

# 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<sup>1,a)</sup> 結縁 祥治<sup>1,b)</sup>

**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、実際の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<sup>1,a)</sup> SHOJI YUEN<sup>1,b)</sup>

**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

---

<sup>1</sup> 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) tiked@sqlab.jp

---

b) yuen@sqlab.jp

## 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における出力を得る。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 [1], [2], [3]。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 [4], [5]。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 [6]。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 [6] 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 節で関連研究とまとめを示す。

## 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

```

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

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 引いてしまっている部分を特定することができる。

## 3. 可逆実行環境

本研究では先行研究 [4], [5] 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

```

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

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。

### 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

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

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．

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  $X$  は別物として扱う機能が必要である．そこで先行研究 [4], [5] 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．

### 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

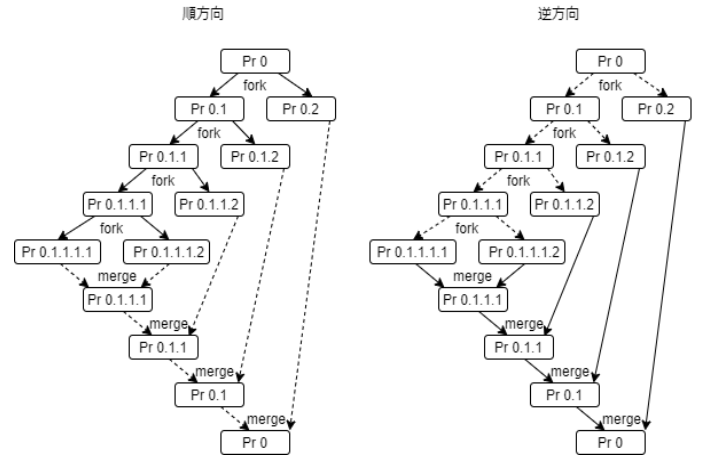


図 3 並列プロセスの生成

Fig. 3 creating parallel process

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．

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．

図 3 に並列プロセスの生成を示している．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 3 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戻り動

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	r_label	0
4	par	{0,1}
5	r_alloc	変数番地
6	free	変数番地
7	r_proc	pn
8	r_return	pn
9	block	bn
10	end	bn
11	r_fork	an
12	merge	an
13	r_w_label	wn
14	w_end	wn
15	nop	0

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。

### 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。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 4 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。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

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

$i(s) = \begin{cases} \epsilon & (s = \epsilon) \\ i(s')inv(c) & (s = cs') \end{cases}$	
$inv(store\ v) = restore\ v,$	$inv(jpc\ a) = r\_label\ 0$
$inv(jmp\ a) = r\_label\ 0,$	$inv(label\ n) = rjmp\ n$
$inv(par\ 0) = par\ 1,$	$inv(par\ 1) = par\ 0$
$inv(alloc\ v) = r\_free\ v,$	$inv(free\ v) = r\_alloc\ v$
$inv(fork\ an) = merge\ an,$	$inv(merge\ an) = r\_fork\ an$
$inv(block\ bn) = end\ bn,$	$inv(end\ bn) = block\ bn$
$inv(proc\ pn) = r\_return\ pn,$	$inv(p\_return\ pn) = r\_proc\ pn$
$inv(func\ fn) = r\_return\ fn,$	$inv(f\_return\ fn) = r\_proc\ pn$
$inv(w\_label\ wn) = w\_end\ wn,$	$inv(w\_end\ wn) = r\_w\_label\ wn$
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

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。

#### 3.2.2 変数更新履歴の利用

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。

#### 3.2.4 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。

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

$\delta$  は、抽象機械が変数を参照するブロック名のパスを示し、 $\chi$  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  $\perp$  となる。 $\rho$ 、 $\xi$ 、 $\sigma$  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。

### 3.3.2 順方向の振舞い定義

プロセス Id  $p$  の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:**  

$$(P, P', w, \delta, \chi, \rho, \xi, \sigma) \xrightarrow{(\text{ipush}, n)}_p (P+1, P, w \cdot n, \delta, \chi, \rho, \xi, \sigma)$$

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:**  

$$(P, P', w, \delta, \chi, \rho, \xi, \sigma) \xrightarrow{(\text{load}, x)}_p (P+1, P, w \cdot \text{lup}(\sigma, x, \delta), \delta, \chi, \rho, \xi, \sigma)$$

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:**  

$$(P, P', w \cdot n, \delta, \chi, \rho, \xi, \sigma) \xrightarrow{(\text{store}, x)}_p (P+1, P, w, \delta, \chi, \rho \cdot (p, \sigma(x)), \xi, \text{sto}(\sigma, x, \delta, n))$$

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。値スタックに保存する前の変数番

地の値を値スタック  $\rho$  にプッシュする。

- **jpc:**  

$$(P, P', w \cdot c, \rho, \xi, \sigma) \xrightarrow{(\text{jpc}, a)}_p \begin{cases} (a, P, w, \rho, \xi, \sigma) & \text{if } c = 1 \\ (P+1, P, w, \rho, \xi, \sigma) & \text{otherwise} \end{cases}$$

