to evoke prolog optimizer.
:-[optimizer_main,
   reading,
   side_effect_free,
   inline_expansion_sl,
   inline_expansion_tr,
   inline_expansion_gr,
   recognize_prog_schemata,
   prop_simp,
   partial_unif,
   determinism,
   cut_insertion,
   integrate,
   distribution,
   del_vars,
   auxiliary_preds,
   apply_opt_tech,
   writing,
   transform_prog_scheme,
   time].
```

付録　B （メインプログラム述語）

```
% This is a main program for prolog_optimizer(version 1).
prolog_optimizer:-
     nl,write('------- PROLOG OPTIMIZER(version 1) starts -------'),nl,nl,
     write('Source program file name to be optimized : '),
     get_string(Input_file_name),nl,
     readin_input_file(Input_file_name,Preds,Directives,NATD),
     classify_pred_defs(Preds,NATD),
     a_determinism(Preds,NATD),
     cut_insertion(Preds,Preds0),
     write('Local optimizations ? '),
     write('(y or n) : '),
     get_answer(S1),nl,
     (S1==121,
      write('Type in a single mnemonics, or a list of mnemonics expressing'),
      write(' optimization techniques, ending with a full stop : '),
      read(L1),nl,
      (atom(L1),apply_opt_tech([L1],Preds0,Preds1) ;
       apply_opt_tech(L1,Preds0,Preds1)),
      write(L1),write(' applied.'),nl,nl
      ; Preds1=Preds0),
     write('Straight-line expansion ? (y or n) :'),
     get_answer(S2),nl,
     (S2==121,
      inline_expansion_sl(Preds1,Preds1,NATD,Preds2),
      write('Inline expansion completed.'),nl,nl,nl,
      writeout_terminal(NATD,Preds2),
      write('Local optimizations ? '),
      write('(y or n) : '),
      get_answer(S3),nl,
      (S3==121,
       write('Type in one of the following: rd., es., or [rd,es].'),
       write(' : '),
       read(L2),nl,
       (atom(L2),apply_opt_tech([L2],Preds2,Preds3) ;
        apply_opt_tech(L2,Preds2,Preds3)),
       write(L2),write(' applied.'),nl,nl ; Preds3=Preds2)),
      write('Tail-recursive expansion ? (y or n) : '),
      get_answer(S4),nl,
      (S4==110,
       apply_opt_tech([ig],Preds3,Preds4a),
       compress_pred_defs(Preds4a,Preds4),
       write('Your Prolog program has been optimized as follows :'),nl,nl,nl,
       writeout_terminal(NATD,Preds4),nl,
       write('Output file name : '),get_string(Output_file_name),nl,
       write_output_file(Output_file_name,Preds4,Directives),
       write('Optimized programs written to the file : '),
       write(Output_file_name),nl,nl,
       write('PROLOG OPTIMIZER ends.'),nl,nl
       ;
       inline_expansion_tr(Preds3,Preds3,NATD,Preds4),
```



```
       write('Tail-recursive inline expansion completed.'),nl,nl,
       write('Local optimizations to the resulting program ? '),
       write('(y or n) : '),
       get_answer(S5),nl,
       (S5==121,
        write('Type in one of the following: rd.,es., or [rd,es]. : '),
        read(L3),nl,
        (atom(L3),apply_opt_tech([L3],Preds4,Preds5) ;
         apply_opt_tech(L3,Preds4,Preds5)),
        write(L3),write(' applied.'),nl,nl ; Preds5=Preds4),
        % Inline expansion for straight_line programs
        write('General recursive inline expansion ? (y or n) : '),
        get_answer(S6),nl,
        (S6==110,
         apply_opt_tech([ig],Preds5,Preds6a),
         compress_pred_defs(Preds6a,Preds6),
         write('Your Prolog program has been optimized as follows :'),
         nl,nl,nl,
         writeout_terminal(NATD,Preds6),nl,
         write('Output file name : '),get_string(Output_file_name),nl,
         write_output_file(Output_file_name,Preds6,Directives),
         write('Optimized programs written to the file : '),
         write(Output_file_name),nl,nl,
         write('PROLOG OPTIMIZER ends.'),nl,nl
         ;
         inline_expansion_gr(Preds5,Preds5,NATD,Preds6),
         write('General recursive inline expansion completed.'),
         nl,nl,
         write('Local optimizations to the resulting program ? '),
         write('(y or n) : '), get_answer(S7),nl,
         (S7==121,
          write('Type in one of the following: rd.,es., or [rd,es]. : '),
          read(L4),nl,
          (atom(L4),apply_opt_tech([L4],Preds6,Preds7) ;
           apply_opt_tech(L4,Preds6,Preds7)),
          write(L4),write(' applied.'),nl,nl ; Preds7=Preds6 ),
          apply_opt_tech([ig],Preds7,Preds8a),
          compress_pred_defs(Preds8a,Preds8),
          write('Your Prolog program has been optimized as follows :'),
          nl,nl,nl,
          writeout_terminal(NATD,Preds8),nl,
          write('Output file name : '),get_string(Output_file_name),nl,
          write_output_file(Output_file_name,Preds8,Directives),
          write('Optimized programs written to the file : '),
          write(Output_file_name),nl,nl,
          write('PROLOG OPTIMIZER ends.'),nl,nl
         )),!.
```

付録　C （直線型プログラムの判定述語）

```
% ---------------- STRAIGHT-LINE CLAUSE ----------------------------------
% straight_line_clause tests whether a clause is straight-line or not.

straight_line_clause((P:-Q)):-!,
     not_include(P,Q),!.
straight_line_clause(P):-!.


