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：可逆計算，並行性，プログラミング言語，可逆実行環境

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: Reversible computation, Concurrency, Programming Languages, Reversible runtime-environment

1. はじめに

可逆計算に基づいたプログラムの逆実行は，プログラムの振舞いの解析において有用な手がかりを与える．プログラムを順方向に実行し，初期状態から最終状態に至る状態遷移系列によって，初期状態における入力から最終状態おける出力を得る．デバッグなどの場合にプログラムの振舞いを解析する場合，計算過程の途中でどのようにプログラムの状態が変化したのかということを詳細に追跡する必要が生じる．順方向実行では，出力を計算するための情報が保持され，出力として必要のない情報は捨てられることが多い．解析のためにはプログラムの振舞いの履歴を残しておくことは有用であり，振舞いの解析のために必要な情報を残すことが望ましい．このため，解析のためにメモリの状態をすべてダンプしたり，必要と思われるログ情報を残すことが行われる．この観点において，逆方向に実行できるだけの情報があれば振舞いを特定することが容易となることが知られており，このアイデアに基づいた可逆プログラミング言語が提案されている [14], [16]．Janus においては，条件分岐構文を拡張して条件分岐がどの方向で発生したか逆方向から辿ることができ，最終状態からプログラムを逆から実行して初期状態に至る振舞いを再現することができる．可逆計算に基づいて必要十分の情報をプログラミング言語に組込むことによって効率的な解析を行なうことができるようになる．

並列プログラムでは概念的に複数のプログラムブロックが同時に実行される．多くの場合，並列の振舞いは共有変数を介したインターリーブによる並行実行で捉えられる．このため，個々のプログラムブロックの実行履歴に加えて，複数のプログラムブロックが全体としてどのように実行されたかという情報が必要になる．並列プログラムの並行実行では実行順序の組み合わせが膨大になることから逆方向の実行に必要な情報に限定することは有用である．Hoey らは並列プログラムに必要なアノテーションを付加することによって並列プログラムの可逆実行意味を示している [5], [8]．ここでは必要な履歴情報をプログラムのアノテーションに基づいて保存することによって逆方向の並行実行を可能とし，逆方向の実行によって初期状態まで戻る計算によってアノテーション情報が残らないことで，履歴情報が必要十分であることを示している．

ここでは並列プログラムの並行実行処理系が情報を保存することで逆方向の振舞いを実現する方法について示す．筆者らは，大域変数のみをもつ単純な並列ブロック構造を持つプログラムに対する実行環境を抽象機械のバイトコードを定義することで実現する方法を示した [9]．個々の抽象機械は局所スタックを持ち，逐次的にバイトコードを実行する．並列ブロックの実行では，個々のブロック毎に抽象機械を生成して並列に実行する．大域的な実行情報として，共有変数の更新情報 (値スタック) とジャンプによる個々の抽象機械のプログラムカウンタの更新情報 (ラベルスタック) を記録する．順方向計算は，入力値，空の値スタック，および空のラベルスタックから計算を開始し，出力値，値スタック，ラベルスタックを与えて終了する．逆方向の計算は，出力値，値スタック，ラベルスタックから開始して，もとの入力値と空の値スタックとラベルスタックで終了する．逆方向の計算を実行するためのバイトコードは，順方向のバイトコードを変換することによって得られる．

本発表では [9] の並列プログラミング言語を拡張し，再帰的な手続き呼び出しを導入する．再帰的なブロック構造を許すことによって動的な並列ブロックの逆方向実行を可能にする．実行環境にプログラムの並列構造を示す構文的なテーブルを導入することによって実現する．このメカニズムによって，逆方向から実行する際に抽象機械をどのように生成すればよいかを知ることができる．手続に加えて関数についても実現し，実用的なプログラミング言語の処理系において可逆計算を可能とするために必要なメカニズムについて示す．さらに，実行環境を Python の Multiprocessing モジュールを用いて実現し，バイトコードへの変換器を Javacc を用いて実現した．

本発表の構成は以下の通りである．2 節において対象とする並列プログラミング言語を定義し，3 節において抽象機械とバイトコードを示す．4 節において実現した実行環境について説明する．5 節で関連研究を述べ，6 節でまとめを示す．

2. 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 || の記号で区切り，**rap** で終わる．

2.1 対象言語の定義

対象言語の定義を**図 1** 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,... 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

[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

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 1　対象言語の定義
Fig. 1　definition of language

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．
- 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．

2.2　プログラム例

図 2 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．このプログラムは二つの agent が並列に動作して seats を売っていく航空券販売のプログラムを表している．実際の動作について，6 から 15 行目と 16 から 25 行目は並列に動作し seats が 0 にならない限り seats を減らしていく．しかし，9 行目と 18 行目の 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 seats が 0 であるのに-1 を行い seats の値が-1 という望ましくない結果が得られることがある．これを実行する際に逆方向実行に必要な情報を残すことでこの実行を逆に辿り seats の値が不正に更新された部分を探すことを考える．この実行を逆に辿っていくと 10 行目もしくは 19 行目に対応する実行で seats の値が-1 から 0 に戻される．これによって seats が 0 より大きいという条件判定のもと seats=seats-1 を実行するはずが seats の値が既に 0 になってしまっていて不正に 1 引いてしてしまっている部分を特定することができる．

```
 1: begin b1
 2:     var seats;
 3:     var agent1;
 4:     var agent2;
 5:     proc p1 airline() is
 6:         par a1
 7:             begin b2
 8:                 while w1 (agent1==1) do
 9:                     if (seats>0) then
10:                        seats=seats-1
11:                     else
12:                         agent1=0
13:                     fi
14:                 od
15:             end
16:         ||  begin b3
17:                 while w2 (agent2==1) do
18:                     if (seats>0) then
19:                         seats=seats-1
20:                     else
21:                         agent2=0
22:                     fi
23:                 od
24:             end
25:         rap
26:     end
27:     seats=3;
28:     agent1=1;
29:     agent2=1;
30:     call c1 airline()
31:     remove agent2;
32:     remove agent1;
33:     remove seats;
34: end
```

