ICOT Technical Memorandum: TM-0390

TM-0390

自己適用可能な部分計算プログラムの実現と応用

藤田　博，古川康一

September, 1987

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

自己適用可能な部分計算プログラムの実現と応用

藤田　博　．　古川　康一

（財）新世代コンピュータ技術開発機構

部分計算プログラムをある言語のインタプリタについて部分計算して特殊化すると，その言語のコンパイラ相当のプログラムが得られることが知られている．さらに，部分計算プログラムを部分計算プログラムについて部分計算して特殊化すると，コンパイラジェネレータ相当のプログラムが得られる．１つの部分計算プログラムがそれ自身入力データとして計算の対象とされ，有為な出力を得ることができるとき，自己適用可能と言われる．

本報告では，自己適用可能なPrologの部分計算プログラムを実現し，コンパイラ生成，コンパイラジェネレータ生成が達成できることを示す．さらに，段階的コンパイリングへの応用について述べる．

これらの結果は，簡単なPrologの自己インタプリタを拡張することによって得られた．部分計算プログラムとしては機能を最小限に止どめたため，自己適用が容易に実現できた．

An Implementation and Applications of A Self-applicable Partial Evaluator

Hiroshi Fujita and Koichi Furukawa

Institute for New Generation Computer Technology
Mita-kokusai Bldg. 21F
1-4-28, Mita, Minato-ku, Tokyo 108 Japan

It is well known that a partial evaluator specialized wrt an interpreter corresponds to a compiler, and that a partial evaluator specialized wrt another partial evaluator corresponds to a compiler generator. Such a partial evaluator that can process itself is said to be self-applicable.

This report presents an implementation of a self-applicable partial evaluator in Prolog, and its applications to compiler generation and compiler generator generation. Incremental compilation is also presented as an useful application.

These results are obtained by extending a simple self-interpreter of Prolog. Its self-application was realized easily because of its compactness with minimum functionality.

1．はじめに

部分計算はよく知られているとおり，プログラムの実行に必要な情報の一部が知られているとき，その情報だけで計算できるプログラム部分を実行し，残りの情報に関する計算だけを表すような特殊化されたプログラムを得ることである．

$$peval(Prog.Known.P_K) \quad \cdots\cdots①$$

「peval は『ProgをKnown について特殊化したものは P_K である』という関係を表す」と読む．

部分計算はソフトウエア一般に幅広く応用できるが，特にコンパイラ生成やコンパイラジェネレータ生成と関連して面白い可能性が理論的に示されている[1]．

特に，Prologプログラムの部分計算は，メタプログラミングにおいて有用な最適化技法となることが分かっている[2.3]．メタプログラミングでは，ある問題を解くプログラムを問題向きに書かれたプログラム部分（Prog）と，それを解釈するメタプログラム部分(Int) とに分けるため，作成，修正は容易になるが，実行効率が悪い．

そこで，部分計算を使うことが考えられる．　即ち，Int * Prog * Goal の計算のうちInt * Progの計算を部分的に行うことを考える．残るプログラムは，Progで Int を特殊化した Int_{Prog} である．

$$peval(Int.Prog.Int_{Prog}) \quad \cdots\cdots②$$

さらに，peval のプログラムが与えられ，Int がわかっていれば，peval * Int を部分計算できるであろう．

$$peval'(peval.Int.peval_{Int}) \quad \cdots\cdots③$$

その結果の $peval_{Int}$ は，

$$peval_{Int}(Prog.Int_{Prog}) \quad \cdots\cdots④$$

なる関係を与えるものとなっている．

さて，このInt に対応してコンパイラCom があると，

$$Com(Prog.Obj) \quad \cdots\cdots⑤$$

ここで④と⑤を見ると，Int_{Prog} とObj を対応させれば，$peval_{Int}$ とCom が対応するということができよう．

さらに，

$$peval''(peval'.peval.peval'_{peval}) \quad \cdots\cdots⑥$$

によって与えられる $peval'_{peval}$ は，

$$peval'_{peval}(Int.peval_{Int}) \quad \cdots\cdots⑦$$

なる関係を与えるものとなっている．$peval_{Int}$ がコンパイラに当るとすれば，$peval'_{peval}$ はコンパイラジェネレータに当ると言える．$peval'_{peval}$ を作ったPeval''はコンパイラジェネレータジェネレータと言えよう（図１参照）．

部分計算プログラムに求められる条件としては，

(1) 正しい（プログラムの意味を変えない）
(2) 広いクラスのプログラムに対し充分良い特殊化プログラムを生成できる（実用的）
(3) 部分計算中に人の介入を要しない（自動的）
(4) peval = peval' = peval''　（自己適用可能）

