ICOT Technical Memorandum: TM-0250

TM-0250

Prolog プログラムの
部分計算の自動化

藤田　博，古川康一

January, 1987

ICOT

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

(03) 456-3191～5
Telex ICOT J32964

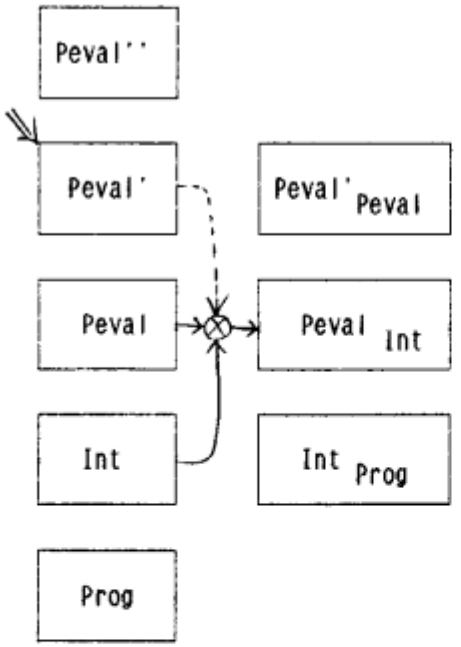
Institute for New Generation Computer Technology

Ｐｒｏｌｏｇプログラムの部分計算の自動化

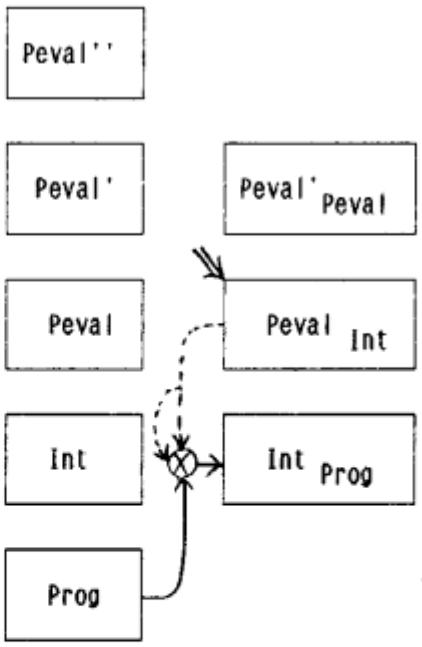
１．はじめに

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．メタプログラミングでは，ある問題を解くプログラムをその問題向きに書かれたプログラム部分（Prog）と，それを解釈するプログラム部分(Int）とに分けて作成し，解釈部にゴールを与えてこれを解く．これだと問題プログラムの実行が解釈部を経由する分効率が悪い．Progが固定されていれば，ゴールが与えられる前にInt の実行を部分的に行うことができるかもしれない．残った計算を行うプログラム部分は，ProgでInt を特殊化したものである．この部分計算を行う者(Peval) はコンパイラに相当し，残ったプログラム（ Int_{Prog} ）はコンパイルドコードに相当する．ここまでは理屈も明快で実現も比較的容易である．

次に，Int は固定されているがProgが未定のときでも，Int について部分計算 (Peval) を部分的に実行することができるかもしれない．残った計算のコード(Peval $_{Int}$) はProgが与えられると残りの部分計算を実行して上と同じコンパイル結果(Int $_{Prog}$) を得る．即ちコンパイラである．従って，部分計算の部分計算を行う者(Peval')はコンパイラジェネレータに当る．部分計算プログラム自体を部分計算するこの試みは，理屈が明快なのとは裏腹に実現は思いの他難しいものである．ただ，食う側と食われる側の二つの部分計算プログラムは異なっていてもよい．しかし，食う側の方は食われる側より強力でなければ



(1) 部分計算時



(2) 残余プログラム実行時

ならないだろう．極端な場合，コンパイラジェネレータ側の部分計算は全く人手であってもよい．また，コンパイラとしての部分計算プログラムは，広いクラスのPrologプログラムに対し充分良いコンパイルドコードを生成できる能力を備えている必要がある．加えてこの部分計算プログラムを実行中に人の介入を要しないことが望ましい．

