A Reversible Runtime Environment for Parallel Programs

Takashi Ikeda and Shoji Yuen(✉)

Graduate School of Informatics, Nagoya University,
Furo-cho, Chikusa-ku, Nagoya 464-8601, Japan
{tikeda,yuen}@sqlab.jp

Abstract. We present a reversible runtime environment for simple parallel programs and its experimental implementation. We aim at a lightweight implementation of the *backtrack reversibility* by the *state-saving* mechanism using stacks. We translate a program to a sequence of simple commands of an executable intermediate representation for reversible stack machines. The parallel composition is implemented using the multiprocessing feature of Python. While executing the commands, the stack machines collect the information for the backward execution in the auxiliary stacks for the update history of the variables and the history of jumps. The commands for the backward execution is obtained by reversing the commands for the forward execution by replacing each command with the corresponding reversed command. In the purpose of behaviour analysis with reversibility such as debugging, our runtime is more portable than the source-to-source translation of a high-level programming language.

Keywords: Reversible computation · Imparative parallel programs · Stack machine · Python multiprocessing

1 Introduction

Reverse execution of programs has been investigated based on the reversible computing recently. Undoing the effect of an execution of a program till returning to the initial state is useful in analysing the finer-grained behavioural property of the program. In general, the execution of a parallel program depends on the environment such as the scheduler and I/O channels. Replaying the program may not reach the same states as the previous run. This makes behavioural analysis difficult to work out the cause of the defect for debugging.

Reversible programming languages such as Janus [4,7] and RFUN [6] are designed for the reversed execution at the level of the design of programming languages. For example, Janus needs the extra-control structure at the end of the conditional branch in order to know which branch is executed to reverse the conditional statement. For this approach, the state-saving mechanism is not needed since the computation is fully reversed. However, introducing parallel composition becomes difficult since the runtime environment is not directly described in programs.

We present a reverse execution mechanism that the runtime collects the information in stacks at a forward execution. At the reverse execution, the runtime executes the program simply in the reversed order using the information stored in the stacks. A source program is compiled to a sequence of simple commands executed by the stack machine. Each process in the parallel composition is dispatched to each stack machine forked from the initial stack machine. We implement the backtrack reversibility with the multiprocessing feature of Python. In the reverse execution, multiple stack machines are invoked, but they are controlled by the stacks to follow back the forward execution.

The report is structured as follows. Section 2 gives the syntax of the programs, Sect. 3 presents the stack machine design and Sect. 4 states the concluding remarks.

2 Programming Language with Parallel Composition

Our parallel programming language is defined as follows where $()^+$ and $()^*$ denotes the repetition of one or more times and zero or more times respectively:

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

The language is a simplified version of that in [2,3]. `par` denotes the parallel composition of sequential procedures. For simplicity, we remove the nested block structure and procedures. `remove` statements at the end of a program correspond to the variables declared at the beginning of the program, where the order of declarations is supposed to be reversed. For example, if variables are declared as `var` X`;` `var` Y, the variables are removed as `remove` Y`;` `remove` X. This ensures the correspondence between the variable and the entry of the symbol table.

3 Reversible Execution of Stack Machine Code

3.1 Reversible Stack Machine

For simplicity, a parallel program in this report is limited in the form that an initial stack machine runs first followed by parallel blocks. For values, we consider the integers Z. $\mathbb{A}$ is the set of *address* $\mathbb{A}$ as the locations of commands in a stack machine code. Here an address is a positive natural number. $\mathbb{P}$ is the set of stack machine identifiers (SMid). An SMid is a natural number. We assume the initial stack machine has the SMid of 0. Other stack machines have id's in a row from 1 to N where N is the number of parallel blocks.

The stack machine configuration is $(PC, PC', w, \rho, \xi)_\sigma$ where PC is the program counter, PC' holds the previous PC value, $w \in \mathbb{Z}^*$ is a local stack, $\rho \in (\mathbb{A} \times \mathbb{P})^*$ is a label stack, and $\xi \in (\mathbb{Z} \times \mathbb{P})^*$ is a value stack. σ is a symbol table that maps a variable to its value. $\sigma(v)$ presents the value of v. For the local stack w and $z \in \mathbb{Z}$, zw is the concatenation of z and w.

