A Certified Study of a Reversible Programming Language*

Luca Paolini[1], Mauro Piccolo[2], and Luca Roversi[3]

1 **Dipartimento di Informatica, Università degli Studi di Torino, corso Svizzera 185, 10149 Torino, Italy**
lpaolini@unito.it
1 **Dipartimento di Informatica, Università degli Studi di Torino, corso Svizzera 185, 10149 Torino, Italy**
mrpiccol@gmail.com
1 **Dipartimento di Informatica, Università degli Studi di Torino, corso Svizzera 185, 10149 Torino, Italy**
lroversi@unito.it

Abstract

We advance in the study of the semantics of Janus, a C-like reversible programming language. Our study makes utterly explicit some backward and forward evaluation symmetries. We want to deepen mathematical knowledge about the foundations and design principles of reversible computing and programming languages. We formalize a big-step operational semantics and a denotational semantics of Janus. We show a full abstraction result between the operational and denotational semantics. Last, we certify our results by means of the proof assistant Matita.

1998 ACM Subject Classification F.1.2 Computation by Abstract Devices: Modes of Computation, F.3.2 Logics and Meanings of Programs: Semantics of Programming Languages

Keywords and phrases reversible computing, Janus, operational semantics, bi-deterministic evaluation, categorical semantics

Digital Object Identifier 10.4230/LIPIcs.TYPES.2015.7

1 Introduction

Reversible computing is an alternative form of computing: the isentropic core of classical computing [14] and quantum computing [21]. A classic computation is (forward-)deterministic, i.e. each state is followed by a unique state. The reversible computation is a classic computation which is also *backward*-deterministic: every state has a unique predecessor state. Research issues on this subject have emerged in a plethora of situations: isentropic digital circuits, conservative logic, computability and computational complexity, program transformation and software verification, view-update problem, unconventional computing models (bio, quantum, etc.), parallel computing and synchronization, processor architecture, debugging systems and other general backtrack-based settings. Perumalla [15] has recently surveyed many reversible computing facets.

Our focus is on the semantics of reversible programming languages.

These languages fully preserve information inherent in the input of their programs and they allow some form of built-in program inversion for free. Program inversion is the concrete counterpart that forward and backward deterministic computations let available

* Partially supported by the LINTEL project.

to the programmer. Each computation step can move forth/back between a unique pair of predecessor/successor states. Therefore, in these languages the backward execution of a program can result in an efficient search/backtrack-free process. We remark that, despite such a restricted-looking form of computation, reversible programming languages exist which can simulate every function which is computable in classical sense.

The subject of this work is Janus which Lutz and Derby conceived as a student project in 1982 at Caltech [10]. Janus is the first imperative structured programming language, explicitly supporting reversible computing. Yokoyama Glück present and study the language in [20]. Extensions, mainly addressed to introduce built-in programming facilities, are in [19, 18] by Yokoyama, Axelsen, and Glück who also formalize an operational semantics in [20] and improve it in [19, 18].

The operational semantics in [19, 18] does not naturally incorporate an efficient implementation of the reversible aspects which are specific to Janus. This is why we provide an all-round certified treatment of denotational and operational semantics for it.

Concerning the operational side, we define a fully self-contained *big-step* operational evaluation that formalizes a relation on terms which is injective. Our big-step operational semantics explicitly completes the one in [20] by formalizing the *"un-call of a procedure"* as a fully embedded deterministic process. The reason to make the "un-call" explicit is to put the backward and the forward computational directions at the same level, which sounds coherent with the spirit of the reversible computation.

Concerning the denotational side, we interpret the statements and the programs of Janus as functional injective relations. The interpretation composes suitable reversible categorical combinators which belong to **Pinj**, the category of sets and functional injective relations. The advantage of this approach is twofold. The correctness of the interpretation follows directly from composing combinators which we already know they are reversible. By the way, this addresses the possibility of synthesizing a combinatorial reversible language along the lines of James and Sabry [9] that we could use as a target language for compiling Janus. Furthermore, we prove that our denotational interpretation is fully abstract with respect to the operational semantics, entailing that operational and denotational equivalences coincide.

Finally, we certify our results by means of Matita [1]. Matita is an interactive theorem prover based on the Calculus of (Co)Inductive Constructions (CIC) — a dependent type theory. At the proof term level, Matita's proofs are compatible with those ones the theorem prover Coq [7] is based on. The certification is obtained by defining an abstract framework for imperative reversible languages. This abstract structure provides sufficient conditions to model classes of denotational and operational semantics for imperative languages which are reversible. It turns out that both the denotational and the operational semantics of Janus are possible instances of the framework. Getting both the operational and the denotational semantics as a instances of a general framework naturally allows us to fill some of the gaps that [20, 19, 18] leave open about the semantics of Janus. The formalization is available on-line [13].

2 Janus, a Reversible Programming Language

We introduce a minor variant of Janus, starting from [20], while we neglect some extensions which [19] provides. Specifically, we extend ground constants to all natural numbers unlike in [20] which limits them to 32-bit non-negative integer ranging from 0 to $2^{32} - 1$.

▶ **Definition 1** (The syntax of Janus)**.** The grammar which generates *expressions*, *programs* and *statements* of the dialect of Janus we focus on is:

$$\begin{array}{lcl} e & ::= & \mathtt{n} \mid x \mid e + e \mid e - e \mid e * e \mid e/e \mid e\%e \mid e <= e \mid e \mathrel{!}= e \mid e == e \\ \mathtt{P} & ::= & (\mathtt{procedure}\ id\ s)(\mathtt{procedure}\ id\ s)^* \\ s & ::= & x \mathrel{+}= e \mid x \mathrel{-}= e \mid \mathtt{call}\ id \mid \mathtt{uncall}\ id \mid \mathtt{skip} \mid s\ s \\ & & \mid \mathtt{if}\ e\ \mathtt{then}\ s\ \mathtt{else}\ s\ \mathtt{fi}\ e \mid \mathtt{from}\ e\ \mathtt{do}\ s\ \mathtt{loop}\ s\ \mathtt{until}\ e \quad . \end{array}$$

An expression e is either a numeral $\mathtt{n}$ (tacitly confused with a natural number in $\mathbb{N}$) a variable x or the application of a binary operator to sub-expressions. Arithmetic operators are $+, -, *, /, \%$. Relational ones are $<=, !=, ==$. Under a standard convention, they return zero meaning `false`, and a number different from zero which stands for `true`. Binary operators are not necessarily reversible.

A program `P` consists of a list of *procedure declaration*. The keyword `procedure` starts a procedure declaration. A procedure identifier id, or procedure name, follows it. The declaration completes by means of a statement s, i.e. the procedure body.

A statement s is one among a *reversible assignment*, a procedure call, a procedure un-call, a skip, a statement sequence, a reversible conditional, or a reversible loop. In both reversible assignments $x \mathrel{+}= e$ and $x \mathrel{-}= e$ we ask that there are no free occurrences of the variable x in the expression e. All variables are global to the whole program as in [19] (we just omit their declaration at the beginning of programs).

We provide a formalization of the programming language Janus in the proof assistant Matita [1]. The formalization is available at [13]. There, we first provide a parametric syntax of Janus, where the choice of values, unary and binary operator used in expression and assignments and their behaviour is not fixed (file `janus.ma`). Then we show that all the properties of the operational semantics (like reversibility) hold for all possible instances of the parameters. Finally we provide the concrete instance of Janus being introduced by the following definition (file `concrjanus.ma`).

We now describe the semantics of Janus, informally. We informally recall the semantics of Janus following [20, 19, 18]. We remark that in informal discussion we confuse booleans and numbers used as truth-values (0 represents false while other numbers represent true). The leftmost diagram in Figure 1 helps understanding the work-flow of a reversible conditional $\mathtt{if}\ e\ \mathtt{then}\ s\ \mathtt{else}\ s\ \mathtt{fi}\ e$. Two branches are available both in forward direction, from left to right, and in the backward one, from right to left. Let us assume we are interpreting it forwardly, entering e_1 from its left. First we evaluate e_1. If it is "`true`", i.e. if e_1 yields a non zero value, we execute s_1. Otherwise, we execute s_2. No matter which between s_1 and s_2 we have executed, the interpretation proceeds as expected, i.e. it exits from e_2, only if e_2 yields the correct value. Such a value *must be* $\neq 0$ if we just came from s_1 and *must be* 0 if we just came from s_2. Every remaining combination results in a "system abort". The interpretation of Figure 1(a) from right to left does not abort only if, conversely, either both e_2 and e_1 yield "`true`" with s_1 executed as intermediate step, or either both e_2 and e_1 yield "`false`" with s_2 executed in-between them.

The rightmost diagram in Figure 1 helps understanding the work-flow of a reversible loop $\mathtt{from}\ e_1\ \mathtt{do}\ s_1\ \mathtt{loop}\ s_2\ \mathtt{until}\ e_2$. Let us assume we are interpreting it forwardly, trying to enter e_1 from its left. If e_1 gives "`false`", then we stress that the computation aborts. Otherwise, entering the loop means executing s_1 and evaluating e_2. As soon as e_2 results in a value $\neq 0$, the interpreter exits the loop. Otherwise the interpreter executes s_2. However, it keeps looping only if the value of e_1 is "`false`" just after the execution of s_2. On the

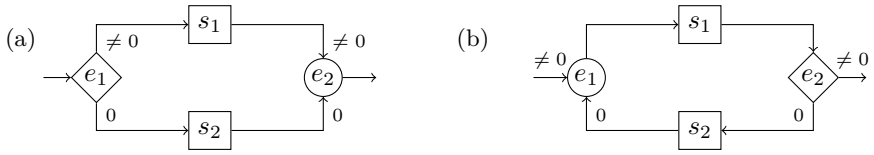


Figure 1 Work-flows of Reversible Conditional (a) and Reversible Loop (b).

contrary, getting a value $\neq 0$ from e_1, would result in a "system abort". The interpretation of Figure 1(b) from right to left is perfectly symmetric. The correct flow is: e_2 evaluates to "`true`", s_1 operates on the state of the program, e_1 evaluates to "`false`", s_2 operates on the state of the program, e_2 evaluates to "`false`" ... and so on until either we exit the loop, or we abort. Exiting is a consequence of evaluating e_1 to "`true`".

Calling and *un-calling* procedures execute the procedures in the right direction and will be the subject of the operational semantics in the coming sections.

We now formally certify the semantics of Janus. A main contribution is the formal certification of the study presented in this paper in Matita. The above definition has been formalized in Matita as an instance of an abstract syntax of Janus presented in the file named `janus.ma`.

