A Reversible Abstract Machine and Its Space Overhead

Michael Lienhardt, Ivan Lanese, Claudio Antares Mezzina, Jean-Bernard Stefani

A Reversible Abstract Machine and Its Space Overhead*

Michael Lienhardt[1], Ivan Lanese[1], Claudio Antares Mezzina[2], and Jean-Bernard Stefani[3]

[1] Focus Team, University of Bologna/INRIA, Italy
{lienhard,lanese}@cs.unibo.it
[2] SOA Unit, FBK, Trento, Italy
mezzina@fbk.eu
[3] INRIA Grenoble-Rhône-Alpes, France
jean-bernard.stefani@inria.fr

Abstract. We study in this paper the cost of making a concurrent programming language reversible. More specifically, we take an abstract machine for a fragment of the Oz programming language and make it reversible. We show that the overhead of the reversible machine with respect to the original one in terms of space is at most linear in the number of execution steps. We also show that this bound is tight since some programs cannot be made reversible without storing a commensurate amount of information.

1 Introduction

There has recently been renewed interest in the notion of reversible computation [4] and new studies initiated on reversible programming languages [7, 12, 18]. This is sparked by the potential usefulness of reversible computation in a number of areas, including low-power computation [10], quantum computing [1] and building recoverable systems, typically using some form of undo [5].

In a previous paper [12], we have studied how to make the higher-order π-calculus (HOπ) reversible, i.e. how to equip this small paradigmatic concurrent higher-order language with a reduction semantics that comprises both forward steps (the usual reductions of HOπ) and backward ones, which precisely undo previous forward reductions. Specifically, if M, N are two reversible HOπ program configurations and M can reduce to N in one forward step, noted $M \twoheadrightarrow N$, then N can reduce to M in one backward step, noted $N \rightsquigarrow M$. The paper also presented a faithful encoding of reversible HOπ into HOπ, which can be seen as a first step towards understanding how to implement such a reversible language. This encoding, however, was quite wasteful in terms of resources, leading in particular to a potential space overhead, compared to standard (forward only) HOπ executions, which can be exponential in the number computation steps. [4]

* This work has been partially supported by the French National Research Agency (ANR), project REVER n. ANR 11 INSE 007.

[4] To explain this without going into details of our reversible HOπ, let us just mention that a forward computation step in this calculus requires retaining in a so-called

In this paper we initiate a study of the *implementation* of a *reversible higher-order concurrent language* and of its attendant costs. We start with a subset of the Oz kernel programming language [16]. This fragment of Oz, called μOz, is very close to HOπ, and its formal operational semantics is specified as a simple and rather classical *stack-based abstract machine*, itself directly inspired by the abstract machine of the Oz kernel programming language, which provides an interesting and well-known point of reference. We then define a reversible variant of μOz by means of an extended abstract machine, and we prove (i) that this new machine implements exactly the forward reductions of the initial one, and (ii) that it indeed implements reversibility for μOz, as characterized above for HOπ. Finally we study the *space overhead* that the reversible abstract machine adds to a forward execution compared to the same execution carried out by the μOz abstract machine. We prove that this overhead is *at worst linear* in the number of execution steps, and that this linear *upper bound is tight*: we show that some reversible μOz programs cannot execute with less than an amount of additional information – required to allow reversing their execution – that is linear in the number of execution steps. It would have been difficult to carry on a similar analysis directly in HOπ, since there is no largely accepted abstract machine for HOπ to be used as a reference. In fact, HOπ operational semantics is not precise enough on the use of memory space to act as a reference in such a context. To the best of our knowledge this is the first study of its kind. There is work investigating the time and space complexity of simulating irreversible computations by reversible ones e.g. [6, 17], as well as recent work investigating the compilation of a reversible sequential language [2], but we do not know of work focusing as we do on the implementation or simulation of a reversible concurrent language and the analysis of its space costs.

Outline. The paper is organized as follows. Section 2 presents the syntax and abstract machine for μOz, the fragment of Oz we consider; Section 3 presents the reversible extension of the μOz abstract machine; Section 4 describes its reversibility properties; Section 5 studies the overhead reversibility adds compared to the μOz abstract machine; Section 6 discusses related work; and Section 7 concludes the paper.

---

memory the message $a\langle P\rangle$ and the receiver process $a(X) \triangleright Q$ that participated in it (in HOπ and its reversible variant the only – forward – computation steps are message receipts). Thus the space overhead of a computation step in reversible HOπ compared to standard HOπ is at least $\|P\|$, the size of the payload of message $a\langle P\rangle$. Now consider the following recursive programs: $P = c(X) \triangleright P \mid a\langle X \mid X\rangle$ and $Q = a(X) \triangleright Q \mid c\langle X \mid X\rangle$. We have $a\langle R\rangle \mid P \mid Q \to P \mid Q \mid c\langle R \mid R\rangle$ so the space overhead of this first step starting from $a\langle R\rangle \mid P \mid Q$ is at least $\|R\|$. On the second step we have $P \mid Q \mid c\langle R \mid R\rangle \to P \mid Q \mid a\langle R \mid R \mid R \mid R\rangle$, so the space overhead of this second step is at least $2\|R\|$. By induction, one can see that the space overhead associated with making the program $a\langle R\rangle \mid P \mid Q$ reversible is at least $2^{n-1}\|R\|$, where n is the number of computation steps taken from the initial state $a\langle R\rangle \mid P \mid Q$.

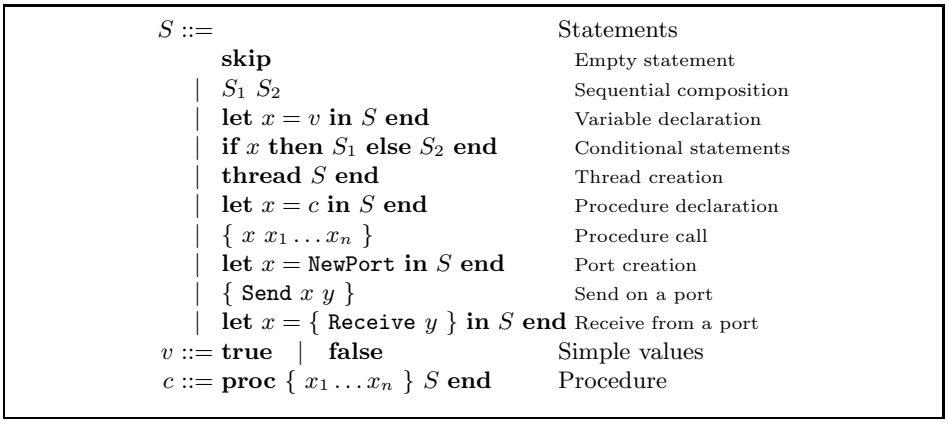

Fig. 1: μOz Syntax

2 The μOz Language

In this section we present the syntax and semantics of μOz, a strict but non-trivial subset of Oz, to be extended with reversibility mechanisms in the next sections. We define the operational semantics of μOz by specifying an abstract machine for executing μOz programs which is directly inspired by the stack-based abstract machine in Chapter 13 of [16]. We refer to [16] for a description of the whole Oz language. We chose Oz because it came with a simple well documented abstract machine, and this subset of Oz because it was very close to HOπ. We do not know of a similarly simple stack-based abstract machine for HOπ in the literature. We could of course have come up with our own abstract machine for HOπ, but starting from on a non reversible abstract machine not devised by us is more appealing.

