トランスレータを利用した機種非依存な実行移送方式

下川僚子† 梅村恭司†

我々は，トランスレータを利用して機種非依存に実行状態を移送する方式を提案する．そしてこの方式を用いたトランスレータを利用するエージェント言語のプロトタイプを作成し，この方式の原理とコストを明らかにする．我々は，異なった機種間でのプロセス移動を可能とするために，プログラムの実行コンテクストがCのコードで処理できるような規約を設定し，その規約に従ったCのコードを生成する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コンテクストを機種独立のテキスト形式に変換でき，また生成された実行コンテクストを用いて実行を再開する機能を持つ．こ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．本方式では，単一のトランスレータで多くの機種間でプログラムを移送することができ，さらにそのプログラムはネイティブコードで動作する．

An Architecture Independent Method for Process Migration with Translator

RYOKO SHIMOKAWA† and KYOJI UMEMURA†

This article proposes a method to transfer a native code process in architecture independent way. This ability is useful to build agent systems. We have implemented a prototype of such agent language systems as a translator. This paper makes the method and the cost clear by this prototype. In order to realize migration among different architecture machines, we have defined constraints and developed a translator whose output satisfies the costraints. The output codes are able to generate an execution context in machine independent format, and able to resume the execution with the context. Though the program is executed in the native code, it can migrate among the machines whose archtecture are different. Moreover, we need only one translator for the different archetecture machines.

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．エージェントとは，分野により異なる意味を持っているが，ここで述べているエージェントとはコンピュータ間の通信において，あるコンピュータから他のコンピュータに対して，実行を依頼する手続きそのものを転送して処理する方式で，中間に立つソフトウェア機能のことである．この処理で重要なのは，この手続きは送り手のコンピュータで処理が行われている途中で送ることができ，受け手側のコンピュータでその処理の続きを行うことができることである．つまり，異なる機種の間でもプロセスを移送することができるということである．我々がこれから述べるエージェントはさらに異機種間で実行コンテクストを移動できる機能のことであり，この機能を持つ言語をエージェント言語と定義する．

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，その上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うえ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，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独立に言語を実現する方法である．この方法では，エージェントのコードが機種依存のネイティブコードで動作する．

我々は，エージェントの実行時間が移動時間よりもはるかに長い応用で有効であるようなトランスレータ方式のエージェント言語について検討し，そのプロトタイプを実現した．具体的には，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言語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作成した．我々の提案する方式では，機種ごとに異なったトランスレータを用いるのではなく，1

† 豊橋技術科学大学情報工学系
Department of Information and Computer Sciences, Toyohashi University of Technology

つのトランスレータで多種類の機種に対応することができる．その結果，適切に設定されたUnixで動作しているシステム上で，走行するコンピュータのアーキテクチャに依存せずにエージェントをネイティブコードで動作させることができるようになった．

本稿では，ネイティブコードで動作するエージェントの異機種間での移送方式について述べる．

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言語を選びトランスレータを作成した．ターゲット言語とは，我々の作成したトランスレータによって翻訳される結果の言語のことである．我々はCが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，これを選択した．実現した言語をP0と呼ぶことにし，方式を確認するために単純なものを設定した．データの型は整数として，配列，分岐，繰返し，再帰できる関数呼び出し，それに，実現した言語の移動先をマシン名で指定する基本機能を持つ．P0は一番単純なものとして表現したが，これらの機能だけでも想定している応用のプログラムを書くための表現力は持っている．また，機能を拡張することについては，通常の言語の機能の拡張方法と同様に行えばよい．

制御と呼び出しの情報を機種非依存で移動できるようにすることが，このトランスレータ方式で実現するうえで重要な項目である．

具体的には，P0はPascalに似た構文を持ち，migrate文を持つ言語である．**図1**に示すのが，単純なプログラム例である．7行目の`migrate`文は引数のマシン名のコンピュータシステムに移動して，プログラムを再実行する機能を実現している．このプログラムは，`i`の値を出力しインクリメントしていくプログラムで，`i`が2以上になると`avenue`というマシンに実行を移し，また`i`の値を出力するプログラムである．

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

図1 P0のサンプルプログラム
Fig. 1 Sample program in P0.