In order to define a concrete instance of the abstract language we are referring to, we need to provide a set of constants representing the data being manipulated by the program (const_type), a special constant being the initial value of all variables appearing in the program (init_val), a sort for unary and binary operators (op1_type and op2_type), a sort for operators being used in reversible assignments (rev_type) together with a self-dual function on them used to reverse the variable assignment (rev). We discuss our Matita representation of Janus, in order to drive the reader in the certified formalization reading and understanding.

```
record params : Type[1] :=
{ const_type : DeqSet
; init_val : const_type
; op1_type : Type[0]
; op2_type : Type[0]
; rev_type : Type[0]
; rev : rev_type → rev_type
; idem_prop : ∀x.rev (rev x) = x
}.
```

Both variables and procedure identifiers are implemented as inductive types with one constructor storing a natural number. This number represents the index in which the value of that variable or the body of that procedure can be found in the state.

```
inductive Variable : Type[0] :=
| var : ℕ → Variable.

inductive FunctionName : Type[0] :=
| a_function_id : ℕ→ FunctionName.
```

The set of expressions in Janus are defined in the following way. Expressions that could be variables, values, the application of an operator of arity 1 to an expression, or the application of an operator of arity 2 to two expressions.

```
inductive Expression (p : params) : Type[0] :=
| VAR : Variable → Expression p
| CONST : const_type p → Expression p
| OP_1 : op1_type p → Expression p → Expression p
| OP_2 : op2_type p → Expression p →
Expression p → Expression p.
```

Thus, the set of statements of Janus is implemented as an inductive type with the expected constructors, one for each syntactic construct.

```
inductive stm (p : params) : Type[0] :=
| ASSIGN : rev_type ...p →∀x : Variable.∀ e : Expression p.
(x ∈ (expr_fv ...e)) = false → stm p
| CALL : FunctionName → stm p
| UNCALL : FunctionName → stm p
| SKIP : stm p
| COMP : stm p → stm p → stm p
| IF : Expression p → stm p → stm p →
Expression p → stm p
| LOOP : Expression p → stm p → stm p →
Expression p → stm p.
```

In the previous definition and in some other following ones, we use the Matita syntactical construt ….. We remind that the dots stand here for an arbitrary number of implicit arguments to be guessed by the system.

It turns out that a program is just a list of statements that constitute the bodies corresponding to each procedure identifier.

```
record program (p : params) : Type[0] :=
{  procs :> list  (stm p)
}.
```

The procedure identifier carries a natural number. Thus it provides the index being the position in the list of statement being the body of the considered procedure.

3 Big-Step Operational Semantics

The operational semantics in Figure 2 comes directly from [19, 18]. It defines two relations $\Downarrow_p$ and $\Downarrow_e$ (program and expression evaluations) whose meaning we shall introduce once recalled some notations.

Let $\mathtt{P}$ be a program as in Definition 1. With $\sigma_{\mathtt{P}}$ we denote a state-function from the variables of $\mathtt{P}$ to $\mathbb{N}$. The state-function represents a statically allocated fragment of memory in a hypothetical real implementation. Thus, $\sigma_{\mathtt{P}}(x)$ is the value of x in $\sigma_{\mathtt{P}}$ whenever x is a variable of $\mathtt{P}$. The allocation of the value $\mathtt{n}$ to the variable x in $\sigma_{\mathtt{P}}$ is $\sigma_{\mathtt{P}}[x \mapsto \mathtt{n}]$. Conversely, the restriction of $\sigma_{\mathtt{P}}$ to all its names but x is $\sigma_{\mathtt{P}}\restriction_x$. With $\mathtt{P}(id)$ we identify the body of $\mathtt{P}$ with name id.

Let $\odot$ range over the operators $+, -, *, /, \%$. By $[\![\odot]\!]$ we denote its obvious interpretation. Every clause $\sigma_{\mathtt{P}}, e \Downarrow_e n$ says that the evaluation of the expression e in state $\sigma_{\mathtt{P}}$ yields the result $n \in \mathbb{N}$ using the rules:

$$\frac{}{\sigma_{\mathtt{P}}, \mathtt{n} \Downarrow_e \mathtt{n}}\ \textsc{Con} \qquad \frac{}{\sigma_{\mathtt{P}}, x \Downarrow_e \sigma_{\mathtt{P}}(x)}\ \textsc{Var} \qquad \frac{\sigma_{\mathtt{P}}, e_1 \Downarrow_e \mathtt{n}_1 \quad \sigma_{\mathtt{P}}, e_2 \Downarrow_e \mathtt{n}_2 \quad \mathtt{n}_1 [\![\odot]\!] \mathtt{n}_2 = \mathtt{n}}{\sigma_{\mathtt{P}}, e_1 \odot e_2 \Downarrow_e \mathtt{n}}\ \textsc{BinOp} \quad . \tag{1}$$

$$\frac{\sigma_{\mathtt{P}}\restriction_x, e \Downarrow_e \mathtt{n}_1 \quad \mathtt{n} = \mathtt{n}_0[\![\odot]\!]\mathtt{n}_1 \quad \odot \in \{+,-\}}{\sigma_{\mathtt{P}}[x \mapsto \mathtt{n}_0], x \odot\!= e \Downarrow_p \sigma_{\mathtt{P}}[x \mapsto \mathtt{n}]}\ \textsc{AssVar} \qquad \frac{}{\sigma_{\mathtt{P}}, \mathtt{skip} \Downarrow_p \sigma_{\mathtt{P}}}\ \textsc{Skip}$$

$$\frac{\sigma_{\mathtt{P}}, e_1 \Downarrow_e \mathtt{n}+1 \quad \sigma_{\mathtt{P}}, s_1 \Downarrow_p \sigma'_{\mathtt{P}} \quad \sigma'_{\mathtt{P}}, e_2 \Downarrow_e \mathtt{n}+1}{\sigma_{\mathtt{P}}, \mathtt{if}\ e_1\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2\ \mathtt{fi}\ e_2 \Downarrow_p \sigma'_{\mathtt{P}}}\ \textsc{IfTrue} \qquad \frac{\sigma_{\mathtt{P}}, e_1 \Downarrow_e 0 \quad \sigma_{\mathtt{P}}, s_2 \Downarrow_p \sigma'_{\mathtt{P}} \quad \sigma'_{\mathtt{P}}, e_2 \Downarrow_e 0}{\sigma_{\mathtt{P}}, \mathtt{if}\ e_1\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2\ \mathtt{fi}\ e_2 \Downarrow_p \sigma'_{\mathtt{P}}}\ \textsc{IfFalse}$$

$$\frac{\sigma_{\mathtt{P}}, e_1 \Downarrow_e \mathtt{n}+1 \quad \sigma_{\mathtt{P}}, s_1 \Downarrow_p \sigma'_{\mathtt{P}} \quad \sigma'_{\mathtt{P}}, (e_1, s_1, s_2, e_2) \Downarrow_p \sigma''_{\mathtt{P}}}{\sigma_{\mathtt{P}}, \mathtt{from}\ e_1\ \mathtt{do}\ s_1\ \mathtt{loop}\ s_2\ \mathtt{until}\ e_2 \Downarrow_p \sigma''_{\mathtt{P}}}\ \textsc{LoopMain} \qquad \frac{\sigma_{\mathtt{P}}, e_2 \Downarrow_e \mathtt{n}+1}{\sigma_{\mathtt{P}}, (e_1, s_1, s_2, e_2) \Downarrow_p \sigma_{\mathtt{P}}}\ \textsc{LoopBase}$$

$$\frac{\sigma_{\mathtt{P}}, e_2 \Downarrow_e 0 \quad \sigma_{\mathtt{P}}, s_2 \Downarrow_p \sigma'_{\mathtt{P}} \quad \sigma'_{\mathtt{P}}, e_1 \Downarrow_e 0 \quad \sigma'_{\mathtt{P}}, s_1 \Downarrow_p \sigma''_{\mathtt{P}} \quad \sigma''_{\mathtt{P}}, (e_1, s_1, s_2, e_2) \Downarrow_p \sigma'''_{\mathtt{P}}}{\sigma_{\mathtt{P}}, (e_1, s_1, s_2, e_2) \Downarrow_p \sigma'''_{\mathtt{P}}}\ \textsc{LoopRec}$$

$$\frac{\sigma_{\mathtt{P}}, \mathtt{P}(id) \Downarrow_p \sigma'_{\mathtt{P}}}{\sigma_{\mathtt{P}}, \mathtt{call}\ id \Downarrow_p \sigma'_{\mathtt{P}}}\ \textsc{Call} \qquad \frac{\sigma'_{\mathtt{P}}, \mathtt{P}(id) \Downarrow_p \sigma_{\mathtt{P}}}{\sigma_{\mathtt{P}}, \mathtt{uncall}\ id \Downarrow_p \sigma'_{\mathtt{P}}}\ \textsc{Uncall} \qquad \frac{\sigma_{\mathtt{P}}, s_1 \Downarrow_p \sigma'_{\mathtt{P}} \quad \sigma'_{\mathtt{P}}, s_2 \Downarrow_p \sigma''_{\mathtt{P}}}{\sigma_{\mathtt{P}}, s_1\ s_2 \Downarrow_p \sigma''_{\mathtt{P}}}\ \textsc{Seq}$$

Figure 2 Original operational semantics of Janus.

$$\frac{\sigma_{\mathtt{P}}\restriction_x, e \Downarrow_e \mathtt{n}_1 \quad \mathtt{n} = \mathtt{n}_0[\![+]\!]n_1}{\sigma_{\mathtt{P}}[x \mapsto \mathtt{n}_0], x -\!= e \Downarrow_p^{®} \sigma_{\mathtt{P}}[x \mapsto \mathtt{n}]}\ +^{®} \qquad \frac{\sigma_{\mathtt{P}}\restriction_x, e \Downarrow_e \mathtt{n}_1 \quad \mathtt{n} = \mathtt{n}_0[\![-]\!]\mathtt{n}_1}{\sigma_{\mathtt{P}}[x \mapsto \mathtt{n}_0], x +\!= e \Downarrow_p^{®} \sigma_{\mathtt{P}}[x \mapsto \mathtt{n}]}\ -^{®} \qquad \frac{}{\sigma_{\mathtt{P}}, \mathtt{skip} \Downarrow_p^{®} \sigma_{\mathtt{P}}}\ \textsc{Skip}^{®}$$

$$\frac{\sigma_{\mathtt{P}}, e_2 \Downarrow_e \mathtt{n}+1 \quad \sigma_{\mathtt{P}}, s_1 \Downarrow_p^{®} \sigma'_{\mathtt{P}} \quad \sigma'_{\mathtt{P}}, e_1 \Downarrow_e \mathtt{n}+1}{\sigma_{\mathtt{P}}, \mathtt{if}\ e_1\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2\ \mathtt{fi}\ e_2 \Downarrow_p^{®} \sigma'_{\mathtt{P}}}\ \textsc{IfTrue}^{®} \qquad \frac{\sigma_{\mathtt{P}}, e_2 \Downarrow_e 0 \quad \sigma_{\mathtt{P}}, s_2 \Downarrow_p^{®} \sigma'_{\mathtt{P}} \quad \sigma'_{\mathtt{P}}, e_1 \Downarrow_e 0}{\sigma_{\mathtt{P}}, \mathtt{if}\ e_1\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2\ \mathtt{fi}\ e_2 \Downarrow_p^{®} \sigma'_{\mathtt{P}}}\ \textsc{IfFalse}^{®}$$

