記 号 処 理　37－8
（１９８６．６．２０）

TAO/ELIS上でのCプログラミング環境

梅村　恭司
(NTT電気通信研究所)

今　昭
(東北大学工学部)

C言語を会話型言語であるＴＡＯに翻訳することにより，Cのプログラミングを ＴＡＯ／ＥＬＩＳ上で行なえるようにした。これにより，実行の途中でプログラムの定義を変更したり，任意の部分から実行をはじめるなどの機能が実現できた。現存するCのデバッガでは，エラーの場所は検出できても修正はその場ではできなかったが，本方式ではその場での定義の変更も容易である。大規模なプログラムにおいて，変更から実行までのサイクルが短いという特徴は非常に有効である。

Realizing an interactive C programming Environment.

Kyoji Umemura
(NTT Electrical Communication Laboratories),

Akira Kon
(Tōhoku University Engineering Department)

An interactive programming environment for C has been realized with a lisp dialect TAO, on a lisp machine ELIS. A program written in C is directly translated into TAO, and then loaded into memory as usual TAO program. The C program runs as fast as a usual TAO program with additional micro programmed functions. C programs are developed interactively just like LISP programs. Any program can be modified during its execution as a LISP program can. This feature is very useful for the large scale development of C program.

1．はじめに

プログラム開発のための道具は近年整備されつつあり，シンボリックデバッガなどを利用すればエラーの生じた時点を詳しく調べることが可能である。しかし，現在のC処理系では，原因が理解できてもその場でプログラムの定義を修正することは不可能である。簡単な変更であれば，そのままオブジェクトコードを変更することも考えられるが，ソースコードとオブジェクトの不一致を招き，問題が多い。この実現には，デバッガからソースを修正するエディタを呼び出し，それをコンパイルし，オブジェクトにとりこむ機能が必要である。

Ｌｉｓｐ，Ｓｍａｌｌｔａｌｋなどの開発環境が整備されている言語には，実行時にプログラムを修正する機能が整っており，プログラムの開発が加速されている。エラー時の各変数の保持する値を名前で参照できるばかりか，複雑な構造をもつデータもうまく表示する。そして実行経過を凍結しておいたままで更に誤りを発見するために部分実行をおこない，誤りを発見し，それをその場で修正して，凍結しておいた実行を継続することができる。Cなどの言語にはそのような機能が整備されていない。Cでこれを実現するには，オブジェクトコードの見直し，メモリ管理の追加など非常に多くの変更を言語処理系に加えなければいけない。そして，デバッガの機能を実現するにも多量のプログラムを必要とするであろう。実行時にプログラムを修正する機能は便利であることは理解されつつも，それを実現するための処理系の変更が著しいため実現がなされなかった。

もし，Cの処理系を　完全につくり直し，実行時にプログラムの変更がなされるようにし，かつデバッグを便利なように整備したものを作ったとしよう。するとそれは，Ｌｉｓｐなどの実行時のプログラムの変更を実現した処理系と非常に似たシステムになると予想される。オブジェクトコードの管理方法，変数名と値の管理などの方法は同一のものとなる。もし，良いＬｉｓｐの処理系があり，それを利用する方法があればプログラムの節約になるであろう。

C言語をＬｉｓｐなどに翻訳することが可能であれば，そのままＬｉｓｐのもつデバッガ，ステッパ，トレーサ，ダイナミックリンキングなどの機能が利用でき，目的としたものが実現できる。実行時にプログラ

```
(defstruct binding-pointer get put)
;
(defmacro hidar (v)
   (make-binding-pointer
      :get #'(lambda () ,v)
      :put #'(lambda (binding-new-value)
               (setf ,v binding-new-value))))
;
(defvar *c-memory*)
;
(defun mig (x)
  (cond ((integerp x) (svref *c-memory* x))
        ((consp x) (car x))
        ((binding-pointer-p x)
         (funcall (binding-pointer-get x)))
        (t (error "cannot deref ~a" x))))
;
(defun make-base-vector (n)
  (setf *c-memory* (make-sequence 'vector n)))
```

図1.　マイクロサポート関数と等価なCommon Lisp関数

ムを変更することも，デバッガの内から翻訳結果をロードするだけのことで，非常に単純にできる。問題はCのもつアセンブラに近い機能をＬｉｓｐで直接に実行可能かどうかである。

ＴＡＯ／ＥＬＩＳにおいては，どうしてもＬｉｓｐで書けない機能，効率上で問題となる機能をマイクロコードでサポートしてCからＴＡＯへの翻訳を実現した。Cの関数がＴＡＯの関数に，Cの変数をＴＡＯの変数とする条件を保ったままで翻訳する。Cの関数はそのままＴＡＯの関数として定義でき，CとＴＡＯの関数を共存させて，互に呼びあうように記述できる。そして，実行時にプログラムの定義を変更できるCの開発環境を実現した。

