Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 253 (2010) 71–81
www.elsevier.com/locate/entcs

Reversible Computation and Reversible Programming Languages

Tetsuo Yokoyama [1]

Department of Software Engineering, Nanzan University
Seirei-cho 27, Seto city, Aichi 489-0863, Japan

Abstract

A reversible programming language supports deterministic forward and backward computation. This tutorial focuses on a high-level reversible programming language Janus. In common with other programming paradigms, reversible programming has its own programming methodology. Janus is simple, yet powerful, and its constructs can serve as a model for designing reversible languages in general.

Keywords: Reversible computing, Reversible programming languages

1 Introduction

Conventional computing models such as Turing machines and random access machines (RAMs) destroy information at each computational step. The symbol written on the tape in the previous state will be overwritten by the new symbol, and the value written on the registers will be updated into the new one. At the first sight, we tend to think the destruction of information is necessary to computation. However, it was shown by Landauer that any irreversible computation can be simulated by reversible computation by adding the extra storage to remember the history of computation [16]. Moreover, this garbage information can be erased by its inverse computation [2]. Thus, in theory we can simulate any irreversible computation with reversible computation provided that a given storage is infinite.

When a conventional computation is physically performed information destruction has a physical cost in the form of heat dissipation. Conversely, if no bit is erased during computaion, in theory there is no lower bound of heat dissipation for the computation. Therefore, the research of reversible computing has some potential

[1] Email: tetsuo@se.nanzan-u.ac.jp

[2] This work is partly supported by EPSRC grant EP/G039550/1, JST CREST and Nanzan University Pache Research Subsidy I-A-2 for the 2009 academic year.

doi:10.1016/j.entcs.2010.02.007

applications such as the low-power CMOS and quantum computing. Note that any quantum computing is necessary to be reversible.

This tutorial focuses on a high-level reversible programming language Janus. In common with other programming paradigms, reversible programming has its own programming methodology. We define the language and give its syntax and operational semantics.

2 The Reversible Language Janus

The imperative language Janus appears to be the first reversible structured programming language: it was invented by Lutz and Derby [17], but remained unpublished for two decades. The language presented here extends our original formalization [32] and has been presented in [30]. Janus is simple, yet powerful, and its constructs can serve as a model for designing reversible languages in general. The main difference from conventional programming languages is that all assignments and control constructs are purely reversible, and the language's inverse semantics can be accessed by uncalling procedures (i.e., executing them backward).

2.1 Example Program: Fibonacci Pairs

To provide a flavor of reversible programming, we show a Janus procedure for computing *Fibonacci pairs*. Given an integer `n`, the procedure `fib` computes the (`n+1`)-th and (`n+2`)-th Fibonacci number. For example, the Fibonacci pair for $\texttt{n} = 4$ is $(5, 8)$. Returning a pair of Fibonacci numbers makes the otherwise non-injective Fibonacci function injective. Variables `n`, `x1`, `x2` are initially set to zero. Parameter passing is pass-by-reference.

```
procedure fib(int x1,int x2,int n)
   if n=0 then x1 += 1
               x2 += 1
          else n -= 1
               call fib(x1,x2,n)
               x1 += x2
               x1 <=> x2
   fi x1=x2

procedure fib_fwd(int x1,int x2,int n)
   n += 4
   call fib(x1,x2,n)     // forward execution

procedure fib_bwd(int x1,int x2,int n)
   x1 += 5
   x2 += 8
   uncall fib(x1,x2,n)   // backward execution
```

Syntax Domains

$prog \in$ Progs	$s \in$ Stms	$d \in$ Vdecs	$\odot \in$ ModOps
$p \in$ Procs	$e \in$ Exps	$t \in$ Types	$\otimes \in$ Ops
$q \in$ PIds	$x \in$ Vars	$c \in$ Cons	

