

〔研究〕

General Tracerの研究 (III) --G.H.C.によるPascal系言語の意味記述--

A Study of General Tracer (III)--A Description of the Semantics of a Pascal-like Language Using G.H.C.--

大谷木 重 夫
S. OHYAGI

A denotational description of Programming language is given by interpreting syntax of the language as functions on $P(\omega)$. This correspondence is usually provided through λ -Calculus. But it can also be done through another language whose semantics has been already known such as Prolog. In this paper a denotational description of Pascal-like language using G.H.C. which is a subset of Prolog is presented.

§ 1 はじめに

プログラム言語の意味記述ではDenotational Approachが理論的には最も望ましい。即ちそれは λ -Calculusに $P(\omega)$ 上の連続関数を対応させ λ -Calculusでプログラム言語の各構文を記述するものである。 λ -Calculusの抽象性から簡潔な表現が得られる。それだけでなく前掲論文³⁾で見たように与えられたプログラムの性質を証明するためにも便利なものであった。

本稿では、 λ -Calculusと同じくらい抽象的で、その意味論も λ -Calculusを媒介として与えられているProlog型言語G.H.C.を用いてPascal系プログラム言語の意味記述を与えたい。Prologそのものを使わない理由は、G.H.C.で十分記述可能であることとそれによってトレーサの制御構造が著しく簡潔になるからである。

Pascal系言語をPrologへ写すとProlog interpreterは過去の変数の値を全て保存するから前方向のトレースを過去に引き戻すことができるので、トレースの際一度通り過ぎてしまった場所へ戻れるから、高速で虫の生じた場所へ到達できる。これはG.H.C.でも同じであるので、より簡単なG.H.C.を採用することが望ましい。

§ 2 Pascal系言語

ここで取り上げるのはDonahueにより意味が与えられた次のBacus Nauer Formをもつ言語である。

```
<identifier> ::= <letter>{<letter>|<digit>}
<variable> ::= <identifier>|<identifier>[<expression>]
<expression> ::= <variable>|<identifier>(<value list>)
                |<constant>|<unary operator><expression>
                |<expression><binary operator><expression>
                |eof
<value list> ::= <empty>|<value>{,<value>}
<value> ::= <identifier>|<expression>
<unary operator> ::= +|-|~
<binary operator> ::= +|-|*|div|mod|&|v|=|≠|<|>|>
<procedure or function block> ::= <procedure
                or function heading><variable block>
<procedure or function heading> ::= procedure
                <identifier>(<identifier list>:<value list>)
                |function<identifier>(<value list>)
<identifier list> ::= <empty>|<identifier>{,<identifier>}
<variable block> ::= var <identifier list>;
                array <identifier list>;<label block>
                |var <identifier list>;<label block>
                |array <identifier list>;<label block>
                |<label block>
<label block> ::= label<identifier>:
                <statement>;<label block>|<procedure block>
<procedure block> ::= <procedure or function
                block><procedure block>
                |begin <statement> end
```

KEY WORDS: パスカル, プログラム指示意味論, G.H.C.

```

<statement> ::= <variable> := <expression>
           |<identifier>(<identifier list>:<value list>)
           |goto <identifier> |null |read(<variable>)
           |write(<expression>)
           |if <expression> then <statement>
           else <statement> fi
           |while <expression> do <statement> od
           |for <identifier> := <expression>
           to <expression> do <statement> od
           |<statement>;<statement>
           |begin <statement> end
<program> ::= program <identifier>;
           <variable block>.

```

Pascalのプログラムは、宣言列とその後の実行文からなっている。宣言列はまずvariable, function, procedure及びlabelの宣言からなっている。ここで述べたPascal subsetではProgramは一つの入力fileと一つの出力fileを使い、結果は出力fileに得られる。

プログラムの構成単位はstatementである。statementには代入文, goto文, procedure call, read文, write文等がある。これらの基本文から、条件文, 複合文, 繰り返し文といった構造文が作られる。

文を構成するものとしてはexpressionがある。expressionはoperandを使って値を得る方法を指定する。operandは変数, 定数, 及び関数等であり、演算子と組み合わせて複雑な式を構成する。expressionの値は、整数か0,1である。