さて，部分計算プログラムが固定されると，その入力となるプログラム(Int) が未定でもその部分計算の部分的実行自体を部分的に実行することができるかもしれない．残った計算のコード(Peval' $_{Peval}$) は，Int が与えられると，残りの部分計算の部分的実行を完行して上と同じコンパイラジェネレーション結果(Peval$_{Int}$) を得る．従って，部分計算の部分計算の部分計算を行う者(Peval'') はコンパイラジェネレータジェネレータに相当する．我々の目標は全ての階層の部分計算プログラム(Peval, Peval', Peval'')を同一のもので実現することである．

我々はまず第一段階として，自動化されたPrologの充分に強力な部分計算プログラムを開発することを目標とした．自動化に当たっての最も大きな問題は，再帰的に定義されている述語の展開をどこまで行うかということである．

2．Prologプログラムの部分計算

Prologの計算の基本はユニフィケーションである．Pure Prolog のユニフィケーションでは変数のバインディングの場所と時期が限定されない．この点，手続き型言語や関数型言語（及び並列型論理言語）と大きく異なり，部分計算においてはメリットとなっていると思われる．既知情報がゴールの引数として与えられる場合，この情報は通常のPrologインタープリタが行うように，ホーン節の左辺から右辺へトップダウンにサブゴールの展開に伴なって伝播される．既知情報がある述語の単位節で与えられる場合は，逆にホーン節の右辺から左辺へボトムアップに伝播される．こうして，ヘッドとボディの双方からinstantiate されたホーン節は，既知情報で特殊化されたプログラム部分であると見なすことができる．

展開中のゴールが単位節とのみユニファイ可能なとき，或いは評価可能なシステム述語である場合は展開後のゴールが消去され，バインディングのみが残って終わるので問題なく実行できる．ただし，一般に兄弟ゴールの複数のalternative な単位節による展開により，展開後の節の数は乗算的に脹らむ．このコード量の増大によるデメリットが，展開による実行時計算量の低減という本来期待していた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．しかし，本稿ではこうしたトレードオフ問題については扱わない．

大域的に非再帰的な述語のゴールについても有限に展開することができる．問題なのは再帰的述語のゴールの展開である．このような展開は一般に，未知情報が定まったときの全計算では有限に終了するものでも部分計算時に有限に終了するとは限らない．

部分計算は以上の各場合のどれに当たるかにより，ゴールを展開すべきか否かをチェックしつつ進める必要がある．[Takeuchi 86] の部分計算プログラムでは，このチェックに

必要な制御情報をユーザが予め与えるか部分計算時に会話的に与えるようにしている．この方法では展開を任意に進めたり止めたりでき，上のトレードオフ問題にもヒューリスティックな解答を見出だす手段となりうるが，再帰的ゴールの展開を適切なところで止めるための制御法については，プログラムを前もって解析することにより自動的に導かれると思われる．

さて，部分計算には何もしないものから全計算のあらゆる場合を尽くすものまで幅がある．しかしトリビアルでない部分計算というものの何らかの目安が与えられるはずである．Prologのトリビアルでない部分計算とはどのようなものかを示唆する例として，Prolog in Prolog インタプリタの自己適用を考える．

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

?- solve(solve(A)). として部分計算
clause(A,B) は上の３行の節をあたえる

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

右辺のsolve を展開

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

solve(clause(A,B)) :- clause(A,B). を使って展開
solve2(A) / solve(solve(A)) の置換を行なう

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

solve2(A)=solve(A)

上のsolve はpure Prolog の任意のゴールを解釈実行するPrologプログラムである．これをsolve ゴールを解くものに特殊化する．そうするとsolve2という述語名（及び一般には変数名）を除いて字句的にも全く等しいプログラムを得る．このsolve のidempotentな特性はself-interpreterであるというプログラムの特殊性によるものだが，字句的に等しいものを導けるのは部分計算の能力に起因する．