$$\frac{\sigma_{\mathtt{P}}, e_2 \Downarrow_e \mathtt{n}+1 \quad \sigma_{\mathtt{P}}, s_1 \Downarrow_p^{®} \sigma'_{\mathtt{P}} \quad \sigma'_{\mathtt{P}}, (e_1, s_1, s_2, e_2) \Downarrow_p^{®} \sigma''_{\mathtt{P}}}{\sigma_{\mathtt{P}}, \mathtt{from}\ e_1\ \mathtt{do}\ s_1\ \mathtt{loop}\ s_2\ \mathtt{until}\ e_2 \Downarrow_p^{®} \sigma''_{\mathtt{P}}}\ \textsc{LoopMain}^{®} \qquad \frac{\sigma_{\mathtt{P}}, e_1 \Downarrow_e^{®} \mathtt{n}+1}{\sigma_{\mathtt{P}}, (e_1, s_1, s_2, e_2) \Downarrow_p \sigma_{\mathtt{P}}}\ \textsc{LoopBase}^{®}$$

$$\frac{\sigma_{\mathtt{P}}, e_1 \Downarrow_e 0 \quad \sigma_{\mathtt{P}}, s_2 \Downarrow_p^{®} \sigma'_{\mathtt{P}} \quad \sigma'_{\mathtt{P}}, e_2 \Downarrow_e 0 \quad \sigma'_{\mathtt{P}}, s_1 \Downarrow_p^{®} \sigma''_{\mathtt{P}} \quad \sigma''_{\mathtt{P}}, (e_1, s_1, s_2, e_2) \Downarrow_p^{®} \sigma'''_{\mathtt{P}}}{\sigma_{\mathtt{P}}, (e_1, s_1, s_2, e_2) \Downarrow_p^{®} \sigma'''_{\mathtt{P}}}\ \textsc{LoopRec}^{®}$$

$$\frac{\sigma_{\mathtt{P}}, \mathtt{P}(id) \Downarrow_p^{®} \sigma'_{\mathtt{P}}}{\sigma_{\mathtt{P}}, \mathtt{call}\ id \Downarrow_p^{®} \sigma'_{\mathtt{P}}}\ \textsc{Call}^{®} \qquad \frac{\sigma_{\mathtt{P}}, \mathtt{P}(id) \Downarrow_p \sigma'_{\mathtt{P}}}{\sigma_{\mathtt{P}}, \mathtt{uncall}\ id \Downarrow_p^{®} \sigma'_{\mathtt{P}}}\ \textsc{coCall}^{®} \qquad \frac{\sigma_{\mathtt{P}}, s_2 \Downarrow_p^{®} \sigma'_{\mathtt{P}} \quad \sigma'_{\mathtt{P}}, s_1 \Downarrow_p^{®} \sigma''_{\mathtt{P}}}{\sigma_{\mathtt{P}}, s_1\ s_2 \Downarrow_p^{®} \sigma''_{\mathtt{P}}}\ \textsc{Seq}^{®}$$

Figure 3 Rules that formalize how the backward interpretation of Janus works.

We remark that no rule here above have side effects on $\sigma_{\mathtt{P}}$.

Concerning $\Downarrow_p$, every clause $\sigma_{\mathtt{P}}, s \Downarrow_p \sigma'_{\mathtt{P}}$ says that the evaluation of the statement s in state $\sigma_{\mathtt{P}}$ yields the state $\sigma'_{\mathtt{P}}$ possibly affected by the side effects of s.

It is usual (albeit not mandatory) to assume that variables are initialized to zero starting the evaluation of a program, i.e. executing the body of the first procedure. The evaluation of $\mathtt{P} = \mathtt{procedure}\ id_1; s_1 \ldots \mathtt{procedure}\ id_n; s_n$ starting from $\sigma_{\mathtt{P}}$ stops whenever $\sigma_{\mathtt{P}}, \mathtt{call}\ id_1 \Downarrow_p \sigma'_{\mathtt{P}}$ holds, for some $\sigma'_{\mathtt{P}}$.

Some remarks on the rules in Figure 2 are worth making and help moving towards our first contribution.

The rules LoopMain, LoopRec and LoopBase introduce the auxiliary syntax (e_1, s_1, s_2, e_2). It separates the interpretation phases of $\mathtt{from}\ e_1\ \mathtt{do}\ s_1\ \mathtt{loop}\ s_2\ \mathtt{until}\ e_2$. LoopMain introduces (e_1, s_1, s_2, e_2) if e_1 evaluates to "$\mathtt{true}$". LoopRec uses (e_1, s_1, s_2, e_2) to keep unfolding the loop only if e_1 and e_2 evaluate to "$\mathtt{false}$". LoopBase concludes the interpretation of the loop.

Last, the rule Uncall in Figure 2 serves to unroll the interpretation of the procedure with name id. The rule is effectively computable but it is non-deterministic and inefficient. Uncall must pick up in the whole set of state-functions, a $\sigma'_{\mathtt{P}}$ such that when evaluating the statement $\mathtt{P}(id)$ we obtain the "input" state $\sigma_{\mathtt{P}}$. Uncall does not provide any explicit guideline on how explicitly and efficiently finding that such a state $\sigma'_{\mathtt{P}}$ exists at the level of big-step semantics.

Our first contribution is the definition of a big-step operational semantics which makes the process of reversing the interpretation of a program both explicit and efficient. The operational semantics we propose contains the union between the set of new rules in Figure 3 and the set of rules in Figure 2 with the proviso of replacing

$$\frac{\sigma_{\mathtt{P}}, \mathtt{P}(id) \Downarrow_p^{\circledR} \sigma'_{\mathtt{P}}}{\sigma_{\mathtt{P}}, \mathtt{uncall}\ id \Downarrow_p \sigma'_{\mathtt{P}}}\ \textsc{coCall} \tag{2}$$

for UNCALL.

An other possible strategy to recover efficiency could be to define a compiler that translates the "un-call" of $\mathtt{P}(id)$ into the equivalent program of Janus that we can interpret with the operational semantics in Figure 2. This is proposed in [20]. The rules in Figure 3 somewhat embed the self-interpreter in [20] directly into the evaluation process. Our completion of the operational semantics greatly improves the understanding of Janus and the possibility to formally reason on it, as we do in Matita.

▶ **Definition 2.** Given a program P, the two mutually defined relations $\Downarrow_p$ and $\Downarrow_p^{\circledR}$ map a pair with a state $\sigma_{\mathtt{P}}$ and a statement s into a state $\sigma'_{\mathtt{P}}$. The relations hold whenever there is a finite derivation of $\sigma_{\mathtt{P}}, s \Downarrow_p \sigma'_{\mathtt{P}}$ or $\sigma_{\mathtt{P}}, s \Downarrow_p^{\circledR} \sigma'_{\mathtt{P}}$, using the rules of Figure 2 with coCALL in place of UNCALL and the rules of Figure 3. The evaluation of P exists if $\sigma_{\mathtt{P}}, \mathtt{P} \Downarrow_p \sigma'_{\mathtt{P}}$, for some $\sigma_{\mathtt{P}}$ and $\sigma'_{\mathtt{P}}$.

Some comments on the rules in Figure 3 are worth making before stating the main properties of the operational semantics we propose. Roughly, those rules interpret programs backwardly.

For instance, SEQ$^{\circledR}$ evaluates $s_1 s_2$ by starting from s_2. The rule LOOPMAIN$^{\circledR}$ mirrors the behavior of LOOPMAIN. It checks whether e_2 evaluates to a value $\neq 0$ and introduces (e_1, s_1, s_2, e_2) to signal that the interpretation of a reversible loop has just started. The choice between LOOPREC$^{\circledR}$ or LOOPBASE$^{\circledR}$ follows from the value that e_1 — not e_2 — produces. Analogously, the interpretation of $\mathtt{if}\ e_1\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2\ \mathtt{fi}\ e_2$ under $\Downarrow_p^{\circledR}$ starts from evaluating e_2. Finally, let us focus on CALL$^{\circledR}$ and coCALL$^{\circledR}$. The first one works like CALL but on the "reversed" procedure. We mean that CALL$^{\circledR}$ executes the procedure from its conclusion, replacing every instruction by means of its "opposite". Instead, coCALL$^{\circledR}$ re-reverses the evaluation "direction" moving again the evaluation control flow to $\Downarrow_p$.

Figure 4 shows the details about what "undoing a procedure-call" means directly inside the new, full blown, big-step operational semantics.

The following Lemma is preliminary to the main properties. If the control-flow enters a loop (cf. Figure 1(b)), then s_1 is executed once and the control moves to the exit-conditional: either the control exits the loop or, s_1, s_2 are both executed (forwardly or backwardly), before to re-check the exit-conditional.

▶ **Lemma 3** (Length of loops and number of states).

1. $\sigma_{\mathtt{P}}, (e_1, s_1, s_2, e_2) \Downarrow_p \sigma'_{\mathtt{P}}$ *if and only if there are* $\sigma^0_{\mathtt{P}}, \ldots, \sigma^{2m}_{\mathtt{P}}$ *where* $m \in \mathbb{N}$ *(*$2m$ *is the double of* m*),* $\sigma^0_{\mathtt{P}} = \sigma_{\mathtt{P}}$*,* $\sigma^{2m}_{\mathtt{P}} = \sigma'_{\mathtt{P}}$*, such that:*
   - $\sigma^{2m}_{\mathtt{P}}, e_2 \Downarrow_e \mathtt{n}+\mathtt{1}$ *and* $\sigma^{2i}_{\mathtt{P}}, e_2 \Downarrow_e \mathtt{0}$*,* $\sigma^{2i}_{\mathtt{P}}, s_2 \Downarrow_p \sigma^{2i+1}_{\mathtt{P}}$ *for* $i < m$*;*
   - $\sigma^{2i+1}_{\mathtt{P}}, e_1 \Downarrow_e \mathtt{0}$ *for* $i < m$*, and* $\sigma^{2i+1}_{\mathtt{P}}, s_1 \Downarrow_p \sigma^{2i+2}_{\mathtt{P}}$ *for* $i \leq m-2$*.*
2. $\sigma_{\mathtt{P}}, (e_1, s_1, s_2, e_2) \Downarrow_p^{\circledR} \sigma'_{\mathtt{P}}$ *if and only if there are* $\sigma^0_{\mathtt{P}}, \ldots, \sigma^{2m}_{\mathtt{P}}$ *where* $m \in \mathbb{N}$*,* $\sigma^0_{\mathtt{P}} = \sigma_{\mathtt{P}}$*,* $\sigma^{2m}_{\mathtt{P}} = \sigma'_{\mathtt{P}}$*, such that:*
   - $\sigma^{2m}_{\mathtt{P}}, e_1 \Downarrow_e \mathtt{n}+\mathtt{1}$ *and* $\sigma^{2i}_{\mathtt{P}}, e_1 \Downarrow_e \mathtt{0}$*,* $\sigma^{2i}_{\mathtt{P}}, s_1 \Downarrow_p^{\circledR} \sigma^{2i+1}_{\mathtt{P}}$ *for* $i < m$*;*
   - $\sigma^{2i+1}_{\mathtt{P}}, e_2 \Downarrow_e \mathtt{0}$ *for* $i < m$*, and* $\sigma^{2i+1}_{\mathtt{P}}, s_2 \Downarrow_p^{\circledR} \sigma^{2i+2}_{\mathtt{P}}$ *for* $i \leq m-2$*.*

Proof. ($\Rightarrow$). By induction on the derivation proving $\sigma_{\mathtt{P}}, (e_1, s_1, s_2, e_2) \Downarrow_p \sigma'_{\mathtt{P}}$.
($\Leftarrow$). By induction on the derivation proving $\sigma_{\mathtt{P}}, (e_1, s_1, s_2, e_2) \Downarrow_p \sigma'_{\mathtt{P}}$. ◀

Forward and backward computation are related in the correct and expected way, as stated by the following Theorem.