3. 実行コンテクストのテキスト化手法の提案

我々は機種に依存せずに実行コンテクストの移送を行うために，プログラムの実行コンテクストを機種独立なテキスト形式に変換する方式を考案した．実行コンテクストとはある時点でのプログラムの状態すべてのことであり，これを使うことによってプログラムを途中から実行することができるようなものである．実行コンテクストをテキスト形式に変換し移送することによって，機種に依存せずに実行コンテクストの移送を行っている．実行コンテクストはcall/return時のアドレスと変数の値からなる．実行コンテクストをテキスト化する際に特に問題となるのは，call/return時のアドレスをどのように表現するかということである．本章ではこのcall/return時のアドレスをテキスト化するための手法について述べる．変数の扱いについては，次章以降で述べる．

3.1 関数呼び出し

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，関数呼び出しの状態をテキストから復元することが問題となる．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，その値を直接操作することができないからである．

そこで，関数呼び出しは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，呼び出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．

翻訳結果のCの関数呼び出しは，呼び出したい関数から戻ってきた後に行う続きの仕事を関数として先にスタックにプッシュしてから，呼び出したい関数の仕事を開始することで実行する．呼び出した関数から戻ってくると，スタックから次に行う仕事の関数をポップしてその関数を呼び出す．これは，関数が終了したあとに行う仕事がスタックに保存されているというように言い換えることもできる．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，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こととした．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が可能となった．このスタックの実行ルーチンは**図2**のようなものである．つまり，実行を待つ仕事を積むスタックがあり，その先頭から1つ取り出しては呼び出すという作業で全体が動作して

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

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

図2 実行のメインルーチン
Fig. 2 Main loop of execution.

```
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3 関数ポインタから関数名への変換
Fig. 3 Translating function pointer into function name.

いる．これによって，関数呼び出しについてテキスト形式の実行コンテクストを生成できるようになる．

3.2 関数ポインタのテキストとの変換

戻りアドレスとは異なり，関数ポインタは操作できる．しかし，関数ポインタは機種ごとに異なった値となるため，機種に依存せずに関数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知っている．関数ポインタの格納には，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3のような構造のモジュールを作ればよい．この関数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，一致したら対応する関数の名前の文字列を求める処理である．この処理は，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り，実行中断時に実行コンテクストをテキスト形式に変換する際に使われる．

また，文字列から関数ポインタへの変換も図4のような関数で実現できる．この関数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り，実行再開時に呼び出される．

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可能であるが，作成

```
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4 関数名から関数ポインタへの変換
Fig. 4 Translating function name into function pointer.

したシステムでは図3，図4で示された実現方式を採用している．しかし改善する方法は明らかである．

4. コーディング規約

コーディング規約は，我々が実現した言語P0をC言語に変換する際の変換規則である．我々は，実行コンテクストをテキスト化するために，前の方針で述べたように変換後のC言語において，Cのスタックを実際には使わないようにする方針を採用している．そのために，関数呼び出し，分岐，ローカル変数については，そのままCに翻訳するのではなく，ある制限に従うような変形が必要である．この制限が本章で述べるコーディング規約であり，これに従ったコードに翻訳することにより，エージェントが正しく動作する．以下に具体的なコーディング規約について述べる．

4.1 関数呼び出し

Cのスタックを使わないようにコードを生成するという制限があるので，Cの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グローバルな変数用のスタックで管理することとした．実現した言語P0のレベルでは再帰呼び出しが可能となっているが，以下に述べるような手法で関数呼び出しを変換するため，それを実行するCのレベルでは，実際には再帰は行われず，関数を順次呼び出すような関数呼び出しだけで再帰と同じ動作をするように翻訳される．また，ここで述べた再帰プログラムとは手続き型言語で使用される再帰定義プログラムを想定している．すなわち，通常の手続き型言語で応用システムを組み立てることを想定しており，無限再帰のようなプログラムを処理することは想定していない．無限再帰を処理するには，それを繰返しとするような前処理を行うことになるが，この処理は実行コンテクストの処理とは直接の関係のないものである．

簡単な引き数のない関数呼び出しの処理について考える．図5のような`fn`が呼び出されているシーケン

```
--in P0--
 /* statementA */
fn();
 /* statementB */

--in C --
 /* statementA */
C[sp++]=&x0001; C[sp++]=&fn;
return;}
void x0001(void){
  /* statementB */
 }
```

図5 関数呼び出しの変換例
Fig. 5 A translated code for function call.

