TR-047

Prologソースレベル・オプティマイザ
—最適化手法のカタログ—

沢村　一、竹島　卓、加藤昭彦
（富士通）

1984. 3

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191～5
Telex ICOT J32964

PROLOG
ソースレベル・オプティマイザ
— 最適化手法のカタログ —

沢村　一・竹島　卓・加藤昭彦（富士通国際研）

1．総論　— Prologソースレベル・オプティマイザ —

プログラムの最適化（optimization）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な変換をいう．　したがって，最適化というより改良（improvement,amelioration）といった方が正確であるかもしれない．

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．

Prologのような論理型言語の最適化に対する研究は未だ少ないので，この節ではPrologプログラムをソースレベルで最適化するに当たって考察しなければならない点を論じ，我々の具体的方針について述べる．

1．1　Prologの非論理制に起因する最適化の困難性

Prologプログラムを最適化しようとするとき，Prologの次のような実行メカニズム，言語要素は論理的な最適化をかなり制限する原因となっている．

①　Prolog特有の実行メカニズム：　上→下，左→右，バックトラック；これらは論理式レベルでの自由な最適化を阻害する．

②　組み込みのevaluable 述語：　side effect を持つ述語（入出力述語），メタ述語，カット，否定，repeat（トートロジー），etc．；これらはオブジェクトとメタ言語をともに考慮に入れた最適化を必要とする．

実際，これらが各種最適化手法の中でどのような障害となって現れるかについては第3節で触れられる．

1．2　最適性の判断規準

最適化の有用性の尺度，すなわち何を規準にして最適化されたかということを明確にしておかねばならないことは言うまでもない．　次の三点から考察する．

(1)　外的規準（処理方式，環境）
　　処理計算機及びその上で言語の処理系がどのように作られているかに依存する

(2)　内的規準（表現）
　　プログラムの表現形式の選択に依存する：e.g.　冗長性のない表現（論理式の簡単化），データ構造，アルゴリズムの選択

(3)　計算方式
　　Prologは関係あるいは非決定性プログラミングを中心に据えた計算方式を表現する言語である．　従って，非決定性プログラミングに特有の現象を把握し最適化を考えねばならない：e.g.　不必要なバックトラックをさせない方法，非決定性の決定性への変換

さらに加えて，Prologは非決定性の枠組みの中で帰納的計算を可能ならしめている．だが，帰納的プログラムの最適化を進める上で参考になる結果はほとんど知られていないのが現状である．

最適化方法といわゆる複雑性の研究は目的とするところはよく似ている．　しかしながら，複雑性の研究では(1)，(3)は問題とならない．　プログラムの最適化問題ではオーダよりむしろ，係数の改良に主眼が置かれるので，(2)のみでなく(1)，(3)も最適化を考えるときの考察対象とされるべきであろう．

1．3　プログラム変換論とプログラムの最適化

狭い意味においては，プログラムの変換論とプログラムの最適化の目的は同じと考えられるが，広義には，プログラムの変換論とは一つのプログラミングparadigmであって，stepwise refinement （Dijkstra,Wirth）の概念にこれまでの伝統的な最適化技術を結びつけたものを意味することが多い．

そして，このparadigmの下では，我々は非効率的であるかもしれないが，明確で正しいプログラムを書き，それから，それを同値性を保存する変換を経て，明確さには欠けるかもしれないがより効率のよいプログラムへと変換する［Burstall 77］，［Scherlis 83］．

他方，実際的観点から眺めるならば，現在のところプログラムの変換論で唱えられているparadigmを実現しているシステムはまだ存在していない．　これまでのところでは，発見的な方法（eurekaの多用）に基づく次のような変換が代表的なものとして知られてい

るのみである：（i）local simplification，（ii）partial evaluation（unfolding）（iii）abstraction（folding），（iv）generalization（関数の拡張，付加的変数の導入）．また対象言語も理論的考察を容易ならしめる範囲に制限されている．　その代わり，極めてheristicでdrastic な変換により，得られたプログラムの進化の度合は大きく，効率も良い．

これとは対照的に，これまでのプログラムの最適化は極めて地味で頑張った割りにはあまり効果が出ないものが多いようである．　それはプログラム最適化がいつも制限されない生身のプログラムを相手にしなければならなかったからかもしれない．

以下で述べられるProlog言語の最適化はPrologのプログラミングの規範を示すことでもなく，またeurekaを多用するdrastic な変換でもない．　それはどちらかと言うと伝統的な従来言語の最適化の精神に通じるものと言える．　従って，プログラムの変換論とは明確に立場が異なっていることを強調しておかなければならない．

1．4　最適化の方針（インライン展開を中心として）

以上のことを踏まえて，この節ではPrologプログラムのインライン展開を中心とした最適化の方針について述べる．　この方策はPrologの最適化方法の一例に過ぎず，根本的に異なるアプローチからの最適化法が他にあるかもしれない．

