A Reversible Programming Language and its Invertible Self-Interpreter

Tetsuo Yokoyama[†,‡] Robert Glück[‡]

[†]DIKU, Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen Ø
[‡]Graduate School of Information Science and Technology, The University of Tokyo
7-3-1 Hongo, Bunkyo-ku, 113-8656 Tokyo, JAPAN
yokoyama@diku.dk, glueck@acm.org

Abstract

A reversible programming language supports deterministic forward and backward computation. We formalize the programming language Janus and prove its reversibility. We provide a program inverter for the language and implement a self-interpreter that achieves deterministic forward and backward interpretation of Janus programs without using a computation history. As the self-interpreter is implemented in a reversible language, it is invertible using local program inversion. Many physical phenomena are reversible and we demonstrate the power of Janus by implementing a reversible program for discrete simulation of the Schrödinger wave equation that can be inverted as well as run forward and backward.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.1.m [*Programming Techniques*]: Miscellaneous; D.3.4 [*Programming Languages*]: Processors—interpreters

General Terms Languages, Theory

Keywords Janus, Non-standard interpreter hierarchy, Program inversion, Reversible computing, Reversible programming language, Self-interpreter

1. Introduction

A *reversible computing system* [30, 4, 10] has, at any time, at most a single previous computation state as well as a single next computation state, and thus a reversible computing system can run programs uniquely forward and backward by following the deterministic trajectory of the computation.

Many reversible computation models are as powerful as their irreversible counterparts. For example, Turing machines are reversible if their transition functions are bijective [3, 20]. Addition of a computation history that keeps track of every computation step provides a general translation from irreversible to reversible Turing machines [19]. Thus, given unlimited resources, reversible Turing machines are as powerful as their irreversible counterparts.

Reversible computing has a close relationship to the energy dissipation of physical computing processes. Landauer showed that a computation process where the previous state is not determined uniquely must dissipate at least a minimum amount of heat [19]. He also conjectured that some computations cannot be made reversible. However, Bennett showed this conjecture to be incorrect [3], which implies, at least in theory, that the amount of heat dissipation in the computing process has no lower bound.

Thus, *reversible computing* holds the promise of reducing the power consumption of physical computation processes (cf., reviews [32, 10, 4]), an idea that has recently attracted renewed interest. Examples are the reversible adder [6, 17] and the reversible microprocessor Pendulum [31, Part II] and the design of the instruction set architectures for reversible chips [31, Part III and Appdx. A][9, Ch. 9 and Appdx. B][16]. However, to gain the greatest degree of profit of recycling energy by reversible computing systems, it is not only necessary to consider low-level hardware issues, but also high-level logical reversibility at the software level.

This paper focuses on formalization of the reversible programming language Janus (Sec. 2.1-2.2) and proving its reversibility (Sec. 2.3). We provide an automatic program inverter for the language (Sec. 2.3.3) and implement a reversible self-interpreter (Sec. 3.1-3.2). To our knowledge, this is the first reversible self-interpreter reported to date. In common with other programming paradigms, reversible programming has its own programming methodology. We explore basic programming techniques based on our practical experience with programming in a reversible language (Sec. 3.3). We show the power of Janus by implementing a reversible program for discrete simulation of the Schrödinger wave equation that can be inverted as well as run forward and backward (Sec. 4) and by running several reversible computing experiments with a tower of interpreters.

Intuitively, reversible computing is closely related to program inversion. For example, a reversible computing system has an inverse system that can be obtained by inverting the direction of its state transitions. We show the relationship between program inversion and reversibility. Several programming languages have been called reversible without further formalizing this notion. Here, we characterize reversibility in terms of local invertibility.

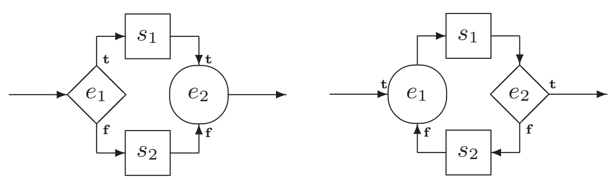
2. The Janus Language

The imperative language Janus appears to be the first reversible structured programming language. Although it was first suggested for a class at Caltech [23], it shows the fundamental constructs that are necessary for reversible languages. The main differences from conventional languages are reversible assignments and con-

PEPM 2007 January 15-16, 2007, Nice, France.

Syntax Domains	**Grammar**
$p \in$ Progs[Janus]	$p ::= d^*$ (`procedure` $id\ s$)$^+$
$x \in$ Vars[Janus]	$d ::= x \mid x$`[`c`]`
$c \in$ Cons[Janus]	$s ::= x \oplus$`=` $e \mid x$`[`e`]` $\oplus$`=` $e \mid$
$id \in$ Idens[Janus]	`if` e `then` s `else` s `fi` $e \mid$
$s \in$ Stmts[Janus]	`from` e `do` s `loop` s `until` $e \mid$
$e \in$ Exps[Janus]	`call` $id \mid$ `uncall` $id \mid$ `skip` $\mid s\ s$
$\oplus \in$ ModOps[Janus]	$e ::= c \mid x \mid x$`[`e`]` $\mid e \odot e$
$\odot \in$ Ops[Janus]	$c ::=$ `0` $\mid$ `1` $\mid \cdots \mid$ `4294967295`
	$\oplus ::=$ `+` $\mid$ `-` $\mid$ `^`
	$\odot ::= \oplus \mid$ `*` $\mid$ `/` $\mid$ `%` $\mid$ `*/` $\mid$ `&` $\mid$ `|` $\mid$ `&&` $\mid$ `||` $\mid$
	`<` $\mid$ `>` $\mid$ `=` $\mid$ `!=` $\mid$ `<=` $\mid$ `>=`

Figure 1. Syntax of Janus



(a) Conditional (b) Loop

Figure 2. Reversible structured control flow

trol constructs, and the possibility of uncalling procedures (i.e., to running them backward). The language is simple, yet powerful, and its constructs can serve as a model for designing other reversible languages.

2.1 Syntax

A Janus *program* consists of variable declarations and procedure declarations (Fig. 1).[1] A *variable declaration* defines a *variable* or a one-dimensional array. There is no other type information as the only values in the language are non-negative integers. Subscripts of an array start from `0`. A procedure declaration consists of a keyword `procedure`, an *identifier*, which is the procedure name, and a statement, which is the procedure body. A *statement* is a reversible assignment, a reversible conditional, a reversible loop, a procedure call, a procedure uncall, a skip, or a statement sequence.

A *reversible assignment* is similar to an assignment in the programming language C. The variable x on the left-hand side of an assignment must not appear in the expression e on the right-hand side. Similarly, array variable x must not appear in the expression e on either side of the assignment. This, together with the reversible modify operator $\oplus$ (addition, subtraction, bitwise exclusive-or), makes the execution of assignments reversible (discussed later). An assignment is the only way of changing the value of a variable.

A *reversible conditional* has two predicates: the predicate after `if` is the *test*, and that after `fi` is the *assertion*. If the test is true, the then-branch is executed and afterward the assertion must be true; if it is false, the conditional is undefined. Similarly, if the test is false, the assertion must be false after executing the else-branch. As usual, zero is considered as false, with any other value considered as true. The control flow is illustrated in Fig. 2(a) where e_1 is the test, e_2 the assertion, s_1 the then-branch, and s_2 the else-branch. We circle the assertion and mark the incoming arcs with true (**t**) and false (**f**), as required. The assertion makes the conditional reversible.

A *reversible loop* has two predicates: the predicate after `from` is the *assertion*, and that after `until` is the *test*. The control flow can be seen in Fig. 2(b): initially, assertion e_1 must be true and do-statement s_1 is executed. If test e_2 is true, the loop terminates, otherwise, loop-statement s_2 is executed, after which e_1 must be false. If the assertion does not have the required value, execution of the loop is undefined. The assertion is only initially true. This makes the loop reversible.

[1] For simplicity, we do not consider input and output operations. Some operators in the original letter [23] were changed into C-like notation.

$$\begin{aligned}
v &\in \mathrm{Vals[Janus]} &&= \{0,\ldots,2^{32}-1\}\\
l &\in \mathrm{Lvals[Janus]} &&= \{\mathtt{a},\mathtt{b},\ldots,\mathtt{a[0]},\mathtt{a[1]},\ldots,\mathtt{b[0]},\ldots\}\\
\sigma &\in \mathrm{Stores[Janus]} &&= \mathrm{Lvals[Janus]} \rightharpoonup \mathrm{Vals[Janus]}\\
\Gamma &\in \mathrm{PMaps[Janus]} &&= \mathrm{Idens[Janus]} \rightharpoonup \mathrm{Stmts[Janus]}
\end{aligned}$$

Figure 3. Semantic values

A *procedure call* executes the procedure body in the global store. There are no parameters or local variables. To pass values to and from a procedure, we use side effects on the global store. A *procedure uncall* invokes inverse computation of the procedure. As discussed later, an inverse procedure call is efficient in Janus.

An *expression* is a constant (a 32-bit non-negative integer ranging from `0` to $2^{32}-1$), a variable, an indexed variable, or a binary expression. A *binary operator* $\odot$ is one of the arithmetic (`+`,`-`,`*`,`/`,`%`,`*/`), bitwise (`&`,`|`,`^`), logical (`&&`,`||`), or relational operators (`<`,`>`,`=`,`!=`,`<=`,`>=`). All are defined on non-negative integers. All arithmetic operations are modulo 2^{32}. The binary operators in expressions need not be injective. We will see later why this does not harm the reversibility of statements.

To denote the syntactic components of a program p, we write $\mathrm{Stmts}[p]$, $\mathrm{Exps}[p]$, etc.. For example, $\mathrm{Stmts}[p]$ is the set of statements in p. We consider only programs that are well-formed (every identifier in `call` and `uncall` in p is declared as a procedure name; every variable used in p appears in the variable declarations).

