システムソフトウェアと
オペレーティング・システム 78－5
（１９９８．５．７）

トランスレータ方式のエージェント言語の実装

下川僚子　　山本英雄　　梅村恭司
豊橋技術科学大学 情報工学系
E-mail:{ryoko, hideo, umemura}@avenue.tutics.tut.ac.jp

トランスレータを利用して、エージェント言語を実現した。異なる機種の間でプロセスの移動を可能とするために、プログラムの実行のコンテクストをＣコンパイラのコードで処理できるものとする規約を設定し、その規約を守るようなコード生成を行なうトランスレータを作成した。生成したコードは、プログラムの実行のコンテクストを機種独立の文字形式に変換できる。それを利用して、エージェントの移動の機能を実現している。

Implemetating Agent Language using translator

Ryoko Shimokawa, Hideo Yamamoto and Kyoji Umemura
Department of Information and computer science
Toyohashi University ofTechnology

We have implemented an agent language system as translator. In order to realize the migration of processes among different architecure machines, we have defined constraints and developed an translator, whose output satisfies the costraints. The ouptput codes are able to generate the execution context in machine independent format. We use this ability to realize the process migration.

1 はじめに

分散処理の基礎技術として、エージェントがある。エージェントを実現する基礎技術は、異なる機種の間でプロセスを移送することである。エージェントを実現する方法の多くは、仮想プロセッサを本来のプロセッサの上に実現して、そのうえで動作するプログラムを移送していた。これは、プログラム言語の実現の上で、機種独立のバイトコードを設定して実現していることと類似点がある。

一方、機種の独立性を高めるプログラミング言語の実現法には、トランスレータを用いる方法がある。すなわち、ある言語を広く使われている言語に翻訳することで、機種に独立に言語を実現する方法である。この方法に類似のエージェントの実現方法はあまり知られていない。

われわれは、トランスレータ方式のエージェント言語を実現した。具体的には、原理的な動作を確認する簡単なエージェント言語を設計し、その言語からCへのトランスレータと、エージェントを移送するためのライブラリを整備した。その結果、適切に設定されたUnixで動作しているシステムについて、走行するコンピュータのアーキテクチャに依存せず、エージェントの動作コードがネイティブコードで動作できるようになった。

2 ターゲット言語と基本方針

我々は、ターゲット言語としてCを選びトランスレータを作成した。Cは高級アセンブラと呼ばれることがあり広く使われている言語であるためである。

エージェントの記述言語をP0と呼ぶことにし，方式を確認する単純なものとすることにした。整数をデータの型として、分岐、繰り返し、再帰できる関数呼出し、それに、エージェントの移動先を番号で指定する基礎機能を持つ。この言語は原理を確認する目的で設計したものである。制御と呼出しの情報が、機種非依存で移動できる機能が、トランスレータ方式を実現できるかどうかの鍵である。分散オブジェクト指向システムでの結果を利用すれば、文字列、浮動小数点、配列などについて、実装する上で原理的な問題はない。現在、Cのようなポインタを実装する上で問題があり、この方式の制限と考えている。しかし、配列を実現すれば、プログラムの記述能力の上で問題はない。

具体的には、P0はPascalに似た構文をもち、migrate文をもつ言語である。図1に示すのが、単純なプログラム例である。6行目のmigrate文はあらかじめ設定してあるn番目のコンピュータシステムに移動して、プログラムを実行する機能を実現している。設定がない場合はエラーであり、現在はエージェントはその場で停止してしまう。実際上は、これでは困りエラーを通知するようにすべきであるが、エージェントが正しく移動できることを実証的に確認することを目的であるので、簡略な仕様とした。このプログラムは、設定してある0から9の計算機の上を渡りあるいて、そこでのコンソールに数字を一つ出力するプログラムである。

サンプルプログラム1

```
1: program sample1;
2: var i;
3: begin
4:     i := 0;
5:     while i < 10 do begin
6:         migrate i;
7:         i := i + 1;
8:         write i
9:     end;
A: end.
```

図1 サンプルプログラム

3 関数呼び出しの分解

Cの実行途中の状態を機種独立なテキストに変換する際、関数の呼び出しの状態をテキストから復元することは、一般には困難である。なぜならば、戻りアドレスはプログラムのコードの内部のエントリーであり、その値はC言語からは直接意味を与えられていないからである。

そこで、関数の呼び出しはC言語の組み込み機能を利用せず、自分で用意したグローバルなスタックを使い、呼びだし戻り場所を関数名に対応させて実現した。このため、関数呼び出しについては性能のペナルティーが生じる。しかし、関数呼び出しのないループなどについてはCと同じ速度で動作させることができる。

