再帰的ブロック構造を持つ並列プログラムに対する可逆実行環境

池田 崇志[1,a)]　結縁 祥治[1,b)]

概要：本論文では，並列に実行されるブロック構造を持つプログラムの実行を解析することを目的として可逆実行環境の実装を示す．並列プログラムを抽象機械の 3 番地コードに変換して実行する．順方向の実行時は逆向き実行に必要な情報をスタックに保存し，その実行を逆向きに辿る実行環境を実装する．この実行環境では，順方向の抽象命令を逆方向の抽象命令に一対一に変換することで逆向き実行を実現する．
抽象命令の変換による実行環境 [T.Ikeda and S.Yuen, 2020] では，並列実行において抽象機械を Python の multiprocessing モジュールでフォークする抽象命令を実装し，順方向および逆方向において並列実行する実行環境を示した．
ここでは，実際的なプログラムの構文要素として，ブロック構造，手続き呼び出し，関数呼び出しを含むように拡張した．Hoey らの手法に従って変数のスコープを扱うために，各ブロックに名前を付け，参照情報をパスとして表し，局所変数を実現する．本研究で新たに提案する方法として抽象命令生成時に作成する並列ブロックの開始及び終了番地を記録したテーブルを用いて並列ブロックを起動することにより順方向，逆方向ともに並列の入れ子構造を実現する．これらの実現手法によって，ブロック構造を持つプログラミング言語に対して単純な抽象機械の実行メカニズムによって逆方向実行が可能となることを示し，並列プログラムのデバッグのための基盤として提案する．

キーワード：a

A reversible runtime for parallel programs with recursive blocks

Takashi Ikeda[1,a)]　Shoji Yuen[1,b)]

Abstract: This paper presents a reversible runtime of simple parallel programs with blocks. A program is translated into a sequence of three-address abstract machine instructions and abstract machines running in parallel execute the instructions. The runtime stores the information of variable updates and program counter jumps associated with process identifies on stacks in the forward execution. In the backward execution, the abstract instructions for forward execution are converted to reverse abstract instructions one-to-one.
In our previous work, we presented a runtime for parallel programs with flat-fixed structures. The runtime executes multiple abstract machines using the multiprocessing module of Python.
This work extends the runtime for practical language features, including blocks, procedure-call, and function-call. To deal with the scope of variables in blocks, we assign the path information with block names following Hoey et.al. Besides variable paths, the runtime records the invocation history of parallel blocks as a table to reverse the invocation of parallel blocks. We realize parallel nested structures in both directions. We illustrate that executing abstract machines makes bi-directional execution simple even with the recursive structure of blocks. We propose them as a foundation for behavioural analysis such as debugging.

Keywords: a

[1] 名古屋大学情報学研究科
Graduate school of imformatics, Nagoya University,
Furo-cho, Chikusa-ward, Nagoya-city, 464-8601
[a)] tikeda@sqlab.jp
[b)] yuen@sqlab.jp

$$
\begin{array}{l}
P ::= \epsilon \mid S \mid P; P \mid P \ par \ P \\
S ::= skip \mid X = E \ pa \mid if \ In \ B \ then \ P \ else \ Q \ end \ pa \mid \\
\quad while \ Wn \ B \ do \ P \ end \ pa \mid \\
\quad begin \ Bn \ BB \ end \mid call \ Cn \ n \ pa \mid \\
\quad runc \ Cn \ P \ end \\
BB ::= DV \ DP \ P; \ RP \ RV \\
E ::= X \mid n \mid (E) \mid E \ Op \ E \\
B ::= T \mid F \mid \neg B \mid (B) \mid E == E \mid E > E \mid B \wedge B \\
DV ::= \epsilon \mid var \ X = E \ pa; \ DV \\
DP ::= \epsilon \mid proc \ Pn \ n \ is \ P \ end \ pa; \ DP \\
RV ::= \epsilon \mid remove \ X = E \ pa; \ RV \\
RP ::= \epsilon \mid remove \ Pn \ n \ is \ P \ end \ pa; \ RP
\end{array}
$$

図 1　対象プログラムの定義
Fig. 1　definition of language

$$
\begin{array}{l}
ann(\epsilon) = \epsilon \\
ann(S; P) = a(S); ann(P) \\
ann(P \ par \ Q) = ann(P) \ par \ ann(Q) \\
a(skip) = skip \ I \\
a(X = e \ pa) = \ X = e \ (pa, A) \\
a(if \ In \ n \ then \ P \ else \ Q \ end \ pa) = \\
\quad if \ In \ b \ then \ ann(P) \ else \ ann(Q) \ end \ (pa, A) \\
a(while \ Wn \ b \ do \ P \ end \ pa) = \\
\quad while \ Wn \ b \ do \ ann(P) \ end \ (pa, A) \\
a(begin \ Bn \ P \ end) = begin \ Bn \ ann(P) \ end \\
a(var \ X = E \ pa) = var \ X = E \ (pa, A) \\
a(proc \ Pn \ n \ is \ P \ end \ pa) = proc \ Pn \ n \ is \ ann(P) \ end \ (pa, A) \\
a(call \ Cn \ n \ pa) = call \ Cn \ n \ (pa, A) \\
a(runc \ Cn \ P \ end) = runc \ Cn \ AP \ end \ A \\
a(remove \ X = E \ pa) = remove \ X = E \ (pa, A) \\
a(remove \ Pn \ n \ is \ P \ end \ pa) = remove \ Pn \ n \ is \ ann(P) \ end \ (pa, A)
\end{array}
$$