▶ **Theorem 4** ($\Downarrow_p$ and $\Downarrow_p^{®}$ annihilate each other). *$\sigma_{\mathtt{P}}, s \Downarrow_p \sigma^*_{\mathtt{P}}$ if and only if $\sigma^*_{\mathtt{P}}, s \Downarrow_p^{®} \sigma_{\mathtt{P}}$.*

Proof. ($\Rightarrow$). The proof is given by induction on s. All cases are straightforward except for loop. Suppose $\sigma_{\mathtt{P}}, \mathtt{from}\ e_1\ \mathtt{do}\ s_1\ \mathtt{loop}\ s_2\ \mathtt{until}\ e_2 \Downarrow_p \sigma^*_{\mathtt{P}}$. Thus $\sigma_{\mathtt{P}}, e_1 \Downarrow_e \mathtt{n+1}$, $\sigma_{\mathtt{P}}, s_1 \Downarrow_p \sigma'_{\mathtt{P}}$ and $\sigma'_{\mathtt{P}}, (e_1, s_1, s_2, e_2) \Downarrow_p \sigma^*_{\mathtt{P}}$. By Lemma 3.1 there are $\sigma^0_{\mathtt{P}}, \ldots, \sigma^{2m}_{\mathtt{P}}$ where $m \in \mathbb{N}$, $\sigma^0_{\mathtt{P}} = \sigma'_{\mathtt{P}}$, $\sigma^{2m}_{\mathtt{P}} = \sigma^*_{\mathtt{P}}$, such that: $\sigma^{2m}_{\mathtt{P}}, e_2 \Downarrow_e \mathtt{n+1}$ and $\sigma^{2i}_{\mathtt{P}}, e_2 \Downarrow_e \mathtt{0}$, $\sigma^{2i}_{\mathtt{P}}, s_2 \Downarrow_p \sigma^{2i+1}_{\mathtt{P}}$ for $i < m$; $\sigma^{2i+1}_{\mathtt{P}}, e_1 \Downarrow_e \mathtt{0}$ for $i < m$, and $\sigma^{2i+1}_{\mathtt{P}}, s_1 \Downarrow_p \sigma^{2i+2}_{\mathtt{P}}$ for $i \leq m-2$. Re-organizing (in reverse order the list of state), we have that the list $\sigma^{2m-1}_{\mathtt{P}}, \ldots, \sigma^0_{\mathtt{P}}, \sigma_{\mathtt{P}}$ satisfies the right-hand side of the statement Lemma 3.2. Thus, $\sigma^{2m-1}_{\mathtt{P}}, (e_1, s_1, s_2, e_2) \Downarrow_p^{®} \sigma_{\mathtt{P}}$. It is easy to conclude $\sigma^*_{\mathtt{P}}, \mathtt{from}\ e_1\ \mathtt{do}\ s_1\ \mathtt{loop}\ s_2\ \mathtt{until}\ e_2 \Downarrow_p^{®} \sigma_{\mathtt{P}}$. ($\Leftarrow$). Dual to the above proof. ◀

The operational semantics we introduce is deterministically syntax driven. Given a statement and chosen which evaluation between $\Downarrow_p$ or $\Downarrow_p^{®}$ to use, we can apply a single rule at every step. The consequence is the following Corollary, whose existence we anticipated in the introduction

▶ **Corollary 5** ($\Downarrow_p$ and $\Downarrow_p^{®}$ are functional and injective). *The relations $\Downarrow_p$ and $\Downarrow_p^{®}$ are functional, i.e.:*
1. *If $\sigma_{\mathtt{P}}, s \Downarrow_p \sigma^1_{\mathtt{P}}$ and $\sigma_{\mathtt{P}}, s \Downarrow_p \sigma^2_{\mathtt{P}}$, then $\sigma^1_{\mathtt{P}} = \sigma^2_{\mathtt{P}}$.*
2. *If $\sigma_{\mathtt{P}}, s \Downarrow_p^{®} \sigma^1_{\mathtt{P}}$ and $\sigma_{\mathtt{P}}, s \Downarrow_p^{®} \sigma^2_{\mathtt{P}}$, then $\sigma^1_{\mathtt{P}} = \sigma^2_{\mathtt{P}}$.*
The relations $\Downarrow_p$ and $\Downarrow_p^{®}$ are injective, i.e.:
1. *If $\sigma^1_{\mathtt{P}}, s \Downarrow_p \sigma_{\mathtt{P}}$ and $\sigma^2_{\mathtt{P}}, s \Downarrow_p \sigma_{\mathtt{P}}$, then $\sigma^1_{\mathtt{P}} = \sigma^2_{\mathtt{P}}$.*
2. *If $\sigma^1_{\mathtt{P}}, s \Downarrow_p^{®} \sigma_{\mathtt{P}}$ and $\sigma^2_{\mathtt{P}}, s \Downarrow_p^{®} \sigma_{\mathtt{P}}$, then $\sigma^1_{\mathtt{P}} = \sigma^2_{\mathtt{P}}$.*

The proofs of Lemma 3 and Theorem 4 are certified in the file `janus.ma`, while the proof of Corollary 5 is formalized in the file `completeness.ma` in [13].

We now give some comments on how we represent and certify the properties of the whole operational semantics in Figure 2 and 3. To this aim we need to comment about how the abstract syntax and its instances work. We start from the evaluation of unary and binary operators. This formalization is in the file `janus.ma` in [13].

```
record sem_params (p : params) : Type[0] :=
{ evaluate_op1 : op1_type …p → const_type …p → option (const_type …p)
; evaluate_op2 : op2_type …p → const_type …p → const_type …p
→ option (const_type …p)
; evaluate_rev : rev_type …p → const_type …p → const_type …p
→ option (const_type …p)
; const_to_bool : const_type …p → bool
; reverse_eval_rev : ∀r,a,b,c.evaluate_rev r a b = return c
→ evaluate_rev (rev …r) c b = return a
}.
```

The behaviour of unary operators is specified by the field evaluate_op1. The behaviour of binary operators is specified by the field evaluate_op2. The evaluation of the binary operators used in reversible assignments is specified by the field evaluate_rev. We require also to specify a canonical projection const_to_bool from constant types to boolean ones, that will be used in the evaluation of the expression guards in conditional and loop statements. Furthermore

we require that the evaluation of a binary operator used in reversible assignment can be reversed by using the specific reverse operator specified by the function specified in the field rev of record params.

All the procedures implementing the evaluation of the above mentioned operators may fail. In order to keep track of this possibility, we use the option monad, which is specified in the standard library of Matita.

```
inductive option (A:Type[0]) : Type[0] :=
None : option A
| Some : A →option A.
```

In the file `concrjanus.ma` of [13] we provide an instance of the above record which correspond to the concrete operational semantics in Figures 2 and 3. Let us see how to instantiate the record sem_params. Since there is no unary operator, we do not provide any implementation. The procedures implementing the behaviour of remaining operators are straightforward by cases. The projection from integers to boolean values is the standard one: zero is `false` and every non-zero value correspond to `true`. Finally a proof of correctness for the specific choice of operators used in reversible assignments is provided.

The specification of the behaviour of all operators involved in the syntax of expressions makes it possible to define how the expressions are evaluated in a given program state.

```
definition syn_state :=λ p : params.list (const_type ...p).

let rec evaluate_expression (p : params) (p' : sem_params p)
(e : Expression p) (st : syn_state p) on e : option (const_type ...p) :=....
```

A state (syn_state) is just a list of constant values. The Matita representation of variables assumes that each variable carries an index identifying uniquely the position where its value is stored. After having provided suitable instances of both the records params and sem_params as specified above the evaluation of an expression e on a given state st is described by the procedure evaluate_expression. Notice that the evaluation of an expression on a given state may fail. When e is a variable, then it is evaluated to its value in the state st (if the index carried by the variable is not beyond the length of the state). The value of a constant is the constant itself. The value of an expression obtained by applying an unary (resp. binary) operator o to a sub-expression(s) e1 (and e2) is equal to the application of the behaviour of that operator to the value of the expression e1 (and the value of e2).

```
let rec fwd_operational_semantics (p : params) (p' : sem_params p)
(env : list (stm p)) (q : stm p) (s1 : syn_state p)
(n : ℕ) on n : option (syn_state p) :=
match n with
[ O ⇒None ?
| S m ⇒match q with [....]
]
and
bwd_operational_semantics (p : params) (p' : sem_params p)
(env : list (stm p)) (q : stm p) (s1 : syn_state p)
(n : ℕ) on n : option (syn_state p) :=
[ O ⇒None ?
| S m ⇒match q with [....]
]
```

$$\left\{\begin{array}{l} x == m \\ y == n \\ z == 0 \\ w == 0 \end{array}\right\} \texttt{call}\ incr; \left\{\begin{array}{l} x == m \\ y == n + m \\ z == m \\ w == 0 \end{array}\right\} \texttt{call}\ copy; \left\{\begin{array}{l} x == m \\ y == n + m \\ z == m \\ w == n + m \end{array}\right\} \texttt{uncall}\ incr; \left\{\begin{array}{l} x == m \\ y == n \\ z == 0 \\ w == n + m \end{array}\right\} \texttt{call}\ exchange; \left\{\begin{array}{l} x == m \\ y == n \\ z == n + m \\ w == 0 \end{array}\right\}$$

Figure 4 The sequence of transformations that `call` *sum* in (3) operates on the initial state.

Since Janus allows both forward and backward execution, we have to provide a proper way to unroll the computation. We defined a fully fledged operational semantics of Janus statements in a given program state. It is realized by two mutually recursive procedures named respectively fwd_operational_semantics and bwd_operational_semantics. The former defines how a statement is executed in forward way while the latter defines how a statement is executed backward way.

