ICOT Technical Report: TR-154

TR-154

Prolog プログラムの最適化

沢村　一（富士通）

January, 1986

ICOT

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

(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

国際情報社会科学研究所研究報告

PROLOGプログラムの最適化

Source to Source Optimization of PROLOG Programs

沢村 一

Hajime Sawamura

昭和61年 1月20日

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考案し、それに基づいたオプティマイザを設計、試作した。特に、述語の決定性が、Prologのような非決定性のプログラム言語の最適化を考える上で重要な概念であることが強調されている。

オプティマイザの設計原理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（呼び出し）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、述語呼び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。

これらの最適化技法を一つに統合することによってPrologオプティマイザが試作された。いくつかの典型的な例による性能評価実験は、我々のオプティマイザが平均20% 位の速度効率を達成することを示した。

本報告の後半では、Prologプログラムの最適化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られている。

ABSTRACT

A source-level optimizer for Prolog, a nondeterministic logic programming language, is designed and implemented for the purpose of Prolog program improvement.

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, del etion 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.

Some theoretical considerations to justify our methodology for optimizing Pr olog programs at the source level and discussions about related works are added to the latter half of the paper.

目次

第１章　はじめに

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るかもしれない〔1〕。

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と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なる。

本書では、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、それは論理の手続き的解釈から産みだされた言語であり、それゆえにプログラムをかなり宣言的に表現することを可能にしているからである。

本書では、純Prologに制限することなく現実のProlog〔2〕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、本書のアプローチはプログラムの変換の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〔2〕を見られたい。）　一言でいうならば、本書の最適化方法は weak but general であり、strong but specializedなプログラム変換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、およ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、

① ゴールのその定義体による展開、

② 論理式としての節の変形

③ 述語定義体単位の変形

などが考えられる。①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。②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。③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。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、言語処理系のそのことに起因する空間的・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有用となる。

本書の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

まずPrologプログラムの最適化に当たって、予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。現在のシステムではそれらはプログラムの型と決定性である。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（第二章で定義されるａ－決定性とｒ－決定性）にしたがってそれが決定的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。これによって知的バックトラックの簡単なケースが達成されている。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二つからなっている：

① サブルーチンリンクのオーバヘッドの除去、

② 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言う方法で行われる。このときPrologプログラムの実行メカニズム"最左及び深さ優先"にしたがわねばならない。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。

本書ではさまざまな局所的最適化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することも、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手法には次のようなものがある。

(a) 部分的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、ゴールの除去、

(d) 等式ゴール列の統合化、

(e) 節の分解。

以下の記述は、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、本書は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、本書の目的はPrologプログラムをソースレベルで最適化するさいの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、本書での関心は、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である。

本書の構成は次のようである。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（呼び出し）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のに有用な結果、議論を含む。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オプティマイザのアウトラインを記述している。第6章はいくつかの最適化の例とその時間評価を含む。それによって本書の方法の有効性が示される。第7章はこの分野の最近の研究の現状を簡単に紹介している。ここで、われわれの方法との定性的な相違点が明らかにされる。最後の章は今後の課題と展望を述べている。

第２章 記法と定義

ここでは、Prolog〔２〕の構文法とその実行的意味に関する知識を仮定する。そして、以下で導入する最適化図式を記述するのに必要な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アリティ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。

以下、Prologの統語要素の上を動く統語変数に対して区別された記号を用いることに注意。

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は述語名あるいは変数を命題を表す。

(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が空列のときそれはゴール "true" を表す。(添字付) の文字A 、B はPrologの項の非空な列を表す。

(3)　Γ(X) を変数X が起こるゴールの列を表すものとするとき、Γ(t) はΓ(X) の中の変数X のすべての生起に項t を代入してできるゴールの列を表す。

(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。

2．2　最適化図式

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。

〔最適化図式〕

最適化図式とは、次のような形式の図式をいう。

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

ここで、Φ、Ψをそれぞれ最適化図式の上式、下式と呼ぶ。

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 と同じ述語名及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 :- Γ1, P1.

⋮

Hi :- Γi, Pi.

⋮

Hn :- Γn, Pn.