図 2　Annotated program の変換規則
Fig. 2　Annotation function

1. はじめに

2. 関連研究

本章では並列プログラムに対する逆方向実行について，先行研究によって提案されている手法について説明する．先行研究によって提案されている手法では，while ループや if 文，手続き呼び出しのブロックそして並列ブロックを持つような単純なプログラムを対象としている．この対象プログラムに対して逆方向実行に必要な情報を残すための処理を行う．この処理を Annotation と呼び対象プログラムに Annotation を適用し逆方向実行に必要な情報を残せる形式にしたプログラムを Annotated プログラムと呼ぶ．Annotated プログラムを実行することによってこのプログラム自体に逆方向実行に必要な情報が書き込まれる．実行された Annotated プログラムを if 文や while 文，手続き呼び出しそして並列ブロックの構造は保持したままでその他の記述順を反転させることで Inverted プログラムが生成され，これを実行することで対象プログラムの実行を逆方向に辿ることができる．

2.1 対象プログラムの定義

逆方向実行を行う対象のプログラムに対する定義を示す．対象とするプログラムは while ループ，if 文，手続き呼び出しのブロックそして並列ブロックを持つようなプログラムであり**図 1** のように定義する．

2.2 Annotated プログラム

図 1 で定義された対象プログラムに対して Annotation を行い，Annotated プログラムを生成しそれを実行する．

図 2 で Annotation の適用規則を示す．対象プログラム P から Annotated プログラム AP への変換は変換規則 $a(), ann()$ に則って行われる．

図 3 のプログラムから**図 4** のプログラムが Annotated プログラムの変換規則に則って生成される．この時，手続き p1 内のステートメントはブロック b1 内のブロック b2 に存在しているのでこれらのステートメントのパスは b2*b1 に変換する．Annotated プログラムにはそれぞれのステートメントに識別子を書き込むためにスタック A を書き加えている．

```
begin b1
   proc p1 fib is
   begin b2
      var T = 0 b2;
      if i1 (N-2 > 0) then
         T = F + S b2;
         F = S b2;
         S = T b2;
         N = N - 1 b2;
       call c2 fib b2;
    end b2
    remove T = 0 b2;
  end
  b1
  call c1fib is P b1;
  remove p1 fib is P b1;
end
```

図 3　対象プログラム
Fig. 3　Original Program

```
begin b1
   proc p1 fib is
   begin b2
      var T = 0 (b2*b1,A);
      if i1 (N-2 > 0) then
         T = F + S (b2*b1,A);
         F = S (b2*b1,A);
         S = T (b2*b1,A);
         N = N - 1 (b2*b1,A);
       call c2 fib (b2*b1,A);
    end (b2*b1,A)
    remove T = 0 (b2*b1,A);
  end
  (b1,A)
  call c1fib is P (b1,A);
  remove p1 fib is P (b1,A);
end
```

図 4　Annotated プログラム
Fig. 4　Annotated Program

図 4 を実行すると**図 5** のようにプログラム自体に Annotation 及びパスが書き込まれる．図 5 では 2 回目の手続き呼び出しを行ったものであり手続き呼び出しのブロック自体をコピーしそこに Annotation とパスを書き込んでいる．

図 6 に実行された Annotated Program から Inverted Program への変換規則を示す．図 6 の AP や AQ はプログラム P，Q に Annotated Program への変換規則を適用したものである．AS はステートメント S に Annotated Program への変換規則を適用したものである．実行されたパス及び Annotation に逆向き実行に必要な情報が残されておりこの規則に則って作成された Inverted Program を

```
begin c1:c2:b2
   var T = 0 (c1:c2:b2*b1,[7]);
   if c1:c2:i1 (N-2 > 0) then
     T = F + S (c1:c2:b2*b1,[8]);
     F = S (c1:c2:b2*b1,[9]);
     S = T (c1:c2:b2*b1,[10]);
     N = N - 1 (c1:c2:b2*b1,[11]);
     call c2 fib (c1:c2:b2*b1,[15]);
  end (c1:c2:b2*b1,[16])
  remove T = 0 (c1:c2:b2*b1,[17]);
end
```

図 5 実行された Annotated プログラム
（2 回目の手続き呼び出しの部分）

Fig. 5 Executed Annotated Program
(part of second procedure call)

$$
\begin{aligned}
&inv(\epsilon) = \epsilon \\
&inv(AS; AP) = inv(AP); i(AS) \\
&inv(AP\ par\ AQ) = inv(AP)\ par\ inv(AQ) \\
&i(skip) = skip\ I \\
&i(X = e\ pa) = \ X = e\ (pa, A) \\
&i(if\ In\ n\ then\ AP\ else\ AQ\ end\ pa) = \\
&\quad if\ In\ b\ then\ inv(AP)\ else\ inv(AQ)\ end\ (pa, A) \\
&i(while\ Wn\ b\ do\ AP\ end\ pa) = \\
&\quad while\ Wn\ b\ do\ inv(AP)\ end\ (pa, A) \\
&i(begin\ Bn\ AP\ end) = begin\ Bn\ inv(AP)\ end \\
&i(var\ X = E\ pa) = var\ X = E\ (pa, A) \\
&i(proc\ Pn\ n\ is\ AP\ end\ pa) = proc\ Pn\ n\ is\ inv(AP)\ end\ (pa, A) \\
&i(call\ Cn\ n\ pa) = call\ Cn\ n\ (pa, A) \\
&i(remove\ X = E\ pa) = remove\ X = E\ (pa, A) \\
&i(remove\ Pn\ n\ is\ AP\ end\ pa) = remove\ Pn\ n\ is\ inv(AP)\ end\ (pa, A)
\end{aligned}
$$

図 6 Inverted program の変換規則

Fig. 6 Inversion function

```
begin c1:c2:b2
   var T = 0 (c1:c2:b2*b1,[17]);
   if c1:c2:i1 (N-2 > 0) then
     call c2 fib (c1:c2:b2*b1,[15]);
     N = N - 1 (c1:c2:b2*b1,[11]);
     S = T (c1:c2:b2*b1,[10]);
     F = S (c1:c2:b2*b1,[9]);
     T = F + S (c1:c2:b2*b1,[8]);
     F = S (c1:c2:b2*b1,[9]);
  end (c1:c2:b2*b1,[16])
  remove T = 0 (c1:c2:b2*b1,[7]);
end
```