まず，Prologの最適化問題に対する取組方は実用及び経験主義的である（少なくとも演繹的ではない）．　実用的という意味は最適化の対象となるPrologが一応fullなDec10 Prologであって，Prologの純論理的なサブセットのみを考えるようなことはしないということである．　また経験主義という意味は以下で述べられる最適化法に論理的必然性があるというわけではないということであり，Prologの最適化の現状から言って当然のあり方であると考える．

従来言語の最適化技法の中でインライン展開（サブルーチン展開）は最も重要な最適化法として知られている．　その大きな目的は次の二つにあるとされる［Scheifler 77］，［Chikayama 83］．

① 起動機構（サブルーチンの呼出機構）の解消
② 他の最適化技法の適用可能範囲の拡大

しかしながら，特に①はいつでもその目的が十分に達成されるとは限らない．　例えば，サブルーチンの呼び出しがループの内側にあるときインライン展開の効果が顕著に出るのはサブルーチン本体の実行時間が起動機構のそれに比べて非常に長い場合であるし，同じサブルーチンの呼び出し箇所が相当数存在する場合などはプログラムテキストはかなり膨らむことになり空間量の増大は見逃がせない問題となる．　また②に関しては，インライン展開後のプログラムに対して，いくつかの最適化手法がある場合その適用の順序が重要となる．　適用の順序いかんによっては潜在的に最適化が可能であっても十分に最適化されたプログラムが得られないことも起こり得る．

Prolog言語に対して上記のような目的のためにインライン展開をしようとするとき，従来言語の方法をそのまま流用することはできない．　展開する段階で最も異なってくるのは次の2点である．

（i）　Prologは述語（手続き）の列からのみ書かれる言語であって，すべての実行文が展開の対象となってしまう．

（ii）　Prologではそもそも呼び出し機構は述語定義の探索とユニフィケーションの二つからなり，その呼び出し機構を完全に解消することは一般に不可能である．　すなわち，述語の呼び出し機構を呼び側の述語とその述語定義の単一化可能性にすりかえる必要がある．　これらはProlog特有の言語形式，計算方式に依っている．　（i）はPrologの最適化を考えるさいインライン展開がその出発点となるべきことを暗示するし，（ii）は起動機構を新たな表現によりプログラムテキストの中に表現することを要求する．

このようなインライン展開によって膨らませられたプログラムに対していく種類かの最適化手法が適用されることになるわけであるが（展開して式を長く保っておくと他の最適化を受けさせるのにいつも都合がよいというわけではないが），ここにおいても，適用の順序が問題になり，個々の最適化手法が有効であってもその順序を誤るとせっかくのインライン展開の効果は半減してしまうことになりかねない．

オプティマイザに関する研究の現状からいって，1．2節で論じた最適性に関する判断規準すべてに適合する最適化システムを構成することは不可能に近い．　ここでは理想的な最適化システムの実現に向けて，first stepとしての方針を取る．

（a）展開レベル

インライン展開を中心としたPrologプログラムの最適化の方法を考えるとき，帰納的プログラムに対してはどのレベルまで展開すべきかという問題が直ちに起こる．　直ぐ思いつくのは展開レベルの導入であろう．　しかしながら，そのこと自体は簡単なことであるが，何等妥当な根拠が見当たらない．　これが妥当な方法として受け入れられるためには次のような問題が解決されたときにおいてであると考えられる．

① プログラムのdynamic な解析ができて平均的な展開レベルを見いだすことができた場合．
② 理想的なプログラムの最適化には時間と空間の双方からの考察が必要であるが，

プログラムの空間的な広がりをできるだけ押さえて，時間効率をいかに上げるかという条件下の最小化問題が形式化でき解けた場合（NP問題の一つであるknapsack問題と同値［Schoiffler 77］）．

現在我々のいく種類かの最適化手法を総合的に見ると，インライン展開は帰納的なプログラムの場合１レベルの展開で十分であると考えられる．　それ以上展開しても他の最適化手法によってdrastic にプログラムが最適化されるという期待は薄いようである．また展開のし過ぎはメモリーの過大な消費による負荷が高くなる．

（b）　時間 vs.　空間効率

プログラムの最適化は実行時間の効率を上げることに集中しがちであるが（実際の最適化システムではこれを目的とすることが多い），計算機システムでは時間も空間も貴重であることは言うまでもない．　特に，プログラムのインライン展開などは，その有効性は，展開によって起動機構（サブルーチンの呼出機構）にかかる時間をどのように減らせるかということと，展開によるプログラムテキストの膨らみがどの程度かということの両面から評価される必要がある．　しかしながら，現段階の研究の現状からいって，時間効率にのみ焦点を合わせた最適化システムの構成とし，空間効率の評価はシステムの使用経験を踏まえて考察していくことにする．