% ---------------- STRAIGHT-LINE PREDICATE DEFINITION ---------------------
% straight_line_pred_def tests whether a predicate definition consists of
% straight_line clauses, or not.

straight_line_pred_def([]):-!.
straight_line_pred_def([H|T]):-
      straight_line_clause(H),!,
      straight_line_pred_def(T),!.
```


付録　D （決定性情報の検出述語）

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% undeterminative_bulit_in_pred

undeterminative_b_pred(clause,2).
undeterminative_b_pred(setof,3).
undeterminative_b_pred(bagof,3).
undeterminative_b_pred('^',2).
undeterminative_b_pred(call,1).
undeterminative_b_pred(repeat,0).
undeterminative_b_pred(retract,1).
undeterminative_b_pred(recorded,3).
undeterminative_b_pred(current_atom,2).
undeterminative_b_pred(current_functor,2).
undeterminative_b_pred(current_predicate,2).
undeterminative_b_pred(current_op,3).
undeterminative_b_pred(';',2).
undeterminative_b_pred('->',2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% a_determinism

a_determinism(Preds,NATDs):-
        abolish(present_pred,2),
        abolish(undeterminative_pred,2),
        a_determinism(Preds,Preds,NATDs),!.
a_determinism(Preds,
              [[N/A/(T/d)|Defs]|Rest_preds],
              [N/A/(T/d)|Rest_NATDs]):-
        asserta(present_pred(N,A)),
        a_determinism_pred(Defs,Preds),
        abolish(present_pred,2),
        a_determinism(Preds,Rest_preds,Rest_NATDs).
a_determinism(Preds,
              [[N/A/_|Defs]|Rest_preds],
              [NATD|Rest_NATDs]):-
        abolish(present_pred,2),
        asserta(undeterminative_pred(N,A)),
        a_determinism(Preds,Rest_preds,Rest_NATDs).
a_determinism(Preds,[],[]):-!.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% a_determinism_pred

a_determinism_pred([(H:-B)],Preds):-
        !,
        (a_determinism_body(B,Preds);
         determinism_goals(B,Preds)).
a_determinism_pred([(H:-B)|Rest_defs],Preds):-
        !,
        a_determinism_body(B,Preds),
        !,
        a_determinism_pred(Rest_defs,Preds).
a_determinism_pred([H|[]],Preds):-!.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% a_determinism_body
```

```
a_determinism_body((!),Preds):-!.
a_determinism_body((!,Rest_goals),Preds):-
        cut_free_goal_n(Rest_goals),
        !,
        determinism_goals(Rest_goals,Preds),
        !.
a_determinism_body((Goal,Rest_goals),Preds):-
        functor(Goal,N,A),
        !,
        modification_free(N,A,Preds),
        !,
        a_determinism_body(Rest_goals,Preds).
a_determinism_body(Goal,Preds):-!,fail.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% determinism_goals

determinism_goals((Goal,Rest_goals),Preds):-
        functor(Goal,N,A),
        !,
        modification_free(N,A,Preds),
        !,
        determinism_goal(Goal,N,A,Preds),
        !,
        determinism_goals(Rest_goals,Preds).
determinism_goals(Goal,Preds):-
        functor(Goal,N,A),
        !,
        modification_free(N,A,Preds),
        !,
        determinism_goal(Goal,N,A,Preds),
        !.

determinism_goal(Goal,N,A,Preds):-
        (undeterminative_b_pred(N,A);
         present_pred(N,A)),
        !,
        fail.
determinism_goal(Goal,N,A,Preds):-
        !,
        (a_determinism_goal(N,A,Preds);
         r_determinism_goal(Goal,N,A,Preds)).

a_determinism_goal(N,A,Preds):-
        undeterminative_pred(N,A),
        !,
        fail.
a_determinism_goal(N,A,Preds):-
        member([N/A/_|Defs],Preds),
        asserta(present_pred(N,A)),
        !,
        a_determinism_pred(Defs,Preds),
        retract(present_pred(N,A)),!.
a_determinism_goal(N,A,Preds):-!.

r_determinism_goal(Goal,N,A,Preds):-
```



```
        member([N/A/_|Defs],Preds),
        !,
        r_determinism_pred(Goal,Defs,Preds).
r_determinism_goal(Goal,N,A,Preds):-
        !,
        ¥+(undeterminative_b_pred(N,A)).

r_determinism_pred(H,[],Preds):-!.
r_determinism_pred(H,[(H1:-B)|Rest_defs],Preds):-
        ¥+(¥+ H=H1 ),
        !,
        determinism_goals(B,Preds),
        !,
        differ(H,Rest_defs),
        !.
r_determinism_pred(H,[H1|Rest_defs],Preds):-
        ¥+(¥+ H=H1),
        !,
        differ(H,Rest_defs),
        !.
r_determinism_pred(Goal,[(H:-B)|Rest_defs],Preds):-
        !,
        modification_free_check(B,Preds),
        !,
        r_determinism_pred(Goal,Rest_defs,Preds).
r_determinism_pred(Goal,[H|Rest_defs],Preds):-
        !,
        r_determinism_pred(Goal,Rest_defs,Preds).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% cut_free_goal_n

cut_free_goal_n(Goal):-
        Goal==!,
        !,
        fail.
cut_free_goal_n((Goal;Rest_goals)):-
        !,
        cut_free_goal_n(Goal),
        !,
        cut_free_goal_n(Rest_goals).
cut_free_goal_n((Goal,Rest_goals)):-
        !,
        cut_free_goal_n(Goal),
        !,
        cut_free_goal_n(Rest_goals).
cut_free_goal_n(Goal):-!.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% modification_free

modification_pred(assert,1).
modification_pred(assert,2).
modification_pred(asserta,1).
modification_pred(asserta,2).
modification_pred(assertz,1).
modification_pred(assertz,2).
modification_pred(retract,1).
```

```
modification_pred(abolish,2).

modification_free(N,A,Preds):-
        abolish(temp_pred,2),
        !,
        modification_free_pred(N,A,Preds).

modification_free_pred(N,A,Preds):-
        modification_pred(N,A),
        !,
        fail.
modification_free_pred(N,A,Preds):-
        temp_pred(N,A).
modification_free_pred(N,A,Preds):-
        member([N/A/_|Defs],Preds),
        asserta(temp_pred(N,A)),
        !,
        modification_free_list(Defs,Preds).
modification_free_pred(N,A,Preds):-
        asserta(temp_pred(N,A)).

modification_free_list([(H:-B)|Rest_defs],Preds):-
        !,
        modification_free_body(B,Preds),
        !,
        modification_free_list(Rest_defs,Preds).
modification_free_list([H|Rest_defs],Preds):-
        !,
        modification_free_list(Rest_defs,Preds).
modification_free_list([],_):-!.

modification_free_body((Goal,Rest_goals),Preds):-
        functor(Goal,N,A),
        !,
        modification_free_pred(N,A,Preds),
        !,
        modification_free_body(Rest_goals,Preds).
modification_free_body(Goal,Preds):-
        functor(Goal,N,A),
        !,
        modification_free_pred(N,A,Preds),
        !.


modification_free_check(Goals,Preds):-
        abolish(temp_pred,2),
        !,
        modification_free_body(Goals,Preds).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% cut_free

cut_free(Goal):-
        Goal==!,
        !,
        fail.
cut_free((Goal,Rest_goals)):-
```



```
        Goal==!,
        !,
        fail.
cut_free((Goal,Rest_goals)):-
        !,
        cut_free(Rest_goals).
cut_free(Goal):-!.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% differ

differ(_,[]):-!.
differ(Goal,[Goal|Rest]):-
        !,
        fail.
differ(Goal,[(Goal:-Body)|Rest]):-
        !,
        fail.
differ(Goal,[Goal1|Rest]):-
        !,
        differ(Goal,Rest).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% member

member(X,[X|Y]):-!.
member(X,[Y|Z]):-
        member(X,Z),!.
member(X,[]):-!,fail.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% r_determinism

r_determinism(Goal,Preds):-
        functor(Goal,N,A),
        !,
        (a_determinism_goal(N,A,Preds);
         r_determinism_goal(Goal,N,A,Preds)).
```