関数呼び出しは、関数から戻ってくるときに行なう仕事をスタックにプッシュして、関数の仕事を開始するというセマンティクスを持つ。これを忠実に実現したという見方もできる。このようなスタックで、関数呼び出しを実現した。翻訳結果

のコードでは、このスタックを関数ポインタのスタックとして表現した。

生成されたCにおける関数は引数のないものに翻訳した。P0の関数／手続きの引数については、関数のポインタとは別のスタックで管理することとした。そして、Cのスタックを実質的に使わないようにコードを生成する。実現するP0言語のレベルでは再帰呼出しができるが、それを実行するCのコードでは、再帰は行なわれない。Cの制御スタックを使わないことで、実行途中の状態をテキストに変換することと、そのテキストから実行途中の状態を回復することが可能となった。

簡単な引数のない関数呼び出しの処理は図2のようになる。まず、翻訳結果では、図2のような実行時ルーチンがあるとする。つまり、やりのこした仕事を積むスタックがあり、その先頭から一つ取り出しては呼び出すという作業で全体が動作するのを規約としてコード生成した。

```
extern int sp;
extern void (*C[])(void);

void main_loop(void){

  for(sp=INIT_SP; sp>=0; )
   { work=C[--sp]; (*work)();}

}
```

図2：関数呼び出しの実行部分

この規約のもと、図3のようなfnが呼びだされているシーケンスを考える。コード生成部では、関数の呼び出しがあるところで、関数を二つに分ける。そして、戻ったあとに行なう部分を独立の関数とする。呼び出しを行なう文では、戻りに対応する関数のポインタと呼び出す関数のポインタをスタックに積み、関数を終了する。すると規約にしたがって、呼び出したい関数が実行され、そのあと戻りのコードが実行される。

このようにすることで、呼出し回数に比例するオーバヘッドが存在することになる。しかし、関数呼び出しを行なわない場合にはオーバヘッドは存在しない。

```
--in P0--
.... fn(); ....

--in C --
.... C[sp++]=&x0001; C[sp++]=&fn; return;}
 void x0001(void){ ......
```

図3：関数呼び出しの処理

4 分岐の分解

P0のソースプログラムでループ中に関数呼び出しを含む場合には、Cのコードでは2つの関数にわかれてれてしまう。すると、ある関数から別の関数に分岐しなければならなくなる。これをCのコードに直接対応させることはできない。そのため、ラベルの場所でも、生成するコードを2つの関数に分解する。その処理は図4のようになる。ラベルの場所に関数宣言を生成する。ラベルにはいる直前では、そのラベルに対応する関数をスタックに積み、実行中の関数を終了する。ラベルへ分岐する場所でも同様で、ラベルに対応する関数をスタックに積み、実行中の関数を終了する。

```
--in P0--
    ....
L:
    ....
    goto L;

--in C --
     ....
     C[sp++] = &L0001;
     return; }
void L0001(void){
     ....
     C[sp++]=&L0001;
     return; }
```

図4：分岐の処理

5 ローカル変数

P0の関数を翻訳後では2つ以上のCの関数に分解するため、P0の関数ローカルの変数はCの

グローバルな領域にとらなければならない。関数のポインタからなるスタックと別のスタックを用意し、P0での関数の開始の場所と終了の場所でスタックを操作するようなコードを生成した。

このスタック操作は関数呼出しに付随するので、この操作も関数呼出しの回数に比例したオーバヘッドとなる。ローカル変数がグローバル変数となることで操作速度が落ちることが予想され、これは全体的な速度低下となることと想定できるが、プログラムの実行時間の大部分が関数呼び出しを含まないループである場合には、オーバヘッドを解消する最適化は可能である。

6 関数ポインタのテキストとの変換

関数ポインタは、機種ごとに異なった値となる。機種に依存せずに関数のポインタのスタックを移送するには工夫が必要である。トランスレータで言語を実現しているので、トランスレータは関数のスタックに積まれる可能性のある関数の全体を数えあげることができる。関数ポインタの格納にグローバルな配列を使用しているので、テキストへの変換は図5のような構造のモジュールを作ればよい。このモジュールはトランスレータが自動生成する。