図 2　プログラム例
Fig. 2　sample program

3.　可逆実行環境

本研究では先行研究 [5], [8] 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．

図 3 に本研究で実装した可逆実行環境のイメージを示す．

3.1　順方向実行環境

順方向の計算で用いるバイトコードセットを**表 1** に示す．

3.1.1　順方向実行環境の概要

- ジャンプ履歴の保存
  順方向の実行では jmp 命令のターゲットには必ず label 命令を生成する．label 命令は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プロセス ID と PC の値 a を保存する．label 命

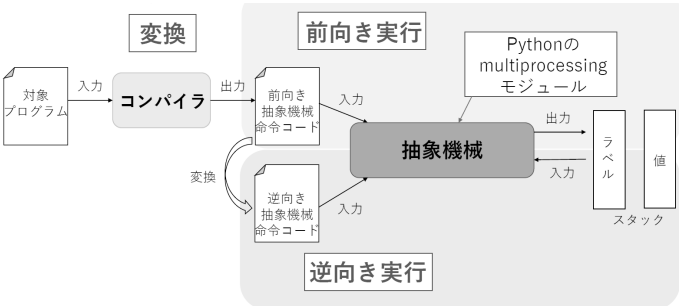

図 3　実装した可逆実行環境
Fig. 3　implimentation of reversible runtime

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

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 a 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．

3.1.2　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を名前の一致する変数名と組にして保存する．

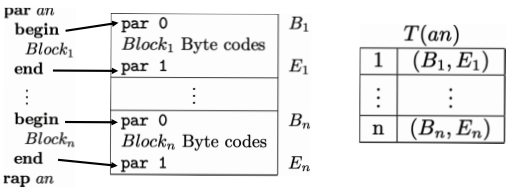

図 4　並列ブロックテーブル
Fig. 4　parallel block table

本発表では，ブロック構造を記述できるように対象言語を拡張する．そのため，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 は別として扱う機能が必要である．そこで先行研究 [5], [8] 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，参照構造を明示する．手続きブロック p1 内で宣言される変数 x は b1.b2.p1.x のように名前付けを行い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．

抽象機械では全体の変数テーブルを σ で表す．σ は，パスと変数名の対を入力としその変数が持つ値を出力する．テーブルにない場合はエラー値 err を返すとする．σ に対して以下の操作を定める．

- $lup(\sigma, \delta, x)$:
  δ のパスから x を参照したときの x の値を返す．
- $upd(\sigma, \delta, x, n)$:
  δ のパスから x を参照したときの変数の値を n に更新する．

3.1.3　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．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名 an である．ソースプログラムにおける an に対する並列ブロックテーブルを生成し，an に含まれる並列ブロックに対して，子プロセスとして生成するプロセスの命令開始番地と命令終了番地を示す静的なテーブル T を生成する．(図 4)

並列ブロックは，構成されるプログラムブロックが終了番地まで実行されてすべて終了したときに終了し，実行が継続する．

並列ブロック an に対する並列ブロックテーブルを $T(an)$ と記述し，i 番目のエントリーには，i 番目のブロックが実行するバイトコードの開始アドレスと終了アドレスを記録する．$T(an)(i) = (B, E)$ は，開始アドレスが B で，終了アドレスが E であることを示す．$|T(an)|$ は an から起動され

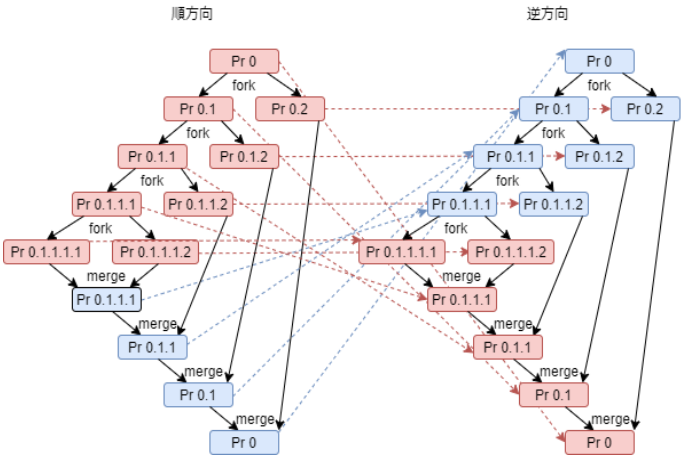

図 5　並列プロセスの生成

Fig. 5　creating parallel process

る並列ブロックの数を表し，$T(an).last$ は，an の最後のブロックの終了アドレスを示す．$T(an)(|T(an)|) = (B_\ell, E_\ell)$ のとき，$T(an).last = E_\ell$ である．

図 5 に並列プロセスの生成を示す．fork によって並列プロセスが生成，起動され，さらに生成されたプロセスから並列プロセスが生成，起動されている．それぞれ生成されたプロセスはその実行が終了すると merge で生成したプロセスに実行の制御を戻していく．

図 5 の順方向の図と逆方向の図は実線と点線がそれぞれ対応しており順方向の fork，merge はそれぞれ逆方向では merge，fork になる．このようにして順方向と同様の並列プロセスの生成を逆方向でも行う．

3.1.4　手続き呼び出しおよび関数呼び出し

抽象機械において，手続きの振舞いは 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，proc 命令の実行後に store 命令でブロック生成時に割り当てられる仮引数に対応する変数に代入する．手続きが終了し呼び出した番地に戻る 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戻り動作を実現する．

一方，関数の振舞いは 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．関数では返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 命令を実行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．

表 2　逆方向の抽象機械命令セット

Table 2　instruction of abstract machine for backward

番号	命令	被演算子
1	rjmp	0
2	restore	変数番地
3	par	{0,1}
4	r_alloc	変数番地
5	r_free	変数番地
6	r_fork	an
7	merge	an
8	nop	0

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

$inv(\text{store } x) = \text{restore } x,$　$inv(\text{label } n) = \text{rjmp } n$

$inv(\text{par } 0) = \text{par } 1,$　$inv(par\ 1) = par\ 0$

$inv(\text{alloc } x) = \text{r\_free } x,$　$inv(\text{free } x) = \text{r\_alloc } x$

$inv(\text{fork } an) = \text{merge } an,$　$inv(\text{merge } an) = \text{r\_fork } an$

$inv(\text{proc } pn) = \text{rjmp } N,$　$inv(\text{func } pn) = \text{rjmp } N$

$inv(\text{w\_label } wn) = \text{rjmp } N,$　$inv(\text{w\_end } wn) = \text{rjmp } N$