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

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

図6 分岐の変換例
Fig. 6 A translated code for branches.

スがあるとすると，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2つに分ける．そして，関数の呼び出し後戻った後に行う部分を独立の関数とする．呼び出しを行うコードは，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．すると規約に従って，呼び出したい関数が実行されそのあと呼び出し後のコードが実行される．

4.2 分　　岐

分岐は，そのまま変換することはできない．Cのコードでは前に述べたように関数呼び出しの前後で2つの関数に分かれる．すると，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とそこで別の関数に分かれる．するとループがうまく実現できない．

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こととした．その処理は図6のようになる．ラベルの場所に関数宣言を生成し，ラベルに入る直前で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．これが等価的に分岐の動作となる．

4.3 ローカル変数

P0の関数ローカル変数は，P0の関数が翻訳後では2つ以上のCの関数に分かれる可能性があるため，Cのローカル変数に翻訳できない．したがって，Cのグローバルな領域にとらなければならない．前に述べた関数のポインタからなるスタックとは別の変数用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．

4.4 ヒープ領域

我々の実現した言語では，ヒープ領域とポインタを実現していないが，実現したとしても翻訳方法に大きな変更はない．トランスレータであるので，実現言語でのポインタが翻訳結果のCでもポインタである必然性はない．したがって，ヒープ領域はグローバルな配列をとることで実現できる．ポインタは配列のインデックスで表現できるため，配列を用いてヒープと同様の処理をCで行わせるようにする．すなわち，言語の表現能力に，提案方式が制限を与えるものではない．

制限は，リソースの管理方式に現れる．グローバルな場所にデータを配置すると，実行コンテクストをテキスト化する際の配列の使用していない部分の転送や，コンパイル時に領域の上限が決まるという問題がある．これは，エージェントが要求するメモリリソース量があらかじめ知ることを要求するべきかどうかという問題にもかかわる．我々の方式でも，コンベンションをさらに設定すれば，リソースを可変にすることもできるが，論文で述べた範囲では，リソース量を固定するという制限を与える方式となっている．

4.5 関数ポインタ

我々の実現した言語では，関数をデータとして渡す機能を実現していないが，実現したとしても翻訳方法に大きな変更はない．データとして扱われる可能性のある関数をトランスレータが特定できれば，リターンアドレスの処理と同様に，その関数に対応するCでのアドレスをテキスト化して転送することができる．また，関数ポインタを実現機能として利用しているような言語拡張についても，実行移送の機能と衝突するものでない．

我々の述べた範囲の方式の制限は，関数の実体をダイナミックに生成するような機能が実現できないことである．我々の想定している応用は，このようなものを必要とはしていない．これを実現すべきかどうかについても，管理という立場から議論がなされるべきものであり，必須の機能とは考えられない．

万一，実現が必須である場合には，プログラム走行中にCのコードを生成し，それをコンパイルし，走行を続行したまま取り込むようにする方法も知られている．このとき，新たに生成するCのコードを注意深く

生成することで実現できるが，これについては，本論文では扱っていない．

4.6 実行時ライブラリ

本格的なエージェント言語を実現するには，言語機能のみならずネットワーク機能やリソース管理の機能などを司るライブラリが必要である．P0では，単純な入力出力できる以上のものは実現していない．エージェント言語のライブラリは通常の言語と異なる要求事項があり，多くの検討課題がある．我々の手法は中間言語を使用した手法と比較したとき，この問題に対して寄与してはいないが，問題を複雑にもしていない．したがって，特に主張する項目はなく，これについては，本論文では扱っていない．

5. トランスレータ

我々の作成したトランスレータは，実現した言語P0をC言語に翻訳するものであり，翻訳されたCのプログラムは上に述べた規約を満たすものである．さらに翻訳されたCのプログラムには，プログラムの実行コンテクストをテキストに変換して生成する機能が含まれている．このトランスレータは，C言語で記述したもので，約1600行のプログラムである．これは実際に，正しいP0のプログラムをすべて処理できていることが確認できている．

P0の簡単なサンプルプログラムとその翻訳結果の一部を**図7**に示す．翻訳にはソースとの対応を見やすくするためにコメントを追加したが，それ以外はトランスレータが出力したコードそのものである．

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．関数呼び出しの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．

