記 号 処 理 55－4
（1990．6．4）

論理プログラミング機能を埋め込んだ手続き型言語“c．”の言語仕様と実現方式

矢野稔裕　小谷善行

東京農工大学工学部電子情報工学科

われわれは論理プログラミング機能を埋め込んだ手続き型言語 c．を設計した。これは、C言語の機能を損なうことなく、リスト処理や非決定性実行処理の記述能力を向上させた拡張C言語である。本稿では、その言語仕様と実現方式についてのべている。論理プログラミングの機能を手続き型言語で利用するための一つの手法を具体的に示している。

“c．”：A Procedural Language with Logic Programming Feature and Its Specifications and Implementation

Toshihiro YANO,　Yoshiyuki KOTANI
Department of Electronic and Information Science
Tokyo University of Agriculture and Technology
2-24-16 Nakacho Koganei Tokyo 184 Japan

A procedural programming language called "c．" is designed, which has logical programming feature in its specifications. It is an extended c language that grows up describability of list processing and non-deterministic control processing. We show its language specifications and the method of implementation. This is also a practical example of how to deal with logical programming paradigm in procedural languages.

１．はじめに

Prolog のような論理型言語であっても、実用的な応用プログラムを書くと、pure でない機能を多用し、手続き的な部分が増加することが多い。また、効率を重視する応用では、手続き型言語によるプログラムと結合することも多い。そのような Prolog を基にした結合とは逆に、手続き型言語を基にして Prolog が持つ論理プログラミングのメカニズムを導入することが別の方向として考えられる。もしこのことが可能となれば、高速な処理が要求されるアプリケーションのなかに、エキスパート知識を埋め込むような応用が広がるであろう。

われわれは、C 言語の上位互換性をもつ論理プログラミング可能な言語 c．(「c period」と呼ぶ)の仕様を設計し、実現手法を検討した。

２．制御の仕様

２．１　論理関数

Prolog における手続き（同じ名前とアリティを持つ節の集合）に相当する処理単位として、論理関数を導入する。c.のプログラムは、通常関数（従来の関数）と論理関数によって構成される。論理関数の記述形式は次のとおりである。

(1)論理関数は、その内部が節と呼ぶ単位に分割され、それぞれに単一化されるパターンが前置きされている。

(2)論理関数であることを示すのは、logic という型を宣言することで行う。

(3)関数 main はプログラムに必ず一つ存在する通常関数である。

(4)論理関数の記憶クラスは static でもよい。

２．２　pattern ラベル

ラベル付き文の構文を拡張して pattern ラベルを新設する。pattern ラベルは論理関数の定義の中だけで使うことができる。論理関数の引数と同数の項を pattern と ':' の間に ',' で区切って並べる。これを引数パターンと呼ぶ。

pattern ラベルは、論理関数のトップレベルに現れなくてはならない。ブロックの中に書くことは許されない。

引数がなく、かつ１節しかない論理関数では pattern ラベルの省略ができる。これは Prolog での引数のない一つの単位節に相当する。

pattern ラベルが一つの文に複数付くことはできない。

構文を図2.1 に示す。

```
<labeled-statement>::=
    <identifier> : <statement>  |
    case <const-exp> : <statement>  |
    default : <statement>  |
    pattern <term-list> : <statement>  |
    pattern : <statement>

<term-list>::=
    <term>  |
    <term-list> , <term>
```

図2.1　pattern ラベルの構文

２．３　節

論理関数は pattern ラベルの付いた一つ以上の部分に分割される。その各部分を節と呼ぶ。節には、式文による論理関数の呼出しが書け、制限はあるが従来の C 言語の記述を含められる。

論理関数から直接呼び出される論理関数を、サブゴールと呼ぶ。サブゴールの呼出しは節のトップレベルだけで行ってよい。ブロックの中に書くことや、if 文、while 文などに現れることは許されない（図2.2 参照）。

```
logic lfunc1();

logic lfunc0()
{
        pattern:
            lfunc1();          /* 可 */
        pattern:
            lfunc1();          /* 可 */
            {
                lfunc1();      /* 不可 */
            }
        if( ... ) lfunc1();    /* 不可 */
}
```