図 7 Inverted プログラム
（2 回目の手続き呼び出しの部分）

Fig. 7 Inverted Program
(part of second procedure call)

実行することで Annotated Program の実行を逆順に辿る実行を行う．

図 5 の実行された Annotated プログラムを図 6 の変換規則に則って Inverted Program へ変換したものが図 **??**である．Annotation の数字がそのステートメントが実行された順番を表していて Annotation に書かれている最大の値から始めて一つずつ Annotation に書かれている数字を

$$
\begin{aligned}
&P ::= begin\ bn\ BB\ end \\
&\quad \mid \text{par an } P(\parallel P)^{+}\ rap \\
&\quad \mid S \\
&BB\ ::=\ DV\ DP\ DF\ P(;P)^{+}\ RV \\
&S\ ::=\ skip \mid X = E \mid if\ C\ then\ P\ else\ P\ fi \mid \\
&\quad while\ wn\ C\ do\ P\ od \mid call\ cn\ a(X?) \\
&DV\ ::=\ (var\ X;)^{*} \\
&DP\ ::=\ (proc\ pn\ a(X?)\ is\ P\ end)^{*} \\
&DF\ ::=\ (func\ fn\ b(X?)\ is\ P\ return)^{*} \\
&RV\ ::=\ (remove\ X;)^{*} \\
&E\ ::=\ X \mid n \mid (E) \mid E\ op\ E \mid \{cn\ b(X?)\} \\
&C\ ::=\ B \mid C\ \&\&\ C \mid not\ C \mid (C) \\
&B\ ::=\ E\ ==\ E \mid E\ >\ E
\end{aligned}
$$

(`X`: 変数，`n`: 整数，`a`: 手続き名，`b`: 関数名，$op : \{+, -, \times\}$)

図 8 対象言語の定義

Fig. 8 definition of language

遡っていくことで Annotated Program で行った順方向の実行を逆方向に辿る実行を行うことができる．

3. 並列プログラミング言語

本章では本論文において対象とするプログラミング言語についてその定義と仕様，および抽象機械命令の仕様について説明する．対象とするプログラミング言語は while ループや if 文，手続き呼び出しのブロック，関数呼び出しのブロック，および並列ブロックを持つプログラミング言語である．このプログラミング言語に従って作成したプログラムを抽象機械命令に変換することで抽象機械によって実行する．プログラミング言語の定義として手続き呼び出しのブロックなどの各ブロックに手続き名などとは別にブロックの名前を付ける．このブロックの名前を繋げたものを参照情報を表すパスとすることで局所変数を実現している．

このプログラムにおいて並列ブロックは par から始まり，各ブロックを || の記号で区切り，rap で終わるように記述する．並列ブロックを含むプログラムから抽象機会命令に変換する際にそれぞれの並列ブロックの開始番地と終了番地を記録したテーブルを作成する．そのテーブルを参照し並列ブロックを起動することで順方向，逆方向ともに並列の入れ子構造を実現している．

抽象機械命令について，対象言語に従って作成されたプログラムをコンパイラによってそれぞれのステートメントを抽象機械命令に変換し順方向の抽象機械のバイトコードを作成する．逆方向のバイトコードは順方向のバイトコードの抽象機械命令を一対一で変換し順序を反転させることで作成する．コンパイラは JavaCC を用いて作成した．

3.1 対象言語の定義

対象言語の定義を図 **7** に示す．対象言語の定義は拡張

bnf によって記述した．bn, an, wn, pn, fn, cn はそれぞれのブロックの名前を示している．それぞれ n は整数値を表し b1,b2,... のように整数値の部分が被らないように振り分ける．DV は変数の宣言，DP は手続きの宣言，DF は関数の宣言，RV は変数の解放を行うステートメントを示している．あるブロック内で宣言された変数はそのブロック内で必ず宣言した順番とは逆の順番で変数の解放を行うステートメントを記述する必要がある．

手続き呼び出しについて，手続きの宣言は proc から始まり，end で終わるように記述する．手続きの引数は変数一つのみとし，値を返すことはしない．call ステートメントを記述することで宣言された手続きを呼び出し，その手続きを実行する．

関数呼び出しについて，関数の宣言は func から始まり，return で終わるように記述する．関数の引数は変数一つのみとし，値を返す．関数内では関数の名前を変数として扱い演算を行うことができ，return を行う時点でのその変数の値を返り値として扱う．関数の呼び出しは呼び出しブロック名と関数名および引数を {} で囲って記述する．変数や整数と同様に演算の右辺に記述することができる．

図 8 は図 7 の定義に従って作成したプログラム例である．ブロック b1 内で変数の宣言，手続きの airline の宣言を行い，メインの処理として変数の値割り当て，手続き airline の呼び出しを行い，最後に最初に宣言した変数の解放を行うプログラムである．

```
begin b1
    var seats;
    var agent1;
    var agent2;
    proc p1 airline() is
        par a1
            begin b2
                while w1 (agent1==1) do
                    if (seats>0) then
                        seats=seats-1
                    else
                        agent1=0
                    fi
                od
            end
        || begin b3
                while w2 (agent2==1) do
                    if (seats>0) then
                        seats=seats-1
                    else
                        agent2=0
                    fi
                od
            end
        rap
    end
    seats=3;
    agent1=1;
    agent2=1;
    call c1 airline()
    remove agent2;
    remove agent1;
    remove seats;
end
```

図 9 プログラム例
Fig. 9 sample program

3.2 可逆抽象機械

本研究では，図 7 の定義に従って作成された図 8 のようなプログラムを各ステートメントをそれぞれ抽象機械命令に変換し抽象機械のバイトコードを作成する．実装としては対象のプログラムをコンパイラによって変換しバイトコードを作成する．このバイトコードをプロセスにおいて固有の演算用のスタックとプロセス間で共有の共有変数スタックを持つ抽象機械によって実行することでプログラムに書かれたステートメントを順方向に実行する．順方向実行時に逆向き実行に必要な情報を保存する．これについては 4 章で詳しく示す．