6. コードの実行

翻訳されたCのプログラムを実行すると，プログラムの処理が図2のループで開始される．そして，実行を移送する命令が実行されたところで，プログラムの実行は中断され，プログラムは転送先で途中から再実行される．この実行が中断された時点で実行コンテクストをテキスト化したファイルが生成される．これがプログラムの実行を転送する命令が実行された時点でのプログラムの実行途中状態である．

このエージェントの移送の際には，トランスレータによって生成されたCのプログラムと，テキスト化

----サンプルプログラム (P0)-----

```
 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.
```

----翻訳結果 (C)-----

```
static void fn(void)
{
        sp += 2;
        r0 = stack[1];   /*from line 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 line 8*/
        stack[sp] = r0; sp++;/*Push r0*/
        sp--; r0 = stack[sp];/*Pop  r0*/
        stack[1] = r0;
        stack[sp] = fp - stack; sp++;
        r0 = 3;          /*from line 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;
        fp = &stack[stack[sp]];
        r0 = stack[0];  /*from line 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
Fig. 7 Sample program and its translated result.

された実行コンテクストのファイルが転送される．そしてその転送先でCのプログラムを実行することにより，プログラムが転送先で途中から再実行される．この実行コンテクストのファイルは実行を移送する命令が実行されたときのプログラムの実行途中状態であり，このファイルとCのプログラムは機種に依存しないようにコードを生成している．

テキスト化された実行コンテクストのサンプルを**図8**に示す．実行状態のサンプルは，最初のコラムだけが使われる．空白から後は，デバッグを助けるた

```
2               : 関数スタックのサイズ
terminate       : 関数スタック [0] の内容
new_function0   : 関数スタック [1] の内容
5               : r0
1               : r1
31              : r2
1               : ローカル変数のポインタ
3               : 変数スタックのサイズ
10              : 変数スタック [0] の内容
3               : 変数スタック [1] の内容
0               : 変数スタック [2] の内容
2               : 変数スタック [3] の内容
```

図8 実行コンテクストのサンプル
Fig. 8 Sample of program context.

めの情報である．たとえば2行目では，最初のコラムの terminate だけが使われ，関数名と関数ポインタの変換は前に述べたようなモジュールで処理される．r0，r1，r2は，式の計算に使われる途中情報である．

現在のシステムで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，タイプごとに場所を特定できるようにコード生成をし，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く，これについては文献18)の方法が利用できる．

以上のように，エージェントを機種独立に転送し，実行している．

7. 実行時間

P0の実行では，P0の関数呼び出しの前後でCの関数を分解することや，ローカル変数をグローバル変数として扱うことで，オーバヘッドが生じる．

オーバヘッドのある条件を計測するために，再帰呼び出しを約100万回行うプログラム，1万個のデータでのソートプログラム，100万個のデータのサーチプログラムについて実行時間を計測した．これらは典型的な処理の代表として選択した．そして，それぞれCのプログラム，トランスレータを使用した場合のP0のプログラムと，コーディング規約に従うようにハンドコーディングしたCのプログラムについて実行時間を計測した（**表1**）．Cのプログラムはそれぞれの条件のプログラムをCで作成したもので，P0のプログラムはCで作成したものと同じプログラムをP0で作成し，トランスレータを使用し，Cのプログラムに翻訳し実行した．ハンドコーディングしたCのプログラムは，トランスレータで翻訳したCのプログラムを手である程度最適化したものである．これは，ハンドコーディングではあるが，機械的に生成できることを考慮したものである．ソートプログラムについて，P0のプログラムを**図9**に，ハンドコーディングしたCのプログラムを**図10**に示す．

表1 実行時間
Table 1 Execution time.

	C	C（ハンドコーディング）	P0（トランスレータを利用）
再帰呼び出し	0.9秒	4.8秒 (5.3倍)	9.1秒 (10.1倍)
ソート (10000)	29.7秒	93.6秒 (3.2倍)	323.5秒 (10.9倍)
サーチ (1000000)	0.6秒	2.0秒 (3.3倍)	7.2秒 (12倍)

※（）内の倍率はCのプログラムとの実行時間の比率
実行環境：Sparc Station5，110 MHz，256 Mbyte