という項目が上げられよう．

(1) は当然であるが，[4.5] に従うものとして，本稿では議論しない．(2) や(3) の条件を満たすためには，可能な限り計算する一方，発散しないように適切な打切り制御を行わなければならない．そのためにはプログラムの特性について解析を必要とする．そうして強化された部分計算プログラムは複雑で大きなプログラムになってしまう．一方，(4) の目的を達成するためには，部分計算プログラムはできるだけコンパクトであることが望ましい．自己適用可能性を断念すれば，Peval は極力コンパクトなものを，Peval'は極力機能の高いものを使って $Peval'_{Peval}$ が得られるだろう．しかし，(4) が実現可能であるか否かを議論すること，また，可能な場合，実際に構成することは興味深いことである．

本稿は，自己適用可能なPrologの部分計算プログラムの実現について報告するものである．従って，部分計算プログラムに課せられる他の条件については，特に考慮しない．なお，Lispについては既に[6] の成果が知られている．また，[7] では，LispとPrologを組み合わせた面白い手法が用いられている．

2．Prologプログラムの部分計算

Prologプログラムの部分計算の基本は，(1) 既知情報がゴールの引数の具体化として与えられるとき，これをサブゴールの展開に伴なってトップダウンに伝播させること，(2) 既知情報がある述語の単位節で与えられる場合は，逆にボトムアップに伝播させること，(3) 具体化された節本体中のアトムで評価可能なものは評価し，展開が危険なもの，無意味なものはそのまま残すことである．こうして具体化された節の数は展開されたAND サブゴールに対する定義節の数の積に比例して生成される．このコード量の増大によるデメリットが，先行して計算した分によって低減した実行時計算量のメリットを上回るということが実際上の問題として生じ得る．これは，space-timeのトレードオフ問題であって，本稿では議論しない．

組込述語の計算

変数が未束縛のとき，全計算においては単一化によって任意に束縛することができるが，部分計算時においてはいつもそうできるわけではない．一般に部分計算時に未束縛でも全計算時には既に束縛されているかもしれないからである．下にDEC-10 Prolog の組込述語について部分計算手続きを与える．これらは，通常の引数の他に１つ余分の引数を持っており，束縛されているべき変数が未束縛の場合，計算を保留してそのゴールの呼び出し自身或いは簡単化されたゴールを返す．

```
is(X,Y,true):-ground(Y),!,call(X is Y).
is(X,Y,(X is Z)):-simplifyExpr(Y,Z).
<(X,Y,true):-ground(X<Y),!,call(X<Y).
```