とする、ここで、

（i）Γ1, ・・・, Γn には名前p の述語呼出しはない。さらに、

（ii）P1, ・・・, Pnはアトミックなゴールで、それらの中には名前がp の述語呼び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、制限（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、 " Hi :-Γi,G1. " 及び " Hi :-Γi,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（呼び出し、またはゴール）の決定性

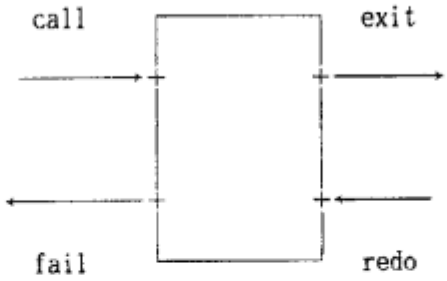
述語（呼出し）の決定性は非決定性プログラムを最適化するさいに重要な役割を果たす。実際、インライン展開、カットの自動挿入、ある種の局所的最適化手法においては述語（呼出し）の決定性は本質的な条件となっている。本書のPrologオプティマイザの方法に必要な決定性の概念を次のように定義する。

【述語呼出しの決定性】

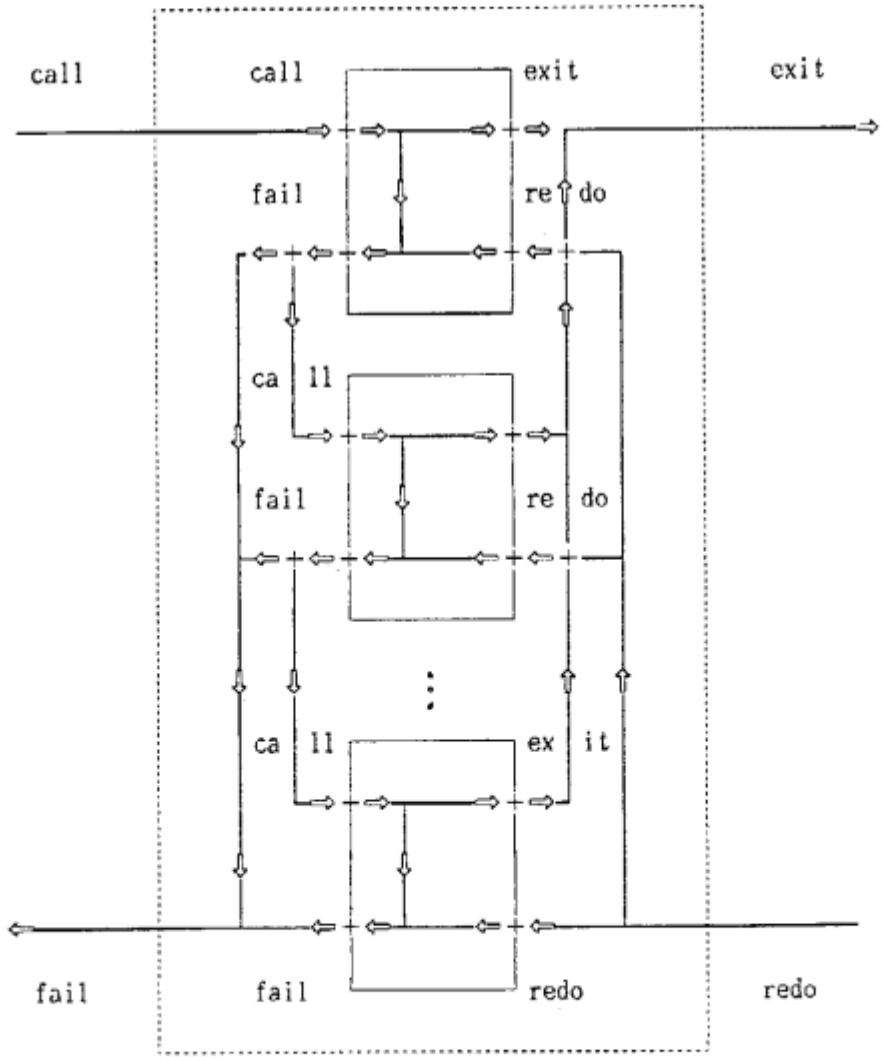
述語呼出しが決定的であるとは、それが呼ばれたとき、その定義体の高々一つの節で成功し、バックトラックされたとき、決して成功することはないことをいう。

この定義を箱による制御フローモデル〔4〕で示してみよう。ひとつのゴールの非決定的な計算は次のように示される。一つのゴールへの制御の出入りを次のような腕が四本

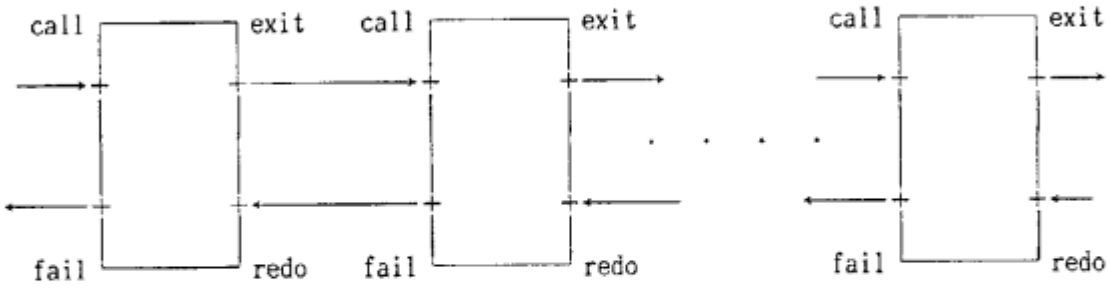
ある箱で表現することにする。



このとき、代替節が複数ある場合、一つのゴールを決定的に達成する場合の制御の可能な流れの全体の様子は次のように表現することができる。これはそのゴールの定義体の排他的OR結合を表すものと考えることができる。