```
program test;
var array[10000],ans,k;

procedure sort();
var i,j,max,jmax;
begin
   i := 0;
   while i < 10000 do
   begin
     max := array[i];
     jmax := i;
     j := i+1;
     while j < 10000 do
       begin
        if array[j] > max then
          begin max := array[j];
            jmax := j
          end;
        array[jmax]:=array[i];
        array[i]:=max;
        j := j +1
       end;
     i := i + 1
   end
end;

begin
   k := 0;
   while k < 10000 do
     begin
       array[k] := k;
       k := k + 1
     end;
   sort()
end.
```

図9 P0のプログラム
Fig. 9 Sample of P0 program.

その結果，プログラムの実行時間を比較すると，今回作成したトランスレータを使用すると10～15倍になり，実行速度は低下している．

このオーバヘッドは，現在のトランスレータの実装上の問題であって，この方式の問題ではない．これはハンドコーディングしたプログラムの実行時間を見れば明らかである．また，これはオーバヘッドの顕著な例を示したものであり，最悪な場合の例である．原理として，内部に関数呼び出しを含まない場合は，オー

```
static void sort(void)
{
      sp += 5;
      fp[1] = 1;
      PUSH(new_function0);
      return;
}
static void new_function0(void){
      if(fp[1] >= 10000){return;}
      fp[3] = stack[1+fp[1]];
      fp[4] = fp[1];
      fp[2] = fp[1]+1;
      PUSH(new_function2);
      return;
}
static void new_function2(void){
      if(fp[2] >= 10000)
        {PUSH(new_function3); return;}
      if(stack[1+fp[2]] <= fp[3])
        {PUSH(new_function4); return;}
      fp[3] = stack[1+fp[2]];
      fp[4] = fp[2];
      PUSH(new_function4);
      return;
}

static void new_function4(void){
      stack[1+fp[4]] = stack[1+fp[1]];
      stack[1+fp[1]] = fp[3];
      fp[2]++;
      PUSH(new_function2);
      return;
}
static void new_function3(void){
      fp[1]++;
      PUSH(new_function0);
      return;
}