The syntax of μOz is in Figure 1. μOz is a higher-order language with thread-based concurrency and asynchronous communication via ports. For the sake of simplicity, the only values we consider in μOz are booleans, ports and closures. We eschew Oz logical variables in favor of simple immutable variables, i.e. read-only variables that are initialized at the time of their declaration. Dealing with the full Oz kernel programming language would not have posed more conceptual difficulties but would have obscured the technical details. The statements in μOz are fairly classical. Let us just point out that communication on a port, by means of send and receive operations, is asynchronous and by way of a FIFO queue. Variable declaration, procedure declaration, port creation and receiving are binders. Specifically, x is bound in **let** $x = v$ **in** S **end**, **let** $x = c$ **in** S **end**, **let** $x =$ `NewPort` **in** S **end**, and **let** $x = \{$ `Receive` y $\}$ **in** S **end**.

Runtime Syntax. To describe the abstract machine defining μOz semantics we need a runtime syntax defining tasks and threads, used for statement execution, port queues, for communication, and the store. The runtime syntax of μOz is

$T ::=$		Thread
	$\langle\rangle$	Null thread
$\mid$	$\langle S\ T\rangle$	Thread with statement
$U, V ::=$		Task
	0	No task
$\mid$	T	Thread
$\mid$	$U \parallel V$	Parallel composition
$w ::=$	$v \quad \mid \quad \xi$	Extended value
$\sigma ::=$		Store
	0	Empty store
$\mid$	$x = w$	Binding
$\mid$	$\xi : c$	Closure
$\mid$	$\xi : Q$	Port
$\mid$	$\sigma \parallel \sigma'$	Conjunction
$Q ::=$		Port queue
	$\bot$	Empty queue
$\mid$	$x \quad \mid \quad Q; Q'$	Sequence of variables

Fig. 2: μOz Runtime Syntax

$$S =_\alpha S' \Rightarrow S \equiv S' \qquad \langle (S_1\ S_2)\ T\rangle \equiv \langle S_1\ \langle S_2\ T\rangle\rangle$$
$$Q_1; (Q_2; Q_3) \equiv (Q_1; Q_2)\ Q_3 \qquad Q; \bot \equiv \bot; Q$$
$$E_1 \parallel (E_2 \parallel E_3) \equiv (E_1 \parallel E_2) \parallel E_3 \qquad E \parallel 0 \equiv E \qquad E_1 \parallel E_2 \equiv E_2 \parallel E_1$$

Fig. 3: μOz Structural Congruence

presented in Figure 2. Tasks are a parallel composition of threads. Threads are stacks of statements. The store (or heap) is a conjunction of bindings, closures, and ports (essentially implemented as named FIFO queues).

Structural Congruence. We consider tasks, threads, statements and the store up to a structural congruence relation. Structural congruence, written $\equiv$, is defined as the smallest congruence that validates the rules presented in Figure 3, where $=_\alpha$ stands for equality up to α-conversion. We write E for an *execution term*, i.e. either a task U or a store σ. The neutral element of concatenation for queues is $\bot$.

Reduction Rules. The μOz semantics is defined as a reduction relation, noted $\rightarrow$, between configurations of the form (U, σ). To follow Oz notation, the relation $\rightarrow$ is defined by a set of rules of the form below, specifying that (U, σ) reduces to (U', σ') if condition G is satisfied:

$$\frac{U \parallel U'}{\sigma \parallel \sigma'} \text{ if } G$$

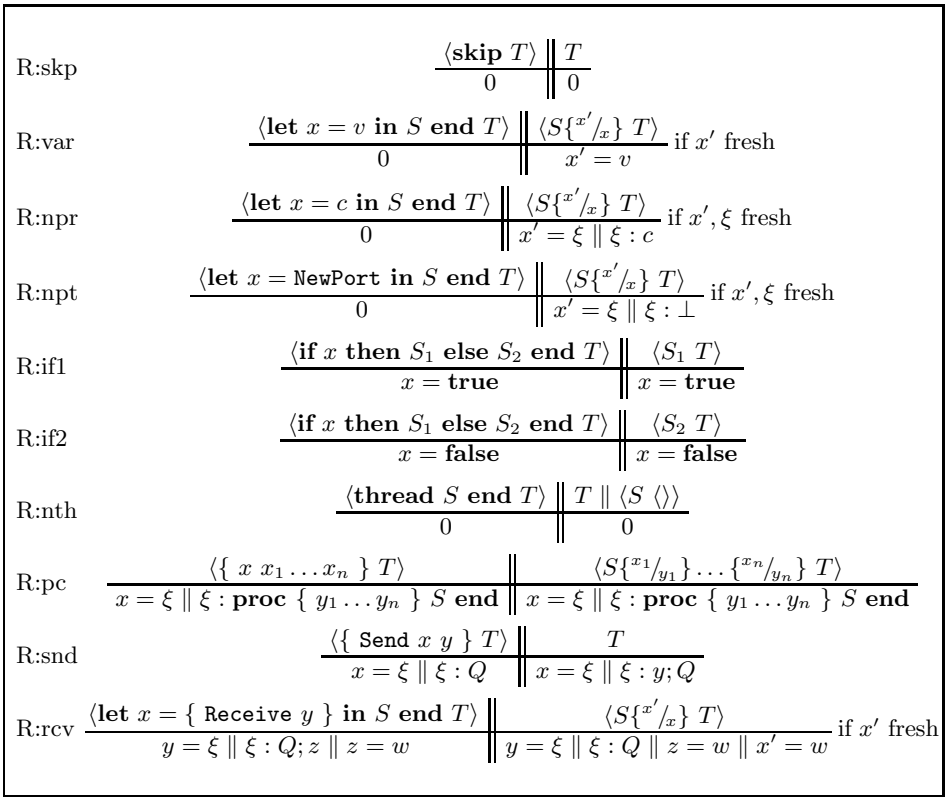

Fig. 4: μOz Abstract Machine Semantics

The reduction relation $\rightarrow$ is closed under evaluation contexts, and is defined modulo structural congruence. We capture this with the notion of evaluation-closed relation.

