Clean Translation of an Imperative Reversible Programming Language

Holger Bock Axelsen

DIKU, Dept. of Computer Science, University of Copenhagen
funkstar@diku.dk

Abstract. We describe the translation techniques used for the code generation in a compiler from the high-level reversible imperative programming language Janus to the low-level reversible assembly language PISA. Our translation is both semantics preserving (correct), in that target programs compute exactly the same functions as their source programs (*cleanly*, with no extraneous garbage output), and efficient, in that target programs conserve the complexities of source programs. In particular, target programs only require a constant amount of temporary garbage space.

The given translation methods are generic, and should be applicable to any (imperative) reversible source language described with reversible flowcharts and reversible updates. To our knowledge, this is the first compiler between reversible languages where the source and target languages were independently developed; the first exhibiting both correctness and efficiency; and just the second compiler for reversible languages overall.

1 Introduction

Reversible computing is the study of computation models that exhibit both forward and backward determinism [2]. Historically, reversible computing originates in the physics of computation: Irreversible computations (for example, seen in our inability to recover the input to a NAND-gate from its output, or the previous value of a variable or register after most assignments) can be shown to have a physical effect on the machines that execute them, in the form of heat dissipation and power consumption [7]. For reversible computations these physical consequences are no longer implied, and it should therefore be possible to lower the power consumption of computing machinery by using reversible components [5]. Lowering power demands is increasingly important as microprocessor technology bottoms out at the atomic level.

To obtain maximal benefit from reversibility a reversible computer should be reversible at every abstraction layer, so reversible hardware [16,13] demands reversible software. However, reversible programming languages are rare and underdeveloped. This is unfortunate, given that reversible programming finds use in many diverse areas of computer science. As an example, in quantum computing [14] programs are necessarily reversible (modulo measurements, which are destructive.) Other application include bidirectional model transformation [10], static analysis of program properties such as average case time complexity [11], and complex program transformations such as inversion [15].

In the context of compiler theory, we find that there are *no* well-established principles for translation between reversible languages. Two baseline criteria for such a compiler `comp` are its *correctness* and *efficiency*. Correctness means that the translation should be *semantics preserving*, and efficiency here means that the translation should be *complexity preserving*: Given a source program `p`, the target program $\texttt{q} = [\![\texttt{comp}]\!]\,\texttt{p}$ should compute the same function as the source program modulo data representation, *i.e.*, $[\![\texttt{p}]\!] = [\![\texttt{q}]\!]$, and the source and target programs should have the same asymptotic complexities *wrt* resource usage. That such translations are possible is critical for the usefulness of reversible languages.

Both correctness and efficiency present novel challenges for the translation of reversible programming languages: High-level reversible languages are usually reversible at a coarser level than low-level reversible languages. In our source language, Janus, reversibility bottoms out with fairly complex (reversible) statements like `x += a *(5+b) - 17/y`, whereas in the target language, PISA, reversibility bottoms out with individual instructions like `ADDI` r_1 `10`, which is much more fine-grained. The problem is that the components of the source language that exist *below* the granularity of reversibility (for instance, the irreversible expressions in Janus) must *still* be implemented reversibly in the target language, without generating any extraneous *garbage data* for the output. Targeting a reversible assembly language also means that fundamental conventional techniques, such as using scratch registers for temporary values, must be revised, as we can *not* allow overwrite registers. Finally, the source languages contain novel program features that no classical languages exhibit, such as the procedure *uncall* statement in Janus, which executes a procedure *backwards.*

In this paper, we present the techniques used for code generation in a correct and efficient translation from the high-level imperative reversible language Janus [9,19,17] to the low-level reversible assembly language PISA [16,6,3,13]. We consider these to be the most developed and well-understood extant reversible languages in their respective classes. A compiler based on the techniques was implemented and tested. To our knowledge, this paper presents the first translation of reversible languages where the source and target languages have been independently developed, and only the second compiler overall (the other being Frank's R-compiler [6, Apps. C & D].) Our main contributions are as follows.