Both procedures take in input also a natural number n used as a threshold limit to the number of steps needed by both the forward and backward executors to reach a final state. It follows that the evaluation predicate $\sigma, s \Downarrow_p \sigma'$ can be expressed in Matita by postulating the existence of a number n such that the final state σ' is reached from σ when evaluating s after having settled n as threshold.

We have proven a monotonicity result of both forward and backward execution i.e. if in the evaluation of a statement a final state is reached in at least n steps threshold, then it can again be reached even if the threshold is augmented. Monotonicity entails that the operational semantics is forward deterministic. Backward determinism is guaranteed by the fact that if the forward evaluation of a statement on a given state s1 evaluates to the state s2 then the backward evaluation of the same statement from s2 evaluates to s1. These results hold for every choice of syntactical and semantic parameters, thus they hold also for the concrete language formalized in `concrjanus.ma`, providing in this way the certification of Theorem 4 and Corollary 5.

```
theorem op_sem_reversibility :
∀p : params.∀p' : sem_params p.∀env : list (stm p).
∀q : stm p.∀s1,s2 : syn_state p.∀n : ℕ.
(fwd_operational_semantics p p' env q s1 n = return s2 →
bwd_operational_semantics p p' env q s2 n = return s1) ∧
(bwd_operational_semantics p p' env q s1 n = return s2 →
fwd_operational_semantics p p' env q s2 n = return s1).

theorem operational_semantics_monotone : ∀p : params.∀p' : sem_params p.
∀env : list (stm p).∀q : stm p.∀s1,s2 : syn_state p.∀n,m : ℕ.n≤m →
(fwd_operational_semantics p p' env q s1 n = return s2 →
fwd_operational_semantics p p' env q s1 m = return s2).

theorem bwd_operational_semantics_monotone : ∀p : params.∀p' : sem_params p.
∀env : list (stm p).∀q : stm p.∀s1,s2 : syn_state p.∀n,m : ℕ.n≤m →
(bwd_operational_semantics p p' env q s1 n = return s2 →
bwd_operational_semantics p p' env q s1 m = return s2).
```

3.1 A Running Example

The sum of two natural numbers is computable, but certainly not injective. More precisely, if we say that the result of a sum is 8, we do not know if it results from summing 5 and 3 or 2 and 6. Following Bennett [4], we can program the sum of x and y in Janus as follows:

$$Rsum(x,y,z) = \begin{cases} (x, y, x+y) & \text{if } z = 0, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The function $Rsum : \mathbb{N}^3 \to \mathbb{N}^3$ is reversible because it is defined to preserve the input values. Using the very rich language of expressions of Janus, *RSum* simply can be `from` $z = 0$ `do` $z+ = (x + y)$ `loop skip until` 1. As a curiosity the same function exists in a restriction of the syntax of expression that only contains successor, predecessor and an equality test. Inside the restriction *RSum* is given by the following Janus-program (where we used semicolons and a natural indentation to make explicit the language parsing).

```
procedure main
    call sum;
procedure sum
    call incr; call copy; uncall incr; call exchange;
procedure incr
    from z = 0 do skip loop z += 1; y += 1 until z = x;
procedure copy
    from w = 0 do skip loop w += 1 until w = y;
procedure exchange
    from z = 0 do skip loop z += 1; w− = 1 until w = 0;
```
(3)

The execution of (3) would yield $\sigma_{\mathtt{P}}, \mathtt{P}(sum) \Downarrow \sigma_{\mathtt{P}}[\mathtt{z} \mapsto \sigma_{\mathtt{P}}(\mathtt{x}) + \sigma_{\mathtt{P}}(\mathtt{y})]$ for all $\sigma_{\mathtt{P}}$ such that $\sigma_{\mathtt{P}}(\mathtt{z}) = 0$ and $\sigma_{\mathtt{P}}(\mathtt{w}) = 0$. We have that the following statements hold:

- $\sigma_{\mathtt{P}}, \mathtt{P}(incr) \Downarrow \sigma_{\mathtt{P}}[\mathtt{y} \mapsto \sigma_{\mathtt{P}}(\mathtt{x}) + \sigma_{\mathtt{P}}(\mathtt{y}), \mathtt{z} = \sigma(\mathtt{x})]$, for all $\sigma_{\mathtt{P}}$ such that $\sigma_{\mathtt{P}}(\mathtt{z}) = 0$,
- $\sigma_{\mathtt{P}}, \mathtt{P}(copy) \Downarrow \sigma_{\mathtt{P}}[\mathtt{w} \mapsto \sigma_{\mathtt{P}}(\mathtt{y})]$, for all $\sigma_{\mathtt{P}}$ such that $\sigma_{\mathtt{P}}(\mathtt{w}) = 0$, and
- $\sigma_{\mathtt{P}}, \mathtt{P}(exchange) \Downarrow \sigma_{\mathtt{P}}[\mathtt{z} \mapsto \sigma_{\mathtt{P}}(\mathtt{w}), \mathtt{w} = 0]$, for all $\sigma_{\mathtt{P}}$ such that $\sigma_{\mathtt{P}}(\mathtt{z}) = 0$.

Compactly, the sequence of predicates that describes the states change along the execution of the procedure are depicted in Figure 4.

Notice that the variable w is a variable for which we can predict its final value at the end of computation of the sum, so it could be safely deleted. For this purpose it might be useful to extend the syntax of Janus with the reversible allocation of local variables as in [19]. Let *decr* be defined as `from` $z = x$ `do` $z -= 1; y -= 1$ `loop skip until` $z = 0$. A call *decr* can be used to replace `uncall` *incr*. Symmetrically, a uncall to *decr* be used to replace `call` *incr*.

4 The Model

In this section, we introduce step by step the components that we will need in the next section to define the model of Janus. We are going to interpret Janus statements and programs in terms of relations between states. These relations are both functional and injective. We show a full abstraction between the equality on programs induced by the denotational semantics and the equality induced by the operational semantics. Both the denotational semantics and the full abstraction theorems have been formalized in the proof assistant Matita. The formalization provided in the files `rel.ma`, `pinj.ma`, `rel_interpretation.ma`. The correspondence results are formalized in `correctness.ma`, `completeness.ma` and `compl_thm.ma` in [13].

4.1 A Category of Partial Injective Functions

We denote with **Rel** the category of sets and relations. A relation $r : A \to B$ is *functional* if $(a, b), (a, b') \in r$ implies $b = b'$ for all a, b, b'. A relation $r : A \to B$ is *injective* if $(a, b), (a', b) \in r$ implies $a = a'$ for all a, a', b. **Pinj** is a subcategory of **Rel** whose objects are

sets and whose morphisms are functional injective relations (partial injective function, in other words). We will denote with **Pfn** the subcategory of **Rel** whose objects are sets and whose morphisms are functional relations.

We will denote with $id_A : A \to A$ the identity relation on A and with $\circ$ the relational composition. When $f : A \to B$ is a functional relation and $a \in A$, we sometimes write $f(a)$ to denote the unique $b \in B$ (if it exists) such that $(a, b) \in f$. When $\odot : A \times B \to C$ is a functional relation, we could also use the infix notation $a \odot b$ to denote the unique $c \in C$ (if it exists) such that $((a, b), c) \in \odot$.

Given $r : A \to B$ in **Pinj** we define the *inverse* of r denoted with $r^\dagger$ as $\{(b, a) \mid (a, b) \in r\}$. It is again a morphism in **Pinj**.

Both **Pinj** and **Pfn** admit two symmetric tensor products. They are the *cartesian product* $\times$ (a.k.a. product) and the *disjoint union* $+$ (a.k.a. sum or coproduct). We will denote with **1** the singleton set i.e. the unit of the product, and with **0** the empty set i.e. the unit of the sum. We will denote with

$$\begin{array}{ll} \alpha^\times_{A,B,C} : A \times (B \times C) \to (A \times B) \times C & \alpha^+_{A,B,C} : A + (B + C) \to (A + B) + C \\ \sigma^\times_{A,B} : A \times B \to B \times A & \sigma^+_{A,B} : A + B \to B + A \\ \lambda^\times_A : A \to \mathbf{1} \times A & \lambda^+_A : A \to \mathbf{0} + A \end{array}$$

respectively the associative laws of product and sum, the symmetric laws of product and sum and the left neutral element laws of product and sum: they are the same morphisms in both **Pinj** and **Pfn**. Sometimes, the apexes and the subscripts will be omitted when clear from the context or uninteresting. These morphisms satisfy the standard properties that make both the sum and the product two symmetric tensor products in both **Pinj** and **Pfn**.

The two tensor products are related by a distributive law that establishes a natural isomorphism $\delta : (B + C) \times A \to (B \times A) + (C \times A)$. It is defined in the following way

$$\delta = \{((inl(b), a), inl(b, a)) \mid a \in A, b \in B\} \cup \{((inr(c), a), inr(c, a)) \mid a \in A, c \in C\}$$

while its inverse $\delta^\dagger_{A,B,C}$ is defined as

$$\delta^\dagger = \{(inl(b, a), (inl(b), a)) \mid a \in A, b \in B\} \cup \{(inr(c, a), (inr(c), a)) \mid a \in A, c \in C\}$$

Suppose $r : X + V \to Y + U$ is a relation. The coproduct injections induce four restricted relations $r_{ll}, r_{rl}, r_{lr}, r_{rr}$ defined as $r_{ll} = \{(x, y) \in X \times Y \mid (inl(x), inl(y)) \in r\}$, $r_{rl} = \{(v, y) \in V \times Y \mid (inr(v), inl(y)) \in r\}$, $r_{lr} = \{(x, u) \in X \times U \mid (inl(x), inr(u)) \in r\}$ and $r_{rr} = \{(v, u) \in V \times U \mid (inr(v), inr(u)) \in r\}$. We denote with $\cdot^\star$ the reflexive transitive closure of a relation. Given $r : X + U \to Y + U$, its trace can be defined as follows:

$$\mathrm{Tr}^U_{X,Y} r = r_{ll} \cup r_{rl} \circ r^\star_{rr} \circ r_{lr} : X \to Y$$

where $r^\star_{rr}$ is the reflexive transitive closure of r_{rr}. This operator is sometimes known as particle-style trace operator. The trace operation preserves injectivity and functionality of relations, namely if $r : X + U \to Y + U$ is functional (resp. injective) then $\mathrm{Tr}^U_{X,Y} r$ is functional (resp. injective). For details see Blute and Scott [6].

▶ **Corollary 6.** **Rel**, **Pfn** *and* **Pinj** *are symmetric traced monoidal categories.*

The proof of the above corollary has been certified in the file `pinj.ma` of [13].

The standard notion of assignment cannot be coherent with the idea of reversible computation. Just observe that the value v in a given variable is lost forever once we overwrite v by $u \neq v$. The way out is a restricted version of assignment, called *reversible update* by Axelsen, and Yokoyama [3], which we recall in the following.