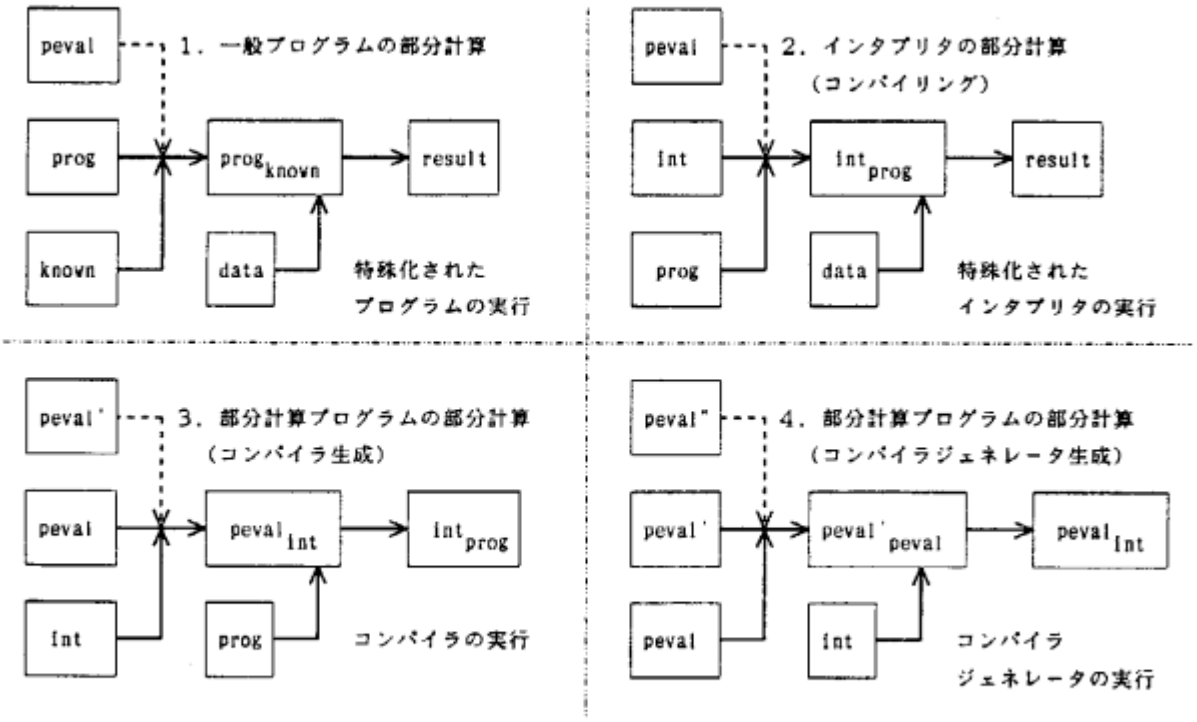

図１　部分計算とコンパイリング

```
<(X.Y.X<Y).
etc.
```

一般（再帰的）述語の展開

再帰的述語のゴールの展開については，全計算では再帰呼出しが有限回で終了するものでも，部分計算時には未束縛変数のために止まらない場合がありうる．しかし，少なくとも全計算が止まるものは部分計算も止めなければならない．特に，引き続く再帰呼出しの列が，ある有礎順序に従って降下していることが確かめられれば十分であろう．例えば，ゴールp(...Ti...) とヘッドユニファイ可能な節が

j: p(... Ti^{j0} ...) :- ...p(...Ti^{jk} ...)....

のように与えられたとき，まず，

Ti is-an-instance-of Ti^{j0}

であって，さらに，

Ti^{jk} is-a-proper-subterm-of Ti^{j0}

があるi で，全ての再帰呼出し(all j, all k)について成り立っていれば，展開を実行することにする．subterm 関係が有礎順序であることから，この展開はいずれ止まることが保証される．

（例）appendが，

```
append([H|X].Y.[H|Z]):-append(X.Y.Z).    ····①
append([].Y.Y).                           ····②
```

と普通に定義されているとき，append([1.2].Y.Z) の部分計算は上の条件を満たすので，①で２回展開された後，②で止まり，Z=[1.2|Y] なる解置換を与える．これは，実は，全計算される場合である．append(X.[].Z)の部分計算は，上の条件を満たさず，これ以上展開できない．同様にして，append([1.2|X].Y.Z) は，append(X.Y.W).Z=[1.2|W] まで，append(X.[].[1.2|Z])は，append(U.[].Z).X=[1.2|U]まで，それぞれ展開される．

ｏｐｅｎ述語の処理

open述語とは，定義が完了していない述語のことである．open述語を呼出すプログラムの部分計算については，将来追加される定義へのアクセスができるようにしておく必要がある．従って，open述語を含むプログラムを部分計算したものは，既に与えられている定義節によって展開されたものの他に，元の呼出しを残したものも含むものとする．このopen述語の扱いは，後で段階的コンパイリングを行うときにポイントとなる．

３．部分計算プログラムの自己適用

言語Lのプログラムのための部分計算プログラムが自己適用可能であるためには，当然のことながら，それ自体が言語Lの範囲内で表現されていなければならない．これは，部分計算プログラムに限らず，自己インタプリタについても同じことが言える．pure Prolog の場合，自己インタプリタは次のように定義される．

```
solve(true).
solve((A.B)):-solve(A).solve(B).
solve(A):-clause(A.B).solve(B).
```

solve(A)をDEC-10 Prolog のスタンダードな実行系で実行すると，A を直接実行したときと同じ結果を得る．ただし，clause(A,B) は，システムにロードされた節集合の中から A とユニファイ可能なヘッドを持つ節の本体をB に返す組込述語である．

この自己インタプリタを次のように拡張してみよう．

```
psolve(true,true).
psolve((A,B),(RA,RB)):-psolve(A,RA),psolve(B,RB).
psolve(A,R):-clause(A,B),psolve(B,R).
psolve(A,A):-residual(A).
```

4番目の節において，residual(A) を満たすようなゴール A は，何らかの理由（実行に必要な情報が不十分であるなど）で解くことを差止められる．差止められたゴールを剰余ゴールと呼ぶ．psolveの第１引数にゴールA を与えると，第２引数に，剰余ゴール（のAND 木）R が返されることになる．こうして，A のサブゴールから生じた剰余ゴールを集めて，R とし，A:-Rなる節を導くことができる．これを，A の剰余節と呼ぶ．clauseで複数の選択がある場合，剰余節もそれに応じて複数生じうる．

特に，A のためのプログラムが定義済みで，A が全計算可能なゴールの場合は，このpsolveの実行は成功して，R にtrueのみからなるAND 木を得るか，失敗するか，ループするかであり，各々，A の直接実行が成功するか，失敗するか，ループするかに対応する．即ち，このプログラムを用いて自己インタプリタが，

```
solve(A):-psolve(A,R),allTrue(R).
```