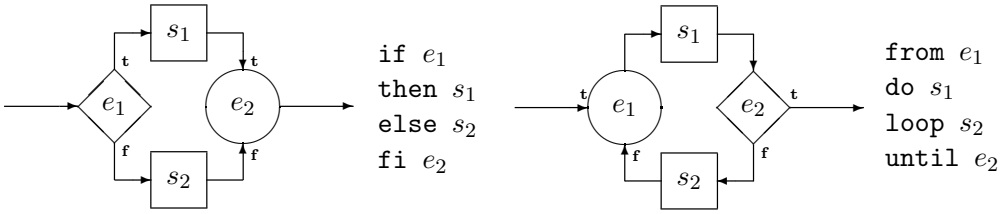
Grammar

$prog$ ::=	$p_{main}\ p^*$	Janus program
d ::=	$x \mid x$`[`c`]`	scalar and array
t ::=	`int` $\mid$ `stack`	data types
p_{main} ::=	`procedure main () (int` $d \mid$ `stack` $x)^*\ s$	main procedure
p ::=	`procedure` q`(`$t\ x$`,` $\ldots$ `,`$t\ x$`)` s	procedure definition
s ::=	$x\ \odot$`=` $e \mid x$`[`e`]` $\odot$`=` e	assignments
	`if` e `then` s `else` s `fi` $e \mid$	conditional
	`from` e `do` s `loop` s `until` $e \mid$	loop
	`push(`x`,`x`)` $\mid$ `pop(`x`,`x`)` $\mid$	stack modification
	`local` $t\ x$ `=` $e\ \ s\ \ $ `delocal` $t\ x$ `=` $e \mid$	local variable block
	`call` q`(`x`,` $\ldots$ `,`x`)` $\mid$ `uncall` q`(`x`,` $\ldots$ `,`x`)` $\mid$	procedure invocation
	`skip` $\mid s\ s$	statement sequence
e ::=	$c \mid x \mid x$`[`e`]` $\mid e \otimes e \mid$ `empty(`x`)` $\mid$ `top(`x`)` $\mid$ `nil`	expression
c ::=	`-2147483648` $\mid \cdots \mid$ `0` $\mid$ `1` $\mid \cdots \mid$ `2147483647`	integer constant $(-2^{31}$ to $2^{31}-1)$
$\odot$ ::=	`+` $\mid$ `-` $\mid$ `^`	operator
$\otimes$ ::=	$\odot \mid$ `*` $\mid$ `/` $\mid$ `%` $\mid$ `&` $\mid$ `|` $\mid$ `&&` $\mid$ `||` $\mid$	operator
	`<` $\mid$ `>` $\mid$ `=` $\mid$ `!=` $\mid$ `<=` $\mid$ `>=`	

Fig. 1. Syntax of Janus

The implementation of procedure `fib` looks conventional, but consists only of reversible assignments (`+=`, `-=`) and a reversible conditional with entry and exit test (`if`...`fi`). Here, `x1 <=> x2` swaps two values. [3]

As a result, procedure `fib` is reversible. It can be invoked with either its standard or inverse semantics. Setting `n` to 4 and calling `fib` in procedure `fib_fwd` (assuming variables `x1` and `x2` are set to zero), computes the Fibonacci pair `x1` = 5 and `x2` = 8. Setting `x1` to 5 and `x2` to 8 and *uncalling* `fib` in procedure `fib_bwd`, computes the pair's index `n` = 4. This shows how the same procedure definition can be used for deterministic forward and backward computation.



(a) Conditional (b) Loop

Fig. 2. Reversible structured control flow

2.2 The Language

A Janus *program* consists of a main procedure followed by a sequence of procedure definitions (Fig. 1).[4] Reversible statements are the basic constructs of Janus. A *statement* is a reversible assignment, a reversible control flow operator (conditional, loop), a stack operation (push, pop), a local variable block, a procedure invocation (call, uncall), a skip or a statement sequence. The main procedure consists of variable declarations and a statement, and has no parameters. A *variable declaration* defines an integer variable, a one-dimensional integer array, or an integer stack. Arrays are indexed by integers starting from zero. The type primitives are 32-bit signed integers and stacks. Variables and array elements are initially zero-cleared and stacks are empty. To keep things simple there are no global variables. The logic value *true* is represented by any non-zero integer and *false* by zero.

2.2.1 Assignments and Expressions

A *reversible assignment* updates an integer variable or an array element. 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 $\odot$ (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.

The *expression* on the right-hand side of an assignment or in a control-flow predicate can be a constant, a variable, an indexed variable, a binary expression, an is-empty predicate for stacks, the top element of a stack, or an empty stack. A *binary operator* $\otimes$ is an arithmetic (`+`,`-`,`*`,`/`,`%`), bitwise (`&`,`|`,`^`), logical (`&&`,`||`), or relational operator (`<`,`>`,`=`,`!=`,`<=`,`>=`). Note that a logical binary operator regards a zero operand as false and any non-zero operand as true, interprets its operands as either false or true, and evaluates to 1 (true) or 0 (false). A binary bitwise operation performs the logical operation on each bit position of its operands.