– We provide a full description of code generation for a *correct* and *efficient* translation of Janus to PISA.
– We give general clean translation methods for generic reversible control flow operators, which avoids code duplication of the computation and uncomputation of their conditional expressions.
– We give methods for explicit register allocation in reversible evaluation of expression trees.

Program

$$p ::= d^* \ (\texttt{procedure}\ id\ s)^+$$
$$d ::= x \mid x\texttt{[}c\texttt{]}$$

Statements

$$\begin{aligned} s ::= {} & x \oplus\texttt{=}\ e \mid x\texttt{[}e\texttt{]} \oplus\texttt{=}\ e \\ \mid {} & \texttt{call}\ id \mid \texttt{uncall}\ id \\ \mid {} & \texttt{if}\ e\ \texttt{then}\ s\ \texttt{else}\ s\ \texttt{fi}\ e \\ \mid {} & \texttt{from}\ e\ \texttt{do}\ s\ \texttt{loop}\ s\ \texttt{until}\ e \\ \mid {} & \texttt{skip} \mid s\ s \end{aligned}$$

Expressions, operators, constants

$$\begin{aligned} e & ::= c \mid x \mid x\texttt{[}e\texttt{]} \mid e \otimes e \\ \otimes & ::= \oplus \mid \texttt{*} \mid \texttt{\&\&} \mid \texttt{<=} \mid \cdots \\ \oplus & ::= \texttt{+} \mid \texttt{-} \mid \texttt{\^{}} \\ c & ::= \cdots \mid \texttt{-1} \mid \texttt{0} \mid \texttt{1} \mid \cdots \end{aligned}$$

Syntax domains

$p \in$ Program	$e \in$ Expression	$\otimes \in$ Operator
$s \in$ Statement	$x \in$ Variable	$id \in$ ProcedureID

Fig. 1. Syntax of Janus

Overview. Sect. 2 presents the target and source languages, Sect. 3 motivates the guiding principles underlying the translation, and Sect. 4 provides schemes for code generation. We discuss the implemented compiler in Sect. 5, related work in Sect. 6, and give conclusions and directions for future work in Sect. 7.

2 Languages

Here, we provide a brief overview of the source and target languages.

2.1 Source Language: Janus

The source language for the translation is *Janus*, an imperative structured high-level reversible language designed in the early 80's [9]. Its recent formalization [19] and extension [17] makes it one of the most well-understood and well-developed reversible language in existence. We specifically use the version of Janus defined in [19].

A Janus program consists of a list of global variable declarations, and a list of procedure (subroutine) declarations, see Fig. 1 (some operators are omitted for space reasons). A global[1] variable is a (32-bit signed) integer x, or a (zero-indexed) static size array of integers $x\texttt{[}c\texttt{]}$. All variables and arrays are initialized to zero.

[1] In this version of Janus there are no local variables.

A procedure is a non-parameterized list of statements. A statement s is a reversible assignment[2] to an integer variable x $\oplus$= e or array entry x[e_1] $\oplus$= e_2; a (recursive) procedure call `call` *id*; a procedure uncall `uncall` *id*; a reversible conditional selection `if` e_1 `then` s_1 `else` s_2 `fi` e_2, or a reversible loop `from` e_1 `do` s_1 `loop` s_2 `until` e_2.

Briefly, the semantics of Janus is as follows. A reversible assignment updates the left value by adding (`+=`), subtracting (`-=`) or (bitwise) xoring (`^=`) the value of the expression on the right hand side to the original value. The variable being updated must neither occur in the right hand side nor in the array index expression, to conserve reversibility. A reversible conditional selection is similar to a classical `if-then-else`, but the value of the `fi`-expression must be true after execution if the `then`-branch was taken, and false otherwise. A reversible loop is similar to a classical `do-while`, but the `from`-assertion must be true when entering the loop, and false in all subsequent iterations (*cf.* Fig. 7). A `call` recursively executes a named procedure, and an `uncall` executes the procedure with its inverse functionality. Such direct programming access to the inverse semantics of a languages is a unique feature of reversible languages.