と定義できる．ただし，allTrue(R)は，R がtrueだけから成るAND 木であるときに成功するものとする．

実際には，以下で使われる部分計算プログラムは次のように定義される．

```
psolve(A,R):-psolvePrim(A,R).                    ....(p1)
psolve((A,B),U-W):-                              ....(p2)
  psolve1(A,U-V),psolve1(B,V-W).
psolve(A,R):-cl(A,B),psolve1(B,R).               ....(p3)
psolve(A,[A|Z]-Z):-open(A).                      ....(p4)
psolve1(A,R):-expandable(A),psolve(A,R).
psolve1(A,[A|Z]-Z):-residual(A).
psolveAll(A):-bagof((A:-R),psolve(A,R),NewCls),
  define(NewCls).
```

(p1): psolveは，ゴールA が組込述語なら，組込述語専用の部分計算プログラムを呼ぶ．(p2): AND ゴールなら，それぞれのアトム毎に剰余ゴールを求めてそれらを接合したものを元の剰余ゴールとする．(p3): ゴールA がユーザ定義されているなら，A を定義節本体に展開して部分計算を行う（定義によるunfolding ）．(p4): ゴールA が未定義述語なら，A は剰余ゴールとされる．ここで，剰余ゴールのconjunction は差分リストで表現されている．

psolve1 は，ゴールA の展開可能性を判定してpsolveに委ねるか，その場で剰余ゴールとするか決定する．

psolveAll は，剰余ゴールの全解を求め，それぞれから１本ずつ剰余節を定義する．

psolvePrim,cl,open,expandable,residualは，組込述語と同じに見なされ，疑組込述語と呼ばれる．これら疑組込述語の実行時（及び部分計算時）の解釈は，一般の組込述語に対する部分計算時の解釈といっしょに，psolvePrimプログラムの中で定義済みとしておくことができる．そこでは，ユーザには解放されていない別の組込述語が使われているかもしれないが，それらの隠された組込述語が，部分計算の結果剰余ゴールに現れることはない．即ち，（疑）組込述語の実体は，ユーザプログラム（及び，psolve）からは完全にブラックボックスである．また，psolveAll は，部分計算プログラムpsolveをトップレベルから起動するマクロコマンドと考える．従って，bagof,defineも，あくまでコマンドprimitive であり，後でpsolveの部分計算を行う場合もプログラムの一部とは見なさないことにする．

さて，ゴールpsolve(A...)の部分計算を考えてみる．ゴールA の部分計算で剰余ゴールR1,R2,....,Rnが得られるならば，psolve(A,R) の部分計算は，psolve(R1,...)のconjunction を与えるだろう．即ち，

```
psolve(psolve(A,U1-W)),
  [psolve(R1,U1-U2),psolve(R2,U2-U3),...
   psolve(Rn,Un-W)|Z]-Z)
```

実際，psolvePrimについては，このような結果を想定してブラックボックス内で定義されている．

```
psolvePrim(psolvePrim(A,R),
           [psolvePrim(A,R)|Z]-Z):-var(A),!.
psolvePrim(psolvePrim(A,Q),R):-
  psolvePrim(A,P),psolvePr(P,Q,R).
psolvePr(A-C,W-W,Z-Z):-A==C,!.
psolvePr([A|B]-C,U-W,
         [psolvePrim(A,U-V)|Y]-Z):-!,
  psolvePr(B-C,V-W,Y-Z).
```

psolveについては，psolve(psolve...) の対処について陽には定義しない．代わりに，psolveの場合，適当な情報を与えることによって剰余ゴールの形を制御する．一般に，A が剰余ゴールとなるような場合，即ち，residual(A) であるとき，A の部分計算もA の部分計算の部分計算も差止めるべきであるということができる．

（疑）組込述語についてはブラックボックスとしたが，実際にはその中身がこの部分計算プログラムpsolveの働きの多くを担っているので，説明しておく．

psolvePrim自体，及びDEC-10 Prolog 組込述語についてはあらまし(is,<)を既に示した．次に，疑組込述語の実体を示す．

clとopen

```
psolvePrim(cl(A.B).[cl(A.B)|Z]-Z):-var(A).!.
psolvePrim(cl(A.B).Z-Z):-cl(A.B).
psolvePrim(cl(A.B).[cl(A.B)|Z]-Z):- open(A).
open(A):-$open(B).match(A.B).
  not(($close(C).match(A.C))).!.
psolvePrim(open(A).[open(A)|Z]-Z):- var(A).!.
```

clは，ユーザが導入した述語の定義節を与えるもので，前もってシステムにロードされているものとする．clの部分計算では，ゴールA の定義節がその時点で与えられていればそれを返す．psolveAll がbagof によって全解を探索するとき，clはA の全ての定義節を返すことになる．さらに，A がopenである場合には，clゴール自身が剰余となる．cl を剰余節に残すことによって，将来A の定義節が追加されたとき，それを呼出すことができるわけである．