Definition 1. *A relation R between configurations is said to be* evaluation-closed *if it satisfies the following inference rules:*

$$\left(\frac{U \;\|\; U'}{\sigma \;\|\; \sigma'} \Rightarrow \frac{U \parallel U_k \;\|\; U' \parallel U_k}{\sigma \parallel \sigma_k \;\|\; \sigma' \parallel \sigma_k}\right)$$

$$\begin{pmatrix} U_1 \equiv U_1' \wedge U_2 \equiv U_2' \\ \sigma_1 \equiv \sigma_1' \wedge \sigma_2 \equiv \sigma_2' \end{pmatrix} \wedge \frac{U_1 \;\|\; U_2}{\sigma_1 \;\|\; \sigma_2} \Rightarrow \frac{U_1' \;\|\; U_2'}{\sigma_1' \;\|\; \sigma_2'}$$

Intuitively, the first rule corresponds to closure under evaluation contexts (parallel composition of threads), while the second rule is closure under structural congruence.

We define the *reduction relation* $\rightarrow$ to be the smallest evaluation-closed relation on configurations that validates the rules in Figure 4.

Rule R:var creates a new binding $x' = v$ in the store. Also, x' is substituted for x in the scope S to avoid variable capture. Rule R:npr is similar, but deals with closures. The closure c is assigned a fresh name ξ, and the name is bound to the fresh variable x'. Similarly, rule R:npt creates a new port, and initializes the queue to $\bot$. Rule R:nth creates a new thread. Rule R:pc executes a procedure call, substituting the actual parameter variables for the formal ones in the code to be executed. Rule R:snd performs a send, by enqueuing a variable in the corresponding queue. Rule R:rcv performs a receive, dequeuing the corresponding element z and fetching its value w. The value w is assigned to the fresh variable x' that substitutes the formal variable x.

3 A Reversible Abstract Machine for μOz

In this section we extend the μOz abstract machine presented in Section 2 to make it reversible. Since our language is concurrent, we aim at defining a causally consistent form of reversibility [7], where the execution can go back to any state that could have been reached in the forward execution by changing the order of execution of concurrent steps. We start by extending μOz runtime syntax.

μOz Reversible Runtime Syntax. The runtime syntax used in the definition of μOz reversible semantics is in Figure 5.

With respect to μOz runtime syntax (Figure 2) we now consider extended stacks C, which may also contain the scope delimiter **esc** as a statement. This is needed to reverse the let and if statements, as well as procedure calls. Consider for instance the case of procedure calls: **esc** is needed to find out where the procedure code ends (the body of the procedure should be removed to reverse the call) and the caller code begins (the caller code should be preserved).

Threads now have a name t (which is unique) and a history H, and execute an extended statement stack C. The history stores information about executed statements. Note that sent variables are actually stored in the queue, not in the history. Also, for an if statement just the discarded branch has to be stored, since the other one is available in the thread code. History is needed also inside ports, to remember the order of communications.

Structural congruence. Structural congruence has to be extended from threads T to extended stacks C, and from store σ to store θ. The rules are however the same as in Figure 3.

Reduction Rules. We define the two reduction relations $\twoheadrightarrow$ (forward execution) and $\rightsquigarrow$ (backward execution) to be the smallest evaluation-closed relations that validate the rules presented in Figure 6 and Figure 7, respectively. In the rules and in the following, we will abbreviate a sequence (possibly empty) of variables

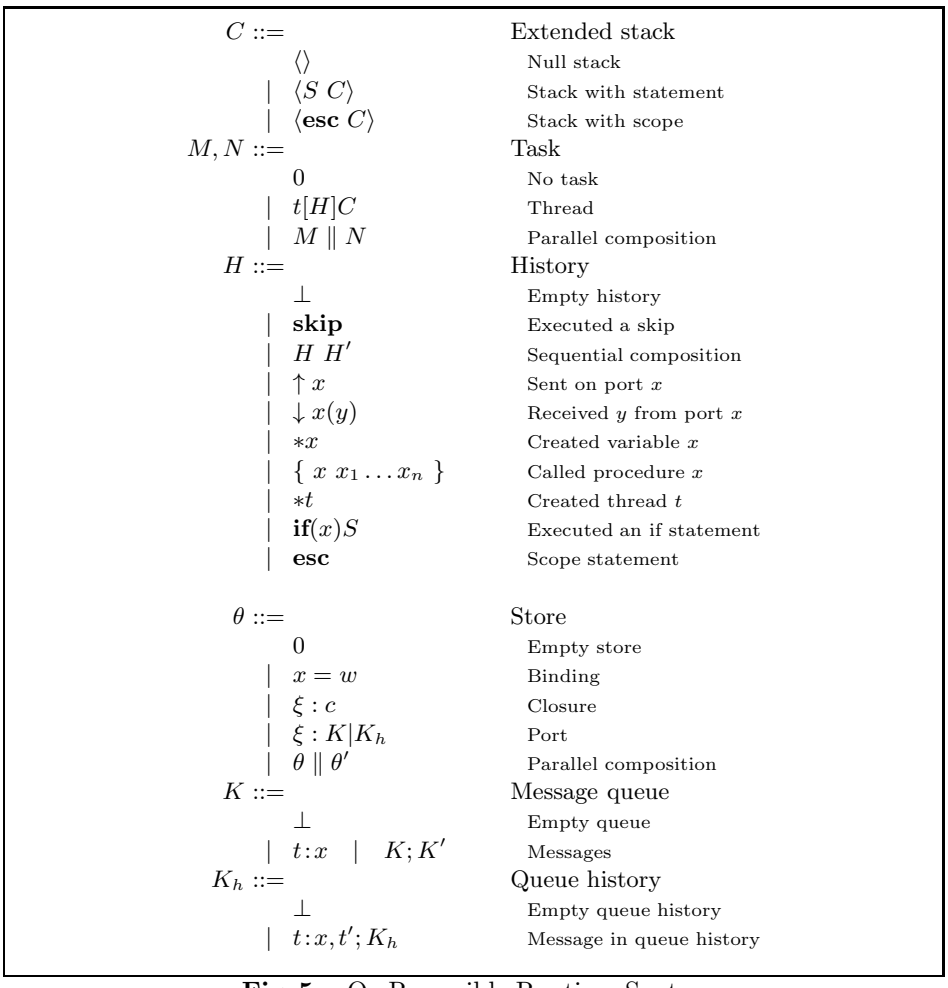

Fig. 5: μOz Reversible Runtime Syntax

$x_1 \dots x_n$ as $(x_i)_1^n$ (with $n \geq 0$). Similarly, we will abbreviate a sequence of substitutions $\{^{x_1}/_{y_1}\} \dots \{^{x_n}/_{y_n}\}$ as$(\{^{x_i}/_{y_i}\})_1^n$. We define the reduction relation $\rightarrow_{vm}$ as $\rightarrow_{vm} = \twoheadrightarrow \cup \rightsquigarrow$.