勿論これは帰納的な図を表しているつもりである。即ち、Prologの質問及び定義節のボディのゴール列は腕が四本あるこのような箱がAND 結合で次のように並んだものである。



上の我々の定義と実質的に同じものは〔5、6〕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)が関数的であるとする決定性の定義よりも一般的な定義である。すなわち、どの引数が入力で、どの引数が出力であるかということには一切関わらない定義である〔7〕。

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章で述べられる）、ここでは互いに関係する二つの決定可能な述語（呼び出し）の部分クラスを定義する。一般に、述語呼び出しの成功、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、決定的な述語呼び出しの引数依存性を反映し、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、カット記号は存在するなら最右とする。

⋮

Hn :- Γn.

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 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、p(A)はHiと単一化可能で、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:

① プログラムが相互帰納的に定義されている場合でも決定的と判定されることがある。例えば、

p :- Γ,!,q.

q :- Θ,p, Δ,!.

では、呼び出しp 、q 共にa-決定的である。より一般的には次のようにいうことができる。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、カット記号は存在するなら最右とする。

⋮

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の間のクロス・レファレンス(cross reference) を考えると（これは一般に有向グラフになる）、これにサイクルがないことが決定性判定のための必要条件として、上の定義には含まれている。すなわち、これは定義の帰納性を保証する。

② 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、現在は簡単さのためにそれらは単に非決定的ゴールとして扱われている。(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。

③ 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①の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。

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がう。(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依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-決定性なゴールは、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）。これらの観察より、次の系が成立する。

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、決定的述語呼出しが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、次のプログラム例から見られるように系 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。このような個々の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を意図した。

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- 決定性とr-決定性の定義の条件(3)は、仮定より排除され、したがって述語p はa-決定的となる。(<=)　明らか。■

補題　6．任意の引数列A に対してゴールp(A)が成功する（証明可能である）ことと、X を変数列とするときp(X)が成功する（証明可能である）ということは同値である。

証明 :　一階論理の次の定理による。

⊢∀A.p(A)　<=> ⊢p(X) ; Xは自由変数である。■

系　7．X を変数の列とする。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と補題６による。■

系４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。

述語呼出 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、カット記号は存在するなら最右とする。

⋮

Hn :- Γn.

このとき、各Hi(1≦i ≦n)に対して、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、p(A)はHiと単一化可能で、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】

ゴールp(X)がr-決定的なとき、a-決定的と呼ばれる。

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。

例　1．述語pは a- 決定的である。

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

すなわち、ゴールp(X)はr-決定的である。

例　2．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。

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

これまでPrologプログラムの最適化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いと考えられる。

実際のPrologプログラミングでは、プログラムは決定的に書くか、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〔8〕。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的な性質(eventuality propertity) や不変的な性質(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る。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Proiog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また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。

この章の最後に、我々の決定性の定義の有力な二つの拡張法について述べておこう。

(1) 述語の型推論〔例えば、９、１０〕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。

(2) (1)と関連するが、r-決定性の定義において、条件(3)の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、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、カット記号は存在するなら最右とする。
 ⋮
Hn :- Γn.
```

第３章 PROLOGプログラムのための最適化手法

この章では、さまざまな最適化図式およびその適用条件を記述する。

３．１ 部分的ユニフィケーション

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等式ゴールとして残る。ここで等式ゴール列と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ものである。部分的ユニフィケーションは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的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利用される、例えば、第３．３．７節で述べられ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３．３．８節で述べられ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。

【最適化図式 １】

二つの項の単一化可能性を表現する、Prologの等式ゴールは等式ゴール列へと次のように変換される。

$$\frac{f(\cdots) = f(\cdots)}{X1 = T1, \cdots, Xn = Tn} \quad , \quad \frac{f(\cdots) = f(\cdots)}{fail}$$

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。こ

の変換の詳細は文献〔１１〕を参照されたい。

例　3.

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

$$X = h(d),\ Y = d,\ Z = d$$

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は重要な役割を果たしている。これを次の場合において行う。

【最適化図式　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。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 4.

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

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

ここで、述語呼出しp(X)は例1 の定義節を呼んでいる。

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。というのは、次の例に見られるように、呼び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.
```

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。以下ではこのようなインライン展開の問題を避けるための条件が与えられる。

インライン展開におけるカットの問題はDEC-10 Prolog 〔2〕における特有の現象に見えるかもしれない。確かに、C-Prolog〔１２〕、KLO 〔１３〕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、本書第３．３．７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作用を持つ変換であるために以下で述べられるようなインライン展開の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 (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 5.

```
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〔14〕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。

【最適化図式　4】

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

```
G = H1',Γi' -> Δi' ;  ・・・  ; G = Hn',Γn' -> Δn'  あるいは
                                G = Hn',Γn'  あるいは
                                G = Hn'
Hi :- 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   :     〃          "Hi :- Γi,!." のとき
G = Hi' -> true       :     〃          "Hi :- !." のとき。
```

例　6.

