部分計算のメタ・プログラミングへの応用

竹内彰一　　近藤浩康　　大木優　　古川康一
（財）新世代コンピュータ技術開発機構

１．　はじめに

プログラムの部分計算とは一般に『稼働環境に関する情報を利用して、汎用のプログラムをより効率の良いプログラムに特殊化すること』と要約される［二村83］。部分計算の原理は任意のプログラミング言語で書かれたプログラムに適用でき、応用上の意義もコンパイラの自動作成など非常に大きい。

本稿では、Prologプログラムを対象にした実用的な部分計算プログラムの実現法について述べる。さらに、Prologプログラミングの中における部分計算の意義を主にメタ・プログラミングとの関連から述べる。

２．　Prologプログラムの部分計算

一般に、プログラムの部分計算では入力データのうち部分的にわかっている部分入力データに合せて特殊化する。Prologプログラムの場合、入力データとなるものには次の２つがある。

① 入力データ的節集合
② ゴール文、特に、ゴール文中の引数の値

Prologプログラムの一般な部分計算では、①、②両種のものについて、わかっている範囲に応じて部分計算することが望ましいが、本稿で述べる部分計算は次の制約をもつ。

(a) 入力データ的節の集合は完全に与えられているものとする。
(b) ゴール文については、その中に現れる述語名についてはわかっていなくてはならない。

従って、本稿で述べる部分計算法は、入力データ的節の集合とゴール文中の述語の引数の値がいくつか与えられているときに、それを解く Prolog プログラムを与えられた情報に合せて特殊化するものであるといえる。

２．１　部分計算の制御

一般に Prolog プログラムの計算はゴールをルート・ノードとし、プログラムに対応して構成される AND-OR 木を depth-first、 left-to-rightに探索することと見なせる。部分計算はこの AND-OR 木を、与えられた情報だけに基づいて部分的に探索し、trueになる枝を見つけるとそれを除去し、かわりに得られた unifierを ANDノードを介して他の枝に反映させたり、 falseになる枝を見つけるとそれを刈ったりする。この部分計算が行なう AND-OR 木上の探索の仕方としては、 top-down 、bottom-up 、middle-outなどが考えられるが、本稿で述べる部分計算法では、 Prolog と同じ、depth-first 、left-to-right を用いている。すなわち、トップ・レベルで与えられたゴールから始まり、サブ・ゴール、サブ・サブ・ゴール、．．．とゴール文に導かれながらそのゴールを解くサブ・プログラムの部分計算を進めている。 Prolog の実行と類似なこの部分計算の制御を以下ゴール主導型と呼ぶことにする。

２．２　ゴール主導型部分計算

前節では部分計算のマクロな流れを述べた。本節では、Prologプログラムの部分計算では『何が計算できて、何が計算できないか』について述べ、アルゴリズムの概略を述べる。

部分計算の原則は与えられたプログラムの中で、計算できる部分を計算し、計算できないものは元のままに残して置くことと言える。一般に関数型言語のような値指向の計算においては、変数の値の有無が計算できる、できないにつながり、値がないままに計算する場合には遅延評価などの特殊な評価方式をとらなくてはならない。しかし、Prologの計算は、ユニフィケーションをベースにしているので、基本的に部分計算時に特殊な評価方式を必要としない。このことはPrologの重要な特徴である。このことを念頭において、次の簡単な Prolog プログラムを部分計算してみる。

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

ただし、データとして、

```
parent(taro,jiro).
parent(jiro,saburo).
```

が与えられているものとし、ゴールとしては ancestor(X,Y)を考える。この例の場合、ゴール ancestor(X,Y)から始めて部分計算していくと、結局、結果として、次の特殊化されたプログラムが得られることが予想される。

```
ancestor(taro,jiro).
ancestor(taro,saburo).
ancestor(jiro,saburo).
```

これは、ancestorのもとのプログラムを2つの parent 関係に関して特殊化したものになっている。ところで、この結果はゴール ancestor(X,Y)の解集合と一致している。すなわち、ゴール ancestor(X,Y)を始点とする部分計算の結果として予想されるものとゴール ancestor(X,Y)の解集合を求める全解計算の結果とが一致している。このことは、Prologプログラムのゴール主導型部分計算のアルゴリズムと全解計算のアルゴリズムが相似であることを意味している。実際対応する AND-OR 木が有限であるような pure な Prolog プログラムの部分計算による特殊化の最たるものは全解計算によるゴールのすべての instance 求めることに他ならない。ところが、多くの Prolog プログラムは有限な AND-OR 木をもたないし、またシステム述語などを用いており pure ではない。従って、 Prolog プログラムの汎用な部分計算は全解計算に相似でありながら、無限な AND-OR 木が扱えたり、システム述語のような pure でない述語が扱えなくてはならない。