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

ここで N は実行バイトコード列の長さ．

図 6　逆方向バイトコードへの変換規則

Fig. 6　conversion rule for backward bytecode

3.1.5　抽象機械の順方向出力

抽象機械は，自分が生成した局所変数を remove するときに，プロセス Id とパスとともにその値を値スタックに記録する．この値は逆方向に実行するときにプロセス Id が起動されたとき，局所変数の初期値となる．このために，free の逆命令として，r_alloc を用意し，そのプロセスが変数を **remove** したときの値を復元する．alloc の逆命令は r_free である．r_free は順方向でその変数を σ から消去するが，値は記録しない．

3.2　逆方向実行環境の概要

逆方向の抽象命令セットを**表 2** に示す．

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．順方向バイトコード系列 s から逆方向バイトコード系列への変換 $i(s)$ を**図 6** に示す．

逆方向の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．

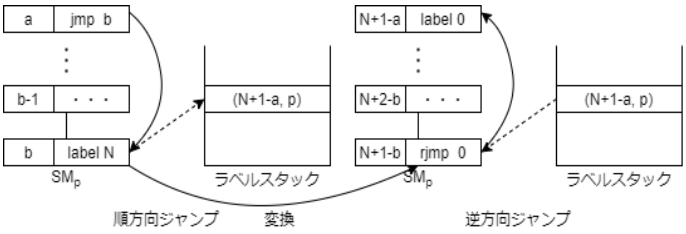

図 7　ジャンプ履歴の保存と利用方法
Fig. 7　reserving and using jump history

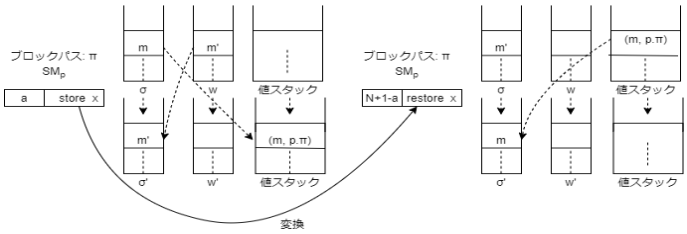

図 9　変数更新履歴の保存と利用方法
Fig. 9　reserving and using store history

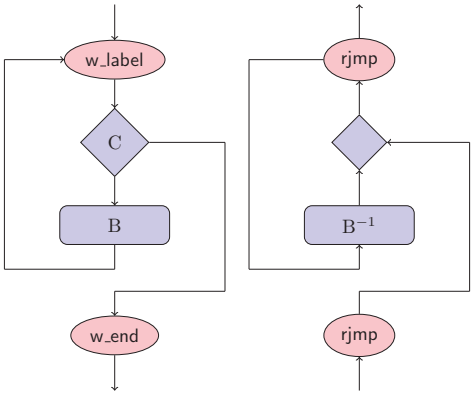

図 8　while ループの反転
Fig. 8　reversing while-loops

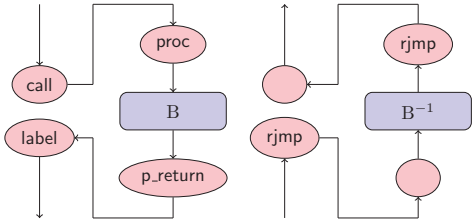

図 10　手続きブロックの反転
Fig. 10　reversing procedures

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．

3.2.1　ジャンプ履歴の利用

図 7 はジャンプ履歴の保存と利用法について示している．プロセス p の抽象機械が SM_p がパス π でジャンプし label 命令を行うと図 7 のようにラベルスタックにジャンプしてきた PC の値とそれを行ったプロセス番号が保存される．

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．

図 8 に while 構造の対応を示す．while ループのエントリにある w_label は繰返しの回数だけラベルスタックに B の最後のジャンプからの記録があるので，rjmp によって，B を逆転したコードを実行するか，while の前に戻るかを決める．w_end は，while 条件のターゲットになっているので rjmp でループに戻る．

3.2.2　変数更新履歴の利用

図 9 には変数更新履歴の保存と利用方法を示している．図 9 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 9 のように store 命令，free 命令を行うと失われるはずの変数の更新前の値を store 命令，free 命令を行ったプロセス番号とパスとともに値スタックに保存する．

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．

3.2.3　手続き，関数の逆方向の振舞い

手続き，および関数は逆方向の命令では両方ともに p_return 命令あるいは f_return 命令のあった番地の nop から始まり，rjmp 命令で終わる．ここで，逆方向実行の際の呼び出しは順方向のようにラベルスタックに呼び出し元のアドレスを記録する必要がないため，手続き，関数のリターン命令の逆命令は特に動作を必要としない．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，手続き，関数の初期状態まで到達したとき，rjmp 命令によって呼び出し側に戻る (**図 10**)．逆方向実行における関数の扱いは戻り値を必要としないため手続きと同様になる．

3.2.4　並列ブロックの逆方向の振舞い

逆方向実行では順方向実行で merge an が実行された時点で，複数の抽象機械を生成する．ここで生成するプロセスは an によって決まる．ただし，並列ブロックテーブルの参照を変更する必要があるので，fork を修正した r_fork 命令を用いる．r_fork an で並列ブロックを生成するために並列ブロックテーブル $T(an)$ を参照する際，並列テーブルに保存されている各ブロックのバイトコード列の開始番地と終了番地を対応する逆方向実行のために入れ替える．この変換を行なったテーブルを $T(an)^{-1}$ と書くことにする．逆方向実行におけるプロセスの生成は図 5 のように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同

じネスト構造のプロセスを生成し，逆方向の並行実行を実現する．

逆方向の並行実行においては，値スタックとラベルスタックのトップにあるプロセス Id を持つプロセスのみが実行可能であり，状態遷移の逆方向の実行のみが行われる．

3.3 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．

3.3.1 振舞い定義

変数集合 X 上の可逆抽象機械の振舞いを以下のように定義する．順方向のバイトコードと逆方向のバイトコードを持ち，並列ブロックを実行する際には複数の抽象機械を生成 (fork 動作) し，並列ブロックがすべて終了した際に制御をマージ (merge 動作) する．