```
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。

【最適化図式　5】

```
Hl :- Γl.
  ⋮
Hi :- Γi  p(A), Δi.
  ⋮
Hm :- Γm.
p(Al) :- Θ1.
  ⋮            Θi(1 ≦i ≦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。
```

```
Hl :- Γl.
  ⋮
Hi :- Γi,
      ( p (A)= p(Al') , Θ1' ; ... ; p(A) = p(An') , Θn') ,
      Δi .
  ⋮
Hm :- Γm.
p(Al) :- 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)がなくとも正しい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 7.

```
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( [] ,L,L),! ;
  append( [b ] ,Y,Z) = append( [X |L1] ,L2, [X |L3] ),
  append(L1,L2,L3),!).
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。

第二節はあとで述べられる最適化手法を使うとさらに簡単化されて次のようになる:

```
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が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。本書では、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られる。この中では最適化図式4 の組み込みは行われていない。

3．3．3　直線的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、直線的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もち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、その定義を、

```
H1 :- Γ1.
⋮
```

Hi :- Γi, [!], Δi., ここでカット記号はもし存在するなら最右とする。

⋮

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 8. 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,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帰納的な言語であるから、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ァであり、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帰納的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が終えた後、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第３．３．３節と同様に展開する。

（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 を第３．３．３節と同様に展開する。

（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【最適化図式　3】によって一回展開する。

(2)　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（第２章で言及した意味において）保存される。

３．３．５　一般帰納型述語定義のインライン展開

一般帰納型述語定義を構成している各節の帰納的でないゴールは第３．３．３節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　述語定義の一般帰納性はインライン展開にって保存される。

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】

(1) ゴール Pは r- 決定的である。

(2) 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図式の反例のいくつかを示す。

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

は正しくない。というのは、上式のp が停止せず、下式のq が停止する場合を考えると同値でなくなる。

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

は正しくない。"var" はメタ述語である。

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

は正しくない。"write" は出力述語である。

次の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しめ示している。

(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。

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

というのは、上式では q(X) の X = aによる成功は "not(X = a)" を無限に繰り返すことになるが、下式では q(X) のX = b による成功は下式の実行を終わらせることになるからである。

次の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。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いえない。

(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" の部分には入ってはならない（条件を少し弱くいうと、"P ;Δ ; P" の各述語呼出しが a- あるいは r- 決定的である）

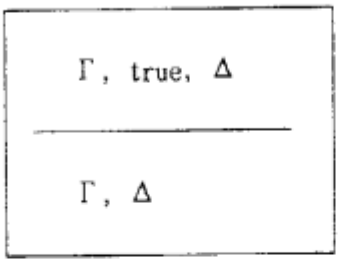
次の図式は条件(2)の意味を例示している。

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

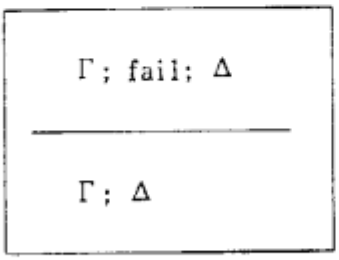
3．4．2 冗長な "true" 、"fail" 述語の除去

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。

【最適化図式 8—1 】



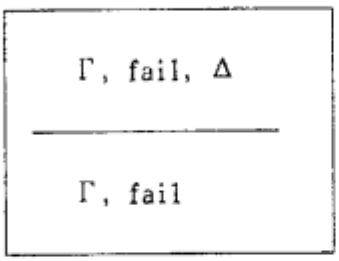
【最適化図式　8－2】



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。

【最適化図式　9】



しかしながら、次の双対な図式は明らかに妥当でない。

Γ; true; Δ

Γ; 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　9.

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

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。

【最適化図式　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(t1, ・・・, tn),  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。というのは、上式、下式共に、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。

【最適化図式　１１－２】

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

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の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。

【最適化図式　１２】

$$\frac{X1 = t1,\ \ \cdots,\ Xn = tn}{f(X1,\ \cdots,\ Xn) = f(t1,\ \cdots,\ tn)}$$

ここで、"f" は適当な関数記号である。

例　9．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）。

【最適化図式　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。

次の例は正しくない。

```
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）を定義節としてすべて数え上げることに相当する。