Each stack machine is identified by (p, N) with p is a process identifier and N is a number of all parallel blocks in a program. The behaviour of a stack machine $SM_{(p,N)}$ for command c is specified by $\xrightarrow{c}_{(p,N)}$ as follows:

nop: $(PC_p, PC'_p, w_p, \rho, \xi)_\sigma \xrightarrow{\langle \texttt{nop 0}\rangle}_{(p,N)} (PC_p + 1, PC_p, w_p, \rho, \xi)_\sigma$
$\langle$`nop` 0$\rangle$ does nothing but increasing the program counter.

ipush: $(PC_p, PC'_p, w_p, \rho, \xi)_\sigma \xrightarrow{\langle \texttt{ipush } z\rangle}_{(p,N)} (PC_p + 1, PC_p, zw_p, \rho, \xi)_\sigma$
$\langle$`ipush` $z\rangle$ pushes an immediate value of z to the local stack.

load: $(PC_p, PC'_p, w_p, \rho, \xi)_\sigma \xrightarrow{\langle \texttt{load } v\rangle}_{(p,N)} (PC_p + 1, PC_p, \sigma(v)w_p, \rho, \xi)_\sigma$
$\langle$`load` $v\rangle$ puts a value of v on the top of the local stack.

store: $(PC_p, PC'_p, zw_p, \rho, \xi)_\sigma \xrightarrow{\langle \texttt{store } v\rangle}_{(p,N)} (PC_p + 1, PC_p, w_p, \rho, \langle \sigma(v), p\rangle\xi)_{\sigma[v \mapsto z]}$
$\langle$`store` $v\rangle$ pops a value from the local stack and store the value z to the local storage σ *after* saving the previous value $\sigma(v)$ to the value stack along with the process number.

jpc: $(PC_p, PC'_p, zw_p, \rho, \xi)_\sigma \xrightarrow{\langle \texttt{jpc } a\rangle}_{(p,N)} \begin{cases} (a, PC_p, w_p, \rho, \xi)_\sigma & \text{if } z \neq 0 \\ (PC_p + 1, PC_p, w_p, \rho, \xi)_\sigma & \text{if } z = 0 \end{cases}$
$\langle$`jpc` $a\rangle$ jumps to a when the stack top is 0. Otherwise, it moves to the next instruction by increasing the program counter.

jmp: $(PC_p, PC'_p, w_p, \rho, \xi)_\sigma \xrightarrow{\langle \texttt{jmp } a\rangle}_{(p,N)} (a, PC_p, w_p, \rho, \xi)_\sigma$
$\langle$`jmp` $a\rangle$ jumps to a unconditionally.

op: $(PC_p, PC'_p, z_1 z_2 w_p, \rho, \xi)_\sigma \xrightarrow{\langle \texttt{op } k\rangle}_{(p,N)} (PC_p + 1, PC_p, \texttt{op}_k(z_1, z_2)w_p, \rho, \xi)_\sigma$
where $\texttt{op}_1 \equiv +, \texttt{op}_2 \equiv \times, \texttt{op}_3 \equiv -, \texttt{op}_4 \equiv <, \texttt{op}_5 \equiv ==$.
$z_1 < z_2$ and $z_1 == z_2$ are 1 when the relations hold and 0 otherwise.
$\langle$`op` $k\rangle$ applies the operation specified by k. Depending on k, it pops two or one from the local stack and pushes the result on the local stack. When $\langle$`op` $k\rangle$ is a relation, it pushes 1 when the relation holds and pushes 0 otherwise.

label: $(PC_p, PC'_p, w_p, \rho, \xi)_\sigma \xrightarrow{\langle \texttt{label } n\rangle} (PC_p + 1, PC_p, w_p, \langle n + 1 - PC'_p, p\rangle\rho, \xi)$
$\langle$`label` $n\rangle$ pushes the address for backward execution to the label stack where n is the number of all instructions. **label** is the only instruction that uses PC'_p.

rjmp: $(PC_p, PC'_p, w_p, \langle a, N + 1 - i\rangle\rho, \xi)_\sigma \xrightarrow{\langle \texttt{rjmp 0}\rangle}_{(p,N)} (a, PC_p, w_p, \rho, \xi)_\sigma$
$\langle$`rjmp` 0$\rangle$ is a reverse jump that pops an address and a stack machine number from the label stack and jump back to the address on that process with the number.