可逆抽象機械の動作は，$(PC, PC', w, \delta, \chi, \rho, \xi, \sigma)$ で表す．

- PC: プログラムカウンター
- PC': 一つ前に実行したバイトコードのプログラムカウンター
- $w \in (\mathbb{Z} \cup \mathbb{A})^*$: ローカルスタック
- $\delta \in \mathbb{L}^*$: 動的コンテクスト
- $\chi \in \mathbb{L} \cup \{\bot\}$: 並列コンテクスト
- $\rho \in (\mathbb{P} \times (\mathbb{L})^* \times \mathbb{P})^*$: 値スタック
- $\xi \in (\mathbb{P} \times \mathbb{A})^*$: ラベルスタック
- $\sigma \in X \times \mathbb{L}^* \to \mathbb{Z}$: 変数値

ここで，$\mathbb{P}$ はプロセス Id の集合，$\mathbb{Z}$ は整数の集合，$\mathbb{A}$ はプログラム番地の集合を示す．

δ は，抽象機械が変数を参照するブロック名のパスを示し，χ は抽象機械が子プロセスを持つ場合，その並列ブロックの名前を保持する．子プロセスを持たない場合 $\bot$ となる．ρ，ξ，σ はすべての抽象機械で共有する．

プロセス Id は，プロセスが新たに生成されるごとにユニークな Id が生成される．以下では，プロセス Id は自然数の系列 $\mathbb{N}^+$ で表し，プロセス Id p が i 番目に生成したプロセスを $p \cdot i$ で表す．

ρ,ξ などのスタック構造は要素の列で表し，末尾をスタックトップとする．

3.3.2 順方向の振舞い定義

プロセス Id が p である抽象機械のバイトコード (b, o) の振舞い $\xrightarrow{b,o}_p$ を以下に示す．

- ipush:
  $(P, P', w, \delta, \chi, \rho, \xi, \sigma) \xrightarrow{(\mathsf{ipush},n)}_p$
  $(P+1, P, w \cdot n, \delta, \chi, \rho, \xi, \sigma)$
  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:
  $(P, P', w, \delta, \chi, \rho, \xi, \sigma) \xrightarrow{(\mathsf{load},x)}_p$
  $(P+1, P, w \cdot lup(\sigma, x, \delta), \delta, \chi, \rho, \xi, \sigma)$
  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:
  $(P, P', w \cdot n, \delta, \chi, \rho, \xi, \sigma) \xrightarrow{(\mathsf{store},x)}_p$
  $(P+1, P, w, \delta, \chi, \rho \cdot (p, \delta, lup(\sigma, \delta, x)), \xi, sto(\sigma, x, \delta, n))$
  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 x に保存する．参照したパスと値スタックに保存する前の変数の値を値スタック ρ に記録する．

- jpc:
  $(P, P', w \cdot c, \rho, \xi, \sigma) \xrightarrow{(\mathsf{jpc},a)}_p$
  $$\begin{cases} (a, P, w, \rho, \xi, \sigma) & \text{if } c = 1 \\ (P+1, P, w, \rho, \xi, \sigma) & \text{otherwise} \end{cases}$$
  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 a を次の PC の値とする．

- jmp:
  $(P, P', w, \delta, \chi, \rho, \xi, \sigma) \xrightarrow{(\mathsf{jmp},a}_p$
  $(a, P, w, \delta, \chi, \rho, \xi, \sigma)$
  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:
  $(P, P', w \cdot \delta, \chi, n' \cdot n, \rho, \xi, \sigma) \xrightarrow{(\mathsf{op},m)}_p$
  $(P+1, P, w \cdot op(m)(n', n), \delta, \chi, \rho, \xi, \sigma)$
  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 m（0,1,2,3,4）に対してそれぞれ $op(m)$（+,-,×,<,==）の演算を行う．

- label:
  $(P, P', w, \delta, \chi, \rho, \xi, \sigma) \xrightarrow{(\mathsf{label},N)}_p$
  $(P+1, P, w, \delta, \chi, \rho, \xi \cdot (p, P'), \sigma)$
  label はラベルスタックに飛び元の番地をプッシュする．

- par:
  $(B, 0, w, \rho, \xi, \sigma) \xrightarrow{(\mathsf{par},0)}_{p \cdot i} (B+1, B, w, \rho, \xi, \sigma)$
  $(E-1, P', w, \rho, \xi, \sigma) \xrightarrow{(\mathsf{par},1)}_{p \cdot i} (E, E-1, w, \rho, \xi, \sigma)$
  ここで $(B, E) = T(p)(i)$．par は親プロセス p の χ の名前を持つ並列ブロックテーブルの開始番地から par 0 を実行し，終了番地において par 1 を実行する．

- alloc:
  $(P, P', w, \delta, \chi, \rho, \xi, \sigma) \xrightarrow{(\mathsf{alloc},x)}_p$
  $(P+1, P, w, \delta, \chi, \rho, \xi, upd(\sigma, \delta, 0))$

alloc は変数 x の領域を σ に追加する．初期値は 0 となっている．

- free:
  $(P, P', w, \delta, \chi, \rho, \xi, \sigma) \xrightarrow{(\mathsf{free}, x)}_p$
  $(P+1, P, w, \delta, \chi, \rho \cdot (p, lup(\sigma, \delta, x), \delta), \xi, \sigma - (\delta, x))$
  free は変数領域の解放を行い，逆方向実行のために最後の値とパスを値スタックに記録する．

- proc:
  $(P, P', w, \delta, \chi, \rho, \xi, \sigma) \xrightarrow{(\mathsf{proc}, pn)}_p$
  $(P+1, P, w \cdot P'+1, \delta \cdot pn, \chi, \rho, \xi \cdot (p, P'), \sigma)$
  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 をプッシュする．帰り番地を保存するために一つ前の PC+1 を演算スタックにプッシュする．

- p_return:
  $(P, P', w \cdot a, \delta \cdot pn, \chi, \rho, \xi, \sigma) \xrightarrow{(\mathsf{p\_return}, pn)}_p$
  $(a+1, P, \delta, \chi, \rho, \xi, \sigma)$
  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 P C をポップしその PC にジャンプする．

- block:
  $(P, P', w, \delta, \chi, \rho, \xi, \sigma) \xrightarrow{(\mathsf{block}, bn)}_p$
  $(P+1, P, w, \delta \cdot bn, \chi, \rho, \xi, \sigma)$
  block はパスに bn を追加する．