付録　E（カットの自動挿入述語）

```
cut_insertion(Preds,Predds0) :-
      cut_insertion(Preds,Preds,Predds0).

cut_insertion([[Pred/Arity/Inf|Defs]|Pred_rest],
        Preds,[[Pred/Arity/Inf|Defs0]|Predds0]) :-
      cut_ins_1(Defs,Preds,Defs0),
      cut_insertion(Pred_rest,Preds,Predds0).
cut_insertion([],_,[]) :- !.

cut_ins_1([(H:-(!,Bs))],Preds,[(H:-Bs0)]) :- !,
      cut_ins_2((!,Bs),Preds,Bs0),!.
cut_ins_1([(H:-Bs)],Preds,[(H:-Bs0)]) :- !,
      cut_ins_2((!,Bs),Preds,(!,Bs0)),!.
cut_ins_1([H],Preds,[(H:-!)]) :- !.
cut_ins_1([(H:-Bs)|Defs],Preds,[(H:-Bs0)|Defs0]) :-
      cut_ins_2(Bs,Preds,Bs0),
      cut_ins_1(Defs,Preds,Defs0).
cut_ins_1([H|Defs],Preds,[H|Defs0]) :-
      cut_ins_1(Defs,Preds,Defs0).

cut_ins_2((!,B,Bs),Preds,(!,B,Bs0)) :-
      check_d(B,Preds),!,
      cut_ins_3(Bs,Preds,Bs0).
cut_ins_2((B;Bs),Preds,(B0;Bs0)) :- !,
      cut_ins_2(B,Preds,B0),
      cut_ins_2(Bs,Preds,Bs0),!.
cut_ins_2(((B;Bs),Rest),Preds,(B0,Rest0)):- !,
      cut_ins_2((B;Bs),Preds,B0),
      cut_ins_2(Rest,Preds,Rest0),!.
cut_ins_2((B1,B2,Bs),Preds,(B1,Bs0)) :- !,
      cut_ins_2((B2,Bs),Preds,Bs0).
cut_ins_2((!,B),Preds,(!,B,!)) :-
      check_d(B,Preds),!.
cut_ins_2(B,_,B) :- !.

cut_ins_3((!,Bs),Preds,Bs0) :-
      cut_ins_2((!,Bs),Preds,Bs0).
cut_ins_3((B,Bs),Preds,(B,Bs0)) :-
      check_d(B,Preds),!,
      cut_ins_3(Bs,Preds,Bs0).
cut_ins_3((B,Bs),Preds,(!,B,Bs0)) :-
      cut_ins_2(Bs,Preds,Bs0).
cut_ins_3(!,Preds,!) :- !.
cut_ins_3(B,Preds,(B,!)) :-
      check_d(B,Preds),!.
cut_ins_3(B,_,(!,B)) :- !.
```



```
check_d(P,_) :-
      non_deter_sys(P),!,fail.
check_d(P,Preds) :-
      functor(P,Pred,Arity),
      ¥+ member([Pred/Arity/(_/D)|_],Preds), !.
check_d(P,Preds) :-
      functor(P,Pred,Arity),
      ¥+ ¥+ (member([Pred/Arity/(_/D)|_],Preds),!,
            (D==d;r_determinism(P,Preds))),!.



member(A,[A|_]) :- !.
member(A,[_|X]) :-
      member(A,X),!.

non_deter_sys(clause(_,_)).
non_deter_sys(setof(_,_,_)).
non_deter_sys(bagof(_,_,_)).
non_deter_sys('^'(_,_)).
non_deter_sys(call(_)).
non_deter_sys(repeat).
non_deter_sys(retract(_)).
non_deter_sys(recorded(_,_,_)).
non_deter_sys(current_atom(_,_)).
non_deter_sys(current_functor(_,_)).
non_deter_sys(current_predicate(_,_)).
non_deter_sys(current_op(_,_,_)).
non_deter_sys(';'(_,_)).
```