本稿で述べるゴール主導型部分計算を実現するプログラムを以下 PEVALと呼ぶ。 PEVALはなるべく多くの Prolog プログラムを部分計算することができる実用的なものを目指して設計され、Prologで実現されている。しかし、現在のところ、 PEVALはカットを直接扱わない。カットは部分計算の立場からは unfoldingを阻害する厄介なものであり、制御プリミティブとしては、if、caseなどのもう少しマクロなものの方が部分計算上は扱い易い。従って現在はもとの Prolog プログラムをカットを使うものから、if, case などを使うものへと変換した後部分計算する方法をとっている。

PEVAL は次のようなプログラムを扱える。

① 対応する AND-OR 木が有限／無限なプログラム
② システム述語を含むプログラム
③ 閉じていないプログラム（使用されるすべての述語の定義を含まないプログラム）
④ ifを含むプログラム（カットは扱えない）

PEVAL の基本アルゴリズムを図1に示す。PEVAL は集合を値としてとる2つの関数、peval-clauses と peval-goals で記述される。

```
peval-goals(Goals, θ0) :=
  if Goals=G1 ∧ ・・・ ∧Gn
     then {(θ1,B1' ∧Grest) |
              (H:-B)∈ peval-clauses(G1・θ0) ,
              θ := unify(H,G1),
              (θ1,Grest) ∈
           peval-goals(G2∧・・∧Gn ,θ0 ・θ)}
     else {(θ0,true)}

peval-clauses(Goals) :=
  {(H':-B') | (H:-B)∈definition(Goal),
              θ0 :=unify(Goal,H),
              (θ1,B')∈ peval-goals(B,θ0),
              H' :=H・θ0 ・θ1}
```

Fig.1 PEVALの基本アルゴリズム

peval-goals(Goals , θ0)はゴールの並び Goals（空のこともある）とユニファイθ0 を受取り、Goals ・θ0 を部分計算した結果得られた新しいゴール列とユニファイを組にしたものの集合を返す。そのために、まずゴール列の先頭のゴール G1 をとり、その定義節を peval-clausesを用いて部分計算し、得られた節集合の各々を用いて、ゴール列中の G1 を unfoldingし、残りのゴール列を peval-goalsでさらに部分計算する。peval-goals は受取ったゴール列が空のときは、ユニファイアと true を組にして返す。peval-clauses(Goal) は Goal の定義節集合の中で Goal と節のヘッドがユニファイ可能であるものについて、その節の右辺を peval-goalsにより部分計算する。peval-clauses の値は、Goalとユニファイ可能なヘッドをもつ部分計算された節の集合である。

部分計算は全体として以下のように進む。まず始点となるゴール列が与えられると、それを peval-goalsを用いて左端のゴールの定義節の部分計算をし、結果の節を用いてゴールを unfoldingし、残りのゴール列について同様のことを行なう。定義節の部分計算は peval-clausesが行ない、ゴールとユニファイ可能な節を抽出し、各節の右辺のゴー

ル列を peval-goalsを用いて部分計算する。このように全体の流れは、Prologと同じ left-to-right、depth-first になっている。

一般には部分計算時にループに陥ることがある。同じゴールの部分計算や、すでに部分計算したゴールの特殊形の部分計算は冗長なので、これを避けるために、部分計算は現在部分計算中のゴールのスタックを管理し、新しく部分計算するゴールがこのスタックにあるゴールと変数名を除いて全く同じか、またはスタック中にあるゴールを特殊化したものに等しいときに部分計算を止め同じゴールを部分計算の結果として返している。また、ゴールがシステム述語である場合、その部分計算を peval-clausesでする際には、そのゴールが評価可能かどうかを調べ、可能であれば実行し、結果として true を返し、そうでなければ結果として同じゴールを返すようにしている。この２つのことによって PEVALは対応する AND-OR 木が無限であるプログラムやシステム述語を含むプログラムを部分計算できるようになっている。

以上は PEVALの基本動作であるが、ユーザは次の２つの形式で PEVALを制御することができる。

(a) type(Goal,Type) :- <Condition>.
(b) inhibit-unfolding(Goal).