- end:
  $(P, P', w, \delta \cdot bn, \chi, \rho, \xi, \sigma) \xrightarrow{(\mathsf{end}, bn)}_p$
  $(P+1, P, w, \delta, \chi, \rho, \xi, \sigma)$
  end はパスから bn を削除する．

- fork:
  $(P, P', w, \delta, \perp, \rho, \xi, \sigma) \xrightarrow{(\mathsf{fork}, an)}_p$
  $(P'', P, w, \delta, an, \rho, \xi, \sigma)$
  ここで，$P'' = T(an).last + 1$
  fork an は並列ブロックテーブル $T(an)$ に基づいて並行プロセスを生成する．$T(an)(i) = (B, E)$ であるとき，開始命令番地 B とするバイトコード列を実行する $p \cdot i$ を Id とする $|T(an)|$ 個の子プロセスを生成して実行する．

- merge:
  $(P, P', w, \delta, \chi, \rho, an, \sigma) \xrightarrow{(\mathsf{merge}, an)}_p$
  $(P+1, P, w, \delta, \perp, \rho, \perp, \sigma)$
  merge 命令は子プロセス $p \cdot i$ の PC がすべて $T(an)(i) = (B_i, E_i)$ の E_i となったときに実行される．

- func:
  $(P, P', w \cdot n, \delta, \chi, \rho, \xi, \sigma) \xrightarrow{(\mathsf{func}, fn)}_p$
  $(P+1, P, w \cdot P'+1 \cdot n, \delta \cdot fn, \chi, \rho, \xi \cdot (P', p), \sigma)$
  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 の値をプッシュする．帰り番地を保存するために一つ前の P C 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:
  $(P, P', w \cdot a \cdot r, \delta \cdot pn, \chi, \rho, \xi, \sigma) \xrightarrow{(\mathsf{f\_return}, pn)}_p$
  $(a, P, w \cdot r, \delta, \chi, \rho, \xi, \sigma)$
  f_return はパスから pn を削除し局所スタックからスタックトップの一つ下にある帰り番地をポップしその番地にジャンプする．

- w_label:
  $(P, P', w, \delta, \chi, \rho, \xi, \sigma) \xrightarrow{(\mathsf{w\_label}, wn)}_p$
  $(P+1, P, w, \delta \cdot wn, \chi, \rho, \xi \cdot (p, P'), \sigma)$
  w_label は while ループの開始点であり，繰り返し動作の最後からジャンプする先である．パスに wn を追加し一つ前の PC をラベルスタックにプッシュする．

- w_end:
  $(P, P', w, \delta, \chi, \rho, \xi, \sigma) \xrightarrow{(\mathsf{w\_end}, wn)}_p$
  $(P+1, P, w, \delta', \chi, \rho, \sigma \cdot (p, P'))$
  ここで，$\delta' = rm(\delta, wn)$
  w_end は while 文のループ条件が成立しなかった場合の飛び先なので，パスから wn を全て削除し，ラベルスタックに while 文のループ条件である命令番地 P' を記録する．

- nop:
  $(P, P', w, \delta, \chi, \rho, \xi, \sigma) \xrightarrow{(\mathsf{nop}, 0)}_p$
  $(P+1, P, w, \delta, \chi, \rho, \sigma)$
  nop は何も操作を行わない．

(store,x) において，値スタックに記録するパスは，変数 x が定義されているパスではなく，抽象機械が参照しているパス δ である．このことにより，逆方向の実行を行うときには，プロセス Id とともにどのパスから参照されたかを復元する．

3.3.3 逆方向振舞い定義

プロセス Id p の抽象機械の逆方向のためのバイトコード (b, o) の振舞い $\overset{b,o}{\rightsquigarrow}_p$ を以下に示す．

- rjmp:
  $(P, P', w, \delta, \chi, \rho, \xi \cdot (p, a), \sigma) \overset{(\mathsf{rjmp}, N)}{\rightsquigarrow}_p$
  $(N+1-a, P, w, \delta, \chi, \rho, \xi, \sigma)$
  rjmp はラベルスタックから値をポップし $N+1-a$ にジャンプする．長さ N の順方向の実行系列において a のアドレスは逆転させたバイトコード列では，

$N+1-a$ となる．

- restore:
  $(P, P', w, \delta, \chi, \rho \cdot (p, \delta', n), \xi, \sigma) \overset{(\text{restore},x)}{\rightsquigarrow}_p$
  $(P+1, P, w, \delta', \chi, \rho, \xi, upd(\sigma, \delta, x, n))$
  restore は値スタックから参照パスと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し，抽象機械の参照パスを更新する．

- par:
  $(B, 0, w, \rho, \xi, \sigma) \overset{(\text{par},0)}{\rightsquigarrow}_{p \cdot i} (B+1, B, w, \rho, \xi, \sigma)$
  $(E-1, P', w, \rho, \xi, \sigma) \overset{(\text{par},1)}{\rightsquigarrow}_{p \cdot i} (E, E-1, w, \rho, \xi, \sigma)$
  ここで $T(p)(i) = (B', E')$ に対して $(B, E) = (N+1-B', N+1-E')$ r_fork によって起動される．par は親プロセス p の χ の名前を持つ並列ブロックテーブルの開始番地から par 0 を実行し，終了番地において par 1 を実行する．

- r_alloc:
  $(P, P', w, \delta, \chi, \rho \cdot (p, \delta', n), \xi, \sigma) \xrightarrow{(\text{r\_alloc},x)}_p$
  $(P+1, P, w, \delta, \chi, \rho, upd(\sigma, \delta', x, n))$
  r_alloc は，値スタックからブロックが終了した時点の参照パスと値を取り出して復元する．

- r_free:
  $(P, P', w, \delta, \chi, \rho, \xi, \sigma) \overset{(\text{free},x)}{\rightsquigarrow}_p$
  $(P+1, P, w, \delta, \chi, \rho, \xi, \sigma - (\delta, x))$
  r_free は変数領域の解放を行う．(ρ への書き込みは行わない．)