To provide Janus with a stand-alone execution behaviour, the last procedure in the procedure list acts as the `main` procedure for a program, and is executed at run-time.[3]

2.2 Target Language: PISA

We target the *Pendulum Instruction Set Architecture* assembly language (PISA), as described in [6,3]. The Pendulum [16] is a RISC architecture, with 32 general purpose 32-bit signed integer registers, designated r_0 to r_{31}.[4] A formal semantics for PISA and an abstract von Neumann machine on which to run it, is described in [3]. A minimal assembly language BobISA, inspired by PISA, is being developed [13] for future implementation in reversible logic [12]. PISA is thus representative of a large class of reversible machine languages.

A PISA program is a list of (possibly labeled) RISC-style machine instructions, see Fig. 2 for a representative excerpt. An instruction is either a data instruction, or a branching instruction, or a special instruction used in program control.

A data instruction has no direct influence on control flow. The data instructions are reversible versions of classical RISC instructions. As an example, the `ADD` r_1 r_2 instruction performs the reversible update $r_1 \leftarrow r_1 + r_2$ (in Janus syntax r_1 `+=` r_2). To conserve reversibility the source and target registers must

[2] A reversible assignment follows the pattern of a *reversible update*: It is a function of the form $g(x, y) = (x \oplus f(y), y)$, where $\oplus$ is a binary operator that is injective in its first argument, *i.e.*, that $b \oplus a = c \oplus a \Rightarrow b = c$. This makes g injective. Note that f is unrestricted, allowing us to use irreversible operators, such as logical conjunction, in the right hand side of a reversible assignment. See also [3,17].

[3] An online interpreter can be found at `http://topps.diku.dk/pirc/janus`

[4] By convention, r_0 is usually preserved as 0, r_1 is the call stack pointer and r_2 is used to store return offsets in procedure calls.

Program

$$\begin{aligned} p &::= ([l\ :]\ i)^{+} \\ i &::= a \mid b \mid s \end{aligned}$$

Data instructions

$$\begin{aligned} a ::= {} & \texttt{ADD}\ r\ r \mid \texttt{SUB}\ r\ r \mid \texttt{NEG}\ r \mid \texttt{XOR}\ r\ r \mid \cdots \\ \mid {} & \texttt{ADDI}\ r\ c \mid \texttt{SUBI}\ r\ c \mid \texttt{XORI}\ r\ c \mid \cdots \\ \mid {} & \texttt{ORX}\ r\ r\ r \mid \texttt{ANDX}\ r\ r\ r \mid \texttt{SLTX}\ r\ r\ r \mid \cdots \\ \mid {} & \texttt{EXCH}\ r\ r \end{aligned}$$

Branching instructions

$$\begin{aligned} b ::= {} & \texttt{BRA}\ l \mid \texttt{RBRA}\ l \\ \mid {} & \texttt{BEQ}\ r\ r\ l \mid \texttt{BNE}\ r\ r\ l \mid \texttt{BGEZ}\ r\ l \mid \cdots \\ \mid {} & \texttt{SWAPBR}\ r \end{aligned}$$

Special instructions

$$s ::= \texttt{DATA}\ c \mid \texttt{START} \mid \texttt{FINISH}$$

Immediates

$$c ::= \cdots \mid \texttt{-1} \mid \texttt{0} \mid \texttt{1} \mid \cdots$$

Syntax domains

$$\begin{array}{lll} p \in \text{Program} & a \in \text{DataInst} & s \in \text{Special} \\ l \in \text{Label} & b \in \text{BranchInst} & r \in \text{Register} \end{array}$$

Fig. 2. Syntax of PISA (Excerpt)

be different. One may also use immediates in place of a source register. For instructions which do not mimic an operation which is injective in an argument, expanding instructions such as `ANDX` r_1 r_2 r_3 (which performs $r_1 \leftarrow r_1$ XOR (r_2 AND r_3)) are available. This is also used in comparison operations, such as *set-if-less-than* `SLTX`. Reversible memory access is provided by `EXCH` r_1 r_2 which exchanges the contents of r_1 with the value in the memory cell pointed to by r_2.