2.2.2 Structured Control Flow

Reversible control flow requires entry and exit predicates (pre- and post-conditions). A *reversible conditional* has two predicates (Fig. 2(a)): a test at the entry (e_1) and an assertion at the exit (e_2) of the conditional. Predicate e_2 must be *true* when the control flow reaches the assertion along the true-edge (labeled **t**) and *false* when the control flow reaches the assertion along the false-edge (labeled **f**); otherwise the operation is undefined (abnormal stop). Statements s_1 and s_2 are the then- and else-branches, respectively. The assertion (marked with a circle in the diagram to distinguish it from a test) makes the conditional backward deterministic; in the backward direction an assertion acts as a test and a test as an assertion. Assertions are an operational part of a programs in the same way as tests.

A *reversible loop* has two predicates (Fig. 2(b)): an assertion at the entry (e_1) and a test at the exit of the loop (e_2). Initially, assertion e_1 must be *true* and then s_1 is executed. The loop terminates if test e_2 is *true*; otherwise, s_2 is executed, after which e_1 must be *false*. The assertion is only initially *true*. The loop is repeated as long as assertion and test are *false*, and terminates when the test is *true*. This makes the loop backward deterministic.

2.2.3 Dynamic Allocation of Storage

A *stack* is an abstract data type that is equipped with the operation `push(c,s)`, which adds element `c` to stack `s` and zero-clears `c`, and the operation `pop(c,s)`, which moves one element from stack `s` to a zero-cleared `c`. Popping an element from an empty stack, or into a non-zero-cleared variable is undefined. Operations `push(c,s)` and `pop(c,s)` are inverse to each other. In expressions the predicate `empty(s)` tests whether stack `s` is empty, `top(s)` returns the value of the topmost element on stack `s`, and `nil` is the empty stack.

A *local variable block* consists of a local variable allocation, a statement, and local variable deallocation. A local variable block allocates memory for local variables and initializes them with the values of the corresponding expressions, and a *variable deallocation* specified by `delocal` releases the memory, where the value of the variable must meet the value of a given expression. Variable x of type t is allocated and the value of e_1 is assigned to x. Under the new store, statement s is executed. The value of x should now be equal to the value of e_2, and can be deallocated (otherwise, the behavior is undefined). If x is already in scope on entry, it is hidden and a fresh x is used during the local block structure. As in the assignment operations, x must not occur in e_1 and e_2. Local variables are allocated and deallocated only in this structured way.

2.2.4 Procedure Calls and Uncalls

Procedure calls provide an elegant and convenient way to access the inverse semantics of Janus and to run a procedure backward. A *procedure call* executes the

[3] The swap operator `x1 <=> x2` is syntactic sugar for the statement sequence `x1 ^= x2; x2 ^= x1; x1 ^= x2`.

[4] Some of the original operators [17] were changed into C-like notation.

$$
\begin{array}{rcl}
v \in \text{Vals} & = & \mathbb{Z}_{32} \cup \text{Stack}_{\mathbb{Z}} \\
l \in \text{Lvals} & = & \{\ \texttt{a}, \texttt{b}, \ldots, \texttt{a[0]}, \texttt{a[1]}, \ldots, \texttt{b[0]}, \ldots\ \} \\
\sigma \in \text{Stores} & = & \text{Lvals} \rightharpoonup \text{Vals} \\
\Gamma \in \text{Pmaps} & = & \text{PIds} \rightharpoonup \text{Procs}
\end{array}
$$

Fig. 3. Semantic values

procedure body in the local store of formal parameter variables. A *procedure uncall* invokes inverse computation of the procedure. All parameters are passed by reference. As usual, the number of parameters in a call must correspond to the number designated in the procedure declaration and the types of the actual parameters should meet those of the formal parameters. The actual parameters must be variable names in the scope of the procedure invocation. To avoid problems with aliasing, we prohibit passing the same reference to more than a single parameter.

2.3 Operational Semantics

The semantics of Janus programs is specified by the rules shown in Fig. 4. The operational semantics have three main judgments: the evaluation of expressions, the execution of statements and execution of programs. Before going into details, we shall briefly describe the semantic values (Fig. 3) along with some notation.