Rule R:fw:var stores $*x'$ in the history, meaning that x' has been used as fresh variable, and uses the scope delimiter **esc** to recall the scope of the binding. Rules R:fw:npr and R:fw:npt are similar. In rule R:fw:npt the created queue comes with an empty history $\bot$. Rules R:fw:if1 and R:fw:if2 store the branch discarded by the choice in the history. In rule R:fw:nth the new thread is given a fresh name t' and an empty history $\bot$. The fresh name t' is also stored in the history of the creating thread. In rule R:fw:pc the name and actual parameters of the invoked procedure are stored in the history. In rule R:fw:snd we store in the queue the name of the thread sending the value, to avoid that a different thread takes

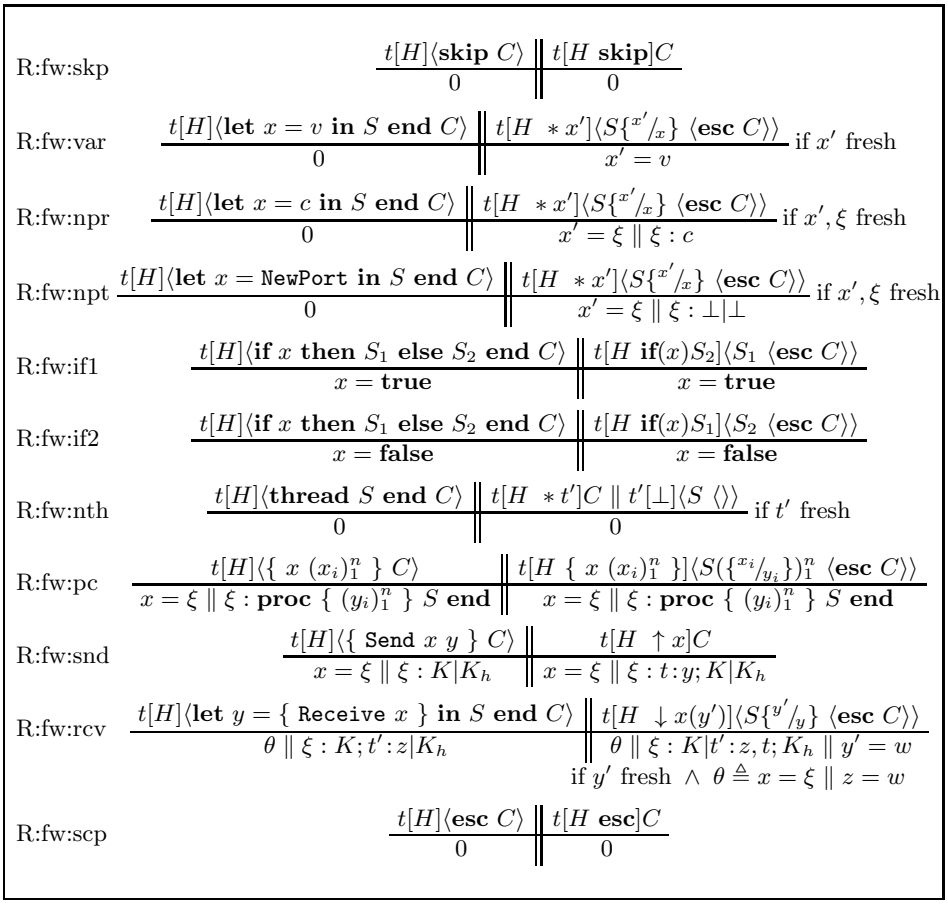

Fig. 6: μOz Abstract Machine Forward Semantics

the value when rolling back. In rule R:fw:rcv the read value is also kept in the queue history, with information on who read it. Rule R:fw:scp is new, allowing to record the scope delimiter **esc** in the history.

The backward rules in Figure 7 are in one to one correspondence with the forward ones, and use the stored information to get back to the original state. Notably, rules R:bk:var, R:bk:npr, R:bk:npt and R:bk:rcv go back to a term which is not the starting one, but which is equivalent up to α-conversion. Also, rules R:bk:var, R:bk:npr, R:bk:npt, R:bk:if1, R:bk:if2, R:bk:pc and R:bk:rcv exploit the scope delimiter **esc** to identify the scope of the statement to be reversed. Note that the occurrence of **esc** in the rule is always matched by the nearest occurrence in the term. In rule R:bk:nth the $\bot$ in the history of the second thread ensures that the thread is rolled-back before its creation is rolled-back. The same happens for ports in R:bk:npt.

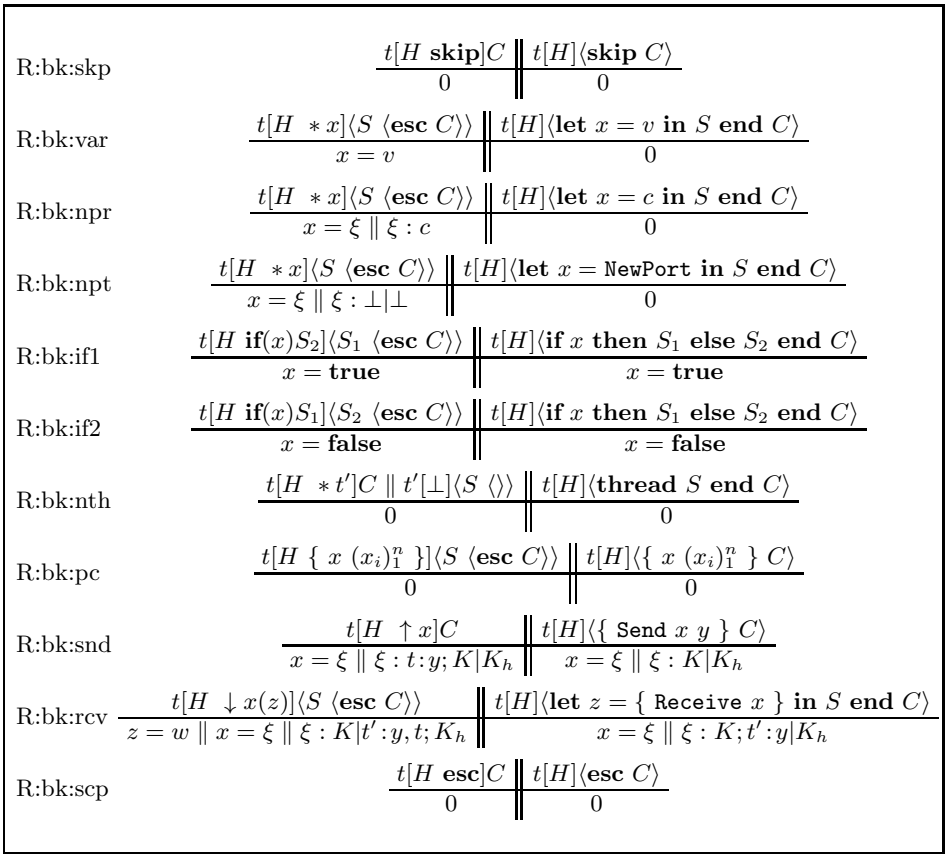

Fig. 7: μOz Abstract Machine Backward Semantics

4 Properties of the Reversible Abstract Machine

In this section we prove that the forward and backward semantics actually define a causally consistent reversible version of μOz, following the lines of [7, 12].