更にくわしく各構文について解説する。まず識別子即ちidentifierであるが、これは英字から始まる英数字の列で、変数, 手続き, 関数, ラベル等を示すのに使われる。変数は、プログラムが操作する値の名前であり、一つの値, あるいは配列をさす。配列はここでは簡単のため一次元のものしか許されていない。関数は()内の引数に対して値を計算し、識別子に値を返すもので、文法的には

```
<identifier>(<value list>)
```

で定められる。<value list>は値, value のリストで値は、識別子か式, expressionである。関数の引数は関数の本体の計算の前に評価されていなければならない。また関数の値として配列をとることは許されない。式は変数, 関数, 及び+, -, 或いは-につけたもの, 式と式を二変数の演算子で結んだもの及びeofである。

eofは入力fileがemptyになるとtrueとなる識別子である。+は和, -は差, ¬は否定を表す一変数の演算子で、*は積, divは商, modは余り, &は論理積, は論理和を表す二変数の演算子である。≠は不等号 =は等号, <は小な

り, ≤は小か等しい, >は大なり ≥は大か等しいを表す演算子で0,1に値もつ。文は、代入文, procedure文, goto文, read文, write文, if文, while文, for文, 文の並び, 及び, begin block からなっている。代入文は変数に式の値を代入するもので

```
<variable> := <expression>
```

で定められる。

procedure文はprocedure名に(して引き数名のリストと値のリストを並べ)をとじる形に書かれる。goto文はgotoの後に文ラベルを書く。read文はreadの後に変数を書く。変数に入力fileの現在の先頭の値が入る。

write文は式の値を出力fileに書き出す。

if文は式, expressionの値を計算し、式の値がtrue(=1)であれば最初のstatementを0であれば次のstatementを評価する。while文はexpressionがtrueであるかぎりstatementの評価を繰り返す。

for文は識別子の値が、最初の式で表される初期値より、次の式で表される最終値迄一つずつ増加させる毎にstatementを評価しつづける。begin endでくくった文もまた文である。Programはprogramの宣言の後、その識別子即ちidentifierを書き、その後にProgramの本体を書く。Programの本体は<variable block>で指定される。

従ってその中味は

1. 変数宣言
2. 配列宣言
3. 文ラベルの宣言
4. 手続き及び関数の宣言

とその後の

```
begin <statement> end
```

で指定されるbegin blockからなる。

変数の宣言は

```
var <identifier list>;
```

また配列の宣言は

```
array <identifier list>;
```

で指定される。

文ラベルの宣言は

```
label <identifier>:
```

の後本体である手続き及び関数宣言, ラベルの宣言, 及びbegin blockが書かれる。手続きの宣言はprocedureの宣言の後その識別子に(してパラメータ(識別子)のリスト及び変数の値を並べて)を閉じた後procedureの本体がつづいたもので、procedureの本体は、他のprocedureや関数の宣言の後のbegin blockである。

ここで簡単なプログラム例を与えよう。

```
program longestupsequence
```

```
var x, n, m
```

```

array a, b
procedure readarray(a, n)
  begin n := 1 end
  label next:
    read(x);
    a(n) := x;
    if  $\neg$  eof then
      goto next
    else fi
procedure solution(a, n, b, m)
  var k, l, i
  array c,d,e,e1
  begin
    if n = 1 then
      b(1) := a(1);
      m := 1;
    else fi
    for k := 1 to n-1
      do
        c(k) := a(k+1)
      od
    for k := 1 to n
      do
        if a(k) > a(1) then
          a(k) := a(1)
        else
          a(k) := -1
        fi
      od;
    l := 1;
    for k := 1 to n
      do
        if a(k) = -1 then
          d(l) := a(k);
          l := l+1
        else fi
      od;
    solution(c, n-1, e, s);
    solution(d, l, e1, s1);
    if s > s1 then
      begin
        for i := 1 to s
          do
            b(i) := e(i)
          od
        end

```

```

    else
      begin
        for i := 1 to s1
          do
            b(i) := e1(i);
          od
        end
      fi
    end
  procedure writesolution(b, m)
    var i
    begin
      for i := 1 to m
        do
          write(b, i)
        od
      end
    begin
      readarray(a, n);
      solution(a, n, b, m);
      writesolution(b, m)
    end