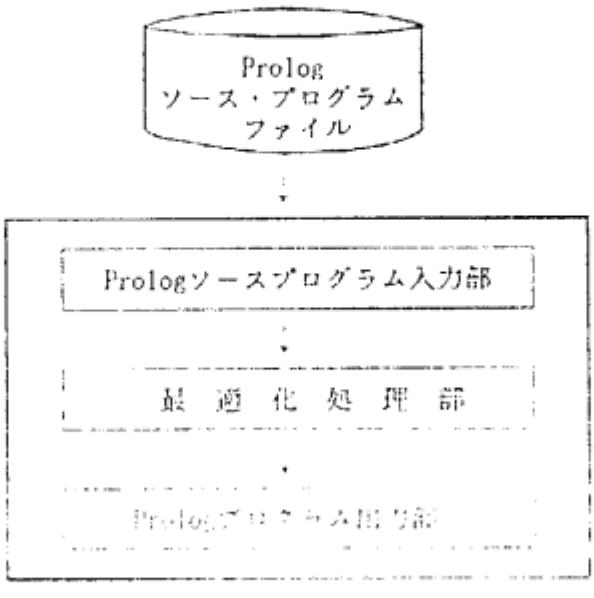
（c）interactive vs. automatic　（システム構成に当たって）

Prologオプティマイザへの入力としてfullなDec system10　Prolog［Bowen 81］系ソースプログラムを対象とするために，適時最適化のための指示を使用者が与えるという方式にする．　これは１．１節で触れたPrologプログラムの最適化の困難性をまだ十分に回避できていないことと（第３節で述べられる最適化手法適用の条件を参照），最も効果的な最適化手法の適用の順序が見い出されていないことによる．　しかしながら，このことは少しずつオプティマイザの知的能力を上げていくことにより完全自動化へと解消されていくと思われる．　現段階では，オプティマイザへの指示により，最適化の安全性を確保しつつ，諸々の最適化手法を組合せて最大の最適化効果を引き出すことを第一の目標とする．　なお，interactive といえども，システムとの複雑なやりとりを強いるというわけではなく，どちらかと言うと一方的に指示を与えるという程のものである．

最後に，我々のPrologプログラムのインライン展開の方法とBurstall［Burstall 77］らのプログラムの変換論における いくつかの概念との間にはいくつかの表面的な類似点が見いだされることについて触れておかねばならない．　例えば，プログラム変換論におけるunfolding はその行為だけではインライン展開と同じであるが，unfolding は後にabstraction によってfolding されることを意図して行われるのでインライン展開の主要目的①，②とは明らかに異なる．またfolding は我々の最適化手法では等式ゴール列の統合化（３．５章）といくらか似ているが，我々の方法では現在のところabstraction という発見的手法を用いることはしていない．　他方，インライン展開もunfolding も形式面からみると共に計算を部分的に進行させているわけであるから，partial evaluation の異なる形態であるとも見なせる．

2.　Prologソースレベル・オプティマイザの概要

以下にPrologソースレベル・オプティマイザの概念図（図１）と基本的構成図（図２）を示す．



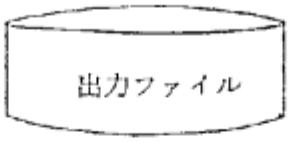


図1.　Prologソースレベル・　オプティマイザ概念図

オプティマイザの主要機能は大きく次の3つに分かれる.
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 各種最適化手法に対応する機能
③ 入出力機能

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，一般の帰納的プログラムの各インライン展開（3．2節参照）の前後に各種の個別の最適化手法（3．3節参照）を適用する形で進行する（図2）．　枝分かれは，そこでさらに最適化を進めるか，終了させるかの判断が行われることを示している.