図2.2　サブゴール呼出し

２．４　論理関数の制御

論理関数が呼ばれると、実引数と pattern ラベルのもつ引数パターンが単一化され、最初に成功した pattern ラベルの付いた節に制御が移る。その節に記述されたプログラムが成功すると、この論理関数を呼び出した親が示す次に実行すべき関数に制御が移る。失敗してバックトラックが発生すると、次の節が試される。代替節がなくなるとこの関数の失敗となって、最も最近の代替節をもつ親（あるいは祖先）の論理関数にバックトラックする。

２．５　論理関数の呼出し

論理関数は、論理関数、通常関数どちらから

も呼ばれうるが、それぞれの意味は異なる。

論理関数が論理関数から呼び出されて成功したときは、通常関数から帰ったかのように、呼出しに続く文が実行される。このとき、呼ばれた論理関数は値を返さない。

失敗したときは、最も最近の代替節をもつ論理関数にバックトラックして次のパターンとの単一化を試みる。

これに対して通常関数から呼び出すと、成功したときは値 SUCC を返し失敗したときは値 FAIL を返す。成功は1回だけ可能であり、さらに別解を求めるためにバックトラックを発生させることはできない。暗黙のカットが入っていると考えられる。

論理関数から呼ばれた通常関数から、論理関数を呼び出してもよい。しかし、その呼出しを越えるようなバックトラックは発生せず、成功、失敗にかかわらず親である通常関数に制御が戻り、返り値によってそれを知らせる。

3．データオブジェクトの仕様

3．1　式の拡張

Prolog に準じた表記で項（アトム、整数、リスト、複合項）を記述するため、定数式を拡張して term 演算子を導入した。この演算子によって、項がリストである場合、文字列形式で書かれたリストである場合、タグ付きの整数である場合 C 言語の式と明確に区別できる。term 演算子の非演算数には、括弧に囲まれた項を一つ記述できる。入れ子にはできない。この式の結果は TERM 型である。

アトム、複合項、pattern ラベルの引数パターンの記述では、term 演算子は不要である。

使用例を図3.1 に、構文を図3.2 に示す。

3．2　アトム

アトムは、それを表記する識別子（印字名）と同じ文字列を値とするような大域的なオブジェクトである。

ユーザは、プログラムに現れるすべてのアトムの印字名を宣言する。複数のファイルに分かれているプログラムなら、同じアトム宣言をアトムを表記しているすべてのファイルで include する。

アトムの宣言は外部だけで行う。つまり関数やブロックに局所的なものは宣言できない。

値は TERM 型を持つ。ライブラリ関数によって文字列を得ることができる。使用例を図3.3 に構文を図3.4 に示す。

3．3　論理引数

論理関数は0個以上の引数を受け取ることができ、それらは単一化の対象となる。この引数を論理引数と呼び、その個数をアリティと呼ぶ。通常関数の引数とは異なりアリティを可変にできないので、宣言、定義、呼出しのアリティを一致させなくてはならない。一致していなければエラーとなる。

論理関数の定義においてアリティを示すため、仮引数リストとして型名 TERM をアリティ分並べる。この仕様は、文献[4]で導入された、引数

```
term( [] );
term( 123 );
term( "ab" ); /*term(['a','b']);の略記*/
term( [ 3, functor( atom1 ) ] );
```

図3.1　term 演算子使用例

```
<constant-expression>::=
    <conditional-expression>   |
    term ( <term> )

<term>::=
    <atom-name>   |
    <atom-name> ( <term-arguments> )   |
    <list>   |
    <string>   |
    <integer-constant>   |
    <character-constant>   |
    <enumeration-constant>   |
    <identifier>

<term-arguments>::=
    <term>   |
    <term> , <term-arguments>

<list>::=
    [ ]   |
    [ <list-expression> ]

<list-expression>::=
    <term>   |
    <term> , <list-expression>   |
    <term> | <term>
```

図3.2　定数式の構文

```
atom { atom1, functl }
 ...
p = atom1;
q = functl( [], atom1 );
```

図3.3　アトム

```
<external-declaration>::=
  <function-definition>   |
  <declaration>   |
  <atom-declaration>

<atom-declaration>::=
  atom { <atom-declarator-list> }

<atom-declarator-list>::=
  <identifier>   |
  <atom-declarator-list> , <identifier>

<constant>::=
  <integer-constant>   |
  <character-constant>   |
  <floating-constant>   |
  <enumeration-constant>   |
  <atom-name>   |
  <atom-name> ( <term-arguments> )
```