$open(A).$close(A)は，ゴールパタンA の定義の未了／完了を指定する制御情報として，ユーザが与える．例えば，

```
$open(int(¥.¥)).
$close(int(true.¥)).
```

ただし，$close(A')は，$open(A)の制御情報が先行しているときに，A の具体化パタンA'について定義の完了を指示する目的（openの制限）にのみ用いられる．

expandableとresidual

```
expandable(A):-not(suspended(A)).
psolvePrim(expandable(A).[expandable(A)|Z]-Z):-
  var(A).!.
psolvePrim(expandable(A).Z-Z):-expandable(A).
residual(A):-suspended(A).
psolvePrim(residual(A).[residual(A)|Z]-Z):-
  var(A).!.
psolvePrim(residual(A).Z-Z):-residual(A).
suspended(A):-$suspend(A).
```

expandableはresidualの否定として定義でき，いささか冗長な述語であるが，これは，psolve1 を定義する際に，if-then-else（あるいはcut ）を避けたいからである．

$suspend(A) は，A の展開を差止めるための制御情報として，ユーザが与える．例えば，

```
$suspend(append(X.¥.Z)):-var(X).var(Z).
```

4．コンパイラ生成

部分計算プログラムpsolveをインタプリタについて特殊化して，コンパイラを作る．

次のようなインタプリタを考える．

```
int(true.[100]).
int((A.B).Z):-int(A.X).int(B.Y).append(X.Y.Z).
int(not(A).[CF]):-int(A.[C]).C<20.CF is 100-C.
int(A.[CF]):-rule(A.B.CF1).int(B.S).cf(CF1.S.CF).
cf(X.Y.Z):-product(Y.100.W).Z is (X*W)/100.
product([A|X].Y.Z):-W is A*Y/100.product(X.W.Z).
product([].Y.Y).
append([A|X].Y.[A|Z]):-append(X.Y.Z).
append([].Y.Y).
```

これは，確信度付きの小さな推論エンジンである．

制御情報は次のように与える．

```
$suspend(psolve1(A.¥)):-var(A).
$suspend(psolve1(A.¥)):-residual(A).
$suspend(int(A.¥)):-var(A).
$suspend(product(A.¥.¥)):-var(A).
$suspend(append(A.¥.C)):-var(A).var(C).
$open(rule(¥.¥.¥)).
```

これらをロードした上で，次のコマンドを与える．

```
?-psolveAll(psolve(int(¥.¥).¥).
```

コンパイラ（特殊化された部分計算プログラム）は次のように得られる．

```
psolve(int(true.[100]).A-A).                    ····(p5)
psolve(int((A.B).C).D-E):-                      ····(p6)
  psolve1(int(A.F).D-G).
  psolve1(int(B.H).G-I).
  psolve1(append(F.H.C).I-E).
psolve(int(not(A).[B]).C-D):-                   ····(p7)
  psolve1(int(A.[E]).C-F).
  psolvePrim(E<20.F-G).
  psolvePrim(B is 100-E.G-D).
psolve(int(A.[B]).C-D):-                        ····(p8)
  cl(rule(A.E.F).G).
  psolve1(G.C-H).
  psolve1(int(E.I).H-J).
  psolve1(product(I.100.K).J-L).
  psolvePrim(B is F*K /100.L-D).
```

この４本の節は，int の４本の節にそれぞれ対応している．(p5)は，int(true.¥) の部分計算が剰余ゴールを残さないことを示している．(p6)はint((A.B).¥)の部分計算による剰余ゴールが，int(A.F).int(B.H).append(F.H.C) をそれぞれ部分計算して得られる剰余ゴールの接合として得られることを示している．(p7).(p8) も同様である．

ruleが未定義述語であったために，(p8)において，cl(rule(A.....).¥) が剰余ゴールとなっていることに注意されたい．この箇所が，後で段階的コンパイリングに効いてくる．int の他に，再帰的述語product.appendのためのコンパイラ断片も作られる．