▶ **Definition 7.** A functional relation $\odot : (A \times B) \to C$ is *first argument injective operator* if and only if $((a, b), c) \in \odot$ and $((a', b), c) \in \odot$ then $a = a'$.

Equivalently, if a exists such that $a \odot b = c$, for some b, c, then a is unique. Then we can define an operator $\oslash : (C \times B) \to A$ as $\oslash = \{((c, b), a) \mid a \odot b = c\}$. Notice that $\oslash$ is again a first argument injective operator and it is such that $\forall a \in A. \forall b \in B. ((a \odot b) \oslash b) = a$ and $((a \oslash b) \odot b) = a$. Sometimes we identify $\oslash$ as *the reverse of* $\odot$.

▶ **Definition 8.** Given a functional (possibly non-injective) relation $f : D \to B$ and a first argument injective operator $\odot : (A \times B) \to C$, a partial function $g : (A \times D) \to (C \times D)$ is a *reversible update wrt to its first argument* if it is functionally equivalent to

$$g(x, y) = (x \odot f(y), y).$$

There always exists a (left) inverse for a reversible update:

$$g^\dagger(x, y) = (x \oslash f(y), y)$$

where $\oslash$ is the reverse of $\odot$. Thus a reversible update g is necessarily injective.

Given an operator $\odot : A \times B \to C$ being injective in his first argument and given a partial function $f : D \to B$, we denote with $ru(\odot, f) : A \times D \to C \times D$ the partial injective function $ru(\odot, f)(x, y) = (x \odot f(y), y)$ which is a reversible update.

Reversible updates are particularly well suited to model changes of a computation state where one part of the state is updated using the remaining part of the state that is not changed. As an example consider $g(x, y) = (x + f(y), y)$ and its inverse $g^\dagger(x, y) = (x - f(y), y)$. A reversible update with the XOR ($\hat{}$) is self-inverse for any f: $g(x, y) = g^\dagger(x, y) = (x \hat{} f(y), y)$. See [3] for more details on reversible updates.

4.2 Formalization of the Model

The formalization of **Pinj** in Matita relies on a domain-theoretic substrate whose purpose is to certify a whole set of standard properties which allows us to prove the categorical/denotational correctness of a model.

We assume some familiarity with the notion and the formalization of *chain complete partial orders* (CPOs), CPO-enriched categories and monoidal categories. The formalization of domain theory is based on Benton, Kennedy, and Varming [5] and it is in `domain.ma`. We remind that the notion of chain on a CPO D is defined as any monotonic function $c : \mathbb{N} \to D$ where $\mathbb{N}$ is ordered in the natural way. Formalization of CPO-enriched category and monoidal categories are in the files `category.ma` and `monoidal_category.ma`. Our formalization of categories adapts a fragment of the COQ library [11]. As usual the homsets are CPOs and the composition of morphisms is a continuous function. Observe that given a CPO, it is always possible to extract a setoid from it by taking the symmetric closure of the preorder in the CPO as the equivalence relation of reference. This is a standard trick useful to tackle the certification of properties of extensional equivalences. The formalization of this concept is in the file `cpo_to_setoid.ma`.

Given A,B : Type[0] in Matita (roughly representing two sets), a *relation* is implemented as an inhabitant of A → B → Prop, where Prop represents the set of logic propositions. *Relational composition* of two relations r : A → B → Prop and s : B → C → Prop is $r \cdot s$ is the relation λa : A.λc : C.∃b : B.r a b ∧ s b c. This approach allows us to stay in a constructive set theory (as Matita-competent readers know). The setoid construction allows us to define the extensional collapse of these intensional definitions of sets. As expected, relations can

be structured in a CPO by taking the inclusion as its underlying pre-order and the generalized union as its lub operator (i.e. $\lambda\,c : (\mathbb{N} \to (A \to B \to \mathrm{Prop})).\lambda\,a : A.\lambda\,b : B.\exists\,n : \mathbb{N}.c\ n\ a\ b$). Relational composition is certified to be a monotonic and continuous function. It is also associative and admits a neutral element being the identity relation $\lambda\,a : A.\lambda\,a' : A.a = a'$.

We aim at certifying that the category **Pinj** of sets and relations, being injective and functional, provides a correct model for Janus. Our approach is very general. First we identify sufficient constraints on a property P on relations about our formal construction in Matita that are sufficient to provide a correct model. Second, we prove that **Pinj** can be obtained from a suitable instance of P. More precisely, if the property $P : (A \to B \to \mathrm{Prop}) \to \mathrm{Prop}$ on relations between A and B satisfies our sufficient constraints then our formal construction is:

1. a CPO-enriched category of relations satisfying the property P;
2. a symmetric monoidal category built on top of the category mentioned on point (1.), where the tensor product is the cartesian product;
3. a symmetric monoidal category built on top of the category mentioned on point (1.), where the tensor product is the disjoint union;
4. a dagger category built on top of the category mentioned on point (1.);
5. a traced monoidal category built on top of the category mentioned on point (3.), where the trace operator is particle-style;
6. is a category admitting a distributive law of cartesian product over disjoint union built on top of the categories mentioned on point (2.) and (3.).

In order to get (1.) we ask that the identity relation satisfies the property P (condition named id_ok), P is closed under logical equivalence (condition named cong_ok) and composition (condition named comp_preserve), the empty relation satisfies the property P (condition named good_bot) and given any chain $c : (\mathbb{N} \to (A \to B \to \mathrm{Prop}))$ such that $c\ n$ satisfies P for all $n : \mathbb{N}$ we have that the lub of c satisfies P (condition named good_lub). Each mathematical structure arising from relations satisfying the property P enjoying these constraints is actually a CPO-enriched category. In the formalization, all these conditions are given in the record good_rel_category in the file `rel.ma` of [13].

In order to get (2.) we ask that the property P is closed under the product of relations (condition named prod_ok) and that, both the isomorphisms describing the associativity law of the product, the left and right identity law of the product (the neutral element is the singleton set unit) and commutativity law of the product (i.e. the isomorphisms between $(A \times (B \times C))$ and $((A \times B) \times C)$ and between A and $(\mathrm{unit} \times A)$ and between A and $(A \times \mathrm{unit})$ and between $(A \times B)$ and $(B \times A)$) satisfy the property P. In the formalization, all these conditions are given in the record good_rel_prod, in the file `rel_prod.ma` of [13].

In order to get (3.), we ask *mutatis mutandis* the same conditions asked for (2.). In the formalization, all these conditions are given in the record good_rel_sum, in the file `rel_sum.ma` of [13].

In order to get (4.), we ask that the property P is closed under the operation of inversion of relations. In the formalization, this condition is given in the record good_rel_dagger, in the file `rel_dagger.ma` of [13].

In order to get (5.), we ask that P is closed under application of the particle style trace operator. In the formalization, this condition is given in the record good_rel_trace, in the file `rel_trace.ma` of [13].

In order to get (6.) we ask that the isomorphisms describing the distributive law of product over disjoint union (i.e. the isomorphisms between $A \times (B+C)$ and $(A \times B) + (A \times C)$) satisfy the property P. In the formalization this condition is given in the record good_rel_distr, in the file `rel_distr.ma` of [13].

A relation r : A → B → Prop is *functional* when if r a b and r a b' hold then b = b'. A relation r : A → B → Prop is *injective* when if r a b and r a' b hold then a = a'. If P is "all relations are functional" or "all relations are injective" then, it turns out that P satisfies all above conditions, except the one expressed by good_rel_dagger. If P is "all relations are both functional and injective" then P satisfies all above conditions, including the one expressed by good_rel_dagger. So it is possible to build the CPO-enriched category of functional injective relations which is named Pinj. The formalization of the results here mentioned can be found in the file `pinj.ma`.

4.3 Graphical Language

After Selinger [16], we use the graphical notation of Figure 5 for monoidal categories to illustrate the semantics of Janus . We do not intend neither to formalize it in the proof assistant nor to introduce it formally, we introduce it only to give a geometrical intuition of the objects being the denotation of our Janus-programs to readers. The general idea of the graphical notation is that combinators are modeled by "wiring diagrams" or "circuits" and that values are modeled as "particles" that flow along the wires. Every wire of the graph is labeled with an object which corresponds to the type of the value (denoted with a particle) flowing along that wire. Evaluation is modeled by the flow of particles along the wires. In this paper we will use graphical conventions introduced by James and Sabry [9].

Every circuit is built up from basic atomic components that are connected together. The identity is a wire. Sum and products are parallel wires: in order to distinguish them graphically, we put a + symbol between wires labeled with objects that are summed. On two summed wires a value can reside in only one of the two, while on two non-summed wires, a values have to stay on both. Commutativity is represented by crisscrossing wires. Distributivity should essentially be thought of as a multiplexer that redirects the flow of $v : A$ depending on what value inhabits the type $B + C$ as shown below. Factoring is the corresponding inverse operation. The trace operation is a looped circuit where the traced type U is shown as flowing backwards. Sum injections and quasi projections are represented as parallel summed wires in which one of them is isolated in order to denote the absence of values on that wire. Moreover an combinatorial representation of reversible update is given.

5 Interpretation

In this section we provide a denotational semantics of Janus statements in terms of injective functional relations, i.e. partial injective functions. Every Janus language construct is interpreted as a suitable composition of some categorical combinators introduced in the previous section. The goal is twofold: we aim at enforcing reversibility (since the interpretation is obtained by composition of some basic reversible functions) and we aim at making evident a connection between Janus and a framework of categorical reversible languages like those introduced in [9].

