A runtime environment for reversible parallel programs

Takashi Ikeda **Shoji Yuen**

Nagoya University

July 10, 2020

Background

Reversible execution of programs.

- Debugging parallel programs is not that easy.
  Replay may select different interleaving combination.
- Additional information is kept in order to recover the computation.
- Stack Annotation[HoeyUlidowski17, HoeyUlidowskiY18, HoeyUlidowski19]
- Runtime by reversible abstract machine for process calculi[Lienhardt+12] .

Backtrack reversibility:

Keeping the log of the detailed interleaving

tracking back according to the log.

Motivation

- Inspired by [HoeyUlidowski19] and Hoey's PhD work.
- Reversing simple parallel programs at high-level with simple block constructs.
- In [HoeyUlidowski19], the operational semantics is defined by SOS rules for syntactic constructs with annotations. (forward/backward)
- The execution mechanism is usually much simpler with variable UPDATES controlled by GOTO's. Not-structured.

Aiming at simpler implementation of reversible runtime:

- Goto control structures
- Preserving variable updates history
- Built-in Concurrency (Python Library)

Contents

- A simple programming language with parallel blocks
- Reversing Jumps and Updates
- Demo (7min 30sec video)
- Concluding remarks

Motivating Example[HoeyUlidowski19]

Airline Ticket Sales simulation:

Two travel agents sell 3 seats.

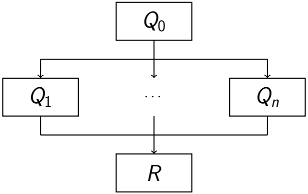
The agents sell 4 seats due to race.

When seats=1, P1 at line 9 and P2 at line 17.

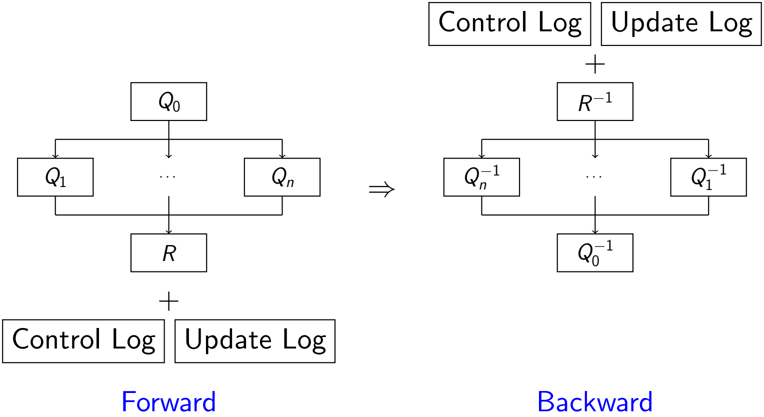
```
 1: var seats;
 2: var agent1;
 3: var agent2;
 4: seats=3;
 5: agent1=1;
 6: agent2=1;
 7: par{
 8:    while (agent1==1) do
 9:        if (seats>0) then
10:            seats=seats-1;
11:        else
12:            agent1=0;
13:        fi;
14:    od
15: }{
16:    while (agent2==1) do
17:        if (seats>0) then
18:            seats=seats-1;
19:        else
20:            agent2=0;
21:        fi;
22:    od
23: }
24: remove agent2;
25: remove agent1;
26: remove seats;
```

Programming Language with Parallel Composition

$$P ::= DQR \mid DQ \ \texttt{par} \ \{Q\}(\{Q\})^{+}R$$
$$D ::= (\texttt{var} \ X\,;)^{*}$$
$$R ::= (\texttt{remove} \ X\,;)^{*}$$
$$Q ::= (S;)^{*}S$$
$$S ::= \texttt{skip} \mid X\texttt{=}E \mid \texttt{if} \ C \ \texttt{then} \ Q \ \texttt{else} \ Q \ \texttt{fi} \mid \texttt{while} \ C \ \texttt{do} \ Q \ \texttt{od}$$
$$E ::= X \mid n \mid E \ op \ E \mid (E)$$
$$C ::= B \mid C \ \texttt{\&\&} \ C \mid \texttt{not} \ C \mid (C)$$
$$B ::= E \ \texttt{==} \ E \mid E \ \texttt{<} \ E$$



