A Small-Step Semantics for Janus

Pietro Lami, Ivan Lanese, Jean-Bernard Stefani

A Small-Step Semantics for Janus

Pietro Lami[1,2(✉)], Ivan Lanese[2], and Jean-Bernard Stefani[1]

[1] Univ. Grenoble Alpes, INRIA, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
pietro.lami@inria.fr
[2] Olas Team, Univ. of Bologna, INRIA, 40126 Bologna, Italy

Abstract. Janus is an imperative, sequential language for reversibility. While heavily studied in the reversibility literature, to the best of our knowledge, no small-step semantics for it exists. Hence, we propose a small-step semantics for Janus and we prove it equivalent to a big-step semantics from the literature, for programs that have no runtime errors and no divergence. Our main motivation is to enable a future extension of Janus with concurrency primitives, which is more easily defined on top of a small-step semantics. As additional feature, a small-step semantics allows one to more easily distinguish between failing and non-terminating computations.

1 Introduction

Janus is the first structured reversible programming language, designed by Christopher Lutz and Howard Derby [8]. Janus is an imperative language, which supports deterministic forward and backward computation. This means that in Janus any forward computation can be undone by a finite sequence of backward steps.

The operational semantics of Janus was formally specified in [17,18,20]. The language and its semantics provide a solid foundation for further research and development in reversible computing. However, to the best of our knowledge, there are no small-step semantics of Janus in the literature. Indeed, the semantics mentioned above are all big-step.

Hence, the main contribution of this paper is to define a small-step semantics for Janus (using the syntax described in [20]), and proving it equivalent to the big-step semantics in the literature in absence of runtime errors and divergence (Theorem 1).

A main motivation for our work is that we are working towards an extension of Janus with concurrency primitives, namely we aim to add the possibility to create processes and to allow processes to communicate via message passing.

The work has been partially supported by French ANR project DCore ANR-18-CE25-0007. The second author has also been partially supported by MSCA-PF project 101106046—ReGraDe-CS and by INdAM – GNCS 2023 project RISICO, code CUP_E53C22001930001. The authors thank the anonymous reviewers for their useful comments and suggestions.

T. Æ. Mogensen and Ł. Mikulski (Eds.): RC 2024, LNCS 14680, pp. 105–123, 2024.
https://doi.org/10.1007/978-3-031-62076-8_8

$$
\begin{array}{rl}
\textit{Programs} & p ::= s\ (\mathsf{procedure}\ id\ s)^{+} \\
\textit{Statements} & s ::= x \oplus = e \mid x[e] \oplus = e \mid \mathsf{if}\ e\ \mathsf{then}\ s\ \mathsf{else}\ s\ \mathsf{fi}\ e \\
 & \quad \mid \mathsf{from}\ e\ \mathsf{do}\ s\ \mathsf{loop}\ s\ \mathsf{until}\ e \mid \mathsf{call}\ id \mid \mathsf{uncall}\ id \mid \mathsf{skip} \mid s\ s \\
\textit{Expressions} & e ::= c \mid x \mid x[e] \mid e \odot e \\
\textit{Constant} & c ::= -2147483648 \mid \cdots \mid 0 \mid \cdots \mid 2147483647 \\
 & \oplus ::= + \mid - \mid \hat{} \\
 & \odot ::= \oplus \mid * \mid / \mid \% \mid \& \mid \&\& \mid \| \mid \prime\prime \mid \prime\prime\prime \mid < \mid > \mid = \mid != \mid <= \mid >=
\end{array}
$$

with x variables and id identifiers

Fig. 1. Language syntax

This is a relevant aim since to the best of our knowledge, there are no pure concurrent reversible languages, namely concurrent languages which make no use of history information to enable reversibility. Indeed, all the concurrent reversible programming languages and calculi we are aware of use history information, as for instance in the cases of RCCS [3], CCSK [12], reversible π [1], reversible higher-order π [6] and reversible Erlang [7].

Notably, the semantics of the formalisms above are all small-step operational semantics, and this is the case also for the most well-known irreversible calculi for concurrency, such as CCS [9] and π-calculus [10]. Indeed, small-step semantics allows one to more easily describe the interleavings among the actions of different processes that take place in concurrent computations.

Summarizing, we believe that it makes sense to study extensions of Janus with concurrency primitives, and that defining a small-step semantics for Janus is a sensible first step in this direction.

Beyond this, small-step semantics and big-step semantics have their own strengths and weaknesses, as discussed, e.g., in [2,11], hence a small-step semantics adds a relevant tool for the study of Janus. We will show for instance that, as noted in [2], small-step semantics allows one to more easily distinguish between failing and non-terminating computations.

2 Background: Janus

Syntax. Janus [17,18,20] is a reversible imperative programming language designed to support both forward and backward computation. We show the syntax of Janus in Fig. 1 (w.r.t. [20] we drop variable declarations, since they are irrelevant for our discussion), where a program consists of a statement and a set of procedure declarations. A procedure declaration includes the keyword `procedure`, an identifier (the procedure name id), and a statement (the procedure body). A statement is a reversible assignment to a variable or an element of array, a reversible conditional, a reversible loop, a procedure call, a procedure uncall, a skip, or a statement sequence. A reversible conditional is formed by two predicates: the test that follows the keyword `if`, and the assertion that follows the keyword `fi`. If the test is true, the then-branch is executed, and afterwards the evaluation of the assertion must be true; if the assertion fails, the semantics

$$\textsc{Con}\ \frac{}{\sigma \vdash c \Downarrow [\![c]\!]} \qquad \textsc{Var}\ \frac{}{\sigma \vdash x \Downarrow \sigma(x)} \qquad \textsc{Arr}\ \frac{\sigma \vdash e \Downarrow v}{\sigma \vdash x[e] \Downarrow \sigma(x[v])}$$

$$\textsc{Bop}\ \frac{\sigma \vdash e_1 \Downarrow v_1 \qquad \sigma \vdash e_2 \Downarrow v_2 \qquad [\![\odot]\!](v_1, v_2) = v}{\sigma \vdash e_1 \odot e_2 \Downarrow v}$$

Fig. 2. Big-step semantics of expressions

of the conditional is undefined. Dually, if the test is false, the assertion must be false after execution of the else-branch. A reversible loop (`from` e_1 `do` s_1 `loop` s_2 `until` e_2) has two predicates, an assertion at the entry (e_1) and a test at the exit of the loop (e_2). Initially, assertion e_1 must be true and then s_1 is executed. The loop terminates if test e_2 is true; otherwise, s_2 is executed, after which e_1 must be false. Indeed, the assertion needs to be true at the first evaluation only. The loop is repeated as long as assertion and test are both false, and terminates when the test is true. A procedure call executes the procedure body in the global store. In this paper, we do not consider parameters or local variables. To pass values to and from a procedure, we use side effects on the global store. A procedure uncall computes the inverse function of the invoked procedure. An expression is a constant, a variable, an indexed variable, or a binary expression. A binary operator $\odot$ is an arithmetic $(+, -, *, /, \%)$, bitwise $(\&, |, \hat{})$, logical $(\&\&, \|)$, or relational operator $(<, >, =, != , <=, >=)$. For a deeper discussion we refer to [20].

Semantics. We describe only the semantics of statements, in a big-step style, and assume that procedure declarations are translated into a function Γ from procedure names to statements. We restrict the attention to programs where all the invoked functions have a definition. We consider the semantics in [20], but for loops we consider the equivalent but simpler semantics in [17,18]. Note that the definition of the semantics requires to extend the syntax of statements. Furthermore, the semantics for statements relies on an auxiliary semantics for expressions.

In Fig. 2 we show the rules for expressions. The rules have the form $\sigma \vdash e \Downarrow v$, where σ is a store, e is an expression and v is a value. A store σ is a function from variable names and indexed variable names to values. We denote with $\sigma[x \mapsto v]$ the update of σ which assigns value v to variable x. For simplicity we assume that σ is defined for all the variables used in the program, and we assume integer variables initialized with value 0.

In Fig. 3 we show the big-step semantics of statements. The rules have the form $\sigma \vdash s \Downarrow \sigma'$ where σ and σ' are stores, and s is a statement. The rules AssVar and AssArr define the assignment. The assignment operator $(\oplus =)$ stands for $(+ =)$, $(- =)$, or $(\hat{} =)$.

A procedure call (defined by rule Call) executes the procedure body $\Gamma(id)$ in the current store. A procedure uncall (defined by rule UnCall) executes the inversion of the body of function $\Gamma(id)$. The inversion is computed thanks to the inverter function $\mathcal{I}$ (defined in Fig. 4), which given a statement computes

$$\textsc{AssVar}\ \frac{\sigma \vdash e \Downarrow v}{\sigma \vdash x \oplus= e \Downarrow \sigma[x \mapsto [\![\oplus]\!](\sigma(x), v)]}$$

$$\textsc{AssArr}\ \frac{\sigma \vdash e_l \Downarrow v_l \qquad \sigma \vdash e \Downarrow v}{\sigma \vdash x[e_l] \oplus= e \Downarrow \sigma[x[v_l] \mapsto [\![\oplus]\!](\sigma(x[v_l]), v)]} \qquad \textsc{Call}\ \frac{\sigma \vdash \Gamma(id) \Downarrow \sigma'}{\sigma \vdash \mathsf{call}\ id \Downarrow \sigma'}$$

$$\textsc{UnCall}\ \frac{\sigma \vdash \mathcal{I}[\![\Gamma(id)]\!] \Downarrow \sigma'}{\sigma \vdash \mathsf{uncall}\ id \Downarrow \sigma'} \qquad \textsc{Seq}\ \frac{\sigma \vdash s_1 \Downarrow \sigma' \qquad \sigma' \vdash s_2 \Downarrow \sigma''}{\sigma \vdash s_1\ s_2 \Downarrow \sigma''}$$

$$\textsc{IfTrue}\ \frac{\sigma \vdash e_1 \Downarrow v_1 \qquad \mathsf{is\_true?}(v_1) \qquad \sigma \vdash s_1 \Downarrow \sigma' \qquad \sigma' \vdash e_2 \Downarrow v_2 \qquad \mathsf{is\_true?}(v_2)}{\sigma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi}\ e_2 \Downarrow \sigma'}$$