図3.4 アトム宣言に関する構文

をとらない関数を定義する際の仮引数リストに、型名 void を書く構文に似ている。

論理関数は、呼び出す前に定義されるか宣言される必要がある。int 型以外の値を返す通常関数と同じである。論理関数の宣言には、アリティを示すための引数宣言が必要となる。型名 TERM を仮引数リスト同様にアリティだけならべる。

論理関数の宣言の引数リスト、定義の仮引数リスト、pattern ラベルのパターン数、これらすべてでアリティが一致しなくてはならない。

論理関数を呼ぶときの実引数は、TERM 型でなくてはならない。

3．4 論理変数

Prolog と同じく論理変数は型のない変数であり、宣言なしで利用できる。

論理関数の定義の内部で、宣言されていない識別子を論理変数として認識し、次の pattern ラベル（最後の節なら関数定義の終わり）までで有効である。ただし、term 演算子の被演算数か、静的なオブジェクトの初期化式ではエラーとなる。

論理変数は TERM 型の式となる。

4．実現の概要

処理系は、先に C 言語の一部である前処理系 cpp を通したソースを入力とする前処理系によって実現する。こうすることで、include、define 、条件コンパイルなどの機能をそのまま利用できる。c．ソースにはヘッダ cperiod.h を include することとし、この中に型定義やマクロを置く。

Prolog 処理系実現のモデルとして WAM[1]がある。本処理系は、これに基づいた C のソースコードを生成する。WAM の命令セットに相当する実行時関数ライブラリと、本処理系の出力したコードをコンパイルしたものとを結合して実行ファイルを作る。

WAM のレジスタは大域変数として確保され、WAM の３本のスタックは C の実行時のスタックとは別に確保された領域が使用される。

項は、タグ付きデータの構造体であるセルへのポインタ値に変換される。引数パターンかサブゴールの引数として現れた項は WAM に準じた命令列（get、put、unify命令）に変換される。

4．1 pattern ラベル

pattern ラベルは Prolog の節頭部に相当する記述であり、WAM の頭部の実現方式に準じた形で変換される。すなわち、get 命令と unify 命令の並びに展開される。変換例を図4.1 に示す。

```
pattern [ X | L1 ], L2, [ X | L3 ]:
        <節>
           (a) c．ソース

if(   _get_list( 0 ) &&
      _unify_variable_x( 3 ) &&
      _unify_variable_x( 0 ) &&
      _get_list( 2 ) &&
      _unify_value_x( 3 ) &&
      _unify_variable_x( 2 )
){
      <節のコード>
}
           (b) C ソース
```

図4.1 pattern ラベル変換例

この図で、if 文の条件式に現れる関数は、単一化を行う実行時関数である。単一化に成功すると条件式は真となり、本体の実行に制御が移る。失敗すると条件式は偽となり本体は実行されない。次の pattern ラベルがあればその処理に、なければこの関数の失敗となってバックトラックが発生するようなコードが次に続く。

4．2 節

節は論理関数中の pattern ラベルの付いた単

文や複文である。各節は含まれる N 個のサブゴールによって N+1 個に分割された関数に変換される。最初の部分は論理関数本体から変換された関数に含まれ、残り N 個は N 個の補助関数となり、それらへのポインタを持つ継続配列が生成される。補助関数に分けることで、処理の入口を関数へのポインタとして得ている。

継続配列は、論理関数の成功時に次に飛ぶべき関数へのポインタと、そのときに有効な環境の大きさをもつ配列である。CP レジスタがその要素を指し示している。WAM では、CP レジスタはコード領域の手続きを直接指すポインタであるが、この実現では間接的になる。また、環境の大きさは call 命令の第2オペランドで示されていた。しかし CP レジスタを基準にしてアクセスすることは代わってない。継続配列の要素の定義を図4.2 に示す。

```
typedef struct {
        logic     (*goal)();
        int       envSize;
} _ContinBlock;
```