第４章　理論的考察

この章では、これまで述べてきたPrologプログラムの最適化手法に関する理論的考察を行う。これらは、信頼性の高いオプティマイザを実際に設計、試作するとき有用になるものである。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する一つの証明、及び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、第三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、またそれはゴールをインライン展開するための我々の判断基準ともなっていた。

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

インライン展開は述語定義体の各ゴールが、展開の条件をみたす限りPrologの実行順序にしたがって行われた。これらは次の命題によって停止することが保証される。

命題　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示せば十分である。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。■

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〔15〕。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手法である、dtl:【最適化図式　8－1】、dt2:【最適化図式　8－1】、du:【最適化図式9】を取り上げその適用順序について論ずる。他の最適化手法の適用順序については、ユーザプログラムの特性に応じてユーザの指示によって決めるようになっている。特に【最適化図式　13】はいつそれを適用すべきかは微妙である。

$$\text{dtl:}\ \frac{\Gamma, \text{true}, \Delta}{\Gamma,\ \Delta}\ 、\qquad \text{dt2:}\ \frac{\Gamma ; \text{fail} ; \Delta}{\Gamma ;\ \Delta}$$

ここで、Γ、Δは非空である。

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

ここで、Δは非空である。

dtを規則dtl かdt2 のいずれかを表すものとし、関係 dt->（下で定義される）を関係 dtl # 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）　iff　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 iff 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 iff dt 、du　のいずれもが列Γには適用可能ではない。

定義　4．　dt、duをそれぞれ列Γに適用することを dt(Γ) 、 du(Γ) と書く。

定理　4．　Γを列、Δをdt及びduに関する標準形とする。このとき、

$$\Gamma * \Delta \text{ iff } \Gamma ** \Delta$$

たたし、関係 ** は $\#_{m>0,\ n>0}$ $(du \text{ -> })^m(dt \text{ ->})^n$ である、

ここで、 $(du \text{ -> })^m(dt \text{ ->})^n$はそれぞれ関係 du 、dtのm、n重積を表す。

証明：

(<=) Γ、ΔをΓ * Δであるような列とする。dt、duの規則の形式から、任意の列に対して、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。したがって、一般性を失うことなく、始めに、列Γにduを高々N 回適用し、次に、dtを高々M 回適用することができる、ただし、N はΓの連言ゴール列に起こる述語 "fail" の数を、M は結果の列に起こる述語 "true" 及び "fail" の全数である。このようにして、標準形Δを得ることができる。以上の観察より、Γ $(du \text{ -> })^N(dt \text{ ->})^M$Δであるような数N とM が存在することがわかる。

(=>) 明らか。■

定理 4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。

命題　5．duに関する標準形は一意に定まる。

命題　6．dtに関する標準形は一意に定まる。

命題　7．du及びdtに関する標準形は一意に定まる。

証明: 定理7、命題5、命題6による。■

定理　8．任意の列に対して、関係＊はChurch-Rosser 性をもつ。

証明: 定理４、定理７による。■

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$(dt)^n$ $(n \geq 0)$、$(du)^m$ $(m \geq 0)$ として実現されている。そして、任意の列に対して、$(du)^m$ $(m \geq 0)$ が最初に適用され、その次に$(dt)^n$ $(n \geq 0)$ 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 〔１６〕、Dijkstra〔１７〕による言語、人工知能向き言語であるMicro-planner 〔１８〕は、最近のProlog、Concurrent Prolog 〔１９〕に加えてよく知られた非決定性言語である。

これらの言語で書かれたプログラムは二つの主要な意味で非決定的である。すなわち、don't care(DC)とdon't know (DK) である〔２０〕。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、それは非可解である〔２１〕。

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結果に帰着させることによって解かれることが多い〔２２、２３〕。以下で与えられる決定性の帰納的非可解性の証明は、Russell のパラドックスの如く、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〔２４、２５〕。

ここで、本書における決定性の定義を思い起こしておこう。

定義　5．述語呼出し（ゴール）が決定的であるとは、それが呼ばれたとき、その定義体の高々一つの節で成功し、バックトラックされたとき、決して成功することはないことをいう。

注意

①　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。

②　Kowalskiは決定性の判定は原理的に決定不能であると述べている〔２０〕。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。

$$Prog \rightarrow \forall x,y,z(F(x,y) \wedge F(x,z) \rightarrow y = z)$$

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、下の定理９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　9．任意の述語呼出しに対して、それが決定的述語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:

任意の述語呼び出し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:

① 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存在仮定は誤りであることを述べている。

② 証明の論証は述語det をdet(P,Defs) のように二項述語として明示的に書かれた場合にもあてはまる、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。

系　１０．任意の述語呼び出しに対して、それが非決定的述語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、決定的述語の判定アルゴリズムも存在することになり定理９に反する。 ■

系　１１．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９によって不可能である。■

系　１２．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９の証明の中で述語ｐを命題ｐと置き換えればよい。　■