$$\textsc{IfFalse}\ \frac{\sigma \vdash e_1 \Downarrow v_1 \qquad \mathsf{is\_false?}(v_1) \qquad \sigma \vdash s_2 \Downarrow \sigma' \qquad \sigma' \vdash e_2 \Downarrow v_2 \qquad \mathsf{is\_false?}(v_2)}{\sigma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi}\ e_2 \Downarrow \sigma'}$$

$$\textsc{LoopMain}\ \frac{\sigma \vdash e_1 \Downarrow v_1 \qquad \mathsf{is\_true?}(v_1) \qquad \sigma \vdash s_1 \Downarrow \sigma' \qquad \sigma' \vdash (e_1, s_1, e_2, s_2) \Downarrow \sigma''}{\sigma \vdash \mathsf{from}\ e_1\ \mathsf{do}\ s_1\ \mathsf{loop}\ s_2\ \mathsf{until}\ e_2 \Downarrow \sigma''}$$

$$\textsc{LoopBase}\ \frac{\sigma \vdash e_2 \Downarrow v_2 \qquad \mathsf{is\_true?}(v_2)}{\sigma \vdash (e_1, s_1, e_2, s_2) \Downarrow \sigma}$$

$$\textsc{LoopRec}\ \frac{\begin{array}{ccc}\sigma \vdash e_2 \Downarrow v_2 & \mathsf{is\_false?}(v_2) & \sigma \vdash s_2 \Downarrow \sigma' \\ \sigma' \vdash e_1 \Downarrow v_1 & \mathsf{is\_false?}(v_1) & \sigma' \vdash s_1 \Downarrow \sigma'' \qquad \sigma'' \vdash (e_1, s_1, e_2, s_2) \Downarrow \sigma'''\end{array}}{\sigma \vdash (e_1, s_1, e_2, s_2) \Downarrow \sigma'''}$$

$$\textsc{Skip}\ \frac{}{\sigma \vdash \mathsf{skip} \Downarrow \sigma}$$

Fig. 3. Big-step semantics of Janus statements

another statement executing the inverse computation. Rule UNCALL is actually different from the one in [20], which is as follows:

$$\textsc{UnCall}\ \frac{\sigma' \vdash \Gamma(id) \Downarrow \sigma}{\sigma \vdash \mathsf{uncall}\ id \Downarrow \sigma'}$$

However, [20, Theorem 4] states that $\sigma \vdash s \Downarrow \sigma'$ iff $\sigma' \vdash \mathcal{I}[\![s]\!] \Downarrow \sigma$ hence the two rules are equivalent. We prefer the one in Fig. 3, since it allows for a more direct correspondence with the small-step semantics. The execution of a statement sequence is defined by rule SEQ.

The conditional is defined by rules IFTRUE and IFFALSE, and which rule applies depends on the value of e_1 and e_2. Indeed, predicates $\mathsf{is\_true?}(v)$ and $\mathsf{is\_false?}(v)$ check the truth value of a value v. Notice that the semantics of conditional is defined only if the two expressions have the same truth value.

The loop is defined by three rules: a rule for entering the loop, LOOPMAIN, a rule for exiting, LOOPBASE, and a rule for iteration, LOOPREC. Rule LOOPMAIN requires the truth of assertion e_1 and then executes statement s_1. The execution of the loop terminates if test e_2 is true, with an application of rule LOOPBASE. Otherwise, the loop continues with rule LOOPREC, which executes s_2 and s_1, and which requires both test e_2 and assertion e_1 to be false. As for conditional,

$$\begin{aligned}
\mathcal{I}_{op}[\![+]\!] &::= - \qquad \mathcal{I}_{op}[\![-]\!] ::= + \qquad \mathcal{I}_{op}[\![\hat{}]\!] ::= \hat{} \\
\mathcal{I}[\![x \oplus = e]\!] &::= x\ \mathcal{I}_{op}[\![\oplus]\!] = e \\
\mathcal{I}[\![x[e_1] \oplus = e_2]\!] &::= x[e_1]\ \mathcal{I}_{op}[\![\oplus]\!] = e_2 \\
\mathcal{I}[\![\mathsf{if}\ e_1\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi}\ e_2]\!] &::= \mathsf{if}\ e_2\ \mathsf{then}\ \mathcal{I}[\![s_1]\!]\ \mathsf{else}\ \mathcal{I}[\![s_2]\!]\ \mathsf{fi}\ e_1 \\
\mathcal{I}[\![\mathsf{from}\ e_1\ \mathsf{do}\ s_1\ \mathsf{loop}\ s_2\ \mathsf{until}\ e_2]\!] &::= \mathsf{from}\ e_2\ \mathsf{do}\ \mathcal{I}[\![s_1]\!]\ \mathsf{loop}\ \mathcal{I}[\![s_2]\!]\ \mathsf{until}\ e_1 \\
\mathcal{I}[\![\mathsf{call}\ id]\!] &::= \mathsf{uncall}\ id \\
\mathcal{I}[\![\mathsf{uncall}\ id]\!] &::= \mathsf{call}\ id \\
\mathcal{I}[\![\mathsf{skip}]\!] &::= \mathsf{skip} \\
\mathcal{I}[\![s_1\ s_2]\!] &::= \mathcal{I}[\![s_1]\!]\ \mathcal{I}[\![s_2]\!]
\end{aligned}$$

Fig. 4. Inverter function for Janus statements

$$\textsc{ConS}\ \frac{}{\langle \sigma, c\rangle \to \langle \sigma, [\![c]\!]\rangle} \qquad \textsc{VarS}\ \frac{}{\langle \sigma, x\rangle \to \langle \sigma, \sigma(x)\rangle} \qquad \textsc{Arr1}\ \frac{\langle \sigma, e\rangle \to \langle \sigma, e'\rangle}{\langle \sigma, x[e]\rangle \to \langle \sigma, x[e']\rangle}$$

$$\textsc{Arr2}\ \frac{}{\langle \sigma, x[v]\rangle \to \langle \sigma, \sigma(x[v])\rangle} \qquad \textsc{Bop1}\ \frac{\langle \sigma, e_1\rangle \to \langle \sigma, e_1'\rangle}{\langle \sigma, e_1 \odot e_2\rangle \to \langle \sigma, e_1' \odot e_2\rangle}$$

$$\textsc{Bop2}\ \frac{\langle \sigma, e_2\rangle \to \langle \sigma, e_2'\rangle}{\langle \sigma, v_1 \odot e_2\rangle \to \langle \sigma, v_1 \odot e_2'\rangle} \qquad \textsc{Bop3}\ \frac{[\![\odot]\!](v_1, v_2) = v}{\langle \sigma, v_1 \odot v_2\rangle \to \langle \sigma, v\rangle}$$

Fig. 5. Small-step semantics of expressions

the semantics of loop is not defined if the value of assertions is not the required one. The Skip rule does nothing. For a deeper discussion we refer to [17,18,20].

3 A Small-Step Semantics for Janus

In this section, we describe the small-step semantics for Janus that we propose, and prove it equivalent to the big-step semantics in the previous section. As for the big-step semantics, we need to extend the syntax of statements.

For the evaluation of expressions, we use the small-step semantics depicted in Fig. 5, where σ is a store and e and e' are expressions. Actually, we need to extend the syntax of expressions to also allow for values as leafs, to cope with partially evaluated expressions, such as $[\![3]\!] + 3$, where the first summand is the value 3 and the second one the corresponding constant. The evaluation stops when the expression reduces to a value v.

The rules for statements, depicted in Figs. 6, 7, 8 and 9, have either the form $\langle \sigma, s\rangle \to \langle \sigma', s'\rangle$, where σ and σ' are stores, and s, s' are statements, to denote a successful step, or the form $\langle \sigma, s\rangle \to \bot$ to denote a runtime error due to a failed assertion. In order to give a compact definition of the semantics, we use contextual rules to lift the evaluation of expressions and statements to larger contexts. To this end we need to define evaluation contexts.

Definition 1 (Evaluation contexts). *An* evaluation context *is a statement where a subterm is replaced by a* $\bullet$*. We distinguish expression contexts* $C_e[\bullet]$ *and statement contexts* $C_s[\bullet]$*. We denote with* $C_e[e]$ *the statement obtained by*

$$\textsc{AssVarS}\ \frac{}{\langle \sigma, x \oplus= v\rangle \to \langle \sigma[x \mapsto [\![\oplus]\!](\sigma(x), v)], \mathsf{skip}\rangle}$$

$$\textsc{AssArrS}\ \frac{}{\langle \sigma, x[v_l] \oplus= v\rangle \to \langle \sigma[x[v_l] \mapsto [\![\oplus]\!](\sigma(x[v_l]), v)], \mathsf{skip}\rangle}$$

$$\textsc{CallS}\ \frac{}{\langle \sigma, \mathsf{call}\ id\rangle \to \langle \sigma, \Gamma(id)\rangle} \qquad \textsc{UnCallS}\ \frac{}{\langle \sigma, \mathsf{uncall}\ id\rangle \to \langle \sigma, \mathcal{I}[\![\Gamma(id)]\!]\rangle}$$

$$\textsc{SeqS}\ \frac{}{\langle \sigma, \mathsf{skip}\ s_2\rangle \to \langle \sigma, s_2\rangle}$$

Fig. 6. Small-step semantics: rules for basic constructs

$$\textsc{CtxExp}\ \frac{\langle \sigma, e\rangle \to \langle \sigma, e'\rangle}{\langle \sigma, C_e[e]\rangle \to \langle \sigma, C_e[e']\rangle} \qquad \textsc{CtxStmt}\ \frac{\langle \sigma, s\rangle \to \langle \sigma', s'\rangle}{\langle \sigma, C_s[s]\rangle \to \langle \sigma', C_s[s']\rangle}$$

$$\textsc{CtxError}\ \frac{\langle \sigma, s\rangle \to \bot}{\langle \sigma, C_s[s]\rangle \to \bot}$$

