Prolog ソースレベル・オプティマイザの試作とその性能評価†

沢 村 一††

現実の Prolog プログラムを効率化のためにソースレベルで変換を行う最適化変換手法が，著者らの他の論文において提案されている．それらにはインライン展開法，カットの自動挿入法，数々の局所的最適化手法等が含まれている．本論文では，このような最適化手法の概要を述べ，それらを統合化して試作されたオプティマイザの性能評価結果を示しその有効性を検証する．本論文の方法はプログラムの最適化変換に向けての素朴であるが実際的で有望なアプローチの一つとして特徴づけられるであろう．

1. はじめに

Prolog[1] はこれまでのプログラム言語に比べて非手続き性あるいは記述性の高いプログラム言語であると言われ，新世代の言語として注目されている言語の一つである．プログラム言語が広く一般に受け入れられるためには，言語の設計思想を反映した高い記述能力にそれを効率よく実現するための実現技法が伴っていなければならない．Prolog に対してもこの両面からの研究努力は盛んに続けられており（例えば，文献2),3)），論理型言語としての Prolog はまだ進化の途中の段階にある言語であると言える．本論文は，Prolog プログラムの最適化に向けての変換手法群およびそれらの手法を統合化したオプティマイザを提案し，その評価を行うものである．Prolog オプティマイザは現実の Prolog プログラム[1] を対象とし，ソースレベルにおいて主として時間効率改善のための変換を行う．

Prolog はユニフィケーションとバックトラック機構に基づく非決定性プログラム言語であるため，制御とデータの流れが従来の言語に比べて複雑である．そして，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，最適化の考察の対象となるプログラムの範囲によって次のように分けることができる：

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

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ことは言うまでもない．中でも述語呼び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が，そのことに起因する空間的・時間的負荷はかなり大きい．述語呼び出し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極めて重要となる[4]．次の節では上の三つの範疇に属する最適化手法の概要を述べる．第三節では，それらの手法を一つに統合して試作された Prolog オプティマイザの構成に簡単に触れ，我々の方法論に基づくオプティマイザの性能評価の結果をいくつかの小規模なプログラムをとおして示す．

2. 最適化手法の概要

以下では Prolog オプティマイザの機能の概要を述べるに止める．各種最適化手法の適用条件の詳細，および理論的検討については文献 4)～7) を参照されたい．

2.1 プログラム情報の抽出

一般に，プログラムの最適化に当たってはプログラムから最適化に必要とされるプログラム情報をあらかじめ抽出しておくと都合がよい．次のプログラムの型と述語の決定性情報は我々の Prolog プログラムの最適化方法論において重要な働きをするものである．

† A Source-to-Source Prolog Optimizer and Its Performance Evaluation by HAJIME SAWAMURA (International Institute for Advanced Study of Social Information Science (IIAS-SIS), FUJITSU LIMITED).

†† 富士通(株)国際情報社会科学研究所

（1） 述語定義（手続き）の型

述語定義（手続き）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，この分類に従って段階的にインライン展開が行われる．

（2） 決 定 性

プログラムを構成している各述語呼び出し（ゴール）が決定的かどうかを検出する．我々の方法に必要な決定性の定義は次のようなものである：「述語呼び出しが決定的であるとは，それが呼ばれたとき，その定義体のたかだか一つの節で成功し，バックトラックされたとき，その述語呼び出しは決して成功することはないことをいう」．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．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[4]，次のような内容をもち互いに関係する二つの決定可能な決定性を定義する．

① a-決定性：これは絶対的な決定性を意味し，与えられた述語呼び出しがその引数の内容にかかわらず決定的であるような決定性である．

② r-決定性：これは相対的な決定性を意味し，与えられた述語呼び出しがその引数の内容ゆえに決定的である決定性である．

ここで，述語呼び出し $p(A)$ が a-決定的，あるいは r-決定的であるということを，次のような相互帰納法によって形式的に定義する，ただしギリシャ文字はゴールのゼロ個以上の列を表し，述語定義体にはプログラムを動的に変更する述語，例えば，“assert”，“retract” 等は現れないことを仮定する：

（i） p が組み込みの述語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

$H1$:—$\Gamma 1$.
⋮
Hi:—Γi, [!,]Δi.
⋮ （ここで，カット記号は存在するならボディ部の中の最右出現とする）
Hn:—Γn.