Example program Given a number `n`, procedure `fib` computes the (`n+1`)-th and (`n+2`)-th Fibonacci number in `x1` and `x2` [13]. The variable declarations are `n x1 x2`. All variables are initially set to zero. The procedure declarations are (`main_fwd` sets `n` to 4 by adding it to zero-cleared `n`):[2]

```
procedure fib                   procedure main_fwd
  if n=0 then x1 += 1             n += 4
              x2 += 1             call fib
         else n -= 1
              call fib          procedure main_bwd
              x1 += x2            x1 += 5
              x1 <=> x2           x2 += 8
  fi x1=x2                        uncall fib
```

All Janus procedures are reversible. Setting `n` to 4 and calling `fib` in `main_fwd` computes the Fibonacci numbers `x1` $= 5$ and `x2` $= 8$. Setting `x1` to 5 and `x2` to 8 in `main_bwd`, computes `n` $= 4$ by uncalling `fib`. Note that the same procedure, `fib`, is used for deterministic forward and backward computation.

2.2 Operational Semantics

The semantics of Janus programs is specified by the rules shown in Fig. 4. The semantics have two main judgments: the evaluation of expressions and the execution of statements. Before we explain the rules, we will briefly describe the semantic values in Fig. 3 along with some notation.

Preliminaries A *value* v is a non-negative integer ranging from 0 to $2^{32}-1$. A *left-value* l is a variable name or an indexed

[2] Swap `x1 <=> x2` is an abbreviation for the statement sequence `x1 ^= x2; x2 ^= x1; x1 ^= x2`. Bitwise exclusive-or (`^`) allows to swap values without destroying existing ones.

Evaluation of Expressions

$$\frac{}{\sigma \vdash_{expr} c \Rightarrow [\![c]\!]}\ \textsc{Con} \qquad \frac{}{\sigma \vdash_{expr} x \Rightarrow \sigma(x)}\ \textsc{Var} \qquad \frac{\sigma \vdash_{expr} e \Rightarrow v}{\sigma \vdash_{expr} x\texttt{[}e\texttt{]} \Rightarrow \sigma(x\texttt{[}v\texttt{]})}\ \textsc{Arr} \qquad \frac{\begin{array}{c}\sigma \vdash_{expr} e_1 \Rightarrow v_1 \quad \sigma \vdash_{expr} e_2 \Rightarrow v_2 \\ [\![\odot]\!](v_1, v_2) = v\end{array}}{\sigma \vdash_{expr} e_1 \odot e_2 \Rightarrow v}\ \textsc{Bop}$$

Execution of Statements

$$\frac{\sigma \vdash_{expr} e \Rightarrow v}{\sigma \vdash_{stmt} x \oplus\texttt{=}\, e \Rightarrow \sigma[x \mapsto [\![\oplus]\!](\sigma(x), v)]}\ \textsc{AssVar} \qquad \frac{\sigma \vdash_{expr} e_l \Rightarrow v_l \quad \sigma \vdash_{expr} e \Rightarrow v}{\sigma \vdash_{stmt} x\texttt{[}e_l\texttt{]} \oplus\texttt{=}\, e \Rightarrow \sigma[x\texttt{[}v_l\texttt{]} \mapsto [\![\oplus]\!](\sigma(x\texttt{[}v_l\texttt{]}), v)]}\ \textsc{AssArr}$$

$$\frac{\begin{array}{c}\sigma \vdash_{expr} e_1 \Rightarrow v_1 \\ \textsf{is-true?}(v_1)\end{array} \quad \sigma \vdash_{stmt} s_1 \Rightarrow \sigma' \quad \begin{array}{c}\sigma' \vdash_{expr} e_2 \Rightarrow v_2 \\ \textsf{is-true?}(v_2)\end{array}}{\sigma \vdash_{stmt} \texttt{if}\ e_1\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi}\ e_2 \Rightarrow \sigma'}\ \textsc{IfTrue} \qquad \frac{\begin{array}{c}\sigma \vdash_{expr} e_1 \Rightarrow v_1 \\ \textsf{is-false?}(v_1)\end{array} \quad \sigma \vdash_{stmt} s_2 \Rightarrow \sigma' \quad \begin{array}{c}\sigma' \vdash_{expr} e_2 \Rightarrow v_2 \\ \textsf{is-false?}(v_2)\end{array}}{\sigma \vdash_{stmt} \texttt{if}\ e_1\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi}\ e_2 \Rightarrow \sigma'}\ \textsc{IfFalse}$$

$$\frac{\begin{array}{c}\sigma \vdash_{expr} e_1 \Rightarrow v_1 \\ \textsf{is-true?}(v_1)\end{array} \quad \sigma \vdash_{loop1} (e_1, s_1, s_2, e_2) \Rightarrow \sigma' \quad \begin{array}{c}\sigma' \vdash_{expr} e_2 \Rightarrow v_2 \\ \textsf{is-true?}(v_2)\end{array}}{\sigma \vdash_{stmt} \texttt{from}\ e_1\ \texttt{do}\ s_1\ \texttt{loop}\ s_2\ \texttt{until}\ e_2 \Rightarrow \sigma'}\ \textsc{LoopMain}$$

$$\frac{\sigma \vdash_{stmt} s_1 \Rightarrow \sigma'}{\sigma \vdash_{loop1} (e_1, s_1, s_2, e_2) \Rightarrow \sigma'}\ \textsc{Loop1Base} \qquad \frac{\sigma \vdash_{stmt} s_2 \Rightarrow \sigma'}{\sigma \vdash_{loop2} (e_1, s_1, s_2, e_2) \Rightarrow \sigma'}\ \textsc{Loop2Base}$$

$$\frac{\sigma \vdash_{stmt} s_1 \Rightarrow \sigma' \quad \begin{array}{c}\sigma' \vdash_{expr} e_2 \Rightarrow v_2 \\ \textsf{is-false?}(v_2)\end{array} \quad \sigma' \vdash_{loop2} (e_1, s_1, s_2, e_2) \Rightarrow \sigma'' \quad \begin{array}{c}\sigma'' \vdash_{expr} e_1 \Rightarrow v_1 \\ \textsf{is-false?}(v_1)\end{array} \quad \sigma'' \vdash_{stmt} s_1 \Rightarrow \sigma'''}{\sigma \vdash_{loop1} (e_1, s_1, s_2, e_2) \Rightarrow \sigma'''}\ \textsc{Loop1Rec}$$

$$\frac{\sigma \vdash_{stmt} s_2 \Rightarrow \sigma' \quad \begin{array}{c}\sigma' \vdash_{expr} e_1 \Rightarrow v_1 \\ \textsf{is-false?}(v_1)\end{array} \quad \sigma' \vdash_{loop1} (e_1, s_1, s_2, e_2) \Rightarrow \sigma'' \quad \begin{array}{c}\sigma'' \vdash_{expr} e_2 \Rightarrow v_2 \\ \textsf{is-false?}(v_2)\end{array} \quad \sigma'' \vdash_{stmt} s_2 \Rightarrow \sigma'''}{\sigma \vdash_{loop2} (e_1, s_1, s_2, e_2) \Rightarrow \sigma'''}\ \textsc{Loop2Rec}$$

$$\frac{\sigma \vdash_{stmt} \Gamma(id) \Rightarrow \sigma'}{\sigma \vdash_{stmt} \texttt{call}\ id \Rightarrow \sigma'}\ \textsc{Call} \qquad \frac{\sigma' \vdash_{stmt} \Gamma(id) \Rightarrow \sigma}{\sigma \vdash_{stmt} \texttt{uncall}\ id \Rightarrow \sigma'}\ \textsc{Uncall} \qquad \frac{}{\sigma \vdash_{stmt} \texttt{skip} \Rightarrow \sigma}\ \textsc{Skip} \qquad \frac{\begin{array}{c}\sigma \vdash_{stmt} s_1 \Rightarrow \sigma' \\ \sigma' \vdash_{stmt} s_2 \Rightarrow \sigma''\end{array}}{\sigma \vdash_{stmt} s_1\ s_2 \Rightarrow \sigma''}\ \textsc{Seq}$$

Figure 4. Operational semantics of Janus programs

variable name. The *store* σ is a partial function from left-values to values. The application of a store σ to a left-value l is denoted by $\sigma(l)$. Given a program p, the domain of σ is $\text{Lvals}[p]$ and $\sigma(l)$ is undefined if $l \notin \text{Lvals}[p]$. Update $\sigma[l \mapsto v]$ denotes the same mapping as σ except that l maps to v. Equality of two stores, σ and σ', is defined by $\sigma = \sigma' \stackrel{\text{def}}{=} \forall l \in \text{Lvals}[p].\ \sigma(l) = \sigma'(l)$. A *procedure-map* Γ is a partial function from identifiers to statements. Program execution starts in a zero-cleared *initial store* σ_{init} (all $l \in \text{Lvals}[p]$ map to 0).

Evaluation of Expressions Judgment $\sigma \vdash_{expr} e \Rightarrow v$ defines the meaning of expressions where σ is a store, e an expression, and v a value. We say that under store σ, expression e evaluates to value v. There is no side effect on the store. All operators $\odot$ are defined wrt integers. Some definitions are as follows (others are similar):

$$\begin{array}{ll} [\![\texttt{+}]\!](v_1, v_2) = v_1 +_{32} v_2 & [\![\texttt{*/}]\!](v_1, v_2) = \lfloor (v_1 * v_2)/2^{32} \rfloor \\ [\![\texttt{-}]\!](v_1, v_2) = v_1 -_{32} v_2 & [\![\texttt{=}]\!](v_1, v_2) \ = \begin{cases} 0 \ \textit{if}\ v_1 \neq v_2 \\ 1 \ \textit{if}\ v_1 = v_2 \end{cases} \\ [\![\texttt{\^{}}]\!](v_1, v_2) = v_1\ \textit{xor}\ v_2 & \end{array}$$

where $v_1 \odot_{32} v_2 \stackrel{\text{def}}{=} (v_1 \odot v_2) \mod 2^{32}$, floor $\lfloor v \rfloor$ is the largest integer less than or equal to v, and $\textit{xor}$ is bitwise exclusive-or on the binary representation of integers. Operator `*/` is the fractional product where one factor is regarded as an integer and the other as a fraction between 0 and 1.