The interpretation of numerals and states is straightforward and it is given by the identity function. We denote with Σ the set of all states. From sake of simplicity, we stop to annotate states with the involved program (it is implicit in the context), but we still assume that the state is a function from all involved variables to natural numbers. An expression e of the language is interpreted into a functional relation $[\![e]\!]$ from states to $\mathbb{N}$ in the following way.

$$
\begin{array}{lcl}
[\![\mathtt{n}]\!] & = & \{(\sigma, n) \mid \sigma \in \Sigma\} \\
[\![y]\!] & = & \{(\sigma, \sigma(y)) \mid \sigma \in \Sigma\} \\
[\![e_1 \circledast e_2]\!] & = & \{(\sigma, n) \mid (\sigma, n_1) \in [\![e_1]\!], (\sigma, n_2) \in [\![e_2]\!], n = n_1 [\![\circledast]\!] n_2\}
\end{array}
$$

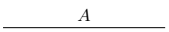
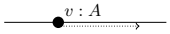
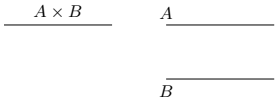
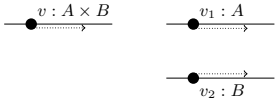
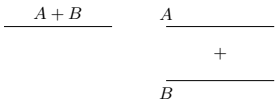
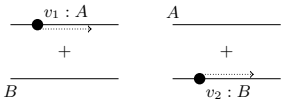
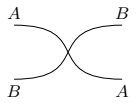
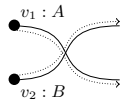
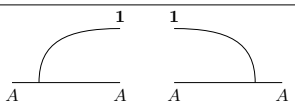
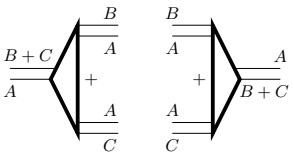
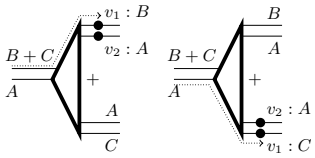
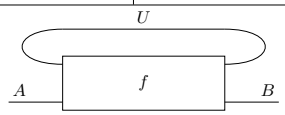
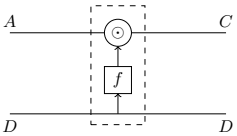
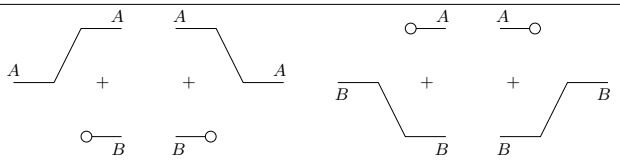
Combinator	Graphical convention	Evaluation
identity relation on A		
Cartesian product. Tokens flow in parallel along the two wires		
Disjoint union. Only one token can flow in one of the two wires.		
Isomorphism between $A \times B$ and $B \times A$		
Isomorphism between $A \times unit$ and A		
Isomorphism between $A \times (B + C)$ and $(A \times B) + (A \times C)$		
Particle-style trace on the relation $f : A + U \to B + U$		
Reversible update built from a first argument injective operator $\odot$ and a functional relation f		
Left and right injection, left and right quasi-projections (i.e. the inverses of injections)		

Figure 5 Graphical convention for circuits.

Notice that the relation being the interpretation of an expression may not be injective.

▶ **Lemma 9.** *$\sigma, e \Downarrow_e n$ if and only if $(\sigma, n) \in [\![e]\!]$.*

Lemma 9 is certified in `rel_interpretation.ma`.

To interpret guards of the conditional and loop constructs, we give a representation of truth-values. The set of boolean values is the set of functional (injective) relations between $\mathbf{1}$ and $\mathbf{1} + \mathbf{1}$ which contains only the injections $\{inl, inr\}$ as elements. True is identified by inr while false is identified by inl. The idea is to use them in conjunction with the distributive law of product over sum in order to redirect the flow in the correct branch of the conditional.

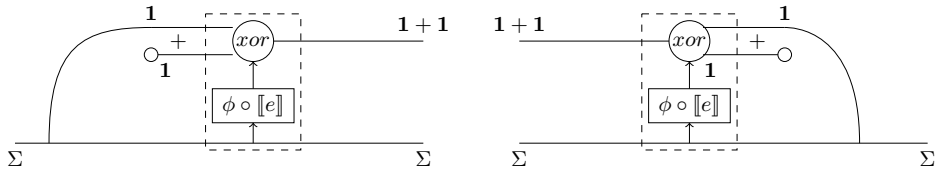


Figure 6 (a.) The testing morphism $test(e)_{\vec{x}}$. (b.) The assertion morphism $ass(e)_{\vec{x}}$.

Let $\phi : \mathbb{N} \to (\mathbf{1} + \mathbf{1})$ be the functional (non-injective) relation performing the canonical embedding of natural numbers into booleans, i.e. the functional relation mapping the natural number 0 into $inl(it)$ and all other non-zero values into $inr(it)$, where it is the unique element of the set $\mathbf{1}$. Let e be an expression. The *testing morphism* $test(e) : \Sigma \to ((\mathbf{1} + \mathbf{1}) \times \Sigma)$ generates a state annotated with the evaluation of the expression e. More formally

$$test(e) = \{(\sigma, (b, \sigma)) \mid \sigma \in \Sigma, (\sigma, n) \in [\![e]\!], (n, b) \in \phi\}$$

which is obviously an injective functional relation.

Its inverse is the *assertion morphism* $ass(e) = test(e)^\dagger : ((\mathbf{1} + \mathbf{1}) \times \Sigma) \to \Sigma$. It asserts whether in the annotated state the value of the additional variable coincides with the value of e (seen as a boolean). More formally:

$$ass(e) = \{((b, \sigma), \sigma) \mid \sigma \in \Sigma, (\sigma, n) \in [\![e]\!], (n, b) \in \phi\}$$

In Figure 6 we depicted the circuits realizing both tests and assertions.

Each statement is interpreted as a functional injective relation on states. Suppose that $\vec{id} = (id_1, \ldots, id_k)$ are the identifiers of the procedures defined. A functional environment is a k-pla $\varphi = (\varphi_1, \ldots, \varphi_k)$ where each $\varphi_i : \Sigma \to \Sigma$ is a morphism of **Pinj**, i.e. a partial injective function. We denote with $\mathbf{FEnv}_{\vec{id}}$ the set of all functional environments.

▶ **Definition 10** (Interpretation of statements). Let $\varphi = (\varphi_1, \ldots, \varphi_k)$ be a functional environment, let s be a statement. $[\![s]\!]\varphi$ is a partial injective function on Σ i.e. $[\![s]\!]\varphi : \Sigma \to \Sigma$ and it is defined by structural induction on s as follows.

- $[\![z \odot\!= e]\!]\varphi = \{(\sigma[z \mapsto n], \sigma[z \mapsto m]) \mid \sigma \in \Sigma, (\sigma\restriction_z, n') \in [\![e]\!], m = n[\![\odot]\!]n'\}$
- $[\![\texttt{call}\ id_i]\!]\varphi = \varphi_i$
- $[\![\texttt{uncall}\ id_i]\!]\varphi = \varphi_i^\dagger$
- $[\![\texttt{skip}]\!]\varphi = id_\Sigma$
- $[\![s_1\ s_2]\!]\varphi = [\![s_2]\!]\varphi \circ [\![s_1]\!]\varphi$
- $[\![\texttt{if}\ e_1\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi}\ e_2]\!]\varphi = ass(e_2) \circ \delta^\dagger \circ ((id_{\mathbf{1}} \times [\![s_2]\!]\varphi) + (id_{\mathbf{1}} \times [\![s_1]\!]\varphi)) \circ \delta \circ test(e_1)$
- $[\![\texttt{from}\ e_1\ \texttt{do}\ s_1\ \texttt{loop}\ s_2\ \texttt{until}\ e_2]\!]\varphi = \mathrm{Tr}^{\Sigma}_{\Sigma,\Sigma}(f)$ where
  $f = ((\lambda^\times)^\dagger + ((\lambda^\times)^\dagger \circ [\![s_2]\!]\varphi)) \circ \delta \circ test(e_2) \circ [\![s_1]\!]\varphi \circ ass(e_1) \circ \delta^\dagger \circ (\lambda^\times + \lambda^\times)$

The interpretation of the assignment, the conditional and the loop deserves some explanations. The morphism denoting $z \odot\!= e$ is a reversible update that allows the embedding in Janus of possibly irreversible evaluation of expressions. Clearly $\odot$ is a first argument injective operator since addition and subtraction are. We recall that z cannot occur in e because of an explicit proviso in Definition 1. In the formalization the restriction of state σ to all its names but z is implemented by setting the value of z to 0. This operation is again inane since z does not appear in e. The morphisms denoting $\texttt{if}\ e_1\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi}\ e_2$ evaluates the guard e_1 to annotate the state with a corresponding boolean. Then, according to the value of this

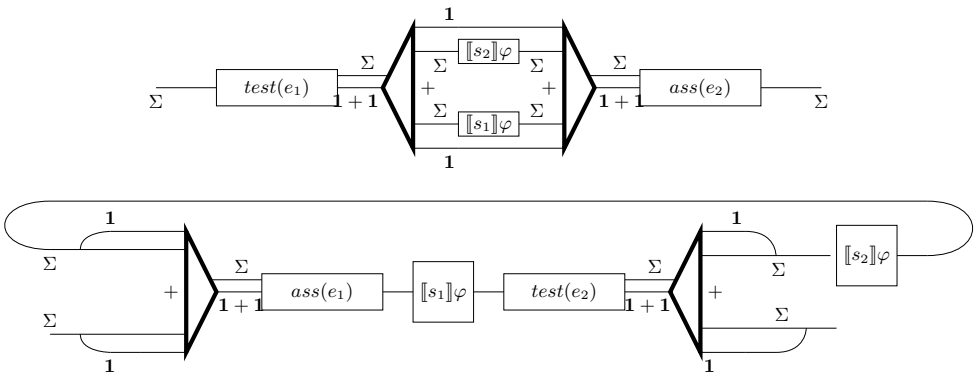


Figure 7 Graphical representation of the interpretation of conditional and loop.

annotation, the flow is redirected either to the circuit denoting s_1 or to the circuit denoting s_2 giving back the transformed state. Finally the annotation is tested to be the same as the value of the expression e_2: if the test is satisfied then the annotation is removed otherwise the operation is undefined. The morphism denoting `from` e_1 `do` s_1 `loop` s_2 `until` e_2 is obtained applying the trace operator on a function $f : (\Sigma + \Sigma) \to (\Sigma + \Sigma)$. This function is the composition of many mappings that we discuss step-by-step. First of all, f annotates the state with a boolean recording which side of the input sum was filled. Then it "asserts" whether the value of annotation corresponds to the value of the expression e_1 and if so, the annotation is removed (otherwise the function is undefined). Afterwards it evaluates the circuit denoting s_1. Then it evaluates the guard e_2 to annotate the state. Then the transformed state is redirected either on the left or on the right side of a sum according to the state annotation. In right case, the state is again transformed according to the evaluation of the circuit denoting s_2. A graphical representation of these circuits is given in Figure 7.

The set of morphisms of **Pinj** between two sets can be CPO-enriched with the set-theoretical order. For the same reason, also the set $\mathbf{FEnv}_{\vec{id}}$ endowed with the pointwise order is still a CPO. Following Winskell [17, p. 162, Lemma 9.3] one can prove that $[\![s]\!]$ is a continuous function. Thus, we can apply the fixpoint theorem to define the denotation of a Janus program containing procedure calls in terms of functional environments as explained in the following definition.

▶ **Definition 11** (Interpretation of programs). If $\mathtt{P} = \mathtt{procedure}\ id_1\ s_1, \ldots, \mathtt{procedure}\ id_k\ s_k$ is a program then, its interpretation is

$$[\![\mathtt{P}]\!] = \mathrm{fix}(\lambda\!\!\lambda\varphi.([\![s_1]\!]\varphi, \ldots, [\![s_k]\!]\varphi)) \in \mathbf{FEnv}_{\vec{id}}.$$

The following statements assert that the interpretation of statements is correct w.r.t. the operational semantics of statements.

▶ **Theorem 12.** *Let* P *be a program. Then* $\sigma_{\mathtt{P}}, s \Downarrow_p \sigma'$ *implies* $(\sigma_{\mathtt{P}}, \sigma') \in [\![s]\!]([\![\mathtt{P}]\!])$.