３．自動部分計算器の構成

自動化された部分計算プログラムは次のように構成されている．

（１）プログラムの静的な解析を行なうフェーズ

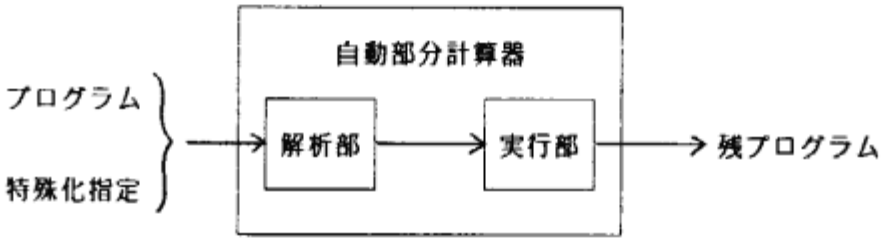
（２）部分計算を実行するフェーズ

（２－１）消去可能な述語の展開

（２－２）評価可能な組み込み述語の計算

（２－３）一般述語の展開

（１）については次節で扱うこととし，本節では（２）について述べる．



３．１　消去可能な述語の展開

単位節のみによって定義されたデータ的な述語，並びに大域的に非再帰的に定義された中間手続き的な述語は，プログラム中における全ての呼び出しを一回展開することによりプログラムから消去することができる．ただし，特殊化指定されたトップレベル述語は消去しない．

３．２　評価可能な組み込み述語の計算

変数がunbound な状態であるとき，全計算においてはユニフィケーションの過程で任意に値をバインド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unbound な変数は全計算時には既にバインドされているかもしれないからである．実際このことが効いてくるのは例えばnot の内側のユニフィケーションである．そこで，部分計算される節の中の変数でそのゴール以前のゴールに出現するものにはannotation（？）を付加することにする．そのゴールで初めて出現する変数はfreeで

あり，任意にバインドできる．

次にannotationつきの変数を含む項に対しlenient なユニフィケーションpunifyを定義する．このユニフィケーションは，特に

（ａ）ground term 同志の場合

（ｂ）不完全項でも項形が不一致のとき

計算し尽くされて全計算のユニフィケーションと同じ結果を与える．

```
punify(XU,Y,[XU=Y | Z]-Z) :- nonvar(XU),XU=X?,!.
punify(X,YU,[X=YU | Z]-Z) :- nonvar(YU),YU=Y?,!.
punify(X,Y,Z-Z) :- (var(X) ; var(Y)),!,X=Y.
punify(X,Y,Z-Z) :- (atomic(X) ; atomic(Y)),!,X=Y.
punify(X,Y,R) :- X=..[F!A], Y=..[F!B], punify¥list(A,B,R).

punify¥list([X | A], [Y | B],R-Z) :-
      punify(X,Y,R-M),!,punify¥list(A,B,M-Z).
punify¥list([],[],Z-Z).
```

punifyの第３引数には残ユニフィケーションの（差分）リストが与えられる．これが空の場合は，全計算のユニフィケーションでも成功して同じバインディングが得られる．また，punifyが失敗したときには，全計算のユニフィケーションでも失敗する．なお，not の内側でないユニフィケーションはannotationによらず全計算される．

次に，その他のいくつかのシステム述語について部分計算手続きを与える．これらは一般に変数がunbound なために失敗（成功）するような場合，計算を保留するように定義されており，通常の引数の他に残計算の（差分）リストを示す引数を持っている．

```
is(X,Y,[X is Y | Z]-Z) :- unknown¥expr(Y),!.
is(X,Y,[X=W | Z]-Z) :- variable(X),!,W is Y.
is(X,Y,Z-Z) :- X is Y.

'<'(X,Y,[X<Y | Z]-Z) :- variable(X) ; variable(Y),!.
'<'(X,Y,Z-Z) :- X<Y.

variable(X) :- var(X) ; unknown¥var(X).
unknown¥var(X) :- nonvar(X), X=¥?.
unknown¥expr(X) :- variable(X).
unknown¥expr(-X) :- unknown¥expr(X).
```

```
unknown¥expr(X+Y) :- unknown¥expr(X) ; unknown¥expr(Y).
unknown¥expr(X-Y) :- unknown¥expr(X) ; unknown¥expr(Y).
unknown¥expr(X*Y) :- unknown¥expr(X) ; unknown¥expr(Y).
unknown¥expr(X/Y) :- unknown¥expr(X) ; unknown¥expr(Y).
```