Fig. 7. Small-step semantics: contextual rules

replacing $\bullet$ *with expression* e *in expression context* $C_e[\bullet]$*, and* $C_s[s]$ *the one obtained by replacing* $\bullet$ *with statement* s *in statement context* $C_s[\bullet]$*. Contexts are defined as:*

$$\begin{aligned}
C_e[\bullet] = {} & x \oplus= \bullet \mid x[\bullet] \oplus= e \mid x[v] \oplus= \bullet \mid \mathsf{if}\ \bullet\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi}\ e_2 \\
& \mid \mathsf{if}\ v_1\ \mathsf{then\ skip\ else}\ s_2\ \mathsf{fi}\ \bullet \qquad \textit{where}\ \mathsf{is_true?}(v_1) \\
& \mid \mathsf{if}\ v_1\ \mathsf{then}\ s_1\ \mathsf{else\ skip\ fi}\ \bullet \qquad \textit{where}\ \mathsf{is_false?}(v_1) \\
& \mid ((\bullet, e_1), s_1, e_2, s_2) \mid (e_1, (\mathsf{skip}, s_1), (\bullet, e_2), s_2) \\
& \mid ((\bullet, e_1), (\mathsf{skip}, s_1), e_2, (\mathsf{skip}, s_2)) \\
C_s[\bullet] = {} & \bullet\ s \\
& \mid \mathsf{if}\ v_1\ \mathsf{then}\ \bullet\ \mathsf{else}\ s_2\ \mathsf{fi}\ e_2 \qquad \textit{where}\ \mathsf{is_true?}(v_1) \\
& \mid \mathsf{if}\ v_1\ \mathsf{then}\ s_1\ \mathsf{else}\ \bullet\ \mathsf{fi}\ e_2 \qquad \textit{where}\ \mathsf{is_false?}(v_1) \\
& \mid (e_1, (\bullet, s_1), (e_2, e_2), s_2) \mid ((e_1, e_1), (\mathsf{skip}, s_1), e_2, (\bullet, s_2))
\end{aligned}$$

Rules AssVarS and AssArrS in Fig. 6 define the assignment. AssVarS evaluates the new value of a variable and updates the store, and AssArrS does the same with an element of an array.

A procedure call (defined by rule CallS) reduces to the procedure body $\Gamma(id)$, in the current store. A procedure uncall (defined by rule UnCallS) reduces to the inversion $\mathcal{I}[\![\Gamma(id)]\!]$ of the body of procedure id, retrieved using function Γ. Here $\mathcal{I}$ is the inverter function defined in Fig. 4.

$$\textsc{IfTrueEnd}\ \frac{\mathsf{is\_true?}(v_1) \qquad \mathsf{is\_true?}(v_2)}{\langle \sigma, \mathsf{if}\ v_1\ \mathsf{then\ skip\ else}\ s_2\ \mathsf{fi}\ v_2\rangle \to \langle \sigma, \mathsf{skip}\rangle}$$

$$\textsc{IfFalseEnd}\ \frac{\mathsf{is\_false?}(v_1) \qquad \mathsf{is\_true?}(v_2)}{\langle \sigma, \mathsf{if}\ v_1\ \mathsf{then}\ s_1\ \mathsf{else\ skip\ fi}\ v_2\rangle \to \langle \sigma, \mathsf{skip}\rangle}$$

$$\textsc{IfError1}\ \frac{\mathsf{is\_true?}(v_1) \qquad \mathsf{is\_false?}(v_2)}{\langle \sigma, \mathsf{if}\ v_1\ \mathsf{then\ skip\ else}\ s_2\ \mathsf{fi}\ v_2\rangle \to \bot}$$

$$\textsc{IfError2}\ \frac{\mathsf{is\_false?}(v_1) \qquad \mathsf{is\_true?}(v_2)}{\langle \sigma, \mathsf{if}\ v_1\ \mathsf{then}\ s_1\ \mathsf{else\ skip\ fi}\ v_2\rangle \to \bot}$$

Fig. 8. Small-step semantics: rules for if

The execution of a sequence is defined by rule SeqS, which removes the first statement if its evaluation is terminated. Notably, there is no rule for skip, since $\langle \sigma, \mathsf{skip}\rangle$ denotes the end of the computation.

The rules in Fig. 7 describe how reductions of expressions (CtxExp) and of statements (CtxStmt) are lifted to larger contexts in successful steps. Similarly, rule CtxError lifts a runtime error.

The conditional (Fig. 8) is defined by four rules: IfTrueEnd, IfFalseEnd, IfError1 and IfError2. These rules are applied after having evaluated, using the CtxExp and CtxStmt rules, the test condition, the statement in the selected branch, and the assertion. The first two rules define a successful step when the test and the assertion are either both true (IfTrueEnd case) or both false (IfFalseEnd case). Rules IfError1 and IfError2 are used to signal a runtime error when the outcomes of the evaluation of the test and of the assertion differ. Notably, in such a case the big-step semantics is not defined.

The loop (Fig. 9) is defined by rules LoopMainS, LoopExp1T, LoopEndS, LoopExp2F, and LoopExp1F. The interplay among these rules as well as the relations with the corresponding rules in the big-step semantics (cf. Fig. 3) is depicted in Fig. 10 to help the understanding. Rule LoopMainS enters the loop. Rule LoopExp1T is fired after the evaluation of the expression, performed thanks to rule CtxExp. Rule LoopExp1T requires assertion e_1 to be true and it enables the execution of statement s_1, done using rule CtxStmt. The execution terminates with rule LoopEndS if test e_2 (evaluated via rule CtxExp) is true. If the evaluation of test e_2 is false, instead, an iteration is needed, and rule LoopExp2F enables the execution of statement s_2 via rule CtxStmt. Then, rule CtxExp evaluates the assertion e_1, that must be false. This lead to another iteration using rule LoopExp1F. The assertion e_1 should be true at the first evaluation, and false in the others. When this is not the case rules LoopError1 or LoopError2 are fired, raising a runtime error. The big step semantics is undefined in these two last cases.

Before proving the equivalence between the big-step and the small-step semantics (when the execution terminates without runtime errors), we prove

$$\textsc{LoopMainS}\ \frac{}{\langle \sigma, \mathsf{from}\ e_1\ \mathsf{do}\ s_1\ \mathsf{loop}\ s_2\ \mathsf{until}\ e_2\rangle \to \langle \sigma, ((e_1, e_1), s_1, e_2, s_2)\rangle}$$

$$\textsc{LoopExp1T}\ \frac{\mathsf{is\_true?}(v_1)}{\langle \sigma, ((v_1, e_1), s_1, e_2, s_2)\rangle \to \langle \sigma, (e_1, (s_1, s_1), (e_2, e_2), s_2)\rangle}$$

$$\textsc{LoopEndS}\ \frac{\mathsf{is\_true?}(v_2)}{\langle \sigma, (e_1, (\mathsf{skip}, s_1), (v_2, e_2), s_2)\rangle \to \langle \sigma, \mathsf{skip}\rangle}$$

$$\textsc{LoopExp2F}\ \frac{\mathsf{is\_false?}(v_2)}{\langle \sigma, (e_1, (\mathsf{skip}, s_1), (v_2, e_2), s_2)\rangle \to \langle \sigma, ((e_1, e_1), (\mathsf{skip}, s_1), e_2, (s_2, s_2))\rangle}$$

$$\textsc{LoopExp1F}\ \frac{\mathsf{is\_false?}(v_1)}{\langle \sigma, ((v_1, e_1), (\mathsf{skip}, s_1), e_2, (\mathsf{skip}, s_2))\rangle \to \langle \sigma, (e_1, (s_1, s_1), (e_2, e_2), s_2)\rangle}$$

$$\textsc{LoopError1}\ \frac{\mathsf{is\_false?}(v_1)}{\langle \sigma, ((v_1, e_1), s_1, e_2, s_2)\rangle \to \bot}$$

$$\textsc{LoopError2}\ \frac{\mathsf{is\_true?}(v_1)}{\langle \sigma, ((v_1, e_1), (\mathsf{skip}, s_1), e_2, (\mathsf{skip}, s_2))\rangle \to \bot}$$

Fig. 9. Small-step semantics: rules for loop

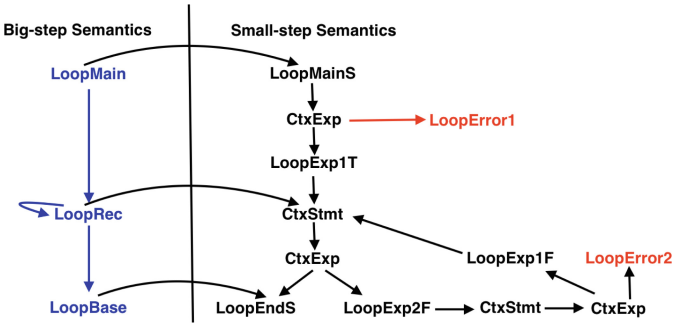

Fig. 10. Loop schema

an analgous result for the evaluation of expressions. We denote with $\to^*$ the reflexive and transitive closure of $\to$ and with $\to^+$ the transitive one.

Lemma 1.

$$\sigma \vdash e \Downarrow v \ \textit{iff}\ \langle \sigma, e\rangle \to^+ \langle \sigma, v\rangle$$

where e does not contain values.

Proof. $\Rightarrow$: The proof is by induction on the derivation, with a case analysis on the last applied rule.

Con: we have that $\sigma \vdash c \Downarrow [\![c]\!]$. The thesis follows using rule ConS to derive $\langle \sigma, c\rangle \to \langle \sigma, [\![c]\!]\rangle$. The rules have no hypothesis.

Var: we have that $\sigma \vdash x \Downarrow \sigma(x)$. The thesis follows using rule VarS to derive $\langle \sigma, x\rangle \to \langle \sigma, \sigma(x)\rangle$. The rules have no hypothesis.