A branching instruction is an unconditional (*e.g.*, `BRA` l) or conditional branch (*e.g.*, `BGEZ` r l) to a label. Branching is made reversible through the architectural design of PISA: At load time, labels are replaced with the *relative offset* of a branch instruction and its target. A branch instruction does not overwrite the program counter *pc* directly, but instead adds the offset to a control register, the branch register *br*. In between the execution of any two instructions, the *pc* is reversibly updated by adding the *br* to it if $br \neq 0$, and proceeding to the next instruction, if not.

To program jumps in PISA one can use *paired branches*: A branch target should contain a branching instruction pointing back to the jump point, in order to clear the *br* and resume normal step-wise execution. A final control register, the direction bit *dir*, controls the interpretation and execution direction, allowing for programmer control of the execution direction through the ingenious reverse branch instruction `RBRA`. To allow for a labeled instruction to be jumped to from many source points, the `SWAPBR` r instruction exchanges the value of br with that in register r. Further details in [6,3].

The special instructions are not executed at run-time: The `DATA` c instruction is used to initialize the memory cell at that point with c at load-time. Execution begins at the `START` instruction, and halts at the `FINISH` instruction.

3 Motivation

Before providing technical details, we motivate the guiding principles for the translation.

Historically, the study of reversible computing (starting with Landauer [7] and Bennett [4]) has been focused on transformations from irreversible to reversible programs (for Turing machines), so-called *reversibilizations*. Critically, such transformations are neither semantics nor complexity preserving: The target program of such a transformation computes a different function from the source program, with additional *garbage data* in the output (*e.g.*, a trace of every computation step), and can be asymptotically very inefficient (*e.g.*, requiring as much space as time, regardless of the space usage of the source program.)

Bennett suggests a method for programs computing *injective* functions, see [4, p. 530], which is semantics preserving. While this is *extensionally* clean (*i.e.*, correct), it is still not satisfactory: The method requires the use of a complete execution trace, so target programs would be extremely inefficient, using as much space as time. This is clearly not acceptable for any realistic computing device.

Thus, using existing reversibilizations is unsuitable when we want both correct and efficient translation between reversible language. In fact, no general reversibilization (from irreversible source to reversible target) will be able to guarantee both properties without breaking widely believed conjectures in complexity theory (such as the existence of one-way functions.) However, we also believe that translations between strictly reversible languages *can* be both correct and efficient. Thus we should *not* rely on general reversibilizations.

The central idea of our translation is to exploit that Janus is reversible at the level of *individual statements* in addition to the overall reversibility between input and ouput. We can aim for an *intensionally* clean translation where each individual statement is translated cleanly, without having to accumulate garbage. We still need to evaluate expressions as a subpart of statements, and this *will* require some use of reversibilization (leading to temporary garbage) as expression evaluation is inherently irreversible. However, we should be able to remove the garbage data immediately following any use of the expression value, exactly because the individual statements are reversible. This will allow us to *reuse the space that would otherwise be filled* by accumulating garbage data!

This has a dramatic effect on efficiency: Expressions are non-recursive, so their size is effectively a (rough) bound on how much garbage we can accumulate by