３．３　一般述語の展開

再帰的に定義された述語呼び出しの展開は次のように行なう．

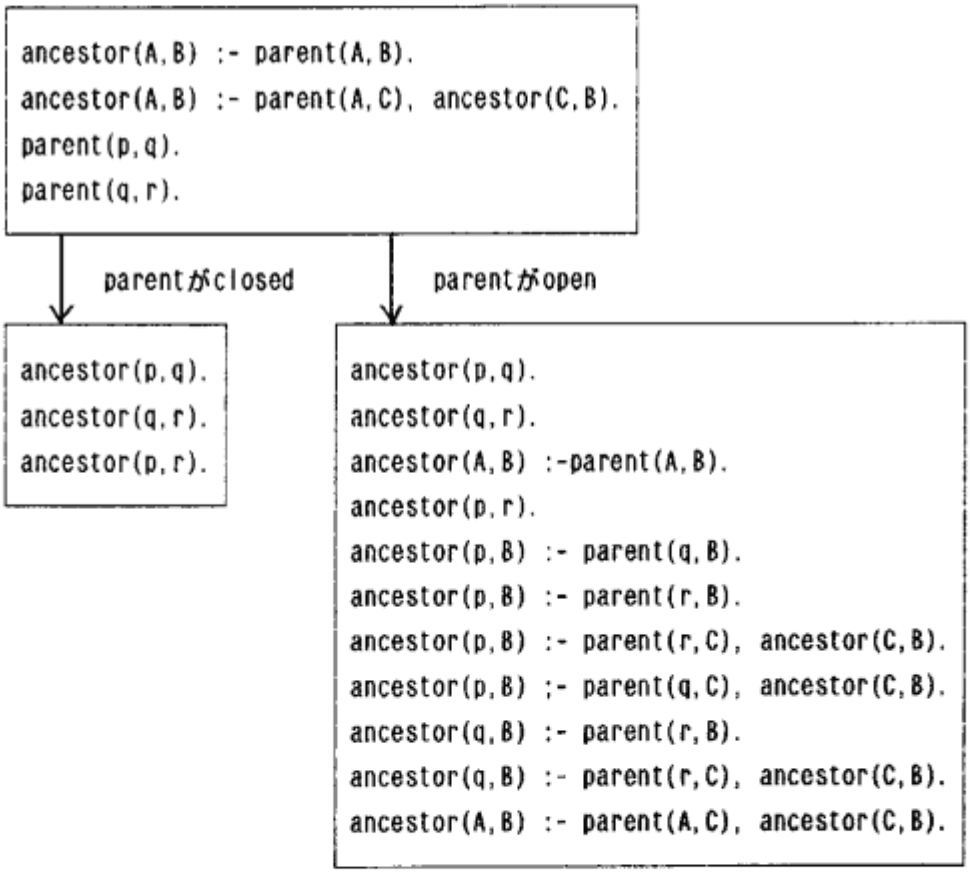
（１）openゴールの指定とmatch した呼び出しは展開しない．

（２）ヘッドユニファイ可能な定義節がない場合，呼び出し側の節は失敗節となるのでこれを捨てる．

（３）大域非再帰的な定義節のみとヘッドユニファイ可能な場合，展開する．

（４）ヘッドユニファイ可能な節が大域再帰的な節を含む場合，全ての再帰呼び出しがあるwell-founded-ordering で降下していることが確かめられる時に限り展開する．

（１）のopen述語とは，閉じていない（定義が完了していない）述語のことである．open述語を呼び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を使う余地を



残す必要がある．こうして，閉じていない述語を含むプログラムを部分計算したものは，特殊化された節の他に元の節も残したものになっている．

４．部分計算のための事前解析

４．１　述語呼び出しの依存関係

述語が（相互）再帰的に呼ばれるかどうかを知るために述語の呼び出し関係を調べる．最も粗い近似は，述語名だけの呼び出し関係を定義節から抽出し，その推移的閉包を計算して得られる．さらに近似度を上げて項レベルまで見たゴールパタンについても同様の解析ができる．

こうして，呼び出し関係が求められると，大域的に非再帰的な述語，相互再帰的な述語の組などの情報が得られる．大域的に非再帰的で，ボディにcut を含まず，特殊化指定（トップレベルで呼ばれる可能性）のない述語は消去可能とマークされる．従って，単位節のみから定義されたデータ的な述語は消去可能となる．再帰的な述語の展開可能性の決定には，部分計算のループを避けるために述語呼び出しの関係の上にさらにある部分順序を見出だすことが必要となる．