このとき，各 $Hi(1 \leqq i \leqq n)$ 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，$p(A)$ は Hj $(i+1 \leqq j \leqq n)$ 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．

述語（呼び出し）の決定性の定義とその利用に関するいくつかの議論については文献 7) を見られたい．以下に，簡単な例を示しておく．

例 1. 述語呼び出し p(X) は a-決定的である．

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

例 2. 述語呼び出し q([a, b]) は r-決定的であるが，q(X) はそうではない，ただし，述語 p の定義体は例 1 のものとする．

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

2.2 インライン展開

インライン展開の主要目的は次の二点にある．

① 述語の呼び出し機構の除去（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，呼び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．すなわち，

G
$H1$:—$\Gamma 1$.
$H2$:—$\Gamma 2$.
⋮
Hn:—Γn.

$G = H1', \Gamma 1'$; … ; $G = Hn', \Gamma n'$

ここで，“′” は変数の名前を全く新しい変数名に換えるリネームオペレータ，“=” は Prolog に標準的に備わっている単一化可能性を表す述語，G は展開されるゴール，Hi:—$\Gamma i (1 \leqq i \leqq n)$ は G の定義節を，そして Γi が存在しないときは $G = Hi'$, $\Gamma i'$ は単に $G = Hi'$ を表すものとする．

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．カットが出現する場合は，語述呼び出し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．それは次のような展開図式によって可能となる．

$H1$:—$\Gamma 1$.
⋮
Hi:—Γi, $p(A)$, Δi.
⋮
Hm:—Γm.
$p(A1)$:—$\Theta 1$.
⋮ ($\Theta j(1 \leqq j \leqq n)$ のいずれかにカットが含まれているものとする)
$p(An)$:—Θ.

$H1$:—$\Gamma 1$.
⋮
Hi:—Γi,
($p(A)=p(A1')$, $\Theta 1'$; …;
$p(A)=p(An')$, $\Theta n'$),
Δi.
⋮
Hm:—Γm.
$p(A1)$:—$\Theta 1$.
⋮
$p(An)$:—Θn.

ここで，Ai', $\Theta i'$ はそれぞれ Ai, Θi をリネームしたものを表す．そして，次の条件を満たさなければならない．

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：
$\Gamma' i$ の中の最も右にあるカットの右にあって $\Gamma' i$ に含まれるすべての述語は a-決定的か，r-決定的である．

例 3. 述語 p と q はそれぞれ例 1 と例 2 において定義されているものとする．

```
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), !.
```

展開後の下式の第二節は後で述べられる局所的最適化手法を適用するとさらに簡単化されて次のようになる：r(b, L, [b|L]):—p(b), !.

Prolog のプログラム（手続きの集まり）が与えられたとき，以上のインライン展開の図式は展開の処理をわかりやすくするために手続きの型にしたがって段階的に手続きのゴールに適用される（大まかな展開の流れは 3.1 節で触れる，また詳しいアルゴリズムは文献 5), 7) に与えられている）．

2.3 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場合，適切な位置にカット記号を挿入して不必要な redo を避ける方法の一つを述べる．この方法においても述語呼び出しの決定性の検出は重要な役割を果たすことになる．このようなカットの自動挿入を次の図式によって実現する．

$H1$:—$\Gamma 1$.
⋮
Hi:—Γi, Δi.
⋮
Hn:—Γn.