(a) は部分計算のいわば前向きの進行を制御するもので、(b) は後向きの進行を制御するものである。(a) は『ゴール Goal は条件 <Condition>を満足するときにタイプTypeとする』と読む。 Type には

e ・・・ 評価可能
t ・・・ 停止
g ・・・ 一般（部分計算）

の３つがある。ゴールをｅと指定することは部分計算時にゴールを完全に評価して良いということであり、ｔと指定することは部分計算の停止、ｇと指定することは部分計算を普通に進めることを意味する。ｔの指定は特に部分計算の対象としているプログラムが閉じていないときに有効である。すなわち、定義節をもたないゴールについて部分計算を止めるという指定が出来る。(b) はゴール Goal の定義節を部分計算した後で、その節を使って Goal を unfoldingすることを禁止する指定である。これは一般に部分計算の結果生成される新しいプログラムのコード量を少なく抑えるために用いる。例えばゴール列 p,qがある場合、 pの新しい定義節がｎ個、 qの新しい定義節がm個あったときに、双方の unfoldingを行なうと新しいゴール列がｎ×m個できる。一方、双方の unfoldingを禁止すると、コード量はｎ＋m個の節と１個のゴール列で済む。一般に unfoldingの禁止はコード量だけでなく、効率にも影響を与えることが予想されるが、これを機械的に制御する方法は未解決である。

最後に部分計算の結果の新しいプログラムの蓄積の仕方について述べる。部分計算後の新しいプログラムは元のプログラムとは区別して Prolog のデータベース上に蓄えられる。生成されるプログラムは冗長になる可能性があるので、新しく蓄えるときには冗長なものは加えないようにしている。部分計算をしなかったものや、途中で止めたもの、また unfoldingをしなかったものについては注意が必要である。このようなものとしては、

①部分計算中にループが検出され、部分計算が中断されたゴールの定義節
②type述語によりｔと指定されたゴールの定義節
③inhibit-unfolding の指定のあったゴールの定義節

この中で、①についてはループ検出の方法から問題ない。すなわち、ループを検出したときにスタック中にあった方のゴール（こちらの方が一般的な形をしている）の新しい定義節は求められ、蓄積されるからである。②については、新しいプログラムの中にこれらのゴールの定義節がないという情報を入れることにしている。③については、自動的に新しいプログラムに蓄積されることになっている。

３． メタ・プログラミングへの応用

３．１ メタ・プログラミング

メタ・プログラミングは論理型プログラミングでも広く使われている強力なプログラミング技法である。メタ・プログラミングは次のように特徴付けることが出来る。

(1) プログラムをデータとして扱えること
(2) データをプログラムとして扱い、評価できること
(3) プログラムの実行結果 (true,fail)をデータとして扱えること

メタ・プログラミングの例として最も良く知られているのは Bowenと Kowalski による demo 述語である[Bowen83]。demo述語はプログラムとゴールの２つを引数として取り、ゴールがプログラムから証明できるときに true となり、それ以外は fail となる。

```
demo(Program,Goal)
        →  true      if  Program ⊢ Goal
        →  fail      otherwise
```

demo述語の論理的意味はゴールのプログラムからの導出可能性という重要な意味であり、Bowenらはこのdemo述語を用い、メタ・レベルとオブジェクト・レベルを融合させた強力なプログラミング例を示した。メタ・プログラミングはまた問題解決においても有用であり、問題解決のメタ・レベルの制御はBundyらにより研究されている[Bundy81]。また、メタ・プログラミングはプログラミング環境を構築する上でも非常に重要である。Shapiroの Algorithmic Program Debuggingなどもメタ・プログラミングを多用している[Shapiro83a]。

3.2 メタ・プログラムの部分計算

メタ・プログラムとして実際に作られるものにはメタ・インタプリタの変型が多い。例えば、上述の debuggerなどはその例である。APES [Hammond83]は論理ベースのエキスパート・システム構築用ツールであるが、これもメタ・インタプリタ的なものを用いている。APESはHorn節を推論ルールと見なし、推論エンジンとしてHorn節のメタ・インタプリタを作ってこの中に説明機能などを埋め込んでいる。certainty factorの処理もこのメタ・インタプリタの中で扱うことが可能と思われる。Shapiro [Shapiro83b]はcertainty factorを扱うようなHorn節のメタ・インタプリタの提案をしている。Prologを用いてメタ・インタプリタ的なものを用いずにエキスパート・システムを作る試みはClarkらによって行なわれた[Clark82]。Clarkらは推論ルールとして用いられるHorn節の各々に説明機能やcertainty factorの扱いをするコードを埋め込む方法をとった。このような方法とメタ・インタプリタ的なものを用いる方法を比較するとメタ・インタプリタ的なものを用いる場合の利点は、オブジェクト・レベルとメタ・レベルを明確に区別することにより、プログラムを見易く、かつ書き易くできるという点にあり、欠点はインタプリティブに実行するため実行速度が遅いということである。逆にメタ・インタプリタを使わない方法の利点は速度が速いということであり、欠点はプログラムが見にくいということである。