なお，非再帰的述語cfについては展開されてしまい，コンパイラ断片にはもはや残らない．

さて，上で与えた制御情報のうち，

```
$suspend(psolve1(A.¥)):-residual(A).
```

を省くとどうなるであろうか．

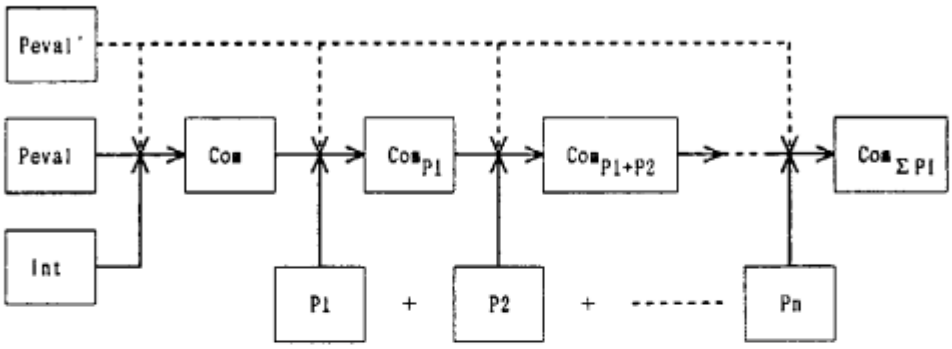

図２　段階的コンパイリング

```
psolve(int(true,[100]),A-A).
psolve(int((A,B),C),
   [int(A,D),int(B,E),append(D,E,C)|F]-P).
   ・・・
```

２番目の節は，int(A,¥)を部分計算すると常にint(A,D),int(B,E),append(D,E,C)の３つを剰余ゴールとして返すことを示す．A が具体化されてint(A,¥)がさらに展開され得るようになっても，この特殊化された部分計算プログラムpsolveは展開を全く行なわない．これは望ましい結果ではない．特殊化されたpsolveは，それが実行されるときに最大限の部分計算を行なうようにできていることが期待されているのである．

この例でわかるように，部分計算をどこまで行ない，どこで止めるべきかを決めることは，自己適用の場合はさらに微妙な問題となる．

５．段階的コンパイリング

前節で得られたコンパイラにソースプログラム断片を段階的に与えてコンパイラを特殊化する（図２参照）．

５．１．メインプログラムの取り込み

前節のint の入力プログラムにあたるのは，ruleである．今，次のようなプログラムを与える．

```
rule(shouldTake(Person,Drug),(
  complainsOf(Person,Symptom),
  suppress(Drug,Symptom),
  not(unsuitable(Drug,Person))),70).
rule(unsuitable(Drug,Person),(
  aggravates(Drug,Condition),
  suffersFrom(Person,Condition)),80).
```

制御情報は次のように与える．

```
$suspend(int(A,¥)):-inst(A,complainsOf(¥,¥));
                    inst(A,suffersFrom(¥,¥)).
:-abolish($open,1).
$open(rule(suppresses(¥,¥),¥,¥)).
$open(rule(aggravates(¥,¥),¥,¥)).
```

int(complainsOf...),int(suffersFrom...) は，このプログラムの実行時に与えられる情報であるから，今はsuspend しておく．ruleは最も一般的な形rule(¥,¥,¥) ではもはやopenではないが，具体化された形のrule(suppresses...),rule(aggravates...) は依然openであるとしている．次のコマンドでコンパイラを特殊化する．

```
?-psolveAll(psolve(int(shouldTake(¥,¥),¥)),¥)).
```

ここでは，shouldTakeが主要な問い合わせであるとする．結果は次のようになる．

```
psolve(int(shouldTake(A,B),[C]),D-E):-     ・・・・(p9)
  psolve1(int(complainsOf(A,F),C),D-H),
  c1(rule(suppresses(B,F),I,J),K),
  psolve1(K,H-L),
  psolve1(int(I,M),L-N),
  psolve1(product(M,100,O),N-P),
  psolvePrim(Q is J*O/100),N-P),
  c1(rule(aggravates(B,S),T,U),V),
  psolve1(V,R-¥),
  psolve1(int(T,X),¥-Y),
  psolve1(product(X,100,Z),Y-A1),
  psolvePrim(B1 is U*Z/100,A1-C1),
  psolve1(int(suffersFrom(A,S),D1),C1-E1),
  psolve1(product(D1,B1,F1),E1-G1),
  psolvePrim(H1 is 80*F1/100,G1-I1),
  psolvePrim(H1<20,I1-J1),
  psolvePrim(K1 is 100-H1,J1-L1),
  psolve1(append(G,[Q,K1],M1),L1-N1),
  psolve1(product(M1,100,O1),N1-P1),
  psolvePrim(C is 70*O1/100,P1-E).
```

rule(shouldTake...),rule(unsuitable...) はいずれも，１本のpsolve節のなかに展開されてしまった．

５．２．プログラム断片の追加①

次のプログラム断片を与えて，上のpsolveをさらに特殊化する．

```
rule(suppresses(aspirin,pain),true,60).
rule(aggravates(aspirin,pepticUlcer),true,70).
```