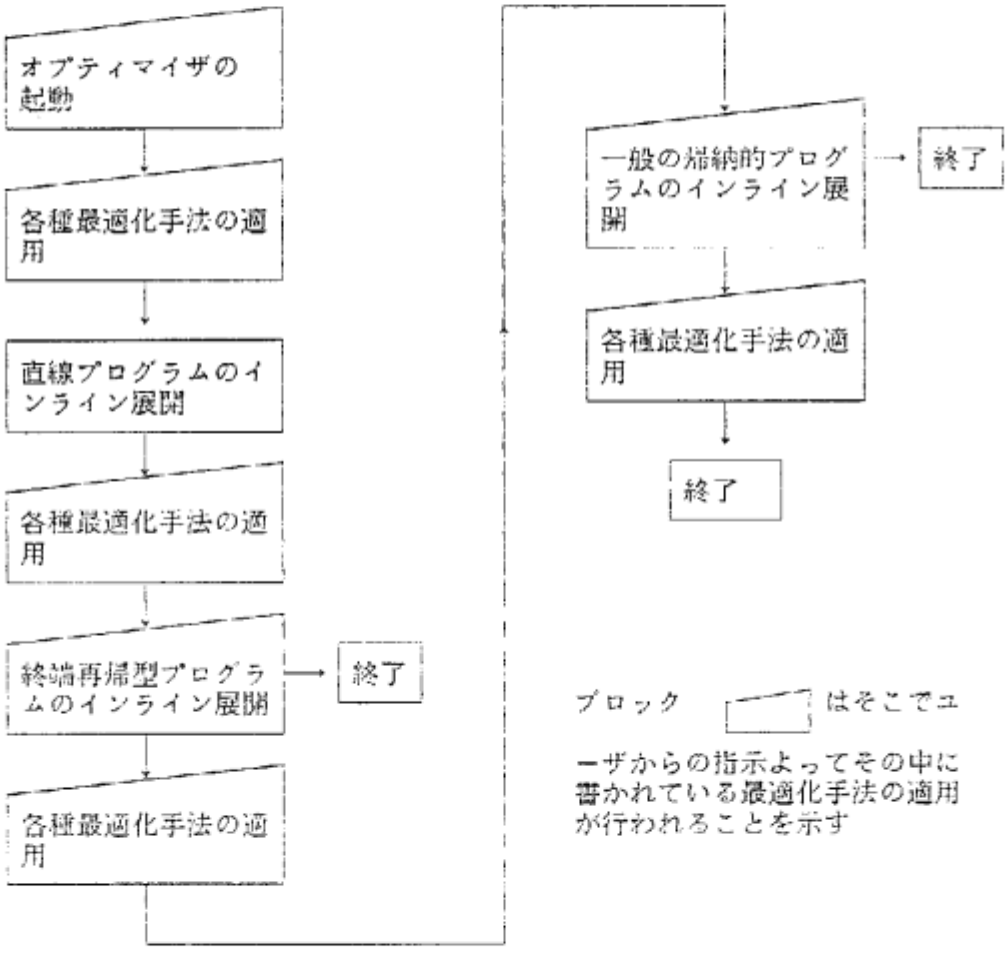


図2.　Prologソースレベル・オプティマイザの基本構成

3. Prologのための最適化手法

［記号の規約］
(1) Γ，Δ，Λなどのギリシャ文字はゴール（述語）の列を表す（空列も含む）．
(2) ΓがプログラムⅠ（節の集合）あるいはゴールの列を表すとき，それを最適化してΔにするということを次のような図形で表現する．

$$\frac{\Gamma}{\Delta}$$

このような図形を最適化図式と呼び，Γを上式，Δを下式と呼ぶ．　Γ及びΔには'｜'（縦棒）が現れることがあるが，これは'，'あるいは';'を表す．
(3) tがPrologの項であるとき，t'はtに現れるすべての変数を全く新しい変数で置き換えてできる項を表す．
(4) Γ（X）が変数Xをいずれかに含む述語の列をあらわすとき，Γ（t）はそのような述語の変数Xに項tを代入してできる述語の列を表す．
(5) その他の記号の使い方はDec10 Prologにおおよそ従い，例外はその都度説明されるか，もしくは文脈から明らかとなるであろう．

3．1　ユニフィケーションの部分実行

通常の言語においてはサブルーチン展開によってその呼びの機構（起動機構）は無くすることができるがPrologでは一般にインライン展開においてそれに必要な起動機構を解消することは難しい．　すなわち，起動機構はゴールとヘッドの単一化可能性を表す述語として残されてしまう．　この単一化可能性をユニフィケーションの部分的実行によって表現しておけば起動機構のオーバヘッドを少しでも軽減するのに役立つ．　さらに，後々の最適化にも役立つ．

［最適化図式　1］　ユニフィケーションの部分実行を等式の形式をしたゴール列として表現する．すなわち，

$$\frac{p(\ldots) = p(\ldots)}{X1 = t1,\ \ldots,\ Xn = tn}$$

3．2　インライン展開

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．

Prologのインライン展開を次のような自然な方策によって行う．　すなわち，「述語の呼びを，呼出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」．

しかしながら，これが自由に行えるのは呼ばれる側の述語の定義本体の中にカット記号が現れないときである．　というのは，バックトラックが起こったとき，展開前と展開後のプログラムの振る舞いが，カット記号があると異なってしまうからである．　例えば，次の展開前と展開後のプログラムにおいて，展開前のプログラムでもしcが失敗したときaの呼びを失敗させ，制御はqに移るが，他方展開後のプログラムではpの呼びを失敗させることになる．

```
展開前：  p :- q,a,r.
          a :- b,!,c.
展開後：  p :- q, (a = a, b,!,c),r.
          a :- b,!,c.
```

このように，カット記号の存在はインライン展開の可能性に重大な影響を与える．以下では，カット記号を含むプログラムによるインライン展開が上記のように行えるための条件を含めて，もう少し形式的にインライン展開の方法について述べる．

3．2．1　カットを含まない定義体によるインライン展開

［最適化図式　2］　ゴールCをカットを全く含まないプログラムHiによって次のようにインライン展開する．

$$\frac{\begin{array}{l}G\\ H1 :- \Gamma 1.\\ H2 :- \Gamma 2.\\ \vdots\\ Hn :- \Gamma n.\end{array}}{G = H1', \Gamma 1'\ ;\ \ldots\ ;\ G = Hn', \Gamma n'}$$

ここで，HiはG の定義節とし，Γi が存在しないときは単にG＝Hi' とする．

3．2．2　カットを含む定義体によるインライン展開

(1)　カット図式　その1

ここでは展開に用いられる各定義節にカットがちょうど一つある特殊な場合について考える．

展開しようとするゴールに対する定義節の集合を次の形とする．

```
H1 :- Γ1, !, Δ1.
H2 :- Γ2, !, Δ2.
 ⋮
Hn :- Γn, !, Δn. | Hn :- Γ. | Hn.
```

ただし，Γi,Δi,Γは '!' を含まないゴールの列を表し，Γi,Δi が空の場合にΓi,!, Δi が表すものは次のとおりである．

Γi,Δi が共に空列のとき：!

