

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对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 informatics, Nagoya University,
Furo-cho, Chikusa-ward, Nagoya-city, 464-8601

^{a)} tikedata@sqlab.jp

^{b)} yuen@sqlab.jp

```


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


```

図 1 対象プログラムの定義

Fig. 1 definition of language

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 ?? 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 ?? のプログラムから図 ?? 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。

図 ?? を実行すると図 ?? 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 ?? 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 ?? に実行された Annotated Program から Inverted Program への変換規則を示す。図 ?? 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 を実行することで Annotated Program の実行を逆順に辿る実行を行う。

図 ?? の実行された Annotated プログラムを図 ?? 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

対象とする並列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。ソース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。並列ブロックは par から始まり、各ブロックを \parallel の記号で区切り、 rap で終わる。

3.1 対象言語の定義

対象言語の定義を図 2 に示す。 bn, an, wn, pn, fn, cn はそれぞれのブロック名を示す。それぞれ n は整数値を表し $b1, b2, \dots$ のように整数値の部分が重複しないとする。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 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。

```

P ::= begin bn BB end
      | par an P(|| P)+ rap
      | S
BB ::= DV DP DF P(; P)+ RV
S ::= skip | X = E | if C then P else P fi |
      while wn C do P od | 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 | n | (E) | E op E | {cn b(X?)}
C ::= B | C && C | not C | (C)
B ::= E == E | E > E

(X: 変数, n: 整数, a: 手続き名, b: 関数名,
op: {+, −, ×})

```

図 2 対象言語の定義

Fig. 2 definition of language

・関数呼び出し 関数の宣言は func から始まり, return で終わるように記述する. 関数は簡単のため引数一つとする. 関数内では関数の名前を返り値とする. 関数の呼び出しは呼び出し名 *cn* と関数名および引数を {} で囲って記述する.

図 3 にプログラム例を示す. 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. *agent1* と *agent2* が並列に *seats* のチケットを売り *seats* が 0 になったら終了する. *seats* が 0 であるときに *seats* が 0 より大きいという条件判定が同時に行われてしまうと存在しない 0 枚目の *seats* を売ることになり *seats* の値が -1 になってしまうという望ましくない動作が生じる.

3.2 可逆抽象機械

プログラムを抽象機械のバイトコードに変換する. 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.

3.2.1 抽象機械の定義

順方向の抽象機械命令セットを表 1 に示す. 各命令に対する説明を表 2 に示す.

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对一で変換することで作成する. 図 4 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.