付録　F （直線型プログラムのインライン展開述語）

```
inline_expansion_sl(Preds,New_preds):-
        inline_expansion_sl(Preds,New_preds,Preds),
        !.

inline_expansion_sl([],[],_):-!.
inline_expansion_sl([[N/A/(T/D)|Defs]|Rest],
                    [[N/A/(T/D)|New_defs]|Result],
                    Preds):-
        T==sl,
        abolish(using_pred,2),
        asserta(using_pred(N,A)),
        (D==d,
         inline_expansion_defs1(Defs,New_defs,Preds);
         inline_expansion_defs2(Defs,New_defs,Preds)),
        inline_expansion_sl(Rest,Result,Preds),
        !.
inline_expansion_sl([Definition|Rest],
                    [Definition|Result],
                     Preds):-
        inline_expansion_sl(Rest,Result,Preds),
        !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  inline_expansion_defs1

inline_expansion_defs1([],[],_):-!.
inline_expansion_defs1([Def|[]],[New_def|[]],Preds):-
        inline_expansion_def1_sp(Def,New_def,Preds),
        !.
inline_expansion_defs1([Def|Rest],[New_def|Result],Preds):-
        inline_expansion_def1(Def,New_def,Preds),
        inline_expansion_defs1(Rest,Result,Preds),
        !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  inline_expansion_def1

inline_expansion_def1((H:-B),(H:-New_B),Preds):-
        inline_expansion_body1(B,New_B,[],Preds),
        !.
inline_expansion_def1(H,H,_):-!.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  inline_expansion_body1

inline_expansion_body1((!,Goals),(!,New_goals),Pre_goals,Preds):-
        cut_free_goal(Goals),
        inline_expansion(Goals,New_goals,Preds),
        !.
inline_expansion_body1((Goal,Rest),(New_goal,Result),Pre_goals,Preds):-
        functor(Goal,N,A),
        ¥+(builtin_pred(N,A)),
        ¥+(using_pred(N,A)),
        member([N/A/(T/D)|Defs],Preds),
        T==sl,
```



```
        (cut_free_defs(Defs),
         construct_disjunctive_terms(Goal,Defs,New_goal,Preds,N,A);
         a_determinism_list(Pre_goals,Preds),
         construct_disjunctive_terms(Goal,Defs,New_goal,Preds,N,A)),
        inline_expansion_body1(Rest,Result,[Goal|Pre_goals],Preds),
        !.
inline_expansion_body1((Goal,Rest),(Goal,Result),Pre_goals,Preds):-
        inline_expansion_body1(Rest,Result,[Goal|Pre_goals],Preds),
        !.
inline_expansion_body1(Goal,New_goal,Pre_goals,Preds):-
        functor(Goal,N,A),
        ¥+(builtin_pred(N,A)),
        ¥+(using_pred(N,A)),
        member([N/A/(T/D)|Defs],Preds),
        T==sl,
        (cut_free_defs(Defs),
         construct_disjunctive_terms(Goal,Defs,New_goal,Preds,N,A);
         a_determinism_list(Pre_goals,Preds),
         construct_disjunctive_terms(Goal,Defs,New_goal,Preds,N,A)),
        !.
inline_expansion_body1(Goal,Goal,_,_):-
        !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  inline_expansion_def1_sp

inline_expansion_def1_sp((H:-B),(H:-New_B),Preds):-
        inline_expansion_body1_sp(B,New_B,[],Preds),
        !.
inline_expansion_def1_sp(H,H,_):-!.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  inline_expansion_body1_sp

inline_expansion_body1_sp((!,Goals),(!,New_goals),Pre_goals,Preds):-
        cut_free_goal(Goals),
        inline_expansion(Goals,New_goals,Preds),
        !.
inline_expansion_body1_sp((Goal,Rest),(New_goal,Result),Pre_goals,Preds):-
        functor(Goal,N,A),
        ¥+(builtin_pred(N,A)),
        ¥+(using_pred(N,A)),
        member([N/A/(T/D)|Defs],Preds),
        T==sl,
        (cut_free_defs(Defs),
         construct_disjunctive_terms(Goal,Defs,New_goal,Preds,N,A);
         determinism_list(Pre_goals,Preds),
         construct_disjunctive_terms(Goal,Defs,New_goal,Preds,N,A)),
        inline_expansion_body1_sp(Rest,Result,[Goal|Pre_goals],Preds),
        !.
inline_expansion_body1_sp((Goal,Rest),(Goal,Result),Pre_goals,Preds):-
        inline_expansion_body1_sp(Rest,Result,[Goal|Pre_goals],Preds),
        !.
inline_expansion_body1_sp(Goal,New_goal,Pre_goals,Preds):-
        functor(Goal,N,A),
        ¥+(builtin_pred(N,A)),
        ¥+(using_pred(N,A)),
        member([N/A/(T/D)|Defs],Preds),
```

```
        T==sl,
        (cut_free_defs(Defs),
         construct_disjunctive_terms(Goal,Defs,New_goal,Preds,N,A);
         determinism_list(Pre_goals,Preds),
         construct_disjunctive_terms(Goal,Defs,New_goal,Preds,N,A)),
        !.
inline_expansion_body1_sp(Goal,Goal,_,_):-
        !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% inline_expansion

inline_expansion((Goal,Rest),(New_goal,Result),Preds):-
        functor(Goal,N,A),
        ¥+(builtin_pred(N,A)),
        ¥+(using_pred(N,A)),
        member([N/A/(T/D)|Defs],Preds),
        construct_disjunctive_terms(Goal,Defs,New_goal,Preds,N,A),
        inline_expansion(Rest,Result,Preds),
        !.
inline_expansion((Goal,Rest),(Goal,Result),Preds):-
        inline_expansion(Rest,Result,Preds),
        !.
inline_expansion(Goal,New_goal,Preds):-
        functor(Goal,N,A),
        ¥+(builtin_pred(N,A)),
        ¥+(using_pred(N,A)),
        member([N/A/(T/D)|Defs],Preds),
        construct_disjunctive_terms(Goal,Defs,New_goal,Preds,N,A),
        !.
inline_expansion(Goal,Goal,_):-
        !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% construct_disjunctive_terms

construct_disjunctive_terms(Goal,Defs,New_goal,Preds,N,A):-
        construct_disjunctive_term(Goal,Defs,New_goal1),
        del_redundant_parentheses(New_goal1,New_goal2),
        red_unexecutable_part(New_goal2,New_goal3),
        red_true_fail(New_goal3,New_goal4),
        (Goal=New_goal1,
         New_goal=New_goal4;
         asserta(using_pred(N,A)),
         inline_expansion_cf(New_goal4,New_goal,Preds)),
        !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% inline_expansion_cf

inline_expansion_cf((Goal,Rest),(New_goal,Result),Preds):-
        functor(Goal,N,A),
        ¥+(builtin_pred(N,A)),
        ¥+(using_pred(N,A)),
        member([N/A/(T/D)|Defs],Preds),
        cut_free_defs(Defs),
        construct_disjunctive_terms(Goal,Defs,New_goal,Preds,N,A),
        inline_expansion_cf(Rest,Result,Preds),
        !.
```



```
inline_expansion_cf((Goal,Rest),(Goal,Result),Preds):-
        inline_expansion_cf(Rest,Result,Preds),
        !.
inline_expansion_cf(Goal,New_goal,Preds):-
        functor(Goal,N,A),
        ¥+(builtin_pred(N,A)),
        ¥+(using_pred(N,A)),
        member([N/A/(T/D)|Defs],Preds),
        cut_free_defs(Defs),
        construct_disjunctive_terms(Goal,Defs,New_goal,Preds,N,A),
        !.
inline_expansion_cf(Goal,Goal,_):-
        !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% cut_free_defs

cut_free_defs([]):-
        !.
cut_free_defs([Def|Rest]):-
        !,
        cut_free_def(Def),
        !,
        cut_free_defs(Rest),
        !.
cut_free_def((H:-B)):-
        !,
        cut_free_goal(B),
        !.
cut_free_def(H):-
        !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% a_determinism_list

a_determinism_list([!|Rest],Preds):-
        !.
a_determinism_list([Goal|Rest],Preds):-
        functor(Goal,N,A),
        !,
        asserta(present_pred(N,A)),
        determinism_goal(Goal,N,A,Preds),
        !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% determinism_list

determinism_list([],Preds):-
        !.
determinism_list([!|Rest],Preds):-
        !.
determinism_list([Goal|Rest],Preds):-
        functor(Goal,N,A),
        !,
        asserta(present_pred(N,A)),
        determinism_goal(Goal,N,A,Preds),
        !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  inline_expansion_defs2
```

```
inline_expansion_defs2([],[],_):-!.
inline_expansion_defs2([Def|[]],[New_def|[]],Preds):-
        inline_expansion_def2_sp(Def,New_def,Preds),
        !.
inline_expansion_defs2([Def|Rest],[New_def|Result],Preds):-
        inline_expansion_def2(Def,New_def,Preds),
        inline_expansion_defs2(Rest,Result,Preds),
        !.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  inline_expansion_def2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

inline_expansion_def2((H:-B),(H:-New_B),Preds):-
        inline_expansion_body2(B,New_B,[],Preds),
        !.
inline_expansion_def2(H,H,_):-!.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  inline_expansion_body2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

inline_expansion_body2((Goal,Rest),(New_goal,Result),Pre_goals,Preds):-
        functor(Goal,N,A),
        ¥+(builtin_pred(N,A)),
        ¥+(using_pred(N,A)),
        member([N/A/(T/D)|Defs],Preds),
        T==sl,
        (cut_free_defs(Defs),
         construct_disjunctive_terms(Goal,Defs,New_goal,Preds,N,A);
         a_determinism_list(Pre_goals,Preds),
         construct_disjunctive_terms(Goal,Defs,New_goal,Preds,N,A)),
        inline_expansion_body2(Rest,Result,[Goal|Pre_goals],Preds),
        !.
inline_expansion_body2((Goal,Rest),(Goal,Result),Pre_goals,Preds):-
        inline_expansion_body2(Rest,Result,[Goal|Pre_goals],Preds),
        !.
inline_expansion_body2(Goal,New_goal,Pre_goals,Preds):-
        functor(Goal,N,A),
        ¥+(builtin_pred(N,A)),
        ¥+(using_pred(N,A)),
        member([N/A/(T/D)|Defs],Preds),
        T==sl,
        (cut_free_defs(Defs),
         construct_disjunctive_terms(Goal,Defs,New_goal,Preds,N,A);
         a_determinism_list(Pre_goals,Preds),
         construct_disjunctive_terms(Goal,Defs,New_goal,Preds,N,A)),
        !.
inline_expansion_body2(Goal,Goal,_,_):-
        !.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  inline_expansion_def2_sp
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

inline_expansion_def2_sp((H:-B),(H:-New_B),Preds):-
        inline_expansion_body2_sp(B,New_B,[],Preds),
        !.
inline_expansion_def2_sp(H,H,_):-!.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  inline_expansion_body2_sp
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```