Γi のみが空列のとき：!, Δi

Δi のみが空列のとき：Γi,!

このとき，ゴールGのインライン展開を次の図式に従って行う．

［最適化図式　3］

```
G
H1 :- Γ1, !, Δ1.
 ⋮
Hn :- Γn, !, Δn.  | Hn :- Γ. | Hn.
―――――――――――――――――――――――――――――――――――――――
G = H1', Γ1'→Δ1' ; ... ; G = Hn', Γn'→Δn' | G =Hn', Γ' | G =Hn
```

ただし，Γi,Δi が空のとき下式は次の規約に従って構成されるものとする．

```
Hi :- !, Δi.     ⇨   G = Hi'→Δi'
Hi :- Γi, !.     ⇨   G = Hi', Γi'→true
Hi :- !.         ⇨   G = Hi'→true
```

(2)　カット図式　その2

インライン展開が文脈に依存して可能となる場合について述べる．

［最適化図式　4］

```
a :- Γ, p(A), Δ.
p (A1) :- Θ1.
 ⋮
p (An) :- Θn.
―――――――――――――――――――――――――――――――――――――――
a :- Γ,
     ( p(A) = p(A1'), Θ1' ; ... ; p(A) = p(An'), Θn')
   , Δ.
```

ただし，次の条件を満たす．

① Γ，Θi （1 ≦i ≦n ）はゴール列（空列であってもよい）を，A,Ai （1 ≦i ≦n ）は項の列を表す

② Γは空かあるいは，Γの中の最も右にあるカット（もし存在するならば）の右側にあるすべての述語は決定的であること（言い換えるとバックトラックしたとき再び成功することはないこと）（定義については［沢村８４］を見よ）

③ a の代替は存在しないか，あるいはａはａの残りのすべての定義節のヘッドと単一化不可能であること

これらの条件を基礎にして，以下の各節において，プログラムの形式に応じた展開の方法について述べる．

3．2．3　直線プログラムのインライン展開

ここでは直線プログラムのインライン展開の方法について述べる．　直線プログラムとは帰納的でないプログラムをいう（プログラム形の定義については［沢村８４］を見よ）．　直線プログラムのインライン展開は通常言語のサブルーチン展開と同じように，帰納的でないプログラムにおいて手続きの呼び（述語の呼出）をその定義本体で置き換えることによって行われる．

［インライン展開　1］直線プログラムのインライン展開を次のように行う．

述語名ｐに関する各定義節ｃの各ゴールａ毎に，次の条件が満たされる限り，Prologの実行順序と同様，最左，深さ優先でインライン展開を行う．

① cは直線プログラムである.
② aは直線プログラムである.
③ aの定義節の中にカット記号は現れない，ただしaの定義節がカット図式2の形式をしているときはこの限りではない． また，aの定義節がカット図式1の形式となっているときは上でのべた図式に従って展開を行う（コンパイラに通す場合はこれを行なわない）.
④ aの定義節の中に述語名pは現れない.
⑤ aの展開はサイクルを生成しないこと.

［補注］

(1) 条件②，④，⑤は直線プログラムのインライン展開を終了させるのに必要な停止条件である.

(2) 次の例を検討する.

```
p :- a1, ... , ai, ... , an.
ai :- b1, ... , bj, ... , bm.        ai≠bk (1 ≤k ≤m )
bj :- c1, ... , p, ... , cl.         ai≠c1, ... , p, ... , cl
```

pの展開にさいして，pの本体のaiの展開はaiの定義節によって行うが,ai の本体のbjを展開することはしない（bjの本体の中に元のp が含まれており，これに対して展開を行うと停止しなくなるので）． しかしながら，aiの展開にさいしては,ai の本体のbjはbjの定義節を用いて展開が行われる． さらに，bjの展開にさいしては,pの展開はp の新しい定義節によって行われる.

(3) 異なった述語名の定義節がいくつかあるとき，その順序を変えると，直線プログラムのインライン展開の結果は異なる． しかしながら，直線プログラムをインライン展開するという本来の目的は達成できている.

(4) 直線プログラムのインライン展開の結果は再び直線プログラムになる． さらに，相互recursive となっているプログラムは直線プログラムと見なされる.

3．2．4 終端再帰型プログラムのインライン展開

ここでは終端再帰型プログラムのインライン展開の方法について述べる 本節と次節で述べられるインライン展開は帰納的なプログラムに対して行われる.
従来の実用的最適化システムでは，明らかな理由によって，帰納的なプログラムをインライン展開するというような野心的な試みは行われていない.

帰納的なプログラムの最適化としてはインライン展開を行うというよりはむしろ，特定の形式をした帰納的プログラムを繰り返し型のプログラムに変換して最適化をするという方法が取られることが多い． 特に，Prologのコンパイラのように終端再帰型のプログラムを繰り返し型にコンパイルするという方法が知られている.