制御情報は次のように与える．

```
$close(rule(suppresses(aspirin,¥),¥,¥)).
$close(rule(aggravates(aspirin,¥),¥,¥)).
```

特殊化コマンドは，

```
?-psolveAll(psolve(int(shouldTake(¥,¥),¥),¥)).
```

結果は，

```
psolve(int(shouldTake(A,aspirin),[B]),C-D):- (p10)
  psolvel(int(complainsOf(A,pain),E),C-F),
  psolvel(int(suffersFrom(A,pepticUlcer),G),F-H),
  psolvel(product(G,70,1),H-J),
  psolvePrim(K is 80*1/100,J-L),
  psolvePrim(K<20,L-M),
  psolvePrim(N is 100-K,M-O),
  psolvel(append(E,[60,N],P),O-Q),
  psolvel(product(P,100,R),Q-S),
  psolvePrim(B is 70*R/100,S-D).
```

これは，(p9)節をaspirin の場合に特殊化したものとなっている．

5．3．プログラム断片の追加②

新しい薬に関する事実を追加する．

```
rule(suppresses(lomotil,diarrhoea),true,65).
rule(aggravates(lomotil,impairedLiverFunction),
     true,70).
```

aspirin の場合と同様にして，(p9)節をlomotil の場合に特殊化した結果が得られる．

このプログラム断片の追加手続きは，さらに新しい薬に関する事実が得られる度に，それまでに特殊化されたコンパイラだけをベースとして行なうことができる．

5．4．コンパイラの実行

特殊化されたコンパイラは，任意の段階でコンパイラとして走らせることができる．

例えば，これまでに得たaspirin とlomotil の知識だけで実行可能なプログラムを得たいとしよう．

制御情報は，

```
:-abolish($open,1).
```

コンパイラ実行は，

```
?-psolveAll(int(shouldTake(¥,¥),¥)).
```

結果は，

```
int(shouldTake(A,aspirin),[B]):-
  int(complainsOf(A,pain),E),
  int(suffersFrom(A,pepticUlcer),G),
  product(G,70,1),
  K is 80*1/100, K<20, N is 100-K,
  append(E,[60,N],P),
  product(P,100,R),
  B is 70*R/100.
int(shouldTake(A,lomotil),[B]):-
  int(complainsOf(A,diarrhoea),E),
  int(suffersFrom(A,impairedLiverFunction),G),
  product(G,70,1),
  K is 80*1/100, K<20, N is 100-K,
  append(E,[65,N],P),
  product(P,100,R),
  B is 70*R/100.
```

5．5．議論

何故オブジェクトコードの段階的特殊化でなく，コンパイラの段階的特殊化なのか？　勿論，前者も可能である．しかし，後者の方が効率が良い場合がある．前者では各段階で，$int_{\Sigma P_j}$を得る．これに新しくプログラム断片Pnを追加するとき，部分計算プログラムは $int_{\Sigma P_j}$を改めて入力し，再計算する必要がある．他方，後者では各段階で，$Com_{\Sigma P_j}$を得る．これは実は，$peval_{int\Sigma P_j}$なる特殊化された部分計算プログラムである．これは前者の場合の部分計算プログラムと違って，既に $int_{\Sigma P_j}$を入力，計算済みである．従って，新しく追加されたPnに関する計算だけを行えばよい．

6．コンパイラジェネレータ生成

次のコマンドを与える．

```
?-psolveAll(psolve(psolve(¥,¥),¥)).
```

結果は，

```
psolve(psolve(A,B),C):-
  psolvePrim(psolvePrim(A,B),C).
psolve(psolve((A,B),C-D),E-F):-
  psolvel(psolvel(A,C-G),E-H),
  psolvel(psolvel(B,G-D),H-F).
psolve(psolve(A,B),C-D):-
  psolvePrim(cl(A,E),C-F),
  psolvel(psolvel(E,B),F-D).
psolve(psolve(A,[A|B]-B),C):-
  psolvePrim(open(A),C).
```

この特殊化されたpsolveは，psolveの定義節において各アトムにpsolve（または，psolvel,psolvePrim）を被せた形になっている．これで，psolveの動作が模倣できるのである．実際，

```
psolve(A,R):-psolve(psolve(A,R),U-¥),U=¥.
```

であって，psolve(A,R) の部分計算が上のpsolveで剰余なしに計算された場合は，psolve(A,R) は実は全計算できたのである．

この観察から，psolve(psolve...) のプログラムがいかにも冗長であるかのような印象を受けるが，確かに特殊化されたpsolveはもとの一般形とは別物であって，別次元のpsolveのためにカスタマイズされたnon-trivial なプログ