Reversible Execution of Programs

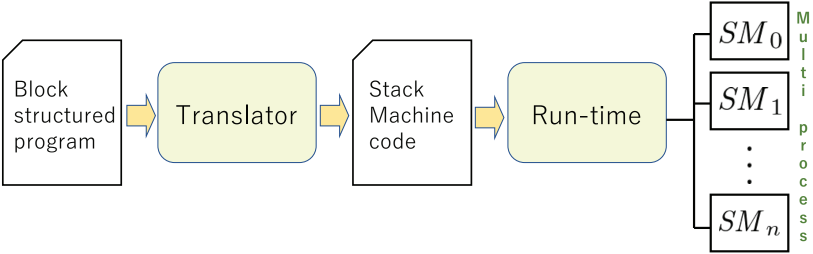


Just reverse the forward executions to be backward

Additional information needs be kept in extra memory.

Lower Level execution with concurrency

- Each block is sequentially executed by a stack machine.
- The backward computation is executed by the same stack machine with additional information.
- The operational semantics does not change for both directions.



Stack Machine Code (Operations)

Forward	**ipush**	i	Load immediate i
	load	v	Load v to the stack top
	store	v	Store the stack top to v and pushes the previous to the value stack.
	jpc	a	Jump to a if the stack top is 0
	jmp	a	Jump to a always
	op	n	Apply op_n to the stack
Backward	**rjmp**		Reverse jump. Jump to the address popped of the label stack.
	restore	v	Pops the value of v from the value stack.

Stack Machine Code(Directives)

label	n	Target of jmp and jpc and pushes the address to the label stack where n is the program length.
alloc	v	Alloc v to the environment and set v popping from the initial stack.
free	v	Dealloc v from the environment and push v to the final stack.

par	0	Beginning of a parallel block. Allocate one stack machine.
par	1	End of a parallel block. Synchronize termination with other parallel blocks.

Code Inversion

Generate the code for backward from forward

s: *Forward* $\Rightarrow \mathtt{i}(s)$: *Backward*

$$\mathtt{i}(s) = \begin{cases} \varepsilon & \text{if } s = \varepsilon \\ \mathtt{i}(s')\mathit{inv}(c) & \text{if } s = cs' \end{cases}$$

$$\begin{array}{ll} \mathtt{inv}(\langle \mathtt{store}\ v \rangle) = \langle \mathtt{restore}\ v \rangle, & \mathtt{inv}(\langle \mathtt{jpc}\ a \rangle) = \langle \mathtt{label}\ 0 \rangle, \\ \mathtt{inv}(\langle \mathtt{jmp}\ a \rangle) = \langle \mathtt{label}\ n \rangle, & \mathtt{inv}(\langle \mathtt{label}\ n \rangle) = \langle \mathtt{rjmp}\ 0 \rangle, \\ \mathtt{inv}(\langle \mathtt{par}\ 0 \rangle) = \langle \mathtt{par}\ 1 \rangle, & \mathtt{inv}(\langle \mathtt{par}\ 1 \rangle) = \langle \mathtt{par}\ 0 \rangle, \\ \mathtt{inv}(\langle \mathtt{alloc}\ v \rangle) = \langle \mathtt{free}\ v \rangle, & \mathtt{inv}(\langle \mathtt{free}\ v \rangle) = \langle \mathtt{alloc}\ v \rangle \end{array}$$

For other c, $\mathtt{inv}(\langle \mathtt{c}\ n \rangle) = \langle \mathtt{nop}\ 0 \rangle$

Stack Machine Behaviour:A Simple Example

```
var x;
x = 0;
par {
 x = 2;
}{
 x = 1;
 x = x + 1;
}
remove x
```

At termination, `x` is either 2 or 3.

`x` is 3, when `x=2` is executed between `x=1` and `x=x+1`.

`x` is 2, otherwise.