restore: $(PC_p, PC'_p, w_p, \rho, \langle z, N + 1 - i\rangle\xi)_\sigma \xrightarrow{\langle \texttt{restore } v\rangle}_{(p,N)} (PC_p + 1, PC_p, zw_p, \rho, \xi)_{\sigma[v \mapsto z]}$
$\langle$`restore` $v\rangle$ pops the value of v and the stack machine number from the value stack on the specified stack machine.

alloc: $(PC_p, PC'_p, w_p, \rho, \xi)_\sigma \xrightarrow{\langle \texttt{alloc } v\rangle}_{(p,N)} (PC_p + 1, PC_p, w_p, \rho, \xi)_{\sigma[v \mapsto 0]}$
$\langle$`alloc` $v\rangle$ adds v to the environment σ and initialises v.

free: $(PC_p, PC'_p, w_p, \rho, \xi)_\sigma \xrightarrow{\langle \texttt{free } v\rangle}_{(p,N)} (PC_p + 1, PC_p, w_p, \rho, \xi)_{\sigma \setminus v}$
$\langle$`free` $v\rangle$ removes v from the environment σ.

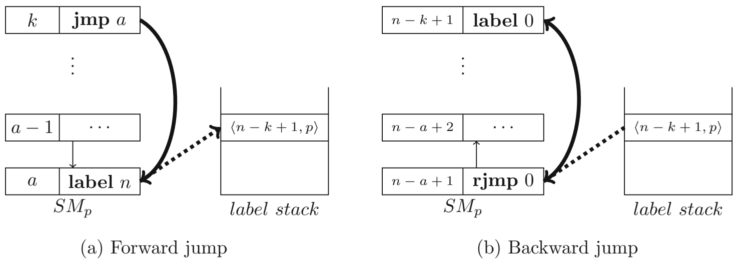

Fig. 1. Reversing jump instructions

`store` and `label` collect the information in a forward execution. `store` updates σ and ξ. In ξ, it records the value is stored by p. But in the backward execution, the process number is also reversed, and it stores $N+1-p$ as the backward process number. `label` records from which address the control reach this place.

`rjmp` and `restore` restore the information for a backward execution. `rjmp` corresponds to `label` and pops the location from ρ and jump back to the location where the forward execution came from. And `restore` puts back the previous value from the value stack. In both cases, the stack machine must be identified where the SMid is inverted since the order of the parallel blocks is reversed[1].

Figure 1 shows the mechanism of **label** and **rjmp**. **label** is a destination of **jpc** and **jmp**. If **label** is executed, it pushes the source address of that jump to the label stack. In the backward execution, **label** is substituted by **rjmp**. By popping the label stack, one of the stack machines executes **rjmp** and jumps back to the source address.

`alloc` allocates a variable slot on the stack and updates the symbol table. `free` removes a variable slot. $\sigma\backslash v$ removes v from the domain of σ. In current target codes, `alloc` and `free` are executed only by the initial stack machine with id 0.

3.2 Inverting Stack Machine Code

We do not present the detailed translation from a source program to the stack machine code here. The translator is implemented using Javacc. In the translation, `label` is inserted at a target of `jmp` and `jpc`. The argument of `label` is the length of the generated code. Since this is not known until the whole translation is done, it can be specified by back-patching. `par` 0 and `par` 1 are inserted for a parallel block.

From a forward stack machine code s, the backward code $\mathtt{i}(s)$ is obtained:

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

[1] Since an address of command is uniquely assigned to a unique stack machine, it is not essential to record p in ρ. Without p in ρ, another table is necessary.

where $\texttt{inv}_n$ for each command is defined as below where n is the :

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

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

3.3 Execution from the Initial Stack Machine

Let s be the forward stack machine code. From the construction of a program, s is partitioned to:

$$s_I s_0 \langle\texttt{par}\ 0\rangle s_1 \langle\texttt{par}\ 1\rangle \cdots \langle\texttt{par}\ 0\rangle s_N \langle\texttt{par}\ 1\rangle s_F$$

where M is the length of the code and $SM_{(0,N)}$ executes s_I, s_0, and s_F where s_I and s_F are **alloc** and **free** for variables. s_0 is the sequential code for SM_0 to initiase the variables followed by the parallel composition. $SM_{(p,N)}$ executes s_p in parallel for $1 \leq p \leq N$.