本節では，終端再帰型のPrologプログラムを繰り返し形に変換するということはせず，繰り返し達成されるゴールの部分をインライン展開するという方法について述べる.
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展開と同じ発想であり，インライン展開の効果が最も期待できる方法である.

さらに，Prologプログラムはそのほとんどが帰納的プログラムであり，プログラムの走行時間の多くを，ループと本質的に同じである終端再帰の実行時間が占めることを考えるとこのような展開の効果は大きい.

3．2．3節の方法で直線プログラムをインライン展開した後，終端再帰型のプログラムを次のようにインライン展開する.

［インライン展開 2］ 述語名pに関する定義節の一つを

```
p (... ) :- Γ,p (... ) .
```

とする，但しΓにはpの呼びはないものとする． このときΓの各ゴールgに対して次の条件を満たすように展開を行う.

① gの定義節のすべての節が直線プログラムならば3．2．3節の③のようにgの展開を行う.
② 述語名pはgの定義節の中に現れない.
③ 述語名pに関する定義節の各々を①，②を満たすように展開した後，定義節の終端にあるp（... ）を一回だけ展開する.

［補注］

(1) 条件①は帰納的プログラムのインライン展開の終了条件が複雑になり過ぎるためと，展開レベルを導入するための客観的尺度が得られていないために設定されている.

(2) 条件②は相互recursion になっている場合を避けるためのものである． もしこれがないと，pの展開節として，pの展開前の定義節を選ばねばならない（pの新しい定義節は未完である）し，また終端再帰型のプログラム形を壊すことにもなる． さらに，(1)と同じ問題に陥いることにもなる.

(3) ①はΓの各述語に対して一度行い，得られた述語に対して再度展開することはしない（直線プログラムは3．2．3節において十分に展開されている）.

(4) 終端再帰型プログラムのインライン展開後のプログラムは拡張された意味において再び終端再帰型となる.

(5) ①, ②, ③は終端再帰型プログラムのインライン展開の停止条件でもある.
(6) 条件③は終端再帰型のプログラム形を保存しつつ, 他のより多くの最適化技法の適用を受けさせるために設定されている.

3．2．5 一般の帰納的プログラムのインライン展開

これまで直線プログラムと終端再帰型のプログラムに対してインライン展開を行う方法について述べてきた. 残っている場合は一般の帰納的なプログラムのインライン展開である.

しかしながら, 一般の帰納的なプログラムに対してはどのレベルまで展開すべきかということについて客観的な展開規準を見い出すにいたっていない（そのような規準はあり得ないのかもしれないが）. そこで本節では一回でもよいから展開して他のより多くの最適化技法に供することを目的とするに留める.

3．2．3節及び3．2．4節の方法で直線プログラム及び終端再帰型のプログラムをインライン展開した後, 一般の帰納的プログラムを次のようにインライン展開する.

［インライン展開 3］述語名pに関する一般の帰納的プログラムの定義節が与えられているとき, その中の次の形式の節に対して以下のように展開を行う.

```
p (... ) :- Γ.
```

ただし述語名pはΓの中に終端以外の場所に起こるものとする.

① Γの中の述語 p（... ）以外のゴールを, その定義体が直線Prologである限り唯一回3．2．3節の条件③のように展開する（前節補注(3)参照）.

3．3 ゴール列の簡単化

この節では, 命題論理的にゴールを簡略化する技法のいくつかについて述べる. これらはほとんどが局所的な簡単化（simplification）や除去戦略（deletion strategy）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.

3．3．1 重複除去

(1) 述言列中の重複除去

ゴール列の中にお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. すなわち, 次の図式に従って簡略化をする.

［最適化図式 5］

$$\frac{\Gamma, q, \Theta, q, \Lambda}{\Gamma, q, \Theta, \Lambda}$$

ただし, 次の条件に従うものとする.

① Θには '!'（カット記号）が含まれていないか, もしくは qは決定的である.

② q はカット記号, 入出力述語, あるいはメタ述語であってはならない.

［制約条件の説明例及び補足例］

(a) 左側の述語を残すことについて：

```
p (... ) ,q (... ) ,p (... )
-----------------------------
p (... ) ,q (... )
```

において, q（・・・）が停止しない場合, 左側のp（・・・）を取り除くと正しくない. 以下の最適化はすべて妥当でない.

(b) メタ述語について：

```
... , var (X ) ,p (X ) ,var (X ) , ...
---------------------------------------
... , var (X ) ,p (X ) , ...
```

(c) 入出力述語について：

```
...  , write (X ) ,p (X ) ,write (X ) , ...
---------------------------------------------
...  , write (X ) ,p (X ) ,  ...
```

次の二つの例は上例以外にも重複除去はできないことがあることを示している.

(d) repeat述語：

次の述語が与えられているものとする.