Stack Machine Behaviour:A Simple Example

```
var x;
x = 0;
par {
 x = 2;
}{
 x = 1;
 x = x + 1;
}
remove x
```

At termination, x is either 2 or 3.

x is 3, when x=2 is executed between x=1 and x=x+1.

x is 2, otherwise.

Stack Machine Behaviour:A Simple Example

```
var x;
x = 0;
par {
  x = 2;
}{
  x = 1;
  x = x + 1;
}
remove x
```

```
alloc  0
ipush  0
store  0
  par  0
ipush  2
store  0
  par  1
  par  0
ipush  1
store  0
 load  0
ipush  1
   op  0
store  0
  par  1
 free  0
```

Forward Code

At termination, x is either 2 or 3.

x is 3, when x=2 is executed between x=1 and x=x+1.

x is 2, otherwise.

Stack Machine Behaviour:A Simple Example

```
var x;
x = 0;
par {
  x = 2;
}{
  x = 1;
  x = x + 1;
}
remove x
```

Forward Code		Backward Code	
alloc	0	alloc	0
ipush	0	par	0
store	0	restore	0
par	0	nop	0
ipush	2	nop	0
store	0	nop	0
par	1	restore	0
par	0	nop	0
ipush	1	par	1
store	0	par	0
load	0	restore	0
ipush	1	nop	0
op	0	par	1
store	0	restore	0
par	1	nop	0
free	0	free	0

At termination, x is either 2 or 3.

x is 3, when x=2 is executed between x=1 and x=x+1.

x is 2, otherwise.

Stack Machine Behaviour: Case 1 (x=2)

Forward execution:	Backward execution:

3:Store(p_0)
$\rightarrow$ 6:store(p_1)
$\rightarrow$ 10:store(p_2)
$\rightarrow$ 14:store(p_2)

1	alloc	0 ←
2	par	0
3	restore	0
4	nop	0
5	nop	0
6	nop	0
7	restore	0
8	nop	0
9	par	1
10	par	0
11	restore	0
12	nop	0
13	par	1
14	restore	0
15	nop	0
16	free	0

Value stack

$\langle 1, 2 \rangle$
$\langle 2, 2 \rangle$
$\langle 0, 1 \rangle$
$\langle 0, 0 \rangle$

`x = 2`

Stack Machine Behaviour: Case 1 (x=2)

Forward execution:	Backward execution:

3:Store(p_0)
→ 6:store(p_1)
→ 10:store(p_2)
→ 14:store(p_2)

3:restore(p_2)

1	alloc	0
2	par	0
3	restore	0 ←
4	nop	0
5	nop	0
6	nop	0
7	restore	0
8	nop	0
9	par	1
10	par	0
11	restore	0 ←
12	nop	0
13	par	1
14	restore	0
15	nop	0
16	free	0

Value stack

$\langle 1, 2 \rangle$ ←
$\langle 2, 2 \rangle$
$\langle 0, 1 \rangle$
$\langle 0, 0 \rangle$

x = 1

Stack Machine Behaviour: Case 1 (x=2)

Forward execution:	Backward execution:

3:Store(p_0)
→ 6:store(p_1)
→ 10:store(p_2)
→ 14:store(p_2)

1	alloc	0
2	par	0
3	restore	0
4	nop	0
5	nop	0
6	nop	0
7	restore	0 ←
8	nop	0
9	par	1
10	par	0
11	restore	0 ←
12	nop	0
13	par	1
14	restore	0
15	nop	0
16	free	0

3:restore(p_2)
→ 7:restore(p_2)

Value stack

$\langle 2, \mathbf{2} \rangle$ ←
$\langle 0, \mathbf{1} \rangle$
$\langle 0, 0 \rangle$

x = 2

Stack Machine Behaviour: Case 1 (x=2)

Forward execution: **Backward execution:**

3:Store(p_0)
→ 6:store(p_1)
→ 10:store(p_2)
→ 14:store(p_2)