static void test(void)
  {
      sp += 10003;
      stack[10002] = 0;
      PUSH(new_function6);
      return;
  }
static void new_function6(void){
      if(stack[10002] >= 10000)
        {PUSH(new_function7); return;}
      stack[1+stack[10002]] = stack[10002];
      stack[10002]++;
      PUSH(new_function6);
      return;
}
static void new_function7(void){
      stack[sp] = fp - stack;
      fp = &stack[sp];
      PUSH(sort);return;
}
```

図10 ハンドコーディングのCプログラム
Fig. 10 Sample of handcorded C program.

バヘッドは圧縮できる．

また，図10を見ると，実現したトランスレータの最適化はまだ不十分であることが分かる．さらに以下のようなことを考慮するとさらに最適化が可能である．

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，内部に関数呼び出しを含まない場合を検出することで，性能の低下をおさ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び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，さらにオーバ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圧縮でき，呼び出しが主体のプログラムにおいても3倍程度でおさ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8)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，呼び出しを行わない場合には，原理としてオーバヘッドは存在しない．このスタック操作は関数呼び出しに付随するので，この操作も関数呼び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が想定できるが，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の実現方法は明らかであり，これは通常のコンパイラの最適化と同様の方法が使用できる．

また，一般の方式と我々の方式との移動コストの差は翻訳結果の転送時間+コンパイル時間である．つまり，通常の実現方法よりも移動コストが高い．いいかえれば，エージェントの実行時間の圧縮と機種非依存性を実現するためのトレードオフとして，移動コストが存在する．データの転送時間とコンパイル時間については環境によって異なり，エージェントの速度への要求も異なる．したがって，ここが我々の方式が有効であるかどうかの判断となる．

我々はエージェントの実行時間がコンパイル時間が問題にならないように長い実行時間の応用を想定している．特に我々は検索にかなりの時間を要するような情報検索の分野で，データのあるマシン間を移動して検索していくようなシステムに応用していくことを考えている．この場合では，本方式においての移動コストであるコンパイル時間が問題になるようなことは

ない．

8. 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，本稿では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．我々の提案した方式は実行効率を除いては中間コードを設定して実現する方式と本質的な差はないと考えている．そのうえで，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と考えている．

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うものと比較して，JITを機種ごとに作製するのではなく，1つのトランスレータで，多くの機種に対応すること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である．この2つは応用によってどちらが優先されるかはケースバイケースであり，本稿の方法が有利なこともある．

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を行うようにすればよい．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本質の差はなく，比較対象である中間言語方式に比べ劣っていることはないと考えている．

OSのシステムコールやAPIの相違をどのように吸収するかという問題については，我々の方式にはOSのシステムコールやAPIの相違を吸収することに寄与するものではないが，この方式を使ったからといって問題を複雑にしたわけではない．対比の対象の中間コードと比較するなら，実行時ライブラリを整備することで同様なレベルには解決できると考えられる．

エージェントに情報が有用かどうか判断させることが必要な応用では，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，中間コード方式よりも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．

9. 関連研究

我々の研究は，文献10)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，1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9), 11)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ったときに，機種非依存でCでの継続を入手する方法があるということで文献12)を参考にした．しかしながら，この文献では継続情報はプログラムの外部に取り出すことができない．したがって，この文献の方法では異機種間の転送には対応できない．継続そのものを異機種間で移送できる方式とは異なるものである．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文献13), 14)のようなアプローチもある．我々は，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．文献15)は方式の記述でありシステムの実現の情報はなく，文献16)ではトランスレータのコード生成について記述されているが，ここではオブジェクトの自発的な移動の情報はない．この2つの文献の方法に従って，自発的なエージェントの移動を実現したものが，文献17)であるが，そのオーバヘッドについて論じられていない．本稿はオーバヘッドに関する評価を行ったものである．

文献1)と3)は，分散オブジェクト指向言語システムのアイディアが提案されており，エージェント言語のもととなったものである．ここで分散されるものは，内部にコンテキストを持たないものであった．文献2)は，分散オペレーティングシステムとして，プロセスを移送するという実験が試みられ，いろいろな問題が明らかになった．この研究では異なるアーキテクチャでのプロセス移送までは述べられていない．文献4)において，異なるアーキテクチャでのプロセス移送の問題が提起され，コンパイラの助けを借りてプロセスを移送するアイディアが述べられている．しかしながら，この文献では実際の実現がなされておらず，原理的な可能性を述べたものとなっている．文献7)において，任意のマシン語を自らのマシン語として実行できるアーキテクチャのマシンを利用して，さまざまなプロセスを移送できるアイディアが述べられている．特殊なハードウェアを利用して，任意の仮想マシンを実マシンにするというアプローチである．我々のアプローチは特殊ハードウェアを必要としないで，仮想マシンを高級言語のレベルに求め，その仮想マシンは実

マシンとして実行できることになる．仮想マシンを実マシンとできるという意味で，問題意識を共通にする研究である．文献8)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．文献18)でも異機種へのプロセスを転送できる方法をコンパイラと協調して実現している．この方法では各アーキテクチャに別々のコンパイラを作成することが必要なのに対し，我々のシステムは1つのトランスレータで多くのアーキテクチャに対応する．