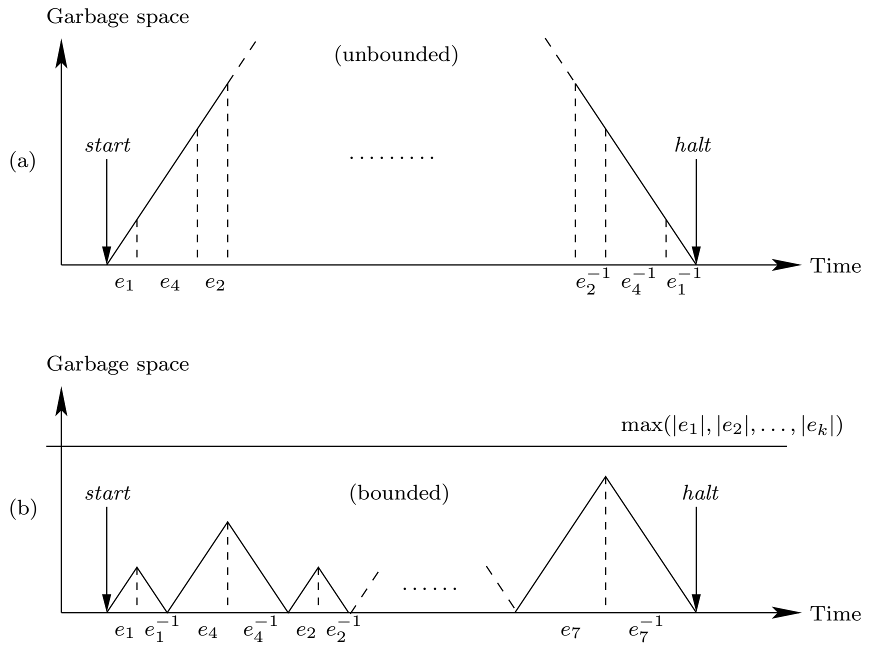


Fig. 3. Garbage space usage during target execution for two different approaches to reversible translation. (a) corresponds to Bennett's method for injective functions. (b) is our translation. $\{e_1, e_2, \ldots, e_k\}$ is the (finite) set of expressions in a particular program.

simulating any single one. This leads to a *constant* upper bound for garbage use for any given program (which only has finitely many expressions, each of fixed size), regardless of how many times we need to evaluate them in a program execution, in stark contrast to what happens with general reversibilization. Fig. 3 shows a conceptual representation of this.

The removal of any garbage data serves to make the translation *correct*, and doing so immediately makes the translation *efficient* (assuming we can translate the other component of Janus correctly and efficiently as well.) Thus, by actively exploiting the reversibility of the source language, we expect the irreversible subparts to be simulable without any impact on neither semantics nor complexity. Of course, this means that we have to come up with good, garbage-free translations for the remaining parts of the source language.

We now turn to the details of the translation based on these ideas.

4 Translation

In this section we present the translation techniques used in our compiler. In particular, we show the development of the *code generation* schemes used in

```
           < variable defs >
           < procedure code >
start  : START                ; Start program here
         ADDI r_sp size(q)    ; Init stack pointer
         BRA main             ; Call main procedure
         SUBI r_sp size(q)    ; Clear stack pointer
finish : FINISH               ; Exit program here
```

Fig. 4. Overall layout of a translated program q

the translation. Although we show the translation for two particular languages, the presentation is intended to be sufficiently abstract that one may adapt the schemes for use with other reversible source and target languages (or parts thereof) as well. Many parts of the compiler (parsing, syntax analysis etc.) were straightforwardly implemented using classical methods, and will not be discussed.

Janus is a *structured* language, we can inductively define the translation over the Janus syntax, and the presentation follows this structure:

– Overall target program structure is shown in Sect. 4.1.
– Procedure encapsulation and procedure calls are translated in Sect. 4.2.
– Reversible assignments (atomic statements) are translated in Sect. 4.3.
– Control flow operators are translated in Sect. 4.4.
– Expression evaluation is implemented in Sect. 4.5.

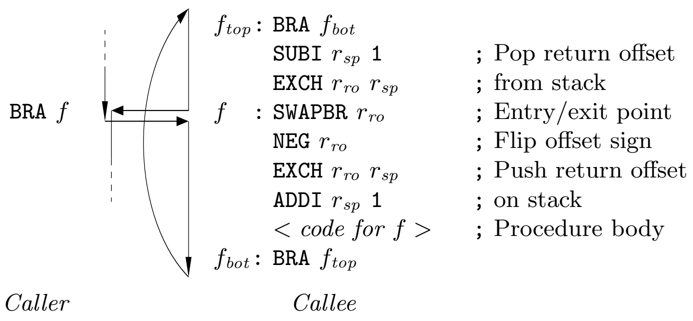
4.1 Overall Program Structure

A source program consists of declaration lists of global variables and procedures. The variables are global and of fixed size, so we can use the DATA 0 instruction to allocate space for them in the translated program. We use the order of declarations from the source, and use the names as labels, *e.g.*

$$\begin{array}{ll} l_x & : \left.\begin{array}{l}\texttt{DATA 0}\\ \quad\vdots\\ \texttt{DATA 0}\end{array}\right\} n \text{ cells for array } x\texttt{[}n\texttt{]}\ , \\ l_y & : \texttt{DATA 0}\ ;\ 1 \text{ cell for integer } y\,. \end{array}$$