具体的には、与えられた関数のポインタを既知の関数のポインタと比較して、一致したら、関数の名前の文字列を求める処理である。これは、実行中断時のスタックの要素の数だけ呼び出され、スタックの要素を引数として呼び出される関数である。またこの関数は、実行中断時にコンテクストをテキストにする時に使われる。

```
char * name_of_function(void (*work)(void))
{
   if(work == &x0001)
          { return("x0001"); };
   if(work == &L0001)
          { return("L0001"); };
     :
     :
   fprintf(stderr,
       "Cannot get name(%x)\n",work);
     exit(1);
}
```

図5： 関数ポインタから関数名への変換

また、文字列から関数ポインタへの変換も原理的には、図6のような関数で実現できる。これもトランスレータが自動生成する。これは、テキストで与えられた関数名と既知の関数の名前を比較し、一致すれば、その関数のポインタを求めるものである。これは、実行中断時のスタックの要素の数だけ呼び出される関数である。またこの関数は、実行再開時に呼び出される。

```
void (*point_of_name(char * name)(void))
{
   if(strcmp(name,"x0001")== 0)
          { return &x0001; };
   if(strcmp(name,"L0001")== 0)
          { return &L0001; };
     :
     :
   fprintf(stderr,
       "Cannot get pointer(%s)\n",name);
   exit(1);
}
```

図6：関数名から関数ポインタへの変換

これらの関数は、2分探索を用いるなどコードの書き方で実行速度を速くすることができるが、現状では単純な実現方式を採用している。

7 トランスレータ

エージェント言語P0を、Cのプログラムに翻訳するトランスレータは、C言語で記述したものである。これは、約1400行のプログラムである。現在最適化は行なっておらず、コードの効率は悪いが、方式の実証的な確認はできている。

P0の簡単なサンプルプログラムと、その翻訳結果は図7に示す。翻訳にはソースとの対応がとれるようにコメントを追加したが、それ以外はコードそのものである。

```
1: program sample2;
2: var i;
```

```
 3:
 4: procedure fn(n);
 5:    i := i + n;
 6:
 7: begin
 8:    i := 0;
 9:    fn(3);
10:    write n
11: end.
```

```
static void fn(void)
{
        sp += 2;
        r0 = stack[1];/*from 5*/
        stack[sp] = r0; sp++;/*Push r0*/
        r0 = fp[1];
        stack[sp] = r0; sp++;/*Push r0*/
        sp--; r1 = stack[sp];/*Pop  r1*/
        sp--; r0 = stack[sp];/*Pop  r0*/
        r0 = r0 + r1;
        stack[sp] = r0; sp++;/*Push r0*/
        sp--; r0 = stack[sp];/*Pop  r0*/
        stack[1] = r0;
        return;
}
static void sample2(void)
{
        sp += 2;
        r0 = 0;/*from 8*/
        stack[sp] = r0; sp++;/*Push r0*/
        sp--; r0 = stack[sp];/*Pop  r0*/
        stack[1] = r0;
        stack[sp] = fp - stack; sp++;
        r0 = 3;/*from 9*/
        stack[sp] = r0; sp++;/*Push r0*/
        sp -= 2; fp = &stack[sp];
        PUSH(new_function0); PUSH(fn);
        return;
}
static void new_function0(void){
        sp = fp - stack; SP = stack[sp];
        fp = &stack[SP];
        r0 = stack[0];/*from 10*/
        fprintf(stdout,"%d",r0);
        fprintf(stdout,"\n");
        return;
}
```

図7：サンプルプログラムとその翻訳結果

サンプルのプログラムは簡単な関数呼び出しの例である。関数の呼ばれた9行目に対応する部分の後で新しい関数に分けられている。そして9行目に対応する部分は、新しく作られた関数new_function0と呼び出す関数fnを関数のスタックにプッシュする命令に翻訳されている。

8 エージェントの移送

実際に実行する際は、まずP0のプログラムを作成しトランスレータを用いる。するとob.cとob.hが作られる。これをgccでコンパイルしa.outを生成する。このa.outを実行することで、実際に実行が始まる。migrate文が実行されると引数に指定された数字で、行き先のシステムが特定される。実行の途中状態を図8に示すようなテキストファイルに生成する。ob.c, ob.h, とこのテキストファイルを転送し、転送先の計算機でコンパイルを行なう。引数にテキストファイルを指定するかたちで、相手のプロセスを起動し、自分は終了する。

```
2 cp
terminate      2bbc
new_function0      2314
3567362 r0
1 r1
3567361 r2
0 fp
3 sp
0 [0]
3567362 [1]
0 [2]
3567362 [3]
```