Execution of Statements Judgment $\sigma \vdash_{stmt} s \Rightarrow \sigma'$ defines the meaning of statements where σ and σ' are stores, and s is a statement. We say that under store σ, the execution of statement s yields the updated store σ'. We call σ the input and σ' the output. As the procedure map Γ is fixed for a given program, we omit it from the judgment for notational simplicity.

The meaning of an assignment is defined by the rules ASSVAR and ASSARR. We distinguish between assignments to variables and to indexed variables. The assignment operator ($\oplus$`=`) stands for (`+=`), (`-=`), (`^=`). The meaning of a conditional is defined by rules IFTRUE and IFFALSE, and which rule applies depends on the value of e_1 and e_2 (cf. Fig. 2). Predicates $\textsf{is-true?}(v)$ and $\textsf{is-false?}(v)$ stand for $v \neq 0$ and $v = 0$, respectively.

The meaning of a loop is defined by a main rule for the entry and exit of a loop and four symmetric rules that specify the statement sequence that is executed when repeating the loop. Rule LOOPMAIN requires assertion e_1 and test e_2 to be true when entering and exiting a loop (cf. Fig. 2). The statement sequence $s_1\, s_2 \ldots s_2\, s_1$ that is executed by the loop is specified by the two judgments indexed by *loop1* and *loop2*. Rule LOOP1REC specifies the portion of the statement sequence that executes s_1 before and after judgment *loop2*, which requires that e_1 and e_2 are both false. Similarly for LOOP2REC. Rules LOOP1BASE and LOOP2BASE specify the innermost statement of the statement sequence, which is the execution of either s_1 or s_2. The rules show the symmetry of a reversible loop which is expressed by a central recursion. Clearly, they can also be formulated in a way using right- and left-recursion that allows for a direct and efficient implementation of the forward and backwards semantics.

A procedure call executes the procedure body $\Gamma(id)$ in the current store. It relates the input store σ and the output store σ' of the call with the corresponding stores of the procedure body. The rule is simple because there are no parameters or local variables. Conversely, a procedure uncall relates σ and σ' with the opposite stores of the procedure body: the input store σ is the body's output store, and the output store σ' is the body's input store. An uncall changes the direction of executing the procedure body. This is an important mechanism of Janus (cf. example `fib` in 2.1).

The SKIP rule does not change the input and output stores. The execution of a statement sequence is defined by rule SEQ.

2.3 Reversibility of Janus

A reversible programming language supports deterministic forward and backward computation. Here, are going to show that Janus is indeed a reversible programming language; we will show that all statements are forward and backward deterministic and that *local inversion* is sufficient to produce *deterministic inverse programs*. Consequently, Janus programs can be run efficiently in both directions. From a theoretical viewpoint, inverse computation of any program is possible using McCarthy's generate-and-test approach [25], regardless of the type of programming language used to write the program, but this approach is much too inefficient to be practical and in general there is no unique solution. This is one of the reasons why we study the conditions for reversible languages.

2.3.1 Forward and Backward Determinism

The inference system for Janus statements is *forward deterministic*: if, for some statement s and some store σ, judgment $\sigma \vdash_{stmt} s \Rightarrow \sigma'$ holds, then store σ' is unique. We will also prove that the system is *backward deterministic*: if, for some statement s and some store σ', $\sigma \vdash_{stmt} s \Rightarrow \sigma'$ holds, then store σ is unique. This is the basis of the reversibility of Janus programs. The following lemma expresses that expression evaluation is forward deterministic.

Lemma 1 (Forward determinism of evaluation)**.**

$$\begin{array}{l}\forall e \in \text{Exps}[\text{Janus}], \forall \sigma \in \text{Stores}[\text{Janus}].\\ \quad \sigma \vdash_{expr} e \Rightarrow v' \ \wedge\ \sigma \vdash_{expr} e \Rightarrow v'' \implies v' = v''\end{array}$$

Proof. Structural induction on expressions. Clearly, axioms CON and VAR are forward deterministic. In the inductive case of ARR, the evaluation of the index expression is assumed to be forward deterministic. In BINOP we use the additional fact that $[\![\odot]\!]$ is a function for every binary operator. □

The evaluation of expressions is not backward deterministic because function $[\![\odot]\!]$ is not injective, and thus there exists no inverse. As we shall see, this does not harm the backward and forward determinism of Janus statements. We have the following theorem which tells us that statement execution is an injective function for any statement, and thus the inverse exists. It also tells us that it is not possible to write irreversible statements in Janus.

Theorem 2 (Forward & backward determinism of execution)**.**

$$\begin{array}{l}\forall \sigma, \sigma', \sigma'' \in \text{Stores}[\text{Janus}].\\ \quad \sigma \vdash_{stmt} s \Rightarrow \sigma' \ \wedge\ \sigma \vdash_{stmt} s \Rightarrow \sigma'' \quad \implies \quad \sigma' = \sigma''\\ \forall \sigma, \sigma', \sigma'' \in \text{Stores}[\text{Janus}].\\ \quad \sigma' \vdash_{stmt} s \Rightarrow \sigma \ \wedge\ \sigma'' \vdash_{stmt} s \Rightarrow \sigma \quad \implies \quad \sigma' = \sigma''\end{array}$$

Proof. We use rule induction using the induction hypothesis that both properties are satisfied for the premises of the inference rules.

An interesting case is to prove forward and backward determinism of assignments. In ASSVAR, expression evaluation $\sigma \vdash_{expr} e \Rightarrow v$ is forward deterministic by Lemma 1 and $[\![\oplus]\!]$ is a function for every modify operator $\oplus$. Therefore, ASSVAR is forward deterministic. The proof of backward determinism uses two properties of Janus. First, the syntactic restriction (Sec. 2.1) that x in $x \oplus$= e does not occur in e. Thus, $\sigma \vdash_{expr} e \Rightarrow v$ is independent of the value of x in σ. This means that e evaluates to the same value v before and after updating x in σ. Second, for every v, function $\lambda v'.\ [\![\oplus]\!](v', v)$ is bijective and, thus, an inverse function exists. Consequently, ASSVAR is backward deterministic. The proof for ASSARR is similar.

For conditionals, IFTRUE and IFFALSE, the induction hypothesis about statement execution immediately implies forward and backward determinism of the conclusion. Because expression evaluation is forward deterministic and because is-true? and is-false? are mutually exclusive, there is precisely one rule that may be used.

For loops, the sequence of statement executions of s_1 and s_2 (the number of loop iterations) uniquely determines the proof tree of LOOP1BASE, LOOP1REC, LOOP2BASE, and LOOP2REC. The induction hypothesis immediately implies forward and backward determinism of the conclusion of each loop rule. The evaluation of assertion e_1 and test e_2 uniquely determines the loop iterations.

The proofs for CALL, SKIP, and SEQ are immediate. In UNCALL, we rely on the assumption of backward determinism to prove forward determinism, and vice versa. □

2.3.2 Local Invertibility of Statements

A statement s_1 is the *inverse* of a statement s_2 iff for all σ and σ'

$$\sigma \vdash_{stmt} s_1 \Rightarrow \sigma' \iff \sigma' \vdash_{stmt} s_2 \Rightarrow \sigma\ .$$

We introduce an equivalence relation $\sim$ on expressions and statements.

$$\begin{array}{l}e_1 \sim e_2 \text{ iff } (\forall v \in \text{Vals}[\text{Janus}], \forall \sigma \in \text{Stores}[\text{Janus}].\\ \qquad\qquad \sigma \vdash_{expr} e_1 \Rightarrow v \iff \sigma \vdash_{expr} e_2 \Rightarrow v)\\ s_1 \sim s_2 \text{ iff } (\forall \sigma, \sigma' \in \text{Stores}[\text{Janus}].\\ \qquad\qquad \sigma \vdash_{stmt} s_1 \Rightarrow \sigma' \iff \sigma \vdash_{stmt} s_2 \Rightarrow \sigma')\end{array}$$

Usually, a statement has more than one inverse statement. They are syntactically different but they are all equivalent to each other. We write an inverse of s as $\breve{s}$. Concatenation of two statements, represented by a space, is associative, i.e., $s_1\ (s_2\ s_3) \sim (s_1\ s_2)\ s_3$, and we omit the parentheses. For simplicity, we write ϵ where $\epsilon \sim$ `skip`. The following lemma states some useful properties about statements.

Lemma 3.

Referential transparency :	$s \sim s' \implies s_1\ s\ s_2 \sim s_1\ s'\ s_2$
Inverse :	$s_1\ s_2 \sim \epsilon \iff s_2\ s_1 \sim \epsilon$
Sequential inverse :	$(s_1\ s_2)\breve{} \sim \breve{s}_2\ \breve{s}_1$

Referential transparency states that the equivalent relation on statements is context independent; that is, we can replace equivalent statements in a statement sequence without changing the meaning. Inverse states that inverse statements are commutative. Sequential inverse states that the inverse of a sequence of statements is equivalent to the reversed sequence of the inverse statements.