1	alloc	0
2	par	0
3	restore	0
4	nop	0
5	nop	0
6	nop	0
7	restore	0
8	nop	0
9	par	1
10	par	0
11	restore	0
12	nop	0
13	par	1
14	restore	0
15	nop	0
16	free	0

3:restore(p_2)
→ 7:restore(p_2)
→ 11:restore(p_1)

Value stack

$\langle 0, 1 \rangle$ ←
$\langle 0, 0 \rangle$

x = 0

Stack Machine Behaviour: Case 1 (x=2)

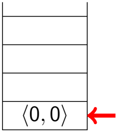
Forward execution:	Backward execution:

$3{:}\texttt{Store}(p_0)$
$\rightarrow 6{:}\texttt{store}(p_1)$
$\rightarrow 10{:}\texttt{store}(p_2)$
$\rightarrow 14{:}\texttt{store}(p_2)$

1	alloc	0
2	par	0
3	restore	0
4	nop	0
5	nop	0
6	nop	0
7	restore	0
8	nop	0
9	par	1
10	par	0
11	restore	0
12	nop	0
13	par	1
14	restore	0 ←
15	nop	0
16	free	0

$3{:}\texttt{restore}(p_2)$
$\rightarrow 7{:}\texttt{restore}(p_2)$
$\rightarrow 11{:}\texttt{restore}(p_1)$
$\rightarrow 14{:}\texttt{restore}(p_0)$

Value stack



`x = 0`

Stack Machine Behaviour: Case 2 (x=3)

Forward execution: | **Backward execution:**

3:$\texttt{Store}(p_0)$
$\rightarrow$ 10:$\texttt{store}(p_2)$
$\rightarrow$ 6:$\texttt{store}(p_1)$
$\rightarrow$ 14:$\texttt{store}(p_2)$

1	alloc	0 ←
2	par	0
3	restore	0
4	nop	0
5	nop	0
6	nop	0
7	restore	0
8	nop	0
9	par	1
10	par	0
11	restore	0
12	nop	0
13	par	1
14	restore	0
15	nop	0
16	free	0

Value stack

$\langle 2, \mathbf{2} \rangle$
$\langle 1, \mathbf{1} \rangle$
$\langle 0, \mathbf{2} \rangle$
$\langle 0, 0 \rangle$

`x = 3`

Stack Machine Behaviour: Case 2 (x=3)

Forward execution:

3:Store(p_0)
→ 10:store(p_2)
→ 6:store(p_1)
→ 14:store(p_2)

Value stack

$\langle 2, \mathbf{2} \rangle$ ←
$\langle 1, \mathbf{1} \rangle$
$\langle 0, \mathbf{2} \rangle$
$\langle 0, 0 \rangle$

x = 2

Backward execution:

1	alloc	0
2	par	0
3	restore	0 ←
4	nop	0
5	nop	0
6	nop	0
7	restore	0
8	nop	0
9	par	1
10	par	0
11	restore	0 ←
12	nop	0
13	par	1
14	restore	0
15	nop	0
16	free	0

3:restore(p_2)

Stack Machine Behaviour: Case 2 (x=3)

Forward execution: **Backward execution:**

3:Store(p_0)
→ 10:store(p_2)
→ 6:store(p_1)
→ 14:store(p_2)

1	alloc	0
2	par	0
3	restore	0
4	nop	0
5	nop	0
6	nop	0
7	restore	0
8	nop	0
9	par	1
10	par	0
11	restore	0
12	nop	0
13	par	1
14	restore	0
15	nop	0
16	free	0

3:restore(p_2)
→ 11:restore(p_1)

Value stack

$\langle 1, 1 \rangle$ ←
$\langle 0, 2 \rangle$
$\langle 0, 0 \rangle$

x = 1

Stack Machine Behaviour: Case 2 (x=3)

Forward execution:

3:`Store`(p_0)
→ 10:`store`(p_2)
→ 6:`store`(p_1)
→ 14:`store`(p_2)

Backward execution:

3:`restore`(p_2)
→ 11:`restore`(p_1)
→ 7:`restore`(p_2)