例えば，次のプログラムで，solve は再帰的に定義されている．

```
solve(true,[100]).
solve((A,B),Z) :- solve(A,X), solve(B,Y), append(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).
```

solve(true,[100]) は単位節で他のゴールを呼ばない．他の３つのヘッドのパタンは，

```
solve((#1,#2),#3), solve(not(#1),[#2]), solve(#1,[#2])
```

であるが，これらは全てボディのゴールパタン

```
solve(#1,#2), solve(#1,[#2])
```

を呼ぶ可能性がある．一般にヘッドパタンのゴールは展開の末やがてはボディパタンの再帰呼び出しに達する可能性がある．そこで，ヘッドパタンに属し，ボディパタンに属さないパタンのinstanceに当っている限り展開を続け，ボディパタンのinstanceになれば展開を止めるという部分計算の戦略が考えられる．即ち，ヘッドパタンとボディパタンの間に計算進度に関する部分順序を考えるわけである．しかし，再帰呼び出しによる部分計算のループを防止するためには部分順序はさらに有礎的(well-founded)であることが必要である．以下でさらに厳密な条件を与える．

４．２　再帰述語呼び出しのwell-founded ordering

全計算では再帰呼び出しがやがて終端に達して止まるものでも，部分計算時にunbound

(unknown) な変数のために終端に達せず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どこかで止めなければならない．そのためには引き続く再帰呼び出しの列があるwell-founded ordering に従って降下していることを確かめることができれば良い．

前節の条件（４）は無限の展開を防ぐための充分条件を与えるものである．ゴール p(...,Ti,...) とヘッドユニファイ可能な節が

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

のように再帰的に定義されており，

$$Ti^{jk}\ \text{is-a-proper-subterm-of}\ Ti^{j0}$$
$$Ti\ \text{is-an-instance-of}\ Ti^{j0}$$

があるi で，全ての再帰呼び出し(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．この展開は，subterm 関係のwell-foundednessによって無限には続かないことが保証される．このwell fouded orderingは部分計算時に呼び出し列の履歴を見てその都度調べても良いが，予めプログラムを静的に調べることによってわかるものもある．

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

において１番めの節のヘッドとボディの再帰呼び出しに注目すると，第１引数と第３引数においてボディ側はヘッド側のproper subtermになっていることがすぐわかる．proper subterm 関係は明らかに一つのwell fouded orderingである．

さて，全計算時においてもappendの第１引数と第３引数とが共にunbound のゴールが呼ばれると，１番めの節で無限ループする．従って，実行時の呼び出し関係が実際にこのsubterm 関係によるwell fouded orderingによって止まる計算となるためにはヘッドユニフィケーション時に呼び側の第１引数或いは第３引数の少なくともいずれか一方はconsパタンになっていなければならない．言い替えると，第１引数或いは第３引数において情報の流れは節の左辺から右辺に向かっていなければならない．

５．抽象解釈に基ずく形式化

部分計算に有用な情報がプログラムを静的に解析して抽出されるが，この解析は抽象解釈の概念によって形式化される．

５．１　抽象解釈

抽象解釈の概念について概略を述べる．

[Cousot 81]によれば，

> ”プログラムはある対象領域における計算を表示している．抽象解釈は，同じプログラムを別の抽象化された領域での計算を表示するものとして捉えなおし，

これを実行するものである．その結果，実際の解釈による計算に関して何らかの情報を得ることができる．”

例えば，整数の領域での計算を表示したプログラムが与えられたとき，これを符号（＋，－）の領域に抽象化して解釈することができる．その結果，例えばそのプログラムのあらゆる計算が正の整数しか扱わないというようなことがわかれば，負の場合を考慮したプログラム部分を省略することができる．

Prologの場合，一般に対象領域はHerbrand空間である．常にinstantiate されているような引数を＋，常にinstantiate されていない引数を－と抽象化してプログラムを解釈すると，いわゆるモード解析を行なうことができる．また，Herbrand空間を項形により区分したものに抽象化してプログラムを解釈すると，項形推定を行なうことができる．さらに，タイプ推定その他の解析が同様にして行われる[Mellish 86],[Kanamori 87].