Arr: we have that $\sigma \vdash x[e] \Downarrow \sigma(x[v])$ with hypothesis $\sigma \vdash e \Downarrow v$. By inductive hypothesis we have $\langle \sigma, e\rangle \rightarrow^{+} \langle \sigma, v\rangle$. Then we can apply rule ARR1 one or more times to derive $\langle \sigma, x[e]\rangle \rightarrow^{+} \langle \sigma, x[v]\rangle$. The thesis follows using rule ARR2 to derive $\langle \sigma, x[v]\rangle \rightarrow \langle \sigma, \sigma(x[v])\rangle$.

Bop: we have that $\sigma \vdash e_1 \odot e_2 \Downarrow v$ with hypothesis $\sigma \vdash e_1 \Downarrow v_1$, $\sigma \vdash e_2 \Downarrow v_2$, $[\![\odot]\!](v_1, v_2) = v$. By inductive hypothesis we have $\langle \sigma, e_1\rangle \rightarrow^{+} \langle \sigma, v_1\rangle$, and $\langle \sigma, e_2\rangle \rightarrow^{+} \langle \sigma, v_2\rangle$. Then we can apply rule BOP1 zero or more times to lift the first computation, and rule BOP2 zero or more times to lift the second one. The thesis follows using rule BOP3 with the last hypothesis.

$\Leftarrow$: The proof is by induction on the number of steps in the computation.

Base case: we have $\langle \sigma, e\rangle \rightarrow \langle \sigma, v\rangle$ in one step. We perform a sub-induction on the derivation of the step. We have a case for each rule.

- **ConS:** we have that $\langle \sigma, c\rangle \rightarrow \langle \sigma, [\![c]\!]\rangle$. The thesis follows using rule CON to derive $\sigma \vdash c \Downarrow [\![c]\!]$. The rules have no hypothesis.
- **VarS:** we have that $\langle \sigma, x\rangle \rightarrow \langle \sigma, \sigma(x)\rangle$. The thesis follows using rule VAR to derive $\sigma \vdash x \Downarrow \sigma(x)$. The rules have no hypothesis.
- **Arr2:** we have that $\langle \sigma, x[v]\rangle \rightarrow \langle \sigma, \sigma(x[v])\rangle$. This case does not match the hypothesis, since $x[v]$ contains a value. Indeed, applications of ARR2 occur only as part of longer derivations which start with rule ARR1. We will prove below the case of ARR1 without relying on hypothesis on the case of ARR2.
- **Bop3:** we have that $\langle \sigma, v_1 \odot v_2\rangle \rightarrow \langle \sigma, v\rangle$. As before, there is nothing to prove here since $v_1 \odot v_2$ contains values.

Inductive case: we have $\langle \sigma, e\rangle \rightarrow \langle \sigma, e'\rangle \rightarrow^{+} \langle \sigma, v\rangle$. We perform a sub-induction on the derivation of the first step. We have a case for each rule.

- **Arr1:** we have $\langle \sigma, x[e]\rangle \rightarrow \langle \sigma, x[e']\rangle \rightarrow^{+} \langle \sigma, \sigma(x[v])\rangle$. By rule inspection we know that first only rule ARR1 is applicable, followed by an application of rule ARR2. By considering the hypotheses of the applications of rule ARR1, we have that $\langle \sigma, e\rangle \rightarrow^{+} \langle \sigma, v\rangle$. By inductive hypothesis we know that $\sigma \vdash e \Downarrow v$. We can derive the conclusion using rule ARR.
- **Bop1:** we have $\langle \sigma, e_1 \odot e_2\rangle \rightarrow \langle \sigma, e_1' \odot e_2\rangle \rightarrow^{+} \langle \sigma, v\rangle$. By rule inspection we know that only rule BOP1 is applicable, and by considering the hypotheses of its applications, we have that $\langle \sigma, e_1\rangle \rightarrow^{+} \langle \sigma, v_1\rangle$. Afterwards, the only applicable rule is BOP2, and from the hypotheses of the rule we have that $\langle \sigma, e_2\rangle \rightarrow^{+} \langle \sigma, v_2\rangle$. In the end the only applicable rule is BOP3, and from the hypotheses of the rule we have that $[\![\odot]\!](v_1, v_2) = v$. By inductive hypothesis we know that $\sigma \vdash e_1 \Downarrow v_1$ and $\sigma \vdash e_2 \Downarrow v_2$. Thanks to this hypothesis of rule BOP3 we can derive the conclusion using rule BOP.
- **Bop2:** we have that $\langle \sigma, v_1 \odot e_2\rangle \rightarrow \langle \sigma, e_2'\rangle$. There is nothing to prove here since $v_1 \odot e_2$ contains a value. □

Now we prove the equivalence between the big-step and the small-step semantics of statements, when the execution terminates without runtime errors. The proof is not trivial, since not all the small-step rules have a direct correspondence in the big-step rules (see, e.g., Fig. 10 for the rules related to the loop),

and since the loop relies on some runtime syntax for evaluation, differently from what happens for classical irreversible while loop semantics.

Theorem 1.

$$\sigma \vdash s \Downarrow \sigma' \textit{ iff } \langle \sigma, s \rangle \rightarrow^* \langle \sigma', \mathsf{skip} \rangle$$

where expressions inside s do not contain values.

Proof. $\Rightarrow$: The proof is by induction on the derivation, with a case analysis on the last applied rule.

We extend the inductive hypothesis to cover the additional case where s includes also the runtime syntax for big-step semantics (e_1, s_1, e_2, s_2), where we have:

$$\sigma \vdash (e_1, s_1, e_2, s_2) \Downarrow \sigma' \text{ iff } \langle \sigma, (e_1, (\mathsf{skip}, s_1), (e_2, e_2), s_2) \rangle \rightarrow^* \langle \sigma', \mathsf{skip} \rangle$$

AssVar: we have $\sigma \vdash x \oplus = e \Downarrow \sigma[x \mapsto [\![\oplus]\!](\sigma(x), v)]$ with hypothesis $\sigma \vdash e \Downarrow v$. From Lemma 1 we deduce that $\langle \sigma, e \rangle \rightarrow^+ \langle \sigma, v \rangle$. The thesis follows using rule CtxExp one or more times to lift the previous computation to $\langle \sigma, x \oplus = e \rangle \rightarrow^* \langle \sigma, x \oplus = v \rangle$. We conclude by applying rule AssVarS to derive $\langle \sigma, x \oplus = v \rangle \rightarrow \langle \sigma[x \mapsto [\![\oplus]\!](\sigma(x), v)], \mathsf{skip} \rangle$.

AssArr: we have $\sigma \vdash x[e_l] \oplus = e \Downarrow \sigma[x[v_l] \mapsto [\![\oplus]\!](\sigma(x[v_l]), v)]$ with hypothesis $\sigma \vdash e_l \Downarrow v_l$ and $\sigma \vdash e \Downarrow v$. From Lemma 1 we can derive $\langle \sigma, e_l \rangle \rightarrow^+ \langle \sigma, v_l \rangle$ and $\langle \sigma, e \rangle \rightarrow^+ \langle \sigma, v \rangle$. The thesis follows using one or more times rule CtxExp and finally rule AssArrS to derive $\langle \sigma, x[e_l] \oplus = e \rangle \rightarrow^* \langle \sigma, x[v_l] \oplus = e \rangle \rightarrow^* \langle \sigma, x[v_l] \oplus = v \rangle \rightarrow \langle \sigma[x[v_l] \mapsto [\![\oplus]\!](\sigma(x[v_l]), v)], \mathsf{skip} \rangle$.

Call: we have $\sigma \vdash \mathsf{call}\ id \Downarrow \sigma'$ with hypothesis $\sigma \vdash \Gamma(id) \Downarrow \sigma'$. By inductive hypothesis we have $\langle \sigma, \Gamma(id) \rangle \rightarrow^* \langle \sigma', \mathsf{skip} \rangle$. Then the thesis follows using rule CallS.

UnCall: we have $\sigma \vdash \mathsf{uncall}\ id \Downarrow \sigma'$ with hypothesis $\sigma \vdash \mathcal{I}[\![\Gamma(id)]\!] \Downarrow \sigma'$. By inductive hypothesis we have $\langle \sigma, \mathcal{I}[\![\Gamma(id)]\!] \rangle \rightarrow^* \langle \sigma', \mathsf{skip} \rangle$. Then the thesis follows using rule UnCallS.

Skip: the thesis follows by executing a zero steps derivation.

IfTrue: by hypothesis we have that $\sigma \vdash e_1 \Downarrow v_1$, $\mathsf{is\_true?}(v_1)$, $\sigma \vdash s_1 \Downarrow \sigma'$, $\sigma' \vdash e_2 \Downarrow v_2$ and $\mathsf{is\_true?}(v_2)$. From Lemma 1 we deduce that $\langle \sigma, e_1 \rangle \rightarrow^+ \langle \sigma, v_1 \rangle$ and $\langle \sigma', e_2 \rangle \rightarrow^+ \langle \sigma', v_2 \rangle$. By inductive hypothesis we have that $\langle \sigma, s_1 \rangle \rightarrow^* \langle \sigma', \mathsf{skip} \rangle$. Then we can apply first one or more times rule CtxExp to derive $\langle \sigma, \mathsf{if}\ e_1\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi}\ e_2 \rangle \rightarrow^+ \langle \sigma, \mathsf{if}\ v_1\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi}\ e_2 \rangle$, then zero or more times rule CtxStmt to derive $\langle \sigma, \mathsf{if}\ v_1\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi}\ e_2 \rangle \rightarrow^* \langle \sigma', \mathsf{if}\ v_1\ \mathsf{then\ skip\ else}\ s_2\ \mathsf{fi}\ e_2 \rangle$, then one or more times rule CtxExp to derive $\langle \sigma', \mathsf{if}\ v_1\ \mathsf{then\ skip\ else}\ s_2\ \mathsf{fi}\ e_2 \rangle \rightarrow^+ \langle \sigma', \mathsf{if}\ v_1\ \mathsf{then\ skip\ else}\ s_2\ \mathsf{fi}\ v_2 \rangle$ and finally rule IfTrueEnd to obtain $\langle \sigma', \mathsf{if}\ v_1\ \mathsf{then\ skip\ else}\ s_2\ \mathsf{fi}\ v_2 \rangle \rightarrow \langle \sigma', \mathsf{skip} \rangle$. The thesis follows.