本稿で述べたPrologプログラムの部分計算法PEVALは上に述べたようなメタ・インタプリタ的なプログラムを効率の良いプログラムへと変換するのに極めて有効である。実際、メタ・インタプリタにとってオブジェクト・プログラムは入力データ的なものであるので、メタ・プログラムをオブジェクト・プログラムを与えて部分計算し、特殊化することができる。この特殊化されたプログラムは、機能的にはメタ・インタプリタ的なものを用いずにすべてをオブジェクト・レベルのプログラムに埋め込んだものに極めて近くなる。従って効率についても、特殊化されたメタ・プログラムは、メタ・インタプリタを使わなかったものと同等のものになる。

具体的な例を用いて説明する。図2(a)に certainty factorを扱うPrologメタ・インタプリタを示す。solveは2引数で第1引数が解くべきゴール、第2引数はそのゴールが解かれたときの certainty factorである。図2(b)はこのメタ・インタプリタにより解釈実行されるオブジェクト・レベルのプログラムである。患者が症状に応じてどういう薬をのむべきかを推論するプログラムである。各Horn節の後に付加されている『 <> N 』はその節の certainty factor が Nであることを表している。図3(a)に図2(a)のメタ・プログラムを図2(b)のオブジェクト・プログラムに関して部分計算するためのデータを示し、図

```
solve(true,[100]).
solve((A,B),Z) :-
    solve(A,X), solve(B,Y), ap(X,Y,Z).
solve(not(A),[CF]) :-
    solve(A,[C]), C < 20, CF is 100-C.
solve(A,[CF]) :-
    rule(A,B,F) , solve(B,S), cf(F,S,CF).

cf(X,Y,Z) :-
    product(Y,100,YY), Z is (X*YY)/100.
product([],A,A).
product([X|Y],A,XX) :-
    B is X*A/100, product(Y,B,XX).

rule(A,B,F) :- ((A:-B)<>F).
rule(A,true,F) :- (A<>F).
```

(a) Meta Program

```
should_take(Person,Drug) :-
   complains_of(Person,Symptom),
   suppresses(Drug,Symptom),
   not(unsuitable(Drug,Person)) <> 70.

suppresses(aspirin,pain) <> 60.
suppresses(lomotil,diarrhoea) <> 65.

unsuitable(Drug,Person) :-
   aggravates(Drug,Condition),
   suffers_from(Person,Condition) <> 80.

aggravates(aspirin,peptic_ulcer) <> 70.
aggravates(lomotil,
           impaired_liver_function) <> 70.
```

(b) Object Program

Fig. 2 Meta Interpreter handling certainty factors

```
type(solve(complains_of(_,_),_),t) :- !.
type(solve(suffers_from(_,_),_),t) :- !.
type(solve(_,_),g) :- !.
type(rule(_,_,_),e) :- !.
type(cf(X,Y,Z),e) :- integer(X), ground(Y), !.
type(cf(X,Y,Z),t) :- var(X) ; \+(ground(Y)).
type(ap(X,Y,Z),e) :- fixed_length(X),!.
type(ap(X,Y,Z),t) :- \+(fixed_length(X)),!.
type(X<Y,e) :- integer(X),integer(Y),!.
type(X<Y,t) :- var(X) ; var(Y).
type(X>Y,e) :- integer(X),integer(Y),!.
type(X>Y,t) :- var(X) ; var(Y).
type(X is Y,e) :- ground(Y),!.
type(X is Y,t) :- \+(ground(Y)),!.
```

(a) Instructions to PEVAL

```
solve(should_take(A,aspirin),[B]) :-
   solve(complains_of(A,pain),C),
   solve(suffers_from(A,peptic_ulcer),D),
   cf(80,[70|D],E),E<20,F is 100-E,
   ap(C,[60,F],G),cf(70,G,B).
solve(should_take(A,lomotil),[B]) :-
   solve(complains_of(A,diarrhoea),C),
   solve(suffers_from(A,
              impaired_liver_function),D),
   cf(80,[70|D],E),E<20,F is 100-E,
   ap(C,[65,F],G),cf(70,G,B).

solve/2 unprocessed.
cf/3 unprocessed.
ap/3 unprocessed.
```