A *statement inverter* transforms a statement into an inverse statement. The statement inverter $\mathcal{I}$ in Fig. 5 transforms a Janus statement into an inverse statement. The inversion is straightforward and performed by recursive descent over the components of a statement. An inversion is called *local* wrt a certain syntactic unit of a program iff for any given program unit the inversion can always produce an inverse unit. The statement inverter $\mathcal{I}$ performs local inversion wrt statements. The following lemma shows not only the existence an inverse statement for every Janus statement but also that an inverse statement can always be constructed by $\mathcal{I}$.

Theorem 4 (Statement inverter)**.**

$$\forall s \in \text{Stmts}[\text{Janus}].\ \sigma \vdash_{stmt} s \Rightarrow \sigma' \iff \sigma' \vdash_{stmt} \mathcal{I}[\![s]\!] \Rightarrow \sigma$$

Proof sketch. We use structural induction on s and prove $s\ \mathcal{I}[\![s]\!] \sim \epsilon$. Each base case is shown by a derivation for $s\ \breve{s}$ using the rules in Fig. 4 and the inductive case of a sequence uses Lemma 3. □

Consequently, a program written in Janus can be run efficiently in both directions because local inversion of statements can be performed on the fly, which allows the implementation of interpreters that supports computation in both directions. There is no need for a global transformation or analysis of the program. Such an interpreter can be viewed as a combination of a standard interpreter and an inverse interpreter where uncall flips the computation direction.

$$
\begin{array}{lll}
\mathcal{I}[\![x \oplus\texttt{=}\, e]\!] & = x\ \oplus'\texttt{=}\, e \quad \textit{where}\ (\oplus') = \mathcal{I}_{op}[\![\oplus]\!] & \mathcal{I}_{op}[\![\texttt{+}]\!]\ =\ \texttt{-} \\
\mathcal{I}[\![x\texttt{[}e_l\texttt{]} \oplus\texttt{=}\, e]\!] & = x\texttt{[}e_l\texttt{]}\ \oplus'\texttt{=}\, e \quad \textit{where}\ (\oplus') = \mathcal{I}_{op}[\![\oplus]\!] & \mathcal{I}_{op}[\![\texttt{-}]\!]\ =\ \texttt{+} \\
\mathcal{I}[\![\texttt{if}\ e_1\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi}\ e_2]\!] & = \texttt{if}\ e_2\ \texttt{then}\ \mathcal{I}[\![s_1]\!]\ \texttt{else}\ \mathcal{I}[\![s_2]\!]\ \texttt{fi}\ e_1 & \mathcal{I}_{op}[\![\texttt{\^{}}]\!]\ =\ \texttt{\^{}} \\
\mathcal{I}[\![\texttt{from}\ e_1\ \texttt{do}\ s_1\ \texttt{loop}\ s_2\ \texttt{until}\ e_2]\!] & = \texttt{from}\ e_2\ \texttt{do}\ \mathcal{I}[\![s_1]\!]\ \texttt{loop}\ \mathcal{I}[\![s_2]\!]\ \texttt{until}\ e_1 & \\
\mathcal{I}[\![\texttt{call}\ id]\!] & = \texttt{uncall}\ id & \\
\mathcal{I}[\![\texttt{uncall}\ id]\!] & = \texttt{call}\ id & \\
\mathcal{I}[\![\texttt{skip}]\!] & = \texttt{skip} & \\
\mathcal{I}[\![s_1\ s_2]\!] & = \mathcal{I}[\![s_2]\!]\ \mathcal{I}[\![s_1]\!] &
\end{array}
$$

Figure 5. Statement inverter for Janus statements

2.3.3 Program Inverter

A *program inverter* transforms a program PGM into an inverse program PGM^{-1}. The inversion of a Janus program can be performed by recursive descent over the program structure and inverting each procedure definition individually. The statement inverter $\mathcal{I}$ in Fig. 5 can be used to invert the body of a procedure.

$$
\begin{array}{ll}
\mathcal{I}[\![d^*\ proc_1 \cdots proc_n]\!] & =\ d^*\ \mathcal{I}[\![proc_1]\!] \cdots \mathcal{I}[\![proc_n]\!] \\
\mathcal{I}[\![\texttt{procedure}\ id\ s]\!] & =\ \texttt{procedure}\ id^{-1}\ \mathcal{I}[\![s]\!]
\end{array}
$$

For each procedure id, an inverse procedure id^{-1} is produced. In contrast to the statement inverter that does not change the source program, the program inverter produces an inverse program by replacing each procedure by its inverse. Thus, it is necessary to modify the inversion rules for `call` and `uncall` such that they make use of the inverse procedures. The two new rules are (all other rules of the statement inverter remain unchanged):

$$
\begin{array}{ll}
\mathcal{I}[\![\texttt{call}\ id]\!] & =\ \texttt{call}\ id^{-1} \\
\mathcal{I}[\![\texttt{uncall}\ id]\!] & =\ \texttt{uncall}\ id^{-1}\,.
\end{array}
$$

The program inverter $\mathcal{I}$ is inverse to itself. Applying $\mathcal{I}$ twice produces a program that is functionally equivalent to the original programs.

$$s \sim \mathcal{I}[\![\mathcal{I}[\![s]\!]]\!]$$

The sizes of the original program and its inverse are the same as the program inverter $\mathcal{I}$ does not change the size of statements. In contrast to program inversion in other (non-reversible) languages [14, 26], program inversion in Janus can be performed by local inversion. This is not possible in conventional languages because programs produced by local inversion may be nondeterministic and a global program analysis and transformations may be required to obtain a deterministic program, if this is possible at all.

3. Self-Interpreter

Here, we will present the design and implementation of self-interpreter SINT for the reversible language Janus. We are looking for an implementation that does not require a global computation history to support reversible computation. As Janus does not allow irreversible statements, SINT itself must be reversible. Moreover, SINT needs to interpret Janus expressions that are not backward deterministic. This raises the question of how to implement the irreversible operations in SINT. Here, we will describe how we met these challenges. To our knowledge, this is the first report of a self-interpreter for a reversible language. It is also the largest reversible program discussed in this paper. SINT has 230 lines of formatted Janus source code (not counting comments).

We also implemented a Janus interpreter INT in Standard ML (SML), and will discuss the differences between SINT and INT.

3.1 Self-Interpreter in Janus

The semantic rules of Janus must be implemented in SINT by reversible statements. We have shown that the rules for the execution of statements are reversible. The problem is that the evaluation of expressions is not backward deterministic (Sec. 2.3.1).

While the inversion of an assignment does not require inversion of the right-hand side expression e (cf. Fig. 5), it is nevertheless necessary to implement the forward computation of e by reversible statements. As we do not wish to use a global computation history to make SINT reversible, we resort to what can be called a ‘local Bennett’s method’ which consists of calling the eval-procedure, copying the result to a variable that is not used in the eval-procedure, and then uncalling the eval-procedure to undo all side effects the procedure may have had on the store.

To make SINT simple, we impose restrictions on the form of Janus programs. We will preprocess Janus programs into those in which any expression is a constant, a variable, an array variable with a variable as index, or a binary expression with variables as arguments. This form does not lose the expressive power since any expression can be transformed into this simplified syntax by adding temporary variables. Temporary variables must be cleared before and after computation to make the programs semantically equivalent. Tests and assertions in conditional statements and in loops are to be transformed in a similar way.

3.2 Encoding of Janus Programs

As Janus has only numerical data, we encode Janus programs into two integer arrays: (1) Array `type[]` contains the integer code of an atomic construct (e.g., integer $n_{\texttt{call}}$ for `call`) or the start and end markers of a composed construct (e.g., integers $n^{start}_{\texttt{aop}}$ and $n^{end}_{\texttt{aop}}$ bracket an assignment). (2) Array `para[]` contains an optional parameter for a syntactic type (e.g., an integer that uniquely identifies the called procedure, an index into the store for a variable, or an operator code for a binary operator) or an offset when it is a start or end marker. The offset of the start marker points forward to the location of the end marker, and the offset of the end marker points backward to the start marker. The offset is the length of an encoded statement plus 1.

Calling procedure `next` (at the bottom of the third column in Fig. 7) when program counter `pc` points to a start marker skips forward over one syntactic block by adding the offset of the start marker to `pc`. Conversely, uncalling `next` at an end marker skips backward one block. Note that one procedure implements both cases (we share the code by call and uncall). The temporary variable `next_tmp` is needed because `pc` on the left-hand side of the assignment may not occur on the right-hand side. The temporary variable is zero-cleared using the offset at the end marker.