図8：実行状態情報のサンプル

実行状態のサンプルは、最初のコラムだけが使われる。空白からあとは、デバッグを助けるための情報である。たとえば、一行目のcpは関数のポインタの総数を示し、次の行の2bbcは関数terminateのアドレスである。この2bbcの情報は無視され、最初のコラムのterminateだけが使われる。r0, r1, r2は、式の計算に使われる途中情報であり、[0], [1], [2]はメモリの状態である。

現在はデータタイプは整数だけであり、タイプの情報を付与していないがデータタイプが増えた場合でも、タイプ毎に場所を特定できるようにコード生成をして、かつ、タイプの情報を追加することでテキスト化することに問題はない。

9 オーバヘッドの計測

最もオーバーヘッドのある条件を計測するために、再帰呼び出しを多用するハノイの塔のプログラムと、単純な関数呼び出しを10万回程度行なうようなプログラムについて実行時間を計測した。そしてそのプログラムに相当するCのプログラムの実行時間を計測した。その結果2つのプログラムの実行時間を比較した場合、今回作成したトランスレータを使用すると20～30倍になり，実行速度は低下している。実際に比較したハノイの塔のプログラムを図9に示す。この2つのプログラ

ムの実行時間の比は、Aのプログラムが Bのプログラムの約30倍であった。

プログラムA
(トランスレータを用いてCに翻訳して実行)

```
program hanoi4;
var n, total;

procedure move(n,x,y);
  begin total := total + 1 end;
procedure hanoi(n,a,b,c);
  var m;
  begin
    if n <= 1 then
      move(n,a,b)
    else
      begin
        m := n - 1;
        hanoi(m,a,c,b);
        move(n,a,b);
        hanoi(n - 1,c,b,a)
      end
  end;
begin
  total := 0;
  n := 20;
  hanoi(n,1,2,3);
  write total
end.
```

プログラムB (同じ動作をするCのプログラム)

```
#include <stdio.h>
int n,total;
void move(void)
{
      total++;
}
void hanoi(n,a,b,c)
{
      int m;
      if(n <= 1){
         move();
      }else{
         m = n-1;
         hanoi(m,a,c,b);
         move();
         hanoi(n-1,c,b,a);
      }
}
void main(void)
{
      total=0;
      n=20;
      hanoi(n,1,2,3);
      fprintf(stderr,"%d\n",total);
}
```

図9:比較したハノイの塔のプログラム

また、単純なループを10万回程度行なうようなプログラムについても比較した場合、トランスレータを用いた場合は直接のCのプログラムの20～30倍であった。これは、現在のトランスレータの問題であって、この方式の問題ではない。これを確認するために、関数の呼出しやローカル変数を同様に操作するという条件で、手で翻訳したプログラムについて 実行時間を計測した。その結果、ハノイのように、呼び出しが主体のプログラムにおいても3倍程度であった。今後トランスレータの改良を行なえば3倍程度で押えることができると考えられる。

分岐の分解において、すべての分岐を分解すると、コンピュータの本来の性能が出ないが、内部に呼びだしを含まない場合を検出することで、性能の低下を押えることができると考えられる。内部に関数を含む場合では、いずれにしろスタック操作が必要となっているので、このオーバヘッドは関数呼び出しのオーバヘッドと等価なレベルとなる。

また、ローカル変数の扱いにおいて関数呼出しをともなわないループを含む部分について、変数のスタックからローカル変数へコピーし直すコードを生成するようにコード生成を最適化すれば、関数呼び出しを含まないループへのオーバヘッドを避けることができると考えられる。

このようなことを考えて、トランスレータを改良することによりオーバーヘッドを減らすことができると考えられる。これらを実現すれば、関数呼び出しにオーバヘッドがかかるが、単純なループが処理時間の多くを占める典型的なプログラムのときにはオーバヘッドがかからないようになり、呼び出しが主体のプログラムにおいても3倍程度で押えられると考えられる。また、この3倍のオーバヘッドについても、文献[9]にあるようなコード生成方法を採用すればさらに縮小できる。また、呼び出しを行なわない場合には、原理的にはオーバーヘッドは存在しない。

10 考察

中間コードを設定して、エージェントを実現するのが一般的であるが、トランスレータを用いてもエージェントが実現できることを示した。エージェントを構成するプログラムが、ネィティブコードで動作することは意味のあることである。

中間コードを高速に実行する方法として、JIT(Just in time compiler)がある。JITなどで処理してネイティブコード実行を行なうものと比較して、JITを機種ごとにつくるのではなく、一つのトランスレータで、多くの機種に対応するのが本方式の利点である。その一方で、コード生成の対象となるCの処理系の性質への影響が残るのは中間