図4.2　継続配列要素

継続配列は、変換後の論理関数に静的な局所配列として定義される。

補助関数名は、なるべく親の関数名と呼び出そうとする関数名を反映しつつ衝突の起こらないような規則で、適切に名付けられる。関数名はコンパイラのエラーメッセージに含まれ、デバッガにも渡されてユーザの目に触れるため、デバッグの助けとなるような配慮である。N が2以上のときの例を図4.3 に示す。

この方法による実現の WAM 命令との対比を表4.1 に示す。

表4.1　WAM の procedural 命令との対比

WAM 命令	本処理系
proceed	(*_CP->goal)()
call pred,N	_CP++,pred()
execute pred	pred()

4．3　非決定的実行処理

引数パターンとマッチングすべく渡された引数が変数であるとき、その引数は出力として働き、引数パターンとの単一化はすべての pattern ラベルにおいて必ず成功する。そのためバックトラックが及ぶごとに順番に単一化しなくてはならない。このような制御を実現する try 系の WAM 命令相当の処理を図4.4 のように実現する。

```
logic   lfunc()
{
    pattern ＜引数パターン＞:
        ＜文１＞
        lfunc1();
        ＜文２＞
        lfunc2();
        ＜文３＞
        lfunc3();
}
```

(a) c．ソース

```
logic   lfunc()
{
    static _ContinBlock _contin1[] = {
        { _lfunc_lfunc2, E1 },
        { _lfunc_lfunc3, E2 }
    };
     ...
    _allocate();
    if( ＜単一化＞ ){
        ＜文１＞
        ＜put,unify命令列＞
        _CP = _contin1;
        lfunc1();
    }
     ...
}
static _lfunc_lfunc2()
{
    ＜文２＞
    ＜put,unify命令列＞
    _CP++;
    lfunc2();
}
static _lfunc_lfunc3()
{
    ＜文３＞
    ＜put,unify命令列＞
    _deallocate();
    lfunc3();
}
```

(b) C ソース

図4.3　節の実現

```
pattern ＜パターン１＞:
        ＜節１＞
pattern ＜パターン２＞:
        ＜節２＞
pattern ＜パターン３＞:
        ＜節３＞
```

図4.4(a)　非決定的処理 c．ソース

```
_makeChoicePoint();
if( !setjmp( _B.cp->jb )){
      if( ＜パターン１の単一化＞ ){
            ＜節１＞
      }
      longjmp( _B.cp->jb, F );
}
_undo();
if( !setjmp( _B.cp->jb )){
      if( ＜パターン２の単一化＞ ){
            ＜節２＞
      }
      longjmp( _B.cp->jb, F );
}
_undo();
_discardChoicePoint();
if( ＜パターン３の単一化＞ ){
      ＜節３＞
}
longjmp( _B.cp->jb, F );
```

図4.4(b)　非決定的処理 C ソース

実行時関数 _makeChoicePoint と _discard-ChoicePoint は、それぞれ選択点の生成と放棄を行う。_undo は、ヒープスタックとトレイルスタックをポップアップして論理変数の束縛を無効にする。

setjmp/longjmp は C 言語の標準ライブラリ関数であり、大域ジャンプの機能を提供する。setjmp(_B.cp->jb) は、B レジスタ（数種のポインタの共用体として定義されている）の指す、現在の選択点の中の jb フィールドに計算機の持つ物理的なレジスタセットを保存して０を返す。この jb フィールドは WAM の選択点の BP フィールドに相当している。この処理で、失敗が発生したときに戻るべき状態を保存したことになる。

longjmp(_B.cp->jb, F) によって、B レジスタの指す選択点の jb フィールドに保存されているレジスタセットを復帰して、setjmp の呼出しから帰ったかのように制御を移す。このとき値 F を返す。

よって、まず最初に if 文が実行されると条件(!setjmp()) は真となって、節の処理が実行される。ここで単一化が失敗するか、あるいは成功して制御が前進したがその先で単一化に失敗すると、longjmp の実行によって先の set-jmp 呼出しから帰る動作に制御が移る。今度は if 文の条件は偽となり、_undo が呼出されバックトラック処理が完了する。

次に単一化を試みるべき節があれば、setjmp によって jb フィールドが更新されて同様に処理が進む。

次の節が最後の節ならば、その節の処理に移る前に選択点を捨てるので（_discardChoice-Point によるポップアップ）、親か祖先が造った選択点が最後の選択点として有効になる。この節あるいはその先での失敗は、この論理関数自体の失敗を意味し、longjmp によって親か祖先に直接バックトラックすることになる。