Preliminaries
Let $\mathbb{Z}_{32}$ designate the set of 32-bit signed integers. A *value* v is an integer in $\mathbb{Z}_{32}$ or an integer stack in $\text{Stack}_{\mathbb{Z}}$. Integer stacks are inductively defined by

$$\text{Stack}_{\mathbb{Z}} = \{nil\} \cup \{hd :: tl \mid hd \in \mathbb{Z}_{32} \wedge tl \in \text{Stack}_{\mathbb{Z}}\}$$

where nil designates the empty stack and $hd :: tl$ designates a non-empty stack with top element $hd \in \mathbb{Z}_{32}$ and remainder stack $tl \in \text{Stack}_{\mathbb{Z}}$. A *left-value* l is a variable name, or an indexed 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 $\sigma(l)$.

Update $\sigma[l \mapsto v]$ denotes the same mapping as σ except that l maps to v. We write a syntactic substitution replacing $x_1, \ldots, x_n$ with $e_1, \ldots, e_n$ as $[e_1/x_1, \ldots, e_n/x_n]$, which is defined on expressions and statements. A *procedure map* Γ is a partial function from identifiers to procedure definitions.

Evaluation of Expressions
A 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. Evaluation of expressions does not cause side effects on the store. Some definitions are (others

Evaluation of Expressions

$$\sigma \vdash_{expr} c \Rightarrow [\![c]\!] \ \textsc{Con} \qquad \sigma \vdash_{expr} \texttt{nil} \Rightarrow nil \ \textsc{Nil} \qquad \sigma \vdash_{expr} x \Rightarrow \sigma(x) \ \textsc{Var} \qquad \frac{\sigma \vdash_{expr} e \Rightarrow v}{\sigma \vdash_{expr} x\texttt{[}e\texttt{]} \Rightarrow \sigma(x\texttt{[}v\texttt{]})} \ \textsc{Arr}$$

$$\frac{\sigma \vdash_{expr} e_1 \Rightarrow v_1 \quad \sigma \vdash_{expr} e_2 \Rightarrow v_2 \quad [\![\otimes]\!](v_1, v_2) = v}{\sigma \vdash_{expr} e_1 \otimes e_2 \Rightarrow v} \ \textsc{BinOp} \qquad \sigma[x \mapsto v_{hd} :: v_{tl}] \vdash_{expr} \texttt{top(}x\texttt{)} \Rightarrow v_{hd} \ \textsc{Top}$$

$$\sigma[x \mapsto nil] \vdash_{expr} \texttt{empty(}x\texttt{)} \Rightarrow 1 \ \textsc{EmptyTrue} \qquad \sigma[x \mapsto v_{hd} :: v_{tl}] \vdash_{expr} \texttt{empty(}x\texttt{)} \Rightarrow 0 \ \textsc{EmptyFalse}$$

Execution of Statements

$$\frac{\sigma \vdash_{expr} e \Rightarrow v \quad v_2 = [\![\odot]\!](v_1, v)}{\sigma[x \mapsto v_1] \vdash_{stmt} x \odot\texttt{=} e \Rightarrow \sigma[x \mapsto v_2]} \ \textsc{AssVar} \qquad \frac{\sigma \vdash_{expr} e_l \Rightarrow v_l \quad \sigma \vdash_{expr} e \Rightarrow v \quad v_2 = [\![\odot]\!](v_1, v)}{\sigma[x\texttt{[}v_l\texttt{]} \mapsto v_1] \vdash_{stmt} x\texttt{[}e_l\texttt{]} \odot\texttt{=} e \Rightarrow \sigma[x\texttt{[}v_l\texttt{]} \mapsto v_2]} \ \textsc{AssArr}$$

$$\frac{\sigma \vdash_{expr} e_1 \not\Rightarrow 0 \ \sigma \vdash_{stmt} s_1 \Rightarrow \sigma' \ \sigma' \vdash_{expr} e_2 \not\Rightarrow 0}{\sigma \vdash_{stmt} \texttt{if } e_1 \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ fi } e_2 \Rightarrow \sigma'} \ \textsc{IfTrue} \qquad \frac{\sigma \vdash_{expr} e_1 \Rightarrow 0 \ \sigma \vdash_{stmt} s_2 \Rightarrow \sigma' \ \sigma' \vdash_{expr} e_2 \Rightarrow 0}{\sigma \vdash_{stmt} \texttt{if } e_1 \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ fi } e_2 \Rightarrow \sigma'} \ \textsc{IfFalse}$$