IfFalse: analogous to the one above.

LoopMain: we have $\sigma \vdash \mathsf{from}\ e_1\ \mathsf{do}\ s_1\ \mathsf{loop}\ s_2\ \mathsf{until}\ e_2 \Downarrow \sigma''$ with hypotheses $\sigma \vdash e_1 \Downarrow v_1$, $\mathsf{is\_true?}(v_1)$, $\sigma \vdash s_1 \Downarrow \sigma'$, and $\sigma' \vdash (e_1, s_1, e_2, s_2) \Downarrow \sigma''$. From Lemma 1 we deduce $\langle \sigma, e_1 \rangle \rightarrow^+ \langle \sigma, v_1 \rangle$. Then, thanks to the two first

hypotheses, we can apply rule LoopMainS, one or more times rule CtxExp and finally rule LoopExp1T and obtain $\langle \sigma, \mathsf{from}\ e_1\ \mathsf{do}\ s_1\ \mathsf{loop}\ s_2\ \mathsf{until}\ e_2\rangle \rightarrow \langle \sigma, ((e_1, e_1), s_1, e_2, s_2)\rangle \rightarrow^+ \langle \sigma, ((v_1, e_1), s_1, e_2, s_2)\rangle \rightarrow \langle \sigma, (e_1, (s_1, s_1), (e_2, e_2), s_2)\rangle$. Thanks to the hypothesis $\sigma \vdash s_1 \Downarrow \sigma'$ we can apply the inductive hypothesis to obtain $\langle \sigma, s_1\rangle \rightarrow^* \langle \sigma', \mathsf{skip}\rangle$. Now, we can apply rule CtxStmt zero or more times to lift the previous computation to $\langle \sigma, (e_1, (s_1, s_1), (e_2, e_2), s_2)\rangle \rightarrow^* \langle \sigma', (e_1, (\mathsf{skip}, s_1), (e_2, e_2), s_2)\rangle$. We can finally apply the extended inductive hypothesis to derive $\langle \sigma', (e_1, (\mathsf{skip}, s_1), (e_2, e_2), s_2)\rangle \rightarrow^* \langle \sigma'', \mathsf{skip}\rangle$.

LoopBase: we have $\sigma \vdash (e_1, s_1, e_2, s_2) \Downarrow \sigma$ with hypotheses $\sigma \vdash e_2 \Downarrow v_2$ and $\mathsf{is\_true?}(v_2)$. From Lemma 1 we can deduce $\langle \sigma, e_2\rangle \rightarrow^+ \langle \sigma, v_2\rangle$. We can then use it to apply one or more times rule CtxExp and then rule LoopEndS to prove the thesis for the extended case of the inductive hypothesis.

LoopRec: we have $\sigma \vdash (e_1, s_1, e_2, s_2) \Downarrow \sigma'''$ with hypotheses $\sigma \vdash e_2 \Downarrow v_2$, $\mathsf{is\_false?}(v_2)$, $\sigma \vdash s_2 \Downarrow \sigma'$, $\sigma' \vdash e_1 \Downarrow v_1$, $\mathsf{is\_false?}(v_1)$, $\sigma' \vdash s_1 \Downarrow \sigma''$ and $\sigma'' \vdash (e_1, s_1, e_2, s_2) \Downarrow \sigma'''$. From Lemma 1 we deduce $\langle \sigma, e_2\rangle \rightarrow^+ \langle \sigma, v_2\rangle$ and $\langle \sigma', e_1\rangle \rightarrow^+ \langle \sigma', v_1\rangle$. By inductive hypothesis we know that $\langle \sigma, s_2\rangle \rightarrow^* \langle \sigma', \mathsf{skip}\rangle$ and that $\langle \sigma', s_1\rangle \rightarrow^* \langle \sigma'', \mathsf{skip}\rangle$.
We have to prove $\langle \sigma, (e_1, (\mathsf{skip}, s_1), (e_2, e_2), s_2)\rangle \rightarrow^* \langle \sigma''', \mathsf{skip}\rangle$.
To do so, we can apply rule CtxExp one or more times to obtain $\langle \sigma, (e_1, (\mathsf{skip}, s_1), (e_2, e_2), s_2)\rangle \rightarrow^+ \langle \sigma, (e_1, (\mathsf{skip}, s_1), (v_2, e_2), s_2)\rangle$. Then, we can apply rule LoopExp2F (since we know that the hypothesis holds) to obtain $\langle \sigma, (e_1, (\mathsf{skip}, s_1), (v_2, e_2), s_2)\rangle \rightarrow \langle \sigma, ((e_1, e_1), (\mathsf{skip}, s_1), e_2, (s_2, s_2))\rangle$, rule CtxStmt zero or more times to derive $(\langle \sigma, ((e_1, e_1), (\mathsf{skip}, s_1), e_2, (s_2, s_2))\rangle \rightarrow^* \langle \sigma', ((e_1, e_1), (\mathsf{skip}, s_1), e_2, (\mathsf{skip}, s_2))\rangle)$. Then we apply rule CtxExp one or more times to derive $\langle \sigma', ((e_1, e_1), (\mathsf{skip}, s_1), e_2, (\mathsf{skip}, s_2))\rangle \rightarrow^+ \langle \sigma', ((v_1, e_1), (\mathsf{skip}, s_1), e_2, (\mathsf{skip}, s_2))\rangle$, and finally rule LoopExp1F to get $(\langle \sigma', ((v_1, e_1), (\mathsf{skip}, s_1), e_2, (\mathsf{skip}, s_2))\rangle \rightarrow \langle \sigma', (e_1, (s_1, s_1), (e_2, e_2), s_2)\rangle)$. We can now apply rule CtxStmt zero or more times to lift $\langle \sigma', s_1\rangle \rightarrow^* \langle \sigma'', \mathsf{skip}\rangle$ to $\langle \sigma', (e_1, (s_1, s_1), (e_2, e_2), s_2)\rangle \rightarrow^* \langle \sigma'', (e_1, (\mathsf{skip}, s_1), (e_2, e_2), s_2)\rangle$. We can finally apply the extended inductive hypothesis to obtain $\langle \sigma'', (e_1, (\mathsf{skip}, s_1), (e_2, e_2), s_2)\rangle \rightarrow^* \langle \sigma''', \mathsf{skip}\rangle$. The thesis follows.

Seq: we have $\sigma \vdash s_1\ s_2 \Downarrow \sigma'$ with hypothesis $\sigma \vdash s_1 \Downarrow \sigma''$ and $\sigma'' \vdash s_2 \Downarrow \sigma'$. By inductive hypothesis we have $\langle \sigma, s_1\rangle \rightarrow^* \langle \sigma'', \mathsf{skip}\rangle$ and $\langle \sigma'', s_2\rangle \rightarrow^* \langle \sigma', \mathsf{skip}\rangle$. The thesis follows by applying rule CtxStmt zero or more times, then rule SeqS and finally using the inductive hypothesis.

$\Leftarrow$: The proof is by induction on the number of steps in the computation.

Base case: $\langle \sigma, \mathsf{skip}\rangle \rightarrow^* \langle \sigma, \mathsf{skip}\rangle$ in zero steps: then $\sigma \vdash \mathsf{skip} \Downarrow \sigma$ using rule Skip.

Inductive case: $\langle \sigma, s\rangle \rightarrow \langle \sigma', s'\rangle \rightarrow^* \langle \sigma'', \mathsf{skip}\rangle$ and by inductive hypothesis we know that $\sigma' \vdash s' \Downarrow \sigma''$. We perform a sub-induction on the derivation of the first step. We have a case for each rule. Note that inductive hypothesis is only needed in some of the cases.

CtxExp: here we do a case analysis on the possible context:

- $x \oplus = \bullet$ **:** we have $\langle \sigma, x \oplus = e \rangle \to \langle \sigma, x \oplus = e' \rangle$. The only possibility for the subsequent steps is to apply zero or more times rule CtxExp and then rule AssVarS to obtain $\langle \sigma, x \oplus = e' \rangle \to^* \langle \sigma, x \oplus = v \rangle \to \langle \sigma[x \mapsto [\![\oplus]\!](\sigma(x), v)], \mathsf{skip} \rangle$. Thanks to the hypotheses of the rules, we can deduce $\sigma \vdash e \Downarrow v$ from Lemma 1. We can derive the conclusion using AssVar.
- $x[\bullet] \oplus = e$**:** we have $\langle \sigma, x[e_l] \oplus = e \rangle \to \langle \sigma, x[e'_l] \oplus = e \rangle$. The only possibility for the subsequent steps is to apply one or more times rule CtxExp to obtain $\langle \sigma, x[e'_l] \oplus = e \rangle \to^* \langle \sigma, x[v_l] \oplus = e \rangle \to^+ \langle \sigma, x[v_l] \oplus = v \rangle$. Then the only applicable rule for the next step is AssArrS and the result is $\langle \sigma, x[v_l] \oplus = v \rangle \to \langle \sigma[x[v_l] \mapsto [\![\oplus]\!](\sigma(x[v_l]), v)], \mathsf{skip} \rangle$. Thanks to the hypotheses of the rules, we can deduce $\sigma \vdash e \Downarrow v$ and $\sigma \vdash e_l \Downarrow v_l$ from Lemma 1. We can derive the conclusion using AssArr.
- $x[v] \oplus = \bullet$**:** as the second part of the previous case.
- $\mathsf{if}\ \bullet\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi}\ e_2$**:** we have $\langle \sigma, \mathsf{if}\ e_1\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi}\ e_2 \rangle \to \langle \sigma, \mathsf{if}\ e'_1\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi}\ e_2 \rangle$. For the subsequent steps we can only apply rule CtxExp zero or more times, to derive $\langle \sigma, \mathsf{if}\ e'_1\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi}\ e_2 \rangle \to^* \langle \sigma, \mathsf{if}\ v_1\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi}\ e_2 \rangle$. The continuation of the computation depends on the truth value of v_1:
  - **true:** by rule inspection we know that only rule CtxStmt is applicable, and by considering the hypotheses of its applications, we have that $\langle \sigma, s_1 \rangle \to^* \langle \sigma', \mathsf{skip} \rangle$. Afterwards, the only applicable rule is CtxExp, and from the hypotheses of the rule we can derive $\sigma \vdash e_2 \Downarrow v_2$ from Lemma 1. Finally the only applicable rule is IfTrueEnd, and from the hypothesis of the rule we have that $\mathsf{is\_true?}(v_2)$. By inductive hypothesis we know that $\sigma \vdash s_1 \Downarrow \sigma'$. We can derive the conclusion using rule IfTrue.
  - **false:** analogous to the true case.
- $\mathsf{if}\ v_1\ \mathsf{then}\ \mathsf{skip}\ \mathsf{else}\ s_2\ \mathsf{fi}\ \bullet$ where $\mathsf{is\_true?}(v_1)$**:** we have $\langle \sigma, \mathsf{if}\ v_1\ \mathsf{then}\ \mathsf{skip}\ \mathsf{else}\ s_2\ \mathsf{fi}\ e_2 \rangle \to \langle \sigma, \mathsf{if}\ v_1\ \mathsf{then}\ \mathsf{skip}\ \mathsf{else}\ s_2\ \mathsf{fi}\ e'_2 \rangle$. For the subsequent steps we can only apply zero or more times rule CtxExp and then rule IfTrueEnd to obtain $\langle \sigma, \mathsf{if}\ v_1\ \mathsf{then}\ \mathsf{skip}\ \mathsf{else}\ s_2\ \mathsf{fi}\ e'_2 \rangle \to^* \langle \sigma, \mathsf{if}\ v_1\ \mathsf{then}\ \mathsf{skip}\ \mathsf{else}\ s_2\ \mathsf{fi}\ v_2 \rangle \to \langle \sigma, \mathsf{skip} \rangle$. Then we know that $\sigma \vdash e_2 \Downarrow v_2$, $\sigma \vdash v_1 \Downarrow v_1$, $\mathsf{is\_true?}(v_1)$ and $\mathsf{is\_true?}(v_2)$. Thanks to these hypotheses we can derive the conclusion using rule IfTrue.
- $\mathsf{if}\ v_1\ \mathsf{then}\ s_1\ \mathsf{else}\ \mathsf{skip}\ \mathsf{fi}\ \bullet$ where $\mathsf{is\_false?}(v_1)$**:** analogous to the previous case.
- $((\bullet, e_1), s_1, e_2, s_2)$**:** the rule never matches our extended inductive hypothesis, hence there is nothing to prove.
- $(e_1, (\mathsf{skip}, s_1), (\bullet, e_2), s_2)$**:** the rule matches our extended inductive hypothesis only when $\bullet = e_2$. We have $\langle \sigma, (e_1, (\mathsf{skip}, s_1), (e_2, e_2), s_2) \rangle \to \langle \sigma, (e_1, (\mathsf{skip}, s_1), (e'_2, e_2), s_2) \rangle \to^* \langle \sigma', \mathsf{skip} \rangle$. By rule inspection we know that the computation $\langle \sigma, (e_1, (\mathsf{skip}, s_1), (e'_2, e_2), s_2) \rangle \to^*$

$\langle \sigma', \mathsf{skip} \rangle$ has a prefix, derived by applying at each step rule CtxExp, of the form $\langle \sigma, (e_1, (\mathsf{skip}, s_1), (e_2', e_2), s_2) \rangle \rightarrow^* \langle \sigma, (e_1, (\mathsf{skip}, s_1), (v_2, e_2), s_2) \rangle$.

By considering the hypothesis of the applications of rule CtxExp we have that $\langle \sigma, e_2 \rangle \rightarrow^+ \langle \sigma, v_2 \rangle$. Now, we can have two possible continuations, depending on the truth value of v_2. If it is true, we can apply rule LoopEndS, and the thesis follows applying rule LoopBase. The second continuation is $\langle \sigma, (e_1, (\mathsf{skip}, s_1), (v_2, e_2), s_2) \rangle \rightarrow \langle \sigma, ((e_1, e_1), (\mathsf{skip}, s_1), e_2, (s_2, s_2)) \rangle \rightarrow^+ \langle \sigma, ((v_1, e_1), (\mathsf{skip}, s_1), e_2, (s_2, s_2)) \rangle \rightarrow^* \langle \sigma'', ((v_1, e_1), (\mathsf{skip}, s_1), e_2, (\mathsf{skip}, s_2)) \rangle \rightarrow \langle \sigma'', (e_1, (s_1, s_1), (e_2, e_2), s_2) \rangle \rightarrow^* \langle \sigma''', (e_1, (\mathsf{skip}, s_1), (e_2, e_2), s_2) \rangle \rightarrow^* \langle \sigma', \mathsf{skip} \rangle$. The derivation is obtained by applying LoopExp2F, then one or more times CtxExp, zero or more times CtxStmt, then rule LoopExp1F and zero or more times rule CtxStmt. By considering the hypothesis of the applications of rule CtxExp we have that $\langle \sigma, e_1 \rangle \rightarrow^+ \langle \sigma, v_1 \rangle$. By considering the hypothesis of the applications of rule CtxStmt (the first time) we have that $\langle \sigma, s_2 \rangle \rightarrow^* \langle \sigma'', \mathsf{skip} \rangle$ and (the second time) $\langle \sigma'', s_1 \rangle \rightarrow^* \langle \sigma''', \mathsf{skip} \rangle$. Then by inductive hypothesis we have $\sigma \vdash s_2 \Downarrow \sigma''$ and $\sigma'' \vdash s_1 \Downarrow \sigma'''$. Then by applying the extended inductive hypothesis to the second part of the computation we know that $\sigma''' \vdash (e_1, s_1, e_2, s_2) \Downarrow \sigma'$. We can finally apply rule LoopRec to derive $\sigma \vdash (e_1, s_1, e_2, s_2) \Downarrow \sigma'$, hence the thesis follows.

$((\bullet, e_1), (\mathsf{skip}, s_1), e_2, (\mathsf{skip}, s_2))$**:** the rule never matches our extended inductive hypothesis, hence there is nothing to prove.

CtxStmt here we do a case analysis on the context:

- s**:** we have that $\langle \sigma, s_1\ s_2 \rangle \rightarrow \langle \sigma''', s_1'\ s_2 \rangle \rightarrow^* \langle \sigma'', s_2 \rangle \rightarrow^* \langle \sigma', \mathsf{skip} \rangle$. By inductive hypothesis we have $\sigma \vdash s_1 \Downarrow \sigma''$ and $\sigma'' \vdash s_2 \Downarrow \sigma'$. Then the thesis follows using rule Seq.

$\mathsf{if}\ v_1\ \mathsf{then}\ \bullet\ \mathsf{else}\ s_2\ \mathsf{fi}\ e_2$ where $\mathsf{is\_true?}(v_1)$**:** analogous to the second part of the case for rule CtxExp with context equal to $\mathsf{if}\ \bullet\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi}\ e_2$ (the true subcase).

$\mathsf{if}\ v_1\ \mathsf{then}\ s_1\ \mathsf{else}\ \bullet\ \mathsf{fi}\ e_2$ where $\mathsf{is\_false?}(v_1)$**:** analogous to the previous case.

$(e_1, (\bullet, s_1), (e_2, e_2), s_2)$**:** the only possible match between the conclusion of the rule and the extended inductive hypothesis is when $s = \mathsf{skip}$, but in this case the premise of the rule is always false (skip cannot take any step), hence there is nothing to prove.

$((e_1, e_1), (\mathsf{skip}, s_1), e_2, (\bullet, s_2))$**:** the rule never matches our extended inductive hypothesis, hence there is nothing to prove.

AssVarS: there is nothing to prove since the statement in the premise contains values.

AssArrS: as above, there is nothing to prove since the statement in the premise contains values.

CallS: we have $\langle \sigma, \mathsf{call}\ id \rangle \rightarrow \langle \sigma, \Gamma(id) \rangle$. By inductive hypothesis we have $\sigma \vdash \Gamma(id) \Downarrow \sigma'$. We can derive the conclusion using rule Call.

UnCallS: we have $\langle \sigma, \mathsf{uncall}\ id \rangle \rightarrow \langle \sigma, \mathcal{I}[\![\Gamma(id)]\!] \rangle$. By inductive hypothesis we have $\sigma \vdash \mathcal{I}[\![\Gamma(id)]\!] \Downarrow \sigma'$, hence we can derive the conclusion using rule UnCall.

IfTrueEnd: there is nothing to prove since the statement in the premise contains values.

IfFalseEnd: analogous to IfTrueEnd.

LoopMainS: we have that $\langle \sigma, \mathsf{from}\ e_1\ \mathsf{do}\ s_1\ \mathsf{loop}\ s_2\ \mathsf{until}\ e_2 \rangle \rightarrow \langle \sigma, ((e_1, e_1), s_1, e_2, s_2) \rangle \rightarrow^* \langle \sigma', \mathsf{skip} \rangle$. By rule inspection we know that the derivation of $\langle \sigma, ((e_1, e_1), s_1, e_2, s_2) \rangle \rightarrow^* \langle \sigma', \mathsf{skip} \rangle$ has the structure $\langle \sigma, ((e_1, e_1), s_1, e_2, s_2) \rangle \rightarrow^+ \langle \sigma, ((v_1, e_1), s_1, e_2, s_2) \rangle \rightarrow \langle \sigma, (e_1, (s_1, s_1), (e_2, e_2), s_2) \rangle \rightarrow^* \langle \sigma'', (e_1, (\mathsf{skip}, s_1), (e_2, e_2), s_2) \rangle \rightarrow^* \langle \sigma', \mathsf{skip} \rangle$ where the steps in the first part are derived using rule Ctx-Exp. Hence by taking the premises we know that there is a derivation $\langle \sigma, e_1 \rangle \rightarrow^+ \langle \sigma, v_1 \rangle$. Instead, the steps in the second part are derived using first rule LoopExp1T and then rule CtxStmt zero or more times. Hence by taking the premises we know that there is a derivation $\langle \sigma, s_1 \rangle \rightarrow^* \langle \sigma'', \mathsf{skip} \rangle$. By applying the inductive hypothesis we know that $\sigma \vdash s_1 \Downarrow \sigma''$. By applying the extended inductive hypothesis to the second part of the derivation we know that $\sigma'' \vdash (e_1, s_1, e_2, s_2) \Downarrow \sigma'$. Then we can apply rule LoopMain and the thesis follows.