逆方向のバイトコードは順方向のバイトコードの抽象命令をそれぞれ対応した抽象命令に一対一で変換し順序を反転させることで作成する．この逆方向のバイトコードと順方向実行時に保存した逆方向実行に必要な情報を使って抽象機械によって実行することで順方向の実行を逆方向に辿る実行をする．

3.2.1 順方向抽象機械命令

順方向の抽象命令セットを以下に示す．

[命令，被演算子]

・[ipush，即値]

ipush はスタックのトップに被演算子の即値をプッシュする．

・[load，変数番地]

load は被演算子の変数番地の値を読み出し，その値をスタックトップにプッシュする．

・[store，変数番地]

store はスタックトップの値をポップし被演算子の変数番地に保存する．値スタックに保存する前の変数番地の値をプッシュする．

・[jpc，ジャンプ先 PC]

jpc はスタックトップから値をポップしその値が 1 ならば被演算子のジャンプ先 PC の値を次の PC の値とする．

・[jmp，ジャンプ先 PC]

jmp は無条件で被演算子のジャンプ先 PC の値を次の PC の値とする．

・[op，演算番号]

op はスタックトップから値を二回ポップしその二つの値に対して被演算子の演算番号（0,1,2,3,4）に対してそれぞれ（+,-,×,¿,==）の演算を行う．

・[label，バイトコード全体の抽象命令の数]

label は一つ前の PC の値をバイトコード全体の命令数+1 から引いた値（逆方向コードにしたときに対応する命令の PC）をラベルスタックにプッシュする．

・[par，{0,1}]

par は並列ブロックの区切りを表し，被演算子が 0 の時は始まりを表し，1 で終了を表す．

・[alloc，変数番地]

alloc は変数番地の番地の確保を行う．初期値は 0 となっている．

・[free，変数番地]

free は変数番地の解放を行う．変数の値を変数テーブルに記録し順方向実行時の最後の変数の値を保存する．

・[proc，pn]

proc は手続きの始まりを表す．パスに pn を追加し label 命令と同様にラベルスタックに一つ前の PC の値をバイトコード全体の命令数+1 から引いた値をプッシュする．帰り番地を保存するために一つ前のＰＣの値を演算スタックにプッシュする．

・[p_return，pn]

p_return は手続きの終了を表す．パスから pn を削除し演算スタックから帰り番地のＰＣをポップしその PC にジャンプする．

・[block，bn]

block はパスに bn を追加する．

・[end，bn]

end はパスから bn を削除する．

・[fork，an]

fork は並列ブロックの始まりを表し，各ブロックのプロセスを並列に実行するプロセスを生成する．プロセスの生成にはバイトコード生成時に作成した並列テーブル an を参照し各ブロックの開始終了番地を各プロセスに与え，実行を開始させる．

・[merge，an]

merge は並列ブロックの終わりを表す．

・[func，fn]

func は関数の始まりを表す．パスに fn を追加し，label 命令と同様にラベルスタックに一つ前の PC の値をバイトコード全体の命令数+1 から引いた値をプッシュする．帰り番地を保存するために一つ前のＰＣの値を演算スタックにプッシュする．演算スタックに既に積まれている実引数の値を演算スタックの一番上に移動させる．

・[f_return，fn]

f_return は関すの終わりを表す．パスから pn を削除し演算スタックからスタックトップの一つ下にある帰り番地のＰＣをポップしその PC にジャンプする．

・[w_label，wn]

w_label はパスに wn を追加し一つ前の PC の値をバイトコード全体の命令数+1 から引いた値（逆方向コードにしたときに対応する命令の PC）をラベルスタックにプッシュする．

・[w_end，wn]

w_end はパスから wn を全て削除する．

・[nop，0]

nop は何も操作を行わない．

```
1 : block  b1          41: op     4
2 : alloc  0           42: jpc    44
3 : alloc  1           43: jmp    61
4 : alloc  2           44: label  80
5 : jmp    66          45: load   0
6 : proc   p1          46: ipush  0
7 : fork   a1          47: op     3
8 : par    0           48: jpc    50
9 : block  b2          49: jmp    56
10: w_label w1         50: label  80
11: load   1           51: load   0
12: ipush  1           52: ipush  1
13: op     4           53: op     2
14: jpc    16          54: store  0
15: jmp    33          55: jmp    59
16: label  80          56: label  80
17: load   0           57: ipush  0
18: ipush  0           58: store 2
19: op     3           59: label  80
20: jpc    22          60: jmp    38
21: jmp    28          61: w_end  w2
22: label  80          62: end    b3
23: load   0           63: par    1
24: ipush  1           64: merge a1
25: op     2           65: p_return p1
26: store  0           66: label  80
27: jmp    31          67: ipush  3
28: label  80          68: store  0
29: ipush  0           69: ipush  1
30: store  1           70: store  1
31: label  80          71: ipush  1
32: jmp    10          72: store  2
33: w_end  w1          73: block  c1
34: end    b2          74: jmp    6
35: par    1           75: label  80
36 : par   0           76: end    c1
37: block  b3          77: free   2
38: w_label w2         78: free   1
39: load   2           79: free   0
40: ipush  1           80: end    b1
```

図 10 プログラム例: 順方向実行のバイトコード

Fig. 10 sample program: a byte code of forward execution

図 8 を順方向実行のバイトコードに変換すると**図 9** になる．左端の数字は PC（プログラムカウンタ）を表し（命令，被演算子）というように抽象機械命令が表示されている．