$H1$:—$\Gamma 1$.
⋮
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．

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 の最右の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．

例 4. 述語 p と q はそれぞれ例 1 と例 2 において定義されているものとする．

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

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

2.4 局所的最適化手法

この節では，命題論理レベルで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簡単

に述べる（詳しい最適化図式の適用条件については文献5),7) に述べられている）．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．

（1） ユニフィケーションの部分実行

通常のプログラム言語では，サブルーチン展開によってその呼び出し機構に伴うオーバヘッドを取り除くことができるが，Prolog では一般にインライン展開において呼び出し機構を解消することは難しい．我々の方法では述語の呼び出し機構はゴールとその定義体ヘッドの単一化可能性を表す一つの等式ゴールとして残ることになる．この単一化可能性をユニフィケーションの部分実行によって実行しておけば呼び出し機構のオーバヘッドを軽減することができる．さらに後々の最適化にも利用できる．ユニフィケーションの部分実行とは例えば，等式ゴール

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

を，左辺が変数で右辺が項であるような同値な等式ゴールの列

X＝h(d), Y＝d, Z＝d

に変形することを言う．

（2） 重複除去

① 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最も左の述語を残し他を除去する．

② 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．

（3） 冗長な true, fail 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．

（4） 不実行部分の除去

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．

（5） 共通のゴールのくくり出し

次のようなゴールの列

p, q ; p, q, r

を共通のゴールのくくり出しによって，

p, q, (true ; r)

のように変形する．

（6） 等式代入による変数除去

述語論理の等式代入規則から暗示される最適化手法である．例えば，節

p(X):−r(X, Y), X＝t, q(Z).

は等式代入によって，

p(t):−r(t, Y), q(Z).

に等価な節である．

これ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もつながる最適化の方法でもある．

（7） 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，その際その呼び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の局所的な最適化にも利用されることになる（例えば，上の(6)によって）．

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．例えば，等式ゴール列

X＝h(d), Y＝d, Z＝d

は，適当な関数記号 'f' を用いて次のように統合化される．

f(X, Y, Z)＝f(h(d), d, d)

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，ユニフィケーションの部分実行とともにゴール自体を簡単化するという働きをもっている．

（8） 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, ($q1$; … ; qn), Δ.

インライン展開とは逆に，このような形式の節を，

p:−Γ, $q1$, Δ.
⋮
p:−Γ, qn, Δ.

のように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，等式代入による変数除去）．

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．

3. Prolog オプティマイザの試作と評価

3.1 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，完全自動化のオプティマイザが実現されるであろう (unfold/fold 法に基づくプログラム変換の自動化の研究については文献8)で行われている)．