First of all we show that the reversible language is a conservative extension of μOz, i.e. forward computations of our reversible machine are indeed a decorated version of the μOz reductions. To this end we define a function, called **erase**, which takes a reversible μOz configuration and erases all the information needed only for reversibility purposes. The function is defined in Figure 8.

Lemma 1. *Let (M, θ) and (N, θ') be two reversible configurations such that $(M, \theta) \twoheadrightarrow (N, \theta')$. We have either*

$$\mathbf{erase}((M, \theta)) \rightarrow \mathbf{erase}((N, \theta')) \text{ or } \mathbf{erase}((M, \theta)) = \mathbf{erase}((N, \theta'))$$

Proof. By case analysis on the reduction rule applied. The equality case holds for rule R:fw:scp only. □

$$\mathbf{erase}(\langle\rangle) \triangleq \langle\rangle \qquad \mathbf{erase}(\langle S\ C\rangle) \triangleq \langle S\ \mathbf{erase}(C)\rangle \qquad \mathbf{erase}(\langle \mathbf{esc}\ C\rangle) \triangleq \mathbf{erase}(C)$$

$$\mathbf{erase}(0) \triangleq 0 \qquad \mathbf{erase}(t[H]C) \triangleq \mathbf{erase}(C)$$
$$\mathbf{erase}(M \parallel N) \triangleq \mathbf{erase}(M) \parallel \mathbf{erase}(N)$$

$$\mathbf{erase}(x = w) \triangleq x = w \qquad \mathbf{erase}(\xi : c) \triangleq \xi : c \qquad \mathbf{erase}(\xi : K|K_h) \triangleq \xi : \mathbf{erase}(K)$$
$$\mathbf{erase}(\theta \parallel \theta') \triangleq \mathbf{erase}(\theta) \parallel \mathbf{erase}(\theta')$$

$$\mathbf{erase}(\bot) \triangleq \bot \qquad \mathbf{erase}(t : x) \triangleq x \qquad \mathbf{erase}(K; K') \triangleq \mathbf{erase}(K); \mathbf{erase}(K')$$

Fig. 8: Function Erasing Reversibility Information

Lemma 2. *Let (T, σ) and (M, θ) be a simple and a reversible configuration such that* $\mathbf{erase}((M, \theta)) \to (T, \sigma)$*. Then, there exists a reversible configuration (N, θ') with* $\mathbf{erase}((N, \theta')) = (T, \sigma)$ *and such that $(M, \theta) \twoheadrightarrow^+ (N, \theta')$ where $\twoheadrightarrow^+$ denotes one or more applications of $\twoheadrightarrow$.*

Proof. By case analysis on the rule applied. Each rule is matched by the corresponding reversible rule. Auxiliary steps using rule R:fw:scp may be needed. □

The Loop Lemma below ensures that every step can be reversed.

Lemma 3 (Loop Lemma). *For all configurations (M, θ) and (M', θ') the following double implication holds:*

$$(M, \theta) \twoheadrightarrow (M', \theta') \quad \Leftrightarrow \quad (M', \theta') \rightsquigarrow (M, \theta)$$

Proof. By case analysis on the used rule. Each rule R:fw:* is reversed by the corresponding rule R:bk:* and viceversa. Relevant issues have been highlighted in the description of the backward rules. □

We call *transition* a pair of the form $(M, \theta) \to_{vm} (N, \theta')$, where (M, θ) and (N, θ') are two configurations. We indicate (M, θ) as the *source* of the transition and (N, θ') as the target of the transition. Two transitions are said to be *coinitial* if they have the same source, *cofinal* if they have the same target, and *composable* if the target of the first is the source of the second. A sequence of pairwise composable transitions is called a *trace*. We let r and its decorated variants range over transitions, τ and its decorated variants range over traces. If r is a transition, we set $r_\bullet$ as its inverse. A transition $(M, \theta) \twoheadrightarrow (N, \theta')$ is said to be forward, while a transition $(M, \theta) \rightsquigarrow (N, \theta')$ is said to be backward. Notions of target, source and composability extend naturally to traces. We denote with $\epsilon_{(M,\theta)}$ the empty trace with source (M, θ), and with $\tau_1; \tau_2$ the composition of two composable traces τ_1 and τ_2.

In order to show that reversibility is causally consistent, we now define the notion of concurrency in our language.

Definition 2 (Concurrent transitions).
Two coinitial transitions $r_1 = (M, \theta) \rightarrow_{vm} (M_1, \theta_1)$ *and* $r_2 = (M, \theta) \rightarrow_{vm} (M_2, \theta_2)$ *are said to be* concurrent *unless* r_1 *and* r_2*:*

- *are executed by the same thread;*
- *are sends or a send and an undo of a send to the same queue;*
- *are receives and/or undo of receives from the same queue;*
- *are an action of a thread and the undo of the thread creation;*
- *are a use of a variable and the undo of its creation;*
- *are a receive on a queue with one element and the undo of its send.*

The definition of concurrent transitions enables the following result.

Lemma 4 (Square lemma). *Given two coinitial concurrent transitions* $r_1 = (M, \theta) \rightarrow_{vm} (M_1, \theta_1)$ *and* $r_2 = (M, \theta) \rightarrow_{vm} (M_2, \theta_2)$*, there exist two cofinal transitions* $r_2/r_1 = (M_1, \theta_1) \rightarrow_{vm} (N, \theta_3)$ *and* $r_1/r_2 = (M_2, \theta_2) \rightarrow_{vm} (N, \theta_3)$*.*

Proof. By case analysis on the form of transitions r_1 and r_2. □

We finally define the notion of causal equivalence between traces, noted $\asymp$, as the least equivalence relation between traces closed under composition that obeys the following rules (where r is forward):

$$r_1; r_2/r_1 \asymp r_2; r_1/r_2 \qquad r; r_\bullet \asymp \epsilon_{\mathtt{source}(r)} \qquad r_\bullet; r \asymp \epsilon_{\mathtt{target}(r)}$$

Following the same proof schema as that used in [7, 12], we can now prove:

Theorem 1 (Causal consistency). *Let* τ_1 *and* τ_2 *be coinitial traces, then* $\tau_1 \asymp \tau_2$ *if and only if* τ_1 *and* τ_2 *are cofinal.*

Informally, this means that, if we consider different computations from the same starting process, we never distinguish processes obtained by causal equivalent computations, while we always distinguish processes obtained by computations which are not causally equivalent, since they should have different backward behaviors. Together, the loop lemma and the causal consistency theorem express that our reversible machine correctly implements step wise reversibility (loop lemma), and that it does so with the maximum amount of flexibility by not distinguishing, as far as reversing a computation is concerned, between causally equivalent traces (causal consistency theorem).