ラムである．このことを自己インタプリタsolve の場合と比較すると面白い．

```
?-psolveAll(solve(solve(A))).
solve(solve(true)).
solve(solve((A,B))):-
  solve(solve(A)),solve(solve(B)).
solve(solve(A)):-
  solve(clause(A,B)),solve(solve(B)).
```

ここで，組み込み述語clauseについて，

```
solve(clause(A,B)):-clause(A,B).
```

と仮定した上，

```
solve2(A) / solve(solve(A))
```

の置換えを行うと，

```
solve2(true).
solve2((A,B)):-solve2(A),solve2(B).
solve2(A):-clause(A,B),solve2(B).
```

を得る．これは，もとのsolve のプログラムと同型である．これに習って，上で得たpsolveにおいて，

```
psolve2((A,B),C) / psolve(psolve(A,B),C)
psolvePrim2((A,B),C) /
  psolvePrim(psolvePrim(A,B),C)
psolve1-2((A,B),C) / psolve1(psolve1(A,B),C)
cl2((A,B),C) / psolvePrim(cl(A,B),C)
open2(A,C) / psolvePrim(open(A),C)
```

の置換えを行うと，

```
psolve2((A,B),C):-psolvePrim2((A,B),C).
psolve2(((A,B),C-D),E-F):-
  psolve1-2((A,C-G),E-H),psolve1-2((B,G-D),H-F).
psolve2((A,B),C-D):-
  cl2((A,E),C-F),psolve1-2((E,B),F-D).
psolve2((A,[A|B]-B),C):-open2(A,C).
```

を得るが，これは，勿論もとのpsolveとは似て非なるものである．結局，solve(solve...) は，解くことを解く冗長な模倣計算であるが，psolve(psolve...) は，異なる次元の部分計算を部分計算する有為な特別の計算である．ともかく，こうして得られたコンパイラジェネレータは，実際，4節のコンパイラ生成のときに用いた一般psolveの代りに用いることができる．

7．おわりに

自己適用可能なPrologの部分計算プログラムを実現した．機能を最小限に止どめたおかげで自己適用は容易であった．この部分計算プログラムを用いて，コンパイラ生成及びコンパイラジェネレータ生成が実現された．これは，著者の知る限り，論理プログラミング言語においては最初の結果であると考える．

さらに，この部分計算プログラムを用いて段階的コンパイリングをも実現した．

これらの結果は，pure Prolog に限られており，効率の点でも満足のいくものであるとは言い難い．cut や not，or，bagof 等を含むfull Prolog での実現が一つの今後の課題である．もう一つの重要な方向として，並列論理型言語（ＧＨＣ）における部分計算についても同様に研究が進められつつある．

参考文献

[1] Futamura,Y., Partial Evaluation of Computation Process - An Approach to a Compiler-compiler, Systems, Computers, Controls 2, No.5(1971)45-50. 二村，部分計算，プログラム変換第４章，知識情報処理シリーズ７，共立出版，1987.

[2] Takeuchi,A. and K.Furukawa, Partial Evaluation of Prolog Programs and its Application to Meta Programming, in Kugler,H.J.(ed.): Information Processing 86, Dublin,Ireland 415-420, North-Holland 1986. 竹内，メタ・プログラミングと部分計算，プログラム変換第５章，知識情報処理シリーズ７，共立出版，1987

[3] Safra,S. and E.Shapiro, Meta Interpreters for Real, in Kugler,H.J.(ed.): Information Processing 86, Dublin, Ireland 271-278, North-Holland, 1986.

[4] Komorowski,H.J., Partial Evaluation as a Means for Inferencing Data Structures in an Applicative Languge: A Theory and Implementation in the Case of Prolog,Ninth ACM Symposium on Principles of Programming Languages,New Mexico(1982)255-267

[5] Tamaki,H. and T.Sato, Unfold/fold Transfomation of Logic Programs, Proc. 2nd International Logic Programming Conference, pp.127-138,Uppsala,1984. 玉木，論理型言語におけるプログラム変換，プログラム変換第３章，知識情報処理シリーズ７，共立出版，1987.

[6] Sestoft,P., The Structure of a Self-applicable Partial Evaluator, in H.Ganzinger and N.D.Jones (eds.): Programs as Data Objects, Copenhagen, Denmark, 1985, Lecture Notes in Computer Science 217(1986) 236-256, Springer-Verlag.

[7] Kahn,K.M. and M.Carlson, The Compilation of Prolog Programs without the Use of a Prolog Compiler, Proceedings of the International Conference on Fifth Generation Computer Systems, Tokyo Japan, 1984, 348-355.