1	alloc	0
2	par	0
3	restore	0
4	nop	0
5	nop	0
6	nop	0
7	restore	0
8	nop	0
9	par	1
10	par	0
11	restore	0
12	nop	0
13	par	1
14	restore	0
15	nop	0
16	free	0

Value stack

$\langle 0, \mathbf{2} \rangle$
$\langle 0, 0 \rangle$

`x = 0`

Stack Machine Behaviour: Case 2 (x=3)

Forward execution: **Backward execution:**

$3{:}\texttt{Store}(p_0)$
$\rightarrow 10{:}\texttt{store}(p_2)$
$\rightarrow 6{:}\texttt{store}(p_1)$
$\rightarrow 14{:}\texttt{store}(p_2)$

1	alloc	0
2	par	0
3	restore	0
4	nop	0
5	nop	0
6	nop	0
7	restore	0
8	nop	0
9	par	1
10	par	0
11	restore	0
12	nop	0
13	par	1
14	restore	0 ←
15	nop	0
16	free	0

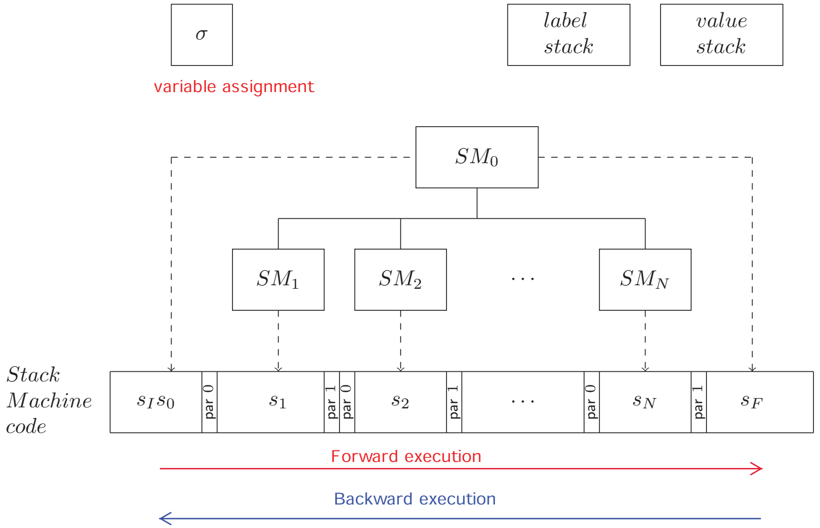
$3{:}\texttt{restore}(p_2)$
$\rightarrow 11{:}\texttt{restore}(p_1)$
$\rightarrow 7{:}\texttt{restore}(p_2)$
$\rightarrow 14{:}\texttt{restore}(p_0)$

Value stack

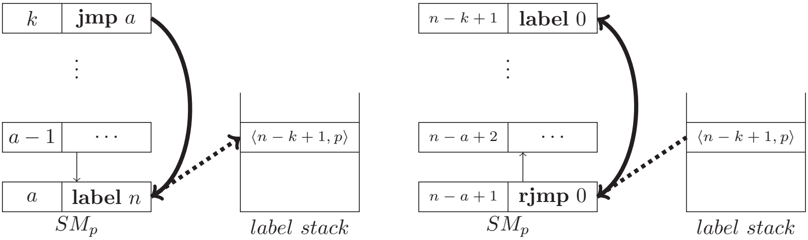
$\langle 0, 0 \rangle$ ←

`x = 0`

Execution by Stack Machines



Reversing Jumps

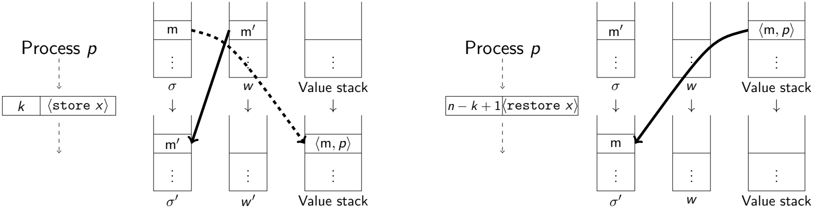


(a) Forward jump (b) Backward jump

(a) `label` remembers the control to push the address of the jump origin with the process number *p*.