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

次の最適化は正しくない.

```
repeat, q (X ) ,repeat, not (X = a)
-----------------------------------
repeat, q (X ) ,not (X = a)
```

なぜなら, 上式はX ＝aのままで"repeat,not（X ＝a）"を無限にくり返すが, 下式はX ＝bで成功し, redoはX ＝cで成功し, さらなるredoはX ＝bあるいはX ＝cを交互に

くり返して成功するからである．　したがって，このような場合，重複述語repeatは除去できない．

(e) not について：

p (f (Y,Z)) . ,　(f (b,c)) . ,　p (f (b,c2)) . ,

q (f (b,U)) . ,　q (f (b2,V)) .

が与えられているとするとき，次の最適化は正しくない．

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

なぜなら，上式ではX = f (b, c 2) となるが，下式ではX = f (b 2, V) となるからである．

(2) 選言列中の重複除去

［最適化図式　6］

$$\frac{\Gamma\ ;\ p\ ;\ \Theta\ ;\ p\ ;\ \Delta}{\Gamma\ ;\ p\ ;\ \Theta\ ;\ \Delta}$$

ただし，pはカット記号，side effect をもつ述語，メタ述語ではない．

［注１］　連言中の重複除去のケースと異なり，Θにはcut が含まれていてもよい．

［注２］　しかしながら，次のようにバックトラックが起こるときは正しくないことがある．

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

3．3．2　冗長なtrue,fail の除去

［最適化図式　7，8］

$$\frac{\Gamma,true,\Theta}{\Gamma,\ \Theta}\ ,\qquad \frac{\Gamma\ ;\ fail\ ;\ \Theta}{\Gamma\ ;\ \Theta}$$

3．3．3　不実行部分の除去

［最適化図式　9］

$$\frac{\Gamma,fail,\Theta}{\Gamma,fail}$$

［注意］　$\dfrac{A\ ;\ true\ ;\ B}{A}$　は妥当でない

3．3．4　共通ゴールのくくり出し

［最適化図式　１０，１１］

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

［注意］　①　Γにside effect があると同値でなくなる．

②　バックトラックが起こらないことが必要である．

3．4　変数除去

この節では，無駄な変数を除去することによって，達成すべきゴールの数を減らすことになる最適化の方法について述べる．　この最適化もまたインライン展開されたプログラムに適用されることになる．

3．4．1　冗長変数除去

次の図式に従って，孤立変数を除去する．

［最適化図式　１２］

$$\frac{p\ (...)\ :-\ \Gamma,X\ =\ f\ (t1,t2,\ ...\ ,tn)\ ,\ X\ =\ f\ (r1,r2,\ ...\ \ ,rn)\ ,\ \Delta}{p\ (...)\ :-\ \Gamma,f\ (t1,t2,\ ...\ ,tn)\ =\ f\ (r1,r2,\ ...\ rn)\ ,\ \Delta}$$

ただし，変数X は p (...) , Γ, Δには現れない.
［制約条件の説明例］　次の冗長変数除去は妥当ではない.

```
... , X is Y+Z, ... , X = f (Y) ,X = f (Z) , ...
-------------------------------------------------
... , X is Y+Z, ... , f (Y) = f (Z) , ...
```

3．4．2　等式代入
［最適化図式　13，14］

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

ただし，Γにはカット及びside effect をもつ述語は含まれない.
［制約条件の説明例］
①　q (X) :- !,X=t, p (X) . は q (t) :- !,p (t) . に等しくならない。　というのは左式はゴールX =t で失敗するとq のコールから飛び出るが，右式では q (t) にunify されないとq の他の代替を探すことになる。　したがって，代替がなければ同値になる。

②　　p (X) :- write (X) ,X=t.
　　---------------------------
　　p (t) :- write (t) .

?- p (a) . に対して，上式，下式共にfailするが，上式はa がside effect としてプリントされてしまう。　等式の左側にside effect 述語がなければ同値になる

3．5　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ている。　そして，この等式の列の一部はその後の局所的な最適化に利用されることになる。
本節では，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法について記述する。
等式ゴールの統合化は達成すべきゴールの数を減らすのに有効であるばかりでなく，ゴール自体を簡単化するという働きももっている。
［最適化図式　15］

```
X1 = t1, ... , Xn = tn
-----------------------------------
dummy-f (X1, ... , Xn) = dummy-f (t1, ... , tn)
```

ここに，dummy-f は適当な関数記号を表す。

3．6　節の複数節への分解
3．2節で述べた方法に従ってインライン展開されたプログラムは一般に次の形式をしている。

　　p :- Γ, (q1 ; ... ; qn) , 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可能となる場合がある（特に，3．4．2節の等式代入）.
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ていることに相当している.
［最適化図式　16］

```
p :- Γ, (q1 ; ... ; qn) , Δ.
-----------------------------
p :- Γ,q1,Δ.
:
p :- Γ,qn,Δ.
```

［補足］
(1) いずれかの場所に '！' があっても条件なしに上式と下式は同値になる。
(2) Γは空か，もしくはΓが引き続く最適化によって消去できるならばこの図式は有効である.