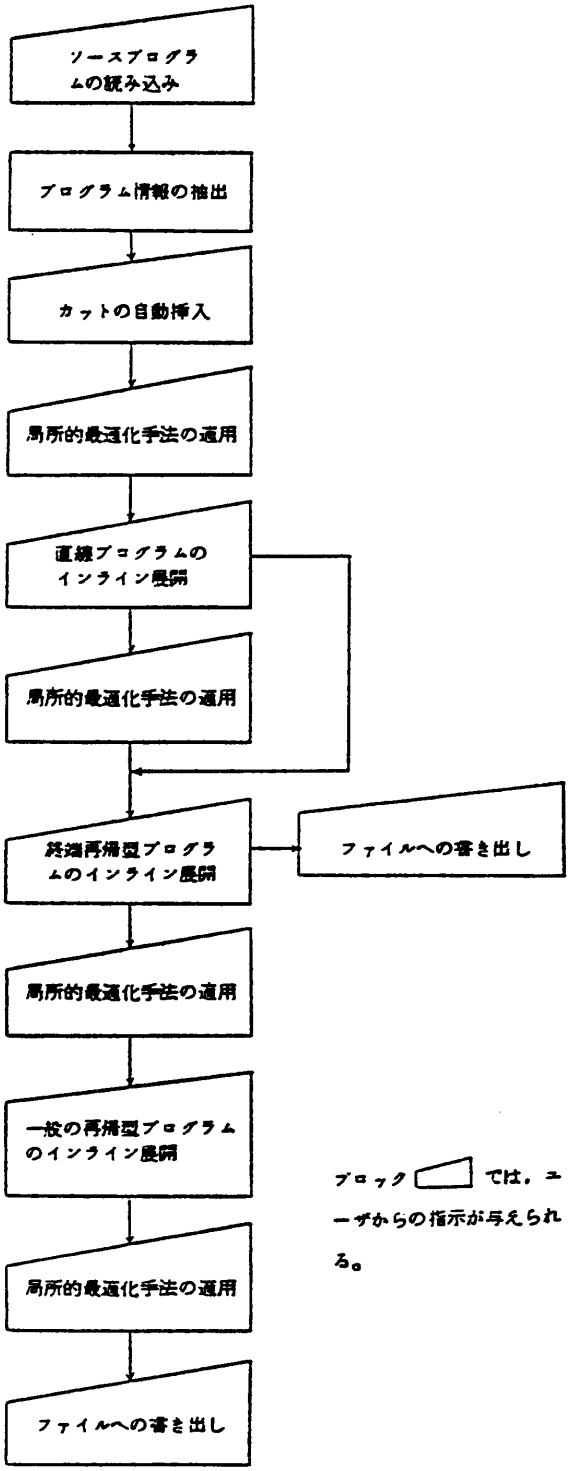

図1 Prolog ソースレベル・オプティマイザの手続きの流れ
Fig. 1 Procedural flow of the Prolog optimizer.

大局的には，本論文の最適化法は，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ると言える．他方，第二節の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にまとめられる．

(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．

オプティマイザへの入力は Dec system-10 Prolog[1] 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 インライン展開

③ 各種局所的最適化手法に対応する機能

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，一般の再帰型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1参照)．

オプティマイザプログラムは VAX 11/780/UNIX 上の CProlog[9] で書かれており，それは約 60,000 バイトの大きさである．

3.2 時間評価

この節では，最適化後の Prolog プログラムの時間評価を，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[1]の下で行った結果について述べる．プログラムの実行時間測定は最適化前と最適化後のプログラムに対して標準的なデータを与え，それを数十回繰り返し実行し，平均をとることによって行われた．

〔1〕 リスト反転プログラム

はじめに，次のよく知られたスタックを用いるリス

トの反転プログラムを対象として，我々の最適化法のストーリーを図示してみよう．

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

ここで，インライン展開の展開基準を，展開するべきゴールはその定義体が直線的である限り展開し，さらに終端再帰型プログラムはその終端再帰ゴールを一回限り展開する（言い換えると，元々のプログラムの型を展開によって壊さないこと）というようにすると，結果は次のようになる（より詳しい展開基準，アルゴリズムについては文献5) あるいは文献7)を見られたい）．

```
| ?- 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 : reversel
Optimized programs written to the file : reversel
PROLOG OPTIMIZER ends.

yes
```

図 2 Prolog オプティマイザの動作例
Fig. 2 An execution example of the Prolog optimizer.

```
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)
```

図2に，この最適化例に対する Prolog オプティマイザの実際の動作例を示す．その変換過程において，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：節の複数節への分解
es：等式代入による変数除去

最適化後のプログラム(1)，(4)，(5)，(6)に対しては，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（推論）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言える．

〔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[10]を実現するプログラムを最適化する．

$p(A)$:-$\Gamma, X=f(t1,\cdots,tn), X=f(r1,\cdots,rn), \Delta$.

$p(A)$:-$\Gamma, f(t1,\cdots,tn)=f(r1,\cdots,rn), \Delta$.