(b) Result of Partial Evaluation

Fig. 3 Partial Evaluation

3(b) に部分計算の結果を示す。

図2(a) のプログラムと図3(b) のプログラムを比較すると図2(a) のプログラムは汎用のインタプリタであるのに対し、図3(b) のプログラムは完全に図2(b) のプログラムへと特殊化されたものになっている。また、図2(b) のプログラムと図3(b) のプログラムと比較すると、図3(b) のプログラムが図2(b) のプログラムにメタ・レベルの certainty factor の処理が埋め込まれたものになっていることがわかる。図4には、inhibit-unfolding 指定をつけて部分計算した例を示す。この例ではコード量は部分計算の結果のプログラムの方が多いが、図からわかるように inhibit-unfolding を指定したことによりオブジェクト・プログラムに良く似た構造をした特殊化されたメタ・プログラムを得ている。以上述べたプログラムの実行効率の比較を表1に示す。表よりコンパイル実行では部分計算により特殊化されたプログラムの効率が図2のプログラムの約3倍であり、また図2(b) のプログラム単体の実行効率の約2／3であることがわかる。

```
inhibit_unfolding(solve(Goal,_)) :-
   \+(Goal=true),\+(Goal=(P,Q)),
   \+(Goal=not(P)).
```

(a) Additional instructions

```
solve(suppresses(aspirin,pain),[60]).
solve(suppresses(lomotil,diarrhoea),[65]).
solve(aggravates(aspirin,peptic_ulcer),[70]).
solve(aggravates(lomotil,
        impaired_liver_function),[70]).
solve(unsuitable(A,B),[C]) :-
   solve(aggravates(A,D),E),
   solve(suffers_from(B,D),F),
   ap(E,F,G),cf(80,G,C).
solve(should_take(A,B),[C]) :-
   solve(complains_of(A,D),E),
   solve(suppresses(B,D),F),
   solve(unsuitable(B,A),[G]),
   G<20,H is 100-G,
   ap(F,[H],I),ap(E,I,J),cf(70,J,C).

solve/2 unprocessed.
ap/3 unprocessed.
cf/3 unprocessed.
```

(b) Result of Partial Evaluation

Fig.4 Partial Evaluation

Table 1 実行時間 (CPU Time) 比較

	p1	p2	p3	p4
インタプリティブ実行	1674	901	1157	95
コンパイル実行	110	39	46	26

p1：メタ＋オブジェクト・プログラム（図2）
p2：特殊化されたプログラム（図3(b)）
p3：特殊化されたプログラム（図4(b)）
p4：certainty factorなしのオブジェクト・プログラム（図2(b)）

4. 推論システム構築の基本ツールとしての部分計算

本節では推論システムを構築する際の部分計算のもたらす意義を考察する。

論理型言語 Prolog は従来次のような理由で推論システムを構築する良いベース言語であると言われてきた。

①論理変数／ユニフィケーション
Prologが備えている論理変数およびユニフィケーションに基づく計算メカニズムが通常のパターン・マ

ッチングやパターン駆動の計算機構を含んだより強力なものであること。

②バックトラッキングによる探索機構

推論システムにおいては、解の探索が不可欠であるが、Prologではバックトラッキングを行なう計算機構が背後にあるため、探索アルゴリズムが容易に記述できる。

しかしながら、Prologが推論システムを構築するための良いベース言語であるということは構築の方法まで示唆しているわけではなく、今まで、Prologに基づいて多くの推論システムが様々な方法で作成されてきた。本節では、従来 Prolog で推論システムを構築する際にとられてきた方法を整理して、問題点を明確にし、部分計算を基本ツールをする新しい方法を提案する。

推論システムは一般的に次のものから構成される。

(a) 推論ルール
(b) 推論部

ルールは対象とする問題領域に関する知識であり、推論部は入力（問題）をユーザから受けとると(a) のルールを用いて、ある戦略に基づき推論を行なうものである。

具体例を挙げる。プロダクション・システムは良く知られているようにルール・ベース推論システムの一種である。プロダクション・ルールが(a) のルールに対応し、プロダクション・システムの推論部がまさに(b) の推論部に対応する。前向き、後向きプロダクション・システムはこの推論部の行なう推論の戦略が後向き（あるいは top-down ）、前向き（あるいは bottom-up）であることにそれぞれ対応する。また、構文解析システムを一般的な推論システムの一例と考えることが出来る。この場合、文法ルール及び辞書が(a) のルールに対応し、解析部が(b) の推論部に対応する。構文解析についても、 top-down 的アルゴリズム、 bottom-up的アルゴリズム等があるが、これは推論部の推論の戦略にまさに対応している。さらに、また Prolog 処理系も広い意味で推論システムとみなせる。この場合、Prologプログラムがルールに、Prologインタプリタが推論部に対応する。