$$\frac{\sigma \vdash_{expr} e_1 \not\Rightarrow 0 \quad \sigma \vdash_{stmt} s_1 \Rightarrow \sigma' \quad \sigma' \vdash_{loop} (e_1, s_1, s_2, e_2) \Rightarrow \sigma''}{\sigma \vdash_{stmt} \texttt{from } e_1 \texttt{ do } s_1 \texttt{ loop } s_2 \texttt{ until } e_2 \Rightarrow \sigma''} \ \textsc{LoopMain} \qquad \frac{\sigma \vdash_{expr} e_2 \not\Rightarrow 0}{\sigma \vdash_{loop} (e_1, s_1, s_2, e_2) \Rightarrow \sigma} \ \textsc{LoopBase}$$

$$\frac{\sigma \vdash_{expr} e_2 \Rightarrow 0 \quad \sigma \vdash_{stmt} s_2 \Rightarrow \sigma' \quad \sigma' \vdash_{expr} e_1 \Rightarrow 0 \quad \sigma' \vdash_{stmt} s_1 \Rightarrow \sigma'' \quad \sigma'' \vdash_{loop} (e_1, s_1, s_2, e_2) \Rightarrow \sigma'''}{\sigma \vdash_{loop} (e_1, s_1, s_2, e_2) \Rightarrow \sigma'''} \ \textsc{LoopRec}$$

$$\frac{}{\sigma[x \mapsto v_{hd}, x_s \mapsto v_{tl}] \vdash_{stmt} \texttt{push(}x\texttt{,}x_s\texttt{)} \Rightarrow \sigma[x \mapsto 0, x_s \mapsto v_{hd} :: v_{tl}]} \ \textsc{Push} \qquad \frac{\sigma' \vdash_{stmt} \texttt{push(}x\texttt{,}x_s\texttt{)} \Rightarrow \sigma}{\sigma \vdash_{stmt} \texttt{pop(}x\texttt{,}x_s\texttt{)} \Rightarrow \sigma'} \ \textsc{Pop}$$

$$\frac{\begin{array}{c}\Gamma(q) = \texttt{procedure } q\texttt{(}t_1\ y_1\texttt{,}\ldots\texttt{,}t_n\ y_n\texttt{)}\ s \\ \sigma \vdash_{stmt} s[x_1/y_1, \ldots, x_n/y_n] \Rightarrow \sigma' \\ \sigma \vdash_{stmt} \texttt{call } q\texttt{(}x_1\texttt{,}\ldots\texttt{,}x_n\texttt{)} \Rightarrow \sigma'\end{array}}{\sigma' \vdash_{stmt} \texttt{call } q\texttt{(}x_1\texttt{,}\ldots\texttt{,}x_n\texttt{)} \Rightarrow \sigma} \ \textsc{Call} \qquad \frac{\sigma' \vdash_{stmt} \texttt{call } q\texttt{(}x_1\texttt{,}\ldots\texttt{,}x_n\texttt{)} \Rightarrow \sigma}{\sigma \vdash_{stmt} \texttt{uncall } q\texttt{(}x_1\texttt{,}\ldots\texttt{,}x_n\texttt{)} \Rightarrow \sigma'} \ \textsc{Uncall}$$

$$\sigma \vdash_{stmt} \texttt{skip} \Rightarrow \sigma \ \textsc{Skip} \qquad \frac{\sigma \vdash_{stmt} s_1 \Rightarrow \sigma' \quad \sigma' \vdash_{stmt} s_2 \Rightarrow \sigma''}{\sigma \vdash_{stmt} s_1\ s_2 \Rightarrow \sigma''} \ \textsc{Seq}$$

$$\frac{\begin{array}{c}\sigma \vdash_{expr} e \Rightarrow v \quad \sigma' \vdash_{expr} e' \Rightarrow v' \quad x_{new} \notin \sigma \cup \sigma' \\ \sigma[x_{new} \mapsto v] \vdash_{stmt} s[x_{new}/x] \Rightarrow \sigma'[x_{new} \mapsto v']\end{array}}{\sigma \vdash_{stmt} \texttt{local } t\ x\texttt{=}e\ \ s\ \ \texttt{delocal } t\ x\texttt{=}e' \Rightarrow \sigma'} \ \textsc{LocMem}$$