コードの方式と比較して不利な側面である。

エージェント言語で問題となるセキュリティや、環境の変化への対応は、翻訳処理をする段階で、ある程度の対処ができる。すなわち、オペレーティングシステムの機能を直接起動するようなことをせずに、実行時のライブラリを呼び出すようにするような処理ができる。このようにすれば、セキュリティや環境への対応は中間コードで実現するものと差はない。

エージェントに情報が有用かどうか判断させる状況では、エージェントの動作速度が問題となる。JIT が必要であるのも、エージェントといえども動作速度が重要視される場合があるからである。このような状況では、トランスレータを利用した方式が有理になる状況が考えられる。

11 関連研究

われわれの研究のきっかけは、文献 [2] で実現されたエージェントを高速に実行させたいというのが、ひとつの動機になっている。エージェントの応用やその実現方法は文献 [1][3] にあるが、トランスレータを利用した方法については知られていない。応用分野については、通常のエージェントと同様なものを想定している。C によるエージェントを実現する検討を行なったときに、文献 [4] を参考にしたが、異機種の間で機種独立性を求めて実現しているところが異なる。分散処理において、エージェント自身は移動しないで、その間の通信方法を整備してシステムを組み立てるアプローチもある（文献 [5][6]）。われわれは、エージェントが移動して、情報を集めるという処理に興味があり、プログラムの実行状態がネットワークを移動するということが本質と考えているが、この点で、これらの研究のアプローチとは異なる。われわれはすでに文献 [7] で基本の方式の検討結果を報告した。そして、文献 [8] でトランスレータのコード生成について発表したが、ここではオブジェクトの自発的な移動はできなかった。この二つの文献の方法にしたがって、自発的なエージェントの移動を実現したものが、本論文の内容である。文献 [9] では、トランスレータ方式で実行状態を退避回復できる方法で、ここに述べた方法よりもオーバヘッドが少ない方法を実現しているが、状態を機種非依存形式にして外部に転送することには言及していない。

12 今後の課題

現在は、トランスレータ方式でエージェント言語を実現できるというアイディアを実証的に示して、その問題点を確認しているところである。本格的な処理に耐えるシステムになっていない。このため、処理時間のオーバーヘッドを減らすようにトランスレータを改良することが次の目標である。また、ポインタの扱いについての検討、本格的なエージェント言語の設計と実装を次の作業の予定としている。

13 まとめ

トランスレータ方式でエージェント言語を実現する方法を論じた。そして、その方法にしたがって作成したトランスレータを実際に作成して、実現できることを確かめた。同時に、この方式のオーバヘッドについて論じた。

14 謝辞

この研究は NTT 基礎研究所とのネットワークシステムに関わる共同研究の成果であり、同時に IPA の独創的先進的情報技術の研究開発の過程で生まれた成果です。NTT と IPA の研究に関するサポートに感謝します。

15 参考文献

[1] Comm. ACM Special Issue on Intelligent Agents, Vol.37, No.7 (1994)

[2] Telescript Technology: The Foundation for the Electronic Marketplace, General Magic White Paper (1994)

[3] M.Genesereth and S.Ketchpel:"Software agents",Communications of the ACM, 37(7) (July 1994)

[4] 多田好克：”移植性のある C の継続ライブラリ”, 情報処理学会プログラミング - 言語・基礎・実践 - 研究会,18-14,PP.105-112,1994

[5] 地引昌弘 , 芦原栄登士 , 山下雄三 , 上田哲朗 , 大木敦雄 , 久野靖: "分散仮想マシンを用いたオブジェクト指向プログラミング環境" 情報処理学会プログラミング研究会研究報告書,Vol.96,No.107 pp.37-42(1996)

[6] 浦田泰裕 , 斎田明夫 , 田村直之 , 金田悠紀夫 , 川村尚生:"分散環境下におけるマルチエージェントシステム記述用言語" , 情報処理学会プログラミング研究会研究報告書 Vol.95,No.82 pp.153-158(1995)

[7] 梅村恭司 , 下川僚子 :"ネイティブコードのエージェントの実装方式の検討 "情報処理学会コンピュータシステムシンポジウム論文集 ,pp.59-64(1997)

[8] 下川僚子, 梅村恭司:"テキスト形式の Continuation の生成とそのプログラミングシステム", 情報処理学会プログラミングシンポジウム論文集、(1998)

[9] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa: "An efficient implementation scheme of concurrent object-oriented language on stock multicomputers," ACM SIGPLAN Symbosium on Principle & Practice of Parallel Programming(PPOPP) pp 218-228, May, 1993