４．４　インデキシング処理

第一引数のタグによって分岐し、それが定数（アトムと整数）あるいは複合項ならば、さらにそれらの値によるハッシュ法の併用で分岐する。本処理系では、switch 文を使って第一引数のタグやハッシュ表をみてそれぞれのグループに分岐する。例を図4.5 に示す。

第一引数が未代入の変数のときは非決定的処理となり、すべての節との単一化が試みられなければならない。case ラベル（case REF:）以後に並ぶ節が順に単一化の対象になる。

第一引数のタグによって、対応する節の単一化処理に直接飛び込んだときは選択点は作られない。このとき失敗時の飛び先は、親あるいは祖先の作った選択点の示すバックトラック点になる。

第一引数が同じタグであるパターンラベルが続くときは、タグによる分岐の後、ハッシュ表を引くようなコードに変換される。実行時関数 _lookUp がハッシュ表を引いて整数を返し、対応する単一化処理に飛ぶ。

WAM のインデキシング命令との対応を表4.2 に示す。

表4.2　WAM の非決定的処理との対比

WAM 命令	本処理系
try_me_else	_makeChoicePoint(), setjmp()
retry_me_else	setjmp()
trust_me_else	_discardChoicePoint()
try	_makeChoicePoint(), setjmp(), goto
retry	setjmp(), goto
trust	_discardChoicePoint(), goto
fail 処理	_undo(), longjmp()

```
logic lfunc( TERM ... )
{
    pattern atom1, ... :
        <節１>
    pattern atom2, ... :
        <節２>
    pattern atom3, ... :
        <節３>
    pattern atom3, ... :
        <節４>
}
```

(a)　c．ソース

```
logic lfunc()
{
    static _HashTable    _hashTable
              = { <ハッシュ表> };

    switch( _deref( &_A[0] )->tag ){
    case INT: case ATOM:
        switch( _lookUp( _hashTable,
                  _A[0].val.integ )){
        case 0:longjmp( _B.cp->jb, F );
        case 1:goto _c1;
        case 2:goto _c2;
        case 3:_makeChoicePoint();
            if( !setjmp( _B.cp->jb ))
                goto _c3;
            _undo();
            _discardChoicePoint();
            goto _c4;
        }
    case REF:
        _makeChoicePoint();
        if( !setjmp( _B.cp->jb )){
    _c1:    <単一化と節１処理>
            longjmp( _B.cp->jb, F );
        }
        _undo();
        if( !setjmp( _B.cp->jb )){
    _c2:    <単一化と節２処理>
            longjmp( _B.cp->jb, F );
        }
        /* c3 略 */
        _discardChoicePoint();
    _c4:    <単一化と節４処理>
            longjmp( _B.cp->jb, F );
    case LIST: case STRUCT:
            longjmp( _B.cp->jb, F );
    }
}
```

(b)　C ソース

図4.5　インデキシング

4．5　switch 文

switch 文を使うインデキシングでは、C コンパイラが、case 定数が連続した整数のときは O(1) の性能を持つコードを生成することを仮定している。身近にあったコンパイラについて調査した限りではその仮定は成立しており、switch 文の値で分岐表を引くコードが生成されている。もしも線形な検索となるようなコードが生成される場合、この実現法は速度的に効果の低いものとなる。逆に、switch 文がハッシュ法によるコードにコンパイルされるならば（調査したコンパイラにはそのようなものはなかったが）、明示的なハッシュ表とその検索関数は不要にできる。

5．データオブジェクトの実現

5．1　項

項は、WAM の内部表現と同じ形式で扱う構造体として実現する。すなわち、タグと値をメンバとする値セルの構造体型 CELL とそれへのポインタ型 TERM を定義している。（図5.1 参照）これらはヘッダに納められる。

```
typedef struct _cell {
    TAG       tag;
    union {
        int           atom;
        int           integ;
        struct _cell  *ref;
        struct _cell  *struc;
        struct _cell  *list;
    } val;
} CELL;
typedef       CELL    *TERM;
```