```
inline_expansion_body2_sp((Goal,Rest),(New_goal,Result),Pre_goals,Preds):-
        functor(Goal,N,A),
        ¥+(builtin_pred(N,A)),
        ¥+(using_pred(N,A)),
        member([N/A/(T/D)|Defs],Preds),
        T==sl,
        (cut_free_defs(Defs),
         construct_disjunctive_terms(Goal,Defs,New_goal,Preds,N,A);
         determinism_list(Pre_goals,Preds),
         construct_disjunctive_terms(Goal,Defs,New_goal,Preds,N,A)),
        inline_expansion_body2_sp(Rest,Result,[Goal|Pre_goals],Preds),
        !.
inline_expansion_body2_sp((Goal,Rest),(Goal,Result),Pre_goals,Preds):-
        inline_expansion_body2_sp(Rest,Result,[Goal|Pre_goals],Preds),
        !.
inline_expansion_body2_sp(Goal,New_goal,Pre_goals,Preds):-
        functor(Goal,N,A),
        ¥+(builtin_pred(N,A)),
        ¥+(using_pred(N,A)),
        member([N/A/(T/D)|Defs],Preds),
        T==sl,
        (cut_free_defs(Defs),
         construct_disjunctive_terms(Goal,Defs,New_goal,Preds,N,A);
         determinism_list(Pre_goals,Preds),
         construct_disjunctive_terms(Goal,Defs,New_goal,Preds,N,A)),
        !.
inline_expansion_body2_sp(Goal,Goal,_,_):-