Figure 2 shows the overview of executing a program. The code starts on the initial stack machine SM_0. After reaching the parallel composition starting with **par** 0, the N stack machines run in parallel. When all the executions of parallel processes terminate, it passes the control to the initial stack machine, freeing the variable at the end. The environment σ, the label stack ρ, and the value stack ξ are shared by all stack machines.

Let ρ and ξ be the label stack and the value stack. $\langle \mathsf{Exec}_s(PC, PC', w), \rho, \xi\rangle$ is a configuration of code s with the initial SM. For the forward execution, the initial configuration is $\langle \mathsf{Exec}_s(1, 0, \varepsilon), \varepsilon, \varepsilon\rangle$ and the final configuration is $\langle \mathsf{Exec}_s(M+1, M, \varepsilon), \rho_F, \xi_F\rangle$. The corresponding backward execution starts with $\langle \mathsf{Exec}_{\texttt{i}(s)}(1, 0, \varepsilon), \rho_F, \xi_F\rangle$ and ends with $\langle \mathsf{Exec}_{\texttt{i}(s)}(M+1, M, \varepsilon), \varepsilon, \varepsilon\rangle$. While the parallel blocks are executed, the configuration is in the form:

$$\langle \mathsf{Exec}^1_s(PC_1, PC'_1, w_1) \| \cdots \| \mathsf{Exec}^N_s(PC_N, PC'_N, \varepsilon), \sigma, \rho, \xi\rangle$$

We define the execution of s as the transition relation between configurations shown in Fig. 3. In the rules above, $PC \in s$ denotes that PC points a code in s. $s(PC)$ is the code pointed by PC and $loc(s_i)$ is the address of s_i in the stack machine code.

- Init defines the behaviour before and after the parallel composition. ℓ is the number of variables. SM_0 constructs the symbol table by s_I and executes the initial sequential code s_0.
- Fork dispatches the development of the parallel blocks once it reaches the first **par** 0. The program counter of stack machine SM_i is set to $loc(s_i)$.
- Par defines the interleaving behaviour of the parallel composition.
- Merge goes back to the initial stack machine and sets the PC to $loc(s_F)$ once all SM_p reaches **par** 0. The execution continues with Init.

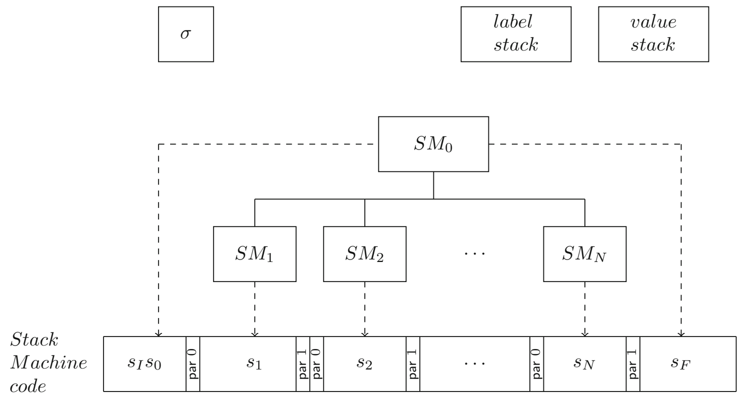

Fig. 2. Execution by stack machines

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

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

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

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

Fig. 3. Execution for code s

In order to implement the operational semantics, it is necessary to scan the whole stack machine code before executing the code to identify N, $loc(s_i)$ and $loc(s_F)$.

4 Concluding Remarks

We present a reversible runtime environment for simple parallel programs and its experimental implementation by Python. The reversibility mechanism is *state-saving* and the environment performs the *back-track* reversibility. The runtime environment is a set of reversible stack machines. The stack machines that execute the parallel blocks share the stacks for value-updates and jumps. Since we focus on the reversibility of states, we do not precisely reverse the forward computation. We replace the commands for computing values with `nop` in the backward code. This eases the concurrency control in the backward execution since it has no effect for states. We regard this is enough for behavioural analysis such as debugging. The approach of forward and backward executions is fundamentally similar to that of [2]. Our approach is finer-grained than [2]. This eases the implementation with the existing runtime since the runtime is often less controlled. As the result, in our approach a backward execution does not precisely undo the forward execution at the level of stack machine code. By sharing the variable environment, the label stack, and the value stack, we manage the consistency of variable updates among the stack machines running in parallel.