５．２　述語呼び出し解析の形式化

前節で行なった述語呼び出しの解析は，ホーン節の各アトムの変数を無名化し，アトム間の変数の共有を無視するという抽象化を行なうことにより抽象解釈の枠組みで形式化できる．

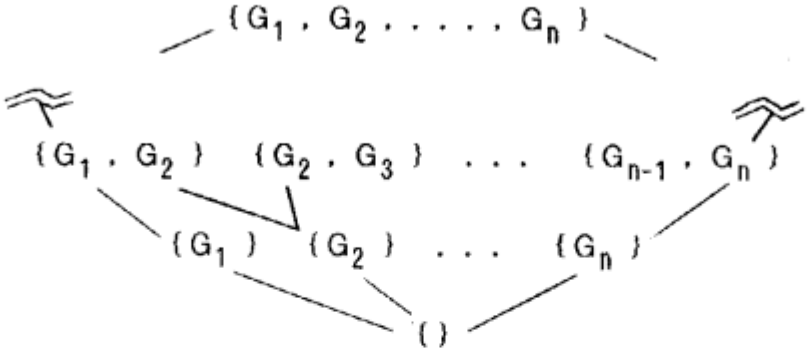
前節のプログラム例で，ゴール間の変数の共有を無視すると，

```
solve(true,[100]).
solve((#1,#2),#3) :- solve(#1,#2), solve(#1,#2), append(#1,#2,#3).
solve(not(#1),[#2]) :- solve(#1,[#2]), #1<20, #1 is 100-#2.
solve(#1,[#2]) :- rule(#1,#2,#3), solve(#1,#2), cf(#1,#2,#3).
```

というプログラムが得られるが，これは元のプログラムを抽象化したものになっている．ここで，変数の#iはアトム内の異なる変数を区別し，別のアトムの#iの出現と同一とは見做さない．また，ヘッドユニフィケーションではユニファイ可能性だけを計算し，変数のバインデイングはボデイに及ばない．このプログラムでsolve((#1,#2),#3) をトップレベルゴールとすると，solve(#1,#2)，append(#1,#2,#3)，solve(#1,[#2])，#1<20 ，#1 is 100-#2，rule(#1,#2,#3)，及びcf(#1,#2,#3)の各ゴールを呼ぶことがわかる．solve(not(#1),[#2]) ，solve(#1,[#2])も同様である．

この解析は次の連立方程式を解くことに相当する．

```
solve(true,[100]) = {}
solve((#1,#2),#3) = solve(#1,#2)  ∪ solve(#1,#2) ∪ append(#1,#2,#3)
solve(not(#1),[#2]) = solve(#1,[#2])  ∪ #1<20  ∪ #1 is 100-#2
solve(#1,[#2]) = rule(#1,#2,#3) ∪ solve(#1,#2) ∪ cf(#1,#2,#3)
```



solve(A)は，solve(A)が呼び出す可能性のあるゴールパタンの集合とする．XUYは，X，Y自身及びXの値，Yの値の集合和とする．ゴールパタンは有限であるから，そのpower set も有限である．集合の間には包含関係によって部分順序がつき，これにより束が形成される．さらに集合和の演算は単調である．従って，各ゴールパタンの初期値を{}とおいて反復法により解けば有限回で次のように収束して不動点解を得る．

```
solve(true,[100]) = {}
solve((#1,#2),#3) =
solve(not(#1),[#2]) =
                  { solve(#1,#2), append(#1,#2,#3),       }
solve(#1,[#2]) = { solve(#1,[#2]), #1<20, #1 is 100-#2, }
                  { rule(#1,#2,#3), cf(#1,#2,#3)          }
```

6．おわりに

Prologプログラムの部分計算ではまず述語の呼び出し関係が再帰的か否かがポイントとなる．大域的に非再帰的な述語は展開し尽くすことができる．再帰的な述語のゴールの展開は，定義節のヘッドとボデイ部の再帰呼び出しとの間のsubterm 関係によるwell founded ordering ，及び部分計算時のゴールと定義節ヘッドの間のinstance関係のチェックにより安全に行なうことができるが，前の条件は部分計算の前にプログラムを静的に解析してチェックされる．こうして自動化された部分計算プログラムは多フェーズの構成をとることになった．