LoopEndS: the rule never matches our extended inductive hypothesis, hence there is nothing to prove.

SeqS: we have that $\langle \sigma, \mathsf{skip}\ s_2 \rangle \rightarrow \langle \sigma, s_2 \rangle \rightarrow^* \langle \sigma', \mathsf{skip} \rangle$, then using rule Skip we have $\sigma \vdash \mathsf{skip} \Downarrow \sigma$ and by inductive hypothesis we have $\sigma \vdash s_2 \Downarrow \sigma'$. Then the thesis follows using rule Seq. □

4 Examples

We now present a few examples of use of our semantics, to clarify its use and features.

Example 1 (Iterative Fibonacci Computation). We show below the Janus code of the iterative computation of the Fibonacci function.

```
procedure main
  n+ = 4
  x1+ = 1
  x1+ = 1
  call fib
```

```
procedure fib
  from x1 = x2 do
     x1+ = x2
     x1 <=> x2
  loop
     n- = 1
  until n = 0
```

In the code, $x_1 <=> x_2$ denotes the swap of the values of variables x_1 and x_2. Extending our approach to cope with it is trivial, otherwise it can be translated as the sequence of assignments $x_1\hat{} = x_2 \;\; x_2\hat{} = x_1 \;\; x_1\hat{} = x_2$.

$$
\begin{array}{rc}
\ldots \rightarrow & \langle \sigma, \mathsf{call}\ fib(x_1, x_2, n)\rangle \\
\textsc{CallS} \rightarrow & \langle \sigma, \mathsf{from}\ x_1 = x_2\ \mathsf{do}\ x_1 {+}= x_2\ x_1 \mathrel{<}{=}\mathrel{>} x_2\ \mathsf{loop}\ n {-}= 1\ \mathsf{until}\ n = 0\rangle \\
\textsc{LoopMainS} \rightarrow & \langle \sigma, ((x_1 = x_2, x_1 = x_2), \begin{smallmatrix} x_1 {+}= x_2 \\ x_1 \mathrel{<}{=}\mathrel{>} x_2 \end{smallmatrix}, n = 0, n {-}= 1)\rangle \\
\textsc{CtxExp} \rightarrow^* & \langle \sigma, ((\mathsf{true}, x_1 = x_2), \begin{smallmatrix} x_1 {+}= x_2 \\ x_1 \mathrel{<}{=}\mathrel{>} x_2 \end{smallmatrix}, n = 0, n {-}= 1)\rangle \\
\textsc{LoopExp1T} \rightarrow & \langle \sigma, (x_1 = x_2, (\begin{smallmatrix} x_1 {+}= x_2 \\ x_1 \mathrel{<}{=}\mathrel{>} x_2 \end{smallmatrix}, \begin{smallmatrix} x_1 {+}= x_2 \\ x_1 \mathrel{<}{=}\mathrel{>} x_2 \end{smallmatrix}), (n = 0, n = 0), n {-}= 1)\rangle \\
\textsc{CtxStmt} \rightarrow^* & \langle \sigma', (x_1 = x_2, (\mathsf{skip}, \begin{smallmatrix} x_1 {+}= x_2 \\ x_1 \mathrel{<}{=}\mathrel{>} x_2 \end{smallmatrix}), (n = 0, n = 0), n {-}= 1)\rangle \\
\textsc{CtxExp} \rightarrow^* & \langle \sigma', (x_1 = x_2, (\mathsf{skip}, \begin{smallmatrix} x_1 {+}= x_2 \\ x_1 \mathrel{<}{=}\mathrel{>} x_2 \end{smallmatrix}), (\mathsf{false}, n = 0), n {-}= 1)\rangle \\
\textsc{LoopExp2F} \rightarrow & \langle \sigma', ((x_1 = x_2, x_1 = x_2), (\mathsf{skip}, \begin{smallmatrix} x_1 {+}= x_2 \\ x_1 \mathrel{<}{=}\mathrel{>} x_2 \end{smallmatrix}), n = 0, (n {-}= 1, n {-}= 1))\rangle \\
\textsc{CtxStmt} \rightarrow^* & \langle \sigma'', ((x_1 = x_2, x_1 = x_2), (\mathsf{skip}, \begin{smallmatrix} x_1 {+}= x_2 \\ x_1 \mathrel{<}{=}\mathrel{>} x_2 \end{smallmatrix}), n = 0, (\mathsf{skip}, n {-}= 1))\rangle \\
\textsc{CtxExp} \rightarrow^* & \langle \sigma'', ((\mathsf{false}, x_1 = x_2), (\mathsf{skip}, \begin{smallmatrix} x_1 {+}= x_2 \\ x_1 \mathrel{<}{=}\mathrel{>} x_2 \end{smallmatrix}), n = 0, (\mathsf{skip}, n {-}= 1))\rangle \\
\textsc{LoopExp1F} \rightarrow & \langle \sigma'', (x_1 = x_2, (\begin{smallmatrix} x_1 {+}= x_2 \\ x_1 \mathrel{<}{=}\mathrel{>} x_2 \end{smallmatrix}, \begin{smallmatrix} x_1 {+}= x_2 \\ x_1 \mathrel{<}{=}\mathrel{>} x_2 \end{smallmatrix}), (n = 0, n = 0), n {-}= 1)\rangle \\
\cdots \rightarrow^* \cdots \rightarrow^* & \langle \sigma^m, (x_1 = x_2, (\mathsf{skip}, \begin{smallmatrix} x_1 {+}= x_2 \\ x_1 \mathrel{<}{=}\mathrel{>} x_2 \end{smallmatrix}), (\mathsf{true}, n = 0), n {-}= 1)\rangle \\
\textsc{LoopEndS} \rightarrow & \langle \sigma^m, \mathsf{skip}\rangle
\end{array}
$$

Fig. 11. Iterative Fibonacci with the small-step semantics (Color figure online)

We now show the evaluation of the code above, starting from the call statement, using both the small-step (Fig. 11) and the big-step (Fig. 13) semantics. Hence, both the evaluations start with the store $\sigma = \{n \mapsto 4; x_1 \mapsto 1; x_2 \mapsto 1\}$.

Clearly, the small-step semantics takes the form of a sequence of steps, while the big-step semantics gives rise to a single derivation. Notably, the intermediate stores are the same in both the cases. We report them here: $\sigma = \{n \mapsto 4; x_1 \mapsto 1; x_2 \mapsto 1\}$, $\sigma' = \{n \mapsto 4; x_1 \mapsto 2; x_2 \mapsto 1\}$, $\sigma'' = \{n \mapsto 4; x_1 \mapsto 1; x_2 \mapsto 2\}$, $\sigma^m = \{n \mapsto 0; x_1 \mapsto 8; x_2 \mapsto 13\}$.

We show in Fig. 12 as an example the derivation of the first step in the sequence denoted with a red $\rightarrow^*$ in Fig. 11 (small-step computation). ◊

Example 2 (Failing and non-terminating computations). As mentioned in the Introduction, the small-step semantics allows one to better distinguish between failing and non-terminating computations. We show this on the two sample programs below, that respectively exhibit a failing computation (on the left) and a non-terminating one (on the right).

```
procedure main
  from x1 = 10 do
    x2+ = x1
  loop
    x1- = 1
  until x1 = 0
```

```
procedure main
  from x1 = 0 do
    x1+ = 1
  loop
    x2+ = x1
  until x1 = 0
```

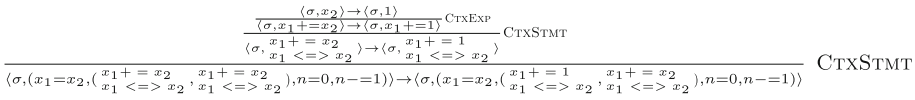

Fig. 12. Derivation of the red step in Fig. 11

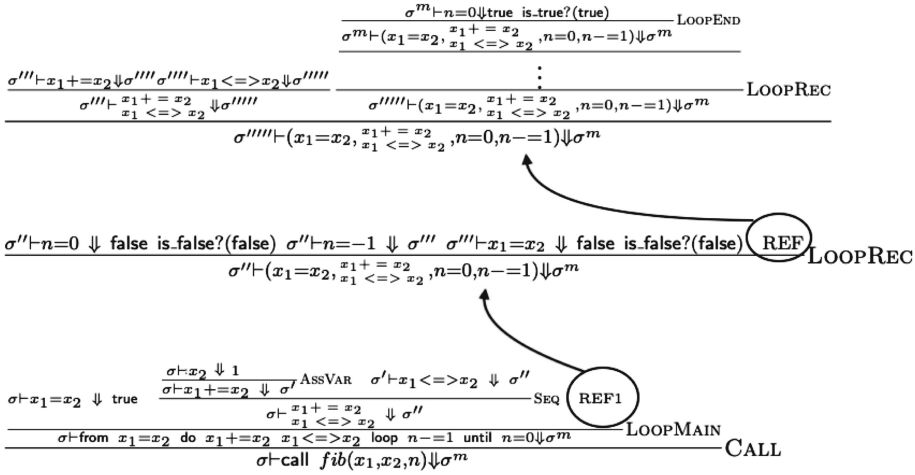

Fig. 13. Iterative Fibonacci with the big-step semantics

In both the cases, σ before the evaluation of the loop is $\sigma = \{x_1 \mapsto 0; x_2 \mapsto 0\}$. For the non-terminating computation, we assume (differently from [20]) that integer values are unbounded. The failing computation raises a runtime error due to the entry guard of the loop being false.