系１２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　１３．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び出し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%(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、決定的でない。

（ii）det%(r) = 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、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

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１に示してある通りである。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、完全自動化のオプティマイザが実現されるであろう。

オプティマイザへの入力はDec system-10 Prolog〔２〕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

オプティマイザの全体の処理は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形で進行する（図１）。

オプティマイザプログラムはVAX11/780/UNIX上のCProlog 及びDEC-system10 Prologで書かれている。それは約60000 バイトの大きさである。プログラムに興味のある読者は〔２６〕を見られたい。

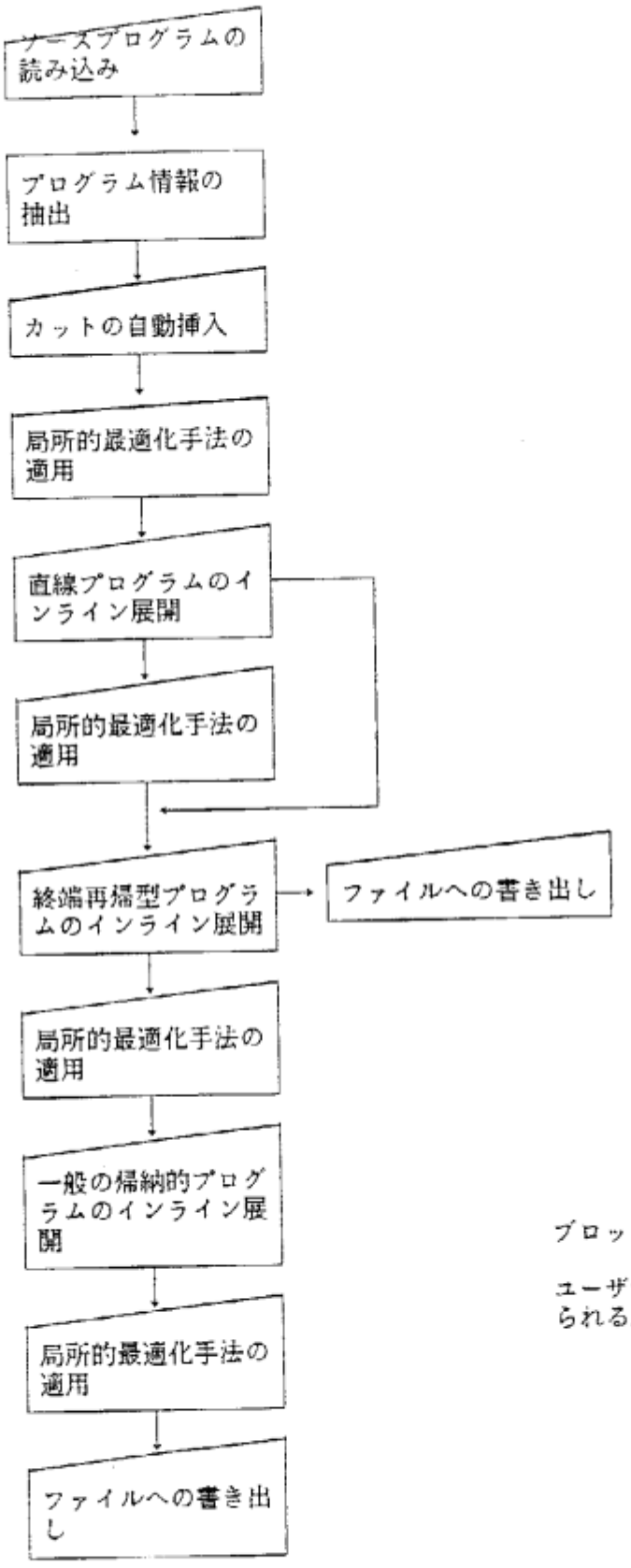


図1：Prologソースレベルオプティマイザの基本構成

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オプティマイザの動作例を示す。

```
reverse(X,Y) :- r(X,Y, [] ) ,!.
r( [] ,Z,Z).
r( [H |T ] ,W,Z) :- r(T,W, [H |Z ] ).
```

図2において、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 : 直線プログラム

tr : 終端再帰プログラム

d : a-決定的

rd : 【最適化図式 １３】（節の複数節への分解）

es : 【最適化図式 １１】（等式による代入）

```
| ?- 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/(sl/d),r/3/(tr/_432)]

reverse/2/(sl/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]))).

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/(sl/d),r/3/(tr/_432)]

reverse/2/(sl/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 : reverse1

Optimized programs written to the file : reverse1

PROLOG OPTIMIZER ends.

yes
```

図2：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、これまでに提案された個々の最適化手法のいくつかの評価、および各種の最適化手法を組み込んだ、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。

各種最適化手法の大まかな適用順序は図１に示した通りである。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であることはすでに述べたとおりである。