図5.1　項の型定義

型 TERM は、基本データ型として拡張されるわけではなくポインタを扱っているだけであり、C 言語での文字列の扱いと同様に注意深く使わなくてはならない。

5．2　アトム

各翻訳単位の変換時に、アトム宣言を基にアトムと内部表現である整数値とを対応づける。アトムの記述は、TERM 型になる。その値は、同じ翻訳単位の静的な変数を指し、そこにアトムを表すセルの定数が置かれる。

アトムは整数に１対１で対応し、その定数はアトム表の添字となっている。アトム表には印字名とアリティが格納される。

アトム表は、デフォルトでは関数 main を含む翻訳単位の変換時だけ実際に生成され、埋め込まれる。

５．３　論理引数の受渡し

引数の受渡しは、WAM の引数レジスタを経由する方法による。引数レジスタは大域変数を用いて実現するため、通常関数と同じスタックには積まれない。

論理関数から呼び出す場合、引数は WAM に従って put 命令、unify 命令の列に展開される。

通常関数から呼び出す場合は、引数レジスタへの代入文に展開され、実行時関数（_callLogic）を通した間接的な呼出しに変換される。この実行時関数は、選択点を一つ造ってその時点までの管理情報を凍結し、目的の論理関数が成功しても失敗しても親の通常関数に戻るために setjmp を用いている。

この実行時関数が最初に呼び出されたときは、WAM に基づくデータ領域（スタック、ヒープ、トレイル）の確保と初期化をも行う。

変換例を図5.2 に示す。

```
logic lfunc( TERM );

pfunc()
{
    lfunc( atom1 );
}
```

(a) ｃ．ソース

```
static CELL   _c0001[] = { ATOM, a1 };

pfunc()
{
    (_A[0].tag = _c0001->tag,
     _A[0].val.atom = _c0001->val.atom,
     _callLogic( lfunc ));
}
```

(b) C ソース

図5.2　通常関数からの論理関数呼び出し

５．４　論理変数

WAM では、述語の呼出しを越えて保持されるべきか否かによって変数を分類する。そのため、サブゴール呼出しの間に記述された論理変数によっても分類に影響がある。

論理関数の記述中に現れた論理変数は、その中から間接的に呼ばれた論理関数による A レジスタの破壊があり得るため必ずパーマネントに分類する。

引数パターンか、サブゴール呼出しの実引数に現れる論理変数に関しては WAM のコンパイルに従い、get 命令、put 命令、unify 命令によって操作される。

６．まとめ

おもに、われわれの設計した言語ｃ．の言語仕様および、それが変換されたｃ言語表現の実現形式を述べた。現在ｃ．システムの作成を行っている。

その特徴は「C 言語の機能を損なうことなく、リスト処理や非決定性実行処理の記述能力を向上させた」ということである。具体的には、次のようにいえる。

(1) 本質的に非決定的である部分以外は、C 言語による従来の記述とすることで効率を優先したシステムが記述できる。

(2) 前処理系の生成する C ソースコードは、C 言語の標準的な機能だけで構成されているため、組込みシステム用の C コンパイラを使えば、組込み可能な論理型言語として使用することができる。

この言語の応用としては、探索問題を解くプログラム、エキスパートシステム、ゲームシステムの記述などがありうる。つまり、高速の処理を要求されるが、同時に論理的なエキスパート知識を容易に表現することが求められる応用である。

つけ加えると、ｃ．は、論理型プログラミング機能を手続き型言語で利用するための一つのやり方を具体的に示している。さらに洗練された言語仕様のためのたたき台にできるだろう。

参考文献

[1] Warren, D.H.D.: An Abstract Prolog Instruction Set, SRI International Technical Note, No. 309 (1983)

[2] J.L.Weiner, S.Ramakrishnan: A Piggyback Compiler For Prolog, Proc. of SIGPLAN '88 Conference on Programing Language Design and Implementation Atlanta, Georgia, June 22-24 (1988)

[3] B.W.カーニハン/D.M.リッチー 著　石田晴久 訳: プログラミング言語Ｃ, 共立出版 (1981)

[4] B.W.カーニハン/D.M.リッチー 著　石田晴久 訳: プログラミング言語Ｃ第２版, 共立出版 (1989)

[5] 矢野稔裕、高田正之、小谷善行: 論理プログラミング機能を埋め込んだ手続き型言語ｃ．, 情報処理学会第40回全国大会講演論文集, 5J-9, pp.968-869(1990)