3.2.2 逆方向抽象機械命令

逆方向抽象機械命令のバイトコードは順方向のバイトコードの抽象機械命令を一対一で変換することで作成する．以下に生成規則を示す．順方向のバイトコードを s とし s から逆方向のバイトコードへの変換を i(s) で表す．inv(s)

は各抽象機械命令の変換を表す．

$$i(s) = \begin{cases} \epsilon & (s = \epsilon) \\ i(s')inv(c) & (s = cs') \end{cases}$$

$inv(store\ v) = restore\ v, \quad inv(jpc\ a) = r\_label\ 0$
$inv(jmp\ a) = r\_label\ 0, \quad inv(label\ n) = rjmp\ 0$
$inv(par\ 0) = par\ 1, \quad inv(par\ 1) = par\ 0$
$inv(alloc\ v) = r\_free\ v, \quad inv(free\ v) = r\_alloc\ v$
$inv(fork\ an) = merge\ an,\ inv(merge\ an) = r\_fork\ an$
$inv(block\ bn) = end\ bn,\ inv(end\ bn) = block\ bn$
$inv(proc\ pn) = r\_return\ pn$
$inv(p\_return\ pn) = r\_proc\ pn$
$inv(func\ fn) = r\_return\ fn$
$inv(f\_return\ fn) = r\_proc\ pn$
$inv(w\_label\ wn) = w\_end\ wn$
$inv(w\_end\ wn) = r\_w\_label\ wn$

その他の命令 c は $inv(c\ n) = nop\ 0$ に変換する．

続いて，逆方向抽象機械命令セットを以下に示す．

[命令，被演算子]

・[rjmp，0]

rjmp はラベルスタックから値をポップしその値を次の PC の値とする．

・[restore，変数番地]

restore は値スタックから値をポップしその値を共有変数スタックの変数番地に格納する．

・[r_label，0]

r_label は rjmp 命令のジャンプ先の対象となっている．

・[par，{0,1}]

par は並列ブロックの区切りを表し，被演算子が 0 の時は始まりを表し，1 で終了を表す．

・[r_alloc，変数番地]

変数番地から値を変数番地の変数の値を読み出しその値を共有変数スタックの変数番地に格納する．

・[free，変数番地]

共有変数スタックの変数番地を解放する．

・[r_proc，pn]

手続きの始まりを表す．パスに pn を追加する

・[r_return，pn]

手続きの終わりを表す．パスに pn を追加する．

・[block，bn]

block はパスに bn を追加する．

・[end，bn]

end はパスから bn を削除する．

・[r_fork，an]

r_fork は並列ブロックの始まりを表し，各ブロックのプロセスを並列に実行するプロセスを生成する．プロセスの生成にはバイトコード生成時に作成した並列テーブル an を fork 命令とは逆順に参照し各ブロックの開始終了番地を各プロセスに与え，実行を開始させる．

```
1 : block  b1          41: nop    0
2 : r_alloc 0          42: nop    0
3 : r_alloc 1          43: w_end  w2
4 : r_alloc 2          44: block  b3
5 : block  c1          45: par    1
6 : rjmp   0           46: par    0
7 : r_label 0           47: end   b2
8 : end    c1          48: r_w_label w1
9 : restore 2          49: r_label 0
10: nop     0          50: rjmp    0
11: restore 1          51: restore 1
12: nop     0          52: nop     0
13: restore  2         53: rjmp    0
14: nop     0          54: r_label 0
15: rjmp    0          55: restore 2
16: r_proc  p1         56: nop     0
17: r_fork  a1         57: nop     0
18: par     0          58: nop     0
19: block   3          59: rjmp    0
20: r_w_label 22       60: r_label 0
21: r_label  0         61: r_label 0
22: rjmp     0         62: nop     0
23: resotre  2         63: nop     0
24: nop      0         64: nop     0
25: rjmp     0         65: rjmp    0
26: r_label  0         66: r_label 0
27: restore  0         67: r_label 0
28: nop      0         68: nop     0
29: nop      0         69: nop     0
30: nop      0         70: nop     0
31: rjmp     0         71: w_end   w1
32: r_label  0         72: end     b1
33: r_label  0         73: par     1
34: nop      0         74: merge   a1
35: nop      0         75: r_return p1
36: nop      0         76: r_label 0
37: rjmp     0         77: free    2
38: r_label  0         78: free    1
39: r_label  0         79: free    0
40: nop      0         80: end     b1
```

図 11　プログラム例: 逆方向実行のバイトコード
Fig. 11　sample program: a byte code of backward execution

・[merge，an]

merge は並列ブロックの終わりを表す．

・[r_w_label，wn]

while ブロックの始まりを表す．w_label はパスに wn を追加する．

・[w_end，wn]

w_end はパスから wn を全て削除する．

・[nop，0]

何も操作を行わない．

図 9 の抽象機械命令を一対一で変換し順序を反転させたものが**図 10** の逆方向実行のバイトコードである．これを用いて順方向実行の実行を逆に辿る実行を行う．

4. 可逆実行環境

本章では，本研究で実装した可逆実行環境について説明する．本研究では先行研究を基に抽象機械命令のバイトコードを抽象機械で実行する実行環境という先行研究より単純化した手法によって可逆実行を実現する．本論文では以前までに実装した可逆実行環境に対して新しく変数のスコープ，並列ブロックのネスト構造，手続き呼び出し及び関数呼び出しの拡張を行った．

ブロック構造を許すように言語の定義を拡張すると，変数のスコープを扱う必要がある．これに対しては先行研究を手法を参考に各ブロックに名前を付け，それらを繋げて参照構造を表すパスとすることで変数のスコープの扱いを可能にしている．

並列ブロックは順方向の抽象機械のバイトコードを作成する際に各ブロックの開始番地と終了番地を記録したテーブルをそれぞれの並列ブロックごとに作成し，fork 命令実行時に参照することで動的にプロセスを生成し，並列のネスト構造の扱いを可能にしている．