- r_fork:
  $(P, P', w, \delta, \bot, \rho, \bot, \sigma) \overset{(\text{fork},an)}{\rightsquigarrow}_p$
  $(P'', P, w, \delta, an, \rho, \xi, \sigma)$
  ここで，$P'' = T(an)_N^{-1}.last + 1$
  r_fork an は並列ブロックテーブル $T(an)$ を逆方向のために変換した $T(an)_N^{-1}$ に基づいて子プロセスを生成する．ここで，$T(an)_N^{-1}$ は以下のように得られる．$T(an)(i) = (B, E)$ であるとき $T(an)_N^{-1}(i) = (N+1-E, N+1-B)$. $T(an)_N^{-1}$ は an に属するプログラムブロックの開始命令番地と終了命令番地を入れかえて得られる並行ブロックテーブルである．

- merge:
  $(P, P', w, \delta, \chi, \rho, an, \sigma) \overset{(\text{merge},an)}{\rightsquigarrow}_p$
  $(P+1, P, w, \delta, \bot, \rho, an, \sigma)$
  merge 命令は子プロセス $p \cdot i$ の PC がすべて $T(an)_N^{-1}(i) = (B_i, E_i)$ の E_i となったときに実行される．

- nop:
  $(P, P', w, \delta, \chi, \rho, \xi, \sigma) \overset{(\text{nop},0)}{\rightsquigarrow}_p$
  $(P+1, P, w, \delta, \chi, \rho, \sigma)$
  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 11　プログラム例: 順方向実行のバイトコード
Fig. 11　sample program: a byte code of forward execution

3.3.4　バイトコードの例

図 2 を順方向実行のバイトコードに変換すると**図 11** 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 11 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 12** 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．

3.4　プログラム例

図 11 は図 2 を順方向実行のバイトコードに変換したものであり，図 12 は図 11 を命令を一対一で置換し順番

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

図 **12** プログラム例: 逆方向実行のバイトコード
Fig. 12 sample program: a byte code of backward execution

を反転させた逆方向実行のバイトコードである．図 11 の PC=17 から PC=20 の実行と PC=45 から PC=48 の実行がそれぞれ図 2 の 9 行目，18 行目に対応している．この部分が seats=1 の状態で並列実行され同時に実行されると seats=-1 になる不正な動作が起こる可能性がある．

この実行を図 12 で逆方向に辿ることを考える．ラベルスタック，値スタックを使って逆方向実行を進めていくと順方向実行のバイトコードの PC=26（store 命令）を変換した PC=55 の restore 命令もしくは順方向実行のバイトコードの PC=54（store 命令）を変換した PC=27 の restore 命令において seats の値が-1 から 0 に戻される．これによって seats が 0 より大きいという条件判定で seats=seats-1 の処理をしたにもかかわらず seats の値がすでに 0 になってしまっていて不正に 1 引いてしまった部分がどこであるかを特定することができる．

4. 実行環境の実現

4.1 抽象機械の実装

本研究では並列プログラムの実行を行うため抽象機械を Python の multiprocessing モジュールを用いて実装した[*1]．並列プロセスの生成は fork 命令を実行する際に並列テーブルを参照し必要な数だけ multiprocessing モジュールの process 関数を用いて生成する．このとき並列プロセスを生成したプロセスは抽象機械の実行としては待ち状態になり生成したプロセスの番号を保持しそれらが終了しているかどうかを監視するプロセスとして動作する．監視プロセスが自分の生成したプロセスが終了した（PC が終了番地に達した）と判定した場合 multiprocessing モジュールの terminate 関数を用いてそのプロセスを終了させる．そのようにしてすべての生成したプロセスが終了したと判定された場合監視を終了し抽象機械の実行を行うプロセスに戻る．

4.2 実行例

本研究で実装した可逆実行環境の実行例を示す．**図 13** 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 13 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 14** 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 14 を抽象機械で実行すると**図 15**，**図 16** 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 15 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，free 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

[*1] https://github.com/iketaka1984/PRO_2021_1

```
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 13 対象プログラム（bug_fact）
Fig. 13 a target program(bug_fact)

0.1 の順番で実行されたことが保存されている．図 16 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 ID 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 が 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．

図 14 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 17** 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 17 の逆方向実行バイトコードと図 15，図 16 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 15 と図 16 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 命令においてプロセス番号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．プロセス番号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．

```
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   0
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 14 順方向実行のバイトコード（bug_fact）
Fig. 14 a byte code of forward execution(bug_fact)

5. 関連研究

Hoey らは，本発表と同様な並列プログラミング言語を可逆的に並行実行する方法を提案している．プログラムソース間の継続関係によって振舞を定義し，順方向の継続の際に逆方向実行に必要なアノテーションをプログラムに加えることで逆方向の計算を実現している．

ここではブロック構造およびステートメントに実行のための名前づけを行うアノテーションを生成し順方向に実行し，実行に関する情報をプログラム上に付加する．

ここで手続き 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

```
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
0 0.1.1.b3.b2.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.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 0.1.1.1.1.b2.f1.c1.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
1 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.b3.b2.f1.c1.b3.b2.f1.c1.b3.b2.f1.c2.b1.E
0 0.1.1.1.b2.f1.c1.b3.b2.f1.c1.b3.b2.f1.c2.b1.E
1 0.1.1.f1.c1.b3.b2.f1.c1.b3.b2.f1.c2.b1.E
1 0.1.1.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
1 0.1.1.b2.f1.c1.b3.b2.f1.c2.b1.E
1 0.1.f1.c1.b3.b2.f1.c2.b1.E
1 0.1.f1.c1.b3.b2.f1.c2.b1.E
0 0.1.b3.b2.f1.c2.b1.E
2 0.1.b2.f1.c2.b1.E
2 0.f1.c2.b1.E
2 0.f1.c2.b1.E
0 0.b1.E
2 0.b1.E
3 0.b1.E
```

図 15 値スタック（bug_fact）
Fig. 15 value stack(bug_fact)

```
72 0
7 0
30 0.2
60 0.1
23 0.2
49 0.1
60 0.1.1
30 0.1.2
23 0.1.2
49 0.1.1
60 0.1.1.1
29 0.1.1.2
21 0.1.1.2
49 0.1.1.1
59 0.1.1.1.1
29 0.1.1.1.2
21 0.1.1.1.2
40 0.1.1.1.1
13 0.1.1.1
43 0.1.1.1
13 0.1.1
43 0.1.1
13 0.1
43 0.1
13 0
```