5 Memory Overhead

We turn now to the analysis of implementation costs. We prove two results in this section. First, we show that the space overhead imposed by our reversible abstract machine compared to the standard μOz machine is linear in the number of steps of a given computation. Second, we show that this cannot be improved, as far as the order of magnitude is concerned, since the amount of information required to reverse certain μOz programs is indeed linear in the number of steps in their executions.

$$\mathbf{size}(\mathbf{skip}) \triangleq 1 \qquad \mathbf{size}(S_1\ S_2) \triangleq \mathbf{size}(S_1) + \mathbf{size}(S_2)$$
$$\mathbf{size}(\mathbf{let}\ x = v\ \mathbf{in}\ S\ \mathbf{end}) \triangleq 3 + \mathbf{size}(S)$$
$$\mathbf{size}(\mathbf{if}\ x\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end}) \triangleq 2 + \mathbf{size}(S_1) + \mathbf{size}(S_2)$$
$$\mathbf{size}(\mathbf{thread}\ S\ \mathbf{end}) \triangleq 1 + \mathbf{size}(S)$$
$$\mathbf{size}(\mathbf{let}\ x = c\ \mathbf{in}\ S\ \mathbf{end}) \triangleq 2 + \mathbf{size}(c) + \mathbf{size}(S)$$
$$\mathbf{size}(\mathbf{proc}\ \{\ x_1 \dots x_n\ \}\ S\ \mathbf{end}) \triangleq n + 1 + \mathbf{size}(S) \qquad \mathbf{size}(\{\ x\ x_1 \dots x_n\ \}) \triangleq n + 2$$
$$\mathbf{size}(\mathbf{let}\ x = \texttt{NewPort}\ \mathbf{in}\ S\ \mathbf{end}) \triangleq 3 + \mathbf{size}(S) \qquad \mathbf{size}(\{\ \texttt{Send}\ x\ y\ \}) \triangleq 3$$
$$\mathbf{size}(\mathbf{let}\ x = \{\ \texttt{Receive}\ y\ \}\ \mathbf{in}\ S\ \mathbf{end}) \triangleq 3 + \mathbf{size}(S)$$

$$\mathbf{size}((U, \sigma)) \triangleq \mathbf{size}(U) + \mathbf{size}(\sigma) \qquad \mathbf{size}(U \parallel V) \triangleq \mathbf{size}(U) + \mathbf{size}(V)$$
$$\mathbf{size}(\sigma \parallel \sigma') \triangleq \mathbf{size}(\sigma) + \mathbf{size}(\sigma') \qquad \mathbf{size}(0) \triangleq 0 \qquad \mathbf{size}(\langle\rangle) \triangleq 1$$
$$\mathbf{size}(\langle S\ T\rangle) \triangleq \mathbf{size}(S) + \mathbf{size}(T) \qquad \mathbf{size}(x = w) \triangleq 3 \qquad \mathbf{size}(\xi : c) \triangleq 2 + \mathbf{size}(c)$$
$$\mathbf{size}(\xi : Q) \triangleq 2 + \mathbf{size}(Q) \qquad \mathbf{size}(\bot) \triangleq 0 \qquad \mathbf{size}(x) \triangleq 1$$
$$\mathbf{size}(Q; Q') \triangleq \mathbf{size}(Q) + \mathbf{size}(Q')$$

Fig. 9: Size of a Simple Configuration

5.1 Overhead of the reversible abstract machine

In order to measure the space overhead of our reversible abstract machine we define a function **size** computing the size of a simple configuration (Figure 9) and a function **rsize** computing the size of a reversible configuration (Figure 10). Essentially the size of a term is computed as follows: we count 1 for the operator, 1 for each name it can have as argument, plus the size of subterms, if any.

Definition 3. *The* overhead *of a reversible configuration* (M, θ) *is defined as:*

$$\mathbf{overhead}(M, \theta) \triangleq \mathbf{rsize}((M, \theta)) - \mathbf{size}(\mathbf{erase}((M, \theta)))$$

To show that the space overhead of our abstract machine is linear in the number of reduction steps, we prove first that the maximal amount of information stored in a single step (computed by function **stsize** defined in Figure 11) is bounded by a constant during the execution of any fixed program:

Lemma 5. *Let* (U, σ) *be a simple configuration. Then for all* (U', σ') *such that* $(U, \sigma) \to (U', \sigma')$ *we have* $\mathbf{stsize}((U', \sigma')) \leq \mathbf{stsize}((U, \sigma))$.

Proof. By case analysis on the reduction rule. □

$$\mathbf{rsize}(\bot) \triangleq 0 \qquad \mathbf{rsize}(\mathbf{skip}) \triangleq 1 \qquad \mathbf{rsize}(H_1\ H_2) \triangleq \mathbf{rsize}(H_1) + \mathbf{rsize}(H_2)$$
$$\mathbf{rsize}(\uparrow x) \triangleq 2 \qquad \mathbf{rsize}(\downarrow x(y)) \triangleq 3 \qquad \mathbf{rsize}(*x) \triangleq 2 \qquad \mathbf{rsize}(*t) \triangleq 2$$
$$\mathbf{rsize}(\{\ x\ x_1 \dots x_n\ \}) \triangleq n+2 \quad \mathbf{rsize}(\mathbf{if}(x)S) \triangleq 2 + \mathbf{size}(S) \quad \mathbf{rsize}(\mathbf{esc}) \triangleq 1$$

$$\mathbf{rsize}((M,\theta)) \triangleq \mathbf{rsize}(M) + \mathbf{rsize}(\theta) \qquad \mathbf{rsize}(0) \triangleq 0$$
$$\mathbf{rsize}(M \parallel N) \triangleq \mathbf{rsize}(M) + \mathbf{rsize}(N) \qquad \mathbf{rsize}(\theta \parallel \theta') \triangleq \mathbf{rsize}(\theta) + \mathbf{rsize}(\theta')$$
$$\mathbf{rsize}(t[H]C) \triangleq 1 + \mathbf{rsize}(H) + \mathbf{rsize}(C) \qquad \mathbf{rsize}(\langle\rangle) \triangleq 1$$
$$\mathbf{rsize}(\langle \mathbf{esc}\ C\rangle) \triangleq 1 + \mathbf{rsize}(C) \qquad \mathbf{rsize}(\langle S\ C\rangle) \triangleq \mathbf{size}(S) + \mathbf{rsize}(C)$$
$$\mathbf{rsize}(x = w) \triangleq 3 \qquad \mathbf{rsize}(\xi : c) \triangleq 2 + \mathbf{size}(c)$$
$$\mathbf{rsize}(\xi : K|K_h) \triangleq 2 + \mathbf{rsize}(K) + \mathbf{rsize}(K_h) \qquad \mathbf{rsize}(\bot) \triangleq 0 \qquad \mathbf{rsize}(t{:}x) \triangleq 3$$
$$\mathbf{rsize}(K; K') \triangleq \mathbf{rsize}(K) + \mathbf{rsize}(K') \qquad \mathbf{rsize}(t{:}x, t'; K_h) \triangleq 4 + \mathbf{rsize}(K_h)$$