手続き呼び出しは呼び出し時に演算スタックに帰り番地を積んでおくことで手続きが終了したら演算スタックから帰り番地を読み出し，その PC にジャンプすることで本研究の抽象機械でも手続き呼び出しの動作をすることができる．

関数呼び出しも手続き呼び出しと同様に動作するが，手続き呼び出しとは異なる機能として返り値がある．そのため帰り番地を読み出す際に演算スタックのスタックトップから一つ下の値を読み出すという動作を行う．

逆方向実行環境について，本研究では順方向実行時に逆方向実行に必要な情報をスタックに保存することで，この逆方向実行に必要な情報と順方向実行のバイトコードの抽象機械命令を一対一で変換した逆方向実行のバイトコードを用いて順方向実行を逆方向に辿る実行を行う．逆方向実行に必要な情報を保存するスタックとして変数の更新履歴を保存する値スタック，ジャンプ履歴を保存するラベルスタックの二つのスタックを用いる．

本研究において，抽象機械は並列ブロックを動作させるために Python の multiprocessing モジュールを用いて実装した．

4.1 順方向実行環境

順方向の実行では実行対象のプログラムをコンパイラで変換した順方向実行のバイトコードと変換する際に生成した各並列ブロックの開始番地と終了番地を保存した並列テーブルを用いて実行する．実行の際には store 命令で変数の更新履歴を値スタックに保存し，label 命令でジャンプ履歴をラベルスタックに保存する．

変数は alloc 命令が実行される際に変数テーブルにその時点でのパスと変数の名前を繋げて固有の変数名とした名前を記録する．free 命令を実行する際に解放する変数の値を名前の一致する変数名と組にして保存する．

4.1.1 変数のスコープ

本論文では，ブロック構造を記述できるように対象言語を拡張した．そのため例えば，手続きブロック内で宣言される変数 X とその外で宣言される変数 X は別物として扱う機能が必要である．そこで先行研究を参考にそれぞれのブロックに名前を付け参照構造を保存するためにパスという機能を実装した．例えば，ブロック b1 内のブロック b2 内の手続きブロック p1 内はパス b1.b2.p1 となり，参照構造が明らかにわかるようになっている．そしてこの手続きブロック p1 内で宣言される変数 X は b1.b2.p1.X と固有の名前を改めて付け以降はこの名前で扱う．load 命令などで変数 X を参照する場合はその時点のパスを内側から検索していき最も近いパスと最後に X の名前がついている変数名の値を読み出す．

4.1.2 並列ブロック

バイトコードにおいて一つの並列ブロックは fork 命令から始まり，各ブロックが par 0，par 1 で囲まれ，merge 命令で終わる構造をしている．バイトコードを作成する際に生成する並列テーブルはこの par 0 と par 1 の番地を組にして保存し各ブロックの開始番地終了番地を保存している．fork の被演算子は並列ブロックの名前である an となっており各並列テーブルはこの an の名前を付けてその並列ブロックのテーブルであるかを識別できるようにしている．この並列ブロックを fork 命令で参照することでプロセス数を何個生成しそれぞれのプロセスに与える開始番地，終了番地がわかるため動的にプロセスを生成することが可能となっている．

抽象機械の並列の動作は Python の multiprocessing モジュールを用いて実装した．並列のネスト構造を実装するために各プロセスは並列プロセスを生成する際に自分の抽象機械としての動作は停止する．そして，自分が生成したプロセスの番号を記録しそれらが終了しているか否かを常に監視するモニタープロセスとして抽象機械の並列動作とは別に並列に動作する．

4.1.3 手続き呼び出しおよび関数呼び出し

本研究の抽象機械において，手続きの宣言は proc 命令から p_return 命令までのブロックで記述され，手続きの呼び出しは proc 命令の PC にジャンプする jmp 命令によって実現している．呼び出しブロック名をパスに追加するためにこの jmp 命令の前に block 命令を生成するようにしている．手続き呼び出しの引数は実引数の値を呼び出しの jmp 前に load 命令で演算スタックに積んでおき，宣言の proc 命令後に store 命令で仮引数を扱う変数に代入する．手続きが終了し呼び出した番地に帰る p_return 命令では proc

命令で演算スタックに積んでおいた呼び出した jmp 命令の PC を演算スタックから読み出しその番地にシャンプすることで手続きからの帰り動作を実現している．

一方，関数宣言は func 命令から f_return 命令までのブロックで記述され，関数からの呼び出した番地への帰り動作と返り値の扱い以外は手続きと同様に動作する．関数では帰り値を扱う必要があるため f_return 命令で呼び出した番地へ帰る前に load 命令で演算スタックに返り値を積む．この際に積む値は関数の名前自体を変数名とした変数の値とする．そのため func 命令をした後関数名を変数名とした変数を alloc 命令によって宣言する．この関数内では関数名自体を局所変数と同様に扱って演算することができる．

4.2 逆方向実行環境

逆方向の実行では，順方向実行のバイトコードの抽象命令を一対一で変換した逆方向実行のバイトコードを抽象機械に与え，順方向実行時にスタックに保存した逆向き実行に必要な情報を用いて順方向の実行を逆向きに辿る実行を行う．逆向き実行に必要な情報はジャンプ履歴，変数更新履歴そして各変数の最後の値である．順方向実行時にジャンプ履歴はラベルスタックに保存し変数更新履歴は値スタックに保存し各変数の最後の値は変数テーブルに保存する．各変数の最後に関しては単純に順方向の free 命令でテーブルに書き込み逆方向の r_alloc 命令でその値を読み込む．

4.2.1 ジャンプ履歴の保存

のような方法でジャンプ履歴を保存し，保存した情報を使って逆順にジャンプする．順方向の実行では jmp 命令の対象には必ず label 命令が生成されるようになっており，label 命令でどこからジャンプしてきたかというジャンプ履歴の保存を行う．ラベルスタックには PC=k の命令を逆方向実行のバイトコードに置き換えたときの PC の値 n-k+1 を保存する．label 命令を実行した時のパスと実行したプロセスを繋げたものもその PCn-k+1 と組にして保存する．