Proof. By induction on the derivation of the evaluation predicates. We develop only the case of loop of point (1.), the other cases being easy. Notice that in the case of `call` we need to use the definition of fixpoint together with the inductive hypothesis, while in the case of `uncall` we can use inductive hypothesis, Theorem 4 and definition of fixpoint. If $s =$`from` e_1 `do` s_1 `loop` s_2 `until` e_2 then we have that $\sigma_{\mathtt{P}}, e_1 \Downarrow_e n+1$, $\sigma_{\mathtt{P}}, s_1 \Downarrow_p \sigma'_{\mathtt{P}}$ and $\sigma'_{\mathtt{P}}, (e_1, s_1, s_2, e_2) \Downarrow_p \sigma'$. By applying Lemma 3 point (i) we know that there are $\sigma^0_{\mathtt{P}}, \ldots, \sigma^{2m}_{\mathtt{P}}$ where $m \in \mathbb{N}$ ($2m$ is the double of m), $\sigma^0_{\mathtt{P}} = \sigma'_{\mathtt{P}}$, $\sigma^{2m}_{\mathtt{P}} = \sigma'$, such that:

- $\sigma_{\mathtt{P}}^{2m}, e_2 \Downarrow_e \mathtt{n}+\mathtt{1}$ and $\sigma_{\mathtt{P}}^{2i}, e_2 \Downarrow_e \mathtt{0}$, $\sigma_{\mathtt{P}}^{2i}, s_2 \Downarrow_p \sigma_{\mathtt{P}}^{2i+1}$ for $i < m$;
- $\sigma_{\mathtt{P}}^{2i+1}, e_1 \Downarrow_e \mathtt{0}$ for $i < m$, and $\sigma_{\mathtt{P}}^{2i+1}, s_1 \Downarrow_p \sigma_{\mathtt{P}}^{2i+2}$ for $i \leq m-2$.

It is not difficult to prove by induction that if $m = 0$ then $(\sigma_{\mathtt{P}}, \sigma') \in f_{ll}$ while $(\sigma_{\mathtt{P}}, \sigma') \in f_{rl} \circ f_{rr}^* \circ f_{lr}$ if $m \geq 0$ by observing that m is the number of iterations in the trace operator. This allows us to conclude. ◀

Let P be a program. Notice that by Theorem 12 in combination with Theorem 4, it follows that $\sigma_{\mathtt{P}}, s \Downarrow_p^{\circledR} \sigma'$ implies $(\sigma_{\mathtt{P}}, \sigma') \in (\llbracket s \rrbracket(\llbracket \mathtt{P} \rrbracket))^{\dagger}$.

▶ **Theorem 13.** *Let* P *be a program. If* $(\sigma_{\mathtt{P}}, \sigma') \in \llbracket s \rrbracket(\llbracket \mathtt{P} \rrbracket)$ *then* $\sigma_{\mathtt{P}}, s \Downarrow_p \sigma'$.

Proof. We follow [17, Lemma 9.7 p. 167]. Let $\mathtt{P} = (\mathtt{procedure}\ id_1\ s_1, \ldots, \mathtt{procedure}\ id_k\ s_k)$ and define the functional environment φ as $(\sigma, \sigma') \in \varphi_i$ if $\sigma, s_i \Downarrow_p \sigma'$. Then we can prove by structural induction on s that if $(\sigma_{\mathtt{P}}, \sigma') \in \llbracket s \rrbracket(\varphi)$ then $\sigma_{\mathtt{P}}, s \Downarrow_p \sigma'$. Then we can prove easily that φ is a pre-fixpoint of the function defined in Definition 11, allowing us to conclude. ◀

The formalization of denotational semantics is given in the file `rel_interpretation.tex`. The interpretation is provided for all suitable instances of the abstract syntax.

```
let rec den_eval_stm (p : params) (p' : sem_params p)
(prog : stm p) (n : ℕ) on prog :
ℒ^{n}_{(Mor Pinj (list (const_type ...p)) (list (const_type ...p)))} →
(Mor Pinj (list (const_type ...p)) (list (const_type ...p))) :=...
```

The interpretation is defined by induction on the statement prog. It gives back a function from ℒ^{n}_{(Mor Pinj (list (const_type ...p)) (list (const_type ...p)))} (which is the implementation of $\mathbf{FEnv}_{\vec{id}}$) to an injective functional relation on states. We proved that the obtained function is both monotonic and continuous. So it is possible to define the interpretation using the fixpoint operator. The analogous of theorems 12 and 13 are certified in `correctness.ma` and `compl_thm.ma`.

6 Conclusions

This paper focuses on various aspects of the semantics of the reversible and imperative basic programming language Janus.

- On one side we focus on the operational semantics of Janus. The reason is standard: constructing efficient compilers and interpreters rely on well formalized and unambiguous operational semantics. We provide a syntax-driven operational semantics which is deterministic. This property does not hold for the first contributions to the operational semantics of Janus [20, 19, 18, 12]. The proposals in [20, 19, 18, 12] rest on a non-deterministic rule which describes a mechanism of "procedure un-call." For sake of completeness, however, we must recall that the authors of [20, 12] explain how to let their evaluation be deterministic by means of an external program transformation.
  A noteworthy by-product of our approach is that the well-known properties expressed by Corollary 5 can be formally expressed and proven in a friendly operational settings (indeed, the functionality and the injectivity of the evaluation are usually proved for Reversible Turing-Machines, for which we refer to Axelsen and Glück [2].)
- On the other side, we focus on a suitable categorical domain where correctly interpreting Janus. Obtaining full-abstraction is somewhat expected because Janus is not a higher-order language. The formalization of the interpretation, however, is not trivial because the semantic analysis of reversible languages is quite a new trend. In particular, we see

our analysis and its relationships with the work by James and Sabry [9, 8] as a starting point to deepen the knowledge about the reversible programming. So we are in the position to exploit a standard by-product of semantics, i.e. simplified tools to investigate equivalences among programs.

- Finally, our (meta)-proofs are certified by Matita.

References

1. Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The Matita interactive theorem prover. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, volume 6803 of *Lecture Notes in Computer Science*, pages 64–69. Springer, 2011. doi:10.1007/978-3-642-22438-6_7.
2. Holger Bock Axelsen and Robert Glück. What do reversible programs compute? In Martin Hofmann, editor, *Foundations of Software Science and Computational Structures - 14th International Conference, FOSSACS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6604 of *Lecture Notes in Computer Science*, pages 42–56. Springer, 2011. doi:10.1007/978-3-642-19805-2_4.
3. Holger Bock Axelsen, Robert Glück, and Tetsuo Yokoyama. Reversible machine code and its abstract processor architecture. In Volker Diekert, Mikhail V. Volkov, and Andrei Voronkov, editors, *Computer Science - Theory and Applications, Second International Symposium on Computer Science in Russia, CSR 2007, Ekaterinburg, Russia, September 3-7, 2007, Proceedings*, volume 4649 of *Lecture Notes in Computer Science*, pages 56–69. Springer, 2007. doi:10.1007/978-3-540-74510-5_9.
4. C. H. Bennett. Logical reversibility of computation. *IBM J. Res. Develop.*, 17(6):525–532, 1973. doi:10.1147/rd.176.0525.
5. Nick Benton, Andrew Kennedy, and Carsten Varming. Some domain theory and denotational semantics in Coq. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2009. doi:10.1007/978-3-642-03359-9_10.
6. R. Blute and P. Scott. Category theory for linear logicians. In T. Ehrhard, J.-Y. Girard, P. Ruet, and P. Scott, editors, *Linear Logic in Computer Science*, volume 316 of *London Math. Soc. Lecture Note Series*, pages 3–64. Cambridge University Press, 2004. doi:10.1017/cbo9780511550850.002.
7. The Coq Development Team. The Coq proof assistant development manual, 2005. URL: http://coq.inria.fr/doc/main.html.
8. R. P. James and A. Sabry. Theseus: A high level language for reversible computing, 2014. URL: http://www.cs.indiana.edu/~sabry/research.html.
9. Roshan P. James and Amr Sabry. Isomorphic interpreters from logically reversible abstract machines. In Robert Glück and Tetsuo Yokoyama, editors, *Reversible Computation, 4th International Workshop, RC 2012, Copenhagen, Denmark, July 2-3, 2012. Revised Papers*, volume 7581 of *Lecture Notes in Computer Science*, pages 57–71. Springer, 2012. doi:10.1007/978-3-642-36315-3_5.
10. C. Lutz. Janus, a time reversible language. Letter to R. Landauer, 1986. URL: http://www.peterhines.net/downloads/others/JANUS.html.
11. A. Megacz. Category theory library for Coq. URL: http://www.cs.berkeley.edu/~megacz/coq-categories/.

12 Torben Æ. Mogensen. Partial evaluation of the reversible language janus. In Siau-Cheng Khoo and Jeremy G. Siek, editors, *Proceedings of the 2011 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2011, Austin, TX, USA, January 24-25, 2011*, pages 23–32. ACM, 2011. doi:10.1145/1929501.1929506.

13 L. Paolini, M. Piccolo, and L. Roversi. Janus formalization in Matita. URL: http://www.di.unito.it/~piccolo/janus.zip.

14 Luca Paolini, Mauro Piccolo, and Luca Roversi. A class of reversible primitive recursive functions. *Electr. Notes Theor. Comput. Sci.*, 322:227–242, 2016. doi:10.1016/j.entcs.2016.03.016.

15 K. S. Perumalla. *Introduction to Reversible Computing*. Chapman & Hall/CRC Computational Science. CRC Press, 2013. doi:10.1201/b15719.

16 P. Selinger. A survey of graphical languages for monoidal categories. In B. Coecke, editor, *New Structures for Physics*, Lect. Notes in Physics, pages 289–355. Springer, 2011. doi:10.1007/978-3-642-12821-9_4.

17 G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

18 Tetsuo Yokoyama. Reversible computation and reversible programming languages. *Electr. Notes Theor. Comput. Sci.*, 253(6):71–81, 2010. doi:10.1016/j.entcs.2010.02.007.

19 Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. Principles of a reversible programming language. In Alex Ramírez, Gianfranco Bilardi, and Michael Gschwind, editors, *Proceedings of the 5th Conference on Computing Frontiers, 2008, Ischia, Italy, May 5-7, 2008*, pages 43–54. ACM, 2008. doi:10.1145/1366230.1366239.

20 Tetsuo Yokoyama and Robert Glück. A reversible programming language and its invertible self-interpreter. In G. Ramalingam and Eelco Visser, editors, *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2007, Nice, France, January 15-16, 2007*, pages 144–153. ACM, 2007. doi:10.1145/1244381.1244404.

21 M. Zorzi. On quantum lambda calculi: A foundational perspective. *Math. Struct. Comput. Sci.*, 26(7):1107–1195, 2016. doi:10.1017/s0960129514000425.