Fig. 10: Size of a Reversible Configuration

$$\mathbf{stsize}(S_1\ S_2) \triangleq \max(\mathbf{stsize}(S_1), \mathbf{stsize}(S_2)) \qquad \mathbf{stsize}(\mathbf{skip}) \triangleq 1$$
$$\mathbf{stsize}(\mathbf{let}\ x = v\ \mathbf{in}\ S\ \mathbf{end}) \triangleq \max(2, \mathbf{stsize}(S))$$
$$\mathbf{stsize}(\mathbf{if}\ x\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end}) \triangleq 3 + \max(\mathbf{size}(S_1)), \mathbf{size}(S_2))$$
$$\mathbf{stsize}(\mathbf{thread}\ S\ \mathbf{end}) \triangleq \max(3, \mathbf{stsize}(S))$$
$$\mathbf{stsize}(\mathbf{let}\ x = c\ \mathbf{in}\ S\ \mathbf{end}) \triangleq \max(3, \mathbf{stsize}(c), \mathbf{stsize}(S))$$
$$\mathbf{stsize}(\mathbf{proc}\ \{\ x_1 \dots x_n\ \}\ S\ \mathbf{end}) \triangleq \mathbf{stsize}(S) \qquad \mathbf{stsize}(\{\ x\ x_1 \dots x_n\ \}) \triangleq n+3$$
$$\mathbf{stsize}(\mathbf{let}\ x = \texttt{NewPort}\ \mathbf{in}\ S\ \mathbf{end}) \triangleq \max(3, \mathbf{stsize}(S)) \quad \mathbf{stsize}(\{\ \texttt{Send}\ x\ y\ \}) \triangleq 4$$
$$\mathbf{stsize}(\mathbf{let}\ x = \{\ \texttt{Receive}\ y\ \}\ \mathbf{in}\ S\ \mathbf{end}) \triangleq \max(7, \mathbf{stsize}(S))$$

$$\mathbf{stsize}((U,\sigma)) \triangleq \max(\mathbf{stsize}(U), \mathbf{stsize}(\sigma)) \qquad \mathbf{stsize}(0) \triangleq 0 \qquad \mathbf{stsize}(\langle\rangle) \triangleq 0$$
$$\mathbf{stsize}(U \parallel V) \triangleq \max(\mathbf{stsize}(U), \mathbf{stsize}(V)) \qquad \mathbf{stsize}(x = w) \triangleq 0$$
$$\mathbf{stsize}(\sigma \parallel \sigma') \triangleq \max(\mathbf{stsize}(\sigma), \mathbf{stsize}(\sigma')) \qquad \mathbf{stsize}(\xi : c) \triangleq \mathbf{stsize}(c)$$
$$\mathbf{stsize}(\langle S\ T\rangle) \triangleq \max(\mathbf{stsize}(S), \mathbf{stsize}(T)) \qquad \mathbf{stsize}(\xi : Q) \triangleq 0$$

Fig. 11: Maximal Overhead added by one Forward Step

We can now bound the overhead of a computation:

Lemma 6. *Let (M,θ) and (N,θ') be configurations such that $(M,\theta) \twoheadrightarrow (N,\theta')$. Then we have*

$$\mathbf{overhead}(N,\theta') \leq \mathbf{overhead}(M,\theta) + \mathbf{stsize}(\mathbf{erase}((M,\theta)))$$

Proof. By case analysis on the reduction rule. □

Theorem 2. *Assume* $(t[\bot]\langle S\ \langle\rangle\rangle, 0) \rightarrow_{vm}^{n} (M, \theta)$*, where* $\rightarrow_{vm}^{n}$ *denotes* $n \rightarrow_{vm}$ *steps. Then*

$$\mathbf{overhead}(M, \theta) \leq n \cdot \mathbf{stsize}(S)$$

Proof. By induction on the number of forward transitions, using Lemma 5 and Lemma 6. There is no need to consider backward transitions, since they reduce the overhead. □

5.2 Lower bound on the cost of reversing μOz programs

In this section we show that to ensure causally consistent reversibility of μOz programs we need a space overhead which is at least linear in the number of execution steps. Specifically, we prove the following:

Theorem 3. *The amount of information to be stored to ensure causally consistent reversibility of μOz programs is at least linear in the number of execution steps.*

Proof. Consider the following program:

```
let a = NewPort in
let x = true in
let y = false in
let p1 = proc {} {Send a x} { p1 } end in
let p2 = proc {} {Send a y} { p2 } end in
let p3 = proc {} let z = {Receive a} in { p3 } end end in
   thread { p1 } end thread { p2 } end thread { p3 } end
end end end end end end
```

The program launches three threads $p1$, $p2$ and $p3$. The threads $p1$ and $p2$ send messages over port a, while $p3$ receives them. Let us consider the set of traces which start with the initial program configuration, and where all the sends are performed before all the receives. Let us further restrict our attention to those traces consisting only of sequences of sends where for each $i \in \{1, \ldots, n\}$ the sends $2i - 1$ and $2i$ are from different threads, for some natural number n. Call T_n this set of traces. For a fixed n, in all the traces in T_n exactly n values **true** and n values **false** are sent. Thus the same program state S is reached in all the traces after all the values have been received, according to the standard (non-reversible) abstract machine semantics, up to α-conversion of the names of the receiving variables.

Now, all the traces in T_n above are coinitial, and they are not causally equivalent since different send actions to the same queue are not concurrent. To have a causally consistent rollback, coinitial traces which are not causally equivalent should lead to different states (cf Theorem 1), thus the state S obtained must be complemented with additional information $c(t)$ to distinguish between the different traces $t \in T_n$. Each trace $t \in T_n$ is uniquely identified by the state of

the queue just before the first receive is executed by thread `p3`. We can encode each pair of elements in the queue with one bit, since the only possible values are (**true**, **false**) and (**false**, **true**). Thus, for a sequence of $2i$ sends we need i bits for distinguishing the different possibilities. All the words of i bits can be obtained, including the ones whose description is an incompressible string according to Kolmogorov complexity theory [14], so this number of bits cannot be improved. This is a lower bound on the size of information $c(t)$ that has to be stored to ensure causally consistent reversibility. The number of execution steps of the computations described above is linear in the number of bits, since a bounded number of steps is required to create the threads and perform two sends and two receives. □

The result above shows that space is needed for keeping track of communications. A linear lower bound can also be proved using the number of created threads. In fact, in a program involving many threads, one has to remember how many steps each of them actually performed. If one chooses a program where the number of steps performed by each thread is bounded, and the number of threads is linear in the number of steps, a space bound which is linear in the number of steps can be proved.