For example, assignment `x += 5` is encoded by

type	$n^{start}_{\texttt{aop}}$	$n_{\texttt{aop}}$	$n_{\texttt{lval}}$	$n_{\texttt{con}}$	$n^{end}_{\texttt{aop}}$
para	4	$n^{\texttt{plus}}_{\texttt{aop}}$	271	5	4

where the parameters of the syntactic types are as follows: 4 is the offset of the start and end marker, $n^{\texttt{plus}}_{\texttt{aop}}$ indicates the assignment operator (`+=`), 271 is assumed to be the location of `x` in the store,

$$\mathcal{T}_{\mathit{prog}}\llbracket d^* \ (\texttt{procedure}\ \mathit{id}\ s)^+\rrbracket = (n^{\mathit{start}}_{\texttt{stmt}} \cdot \mathcal{T}_{\mathit{stm}}\llbracket s\rrbracket \cdot n^{\mathit{end}}_{\texttt{stmt}})^+$$

$$\begin{aligned}
\mathcal{T}_{\mathit{stm}}\llbracket x \oplus\texttt{=}\ e\rrbracket &= n^{\mathit{start}}_{\texttt{aop}} \cdot n_{\texttt{aop}} \cdot n_{\texttt{lval}} \cdot \mathcal{T}_{\mathit{exp}}\llbracket e\rrbracket \cdot n^{\mathit{end}}_{\texttt{aop}} \\
\mathcal{T}_{\mathit{stm}}\llbracket x\,\texttt{[}e_l\texttt{]} \oplus\texttt{=}\ e\rrbracket &= n^{\mathit{start}}_{\texttt{aop}} \cdot n_{\texttt{aop}} \cdot n^{\mathit{start}}_{\texttt{op}} \cdot n_{\texttt{op}} \cdot n_{\texttt{lval}} \cdot \\
&\quad \mathcal{T}_{\mathit{exp}}\llbracket e_l\rrbracket \cdot n^{\mathit{end}}_{\texttt{op}} \cdot \mathcal{T}_{\mathit{exp}}\llbracket e\rrbracket \cdot n^{\mathit{end}}_{\texttt{aop}} \\
\mathcal{T}_{\mathit{stm}}\llbracket \texttt{if}\ e_1\ \texttt{then}\ s_1 &= n^{\mathit{start}}_{\texttt{if}} \cdot \mathcal{T}_{\mathit{exp}}\llbracket e_1\rrbracket \cdot n^{\mathit{end}}_{\texttt{if}} \cdot \\
\quad \texttt{else}\ s_2\ \texttt{fi}\ e_2\rrbracket &\quad n^{\mathit{start}}_{\texttt{stmt}} \cdot \mathcal{T}_{\mathit{stm}}\llbracket s_1\rrbracket \cdot n^{\mathit{end}}_{\texttt{stmt}} \cdot \\
&\quad n^{\mathit{start}}_{\texttt{stmt}} \cdot \mathcal{T}_{\mathit{stm}}\llbracket s_2\rrbracket \cdot n^{\mathit{end}}_{\texttt{stmt}} \cdot \\
&\quad n^{\mathit{start}}_{\texttt{fi}} \cdot \mathcal{T}_{\mathit{exp}}\llbracket e_2\rrbracket \cdot n^{\mathit{end}}_{\texttt{fi}} \\
\mathcal{T}_{\mathit{stm}}\llbracket \texttt{from}\ e_1\ \texttt{do}\ s_1 &= n^{\mathit{start}}_{\texttt{from}} \cdot \mathcal{T}_{\mathit{exp}}\llbracket e_1\rrbracket \cdot n^{\mathit{end}}_{\texttt{from}} \cdot \\
\quad \texttt{loop}\ s_2\ \texttt{until}\ e_2\rrbracket &\quad n^{\mathit{start}}_{\texttt{stmt}} \cdot \mathcal{T}_{\mathit{stm}}\llbracket s_1\rrbracket \cdot n^{\mathit{end}}_{\texttt{stmt}} \cdot \\
&\quad n^{\mathit{start}}_{\texttt{stmt}} \cdot \mathcal{T}_{\mathit{stm}}\llbracket s_2\rrbracket \cdot n^{\mathit{end}}_{\texttt{stmt}} \cdot \\
&\quad n^{\mathit{start}}_{\texttt{until}} \cdot \mathcal{T}_{\mathit{exp}}\llbracket e_2\rrbracket \cdot n^{\mathit{end}}_{\texttt{until}} \\
\mathcal{T}_{\mathit{stm}}\llbracket \texttt{call}\ \mathit{id}\rrbracket &= n_{\texttt{call}} \\
\mathcal{T}_{\mathit{stm}}\llbracket \texttt{uncall}\ \mathit{id}\rrbracket &= n_{\texttt{uncall}} \\
\mathcal{T}_{\mathit{stm}}\llbracket \texttt{skip}\rrbracket &= n_{\texttt{skip}} \\
\mathcal{T}_{\mathit{stm}}\llbracket s_1\ s_2\rrbracket &= \mathcal{T}_{\mathit{stm}}\llbracket s_1\rrbracket \cdot \mathcal{T}_{\mathit{stm}}\llbracket s_2\rrbracket
\end{aligned}$$

$$\begin{aligned}
\mathcal{T}_{\mathit{exp}}\llbracket c\rrbracket &= n_{\texttt{con}} \\
\mathcal{T}_{\mathit{exp}}\llbracket x\rrbracket &= n_{\texttt{var}} \\
\mathcal{T}_{\mathit{exp}}\llbracket x\,\texttt{[}e\texttt{]}\rrbracket &= n^{\mathit{start}}_{\texttt{arr}} \cdot n_{\texttt{arr}} \cdot \mathcal{T}_{\mathit{exp}}\llbracket e\rrbracket \cdot n^{\mathit{end}}_{\texttt{arr}} \\
\mathcal{T}_{\mathit{exp}}\llbracket e_1 \odot e_2\rrbracket &= n^{\mathit{start}}_{\texttt{op}} \cdot n_{\texttt{op}} \cdot \mathcal{T}_{\mathit{exp}}\llbracket e_1\rrbracket \cdot \mathcal{T}_{\mathit{exp}}\llbracket e_2\rrbracket \cdot n^{\mathit{end}}_{\texttt{op}}
\end{aligned}$$

(a) Encoding of syntactic categories

$$\mathcal{P}_{\mathit{prog}}\llbracket d^* \ (\texttt{procedure}\ \mathit{id}\ s)^+\rrbracket \ = \ (\mathit{offset}(\mathcal{P}_{\mathit{stm}}\llbracket s\rrbracket))^+$$

$$\begin{aligned}
\mathcal{P}_{\mathit{stm}}\llbracket x \oplus\texttt{=}\ e\rrbracket &= \mathit{offset}(\mathcal{P}_{\mathit{aop}}\llbracket \oplus\texttt{=}\rrbracket \cdot \mathit{loc}(x) \cdot \mathcal{P}_{\mathit{exp}}\llbracket e\rrbracket) \\
\mathcal{P}_{\mathit{stm}}\llbracket x\,\texttt{[}e_l\texttt{]} \oplus\texttt{=}\ e\rrbracket &= \mathit{offset}(\mathcal{P}_{\mathit{aop}}\llbracket \oplus\texttt{=}\rrbracket \cdot \mathit{offset}(n_{\texttt{plus}} \cdot \\
&\quad \mathit{loc}(x) \cdot \mathcal{P}_{\mathit{exp}}\llbracket e_l\rrbracket) \cdot \mathcal{P}_{\mathit{exp}}\llbracket e\rrbracket) \\
\mathcal{P}_{\mathit{stm}}\llbracket \texttt{if}\ e_1\ \texttt{then}\ s_1 &= \mathit{offset}(\mathcal{P}_{\mathit{exp}}\llbracket e_1\rrbracket) \cdot \mathit{offset}(\mathcal{P}_{\mathit{stm}}\llbracket s_1\rrbracket) \cdot \\
\quad \texttt{else}\ s_2\ \texttt{fi}\ e_2\rrbracket &\quad \mathit{offset}(\mathcal{P}_{\mathit{stm}}\llbracket s_2\rrbracket) \cdot \mathit{offset}(\mathcal{P}_{\mathit{exp}}\llbracket e_2\rrbracket) \\
\mathcal{P}_{\mathit{stm}}\llbracket \texttt{from}\ e_1\ \texttt{do}\ s_1 &= \mathit{offset}(\mathcal{P}_{\mathit{exp}}\llbracket e_1\rrbracket) \cdot \mathit{offset}(\mathcal{P}_{\mathit{stm}}\llbracket s_1\rrbracket) \cdot \\
\quad \texttt{loop}\ s_2\ \texttt{until}\ e_2\rrbracket &\quad \mathit{offset}(\mathcal{P}_{\mathit{stm}}\llbracket s_2\rrbracket) \cdot \mathit{offset}(\mathcal{P}_{\mathit{exp}}\llbracket e_2\rrbracket) \\
\mathcal{P}_{\mathit{stm}}\llbracket \texttt{call}\ \mathit{id}\rrbracket &= \mathit{line}(\mathit{id}) \\
\mathcal{P}_{\mathit{stm}}\llbracket \texttt{uncall}\ \mathit{id}\rrbracket &= \mathit{line}(\mathit{id}) \\
\mathcal{P}_{\mathit{stm}}\llbracket \texttt{skip}\rrbracket &= \texttt{0} \\
\mathcal{P}_{\mathit{stm}}\llbracket s_1\ s_2\rrbracket &= \mathcal{P}_{\mathit{stm}}\llbracket s_1\rrbracket \cdot \mathcal{P}_{\mathit{stm}}\llbracket s_2\rrbracket
\end{aligned}$$

$$\begin{aligned}
\mathcal{P}_{\mathit{exp}}\llbracket c\rrbracket &= c \\
\mathcal{P}_{\mathit{exp}}\llbracket x\rrbracket &= \mathit{loc}(x) \\
\mathcal{P}_{\mathit{exp}}\llbracket x\,\texttt{[}e\texttt{]}\rrbracket &= \mathit{offset}(\mathit{loc}(x) \cdot \mathcal{P}_{\mathit{exp}}\llbracket e\rrbracket) \\
\mathcal{P}_{\mathit{exp}}\llbracket e_1 \odot e_2\rrbracket &= \mathit{offset}(\mathcal{P}_{\mathit{op}}\llbracket \odot\rrbracket \cdot \mathcal{P}_{\mathit{exp}}\llbracket e_1\rrbracket \cdot \mathcal{P}_{\mathit{exp}}\llbracket e_2\rrbracket)
\end{aligned}$$

$$\begin{aligned}
\mathcal{P}_{\mathit{aop}}\llbracket \texttt{+=}\rrbracket &= n^{\texttt{plus}}_{\texttt{aop}} & \mathcal{P}_{\mathit{op}}\llbracket \texttt{+}\rrbracket &= n_{\texttt{plus}} \\
\mathcal{P}_{\mathit{aop}}\llbracket \texttt{-=}\rrbracket &= n^{\texttt{minus}}_{\texttt{aop}} & \mathcal{P}_{\mathit{op}}\llbracket \texttt{-}\rrbracket &= n_{\texttt{minus}} \\
\mathcal{P}_{\mathit{aop}}\llbracket \texttt{\^{}=}\rrbracket &= n^{\texttt{xor}}_{\texttt{aop}} & &\vdots
\end{aligned}$$

(b) Encoding of parameters

Figure 6. Translating Janus program into a numerical encoding

and `5` is the value of the constant. The encoding of all three procedures of the Fibonacci example (Sec. 2.1) has length 2×65.