本報告の最適化法は、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、一つ一つの最適化手法を期待される効果という観点からみると、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る。したがって、このような効果を直接実現できるような処理系では大きな最適化効果が得られるが、異なる処理系、実行マシンによっては期待される効果が得られないこともある。

以下では、現在世の中で最も多く使われているDec system-10 Prologのインタプリタ〔2〕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次のPrologプログラムである〔27〕。

```
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

第5章のオプティマイザの動作例で用いたスタックを用いる述語 "reverse" を再び用いて、最適化の過程を詳しく見てみよう。

```
reverse(X,Y) :-  r (X,Y, [] ) .          (1)
r( [] ,Z,Z) .                            (2)
r( [H | T ] ,W,Z) :-  r (T,W, [H | Z ] ) .   (3)
```

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。

最適化後のプログラムに対しては、Dec system-10 のPrologインタプリタの下で約20％ の時間効率が得られている。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を生成す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〔２８〕を実現するプログラムを最適化する。

$$\frac{p(A) :- \Gamma, X = f(t1,\ldots,tn), X = f(r1,\ldots,rn), \Delta.}{p(A) :- \Gamma, f(t1,\ldots,tn) = f(r1,\ldots,rn), \Delta.}$$

ここで、変数X はp(A)、Γ、Δの部分には出現しない。

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、DEC-10 Prolog 上で約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。

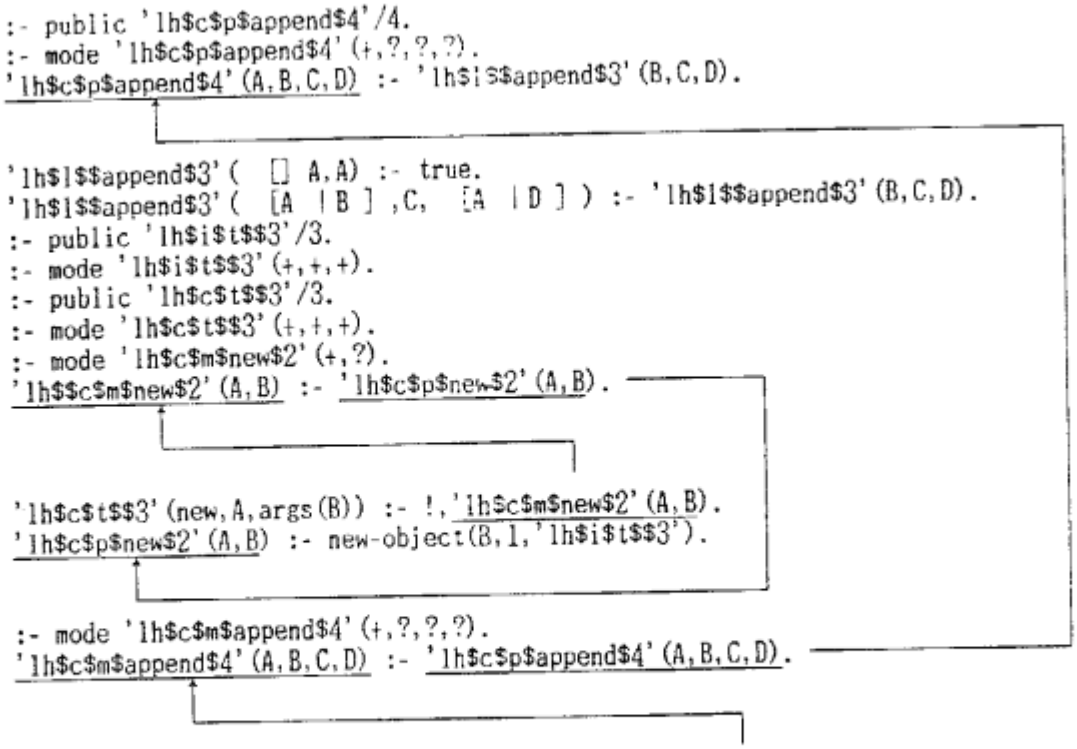
〔3〕ESP コードの最適化

ESP プログラム〔29〕はKLO(あるいは、Prolog)〔13〕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。



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、DEC-10 Prolog 上で約40% の時間効率が得られている。

〔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〔30〕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かを示すのが目的である。

次のようにリストを抽象データ型として定義する。

```
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の実行時間を、VAX11-780/UNIXのC-Prologで評価した結果約60% の時間効率を得ている。

いくつかの小規模プログラムによる実験例は次のことを示しているように思われる。「普通のプログラマが書いた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プログラムはこの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する我々のオプティマイザの有効性を確信させるのに十分であろう。

本章では、Prologオプティマイザの評価を、実際のマシン、言語処理系の上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、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致っていない。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効果について触れておこう。インライン展開は一般に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、それによるテキストの膨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 他の研究との関係