This will allow us to read the result of a program execution directly from these memory locations when the target program halts. Variables are also kept in memory across statements that do not refer to them, to simplify the translation.

Following the space for variables comes a list of translated procedures, see below for details. Finally, a small section of code defines the execution behaviour. We assume that the program is loaded at address 0. Program execution begins with the *pc* at the START instruction labeled *start*. We shall need a call stack for recursive program calls, so a stack pointer r_{sp} (any of the general purpose registers, but usually r_1) is initialized to point at the first free memory cell



```
         f_top : BRA f_bot
                 SUBI r_sp 1         ; Pop return offset
                 EXCH r_ro r_sp      ; from stack
BRA f    f     : SWAPBR r_ro         ; Entry/exit point
                 NEG r_ro            ; Flip offset sign
                 EXCH r_ro r_sp      ; Push return offset
                 ADDI r_sp 1         ; on stack
                 < code for f >      ; Procedure body
         f_bot : BRA f_top
```

Fig. 5. Translation and calling convention for (recursive) procedures. The arrows show the control flow of a procedure call to f.

above the program by offsetting it with the size of the program *size*(q) (which is a constant). Then, we call *main*, (where *main* is the name of the last defined procedure in the program) using the calling convention below. Returning from this procedure call leads to the termination of the program with the FINISH instruction labeled *finish*. Following this, the call stack will be empty, and we clear the stack pointer, for cleanness.

4.2 Procedure Definitions and Procedure Calls

Procedure declarations are translated using a generalized version of the calling convention for PISA defined in [6], with added support for recursion.[5] The translation is most easily explained by considering how a caller and callee interact, see Fig. 5 for code.

The caller uses a simple BRA f instruction to call a procedure. This leads to a jump to the instruction labeled f. This entry/exit point is common to all callers, so the jump offset from caller to callee (the value currently in the branch register br), is moved to a general purpose *return offset* register r_{ro}, using the SWAPBR instruction. If r_{ro} is 0 when the call is made, the combined effect of the BRA and SWAPBR is to jump from caller to callee and proceed with normal step-wise execution from the callee entry point.

When returning from the procedure, the desired offset from callee to caller is the same distance as the original jump offset but with reverse sign, so we negate r_{ro} with NEG to get the return offset. The ADDI and EXCH instruction pushes the return address onto the call stack. The procedure body (which can include recursive calls to f) is then executed. After this, the paired BRA instructions send

[5] In the extended version of Janus [17] procedures are equipped with call-by-reference parameters. The above calling convention can support this straightforwardly in the call sequence by placing the references in an activation record on the call stack. These can be popped into procedure-dependent dedicated registers for formal parameters in the callee prologue (or when needed) and dereferenced when formal parameters are used.

```
(1) <code for $r_a \leftarrow [\![e_1]\!]$ > ; Generates garbage $G_1$
(2) ADDI $r_a$ $l_x$                        ; Add base address to index
(3) <code for $r_e \leftarrow [\![e_2]\!]$ > ; Generates garbage $G_2$
(4) EXCH $r_d$ $r_a$                        ; Swap array entry into $r_d$
(5) ADD $r_d$ $r_e$                         ; Update array entry
(6) EXCH $r_d$ $r_a$                        ; Swap back array entry
(7) <inverse code of 3>                     ; Removes garbage $G_2$
(8) SUBI $r_a$ $l_x$                        ; Subtract base address
(9) <inverse code of 1>                     ; Removes garbage $G_1$
```

Fig. 6. Translation of reversible (array) assignment x`[`e_1`]+=` e_2. For assignments which use `-=` or `^=` substitute the instruction in line 5 with `SUB` or `XOR`.

control to the top of the procedure encapsulation. Here, we pop the return offset from the stack and put it into r_{ro}, with the `SUBI` and `EXCH` instructions. Then, the `SWAPBR` returns control to the caller, where the `BRA` instruction restores the *br* to 0 again, and the caller continues its execution.