Execution of Programs

$$\frac{\begin{array}{c}p_{main} = \texttt{procedure main() } t_1\ d_1 \cdots t_n\ d_n\ s \quad \Gamma = gen(p_1 \cdots p_k) \\ \{d_1 \mapsto init_{t_1}, \ldots, d_n \mapsto init_{t_n}\} \vdash^{\Gamma}_{stmt} s \Rightarrow \sigma\end{array}}{\vdash_{prog} p_{main}\ p_1 \cdots p_k \Rightarrow \sigma} \ \textsc{Main}$$

Fig. 4. Operational semantics of Janus programs

are similar):

$$[\![\texttt{+}]\!](v_1, v_2) = v_1 +_{32} v_2 \qquad [\![\texttt{=}]\!](v_1, v_2) = \begin{cases} 0 & \text{if } v_1 \neq v_2 \\ 1 & \text{if } v_1 = v_2 \end{cases}$$
$$[\![\texttt{\^{}}]\!](v_1, v_2) = v_1 \; xor \; v_2$$

The subscript of binary operator in the form $\otimes_{32}$ defines modular arithmetic on $\mathbb{Z}_{32}$ such that $v_1 \otimes_{32} v_2 \stackrel{\text{def}}{=} ((v_1 \otimes v_2) + 2^{31} \bmod 2^{32}) - 2^{31}$. *xor* is bitwise exclusive-or on the 32-bit binary representation of data. For example, adding one to $2^{32} - 1$ constitutes an overflow $\vdash$ `2147483647 + 1` $\Rightarrow -2147483648$ and since the least significant bit representation of 2 and 5 are 10 and 101, we have $\vdash$ `2 ^ 5` $\Rightarrow 6$.

Execution of Statements
A judgment

$$\sigma \vdash^{\Gamma}_{stmt} s \Rightarrow \sigma'$$

defines the meaning of statements where σ and σ' are stores, Γ a procedure map, and s a statement. As the procedure map Γ is fixed for a given program, we shall usually omit it from the judgment form, writing simply $\vdash_{stmt}$. 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.

The meaning of an assignment is defined by the rules AssVar and AssArr. We distinguish between assignments to integer variables and to array variables. The assignment operator $\odot$`=` stands for one of `+=`, `-=` and `^=`.

The meaning of a conditional is defined by the rules IfTrue and IfFalse, and which rule applies depends on the value of e_1 and e_2 (cf. Fig. 2). We use $\sigma \vdash_{expr} e \not\Rightarrow 0$ for $\sigma \vdash_{expr} e \Rightarrow v$, where $v \neq 0$. The meaning of a loop is defined by a main rule for the entry of the loop, a rule for exiting, and a rule for iteration. Rule LoopMain requires assertion e_1 when entering a loop (cf. Fig. 2). The statement sequence $s_1\, s_2 \cdots s_2\, s_1$ that is executed by the loop is specified by the two judgments indexed by *loop*. The execution exits the loop if the test e_2 is true following rule LoopBase, otherwise the loop continues by rule LoopRec.

A procedure call executes the procedure body under the current store, where the formal parameters $x_1, \ldots, x_n$ appearing in the body are replaced by the actual parameters $y_1, \ldots, y_n$. We use pass-by-reference parameter passing mode. The rule Call relates an input store σ with an output store σ' following execution of the procedure body. Conversely, a procedure uncall relates σ and σ' with the opposite stores of a call: the input store σ of a call is the output store of an uncall, and vice versa. Thus, an uncall effectively reverses the direction of execution for the procedure body.

This is an important mechanism of reversible languages, and capturing the concept by switching input and output store for inverse constructs is a promising semantics technique. We use the same technique in defining a *pop* as the inverse of a *push* (cf. rules Push and Pop).

The Skip rule leaves the store unchanged. The execution of a statement sequence is defined by rule Seq. For local variable allocation in rule LocMem, we add a fresh

variable x_{new} to the store. Note that the arbitrary choice of the name of x_{new} does not affect the determinism of the rule. The store size does not change over the local block structure, in the sense that $dom(\sigma) = dom(\sigma')$.

Execution of Programs
A judgment

$$\vdash_{prog} prog \Rightarrow \sigma$$

defines the meaning of programs where *prog* is a program and σ a store. We say that executing program *prog* gives the output σ. Rule MAIN defines the execution of programs, where the `main` procedure body is executed with store initialization values $init_{\texttt{int}} = 0, init_{\texttt{stack}} = nil$. If $t_i\ d_i$ is an array declaration `int` x`[`c`]`, each cell $x\texttt{[0]}, \ldots, x\texttt{[}c-1\texttt{]}$ is initialized to $init_{\texttt{int}} = 0$. Function *gen* generates a procedure map from a list of procedure declarations.