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% builtin_pred

builtin_pred(call,1).
builtin_pred(op,3).
builtin_pred(write,1).
builtin_pred(display,1).
builtin_pred(nl,0).
builtin_pred(read,1).
builtin_pred(get,1).
builtin_pred(ttyget,1).
builtin_pred(get0,1).
builtin_pred(ttyget0,1).
builtin_pred(put,1).
builtin_pred(ttyput,1).
builtin_pred(var,1).
builtin_pred(nonvar,1).
builtin_pred(atom,1).
builtin_pred(atomic,1).
builtin_pred(integer,1).
builtin_pred(true,0).
builtin_pred(fail,0).
builtin_pred(is,2).
builtin_pred('=',2).
builtin_pred('=..',2).
builtin_pred('==',2).
builtin_pred('<',2).
builtin_pred('>',2).
```

```
builtin_pred('=<',2).
builtin_pred('>=',2).
builtin_pred(bagof,3).
builtin_pred(functor,3).
builtin_pred(clause,2).
builtin_pred('¥+',1).
builtin_pred('!',0).
builtin_pred(see,1).
builtin_pred(seeing,1).
builtin_pred(seen,0).
builtin_pred(tell,1).
builtin_pred(telling,1).
builtin_pred(told,0).
```


付録　G （部分ユニフィケーション述語）

```
% Most-general unifier of two terms is represented by a set of equations.
% For example, unifiability of the terms
%                 f(X,g(Y,X,c)),
%                 f(h(Z),g(Z,h(d),c))
% is equivalently transformed into a Prolog expression
%                 X = h(d), Y = d, Z = d.

% apply_partial_unif applies partial_unifs to a predicate definition.
apply_partial_unif([HIT],[RIS]):-
     partial_unifs(H,R),
     apply_partial_unif(T,S),!.
apply_partial_unif([],[]):-!..

% partial_unifs partially evaluates equations in a clause.
partial_unifs((H:-B),(H:-R)):-
     unify_body(B,R).
partial_unifs(Head,Head).

% unify_body partially evaluates equations in a body.
unify_body(((B1,B2),B3),R):-
     unify_body((B1,B2,B3),R).
unify_body(((B1;B2);B3),R):-
     unify_body((B1;B2;B3),R).
unify_body((T1=T2),R):-
     partial_unif(T1,T2,R);R=fail.
unify_body((Bh,Bt),(Rh,Rt)):-
     unify_body(Bh,Rh),
     unify_body(Bt,Rt).
unify_body((Bh;Bt),(Rh;Rt)):-
     unify_body(Bh,Rh),
     unify_body(Bt,Rt).
unify_body((Bh->Bt),(Rh->Rt)):-
     unify_body(Bh,Rh),
     unify_body(Bt,Rt).
unify_body(Atomic_pred,Atomic_pred).