通常、推論システムの推論部の中では基本的な推論の他にメタ的推論が実現される。メタ的な推論とは『推論についての推論、推論の制御、推論の観察など』とここでは考える。例えば、プロダクション・システムにおいて、各ルールに certainty factor が付いているときに、推論結果に certainty factor を付加し、推論を制御することは推論部で行なわれる。また、Prologプログラムを実行したときに実行の履歴をとるといったある意味で推論の観察を行なうということは実行の履歴をとる Prolog インタプリタを作ることで実現される。

推論システムの作成の方法としては次の２つが代表的である。

(1) 推論部をルール・インタプリタとして作る。
(2) ルールをプログラムへ変換する。

(1) の方法をとっているものとして、前述のAPESや Shapiroの certainty factor を扱うメタ・インタプリタがある。APESは Horn 節をルール表現として用いる『論理型』のエキスパート・システムである。APESではルール表現は Horn 節であるが、推論の履歴をとる部分をルール（＝Horn節）のインタプリタの中に埋め込んでいる。すなわち、システムとしては前述のルール・ベースと推論部の２つからなる。(2) の方法をとっているものとしては、前述のClark らのエキスパート・システム、田中によるプロダクション・システム［田中84］、DCG,BUP などがある。いずれもルールをPrologプログラムへと変換し、推論を推論部というインタプリタなしに Prolog プログラムの直接実行という形で行なっている。

一般に(1) の方法をとる場合の利点は推論の動きが明確で理解しやすく、かつ、推論部の変更が容易であるということにあり、欠点は推論の速度が遅いということにある。一方(2) の方法の利点は、推論がプログラム化されたルールの直接実行であるから速度が速いということであり、欠点はルールをプログラムへと変換するプログラムが理解しずらいという点にある。この変換プログラムは一般にシンタックス変換、入出力等を含んでいて、推論の戦略がその中でどう表現されているかを見つけるのが難しく、従って、推論に本質的な部分を理解すること、および、その部分を変更することが極めて困難である。

このように方法(1),(2) それぞれの利点、欠点は互いに相補的な関係にある。すなわち、実行効率と推論部の汎用性の間のトレード・オフがそのままそれぞれの方法の利点欠点に現れている。本稿で提案する新しい方法は方式(1),(2) を融合したものであり、実行効率と汎用性の間のトレード・オフを部分計算により解消している。その結果、本稿で提案する方式は特徴としてこの両方式の利点を合せ持ち、いずれの方式の欠点も持たない。この新方式では、推論システムは方式(1) と同様に推論部と推論ルールに分け

て構築される。この段階では推論部は汎用のものが記述され、推論の行なわれ方は理解し易く、また変更も容易である。また特にメタ的な推論を推論部に埋め込むのも容易である。この推論部に推論ルールを与えて実行するときは、推論ルールを固定して、部分計算により推論部を汎用のものから特殊なものへと変換することにより方式(2) の実行効率と同等のものを達成できる。

この新方式の中で部分計算の果す役割はまさに方式(1) の汎用性、maintenability, readabilityと方式(2) の実行効率の融合にある。この方式の有効性は、従来 Prolog で構築されてきた推論システムを用いて確かめることができる。前節の certainty factor を扱う Prolog プログラムはその一例である。プログラムは図２にあるように推論部（メタ・プログラム）と推論ルール（オブジェクト・プログラム）に分けてあることにより maintenability, readability ともに良くなっている。また実行効率は表１にあるように部分計算により大きく向上している。