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:**  

$$(P, P', w, \delta, \chi, \rho, \xi, \sigma) \xrightarrow{(\text{jmp}, a)}_p (a, P, w, \delta, \chi, \rho, \xi, \sigma)$$

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。
- $(P, P', w \cdot \delta, \chi, n' \cdot n, \rho, \xi, \sigma) \xrightarrow{(\text{op}, m)}_p (P+1, P, w \cdot \text{op}(m)(n', n), \delta, \chi, \rho, \xi, \sigma)$ 

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  $\text{op}(m)$  (+, -, ×, ÷, ==) の演算を行う。
- **label:**  

$$(P, P', w, \delta, \chi, \rho, \xi, \sigma) \xrightarrow{(\text{label}, N)}_p (P+1, P, w, \delta, \chi, \rho, \xi \cdot (p, P'), \sigma)$$

label はラベルスタックに飛び元の番地をプッシュする。
- **par:**  

$$(B, 0, w, \rho, \xi, \sigma) \xrightarrow{(\text{par}, 0)}_{p \cdot i} (B+1, B, w, \rho, \xi, \sigma)$$

$$(E-1, P', w, \rho, \xi, \sigma) \xrightarrow{(\text{par}, 1)}_{p \cdot i} (E, E-1, w, \rho, \xi, \sigma)$$

ここで  $(B, E) = T(p)(i)$ 。par は親プロセス  $p$  の  $\chi$  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:**  

$$(P, P', w, \delta, \chi, \rho, \xi, \sigma) \xrightarrow{(\text{alloc}, x)}_p (P+1, P, w, \delta, \chi, \rho, \xi, \sigma \cup (\delta, x \mapsto 0))$$

alloc は変数  $x$  の領域を  $\sigma$  に追加する。初期値は 0 となっている。
- **free:**  

$$(P, P', w, \delta, \chi, \rho, \xi, \sigma) \xrightarrow{(\text{free}, x)}_p (P+1, P, w, \delta, \chi, \rho, \xi, \sigma - (\delta, x))$$

free は変数番地の解放を行う。
- **proc:**  

$$(P, P', w, \delta, \chi, \rho, \xi, \sigma) \xrightarrow{(\text{proc}, pn)}_p (P+1, P, w \cdot P'+1, \delta \cdot pn, \chi, \rho, \xi \cdot (p, P'), \sigma)$$

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:

$$(P, P', w \cdot a, \delta \cdot pn, \chi, \rho, \xi, \sigma) \xrightarrow{(p\_return, 0)}_p (a + 1, P, \delta, \chi, \rho, \xi, \sigma)$$

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 PC をポップしその PC にジャンプする。

- block:

$$(P, P', w, \delta, \chi, \rho, \xi, \sigma) \xrightarrow{(block, bn)}_p (P + 1, P, w, \delta \cdot bn, \chi, \rho, \xi, \sigma)$$

block はパスに bn を追加する。

- end:

$$(P, P', w, \delta \cdot bn, \chi, \rho, \xi, \sigma) \xrightarrow{(end, bn)}_p (P + 1, P, w, \delta, \chi, \rho, \xi, \sigma)$$

end はパスから bn を削除する。

- fork:

$$(P, P', w, \delta, \perp, \rho, \xi, \sigma) \xrightarrow{(fork, an)}_p (P'', P, w, \delta, an, \rho, \xi, \sigma)$$

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:

$$(P, P', w, \delta, \chi, \rho, an, \sigma) \xrightarrow{(merge, an)}_p (P + 1, P, w, \delta, \perp, \rho, \perp, \sigma)$$

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:

$$(P, P', w \cdot n, \delta, \chi, \rho, \xi, \sigma) \xrightarrow{(func, fn)}_p (P + 1, P, w \cdot P' + 1 \cdot n, \delta \cdot fn, \chi, \rho, \xi \cdot (P', p), \sigma)$$

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 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:

$$(P, P', w \cdot a \cdot r, \delta \cdot pn, \chi, \rho, \xi, \sigma) \xrightarrow{(f\_return, pn)}_p (a, P, w \cdot r, \delta, \chi, \rho, \xi, \sigma)$$

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:

$$(P, P', w, \delta, \chi, \rho, \xi, \sigma) \xrightarrow{(w\_label, wn)}_p (P + 1, P, w, \delta \cdot wn, \chi, \rho, \xi \cdot (p, P'), \sigma)$$

w\_label は while ループの開始点であり、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:

$$(P, P', w, \delta, \chi, \rho, \xi, \sigma) \xrightarrow{(w\_end, wn)}_p (P + 1, P, w, \delta', \chi, \rho, \xi, \sigma)$$

ここで、 $\delta' = rm(\delta, wn)$

w\_end はパスから wn を全て削除する。

- nop:

$$(P, P', w, \delta, \chi, \rho, \xi, \sigma) \xrightarrow{(nop, 0)}_p (P + 1, P, w, \delta, \chi, \rho, \xi, \sigma)$$

nop は何も操作を行わない。

### 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:

$$(P, P', w, \delta, \chi, \rho, \xi \cdot (p, a), \sigma) \xrightarrow{(rjmp, N)}_p (N + 1 - a, P, w, \delta, \chi, \rho, \xi, \sigma)$$