As related work, our stack machine is close to the basic architecture of [1] in jumping mechanism although only a sequential execution is considered. The label stack maintains the control of jumps across the parallel composition. [5] presents the reversible semantics in the functional programming style at the abstract machine level with communications and concurrency. [5] gives the operational semantics for backward execution, while our approach translates the abstract machine instructions for backwards within the single operational semantics. Our language has no built-in communication mechanism.

For future work, we need to prove the correctness of our translation by strictly formalising the behaviour of the concurrent execution of a program. The programming language in Sect. 2 limits the class of programs although the stack machine operations have more capability. Adding the nested structure of blocks and procedure is possible by extending the reference mechanism for variables. Adding recursion with the parallel composition make the number of parallel processes dynamic. We need to extend the numbering scheme for identifying the sequential processes executed in parallel and how to choose the next available process in the backward execution.

Acknowledgement. The authors thank Dr. I. Ulidowski and Dr. J. Hoey for the valuable suggestions and discussion. This work was supported by JSPS KAKENHI Grant Numbers JP17H01722 and JP17K19969.

A Runtime Environment by Python

The concrete examples and our implementation by Python are shown at https://github.com/syuen1/RevRunTimeEnv.

A source program is compiled to the forward stack machine code.

```
% java Parser "source program"
```

The forward stack machine code is stored in `code.txt`. To run the code forward,

```
% Python vm.py code.txt f v
```

Then, we get `stack.txt`,`rstack.txt`, and `lstack.txt` as the stack for variable values and the stack for labels[2]

[2] The last `v` shows the verbose mode to show all the steps. No intermediate result is shown when `q` is specified.

To invert the forward code,

```
% Python inv.py code.txt invcode.txt
```

And run the backward code,

```
% Python vm.py invcode.txt b v
```

In the backward, `vm.py` reads the stack files. The result for the airline ticket example is shown in the appendix.

A.1 Controlling Parallel Blocks

The runtime can be executed step-by-step choosing which parallel block is executed in the next step in both directions. The execution of the parallel blocks is controlled by the process that is running the initial stack machine. By entering the process number, the program executes one step in the forward and backward execution showing the stacks.

References

1. Axelsen, H.B., Glück, R., Yokoyama, T.: Reversible machine code and its abstract processor architecture. In: Diekert, V., Volkov, M.V., Voronkov, A. (eds.) CSR 2007. LNCS, vol. 4649, pp. 56–69. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74510-5_9
2. Hoey, J., Ulidowski, I.: Reversible imperative parallel programs and debugging. In: Thomsen, M.K., Soeken, M. (eds.) RC 2019. LNCS, vol. 11497, pp. 108–127. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21500-2_7
3. Hoey, J., Ulidowski, I., Yuen, S.: Reversing parallel programs with blocks and procedures. In: EXPRESS/SOS 2018, Beijing, China, 3 September 2018, EPTCS, vol. 276, pp. 69–86 (2018)
4. Levin, R.Y., Sherman, A.T.: A note on Bennett's time-space tradeoff for reversible computation. SIAM J. Comput. **19**(4), 673–677 (1990)
5. Lienhardt, M., Lanese, I., Mezzina, C.A., Stefani, J.-B.: A reversible abstract machine and its space overhead. In: Giese, H., Rosu, G. (eds.) FMOODS/FORTE -2012. LNCS, vol. 7273, pp. 1–17. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30793-5_1
6. Thomsen, M.K., Axelsen, H.B.: Interpretation and programming of the reversible functional language RFUN. In: IFL 2015, Koblenz, Germany, 14–16 September 2015, pp. 8:1–8:13. ACM (2015)
7. Yokoyama, T., Glück, R.: A reversible programming language and its invertible self-interpreter. In: PEPM 2007, pp. 144–153. ACM (2007)