The encoding of a Janus program is defined in Fig. 6. Functions $\mathcal{T}_{\mathit{prog}}$ and $\mathcal{P}_{\mathit{prog}}$ produce the integer sequences for arrays `type[]` and `para[]`, respectively. Function $\mathit{line}(\mathit{id})$ returns the index at which the encoding of procedure id starts in `type[]` and `para[]`. Function $\mathit{loc}(x)$ takes a variable or array name and returns an index into the store where the variable or the first array element is located. Integer sequences are concatenated by an associative operator $\cdot$. Function $\mathit{offset}(s)$ concatenates the offset of an integer sequence s at the start and end of it: $\mathit{offset}(s) = (\mathit{length}(s) + 1) \cdot s \cdot (\mathit{length}(s) + 1)$.

3.3 Programming Techniques

Fig. 7 shows the source code of SINT. For simplicity, we have omitted the declaration of variables. Program counter `pc` points to the syntactic object that is currently interpreted (it is an index into `type[]` and `para[]`). Some parts have been omitted due to space limitations (marked by [snip]). Text after // until the end of the line is a comment. We now discuss several basic programming techniques that were used.

Zero-cleared copying, zero-clearing by a constant In a reversible language, as there are no destructive assignments, copying a value is done reversibly. If a variable `x` is known to be zero-cleared, it can be set to a value by an exclusive-or assignment. For example, `x ^= y` has the effect of reversibly copying the value of `y` into `x`. Similarly, if we know that a variable `x` has the same value as `y`, we can zero-clear `x` using the same exclusive-or assignment `x ^= y`.

Temporary stack How do we zero-clear a variable, say `tmp`, of which the values are not statically determined? It is possible to use an initially zero-cleared stack `tmp_stack[]` and a stack pointer `tmp_sp` for this purpose. Zero-clearing `tmp` is done by calling `alloc_tmp` which pushes the current value of `tmp` onto the stack:

```
procedure alloc_tmp
  tmp_sp += 1
  tmp <=> tmp_stack[tmp_sp]
```

To restore the original value of `tmp`, the same procedure is used: `uncall alloc_tmp`. This time, `tmp` should be zero-cleared before invoking pop. This is a call convention, and even if `tmp` is not zero-cleared the execution of the program is reversible, although the behavior will be different. Push and pop are inverse operations in Janus if we ignore stack over- and underflow.

Code sharing by call and uncall The same procedure can be used with opposite functionality by calling or uncalling it. For example, `alloc_tmp` above works as stack push by call and as stack pop by uncall. We have seen this method also in the Fibonacci example in Sec. 2.1. This is a special feature of reversible languages that allows the same code to be reused and reduces the code size.

Call-uncall When we need the result of a procedure `f` but wish to undo all other side effects the computation may have had on the store, we use the program pattern call-uncall.

```
call f
// copy the result of f
uncall f
```

If procedure `f` does not modify the variables to which the result of `f` is copied, then `uncall f` undoes all changes made by `call f`. This technique is an example of the 'local Bennett's method'.

3.3.1 Implementing Expression Evaluation

The main procedure for expression evaluation is `eval`, which evaluates an expression starting at `pc`. The result of the evaluation is returned in variable `result`, which is supposed to be zero-cleared before calling `eval`. After the evaluation, the program counter `pc` points to the next syntactic object.

Depending on the syntactic type, `type[pc]`, the corresponding operation is made. When it is a constant, the value of the constant, `para[pc]`, is copied to `result`. When it is a variable, its value in the store, `sigma[para[pc]]`, is copied to `result`. The procedure `eval_arr` for evaluating an array variable was omitted due to space constraints. When it is a binary operator, procedure `eval_bop` is called, which leaves the result in `result`.

Procedure `eval_bop` calls `eval_bop_args` to evaluate the argument expressions and then interprets the operator using the cor-

```
// Execution of statements
procedure exec
 from  type[pc] = n_stmt_start
 do    pc += 1
 loop  call exec1
 until type[pc] = n_stmt_end
 pc += 1

procedure exec1
 if type[pc]=n_aop_start
 then call exec_aop
 else if type[pc]=n_if_start
 then call exec_if
 else if type[pc]=n_from_start
 then call exec_from
 else if type[pc]=n_call
 then call exec_call
 else if type[pc]=n_uncall
 then call exec_uncall
 else if type[pc]=n_skip
 then skip
 else error
 fi type[pc]=n_skip
 fi type[pc]=n_uncall
 fi type[pc]=n_call
 fi type[pc]=n_until_end
 fi type[pc]=n_fi_end
 fi type[pc]=n_aop_end

procedure exec_call
 call exec_call_pc_swap
 call exec
 uncall next
 uncall exec_call_pc_swap

procedure exec_call_pc_swap
 tmp ^= para[pc]
 pc <=> tmp
 call alloc_tmp

procedure exec_uncall
 uncall exec_call
```

```
procedure exec_if
 pc += 1
 call eval
 if result
 then uncall eval
      pc -= 1
      call next  // to then
      call exec
      call next  // to fi
      pc += 1
      call eval
 else uncall eval
      pc -= 1
      call next  // to then
      call next  // to else
      call exec
      pc += 1
      call eval
 fi result
 uncall eval
 pc -= 1
 call next  // to end of stmt
 pc -= 1

procedure exec_aop
 call exec_aop_args
 call exec_aop_upd
 uncall exec_aop_args
 call next  // to end of stmt
 pc -= 1

procedure exec_aop_args
 pc += 1
 aop ^= para[pc]
            // copy assignment op.
 pc += 1
 call eval  // evaluation of lhs
 tmp_lhs <=> result
 call eval  // evaluation of rhs
```

```
procedure exec_from
 pc += 1
 call eval
 from result
 do   uncall eval
      pc -= 1
      call next  // to do
      call exec
      call next  // to until
      pc += 1
      call eval
 loop uncall eval
      pc -= 1
      uncall next  // to loop
      call exec
      uncall next  // to loop
      uncall next  // to do
      uncall next  // to from
      pc += 1
      call eval
 until result
 uncall eval
 pc -= 1
 call next  // to end of stmt
 pc -= 1

procedure exec_aop_upd
 if aop = n_aop_plus
 then sigma[tmp_lhs]+=result
 else if aop = n_aop_minus
 then sigma[tmp_lhs]-=result
 else if aop = n_aop_xor
 then sigma[tmp_lhs]^=result
 else error
 fi aop = n_aop_xor
 fi aop = n_aop_minus
 fi aop = n_aop_plus

procedure next
  next_tmp ^= para[pc]
  pc += next_tmp
  next_tmp ^= para[pc]
  pc += 1
```

```
// Evaluation of expressions
procedure eval
 if type[pc] = n_con
 then result ^= para[pc]
 else if type[pc] = n_var
 then result^=sigma[para[pc]]
 else if type[pc]=n_arr_start
 then call eval_arr
      call next
      pc-=1
 else if type[pc]=n_bop_start
 then call eval_bop
      call next
      pc-=1
 else error
 fi type[pc] = n_bop_end
 fi type[pc] = n_arr_end
 fi type[pc] = n_var
 fi type[pc] = n_con
 pc+=1

procedure eval_bop
 call eval_bop_args
 if op = n_plus
 then bop_tmp ^= arg1 + arg2
 else if op = n_minus
 then bop_tmp ^= arg1 - arg2
 else if op = n_xor
 then bop_tmp ^= arg1 ^ arg2
  // [snip]
 fi op = n_xor
 fi op = n_minus
 fi op = n_plus
 uncall eval_bop_args
 result <=> bop_tmp

procedure eval_bop_args
 pc += 1
 op ^= para[pc]   // copy op.
 pc += 1
 call eval
 arg1 <=> result
 call eval
 arg2 <=> result
```

Figure 7. Source code of the reversible self-interpreter SINT (for readability, some constants representing syntax are underlined)

responding built-in primitive operation. Afterward, the temporary variables `op`, `arg1` and `arg2` are reset by uncalling `eval_bop_args`. As we assume that an expression contains at most one binary operator and that the arguments are variables, no other temporary variables are necessary. In `eval_bop_args`, the operator code is first stored in `op` and the two argument expressions are then evaluated by calling `eval` (even though they are always variables).

3.3.2 Implementing Statement Execution

`Exec` executes statements encoded between `n_stmt_start` and `n_stmt_end`. Each loop executes one statement by calling `exec1` and increments `pc` by 1. Depending on the syntactic categories of statements, `exec1` dispatches the corresponding procedure.

In `exec_if`, the test expression is first evaluated, and the result is copied to `result` by zero-cleared copying. Depending on `result` the `then` or `else` branch is selected. In both branches, the effect of this evaluation is undone by uncalling `eval` (call-uncall). In the `then` branch, `pc` is set to the start of the `then` statement, the branch is executed, the `else` branch is skipped by `next`, and the assertion is evaluated. The result of the evaluation must be true. Then, the effect of the evaluation is canceled (call-uncall). Finally, we reach the end of the `if` statement by `next`. The `else` branch is similar. The `if` statement is interpreted by an `if` statement.

`Loop` statements are similar to `if` statements. We will use the underlying `loop` statements to execute `loop` statements.