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  $a$  のアドレスは逆方向では、 $N + 1 - a$  となる。

- restore:

$$(P, P', w, \delta, \chi, \rho \cdot (p, n), \xi, \sigma) \xrightarrow{(restore, x)}_p (P + 1, P, w, \delta, \chi, \rho, \xi, upd(\sigma, \delta, x, n))$$

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:

$$(P, P', w, \delta, \chi, \rho, \xi, \sigma) \xrightarrow{(r\_label)}_p (P + 1, P, w, \delta, \chi, \rho, \xi, \sigma)$$

r\_label は rjmp のターゲットとなる。

- par:

$$(B, 0, w, \rho, \xi, \sigma) \xrightarrow{(par, 0)}_{p \cdot i} (B + 1, B, w, \rho, \xi, \sigma) \\ (E - 1, P', w, \rho, \xi, \sigma) \xrightarrow{(par, 1)}_{p \cdot i} (E, E - 1, w, \rho, \xi, \sigma)$$

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  $\chi$  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:

[to be filled: 逆方向計算の定義を修正]

- free:

$$(P, P', w, \delta, \chi, \rho, \xi, \sigma) \xrightarrow{(free, x)}_p$$

$(P + 1, P, w, \delta, \chi, \rho, \xi, \sigma - (\delta, x)$   
free は変数番地の解放を行う。

- r\_proc:

$(P, P', w, \delta, \chi, \rho, \xi, \sigma) \xrightarrow{(r\_proc, pn)}_p$   
 $(P + 1, P, w, \delta \cdot pn, \chi, \rho, \xi, \sigma)$

手続きの始まりを表す。パスに pn を追加する。

- r\_return:

$(P, P', w, \delta \cdot pn, \chi, \rho, \xi, \sigma) \xrightarrow{(r\_return, pn)}_p$   
 $(P + 1, P, w, \delta \cdot pn, \chi, \rho, \xi, \sigma)$

手続きの終了を表す。パスから pn を削除する。

- block:

$(P, P', w, \delta, \chi, \rho, \xi, \sigma) \xrightarrow{(block, bn)}_p$   
 $(P + 1, P, w, \delta \cdot bn, \chi, \rho, \xi, \sigma)$

block はパスに bn を追加する。

- end:

$(P, P', w, \delta \cdot bn, \chi, \rho, \xi, \sigma) \xrightarrow{(end, bn)}_p$   
 $(P + 1, P, w, \delta, \chi, \rho, \xi, \sigma)$

end はパスから bn を削除する。

- r\_fork:

$(P, P', w, \delta, \perp, \rho, \perp, \sigma) \xrightarrow{(fork, an)}_p$   
 $(P'', P, w, \delta, an, \rho, \xi, \sigma)$

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:

$(P, P', w, \delta, \chi, \rho, an, \sigma) \xrightarrow{(merge, an)}_p$   
 $(P + 1, P, w, \delta, \perp, \rho, an, \sigma)$

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。

- r\_w\_label:

- r\_w\_end:

### 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 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。

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

### 3.4 プログラム例

図 5 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 6 は図 5 を命令を一对一で置換し順番を反転させた逆方向実行のバイトコードである。図 5 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 6 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



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 0
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 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

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ている。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

```
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)

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 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をすることを考える。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。

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)

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

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
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 10 ラベルスタック (bug\_fact)

Fig. 10 label stack(bug\_fact)

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 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

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 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)

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。

## 5. 関連研究

並列プログラムに対する逆方向実行について、先行研究 [4], [5] によって提案されている手法について説明する。

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

図 12 対象プログラムの定義

Fig. 12 definition of language

先行研究 [4], [5] 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。

### 5.1 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 12 のように定義する。

### 5.2 Annotated プログラム

図 12 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 13 のプログラムから図 14 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 14 を実行すると図 15 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 15 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。

<pre> begin b1   proc p1 fib is     begin b2       var T = 0 b2;       if i1 (N-2 &gt; 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 </pre>	<pre> begin b1   proc p1 fib is     begin b2       var T = 0 (b2*b1,A);       if i1 (N-2 &gt; 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 </pre>
---	---

図 13 対象プログラム

図 14 Annotated プログラム

Fig. 13 Original Program

Fig. 14 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 15 実行された Annotated プログラム  
(2 回目の手続き呼び出し)Fig. 15 Executed Annotated Program  
(second procedure call)

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。

### 5.3 Inverted プログラム

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実行された Annotated Program から変換された 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 15 の実行された Annotated プログラムを Inverted Program へ変換したものが図 16 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。

```

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 16 Inverted プログラム  
(2 回目の手続き呼び出しの部分)Fig. 16 Inverted Program  
(part of second procedure call)

## 6. おわりに

### 参考文献

- [1] 1
- [2] 2
- [3] 3
- [4] 4
- [5] 5
- [6] 6