% partial_unif unifies two terms, resulting a list of equations.
partial_unif(T1,T2,R):-
     equation(1,[T1=T2],R1),bra_to_paren(R1,R),!.

% This is an implementation of Martelli and Monteiro's unification algorithm
equation(0,E,E).
equation(N,[],true).
equation(N,E,R):-!,term_red(M1,E,R1),
                   exchange(M2,R1,R2),
                   erase(M3,R2,R3),
                   !,var_elim(M4,[],R3,R4),
                   M is M1+M2+M3+M4,
```

```
                    equation(M,R4,R).

exchange(N,[T1=T2|T],[T2=T1|T]):-var(T2),nonvar(T1),N is 1.
exchange(N,[H|T],[H|R]):-exchange(N,T,R).
exchange(N,[],[]):-N is 0.

erase(N,[X=Y|T],T):-X==Y,N is 1.
erase(N,[H|T],[H|R]):-erase(N,T,R).
erase(N,[],[]):-N is 0.

term_red(N,[T1=T2|T],R):-nonvar(T1),nonvar(T2),
                         T1=..[F|Arg1],T2=..[G|Arg2],!,F==G,
                         length(Arg1,N1),length(Arg2,N2),!,N1=:=N2,
                         tr1([],Arg1,Arg2,Z),append(T,Z,R),N is 1.

tr1(L,[],[],L).
tr1(L,[H1|T1],[H2|T2],R):-tr1([H1=H2|L],T1,T2,R).

term_red(N,[H|T],[H|R]):-term_red(N,T,R).
term_red(N,[],[]):-N is 0.

var_elim(N,[],[X=T],[X=T]):-!,notoccur(X,T),N is 0.
var_elim(N,L,[H|T1],R):-H=(X=T),var(X),append(L,T1,Z),
                        ¥+ notoccur(X,Z),
                        !,notoccur(X,T),
                        subst(Z,X,T,T2),
                        append(T2,[X=T],R),
                        N is 1.
var_elim(N,L,[H|T],R):-var_elim(N,[H|L],T,R).
var_elim(N,L,[],L):-N is 0.

% bra_to_paren converts a list notation to a parenthesized notation.
bra_to_paren(true,true):-!.
bra_to_paren([L],L):-!.
bra_to_paren([H|T],(H,R)):-bra_to_paren(T,R),!.
```


付録　H　（不実行部分の除去述語）

```
% ----------------- DEL-UNEXECUTABLE-PART --------------------------------
% When there exists a 'fail' predicate in a sequence of conjuncts, the
% goals preceded by it may be deleted.

% apply_del_unexecutable_part applies del_unexecutable_part to a
% predicate definition.
apply_del_unexecutable_part([H|T],[R|S]):-
     del_unexecutable_part(H,R),
     apply_del_unexecutable_part(T,S),!.
apply_del_unexecutable_part([],[]):-!.

% del_unexecutable_part deletes such unexecutable goals preceded by a 'fail
% predicate in a clause.
del_unexecutable_part((P:-Body),(P:-New_body)):-
     red_unexecutable_part(Body,New_body),!.
del_unexecutable_part(Atomic_pred,Atomic_pred):-!.

% red_unexecutable_part deletes the unexecutable goals in the body of a
% clause.

red_unexecutable_part(((B1,B2),B3),New_body):-
     red_unexecutable_part((B1,B2,B3),New_body).
red_unexecutable_part(((B1;B2);B3),New_body):-
     red_unexecutable_part((B1;B2;B3),New_body).
red_unexecutable_part((fail,Rest),fail).
red_unexecutable_part((Bh,Bt),(Rh,Rt)):-
     red_unexecutable_part(Bh,Rh),
     red_unexecutable_part(Bt,Rt).
red_unexecutable_part((Bh;Bt),(Rh;Rt)):-
     red_unexecutable_part(Bh,Rh),
     red_unexecutable_part(Bt,Rt).
red_unexecutable_part((Bh->Bt1;Bt2),(Rh->Rt1;Rt2)):-
     red_unexecutable_part(Bh,Rh),
     red_unexecutable_part(Bt1,Rt1),
     red_unexecutable_part(Bt2,Rt2).
red_unexecutable_part((Bh->Bt),(Rh->Rt)):-
     red_unexecutable_part(Bh,Rh),
     red_unexecutable_part(Bt,Rt).
red_unexecutable_part(Atomic_pred,Atomic_pred).
```

付録　I（等式代入による変数除去述語）

```
% apply_equality_subst applies equality_subst to a predicate definition.
apply_equality_subst([(H:-Body)|T],[R|S]):-
     del_redundant_parentheses(Body,Body1),
     equality_subst((H:-Body1),R),
     apply_equality_subst(T,S),!.
apply_equality_subst([Head|T],[Head|R]):-
     apply_equality_subst(T,R),!.
apply_equality_subst([],[]):-!.

% equality_subst is a rule of equality substitution.
equality_subst((H:-X=T),Result) :-
   var(X),
   subst(H,X,T,Result),!.
equality_subst((H:-Goals1;Goals2),(H:-Result1;Result2)) :-
   equality_subst_O(Goals1,Result1),
   equality_subst_O(Goals2,Result2),!.
equality_subst((H:-Body),Result) :-
   search_and_del_eq(Body,X,T,New_body),
   subst((H:-New_body),X,T,Pre_result),
   equality_subst(Pre_result,Result),!.
equality_subst(Clause,Clause) :- !.