2.4 Power of Reversible Languages

Reversible programming languages are sufficiently different from classical programming languages, so that it is not obvious that the results from classical programming languages hold in the reversible paradigm. Since reversible languages cannot compute non-injective functions, Janus is not universal. However, Janus with unbounded size stacks is r-Turing complete [31], meaning that any reversible Turing machine (RTM) can be simulated without returning the irrelevant garbage information. Here, an RTM is a Turing machine with forward and backward deterministic transition rules. As RTM does [14], the reversible language can compute all the injective functions computable by Turing machines. If we allow the garbage output extraneous to the intended output, any irreversible function can be embedded into reversible programs [16].

In classical programming languages, it is well known that structured and unstructured programs have the same expressive power and any unstructured programs can be transformed into a structured programs of the same behavior [4]. This also holds in reversible programming languages and any unstructured reversible programs can be transformed into structured Janus programs (the Structured Reversible Program Theorem) [31].

A Janus program without unbounded size stacks is guaranteed to be terminating [31]. Note that this does not always hold in classical programming languages and the halting problem is undecidable over classical Turing machine.

Because of backward determinism, in reversible languages, program inversion is realized by lightweight local inversion and has unique solution [32].

Each programming paradigm has its own methodology. Reversible programming also has its own techniques [30,32]. For example, Janus can implement Janus interpreter and the tower of this reversible self-interpreter constitutes non-standard hierarchy. Any level of self-interpreters can be both inverted and uncalled. A reversible self-interpreter for the original Janus and a tower of reversible interpreters were reported in [32].

3 Further Reading

Several introductory articles and surveys on reversible computing have been published (e.g., [13,24,9,20,3]). The concept of reversibility has been studied by using various computation models, including reversible Turing machines [2,21], reversible cellular automata [20], reversible flowchart [31], reversible combinatory logic [6], reversible process calculi [25], reversible Boolean logic circuits [10,5], and reversible finite automata [26].

Several reversible programming languages have been proposed. Especially, reversible languages that ensure the reversiblity of programs by reversibly composing reversible primitives are as follows. To our knowledge, Janus [17] is the first reversible language, which has been recently formalized by the authors [30,32]. Given R [8] source code, R compiler generates PISA code, which runs on the reversible processor Pendulum [29,1]. Gries' invertible language [12], an injective functional language Inv [22] and (E)SRL [18] also belong to this language class. Saving a trace of computation enables embedding irreversible computation into reversible computation [16]. Reversible languages using such reversible simulation also have been extensively studied [27,33,15]. The simulation technique has been successfully applied to several computation models [28,10,6,31].

One of closely related concept to reversible programming languages is program inversion [11]. Generalized program inversion generates a semi-inversed program, in the sense that given some of the original inputs and outputs it returns the remaining inputs and outputs [23,19]. Bidirectional languages, which also have the concept of forward and backward semantics, are desined for the view updating problem [7,22].

Acknowledgement

The authors wish to thank Irek Ulidowski for his comments on the earlier version of this tutorial paper.

References

[1] Axelsen, H., R. Glück and T. Yokoyama, *Reversible machine code and its abstract processor architecture*, in: V. Diekert, M. V. Volkov and A. Voronkov, editors, *Computer Science – Theory and Applications, Proceedings*, LNCS **4649** (2007), pp. 56–69.

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

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

[4] Böhm, C. and G. Jacopini, *Flow diagrams, Turing machines and languages with only two formation rules*, Commun. ACM **9** (1966), pp. 366–371.

[5] De Vos, A., Y. Van Rentergem and K. De Keyser, *The decomposition of an arbitrary reversible logic circuit*, Journal of Physics A: Mathematical and General **39** (2006), pp. 5015–5035.

[6] Di Pierro, A., C. Hankin and H. Wiklicky, *Reversible combinatory logic*, Mathematical Structures in Computer Science **16** (2006), pp. 621–637.

[7] Foster, J. N., 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. **29** (2007), Article 17, pp. 1–65.

[8] Frank, M. P., "Reversibility for Efficient Computing," Ph.D. thesis, EECS Dept., MIT, Cambridge, Massachusetts (1999).

[9] Frank, M. P., *Introduction to reversible computing: Motivation, progress, and challenges*, in: *Computing Frontiers, Proceedings* (2005), pp. 385–390.