文献19)はJavaにおける実行移送の可能なコードの規約を述べ，同一の仮想機械を持つ場合でも実行移送にはプログラムの構造上の規約が現れる例を報告している．文献20)では，エージェント言語には通信，可搬性，管理と制御が必要であると述べられているが，通信，管理と制御は実行時ライブラリで対応でき，我々はその中で可搬性を言語実行メカニズムの本質と考えた．文献21)には，既存の分散オブジェクト指向でもエージェントの移送が問題となっていることが示されている．文献22)ではJava上でAgent Spaceというフレームワークが作られている．また，同じくJava上でμCODE（文献23)）などのシステムが作られている．我々はこのようなものをJavaではなくCで実現した．

10. 今後の課題

本稿では，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し，その問題点を確認した．実現したプロトタイプの生成するコードは，まだ十分でない．既存のコンパイラ技術を利用して処理時間のオーバ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今後の課題である．

11. おわりに

我々は，エージェントの実行時間が移動時間よりもはるかに長いような応用を想定して，プログラムを異機種間で移送する方法の1つを提案した．我々の方式は，エージェント記述言語をトランスレータで一般的に広く使われているC言語に翻訳する．さらに実行コンテクストを機種非依存なテキスト形式にする．これは，トランスレータによって翻訳されたC言語のプログラムによって処理される．本稿で提案した方式では，単一のトランスレータで多くの機種間でプログラムを移送でき，プログラムはネイティブコードで動作することが可能となっている．

そして，我々はその方法に従ったエージェント言語のプロトタイプを設定し，トランスレータを実際に作成し実現できることを確かめ，エージェント言語を実現するための原理とコストを明らかにした．

謝辞 この研究は住友電工，NTT光ネット研究所とのネットワークシステムにかかわる共同研究の成果であり，同時に住友電工との共同プロジェクトとしての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ある．住友電工，NTTとIPAの研究に関するサポートに感謝します．

参考文献

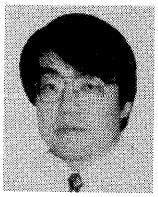
1) Jul, E., Levy, H., Hutchinson, N. and Black, A.: Fine-grained mobility in the Emerald system, *ACM Trans. Computer Systems* (1988).
2) Cheriton, D.R.: The V distributed system, *Comm. ACM*, Vol.31, No.3, pp.314–333 (1988).
3) Raj, R.K., Tempero, E., Levy, H.M., Black, A.P., Hutchinson, N.C. and Jul, E.: Emerald: A general-purpose programming language, Software - Practice and Experience, Vol.21, No.1, pp.91–118 (1991).
4) Theimer, M.M. and Hayes, B.: Heterogeneous process migraion by recompilation, *11th International Conference on Distributed Computing Systems* (May 1991).
5) 森山茂男，多田好克：利用者レベルで実現したプロセス移送ライブラリ，92-DPS-58, pp.41–47 (1991).
6) 長澤育範，相田 仁，斎藤忠夫：異機種間プロセス移送メカニズムの試作，91-OS-51, pp.173–180 (1992).
7) Silberman, G.M. and Ebcioglu, K.: *An Architectual Framework for Supporting Heterogeneous InstructionSet Architectures*, pp.39–56, IEEE (1993).
8) Taura, K., Matsuoka, S. and Yonezawa, A.: 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).
9) *Comm. ACM Special Issue on Intelligent Agents*, Vol.37, No.7 (1994).
10) Telescript Technology: The Foundation for the Electronic Marketplace, General Magic White Paper (1994).
11) Genesereth, M. and Ketchpel, S.: Software agents, *Comm. ACM*, Vol.37, No.7 (1994).
12) 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).

13) 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).
14) 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).
15) 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).
16) 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).
17) 下川僚子，山本英雄，梅村恭司：トランスレータ方式のエージェントの実装，情報処理学会研究報告，98-OS-78, pp.31–38 (1998).
18) Smith, P. and Hutchinson, N.C.: Heterogeneous Process Migration: The Tui System, *Software Practice and Experience*, Vol.28, No.6, pp.611–639 (1998).
19) Funfrocken, S.: Transparent Migration of Java-Based Mobile Agents, *WorkShop on Mobile Agentss98*, pp.26–37 (1998).
20) Aridor, Y. and Oshima, M.: Infrastructure for Mobile Agents: Reuirements and Design, *WorkShop on Mobile Agentss98*, pp.38–49 (1998).
21) Milojicic, D., et al.: MASIF The OMG Mobile Agent System Interoperability, *WorkShop on Mobile Agentss98*, pp.50–67 (1998).
22) Silva, A., da Silva, M.M. and Delgado, J.: An Overview of AgentSpace: A Next-Generation Mobile Agent System, *WorkShop on Mobile Agentss98*, pp.148–159 (1998).
23) Picco, G.P.: μCODE: A Lightweight and Flexible Mobile Code Toolkit, *WorkShop on Mobile Agentss98*, pp.160–171 (1998).

(平成 10 年 12 月 1 日受付)
(平成 11 年 4 月 1 日採録)

下川　僚子（学生会員）

1975 年生．平成 10 年豊橋技術科学大学工学部情報工学課程卒業．現在，同大学大学院工学研究科修士課程情報工学専攻に在学中．



梅村　恭司（正会員）

1959 年生．1983 年東京大学大学院工学系研究科情報工学専攻修士課程修了．同年，日本電信電話公社電気通信研究所入所．1995 年豊橋技術科学大学工学部情報工学系助教授，現在に至る．博士（工学），システムプログラム，記号処理の研究に従事．ACM，ソフトウェア科学会，電子情報通信学会，計量国語学会各会員．