4. 最適化の例

reverse 述語のインライン展開の例：

```
reverse ( [] , [] ) .
reverse ( [X] , [X] ) .
reverse ( [X | Y] ,Z) :- reverse (Y,Z1) ,append (Z1, [X] ,Z) .
```

スタック変数を用いたreverse の変換版：

```
rev (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．

```
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) .
  ( [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] ) .
```

(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)
```

③ ここでさらに，(1)の r (X,Y, []) を新しいr で展開する．

```
rev (X,Y ) :-
  r (X,Y, [] ) = r ( [] ,Z,Z) ;
  r (X,Y, [] ) = r ( [H] , [H | Z] ,Z) ;
  r (X,Y, [] ) = r ( [H | [H1 | T1] ] ,W1,Z ) ,r (T1,W1, [H1 | [H | Z] ] )
```

(ユニフィケーションの部分実行)

```
rev (X ,Y) :-
  X= [] ,Y=Z,Z = [] ;
  X= [H] ,Y= [H | Z] ,Z= [] ;
  X= [H | [H1 | T1] ] ,Y=W1,Z= [] ,r (T1,W1, [H1 | [H | Z] ] ) .
```

(節の複数節への分解)

```
rev (X,Y ) :- X = [] ,Y=Z,Z = [] .
rev (X,Y ) :- X = [H] ,Y= [H | Z] ,Z= [] .
rev (X,Y ) :- X = [H | [H1 | T1] ] ,Y=W1,Z= [] ,
              r (T1,W1, [H1 | [H | Z] ] ) .
```

(代入)

```
rev ( [] , [] ) .                                               (7)
rev ( [H] , [H] ) .                                             (8)
rev ( [H | [H1 | T1] ] ,W1 ) :-r (T1,W1, [H1 | [H] ] ) .       (9)
```

(1), (4), (5), (6)あるいは(4), (5), (6), (7), (8), (9)はreverse の定義(1), (2), (3)の最適化版である． (1)—(3)から，(4)—(9)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．

5. まとめ

本稿では主に経験的及び直観的に，Prolog言語のソースレベルでの最適化手法について検討してきた．　そして多くの有効な最適化手法が見い出された．　それらは従来言語の最適化手法に比べて論理的には複雑である．　これはProlog言語のもつ非手続き性と手続き性の二面性に由来するものと思われる．　実際，Prologの純論理的な部分は単なる論理式の変形という意味での最適化を可能にするし，他方，Prolog特有の実行メカニズム（上一下，左一右，バックトラック）及び数々の超あるいは非論理的言語要素は論理式レベルての最適化を阻害する要因にもなっている．

我々はPrologプログラムの変換論と異なる立場から，どららかと言うとあまり論理的と思われないPrologのプログラムをも対象として，実際の最適化システムを作ることを主目的にしてきた．　従って，各種最適化手法がいつでも真に同値性を保証しているか否かについての形式的な証明に関する議論についてはまだ触れられていない．　この意味では通常言語の最適化システムでも同じである．　唯異なるのはProlog言語の使用に関する歴史があまりに浅く，言語の挙動に関する検証の社会的過程が十分でないということである

最後に，我々の最適化システムの機能を次のように特徴づけ，列挙しておく．

(1) interactive
(2) inline expansion based
(3) 同値性はoperationally に確認されている
(4) 帰納的なプログラムの一つのreasonableなインライン展開法を試みた
(5) 我々の方法は最適化もさることながらpartial evaluationの一方法でもあるとも見なせる
(6) 時間効率のみを目標にしている
(7) 制御構造の最適化に重点が置かれている

今後の検討課題：

① 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こと

② 最適化技法が与えられているとき，最適化技法の適用順序の分析を行いCR性を調べる．　このとき，最適化図式を公理あるいは推論規則（あるいはreduction rule）と見なす．

③ 最適化システムのmain routineを換えて，最適化のあらゆるパスがinteractiveに得られるようにする．　これによって，どのパスが最もtime-space効率に寄与したかという統計情報が取れる．　また，userがどのような最適化技法の系列を選んだかという情報を記録しておくことも重要である．

④ データ構造の最適化
⑤ 知的プログラミングへの接続
⑥ プログラム変換論としての展開

謝辞　本研究は第５世代コンピュータ・プロジェクトの一環として行われたものである．近山隆氏（ICOT）の有用なコメントに対して感謝する．

参考文献

[Scherlis 83] W. L. Scherlis and D. S. Scott, First step towards inferential programming, Information Processing 83, R. E. A. Mason (ed.), North-Holland, 199-212 (1983)

[Burstall 77] R. M. Burstall and J. Darlington, A Transformational system for developing recursive programs, JACM, Vol.24, No.1, 44-67 (1977).

[Scheifler 77] R. W. Scheifler, An analysis of inline substitution for a structured programming language, CACM, Vol.20, No.9, 647-654 (1977), (近山訳： 構造的プログラム言語におけるサブルーチン展開の解析，bit, Vol.10, No.8, 77-86 (1978)).

[Chikayama 83] T. Chikayama, Source level optimization in logic programming languages, draft, (1983).

[BOWEN 81] D. L. Bowen, Dec system-10 Prolog user's manual, version 3.43, Dept. of Artificial Intelligence, Univ. of Edingburgh, (1981).

[沢村　８４] 沢村・竹島，Prologソースレベル・オプティマイザ，メモノート，1984.