図 16 ラベルスタック（bug_fact）
Fig. 16 label stack(bug_fact)

```
1 : nop     0          39: rjmp    75
2 : r_alloc 0          40: restore 2
3 : r_alloc 1          41: nop     0
4 : restore 1          42: rjmp    75
5 : nop     0          43: nop     0
6 : rjmp    75         44: nop     0
7 : nop     0          45: restore 2
8 : nop     0          46: nop     0
9 : nop     0          47: nop     0
10: resotre 0          48: rjmp    75
11: nop     0          49: nop     0
12: rjmp    75         50: nop     0
13: nop     0          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: nop     0          57: nop     0
20: rjmp    75         58: rjmp    75
21: nop     0          59: nop     0
22: rjmp    75         60: nop     0
23: nop     0          61: nop     0
24: restore 0          62: nop     0
25: nop     0          63: nop     0
26: nop     0          64: r_free  3
27: nop     0          65: nop     0
28: rjmp    75         66: par     1
29: nop     0          67: merge   a1
30: nop     0          68: restore 0
31: nop     0          69: r_free  0
32: nop     0          70: r_free  2
33: nop     0          71: rjmp    75
34: nop     0          72: nop     0
35: par     1          73: r_free  1
36: par     0          74: r_free  0
37: nop     0          75: nop     0
38: r_alloc 3
```

図 17 逆方向実行のバイトコード（bug_fact）
Fig. 17 a byte code of backward execution(bug_fact)

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 を書き加える．

図 19 を実行すると**図 20** 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 20 は 2 回目の手続き呼び出しのブロック自体をコピーして Annotation とパスを書き込んでいる．ここで変数 T=0 が var ステートメントで宣言されているが，この変数は c1:c2:b2*b1 の T として扱われる．このようにして変数を宣言する際にパスを要素に組み込むことで局所変数を実現している．

図 20 の実行情報を含む Annotated プログラムを**図 21** Inverted Program へ変換する．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．

```
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 18 対象プログラム
Fig. 18 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 19 Annotated プログラム
Fig. 19 Annotated Program