In both the cases, under the big-step semantics, no derivation exists. For the failing computation, we show below how such a derivation should terminate, but there is no applicable rule.

$$\frac{\sigma \vdash x_1 = 10 \ \Downarrow \ \mathsf{false} \ \ \mathsf{is\_false?(false)} \quad \ldots}{\sigma \vdash \mathsf{from}\ x_1 = 10\ \mathsf{do}\ x_1 {+}{=}\ 1\ \mathsf{loop}\ x_2 {+}{=}\ 1\ \mathsf{until}\ x_1 = 0 \ \Downarrow \ ???}\ ???$$

For the non-terminating computation, the derivation is infinite hence one is not able to close it. A fragment of the infinite derivation is in Fig. 14. Using the small-step semantics, instead, on the failing computation we have a computation ending with the error state $\perp$:

$$\frac{\mathsf{is\_false?(false)}}{\langle \sigma, ((\mathsf{false}, x_1 = 10), x_1 {+}{=}\ 1, x_1 = 0, x_2 {+}{=}\ 1)\rangle \rightarrow \perp}\textsc{LoopError1}$$

In the non-terminating computation instead, we can derive an infinite amount of steps, that makes it visible how the computation evolves, as we can see in Fig. 15.

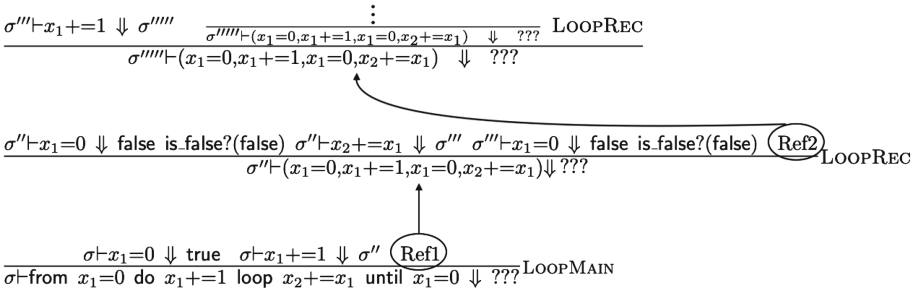

Fig. 14. Fragment of non-terminating derivation in big-step semantics

5 Related Work, Conclusion and Future Work

We proposed a small-step semantics for Janus, an imperative reversible programming language, and established its equivalence to existing big-step semantics. The Janus literature also covers extensions of Janus with other constructs, such as local variables [17,18] and stacks [17,18]. As immediate next step, we will extend our approach to cover such constructs as well. Beyond Janus, the literature on pure reversible programming languages includes also object-oriented languages such as ROOPL [4,15] and Joule [13], and functional languages such as RFun [19] and CoreFun [5]. They are all equipped with big-step semantics. Hence, as future work it makes sense to apply our approach to define small-step semantics for such languages as well.

The only small-step reversible semantics we are aware of are in the context of domain-specific languages for specifying assembly sequences in industrial robots [14,16]. However, the semantics of these languages is quite different from the one of Janus or functional or object-oriented reversible languages, since it involves the position and actions of the robot in the real world. Hence the relation with our work is quite limited. Also, they do not discuss big-step semantics, hence the issue of relating big-step and small-step semantics never emerges. As already mentioned in the Introduction, another promising area for future exploration is the integration of concurrency support into Janus. By enabling the creation of processes and enabling them to exchange messages, such an extension could allow one to write concurrent reversible programs. The definition of a small-step semantics provided in this paper is a relevant first step in this direction.

$$
\begin{aligned}
\cdots \rightarrow \quad & \langle \sigma, \mathsf{from}\ x_1 = 0\ \mathsf{do}\ x_1 {+}{=} 1\ \mathsf{loop}\ x_2 {+}{=} x_1\ \mathsf{until}\ x_1 = 0 \rangle \\
\textsc{LoopMainS} \rightarrow \quad & \langle \sigma, ((x_1 = 0, x_1 = 0), x_1 {+}{=} 1, x_1 = 0, x_2 {+}{=} x_1) \rangle \\
\textsc{CtxExp} \rightarrow^* \quad & \langle \sigma, ((\mathsf{true}, x_1 = 0), x_1 {+}{=} 1, x_1 = 0, x_2 {+}{=} x_1) \rangle \\
\textsc{LoopExp1T} \rightarrow \quad & \langle \sigma, (x_1 = 0, (x_1 {+}{=} 1, x_1 {+}{=} 1), (x_1 = 0, x_1 = 0), x_2 {+}{=} x_1) \rangle \\
\textsc{CtxStmt} \rightarrow^* \quad & \langle \sigma', (x_1 = 0, (\mathsf{skip}, x_1 {+}{=} 1), (x_1 = 0, x_1 = 0), x_2 {+}{=} x_1) \rangle \\
\textsc{CtxExp} \rightarrow^* \quad & \langle \sigma', (x_1 = 0, (\mathsf{skip}, x_1 {+}{=} 1), (\mathsf{false}, x_1 = 0), x_2 {+}{=} x_1) \rangle \\
\textsc{LoopExp2F} \rightarrow \quad & \langle \sigma', ((x_1 = 0, x_1 = 0), (\mathsf{skip}, x_1 {+}{=} 1), x_1 = 0, (x_2 {+}{=} x_1, x_2 {+}{=} x_1)) \rangle \\
\textsc{CtxStmt} \rightarrow^* \quad & \langle \sigma'', ((x_1 = 0, x_1 = 0), (\mathsf{skip}, x_1 {+}{=} 1), x_1 = 0, (\mathsf{skip}, x_2 {+}{=} x_1)) \rangle \\
\textsc{CtxExp} \rightarrow^* \quad & \langle \sigma'', ((\mathsf{false}, x_1 = 0), (\mathsf{skip}, x_1 {+}{=} 1), x_1 = 0, (\mathsf{skip}, x_2 {+}{=} x_1)) \rangle \\
\textsc{LoopExp1F} \rightarrow \quad & \langle \sigma'', (x_1 = 0, (x_1 {+}{=} 1, x_1 {+}{=} 1), (x_1 = 0, x_1 = 0), x_2 {+}{=} x_1) \rangle \\
\rightarrow \quad & \cdots
\end{aligned}
$$

where: $\sigma = \{x_1 \mapsto 0; x_2 \mapsto 0\}$, $\sigma' = \{x_1 \mapsto 1; x_2 \mapsto 0\}$, $\sigma'' = \{x_1 \mapsto 1; x_2 \mapsto 1\}$

Fig. 15. Non-terminating computation in the small-step semantics

References

1. Cristescu, I., Krivine, J., Varacca, D.: A compositional semantics for the reversible pi-calculus. In: LICS, pp. 388–397. IEEE (2013)
2. Dagnino, F.: A meta-theory for big-step semantics. ACM Trans. Comput. Log. **23**(3), 20:1–20:50 (2022)
3. Danos, V., Krivine, J.: Reversible communicating systems. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 292–307. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-28644-8_19
4. Haulund, T.: Design and implementation of a reversible object-oriented programming language (2017)
5. Jacobsen, P.A.H., Kaarsgaard, R., Thomsen, M.K.: CoreFun: a typed functional reversible core language. In: Kari, J., Ulidowski, I. (eds.) RC 2018. LNCS, vol. 11106, pp. 304–321. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99498-7_21
6. Lanese, I., Mezzina, C.A., Stefani, J.: Reversibility in the higher-order π-calculus. Theor. Comput. Sci. **625**, 25–84 (2016)
7. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: A theory of reversibility for Erlang. J. Log. Algebraic Methods Program. **100**, 71–97 (2018)
8. Lutz, C., Derby, H.: Janus: a time-reversible language. Letter to R. Landauer **2** (1986)
9. Milner, R.: Communication and Concurrency, vol. 84. Prentice Hall Englewood Cliffs (1989)
10. Milner, R.: Communicating and Mobile Systems: The Pi Calculus. Cambridge University Press (1999)
11. Owens, S., Myreen, M.O., Kumar, R., Tan, Y.K.: Functional big-step semantics. In: Thiemann, P. (ed.) ESOP 2016. LNCS, vol. 9632, pp. 589–615. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49498-1_23
12. Phillips, I., Ulidowski, I.: Reversing algebraic process calculi. J. Log. Algebr. Program. **73**(1–2), 70–96 (2007)
13. Schultz, U.P.: Reversible object-oriented programming with region-based memory management. In: Kari, J., Ulidowski, I. (eds.) RC 2018. LNCS, vol. 11106, pp. 322–328. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99498-7_22

14. Schultz, U.P.: Reversible control of robots. In: Ulidowski, I., Lanese, I., Schultz, U.P., Ferreira, C. (eds.) RC 2020. LNCS, vol. 12070, pp. 177–186. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-47361-7_8
15. Schultz, U.P., Axelsen, H.B.: Elements of a reversible object-oriented language. In: Devitt, S., Lanese, I. (eds.) RC 2016. LNCS, vol. 9720, pp. 153–159. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40578-0_10
16. Schultz, U.P., Laursen, J.S., Ellekilde, L.-P., Axelsen, H.B.: Towards a domain-specific language for reversible assembly sequences. In: Krivine, J., Stefani, J.-B. (eds.) RC 2015. LNCS, vol. 9138, pp. 111–126. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-20860-2_7
17. Yokoyama, T.: Reversible computation and reversible programming languages. Electron. Notes Theor. Comput. Sci. **253**(6), 71–81 (2010)
18. Yokoyama, T., Axelsen, H.B., Glück, R.: Principles of a reversible programming language. In: Proceedings of the 5th Conference on Computing Frontiers, CF 2008, pp. 43–54. ACM (2008)
19. Yokoyama, T., Axelsen, H.B., Glück, R.: Towards a reversible functional language. In: De Vos, A., Wille, R. (eds.) RC 2011. LNCS, vol. 7165, pp. 14–29. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29517-1_2
20. Yokoyama, T., Glück, R.: A reversible programming language and its invertible self-interpreter. In: PEPM, pp. 144–153. ACM Press (2007)