本方式の有効性を確認する別の例として BUP(Bottom Up Parser)[Matsumoto83] を取上げる。BUP はCFG のボトム・アップ・パーザであり、前述のように構文解析システムもまた推論システムの一例と考えることができる。推論システムの構築の方法という観点から BUPを眺めると、従来 BUPは方式(2) で実現されてきた。すなわち、 CFGルールを BUPトランスレータにより Prolog プログラムへ変換して実行し、高い実行効率を得ていた。しかしながら、前述のようにこの方法をとる場合には、maintenability、readability に問題がある。そこで本方式の有効性を示すために、 BUPのインタプリタを記述し、その maintenability, readability の良さを確認し、その後、この BUPインタプリタを特定の CFGルールを与えて特殊化し、結果として BUPトランスレータが生成するのと同等のコードが得られることを示す。図５(a) に BUPのインタプリタ、図５(b) に CFGルールを示す。また図５(c) に BUPトランスレータにより図５(b) のCFG ルールを変換した結果の Prolog プロ

```
goal((P,Q),S0,S) :- goal(P,S0,S1), goal(Q,S1,S).
goal(C,S,S1) :- dict(F,S,S2),link(F,C),derive(F,S2,C,S1).

derive(F,S,F,S).
derive(F,S2,C,S1) :-
    rule1((Lemma <= (F,Rest))),link(Lemma,C),
    goal(Rest,S2,S3),derive(Lemma,S3,C,S1).
derive(F,S2,C,S1) :-
    rule2((Lemma <= F)),link(Lemma,C),
    derive(Lemma,S2,C,S1).

link(C,C).
link(F,C) :- rule1((Lemma <= (F,_))),link(Lemma,C).
link(F,C) :- rule2((Lemma <= F)),link(Lemma,C).

dict(F,[X|S],S) :- rule((F <= [X])).

rule1((A <= (B,C))) :- rule((A <= (B,C))).
rule2((A <= B)) :- rule((A <= B)),\+(B=(_,_)),\+(B=[_|_]).
```

(a) BUP interpreter

```
rule((s <= (np,vp))).          rule((np <= (det,n))).
rule((vp <= vi)).              rule((vp <= (vt,np))).
rule((n <= [boy])).            rule((n <= [girl])).
rule((vi <= [walks])).         rule((vt <= [likes])).
rule((det <= [a])).            rule((det <= [the])).
```

(b) CFG rules

```
dict(n,[],[boy|A],A).          dict(n,[],[girl|A],A).
dict(vi,[],[walks|A],A).       dict(vt,[],[likes|A],A).
dict(det,[],[a|A],A).          dict(det,[],[the|A],A).
link(X,X).                     link(det,np).
link(vt,vp).                   link(vi,vp).
link(np,s).                    link(det,s).
vt(vt,_421,_422,_422,_421).
vi(vi,_443,_444,_444,_443).
det(det,_465,_466,_466,_465).
n(n,_487,_488,_488,_487).
np(np,_509,_510,_510,_509).
vp(vp,_531,_532,_532,_531).
s(s,_553,_554,_554,_553).
np(B,[],C,D,E) :-
   link(s,B), goal(vp,[],C,F),call(s(B,[],F,D,E)).
det(B,[],C,D,E) :-
   link(np,B), goal(n,[],C,F),call(np(B,[],F,D,E)).
vi(B,[],C,D,E) :-
   link(vp,B), call(vp(B,[],C,D,E)).
vt(B,[],C,D,E) :-
   link(vp,B), goal(np,[],C,F),call(vp(B,[],F,D,E)).
goal(CurGoal,Arg,S0,S) :-
   dict(Nt,Arg1,S0,S1), link(Nt,CurGoal),
   functor(Pred,Nt,5), arg(1,Pred,CurGoal),
   arg(2,Pred,Arg1), arg(3,Pred,S1),
   arg(4,Pred,S), arg(5,Pred,Arg),
   call(Pred).
```

(c) Code generated by BUP translator

Fig. 5 Bottom-Up Parser

```
type(dict(_,_,_),e).           type(rule1(_),e).
type(rule2(_),e).              type(link(_,_),e).
type(goal(_,_,_),g).           type(derive(_,_,_,_),g).

inhibit_unfolding(dict(_,_,_)).
inhibit_unfolding(link(_,_)).
inhibit_unfolding(goal(C,_,_)) :- \+(C=(P,Q)).
inhibit_unfolding(derive(_,_,_,_)).
```

(a) Instructions to PEVAL

```
dict(det,[a|A],A).             dict(det,[the|A],A).
dict(n,[boy|A],A).             dict(n,[girl|A],A).
dict(vi,[walks|A],A).          dict(vt,[likes|A],A).
link(A,A).                     link(det,np).
link(det,s).                   link(np,s).
link(vi,vp).                   link(vt,vp).
derive(A,B,A,B).
derive(det,A,B,C) :-
   link(np,B),goal(n,A,D),derive(np,D,B,C).
derive(np,A,B,C) :-
   link(s,B),goal(vp,A,D),derive(s,D,B,C).
derive(vt,A,B,C) :-
   link(vp,B),goal(np,A,D),derive(vp,D,B,C).
derive(vi,A,B,C) :-
   link(vp,B),derive(vp,A,B,C).
goal((A,B),C,D) :-
   goal(A,C,E),goal(B,E,D).
goal(A,B,C) :-
   dict(D,B,E),link(D,A),derive(D,E,A,C).
```