Executing an assignment operation consists of four parts: evaluating the left and right expressions and remembering assignment operators (`exec_aop_args`), updating the value pointed to by the left value (`exec_aop_upd`), the effect of the calculation of the left and right values is undone by uncalling `exec_aop_args` (call-uncall), and setting `pc` to the next command by `next`. We use two auxiliary variables: `aop` for assignment operators and `tmp_lhs` for the left value. In `exec_aop_args`, an assignment operator is first remembered by `aop`. Then, the left expression is evaluated and its value is set to `tmp_lhs` by zero-cleared copying. The right expression is then evaluated. We update the store of the left value by `exec_aop_upd` depending on `aop`. Each encoded assignment operation is done by the corresponding primitive operations in the underlying interpreter INT.

To call a procedure, we remember the current `pc` and set the new program counter by `exec_call_pc_swap`, execute the statements of the called procedure bodies, return to the place where it was called by uncalling `next`, and set the old `pc` by uncalling `exec_call_pc_swap` (call-uncall). More precisely, procedure `exec_call_pc_swap` moves the value of the current `pc` to the temporary stack by `alloc_tmp` and simultaneously the new program counter will be set to the value of `para[pc]`.

Uncalling a procedure is the opposite of calling a procedure: we simply uncall `exec_call`. Again, we can share the same procedure for the opposite function. Note that this means that we shared the forward and backward implementation of all statements. This is only possible when the underlying interpreter is reversible.

In `exec_skip` the underlying `skip` is executed, which does nothing.

3.4 A Janus Interpreter in a Irreversible Language

Based on the operational semantics in Sec. 2.2, we implemented an interpreter INT for Janus in SML. As SML can run programs only in the forward direction, it is necessary to prepare two versions of interpretation for each semantic rule. We cannot share one version with call and uncall as in the self-interpreter. Most semantic rules lead directly to an efficient implementation for either computation direction, but the rules for loop and call/uncall do not. For example, we cannot make a deterministic choice between the base case and the recursive case of a loop in Fig. 4 due to central recursion.[3] As we saw in the proof of Lemma 2, this nondeterminism can be dissolved in the proof tree. For the actual implementation, we prepared two versions of the loop rules: right- and left-recursive for forward and backward computation, respectively.

As all Janus statements are locally invertible (Thm. 4), inverse interpretation of a given source program can be done on the fly in the interpreter. To implement the forward and backward semantics of each language construct, we follow the inversion rules in Fig. 5. The interpretation direction is flipped by uncall. The implementation is otherwise rather straightforward. The Janus interpreter written in SML consists of 1197 lines of formatted source code (not counting comments).

4. Experiments with Janus

4.1 Physical Simulation: Schrödinger Wave Equation

Many physical phenomena are reversible and we show the power of Janus by implementing a program SCH for discrete simulation of the Schrödinger wave equation that can be inverted by the program inverter $\mathcal{I}$ (Sec. 2.3.3) as well as run forward and backward on the Janus interpreter INT (Sec. 3.4). The Schrödinger wave equation is the fundamental equation of physics for describing quantum mechanical behavior. It is a partial differential equation that describes how the wave function of a physical system evolves over time.

Discrete Simulation Without going into more mathematical details, the rules [11] [9, Appdx. E, Eq. E.5] for updating the real parts $\mathcal{X}$ and imaginary parts $\mathcal{Y}$ of a vector at time step $n+1$ of the discrete simulation of the Schrödinger wave equation are

$$\mathcal{X}_{i,n+1} = \mathcal{X}_{i,n} + \alpha_i \mathcal{Y}_{i,n} - \epsilon(\mathcal{Y}_{i+1,n} + \mathcal{Y}_{i-1,n})$$
$$\mathcal{Y}_{i,n+1} = \mathcal{Y}_{i,n} - \alpha_i \mathcal{X}_{i,n+1} + \epsilon(\mathcal{X}_{i+1,n+1} + \mathcal{X}_{i-1,n+1})$$

where α_i and ϵ are constants. Values $\mathcal{X}_{i,n+1}$ and $\mathcal{Y}_{i,n+1}$ are uniquely determined by the rules. We impose the periodic boundary conditions $\mathcal{X}_{i,n} = \mathcal{X}_{i+128,n}$ and $\mathcal{Y}_{i,n} = \mathcal{Y}_{i+128,n}$ where 128 is the length of the vectors that we consider.

The simulation starts at time step $n = 0$ with suitable initialization of the vectors $\mathcal{X}$ and $\mathcal{Y}$. For implementation of the update rules, we note that $\mathcal{Y}_{i,n+1}$ (time step $n+1$) depends only on the previous $\mathcal{Y}_{i,n}$ (time step n) and three current $\mathcal{X}_{\_,n+1}$'s (time step $n+1$). Thus, the vector $\mathcal{X}_{n+1}$ can be computed and replace $\mathcal{X}_n$ before computing vector $\mathcal{Y}_{n+1}$. The following Janus program makes use of this property. Procedure `stepX` calculates vector $\mathcal{X}_{n+1}$ from $\mathcal{X}_n$ and $\mathcal{Y}_n$ and procedure `stepY` calculates $\mathcal{Y}_{n+1}$ from $\mathcal{X}_{n+1}$ and $\mathcal{Y}_n$. The vectors $\mathcal{X}$ and $\mathcal{Y}$ are declared as arrays `X[128]` and `Y[128]`, respectively.

```
procedure main                  procedure step
  ... // initialize arrays        call stepX
  from  n=0                       call stepY
  loop  call step
        n += 1
  until n=maxn

procedure updateX               procedure stepX
  X[i] += alpha[i] */ Y[i]        from  i=0
  X[i] -= epsilon */              loop  call updateX
          (Y[(i+1)%128] +               i += 1
           Y[(i-1)%128])          until i=128
                                  i -= 128

procedure updateY               procedure stepY
  Y[i] -= alpha[i] */ X[i]        from  i=0
  Y[i] += epsilon */              loop  call updateY
          (X[(i+1)%128] +               i += 1
           X[(i-1)%128])          until i=128
                                  i -= 128
```

That this discrete simulation can be written in Janus and that it does not require a computation history for backward computation represent constructive proof that the simulation is reversible and that computation in both directions requires only constant space regardless of how many steps the program runs in either direction. There is no information loss. This reflects the microscopic reversibility of the physical phenomenon.

If we were to implement the same simulation in a conventional programming language, such as C, two versions of the simulation would be required: the standard procedure computing forward into the future (from time 0 to n) and the inverse procedure computing backward into the past (from time n to 0). The number of the procedure nesting levels is statically bound. Hence, SCH can be run both ways for any number of steps within a constant space. There is no need to maintain a computation history, which might lead to stack overflow, or to maintain two separate versions, which may be prone to error. We obtain two for the price of one. Reverse computation has its limitations, as any computation model, but there are interesting applications where this paradigm excels and leads to novel solutions, which is why we believe exploring this this computation model is worthwhile.

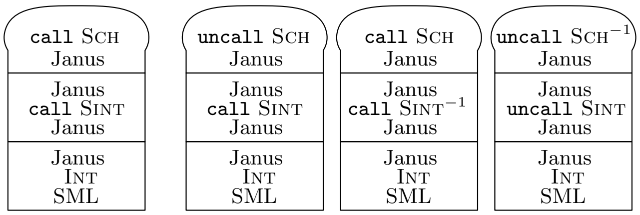
Inverse call and program inversion For each computation direction, there are two functionally equivalent ways to run a program:

$$\begin{array}{rl} \textit{forward}: & \texttt{call}\ \mathrm{PGM} \sim \texttt{uncall}\ \mathrm{PGM}^{-1} \\ \textit{backward}: & \texttt{uncall}\ \mathrm{PGM} \sim \texttt{call}\ \mathrm{PGM}^{-1} \end{array}$$

A Janus program PGM can be run forward either by calling it or by uncalling its inverse program PGM^{-1} where $\mathrm{PGM}^{-1} = \mathcal{I}[\![\mathrm{PGM}]\!]$. Similarly, a program can be run backward either by uncalling it or by calling its inverse program. Due to the reversibility of all Janus statements and on-the-fly program inversion, we expect that the running times will be the same in all four cases. This was confirmed by our experiments. For example, the running time of program SCH with $n = 100$ is 1.0 s (within 4 ms) $n = 1000$ is 9.9 s in the forward and backward directions (within 90 ms).[4]

[3] Cf. grammar $A \to bAb \mid b$ is not an LR(k)-grammar for any k.

[4] Intel(R) Pentium(R) 4 CPU 3.00GHz GNU/Linux, SML version 110.0.7.



(a) Forward (b) Backward

Figure 8. Tower of interpreters

4.2 Reversible Self-Interpretation

We have discussed a program inverter $\mathcal{I}$, a self-interpreter SINT, an interpreter INT, and a Schrödinger wave simulation program SCH. We will now illustrate the properties of Janus with a number of experiments.

Using the self-interpreter, interpreter hierarchies with any number of levels can be built.[5] The properties of Janus add a special twist to these interpreter hierarchies because SINT itself can be uncalled and, because we can invert any Janus program with program inverter $\mathcal{I}$, SINT also can be inverted. The inverted interpreter SINT^{-1} runs programs backward with `call` and forward with `uncall`.

The standard interpreter hierarchy is shown in Fig. 8(a): The program SCH is run forward on SINT by a call to SINT and a call to SINT on the underlying interpreter. Instead, both programs can be uncalled instead of called. The overall effect is that of forward computation of SCH. This is non-standard interpretation: inverse computation of inverse computation.