% equality_subst_O applies equality rule to conjuncts in a disjunctive term.
equality_subst_O((Goals1;Goals2),(Result1;Result2)) :-
   equality_subst_O(Goals1,Result1),
   equality_subst_O(Goals2,Result2),!.
equality_subst_O(Goals,Result) :-
   search_and_del_eq(Goals,X,T,New_goals),
   subst(New_goals,X,T,Pre_result),
   equality_subst_O(Pre_result,Result),!.
equality_subst_O(Goals,Goals) :- !.

% search_and_del_eq finds an equation from the sequence of conjunctive
% goals and deletes it.
search_and_del_eq(((Goal1;Goal2)),X,T,((Result1;Result2))):-
   equality_subst_O(Goal1,Result1),
   equality_subst_O(Goal2,Result2).
search_and_del_eq(((Goal1;Goal2),Rest),X,T,((Result1;Result2),Result)):-
   equality_subst_O(Goal1,Result1),
   equality_subst_O(Goal2,Result2),
   search_and_del_eq(Rest,X,T,Result).
search_and_del_eq((X=T,Rest),X,T,Rest) :-
   var(X),!.
search_and_del_eq((Goal,X=T),X,T,Goal) :-!,
   cut_free_goal(Goal),
   var(X),!.
search_and_del_eq((Goal,Rest),X,T,(Goal,Result)) :-
   cut_free_goal(Goal),
   search_and_del_eq(Rest,X,T,Result),!.
```


付録　J（述語呼び出しの決定性判定問題の非可解性）

第４章，４．４節で述語の決定性を判定する問題が決定不能であることを示した．ここでは，論文〔Sawamura 90a〕に基づき，この問題に対してより形式的な二つの証明を与える．一つは，帰納関数論の言葉，結果を用いたいわゆる対角線論法による証明であり，他の一つは帰納関数論におけるRiceの定理を直接適用することによって得られるものである．

定理（述語呼び出しの決定性判定問題の非可解性）．

任意の述語呼び出しに対して，それが決定的であるか否かを判定するアルゴリズムは存在しない．

（１）対角線論法による証明

述語定義と述語呼び出しの数え上げが与えられているものとする．Pn, qmをそれぞれこれらの数え上げにおけるｎ番目の述語定義，m番目の述語呼び出しとする．次のような述語det2を実現するアルゴリズムが存在すると仮定する（以下の図参照）．

$$\mathrm{det2}(n, m) = \begin{cases} \text{qmがPnの下で決定的のとき、成功} \\ \\ \text{そうでなければ、失敗} \end{cases}$$

この述語を用いて，次のように述語ｒを定義する．

$$r(X) = \begin{cases} \text{det2(X,X)=失敗のとき、成功} \\ \\ \text{det2(X,X)=成功のとき、非停止} \end{cases}$$

この述語 r をホーンプログラムで決定的であるように定義することができる．例えば，次のように定義できる．

```
r(X) :- det2(X,X), repeat.
r(X) :- true.
repeat :- repeat.
```

ここで，この述語定義の数え上げにおけるインデックス(ゲーデル数) を z とすると， r の定義から，

qzがPzの下で決定的に成功する　<=>　det2(z,z) = 失敗

他方，述語det2の定義より，

det2(z,z) = 失敗　<=>　qzはPzの下で決定的でない．

これは矛盾である．故に，決定性を判定するアルゴリズムは存在しない．■

P_n \ q_m	0	1	…	m	…
0	det(0,0)	det(0,1)	…	det(0,m)	…
1	det(1,0)	det(1,1)	…	det(1,m)	…
:	:	:		:	
m	det(m,0)	det(m,1)	…	det(m,m)	…
:	:	:		:	
z					
:					

$$P_z = r(X) = \begin{cases} \text{success}, & \text{if det2(X,X) = failure} \\ \text{divergent}, & \text{if det2(X,X) = success} \end{cases}$$

図　対角線論法

以上の証明では，Prologの実行メカニズムとか，det2のようなメタ述語が用いられていた．次に，このような証明の資源には一切頼らない証明を与える．その代わりここでは，一階のホーン論理はすべての計算可能関数を表現できるという事実，および次のRiceの定理を用いる〔Sawamura 90a〕．

Riceの定理

Ｒをすべての一変数の部分帰納関数の集合，ＣをＲの真部分集合，すなわちＣ⊂Ｒとする．この時，集合 $\{x \mid \varphi_x \in C\}$ はいかなる帰納的特性関数ももたない　<=>　$\phi \neq C \subset R$.

（2）Riceの定理を適用した証明

Ωをすべての決定的な一変数数論的述語の集合とする．次の集合を考える：

$\Omega' = \{\varphi \mid P \in \Omega$ かつ任意の述語呼び出し p(A)に対して，

$$\varphi(A) = \begin{cases} 1, \text{p(A)がPの下で成功するとき，} \\ 0, \text{p(A)がPの下で失敗するとき，} \\ \text{発散, p(A)がPの下で発散するとき}\} \end{cases}$$

各 $\varphi \in \Omega'$はある数え上げの下で，φ_nと表現される．Rice の定理により，

$\phi \neq \Omega' \subset R$ <=> $\{n \mid \varphi_n \in \Omega'\}$はいかなる帰納的特性関数ももたない．

ゆえに，決定的述語の部分集合であるΩに対しても，いかなる帰納的特性関数も存在しない．■