一般にプログラムの最適化に関する研究には多くの成果の積み重ねが知られている(例えば、〔１、３１、３２〕)。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、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、〔４〕）、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と同様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〔３〕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ィミィ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、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、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、相違点を明らかにする。

IC-Prolog 〔３３〕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〔３４〕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)=l1,l2
    partition( [x.l ] ,y) = [x.l1] ,l2 ← x≦y / l1,l2=partition(l,y).
    partition( [x.l ] ,y) = l1, [x.l2] ← x>y / l1,l2=partition(l,y).
    partition( [] ,y) = [] , [] .
```

近山〔２８〕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〔３０〕は、展開（unfold）とたたみ込み（fold）技法〔３５〕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で自然であるが、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ものの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 〔３６〕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〔３７〕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〔３８〕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〔３９〕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定義されたメタインタプリタで実行する方法を取っている。

Mellish〔６〕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における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〔４０〕は、対話による成長的プログラミング（incremetal 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、我々の方法では初めから質問があた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〔４１〕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。我々は本報告で、プログラマー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。概念的には、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対応し、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対応する。

Hogger〔４２〕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〔４３〕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は、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〔44〕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。

第８章 まとめ

本報告で、我々はPrologの様々な最適化手法を提案した。そして特に述語（述語呼び出し）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であることを強調した。一般に、述語（述語呼び出し）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い。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因子を与えることになる。

Prologオプティマイザは、Dec system-10 Prologを対象とし、プログラムの統語的側面からのみ最適化を行っている。したがって、これはweak but generalなオプティマイザにならざるを得ない(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）。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、意味的解析が必要となってくるであろう（例えば、型推論、知的バックトラック、データ / 制御フロー解析等）。

本報告の最適化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章で見たような効果が得られることが実証されたことは本報告のもう一つの寄与である。

今後の課題としては、Prologプログラムの同値性を公理的に、あるいは他の形式的な方法により証明する方法の確立とか、各種最適化技法の適用順序の分析といった理論的課題の他に、上で述べたような意味解析を伴った決定性の解析が重要な課題である。また、本報告のオプティマイザの機能を、第６章でも触れたように、Prologプログラムの変換技法として展開すること、及び知的プログラミング〔４５〕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。

謝辞

日頃御指導、御鞭撻をいただく北川敏男会長、及び榎本肇所長に感謝いたします。

本報告は論文〔7、46〕及びその後の成果〔47〕をまとめたものである。共同研究者であった竹島卓研究員、いくつかの局所的最適化法に関して協力してくれた加藤昭彦研究員に感謝します。また、オプティマイザに関して日頃熱心に討論してくれた横森貴研究員、南俊朗研究員にも感謝します。

岸本光弘氏（富士通研・AI研）はαPrologの開発者の立場から、本報告の最適化手法の有効性について貴重なコメントをくれた。

近山隆氏(ICOT)はPrologの最適化全般にわたって討論してくれた。また、第7章のESP の例を示唆してくれた。

R. Venken 氏(ベルギー・マネジメント研究所) は我々の研究報告〔46〕を詳細に読んで、最適化手法の一つ一つにわたり彼の最適化手法と対比したコメントを送ってくれた。その一部は本報告をまとめるにあたり有用であった。

なお、本研究の一部は第5世代コンピュータプロジェクトの一環としてICOTの委託で行ったものである。

参考文献

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

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

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

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

〔5〕 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.

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

〔7〕 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, Boston, Ma., pp. 200-207, 1985.

〔8〕 近山 隆 : Prologプログラミングスタイル、メモ、1983.

〔9〕 A. Ramsay : Type-checking in an untyped language, Int. J. on Man-Machine Studies, Vol. 20, pp. 157-167, 1984.

〔10〕 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.

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

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

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

〔14〕 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.

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

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

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

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

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

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

〔21〕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.

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

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

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

〔25〕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.

〔26〕富士通㈱編：Prologソースレベルオプティマイザ詳細仕様書、昭和60年 3月.

〔27〕国藤：私信、1983.

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

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

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

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

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

〔３３〕K. L. Clark and F. G. McCabe : The control facilities of IC-Prolog, 1979, in Expert Systems in the Micro Electronic Age, edited by P. Michie , Edinburgh Univ. Press, pp. 122-149, 1979.

〔３４〕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.

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

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

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

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

〔39〕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.

〔40〕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.

〔41〕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.

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

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

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

〔45〕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.

〔46〕H. Sawamura, T. Takeshima and A. Kato, Source-level optimization techniques for Prolog, IIAS R.R. No. 52, ICOT-TR-0091, 1985, also submitted to J. of New Generation Computing.

〔４７〕沢村　一：PROLOG 述語の決定性、ソフトウェア科学会論文誌投稿予定、昭和61年 2月.