5.3 Discussion

A few remarks concerning our reversible abstract machine and our results are in order. First, as should be clear from the abstract machine rules and the accompanying explanations, our reversibility machinery is based on an explicit history mechanism, in contrast, say, to reversible operators used in [19]. This may seem a naive choice, but our lower bound in Section 5.2 and the proof of the result there suggest that, in presence of non-deterministic concurrency as is the case with μOz, we have to resort to some sort of history, at least to be able to revert the effects of communication (message sends and receives) and thread creation.

Second, we avoid the exponential blowup in space overhead that we had in [12]. The main reason for this is that in our histories we only store pointers to values that are already present in the store, created by normal forward computation. We can recreate in μOz the equivalent of the sample HOπ program given in the Introduction, by creating appropriate additional closures corresponding to the values $R \mid R$, $R \mid R \mid R \mid R$, etc. But the reversibility machinery would only add pointers to these closures, created by normal forward computation, in thread histories, thus avoiding the exponential overhead.

Finally, one can note that our result in Section 5.2 is fairly robust, since it really is dependent only on the non-deterministic occurrences of non-concurrent events such as putting messages in a port queue from different threads. Such features are likely to be present in any physically distributed program or any concurrent program performing I/O operations, regardless of the actual language constructs used.

6 Related Work

Different works have dealt with designing reversible programming languages both in the *sequential* and *concurrent* settings.

In the sequential setting there is no need to save causal information among events. A framework for adding a general undo capability (and hence reversibility) to a programming language is presented in [13]. Computational history is saved by means of undo-lists, storing previous states of the execution. In [3, 19, 2] a reversible programming language, its virtual machine and compilation are presented. The key aspect of this language is that all its constructs (including assignments) are *bijective* (and hence reversible). In order to have reversible assignments a syntactic restriction on the possible expressions is imposed. The language is sequential and first-order, however, compared to our higher-order concurrent one. [19] contains several references to reversible sequential languages. A reversible abstract machine which implements the linear head reduction strategy for λ-calculus is presented in [8], where it is related to the well-known Krivine abstract machine.

In the concurrent setting, the works closer to ours are those dealing with programming languages with transactional constructs, such as [15], which exploits a form of undo to implement transactions (but only in a mono-processor setting), or those dealing with explicit checkpointing mechanisms, such as [9], but which implements an imperfect form of reversibility (e.g. rollbacks may not reach a global checkpoint).

A general upper bound on the trade-off between space and time to simulate irreversible computations by means of reversible ones is given in [6]. Moreover, [6] also provides a lower bound on the extra storage space required by step-wise reversible simulation of irreversible computations. Our lower bound on the space overhead that the reversible mechanism requires with respect to a non-reversible computation, is consistent with, though not reducible to, the one presented in [6].

7 Conclusion

We have presented a reversible abstract machine for a small higher-order concurrent programming language, μOz, which is a fragment of the Oz kernel programming language. We have shown that its space overhead on a program execution, compared to a non-reversible abstract machine for μOz directly inspired by the Oz abstract machine, is at most linear in the number of execution steps. This result cannot be much improved for we have also shown that reversing a μOz program requires at least such an amount of information.

There are however a number of ways that our abstract machine can be improved. Notice in particular that the information absolutely required for reversibility is related to potential sources of non-determinism in the execution of μOz programs. One can thus aim to reduce the information stored pertaining to deterministic steps, in particular trading space costs for time costs in backward steps. This would also improve the time costs incurred by forward executions

in our abstract machine. It would also be interesting to see whether insights on lambda-machines in [8] can be leveraged to optimize the deterministic and sequential part of our machine.

On a different track, it would be interesting to study in more detail the costs of implementing *controlled* reversibility as introduced in [11], and to see how we can leverage the presence of explicit instructions for reversibility for different time and space trade-offs.

References

1. T. Altenkirch and J. Grattage. A functional quantum programming language. In *Proc. of LICS'05*, 2005.
2. H. B. Axelsen. Clean translation of an imperative reversible programming language. In *Proc. of ICCC 2011*, volume 6601 of *LNCS*. Springer, 2011.
3. H. B. Axelsen, R. Glück, and T. Yokoyama. Reversible Machine Code and Its Abstract Processor Architecture. In *Proc. of CSR'07*, 2007.
4. C.H. Bennett. Notes on the history of reversible computation. *IBM Journal of Research and Development*, 32(1), 1988.
5. A. B. Brown and D. A. Patterson. Undo for operators: Building an undoable e-mail store. In *USENIX Annual Technical Conference, General Track*. USENIX, 2003.
6. H. Buhrman, J. Tromp, and P. M. B. Vitányi. Time and space bounds for reversible simulation. In *Proc. of ICALP'01*, volume 2076 of *LNCS*. Springer, 2001.
7. V. Danos and J. Krivine. Reversible communicating systems. In *Proc. of CONCUR'04*, volume 3170 of *LNCS*. Springer, 2004.
8. V. Danos and L. Regnier. Reversible, irreversible and optimal lambda-machines. *Theor. Comput. Sci.*, 227(1-2), 1999.
9. J. Field and C.A. Varela. Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments. In *Proc. of POPL'05*. ACM, 2005.
10. M. P. Frank. Introduction to reversible computing: motivation, progress, and challenges. In *2nd Conference on Computing Frontiers*. ACM, 2005.
11. I. Lanese, C.A. Mezzina, A. Schmitt, and J.B. Stefani. Controlling reversibility in higher-order pi. In *Proc. of CONCUR 2011*, volume 6901 of *LNCS*, 2011.
12. I. Lanese, C.A. Mezzina, and J.B. Stefani. Reversing higher-order pi. In *Proc. of CONCUR 2010*, volume 6269 of *LNCS*. Springer, 2010.
13. G.B. Leeman. A formal approach to undo operations in programming languages. *ACM Trans. Program. Lang. Syst.*, 8(1), 1986.
14. M. Li and P. Vitanyi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer, 3rd edition, 2008.
15. M. F. Ringenburg and D. Grossman. AtomCaml: first-class atomicity via rollback. In *Proc. of ICFP'05*. ACM, 2005.
16. P. Van Roy and S. Haridi. *Concepts, Techniques and Models of Computer Programming*. MIT Press, 2004.
17. P. M. B. Vitányi. Time, space, and energy in reversible computing. In *2nd Conference on Computing Frontiers*. ACM, 2005.
18. T. Yokoyama. Reversible computation and reversible programming languages. *Electr. Notes Theor. Comput. Sci.*, 253(6), 2010.
19. T. Yokoyama and R. Glück. A reversible programming language and its invertible self-interpreter. In *Proc. of PEPM'07*. ACM, 2007.