```
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 20 実行された Annotated プログラム
（2 回目の手続き呼び出し）
Fig. 20 Executed Annotated Program
(second procedure call)

```
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 21 Inverted プログラム
（2 回目の手続き呼び出しの部分）
Fig. 21 Inverted Program
(part of second procedure call)

本発表では，変数の参照および値の更新履歴においては同じアイデアを用いている．Hoey らの可逆実行では制御フローの逆転については，ソースプログラムに Annotation という形でうめこむことで実現している．これに対して本発表では制御フローに着目し，抽象機械とそのバイトコードによって値更新と分離して扱うことで順方向の実行フローから逆方向の実行フローは，対応するバイトコードを個別に変換しコードの順番を逆に並べるだけで得ることが可能になっている．

抽象機械の概念を利用したプログラミング言語処理系の可逆実行環境として，[13] で提案されている．ここでは，マルチスレッド言語 μOz に対して可逆抽象機械を提案している．μOz 言語では，スレッド間の相互作用はポートによる同期通信によって行われる．このため，各スレッドは自スレッドの状態遷移と通信履歴を保持することによって可逆実行を実現する．この場合，変数更新の依存関係がたもたれる限りにおいて逆方向計算の自由度があり，因果無矛盾性 (Causal consistency) が保証される．これに対して本発表の逆方向計算および [5] は，全ての抽象機械が共有する値スタックとラベルスタックによって順方向計算の状態遷移を保存してバックトラック [1], [7] するため，逆方向の計算はより制限される．このためにより多くの履歴を保存している．

[11] では，Erlang 言語に対する可逆実行について提案されている．Erlang ではアクターモデルに基づいた分散した非同期の振舞いをおこなう．因果無矛盾性を持つ可逆意味が定義されている．さらに，Roll-back 演算子を定義することで，逆方向の計算を制御する手法を提案している．

可逆プログラム言語に対する抽象機械としては，[2] においてアーキテクチャが提案されている．ここでは，Janus などの状態保存を前提としないプログラミング言語の実行環境を目的としており，並行性を含む振舞いについては考慮されていない．

可逆計算に基づくデバッグ手法として，GNU Debugger[4] に逆方向のステップ実行機能が導入されている．並行性を含む枠組みとして，Erlang に対するデバッガ [10], [12] および CSP モデルに対する逆方向計算の可視化ツール [3] が提案されている．この他，ソフトウェア一般の商用のツールとして UNDO 社から発表 [15] されている．

6. おわりに

本発表では，再帰的な手続き呼出しによって再帰的なブロック構造を持つ簡単なプログラミング言語に対する可逆的な実行環境を提案した．ソースプログラムを並行に実行される複数の抽象機械のバイトコードに変換し，振舞いを単純なバイトコード列によって表現する．順方向実行のバイトコード列は直観的なプログラムの操作的意味論に従って生成することができ，Javacc を用いて実現した．逆方向実行のバイトコード列は順方向実行のバイトコード列の順番を逆転し，対応するバイトコードに変換することによって得られる．バイトコードを設計するとともに Python によって抽象機械を実現し，Multiprocess モジュールを用い

て抽象機械を並行に実行することで可逆並行実行環境を作成した．

ここでは，[9] をより複雑な構造を持つプログラミング言語に拡張し，再帰呼び出しに従って生成される並列ブロックを実行するプロセス動的に生成することができるように拡張した．この実現のために，並行ブロックに名前をつけ，並行ブロック毎にプロセス生成のスキーム (並列ブロックテーブル) をプログラムの構造から静的に生成することで実現した．逆方向実行の場合には同じテーブルを参照し，逆方向に各ブロックを逆方向に並行実行するプロセスを生成する．

ブロック構造による変数スコープの参照は [5], [8] の方法に従って，参照が発生した呼出パスを変数名に付加することで，大域的な変数と同様に扱っている．このことで，手続呼出と関数呼出において通常の逐次プログラムにおける駆動レコードを生成することなく，局所変数のスコープを実現している．

本発表における枠組みでは，プログラムの実行制御フローをバイトコードとして分離することで可逆実行における順方向と逆方向の制御フローの対応を明確に示すことができた．ブロックの再帰構造を導入することで，実際的なプログラムに対して可逆実行環境の基本的な導入手法を示した．ここでは，変数の値割当を状態とし，変数更新を状態遷移とし，逆方向で順方向の状態遷移を復元することを目標とした．順方向で計算された値は値スタックに参照が発生したパスとともに記録されているので，逆方向では多くの命令が nop に変換され，ブロック構造は順方向で値スタックに記録されたパスをもとに復元される．

今後の課題として，本発表では実現の観点からバイトコードと実行環境を提案したが，プログラムから正しい可逆計算が可能なバイトコードの生成されていることを形式的に示すことがあげられる．任意のバイトコード列が逆方向計算を表しているわけではない．例えば，jmp や jpc の飛び先には必ず，label がなければならない．また，実行可能なプロセスは値スタックやラベルスタックによって制御されるため，デッドロックを起こすことなく初期状態に至ることは自明ではない．プログラムに正しく対応するバイトコード列が，順方向から逆方向の変換によって順方向の任意の状態遷移を逆方向にすべてたどることができ，最後に無駄な情報を残さないことを示す．

現在の実行環境上では，計算が終了しないと逆方向実行ができないため，デバッグにはそのままでは使うことができない．Hoey らの枠組みにおいてもデバッガへの応用が研究されている [6]．デバッグのためのブレークポイント設定と順方向と逆方向をバイトコード単位で柔軟に実行できるようにすることは，今後の課題である．

謝辞

本研究をすすめるにあたり有益なアドバイスを頂いたレスター大学の Irek Ulidowski 博士に感謝します．また，日頃より議論を頂く名古屋大学の中澤巧爾准教授，関浩之教授，楫勇二教授ならびに結縁・中澤研究室の皆様に感謝いたします．本研究は JSPS 科研費 17H01722 の助成をうけたものです．

参考文献

[1] Agrawal, H., DeMillo, R. A. and Spafford, E. H.: An Execution-Backtracking Approach to Debugging, *IEEE Softw.*, Vol. 8, No. 3, pp. 21–26 (1991).

[2] Axelsen, H. B., Glück, R. and Yokoyama, T.: Reversible Machine Code and Its Abstract Processor Architecture, *Computer Science - Theory and Applications, Second International Symposium on Computer Science in Russia, CSR 2007, Ekaterinburg, Russia, September 3-7, 2007, Proceedings*, Lecture Notes in Computer Science, Vol. 4649, Springer, pp. 56–69 (2007).

[3] Galindo, C., Nishida, N., Silva, J. and Tamarit, S.: ReverCSP: Time-Travelling in CSP Computations, *Reversible Computation - 12th International Conference, RC 2020, Oslo, Norway, July 9-10, 2020, Proceedings*, Lecture Notes in Computer Science, Vol. 12227, Springer, pp. 239–245 (2020).

[4] GNU Project: GDB: The GNU Project Debugger, `https://www.gnu.org/software/gdb/`.

[5] Hoey, J.: Reversing an imparative concurrent programming language, PhD Thesis, University of Leicester (2020).

[6] Hoey, J. and Ulidowski, I.: Reversible Imperative Parallel Programs and Debugging, *Reversible Computation - 11th International Conference, RC 2019, Lausanne, Switzerland, June 24-25, 2019, Proceedings*, Lecture Notes in Computer Science, Vol. 11497, Springer, pp. 108–127 (2019).

[7] Hoey, J., Ulidowski, I. and Yuen, S.: Reversing Imperative Parallel Programs, *Proceedings Combined 24th International Workshop on Expressiveness in Concurrency and 14th Workshop on Structural Operational Semantics, EXPRESS/SOS 2017, Berlin, Germany, 4th September 2017*, EPTCS, Vol. 255, pp. 51–66 (2017).

[8] Hoey, J., Ulidowski, I. and Yuen, S.: Reversing Parallel Programs with Blocks and Procedures, *Proceedings Combined 25th International Workshop on Expressiveness in Concurrency and 15th Workshop on Structural Operational Semantics, EXPRESS/SOS 2018, Beijing, China, September 3, 2018*, EPTCS, Vol. 276, pp. 69–86 (2018).

[9] Ikeda, T. and Yuen, S.: A Reversible Runtime Environment for Parallel Programs, *Reversible Computation - 12th International Conference, RC 2020, Oslo, Norway, July 9-10, 2020, Proceedings*, Lecture Notes in Computer Science, Vol. 12227, Springer, pp. 272–279 (2020).

[10] Lanese, I., Nishida, N., Palacios, A. and Vidal, G.: CauDEr: A Causal-Consistent Reversible Debugger for Erlang, *Functional and Logic Programming - 14th International Symposium, FLOPS 2018, Nagoya, Japan, May 9-11, 2018, Proceedings*, Lecture Notes in Computer Science, Vol. 10818, Springer, pp. 247–263 (2018).

[11] Lanese, I., Nishida, N., Palacios, A. and Vidal, G.: A theory of reversibility for Erlang, *J. Log. Algebraic*

Methods Program., Vol. 100, pp. 71–97 (2018).

[12] Lanese, I., Palacios, A. and Vidal, G.: Causal-Consistent Replay Debugging for Message Passing Programs, *Formal Techniques for Distributed Objects, Components, and Systems - 39th IFIP WG 6.1 International Conference, FORTE 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17-21, 2019, Proceedings*, Lecture Notes in Computer Science, Vol. 11535, Springer, pp. 167–184 (2019).

[13] Lienhardt, M., Lanese, I., Mezzina, C. A. and Stefani, J.: A Reversible Abstract Machine and Its Space Overhead, *Formal Techniques for Distributed Systems - Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE 2012, Stockholm, Sweden, June 13-16, 2012. Proceedings*, Lecture Notes in Computer Science, Vol. 7273, Springer, pp. 1–17 (2012).

[14] Thomsen, M. K. and Axelsen, H. B.: Interpretation and programming of the reversible functional language RFUN, *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages, IFL '15, Koblenz, Germany, September 14-16, 2015*, ACM, pp. 8:1–8:13 (2015).

[15] UNDO: UDB - debugger for C/C++, `https://undo.io/solutions/products/udb/`.

[16] Yokoyama, T.: Reversible Computation and Reversible Programming Languages, *Electron. Notes Theor. Comput. Sci.*, Vol. 253, No. 6, pp. 71–81 (2010).