(b) `rjmp` jumps back by poping up the address to return on process *p*.

Reversing Updates



(a) Forward store (b) Backward restore

(a) `store` updates the environment from σ to σ' popping the stack top and store the old value to the value stack along with the process number p.

(b) `restore` pops the old value from the value stack for the recorded process p.

Runtime Semantics

$$\frac{PC^1 \in s_0, s_I, s_F, (PC^1, PC'^1, w^1, \rho^1, \xi^1)_{\sigma^1} \xrightarrow{s(PC^1)}_{(0,N)} (PC^2, PC'^2, \rho^2, \xi^2)_{\sigma^2}}{\langle \mathsf{Exec}_s(PC^1, PC'^1, w^1), \rho^1, \xi^1\rangle \to \langle \mathsf{Exec}_s(PC^2, PC'^2, w^2), \rho^2, \xi^2\rangle} \text{[Init]}$$

$$\frac{s(PC_0) = \langle \mathtt{par}\ 0\rangle}{\begin{array}{l}\langle \mathsf{Exec}_s(PC_0, PC'_0, w_0), \rho_0, \xi_0\rangle \to \\ \quad \langle \mathsf{Exec}^1_s(loc(s_1), PC_0, \varepsilon) \| \cdots \| \mathsf{Exec}^N_s(loc(s_N), PC_0, \varepsilon), \sigma, \rho_0, \xi_0\rangle\end{array}} \text{[Fork]}$$

$$\frac{(PC^1_p, PC'^1_p, w_p, \rho, \xi)_\sigma \xrightarrow{s(PC^1_p)}_{(p,N)} (PC^2_p, PC'^2_p, w'_p, \rho', \xi')_{\sigma'}, PC^1_p \in s_p}{\begin{array}{l}\langle \mathsf{Exec}^1_s(PC_1, PC'_1, w_1) \| \cdots \| \mathsf{Exec}^p_s(PC^1_p, PC'^1_p, w_p) \| \cdots \| \mathsf{Exec}^N_s(PC_N, PC'_N, \varepsilon), \sigma, \rho, \xi\rangle \\ \to \langle \mathsf{Exec}^1_s(PC_1, PC'_1, w_1) \| \cdots \| \mathsf{Exec}^p_s(PC^2_p, PC'^2_p, w'_p) \| \cdots \| \mathsf{Exec}^N_s(PC_N, PC'_N, \varepsilon), \sigma', \rho', \xi'\rangle\end{array}} \text{[Par]}$$

$$\frac{\wedge_p s(PC_p) = \langle \mathtt{par}\ 1\rangle}{\langle \mathsf{Exec}^1_s(PC_1, PC'_1, w_1) \| \cdots \| \mathsf{Exec}^N_s(PC_N, PC'_N, w_N), \sigma, \rho, \xi\rangle \to \langle \mathsf{Exec}_s(loc(s_F), 0, \sigma), \rho, \xi'\rangle} \text{[Merge]}$$

Implementation

- Stack machine code generation:
  - Translator from Block-structured programs to Stack Machine codes is implemented by JAVACC.
- Stack Machine by Python
  - Running SMs with Python Multi-process module.
  - Execute VMs backward poping the value stack and the label stack.
  - `nops` are executed concurrently.

Demo

https://github.com/syuen1/RevRunTimeEnv

- Running the airline example with our SMs.
- For better understanding, we present an experimental GUI.

Concluding remarks

Summary:

- Executing a simple parallel program with Stack Machines
- Stack machine codes are flat and not-structured.
- Code for a backward execution are generated from the code for the forward execusion by changing the operations one by one and reversing the code sequence.
- Backward execution only preserve updates points. Other operations are discarded (unlike JANUS).
  Enough for the basic debugging?

Future work:

- Development of a debugger.
  Moving forward/backward at update points.
- Extend the parallel program syntax.
  Recursion with dynamic process invocation
- Show the basic properties for reversibility.