ここで，変数 X は $p(A), \Gamma, \Delta$ の部分には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，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[11]は KLO（あるいは，Prolog）[12]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$1$$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$$3' (append, A, args(B, C, D)):-!, 'lh$c$m$append$4' (A, B, C, D).
'lh$i$t$$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，約 40% の時間効率が得られている．

〔4〕 抽象データ型の定義に対するオプティマイザの適用

次のようにリストを抽象データ型として定義する(この例は文献 13)から取られた)．

```
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とは明らかである．

この二つの append の実行時間比は 1 対 0.4 であった．

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．

これまで，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．インライン展開は一般に Prolog プログラムのように述語（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しれない．

4. ま と め

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[14]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．

本論文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 3 節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論文の大きな寄与と考える．

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課題のほかに，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，本論文のオプティマイザの機能を，Prolog プログラムの変換技法として展開すること，および知的プログラミング[15]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ろう．

謝辞 日頃御指導，御鞭撻をいただく北川敏男会長，および榎本肇所長に感謝いたします．オプティマイザ技法に関して日頃熱心に討論してくれた国際研研究員，竹島卓，加藤昭彦，横森貴，南俊朗の諸氏に感謝します．富士通 SSL の黒川伊保子さんには，オプティマイザプログラムの開発に多大な協力をしてくれたことに対し感謝します．岸本光弘氏（富士通研・AI 研）は αProlog の開発者の立場から，本論文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）には Prolog の最適化全般にわたって御教示，御討論していただき，また第 4 節の ESP の例を示唆していただいた．R. Venken 氏（ベルギー・マネジメント研究所）は我々の研究報告[5]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．その一部は本論文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.

最後に，査読者の適切なコメントに対し感謝したい.

参 考 文 献

1) Bowen, D. L.: Dec System-10 Prolog User's Manual, Version 3.43, Dept. of Artificial Intelligence, Univ. of Edinburgh, Edinburgh (1981).
2) Warren, D. H. D.: Implementing Prolog-Compiling Predicate Logic Programs, D. A. I. Research Report, No. 39 and No. 40, Dept. of Artificial Intelligence, Univ. of Edinburgh, Edinburgh (1977).
3) Mellish, C. S.: Some Global Optimizations for a Prolog Compiler, *J. of Logic Programming*, Vol. 2, No. 1, pp. 43-66 (1985).
4) Sawamura, H. and Takeshima, T.: 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).
5) Sawamura, H., Takeshima, T. and Kato, A.: Source-level Optimization Techniques for Prolog, IIAS R. R. No. 52, ICOT-TM-0091 (1985) (also submitted to *J. of New Generation Computing*).
6) 沢村：PROLOG 述語（呼び出し）の決定性，コンピュータソフトウェア，ソフトウェア科学会に投稿中（1986）.
7) 沢村：PROLOG プログラムの最適化，国際研研究報告，第19号，富士通(株)（1986）.
8) 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，Vol. 23, No. 5, pp. 905-912 (1985).
9) Pereira, F. (ed.): C-Prolog User's Manual, Version 1. 4a, Nihon DEC (1983).
10) Chikayama, T.: Source-level Optimization in Logic Programming Languages, Draft (1983).
11) Chikayama, T.: ESP Reference Manual, ICOT-TR-044 (1984).
12) Chikayama, T., Yokota, M. and Hattori, T.: Fifth Generation Kernel Language, Version 0, ICOT-TR (1983).
13) Venken, R.: A Prolog Meta-interpreter for Partial Evaluation and Its Application to Source-to-source Transformation and Query-optimization, *ECAI 84: Advances in Artificial Intelligence*, O'Shea, T. (ed.), pp. 91-100, Elsevier Science Publishers B. V., North-Holland (1984).
14) Sato, T. and Tamaki, H.: Unfold/fold Transformation of Logic Programs, *Proc. of 2nd Int. Logic Programming Conference*, Uppsala, pp. 127-138 (1984).
15) 玉木，佐藤：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).

（昭和61年 7 月14日受付）
（昭和61年12月10日採録）

沢村 一（正会員）

昭和24年生．昭和53年北海道大学工学部情報工学専攻修了．富士通(株)国際情報社会科学研究所に勤務．論理学とその応用，計算美学，Macintosh 等に興味をもつ．日本ソフトウェア科学会，日本数学会，科学哲学会，RSSA 等の会員.