逆方向の抽象命令セットを表 3 に示す. 各命令に対する説明を表 4 に示す.

```

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 3 プログラム例

Fig. 3 sample program

3.2.2 バイトコードの例

図 3 を順方向実行のバイトコードに変換すると図 5 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.

図 5 の抽象機械命令を一对一で変換し順序を反転させたものが図 6 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. 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う.

ブロック構造に対する変数に対しては先行研究の手法を参考に各ブロックに名前を付け, 参照構造を表すパスとすることで変数のスコープを実現する.

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

表 1 順方向の抽象機械命令セット

Table 1 instruction of abstract machine for forward

番号	命令	被演算子
1	<i>ipush</i>	即値
2	<i>load</i>	変数番地
3	<i>store</i>	変数番地
4	<i>jpc</i>	ジャンプ先 PC
5	<i>jmp</i>	ジャンプ先 PC
6	<i>op</i>	演算番号
7	<i>label</i>	バイトコード全体の抽象命令数
8	<i>par</i>	{0,1}
9	<i>alloc</i>	変数番地
10	<i>free</i>	変数番地
11	<i>proc</i>	pn
12	<i>p_return</i>	pn
13	<i>block</i>	bn
14	<i>end</i>	bn
15	<i>fork</i>	an
16	<i>merge</i>	an
17	<i>func</i>	fn
18	<i>f_return</i>	fn
19	<i>w_label</i>	wn
20	<i>w_end</i>	wn
21	<i>nop</i>	0

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。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

4.1.1 抽象機械の動作

・ジャンプ履歴の保存

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 PC_{n-k+1} と組にして保存する。

・変数更新履歴の保存

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* 命令を実行した時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。

4.1.2 変数のスコープ

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

表 2 順方向の各命令に対する説明

Table 2 explanation of each instruction for forward

1	<i>ipush</i> はスタックのトップに被演算子の即値をプッシュする。
2	<i>load</i> 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。
3	<i>store</i> 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。
4	<i>jpc</i> 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。
5	<i>jmp</i> 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。
6	<i>op</i> 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。
7	<i>label</i> 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。
8	<i>par</i> 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。
9	<i>alloc</i> 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。
10	<i>free</i> 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。
11	<i>proc</i> は手続きの始まりを表す。パスに pn を追加し <i>label</i> 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。
12	<i>p_return</i> 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。
13	<i>block</i> はパスに bn を追加する。
14	<i>end</i> はパスから bn を削除する。
15	<i>fork</i> 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。
16	<i>merge</i> は並列ブロックの終わりを表す。
17	<i>func</i> は関数の始まりを表す。パスに fn を追加し、 <i>label</i> 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。
18	<i>f_return</i> は関数の終わりを表す。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。
19	<i>w_label</i> 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。
20	<i>w_end</i> はパスから wn を全て削除する。
21	<i>nop</i> は何も操作を行わない。

を名前の一一致する変数名と組にして保存する。

本論文では、ブロック構造を記述できるように対象言語

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

$$\begin{aligned} inv(store\ v) &= restore\ v, & inv(jpc\ a) &= r_label\ 0 \\ inv(jmp\ a) &= r_label\ 0, & inv(label\ n) &= r_jmp\ 0 \\ inv(par\ 0) &= par\ 1, & inv(par\ 1) &= par\ 0 \\ inv(alloc\ v) &= r_free\ v, & inv(free\ v) &= r_alloc\ v \\ inv(fork\ an) &= merge\ an, & inv(merge\ an) &= r_fork\ an \\ inv(block\ bn) &= end\ bn, & inv(end\ bn) &= block\ bn \\ inv(proc\ pn) &= r_return\ pn, & inv(p_return\ pn) &= r_proc\ pn \\ inv(func\ fn) &= r_return\ fn, & inv(f_return\ fn) &= r_proc\ pn \\ inv(w_label\ wn) &= w_end\ wn, & inv(w_end\ wn) &= r_w_label\ wn \end{aligned}$$

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

図 4 逆方向バイトコードへの変換規則

Fig. 4 conversion rule for backward bytecode

表 3 逆方向の抽象機械命令セット

Table 3 instruction of abstract machine for backward

番号	命令	被演算子
1	<i>rjmp</i>	0
2	<i>restore</i>	変数番地
3	<i>r_label</i>	0
4	<i>par</i>	{0,1}
5	<i>r_alloc</i>	変数番地
6	<i>free</i>	変数番地
7	<i>r_proc</i>	pn
8	<i>r_return</i>	pn
9	<i>block</i>	bn
10	<i>end</i>	bn
11	<i>r_fork</i>	an
12	<i>merge</i>	an
13	<i>r_w_label</i>	wn
14	<i>w_end</i>	wn
15	<i>nop</i>	0

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行う。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。

4.1.3 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

表 4 逆方向の各命令に対する説明

Table 4 explanation of each instruction for backward

1	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。
2	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。
3	r_label は rjmp 命令のジャンプ先の対象となる。
4	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。
5	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。
6	共有変数スタックの変数番地を解放する。
7	手続きの始まりを表す。パスに pn を追加する。
8	手続きの終わりを表す。パスに pn を追加する。
9	block はパスに bn を追加する。
10	end はパスから bn を削除する。
11	r_fork は並列ブロックの始まりを表し、各ブロック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。
12	merge は並列ブロックの終わりを表す。
13	while ブロックの始まりを表す。w_label はパスに wn を追加する。
14	w_end はパスから wn を全て削除する。
15	何も操作を行わない。

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。

4.1.4 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

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 5 プログラム例: 順方向実行のバイトコード

Fig. 5 sample program: a byte code of forward execution

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ジ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戻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戻り値を積む。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。

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: restore 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 6 プログラム例: 逆方向実行のバイトコード

Fig. 6 sample program: a byte code of backward execution

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对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 ジャンプ履歴の利用

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 変数更新履歴の利用

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。

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。

4.2.3 手続き、関数の逆方向の振る舞い

手続き、および関数は逆方向の命令では両方ともに r_proc 命令から始まり、r_return 命令で終わる。逆方向実行では呼び出しを行った命令の PC や手続き、関数が終了して戻る PC はジャンプ履歴としてラベルスタックに保存されているため、r_proc はパスを追加する機能を持った r_label 命令、r_return 命令はパスを削除する機能を持った rjmp 命令として動作することで逆方向の手続き、関数の実行を実現している。

4.2.4 並列ブロックの逆方向の振る舞い

逆方向実行では r_fork 命令で並列ブロックを生成するために並列テーブルの参照する際、並列テーブルに保存されている各ブロックの終わりの PC および始まりの PC と対応する逆方向実行のバイトコードの PC をそれぞれ始まりの PC と終わりの PC とする。これによって順方向実行で生成した際と同じプロセス番号で逆方向実行でも並列プロセスを生成することができる。ネスト構造を含んでいても必ず同じプロセス番号のプロセスがネストするプロセスを生成するため順方向実行と逆方向実行のネストの構造は一致する。

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 7 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することを考える。図 7 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。

```
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 7 対象プログラム (bug_fact)

Fig. 7 a target program(bug_fact)

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 8 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 8 を抽象機械で実行すると図 9、図 10 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 9 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 10 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

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 8 順方向実行のバイトコード (bug_fact)

Fig. 8 a byte code of forward execution(bug_fact)

```

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 9 値スタック (bug_fact)

Fig. 9 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.c2.b1.E
59 0.1.1.1.1.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.c2.b1.E
21 0.1.1.1.2.b4.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.c2.b1.E
13 0.1.1.1.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 10 ラベルスタック (bug_fact)

Fig. 10 label stack(bug_fact)

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。

図 8 の抽象命令を一对一で変換し順番を反転させたものが図 11 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 11 の逆方向実行バイトコードと図 9，図 10 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 9 と図 10 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

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: restore 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 11 逆方向実行のバイトコード (bug_fact)

Fig. 11 a byte code of backward execution(bug_fact)