2．マイクロサポート関数

変数のアドレス渡しと，ポインタを効率良く実現するために，ｈｉｄａｒ，ｍｉｇ，とｍａｋｅ－ｂａｓｅ－ｖｅｃｔｏｒの三つの関数をマイクロプログラムで用意した。ｈｉｄａｒは，変数の左辺値をとる&演算子に対応する。ｍｉｇはポインタの参照をする＊演算子に対応する関数である。ｍａｋｅ－ｂａｓｅ－ｖｅｃｔｏｒはプロセスごとの領域を宣言するものに対応する。（図1）

（1）　ｈｉｄａｒは変数の値のアドレスを返すもので，変数のｂｉｎｄｉｎｇという特別のものを返す。概念的には変数の値をとる関数閉包と，変数の値を変更する関数閉包の組を返すものと考えればよい。実際にはこのようなものは非常にコストが高いので使用できない。ＴＡＯにおいてはバインディングされている場所のアドレスを適切なタグを立てて返して実現している。

（2）　ｍｉｇは，先をたどることにより値をとる関数であり，数にたいして宣言された配列の要素をとり出し，ｈｉｄａｒで作られたｂｉｎｄｉｎｇであればその値をとり出し，リストであればそのｃａｒをとるという関数である。リストはストラクチャに使用するための措置である。ｍｉｇはCのプログラムに多用されるので，マイクロで書く意味がある。実際にはほぼｃａｒと同程度の速度で動作する。概念的に作るとｆｕｎｃａｌｌを実行することになるが，ＴＡＯにおいてはｆｕｎｃａｌｌを実行せず，メモリ参照をするだけである。更にｍｉｇは単なるＬＩＳＰであれば，ｓｅｔｆで代入できるよう，ｓｅｔｆマクロを定義する必要がある。ＴＡＯにおいては左辺値を扱う機構が働き，ｓｅｔｆマクロを定義しなくても動作する。

（3）　ｍａｋｅ－ｂａｓｅ－ｖｅｃｔｏｒは，Cで使用する全メモリを宣言するものである。概念的にはグローバルな変数にベクターを代入するものであるが，ｍｉｇを高速に実行するためにグローバル変数を使用せず，プロセスオブジェクトのスロットにベクターを代入する。そして，走行中にはマシンのレジスタにそのベクターの番地を保持する。これらの関数により変数アドレスを扱うCのプログラムが実行できる。
（図2）

3．Cのデータ形式のＴＡＯへのマッピング

3．1．基本のデータ形式

Cの整数はＴＡＯの整数に，Cの実数はＴＡＯの実数で表現する。ｌｏｎｇとかｓｈｏｒｔなどの長さの指定は無視して，すべてｌｏｎｇで処理するのと等価である。ＴＡＯでは整数の多倍長化は自動的に行なわれるので，このCの整数の桁数の上限はなく，多倍長の計算ができる。実数はすべて倍精度で実行する。
（図3）

```
int i=1;
int mem[1];

main ()
{
  mem[0] = 2;
  /* variable memory swap */
  printf("initial i=%d, j=%d\n",i,mem[0]);
  swap(&i,&mem[0]);
  printf("final   i=%d, j=%d\n",i,mem[0]);
}

swap (x , y) int *x,*y;
 {int temp;
      temp = *x;
      *x = *y;
      *y = temp;
 }
```

```
(def-cvar i 1)
(def-cvar mem (make-c-vector 1))
(defun .main nil
 (setf (mig2 mem 0) 2)
 (.printf (c-string "initial i=%d, j=%d\n") i (mig2 mem 0))
 (.swap (hidar i) (+ mem 0))
 (.printf (c-string "final   i=%d, j=%d\n") i (mig2 mem 0)))
(defun .swap (x y)
              (let* ((temp nil))
                    (setf temp (mig x))
                    (setf (mig x) (mig y))
                    (setf (mig y) temp)))
```

```
initial i=2, j=2
final   i=2, j=2
```

図2. 変数の左辺値をとるプログラムの例

Cの文字はＴＡＯの整数に，文字列はＴＡＯの文字列ではなくて，整数の配列と同じ形式で表現する。文字列の最後は数の０で終了する。この形式はCのプログラムの互換性により必要である。文字はバイトの列でなく，１語を使用していてメモリ効率は悪い。プロセスごとに高速操作できる領域はＬＩＳＰデータの配列なので，文字だけを詰めて配置することが困難である。そのため数と同一の表現とした。本格的な処理を行うときメモリ効率が問題となる可能性がある。

Cプログラムのデバッグにおいては，文字列が数字であることが大きな障害となる。トレス情報などの中に数字があらわれても，それが何へのポインタであるかわからなく，多少デバック能率が低下する。この点の改良は現在検討中である。