逆方向の実行では label 命令から変換した rjmp 命令でラベルスタックに積まれたジャンプ履歴を取り出しその PC にジャンプする．ただしパスとプロセス番号が一致しているかを確認し一致していない場合実行することができない．その場合このプロセスは待ち状態となり別のプロセスが実行を進めていく．このようにして順方向の実行で起きたジャンプをちょうど逆順に辿る実行を行う．

4.2.2 変数更新履歴の保存

図の σ は共有変数スタックを表し，w は演算スタックを表す．図のような方法で変数更新履歴を保存し，保存した情報を使って変数の値を逆順に戻していく．順方向の実行では store 命令を実行する際に演算スタックのトップから値をポップし共有変数スタックに値を保存する．その際に失われるはずのそれまでの変数の値を値スタックに保存する．store 命令を実行したときのパスと実行したプロセスを繋げたものもその値と組にして保存する．

逆方向の実行では store 命令から変換した restore 命令で値スタックに積まれた変数の値を取り出しその値を被演算子と現在のパスが表す番地に保存することで順方向の変数の更新とは逆順に変数の値を戻す実行を行う．

```
begin b1
    var x;
    var y;
    func f1 bug_fact(x) is
        par a1
            begin b2
                var z;
                if (x>0) then
                    begin b3
                        z=x-1;
                        fact = x*{c1 fact(z)}
                    end
                else
                    fact=1
                fi
                remove z;
            end
        || begin b4
                if (x>1) then
                    x = x-1
                else
                    skip
                fi
            end
        rap
    return
    x=3;
    y={c2 bug_fact(x)}
    remove y;
    remove x;
end
```

図 12 対象プログラム（bug_fact）
Fig. 12 a target program(bug_fact)

5. 実行例

本章では本研究で実装した可逆実行環境の実行例を示す．**図 11** の対象プログラムを順方向実行しその実行を逆に辿る実行をすることを考える．図 11 のプログラムは 3 の階乗を計算するプログラムで，関数 bug_fact(x) は再帰的に計算を行い x の階乗を返す関数である．しかし bug_fact(x) は並列に二つのプロセスを実行し一つのプロセスは順当に階乗の計算を再帰的に行う．もう一つのプロセスは順当に行う階乗の計算を妨害するように仮引数 x の値をいずれかのタイミングで 1 引くプロセスとなっている．この妨害プロセスがどのタイミングで行われるかによって階乗計算の結果と再帰する数及び並列プロセスの生成数が異なる例となっている．

このプログラムをコンパイラに与えることでそれぞれの

```
1 : block  b1          39: end    b2
2 : alloc  0           40: par    1
3 : alloc  1           41: par    0
4 : jmp    64          42: block  b4
5 : func   f1          43: load   0
6 : alloc  2           44: ipush  1
7 : alloc  0           45: op     3
8 : store  0           46: jpc    48
9 : fork   a1          47: jmp    54
10: par     0          48: label  75
11: block  b2          49: load   0
12: alloc  3           50: ipush  1
13: load   0           51: op     2
14: ipush  0           52: store  0
15: op     3           53: jmp    56
16: jpc    18          54: label  75
17: jmp    34          55: nop    0
18: label  75          56: label  75
19: block  b3          57: end    b4
20: load   0           58: par    1
21: ipush  1           59: merge  a1
22: op     2           60: load   2
23: store  3           61: free   2
24: load   0           62: free   2
25: load   3           63: f_return f1
26: block  c1          64: label  75
27: jmp    5           65: ipush  3
28: label  75          66: store  0
29: end    c1          67: load   0
30: op     1           68: block  c2
31: store  2           69: jmp    5
32: end    b3          70: label  75
33: jmp    37          71: end    c2
34: label  75          72: store  1
35: ipush  1           73: free   1
36: store  2           74: free   0
37: label  75          75: end    b1
38: free    3
```

図 13 順方向実行のバイトコード（bug_fact）
Fig. 13 a byte code of forward execution(bug_fact)

ステートメントを抽象機械命令に変換し順方向実行のバイトコードを生成する．**図 12** が生成した順方向実行のバイトコードである．このバイトコードを抽象機械に与えることで順方向の実行を行う．

図 12 を抽象機械で実行すると**図 13**，**図 14** のように値スタック，ラベルスタックに逆方向実行に必要な情報が保存される．図 13 は値スタックを示し，一行のうち左側に変数の値，右側に store 命令を行ったプロセスとパスが保存されている．例えば一行目の (0 0.b1.E) はプロセス 0 のパス b1 の状態で何らかの変数の値を更新しその変数のそれまでの値が 0 であったことを示す．プロセス 0.1，プロセス 0.2 は並列で動作しているプロセスだが三行目，四行目を見るとその実行順がプロセス 0.2，プロセス 0.1 の順番で実行されたことが保存されている．図 14 はラベルスタックを示し，一行のうち左側にジャンプした PC の値，右側に label 命令を行ったプロセスとパスが保存されている．例えば一行目の (72 0.b1.E) はプロセス 0 のパス b1 の状態で PC=72 の命令から label 命令にジャンプしてきたことを示す．特に条件分岐について条件判定を行わずともどこから分岐（ジャンプ）したかという情報が残されているためラベルスタックを見るだけでどのように分岐したかがわかる．

```
0 0.b1.E
0 0.f1.c2.b1.E
3 0.2.b4.f1.c2.b1.E
0 0.1.b3.b2.f1.c2.b1.E
0 0.1.f1.c1.b3.b2.f1.c2.b1.E
2 0.1.2.b4.f1.c1.b3.b2.f1.c2.b1.E
0 0.1.1.b3.b2.f1.c1.b3.b2.f1.c2.b1.E
0 0.1.1.f1.c1.b3.b2.f1.c1.b3.b2.f1.c2.b1.E
0 0.1.1.1.b3.b2.f1.c1.b3.b2.f1.c1.b3.b2.f1.c2.b1.E
0 0.1.1.1.f1.c1.b3.b2.f1.c1.b3.b2.f1.c1.b3.b2.f1.c2.b1.E
0 0.1.1.1.1.b2.f1.c1.b3.b2.f1.c1.b3.b2.f1.c1.b3.b2.f1.c2.b1.E
0 0.1.1.1.b3.b2.f1.c1.b3.b2.f1.c1.b3.b2.f1.c2.b1.E
0 0.1.1.b3.b2.f1.c1.b3.b2.f1.c2.b1.E
0 0.1.b3.b2.f1.c2.b1.E
0 0.b1.E
```