(b) Code generated by PEVAL

Fig. 6 Partial Evaluation of BUP interpreter

グラムを示す。図5(a) の BUPインタプリタは BUPトランスレータと比べるとコンパクトであり、図5(c) のプログラムと比べるとボトム・アップな構文解析アルゴリズムを理解し易く、従ってこれを修正するのも容易である。図5(a) のインタプリタを部分計算するための補助情報を図6(a) に、また部分計算の結果を図6(b) に示す。図5(c) の BUPトランスレータの生成したプログラムと図6(b) の部分計算の結果のプログラムを比較すると両者が本質的に同じであることが容易にわかる。すなわち、部分計算の結果生成されたプログラムの実行効率は BUPトランスレータの生成するプログラムのそれに等しい。従って、本方式を用いて構文解析システムを実現すると maintenability, readability 、実行効率すべての点で良いものが得られる。

部分計算を基本ツールとする本方式による推論システムの構築法は以上のように Prolog 上に作成された既存のものについても有効に適用することができる。また本方式は十分に一般的であるので、推論システムを構築する方法のの今後の方向を示唆するものと考える。

5. まとめ

Prologプログラムの部分計算法について述べ、そのメタ・プログラミングへの応用について報告した。Prologプログラムの部分計算は Komorowski[Komorowski81] により Lisp の方言である QLOG で実現されてもいるが、本稿の PEVALは Prolog で記述されている。Prologの計算はユニフィケーションをベースにしており、部分計算時にも特別な計算機構なしにユニフィケーションを計算のベースにすることができる。従って実現の容易さという点では Prolog によるのが良いと思われる。

メタ・プログラミングは Prolog プログラミングの中でその表現力の強力さ故に重要な役割を果しつつあるが、部分計算はこのメタ・プログラムの実行効率を向上させることによりメタ・プログラミングをより実用的なプログラミング技法にすることができるといえよう。部分計算のメタ・プログラミングへの応用としては推論システムの効率化以外にも、任意のプログラムにメタ的制御やメタ的入出力を付加することが考えられる。それは、付加すべき制御や入出力をメタ・インタプリタで定義しておき、このメタ・インタプリタをプログラムを固定しておいて部分計算を行なうことによって実現されるのであるが、これについては別の機会に報告する。

今後の方向としては PEVALを改良しより広い範囲の Prolog プログラムを扱える強力なものにすること、及びプログラミング環境の中に部分計算をツールとして有機的に取り込む方法などについて研究する予定である。

謝辞

本研究を行なう機会を与えていただいた渕一博 ICOT 研究所所長に感謝致します。また有益な助言をいただいた第2研究室をはじめとする ICOT 研究所の方々に感謝致します。

文献

[Bowen83] K.Bowen, R.Kowalski : Amalgamating Language and Metalanguage in Logic Pragramming, In K.Clark and S.Tarnlund (eds.) Logic Programming、Academic Press (1983).

[Bundy81] A.Bundy, B.Welham: Using Meta-level Inference for Selective Application of Multiple Rewrite Rules in Algebraic Manipulation, Artificial Intelligence 16 (1981).

[Clark82] K.Clark, F.McCabe: Prolog: A Language for Implementing Expert Systems, In D.Michie and Y.H. Pao (eds.) Machine Intelligence 10 (1982).

［二村83］二村良彦：プログラムの部分計算法、電子通信学会誌 Vol.66, No.2 (1983).

[Hammond83] P.Hammond et al.: A Prolog Shell for Logic Based Expert Systems, Imperial College, 1983.

[Komorowski81] J.Komorowski: A Specification of Abstract Prolog Machine and its Application to Partial Evaluation, Linkoping studies in Science and Technology Dissertations No.69 (1981).

[Matsumoto83] Y.Matsumoto et al.: BUP: A Bottom-Up Parser Embedded in Prolog, New Generation Computing , Vol.1, No.2 (1983).

[Shapiro83a] E.Shapiro: Algorithmic Program Debugging, The MIT press, 1983.

[Shapiro83b] E.Shapiro: Logic Programs with Uncertainties: A Tool for Implementing Rule-Based Systems, Proc. of IJCAI'83, 1983.

［田中84］田中穂積：Prologによるルール型知識表現法とその利用、電子通信学会オートマトンと言語研究会資料 AL84-48 、1984.