3．2．配列

Cの配列は大きな配列の添字として表現する。多次元の配列は配列ポインタの配列の形式で実現する。当初，Cの配列をＬｉｓｐ流のベクターに合致できないかと検討したが，ポインタ操作で加算，減算が不可能などの理由で断念した。Cの配列をＬｉｓｐのベクトルに合致させるためには，ベクトル本体と添字の二つを一つのデータとしてあつかい，かつ，そこへの加算などが定義されなければならない。加算に手を加えるのは，マイクロコードの変更でコストをかけずに実現できるものの，二つのデータを一つのデータとしてあ

```
main()
 { int i;
   for (i=1;
        i<25;
        i++      )
        {printf("factorical(%10d)=%30d\n",i,fact(i));
        }
  }

int
fact (n) int n;
{ if (n == 1) return 1;
        else return n*fact(n-1);
}
```

```
factorical(         1)=                              1
factorical(         2)=                              2
factorical(         3)=                              6
factorical(         4)=                             24
factorical(         5)=                            120
factorical(         6)=                            720
factorical(         7)=                           5040
factorical(         8)=                          40320
factorical(         9)=                         362880
factorical(        10)=                        3628800
factorical(        11)=                       39916800
factorical(        12)=                      479001600
factorical(        13)=                     6227020800
factorical(        14)=                    87178291200
factorical(        15)=                  1307674368000
factorical(        16)=                 20922789888000
factorical(        17)=                355687428096000
factorical(        18)=               6402373705728000
factorical(        19)=             121645100408832000
factorical(        20)=            2432902008176640000
factorical(        21)=           51090942171709440000
factorical(        22)=         1124000727777607680000
factorical(        23)=        25852016738884976640000
factorical(        24)=       620448401733239439360000
```

図3. 簡単な計算例

つかうためには演算のたびにメモリ領域を消費するので，不適当と判断した。

Cプログラムの全体のメモリ領域を実行前に指定しなければならない。それは，最初に大きな領域を確保する必要があるからである。メモリ管理のある言語の上での処理系にもかかわらず，そのメモリ管理の機能が利用できないのは不満であるが，ポインタ操作との整合性からやむをえない。Cのポインタからでは生きているメモリ領域がどこであるかを決められない。メモリ領域が不足した場合には，新しい領域に古いデータをすべてコピーする方法で領域を広げることが可能である。しかし，古いデータにアクセスする可能性がない領域が残っていても，再利用ができない。

3．3．構造体

Cの構造体はリストで表現することにした。構造体メンバへのアドレスは，それを保持しているＣｅｌｌで表現する。たとえば，２番目のメンバのアドレスはｃｄｒ，２番目のメンバの値はｃａｄｒで求める。値を求めるｍｉｇにおいてｃｅｌｌであればｃａｒをとるのは，構造体を実現するための措置である。（図4）

構造体をリストで表現することによる問題は，構造体そのものと構造体へのポインタの区別がなくなることである。トランスレータは変数のタイプによりメモリ参照の回数を調節する必要がある。しかし，メモリ参照の回数が正しければ構造体型の変数はポインタを保持することにしても不都合は生じない

構造体がリストであることに付随して，構造体を値として返す関数なども容易に実現できる。システムが自動的にメモリ管理を行ってくれるので，構造体が静的でも動的でも一律にコードの生成ができる。構造体へのポインタでは，実体が消えてポインタだけがのこるという問題は起きない。

構造体のメンバのアドレスを操作して次のメンバのアドレスを求めることはできない。しかし，このような操作は多くの場合，マシン依存のものであるので実現しなくても構わないと判断した。このような操作は，実行時に完全にチェックされる。

```
typedef struct
        { int one;
          int two;
        } pair;
pair x,*y;
main()
{       pair z;
        y = &x;
        x.one = x.two = 1;
        z.one = z.two = 2;
        int_swap(&z.two ,&y->two);
        printf("%d %d %d\n", x.one, y->one, z.one);
        printf("%d %d %d\n", x.two, y->two, z.two);
}
int_swap(x, y) int *x,*y;
{       int temp;
        temp=*x;
        *x=*y;
        *y=temp;
}
```

```
(def-cvar x (list nil nil))
(def-cvar y nil)
(defun .main nil
 (let* ((z (list nil nil)))
       (setf y x)
       (setf (first x) (setf (second x) 1))
       (setf (first z) (setf (second z) 2))
       (.int_swap (cdr z) (cdr y))
       (.printf (c-string "%d %d %d\n")
                (first x)
                (first y)
                (first z))
       (.printf (c-string "%d %d %d\n")
                (second x)
                (second y)
                (second z))))
(defun .int_swap (x y)
                 (let* ((temp nil))
                       (setf temp (mig x))
                       (setf (mig x) (mig y))
                       (setf (mig y) temp)))
```

```
1 1 2
2 2 1
```