```

このプログラムは自然数のリストが与えられたときその部分リストで小さなもの順に並んだもののうち最大の長さのものをdisplayするものである。まずreadarrayで入力ファイル中の自然数列をarray aに読み込み手続きsolution(a, n, b, m)で解となるリストを配列bにしまう。solutionの内容はまず最初の数を除いた残りのリストをcにしまう。次に最初の数を含み最初の数以上の大きさの数をリストアップした配列dを得る。solutionの再帰呼び出しによりcとdからそれに対応する解e, e1を得る。最後にeとe1のうち長い方を解bとする。最後に解を出力ファイルに書きだす。

§ 3 Pascal 系言語の G.H.C. による semantics

前節で定義されたPascal系言語に G.H.C.で semantics を与えよう。G.H.C.には denotational semantics があたえられているので述語変数に関数を代入することができる。従ってcontinuation function等も扱えるはずではあるが、本論文では普通の G.H.C の場合に制限しておくことにする。このためプログラムは状態と状態の関係と見なす。

状態 s はプログラムに現れる全ての変数に対して値を定めるものである。また環境 e は名前に対し、それが指す関数、あるいは手続きあるいはラベルを定める。sin は入

カファイルsoutは出力ファイルの内容を表す。

sは具体的には

```
s([id1, v1, [id2, v2, ..., [idn, vn]]...])
```

というskolem関数である。

```
s(a) <- b
```

は

```
s([...[a,b,[...[]]...])
```

を意味する。eについても同様である。プログラムの意味は次の関係 m(p) により定められる。

```
m(p) [ [ program id : variable_block ] ] (v*, y*)
```

```
:- m(s) [ [ procedure id : variable_block;
  begin id end ] ] (e_init, s <- T, sin(v*), sout(nil),
  e', s', sin', sout(y*)), !.
```

プログラムの意味はそれを本体とする手続きの定義とそのプログラムの呼び出しからなる。

手続きの意味は

```
m(s) [ [ procedure id (id1* : id2*)
  variable_block procedure_block ] ]
(e, s, sin, sout, e', s', sin', sout')
```

```
:-
m(s) [ [ procedure_block ] ] (e(id) <-
p [ [ procedure id (id ; id) variable_block ] ]
(u1, u2), s(id) <- T, sin, sout, e', s', sin', sout') !.
```

手続きの定義の意味は、環境の手続き名の欄にその本体を結合させることである。

```
p [ [ procedure id (id1* : id2*) variable_block ] ]
(u1, u2) (e, s, sin, sout, e', s', sin', sout')
```

```
:-
m(s) [ [ variable_block ] ] (e(id) <-
p [ [ procedure id (id1* : id2*) variable_block ] ] (v1, v2),
s(id1*) <- u1, s(id) <- T, sin, sout, e', s'(id2*) = u2,
sin', sout') !.
```

p [[procedure id (id₁* : id₂*) variable_block]] はその本体の記述である。

```
m(s) [ [ function id (id*) vblock, pblock ] ]
(e, s, sin, sout, e', s', sin', sout')
```

```
:-
distinct(ide),
branch [ [ function id (id*) : variable_block;
  procedure_block ] ]
(e, s, sin, sout, e', s', sin, sout) !.
```

```
branch [ [ function id (id*), variable_block;
  procedure_block ] ] (e, s, sin, sout, e', s', sin, sout)
```

```
:- assigns_to([ variable_block ], [ id* ]), !, fail.
```

```
branch [ [ function id (id*) variable_block;
  procedure_block ] ] (e, s, sin, sout, e', s', sin, sout)
```

```
:- m(s) [ [ procedure_block ] ]
(e(id) <- func [ [ function id (id*)
  variable_block ] ] (u)(e, s(id) <- T,
  sin, sout, e', s', sin', sout) !).
```

```
func [ [ function id (id*) variable_block ] ] (u)
(e, s, sin, sout, e', s'(id)=y, sin, sout)
```

```
:-
m(s) [ [ variable_block ] ] (e, s(id*) <- u,
s(id) <- T, sin <- T, sout <- T, e', s', sin, sout) !.
```

関数の定義にはまず関数の本体を環境に登録すること及び関数の本体の記述を与えることである。その際ファイルの操作が行われないことに注意する。即ち sin'=sin, sout'=sout である。

```
m(s) [ [ label id : label_block ] ] (e, s, sin, sout,
e(id) <- m(s) [ [ label_block ] ], s(id) <- T, sin, sout) !.
```