図 14 値スタック（bug_fact）
Fig. 14 value stack(bug_fact)

```
72 0.b1.E
7 0.c2.b1.E
30 0.2.b4.f1.c2.b1.E
60 0.1.b2.f1.c2.b1.E
23 0.2.b4.f1.c2.b1.E
49 0.1.c1.b3.b2.f1.c2.b1.E
60 0.1.1.b2.f1.c1.b3.b2.f1.c2.b1.E
30 0.1.2.b4.f1.c1.b3.b2.f1.c2.b1.E
23 0.1.2.b4.f1.c1.b3.b2.f1.c2.b1.E
49 0.1.1.c1.b3.b2.f1.c1.b3.b2.f1.c2.b1.E
60 0.1.1.1.b2.f1.c1.b3.b2.f1.c1.b3.b2.f1.c2.b1.E
29 0.1.1.2.b4.f1.c1.b3.b2.f1.c1.b3.b2.f1.c2.b1.E
21 0.1.1.2.b4.f1.c1.b3.b2.f1.c1.b3.b2.f1.c2.b1.E
49 0.1.1.1.c1.b3.b2.f1.c1.b3.b2.f1.c1.b3.b2.f1.c2.b1.E
59 0.1.1.1.1.b2.f1.c1.b3.b2.f1.c1.b3.b2.f1.c1.b3.b2.f1.c2.b1.E
29 0.1.1.1.2.b4.f1.c1.b3.b2.f1.c1.b3.b2.f1.c1.b3.b2.f1.c2.b1.E
21 0.1.1.1.2.b4.f1.c1.b3.b2.f1.c1.b3.b2.f1.c1.b3.b2.f1.c2.b1.E
40 0.1.1.1.1.b2.f1.c1.b3.b2.f1.c1.b3.b2.f1.c1.b3.b2.f1.c2.b1.E
13 0.1.1.1.c1.b3.b2.f1.c1.b3.b2.f1.c1.b3.b2.f1.c2.b1.E
43 0.1.1.1.b2.f1.c1.b3.b2.f1.c1.b3.b2.f1.c2.b1.E
13 0.1.1.c1.b3.b2.f1.c1.b3.b2.f1.c2.b1.E
43 0.1.1.b2.f1.c1.b3.b2.f1.c2.b1.E
13 0.1.c1.b3.b2.f1.c2.b1.E
43 0.1.b2.f1.c2.b1.E
13 0.c2.b1.E
```

図 15 ラベルスタック（bug_fact）
Fig. 15 label stack(bug_fact)

図 12 の抽象命令を一対一で変換し順番を反転させたものが**図 15** である．変数の宣言，更新，解放やジャンプやパスの追加，削除そして並列ブロックに関わる命令以外は全て nop に変換されている．これは本研究における逆方向実行は変数の値を元に戻すということを主目的としている

```
1 : block   b1          39: rjmp    0
2 : r_alloc 0           40: restore 2
3 : r_alloc 1           41: nop     0
4 : restore 1           42: rjmp    0
5 : block   c2          43: r_label 0
6 : rjmp    0           44: block   b3
7 : r_label 0           45: restore 2
8 : end     c2          46: nop     0
9 : nop     0           47: block   c1
10: resotre 0           48: rjmp    0
11: nop     0           49: r_label 0
12: rjmp    0           50:  end    c1
13: r_proc  f1          51: nop     0
14: r_alloc 2           52: nop     0
15: r_alloc 0           53: restore 3
16: nop     0           54: nop     0
17: r_fork  a1          55: nop     0
18: par     0           56: nop     0
19: block   b4          57: end     b3
20: rjmp    0           58: rjmp    0
21: nop     0           59: r_label 0
22: rjmp    0           60: r_label 0
23: r_label 0           61: nop     0
24: restore 0           62: nop     0
25: nop     0           63: nop     0
26: nop     0           64: free    3
27: nop     0           65: end     b2
28: rjmp    0           66: par     1
29: r_label 0           67: merge   a1
30: r_label 0           68: restore 0
31: nop     0           69: free    0
32: nop     0           70: free    2
33: nop     0           71: r_return f1
34: end     b4          72: r_label 0
35: par     1           73: free    1
36: par     0           74: free    0
37: block   b2          75: end     b1
38: r_alloc 3
```

図 16 逆方向実行のバイトコード（bug_fact）

Fig. 16 a byte code of backward execution(bug_fact)

ためである．そのため演算スタックを元に戻すという動作が存在しない．

図 15 の逆方向実行バイトコードと図 13，図 14 の逆方向実行に必要な情報を用いて順方向の実行を逆方向に辿る．図 13 と図 14 の下から保存した情報を消費していく．それぞれ restore 命令と rjmp 命令においてパスが一致しているか否かを判定し一致している場合左側の値を消費して変数の値を戻したりジャンプを逆方向に辿っていく．パスが一致していない場合そのプロセスの実行は待ち状態になり別プロセスが実行を進める．このようにして順方向で実行した順番とちょうど逆順に変数の更新と逆方向ジャンプを行う．

6. おわりに