図4. 構造体を使用したプログラム

4．性能評価

プログラム開発用の処理系であると言っても，プログラムの実行速度が極端に遅いと開発に使用できない。ＴＡＯ／ＥＬＩＳではマイクロプログラムの追加により，通常のＬＩＳＰプログラムと遜色ない実行速度が得られた。とくにＴＡＯはインタプリタの速度が早いので，インタプリタの実行でＣ／ＶＡＸ７５０の１／３程度の速度で動作する。更にＴＡＯコンパイラを使用するとＶＡＸ７５０と同等の速度が得られ，Ｃの開発用として十分な速度を持つ。インタプリタで良い速度が得られることは，開発の環境として重要な性質である。

ＥＬＩＳはAMD２９シリーズを使用したＴＴＬマシンであり，ＶＡＸ７５０と同じテクノロジーの計算機であることを考えると，このＣで行っている実行時の検査がオーバーヘッドになっていないことがわかる。ＶＡＸ－７８０のＶＡＸ－ＬＩＳＰによるトランスレート結果に比較すると，ＶＡＸ－ＬＩＳＰコンパイラを使用しても，ＴＡＯのインタプリタでの実行よりも遅く，通常の計算機では実行時の検査がかなりの負荷になっているのがわかる。ＥＬＩＳにおいては，実行時の検査がオーバーヘッドになっていないのであるから，ＴＡＯに翻訳する形式の言語処理系は通常の機械語に翻訳する形式の処理系を作成した場合と比べても速度の差はない。

5．開発環境の評価

ＣをＴＡＯに翻訳できるので，Ｃのプログラムの修正はＬＩＳＰのプログラムと全く同様にでき，容易なものとなった。一関数ごとに定義をいれかえて実行できるという特徴は，大きなシステムのデバッグ時に有益なものとなる。大きなソースプログラムがあっても，その一部の関数をエディタで切り出すことにより修正ができる。普通の方法ではいくらリンカの速度が早くとも，ソースプログラムの一部だけをとり出して，その部分を変更することはできない。ＵＮＩＸなどでは，ソースプログラムを分割後，オブジェクトを生成する方法を記したｍａｋｅファイルを作り直して，分割した両方をコンパイルするという操作が不可避である。本論の方法では変更部分だけのコンパイルで実行できる。Ｌｉｓｐなどの言語に備わっている実行時での関数定義の変更の機能がＣでも実現できたことにより，ソースコード修正から実行までのサイクルを短くなり，プログラム開発の速度を著しく向上する。

実行時のエラーにしても，エラーの起きた関数や，そのときの変数などの値が即座に検査でき，場合によってはメインプログラムでない場所から実行可能であり，Ｌｉｓｐの環境そのままでＣのプログラムの開発ができる。常に強力なデバッガと共に動作するのでプログラムの開発は早く，かつ生成されたプログラムは他の小さなコンピュータでも動作する。

6．まとめ

ＣをＬｉｓｐの一方言であるＴＡＯに翻訳する方法で，高い開発効率をもつＣの処理系が作成できた。プログラムの部分変更が再リンクなしで可能なため，エラーを修正しながら実行を続けるという今までのＣの処理系では実現できなかった特徴が実現できた。ソースの修正から再実行までのサイクルは短く，大規模なプログラムにおいて特にこの機能が有効となる。さらに，マイクロプログラムの追加により，実行速度もＴＡＯ／ＥＬＩＳ本来のものにすることができた。このような処理系を他の言語について作成することも容易であり，ＴＡＯを使用して他の言語の開発システムをつくる方法が有効であることを，実証的に確かめた。

謝辞

MICROPROGRAMMING の追加を実現してくれた竹内郁雄氏に感謝する。Ｃの言語仕様の細部の確認に協力してくれた天海良治氏，山崎憲一氏に感謝する。入出力関数の整備に協力してくれた沢　浩一氏に感謝する。

参考文献

1. Brian W.Kernighan and Dennis M.Ritchie:"THE C PROGRAMMING LANGUAGE",PRENTICE-HALL,INC.,Englewood Cliffs, New Jersey 07632.

2. Guy L. Steele Jr.:"COMMON LISP,THE LANGUAGE",Digital Press,EY-00031-DP.

3. I,Takeuchi, H.G.Okuno, and N.Osato:"TAO - A harmonic mean of Lisp, Prolog, and Smalltalk", SIGPLAN NOTICES, Vol.18, No.7, July 1983.

4. H.G.Okuno and et al:"TAO : A Fast Interpreter-Centered System on Lisp Machine ELIS", ACM LISP and Functional Programming Conference, 1984.