ラベルの定義は単に label_block をラベル名と結合させることである。

変数や配列の宣言は

```
m(s) [ [ var id1* : array id2* : label_block ] ]
(e, s, sin, sout, e', s', sin', sout')
```

```
:-
m(s) [ [ label_block ] ]
(e(id1*) <- T, e(id2*) <- T, s(id1*) <- , s(id2*) <- ,
sin, sout, e', s', sin', sout') !.
```

である。

begin_block の意味は

```
m(s) [ [ begin statement end ] ]
(e, s, sin, sout, e', s', sin', sout')
```

```
:-
m(s) [ [ statement ] ] (e, s, sin, sout, e', s', sin', sout') !.
```

である。

次に繰り返し構文について考えよう。

```
m(s) [ [ for id := exp1 to exp2 do statement od ] ]
(e, s, sin, sout, e', s', sin', sout')
```

```
:-
m(e) [ [ exp1 ] ] (v1)(e, s), Int(v1),
m(e) [ [ exp2 ] ] (v2)(e, s), Int(v2),
iterate([ id ], v2, [ statement ] )
(e, s(id)=v1, sin, sout, e', s', sin', sout') !.
```

```
iterate([ id ], v, [ statement ] )
(e, s(id)=u, sin, sout, e, s, sin, sout)
:- (u > v) !.
```

```
iterate([ id ], v, [ statement ] )
(e, s(id)=u, sin, sout, e", s", sin", sout")
```

```
:- (u ≤ v),
m(s) [ [ statement ] ] (e, s, sin, sout, e', s', sin', sout') !.
```


関数の意味は

$$\begin{aligned} m(e)(\llbracket \text{id}(\text{varg}^*) \rrbracket, y)(e(\text{id})=u, s, \text{sin}, \text{sout}) &:- \text{Func}(u), \\ m(a)(\llbracket \text{varg}^* \rrbracket, v)(e, s, \text{sin}, \text{sout}), &\text{Val}(v), \\ u(v)(e, s, \text{sin}, \text{sout}, e', s'(\text{id})=y, \text{sin}', \text{sout}') &!, \end{aligned}$$

単項演算は

$$\begin{aligned} m(e)(\llbracket \text{uop expression} \rrbracket, x)(e, s, \text{sin}, \text{sout}) \\ :- m(e)(\llbracket \text{expression} \rrbracket, u)(e, s, \text{sin}, \text{sout}), \\ \text{Unop}(\llbracket \text{uop} \rrbracket, a), a(u, x), !. \end{aligned}$$

二項演算は

$$\begin{aligned} m(e)(\llbracket \text{expression1 bop expression2} \rrbracket, x) \\ (e, s, \text{sin}, \text{sout}) &:- \text{Binop}(\llbracket \text{bop} \rrbracket, b), \\ m(e)(\llbracket \text{expression1} \rrbracket, v)(e, s, \text{sin}, \text{sout}), \\ m(e)(\llbracket \text{expression2} \rrbracket, w)(e, s, \text{sin}, \text{sout}), \\ b(v, w, x), !. \end{aligned}$$

§4 おわりに

本論文の記述はほとんど Donahue¹⁾をたどった。ただここで言いたいのは Pascal 系言語の semantics を G.H.C. で書けるということである。従って G.H.C. のトレーサで Pascal 系言語のトレーサがかかる。

謝 辞

本論文は情報アーキテクチャ分散システム研究室で筆者が行った研究をまとめたものである。研究室長であった塚本亨治氏および研究室の皆様には公私にわたりたいへんお世話になりました。ここに厚く御礼申し上げます。また論文の作成に当たりましては大蒔情報アーキテクチャ部長にアドバイスをいただきました。また内容につきましては木下算譜言語論代表世話人にお世話になりました。

参 考 文 献

- 1) J.E.Donahue "Complementary definition of programming language semantics", LNCS 42 1976
- 2) S.Ohyagi "General tracer の研究 -denotational description of Prolog-"
- 3) S.Ohyagi "General tracer の研究 -G.H.C. の abstract interpreter-"

(1998. 3.12受付)