プログラムの静的な解析は全て抽象解釈の概念で統一的に形式化されることが示されつつある．本研究の次の目標は自己適用可能なPrologの部分計算プログラムの実現であるが，実用的な部分計算プログラムの実現にはプログラムの特性に関するより精密な解析を行なう必要があると思われる．それらの解析の抽象解釈による形式的取り扱いもそれ自体有益なテーマであると考えられる．

References

[Boyer 79] Robert S. Boyer and J Strother Moore, *ACM Monograph Series: A Computational Logic*, Academic Press Inc., 1979.

[Cousot 77] Patrick Cousot and Radhia Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," *Conf. Record of the 4th ACM Symp. on Principles of Programming Languages, Los Angeles*, 1977, pp.238-252.

[Cousot 78] Patrick Cousot and Radhia Cousot, "Static Determination of Dynamic Properties of recursive Procedures," Neuhold, E.J.(ed.): *Formal Description of Programming Concepts*, North-Holland, 1978, pp.237-277.

[Ershov 78] Andrei P. Ershov, "On the essence of compilation," ibid. pp.391-420.

[Futamura 83] Yoshihiko Futamura, "Partial evaluation of programs," *Proc. RIMS Symp. Software Science and Engineering, Kyoto, Springer LNCS 147*, 1983, pp.1-35.

[Jones 85] Neil D. Jones, Peter Sestoft and Harald Søndergaard, "An experiment in partial evaluation: The generation of a compiler generator," *Proc. 1st Intl. Conf. on Rewriting Techniques and Applications, Dijon, France, Springer LNCS 202*, 1985, pp.124-140.

[Jones 86] Neil D. Jones, "A Semantics-based Framework for The Abstract Interpretation of Prolog," (To appear).

[Kahn 84] Kenneth M. Kahn and Mats Carlson, "The Compilation of Prolog Programs without the Use of a Prolog Coimpiler," *Proc. of the International Conference on Fifth Generation Computer Systems*, 1984, pp.348-355.

[Kanamori 84a] Tadashi Kanamori and Hirohisa Seki, "Verification of Prolog Programs Using An Extention of Execution," ICOT TR-093, 1984.

[Kanamori 84b] Tadashi Kanamori and Hiroshi Fujita, "Formulation of Induction Formulas in Verification of Prolog Programs," ICOT TR-094, 1984.

[Kanamori 87] Tadashi Kanamori, "Analyzing Success Patterns of Logic Programs by Abstract Interpretation," ICOT TR (To appear).

[Kursawe 86] Peter Kursawe, "How to Invent a Prolog Machine," *Intl. Conf. on Logic Programming*, 1986, pp.134-148.

[Mellish 85] C.S. Mellish, "Some Global Optimizations for a Prolog Compiler," *Journal of Logic Programming, Vol.2, No.1*, 1985, pp.43-66.

[Mellish 86] C.S. Mellish, "Abstract Interpretation of Prolog Programs," *Intl. Conf. on Logic Programming*, 1986, pp.463-474.

[Ono 84] Satoshi Ono, Naohisa Takahashi and Makoto Amamiya, "Non-Strict Partial Computation with a Dataflow Machine," *Proc. 6th RIMS Symposium on Mathematical Methods in Software Science and Engineering, Kokyuroku 547, RIMS Kyoto Univ.*, 1985, pp.196-229.

[Safra 86] Shmuel Safra and Ehud Shapiro, "Meta Interpreters for Real," CS86-11, Department of Computer Science, The Weizmann Institute of Science, Isreal, May 1986.

[Sestoft 86] Peter Sestoft, "*The Structure of a Self-Applicative Partial Evaluator, Proc. of a Workshop on Programs as Data Objects, Springer LNCS 217*, 1985, pp.236-256.

[Takeuchi 86] Akikazu Takeuchi and Koichi Furukawa, "Partial Evaluation of Prolog Programs and its Application to Meta Programming," *Information Processing 86, Proc. of the IFIP 10th World Computer Congress*, 1986, pp.415-420.