[10] Fredkin, E. and T. Toffoli, *Conservative logic*, International Journal of Theoretical Physics **21** (1982), pp. 219–253.

[11] Glück, R. and M. Kawabe, *Derivation of deterministic inverse programs based on LR parsing*, in: Y. Kameyama and P. J. Stuckey, editors, *Functional and Logic Programming, Proceedings*, LNCS **2998** (2004), pp. 291–306.

[12] Gries, D., "The Science of Programming," ch. 21: Inverting Programs, Texts and Monographs in Computer Science, Springer, Heidelberg, 1981 pp. 265–274.

[13] Hayes, B., *Reverse engineering*, American Scientist **94** (2006), pp. 107–111.

[14] Jacopini, G., P. Mentrasti and G. Sontacchi, *Reversible Turing machines and polynomial time reversibly computable functions*, SIAM Journal on Discrete Mathematics **3** (1990), pp. 241–254.

[15] Kluge, W. E., *A reversible SE(M)CD machine*, in: P. Koopman and C. Clack, editors, *Implementation of Functional Languages, Proceedings*, Selected Papers, LNCS **1868** (2000), pp. 95–113.

[16] Landauer, R., *Irreversibility and heat generation in the computing process*, IBM J. Res. Dev. **5** (1961), pp. 183–191.

[17] Lutz, C., *Janus: a time-reversible language*, *Letter to R. Landauer* (1986). http://www.cise.ufl.edu/~mpf/rc/janus.html

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

[19] Mogensen, T. Æ., *Semi-inversion of guarded equations*, in: R. Glück and M. Lowry, editors, *Generative Programming and Component Engineering, Proceedings*, LNCS **3676** (2005), pp. 189–204.

[20] Morita, K., *Reversible computing and cellular automata — A survey*, Theor. Comput. Sci. **395** (2008), pp. 101–131.

[21] Morita, K. and Y. Yamaguchi, *A universal reversible Turing machine*, in: J. Durand-Lose and M. Margenstern, editors, *Machines, Computations, and Universality, Proceedings*, LNCS **4664** (2007), pp. 90–98.

[22] Mu, S.-C., Z. Hu and M. Takeichi, *An injective language for reversible computation*, in: D. Kozen, editor, *Mathematics of Program Construction, Proceedings*, LNCS **3125** (2004), pp. 289–313.

[23] Nishida, N., M. Sakai and T. Sakabe, *Partial inversion of constructor term rewriting systems*, in: J. Giesl, editor, *Term Rewriting and Applications, Proceedings*, LNCS **3467**, 2005, pp. 264–278.

[24] Pan, W. and M. Nalasani, *Reversible logic*, Potentials, IEEE **24** (2005), pp. 38–41.

[25] Phillips, I. and I. Ulidowski, *Reversing algebraic process calculi*, Journal of Logic and Algebraic Programming **73** (2007), pp. 70–96.

[26] Pin, J.-E., *On the language accepted by finite reversible automata*, in: T. Ottmann, editor, *International Colloquium on Automata, Languages and Programming, Proceedings*, LNCS **267** (1987), pp. 237–249.

[27] Stoddart, B., R. Lynas and F. Zeyda, *A reversible virtual machine*, in: I. Ulidowski, editor, *Reversible Computation, Preliminary Proceedings*, 2009, pp. 18–32.

[28] Toffoli, T., *Computation and construction universality of reversible cellular automata*, Journal of Computer and System Sciences **15** (1977), pp. 213–231.

[29] Vieri, C. J., "Reversible computer engineering and architecture," Ph.D. thesis, MIT (1999).

[30] Yokoyama, T., H. Axelsen and R. Glück, *Principles of a reversible programming language*, in: *Computing Frontiers, Proceedings* (2008), pp. 43–54.

[31] Yokoyama, T., H. Axelsen and R. Glück, *Reversible flowchart languages and the structured reversible program theorem*, in: L. A. I. Damgård, L. A. Goldberg, M. M. Halldórsson and A. I. I. Walukiewicz, editors, *International Colloquium on Automata, Languages and Programming, Proceedings*, LNCS **5126**, 2008, pp. 258–270.

[32] Yokoyama, T. and R. Glück, *A reversible programming language and its invertible self-interpreter*, in: *ACM/SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, Proceedings*, ACM Press, 2007, pp. 144–153.

[33] Zuliani, P., *Logical reversibility*, IBM J. Res. Dev. **45** (2001), pp. 807–818.