Non-standard interpreter hierarchies are shown in Fig. 8(b): The program SCH is invoked by an uncall, while SINT is invoked by a call. The overall effect is backward computation of SCH. In the second tower, SINT^{-1} runs SCH backward. In the third tower, SCH^{-1} is uncalled on the uncalled SINT. Again, the overall effect is backward computation of SCH. These are three of eight possible towers to run a program backward on a 2-level interpreter tower. Again, we expect that the running times will be the same in all eight cases. This was also confirmed by our experiments. Running SCH with $n = 10$ on SINT takes 11.0 s in all eight cases (within 52 ms). The constant factor overhead of self-interpretation is about 110. Running SCH with $n = 10$ on a 2-level self-interpreter (SINT on SINT) takes 1410 s within 2.8 s, with interpretive overhead of about 130. These experiments also demonstrate some of the theory of non-standard interpreter hierarchies [1].

5. Related Work

There are two main approaches when dealing with reversibility at the software level: converting existing irreversible programs into reversible programs and building new reversible programs from locally invertible components.

Generally, irreversibility of a program is caused by loss of information. This usually happens in two situations: (i) after conditional branches because the control information regarding which of the branches was used is lost, and (ii) use of non-injective primitive operations (e.g., destructive assignments) because no operation can uniquely determine the original arguments from the result. These problems can always be solved by transforming a program p into a program p' that additionally records a computation history [19], which is the typical approach implemented for undo operations. The disadvantage is that the computation history, which is needed to run a computation deterministically backward, has a size proportional to the length of the computation, and is therefore potentially unbounded in size. It was found later that, after running p' and saving its output, the entire computation history can be cleaned up by inverse computation of p' and then returning the input instead (Bennett's method [3]). The program is reversible. The variants of Bennett's method were later improved in time and space complexities [21, 20]; this approach is often used.

For the other approach, programs that are built from locally invertible primitives and control flow operators have the potential benefit of reversibility of the underlying reversible structures. Programming languages that support this approach are Janus [23], Pendulum instruction set architecture (PISA) assembly language [31][9, Appdx. B], R [9, Appdx. C and D], and SRL and ESRL [24]. We call these programming languages *reversible*, a concept discussed in detail in this paper. We can also view Gries' invertible language [15] as belonging to this category.[6]

The concept of reversibility has been defined for several computation models, e.g., reversible Turing machines [3, 20], reversible combinatory logic [7], reversible Boolean logic circuits [12, 27], and reversible finite automata [28, 22]. It was shown that any irreversible automaton can be simulated by a reversible automaton [29]. One of the original motivations for reversible computing was the reduction of energy dissipation during computation. Other potential applications of reversible computing include parallel computation [5], physical simulation [9], debugging [2][9, Ch. 10], and garbage collection [2].

A related concept is bidirectional languages, which are designed for the view update problem [18, 8]. These languages also have forward and backward semantics. However, in contrast to reversible languages, these languages are not necessarily locally invertible.

6. Conclusions

We formalized the structured programming language Janus and proved its reversibility. We showed that all statements in the language are forward and backward deterministic and that local inversion is always sufficient to produce inverse programs. We identified this as the key for deciding about the reversibility of programming languages and argued that this transformational property is what sets reversible languages apart from conventional languages. We identified program reversibility as an instance of the general concept of program inversion and argued that it is the reversibility that allows the implementation of efficient interpreters for forward and backward computation because they can perform program inversion on the fly. In this paper, we connected the concept of reversibility to program inversion and inverse computation.

Moreover, we found that reversible languages and their computing devices are a computing paradigm that is worthwhile to study in its own right, because of the number of potential theoretical and practical implications of this specialized paradigm, not only because of the promise it holds for reducing energy dissipation of the computing process. We designed and implemented a self-interpreter for Janus and a reversible program for Schrödinger wave equations. These implementations are also interesting regarding the methodology for reversible programming. The reversible self-interpreter appears to be the first such program described in the literature on reversible computing.

Future work must aim at identifying more basic programming techniques as not all irreversible algorithms can simply be rewritten

[5] Given a k-level interpreter tower, there are $2^{2(k+1)}$ possibilities to run the tower ($k \geq 0$).

[6] Gries presented a guarded commands language annotated with assertions to facilitate program inversion. We view these annotations constructively and as part of the guarded commands language (in the same way as assertions are part of the control flow operators in Janus).

statement-by-statement as reversible algorithms. More work will be needed to gain experience on a larger scale.

Our implementations are efficient in the sense that they do not require a computation history to make their computation processes reversible. We performed a number of experiments with nonstandard interpreter hierarchies and inverted their computation on different levels of the interpreter hierarchy. Clearly, a goal for further work on Janus is the development of suitable language concepts that allow local variables, true recursion, richer data types, and I/O features, while remaining within the reversible programming paradigm. Another aim will be the development of a compiler from Janus into reversible machine code that will allow the execution of Janus programs directly on suitable abstract machines and microprocessors.

Acknowledgments

The authors thank Holger Axelsen, Torben Mogensen, Hiroyuki Ozaki, and the anonymous reviewers for their insightful comments. This work was supported in part by the Circle for the Promotion of Science and Engineering, Research Fellowships of the Japan Society for the Promotion of Science for Young Scientists.

References

[1] S. M. Abramov and R. Glück. Combining semantics with non-standard interpreter hierarchies. In *FST TCS*, volume 1974 of *LNCS*, pages 201–213. Springer-Verlag, 2000.

[2] H. G. Baker. NREVERSAL of fortune - the thermodynamics of garbage collection. In *the Int'l Workshop on Memory Management*, pages 507–524. Springer-Verlag, 1992.

[3] C. H. Bennett. Logical reversibility of computation. *IBM J. Res. Dev.*, 17(6):525–532, 1973.

[4] C. H. Bennett. Notes on the history of reversible computation. *IBM J. Res. Dev.*, 32(1):16–23, 1988.

[5] C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Trans. Model. Comput. Simul.*, 9(3):224–253, 1999.

[6] A. De Vos and Y. Van Rentergem. Reversible computing: from mathematical group theory to electronical circuit experiment. In *2nd Conf. on Computing Frontiers*, pages 35–44. ACM Press, 2005.

[7] A. Di Pierro, C. Hankin, and H. Wiklicky. Reversible combinatory logic. *Mathematical. Structures in Comp. Sci.*, 16(4):621–637, 2006.

[8] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. *ACM Trans. Prog. Lang. Syst.*, 2006.

[9] M. P. Frank. *Reversibility for Efficient Computing*. PhD thesis, EECS Dept., MIT, 1999.

[10] M. P. Frank. Introduction to reversible computing: motivation, progress, and challenges. In *2nd Conf. on Computing Frontiers*, pages 385–390. ACM Press, 2005.

[11] E. Fredkin. Feynman, Barton and the reversible Schrödinger difference equation. In A. J. G. Hey, editor, *Feynman and Computation: Exploring the Limits of Computers*, pages 337–348. Perseus Books, 1999.

[12] E. Fredkin and T. Toffoli. Conservative Logic. *Internat. J. Theor. Phy.*, 21:219–253, 1982.

[13] R. Glück and M. Kawabe. Derivation of deterministic inverse programs based on LR parsing. In *FLOPS*, pages 291–306, 2004.

[14] R. Glück and M. Kawabe. Revisiting an automatic program inverter for Lisp. *SIGPLAN Not.*, 40(5):8–17, 2005.

[15] D. Gries. *The Science of Programming*, chapter 21 Inverting Programs. Texts and Monographs in Computer Science. Springer-Verlag, 1981.

[16] J. S. Hall. A reversible instruction set architecture and algorithms. In *Physics and Computation*, pages 128–134. IEEE Press, 1994.

[17] H. M. Hasan Babu and A. R. Chowdhury. Design of a compact reversible binary coded decimal adder circuit. *J. Syst. Archit.*, 52(5):272–282, 2006.

[18] Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *PEPM*, pages 178–189. ACM Press, 2004.

[19] R. Landauer. Irreversibility and heat generation in the computing process. *IBM J. Res. Dev.*, 5(3):183–91, 1961.

[20] K.-J. Lange, P. McKenzie, and A. Tapp. Reversible space equals deterministic space. *J. Comput. and Sys. Sci.*, 60(2):354–367, 2000.

[21] R. Y. Levine and A. T. Sherman. A note on Bennett's time space tradeoff for reversible computation. *SIAM J. Comput.*, 19(4):673–677, 1990.

[22] S. Lombardy. On the construction of reversible automata for reversible languages. In *Int'l Coll. Automata, Languages, and Programming*, pages 170–182. Springer-Verlag, 2002.

[23] C. Lutz. Janus: a time-reversible language. A letter to Landauer. `http://www.cise.ufl.edu/~mpf/rc/janus.html`, 1986.

[24] A. B. Matos. Linear programs in a simple reversible language. *Theor. Comput. Sci.*, 290(3):2063–2074, 2003.

[25] J. McCarthy. The inversion of functions defined by Turing machines. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 177–181. Princeton University Press, 1956.

[26] T. Æ. Mogensen. Semi-inversion of guarded equations. In *GPCE*, volume 3676 of *LNCS*, pages 189–204, 2005.

[27] K. Morita, T. Ogiro, K. Tanaka, and H. Kato. Classification and universality of reversible logic elements with one-bit memory. In *MCU*, volume 3354 of *LNCS*, pages 245–256. Springer, 2004.

[28] J.-E. Pin. On the language accepted by finite reversible automata. In *Int'l Coll. Automata, Languages, and Programming*, volume 267 of *LNCS*, pages 237–249. Springer-Verlag, 1987.

[29] T. Toffoli. Computation and construction universality of reversible cellular automata. *J. Comput. Sys. Sci.*, 15:213–231, 1977.

[30] T. Toffoli. Reversible computing. In *Int'l Coll. Automata, Languages, and Programming*, pages 632–644. Springer-Verlag, 1980.

[31] C. J. Vieri. *Reversible computer engineering and architecture*. PhD thesis, MIT, 1999.

[32] P. Vitányi. Time, space, and energy in reversible computing. In *2nd Conf. on Computing Frontiers*, pages 435–444. ACM Press, 2005.