Note that the use of a call stack does *not* lead to garbage data: Any data that is added to the stack by a call is also *cleared* when returning from the call, so the size of the stack is conserved over, though not during, calls. Since the stack is initially *empty* when the program calls the *main* procedure, it will also be so when the program terminates.

Finally, procedure uncalls are supported by the use of the `RBRA` instruction in place of `BRA` in the caller. Nothing needs to change in the callee, which means that the *same* code can be shared for both calls and uncalls.

4.3 Reversible Assignments

Janus assignment statements are reversible updates. These can be implemented using a generalized version of Bennett's method, where the intermediate copying phase is replaced by in-place updating the left hand side variable, see [3]. The main difficulty we face is that expression evaluation is, in general, irreversible, and embedding this evaluation in PISA will necessarily generate garbage data. For a *clean* translation, this garbage must be disposed of, *i.e.*, it must be reversibly cleared.

We shall detail the case of assignment to an array variable, x`[`e_1`] +=` e_2. The translation of this is as follows (see Fig. 6 for a corresponding code template):

(1) Reversibly evaluate the index expression e_1, placing the result in (zero-cleared) register r_a. This will generate some garbage data G_1. (2) Add the base address l_x of the array x to r_a, yielding the exact address of the entry in memory. (3) Reversibly evaluate the update expression e_2, placing the result in zero-cleared register r_e. This will generate garbage data G_2. (4) Swap the array entry from its location in memory (given by r_a) with some register r_d, which need *not* be zero-cleared, but which must be different from r_a and r_e. (5) Update the array entry value in r_d by adding (subtracting, xoring) r_e. (6) Swap the updated array entry back to its memory location, restoring r_d to its original

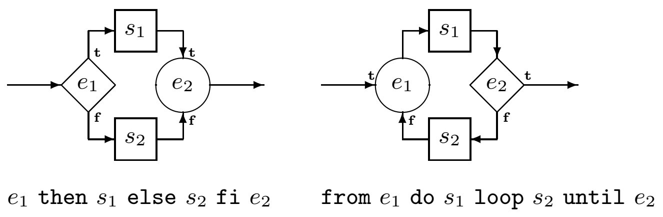

s_1
s_2
e_1
e_2
t
f
t
f
if e_1 then s_1 else s_2 fi e_2
s_1
s_2
e_1
e_2
t
f
t
f
from e_1 do s_1 loop s_2 until e_2

Fig. 7. Reversible flowcharts for the Janus CFOs

value. (7) *Uncompute* (unevaluate) expression e_2, removing all the garbage data G_2 generated in the forwards evaluation of this expression and clearing r_e. This can be done with the *inverse* of the code from step 3, see below. (8) *Subtract* the base address of x from r_a, leaving only the index. (9) *Uncompute* the index expression e_1, clearing garbage data G_1 and register r_a. For this, use the inverse of the code generated for step 1.

Steps 1–6 are almost completely conventional, except for the fact that the computations in steps 1 and 3 have the side effect of generating garbage data. If we ignore garbage we would not need to perform steps 7–9, but seeing as we want a clean semantics-preserving translation at the statement level these uncomputations are necessary.

All PISA instructions have inverses that are single PISA instructions as well. The inverse of `ADD` is `SUB`, the inverse of `ANDX` is itself, *etc.* This can be exploited to generate the inverse code for the uncomputations in steps 7 and 9 in a very straightforward manner: Reverse the list of instructions, and invert each instruction in the code from steps 1 and 3. This also means that steps 6–9 are actually the inverses of steps 1–4, so the entire effect of the translated code will be to update the array entry (step 5), with no garbage data left afterwards.

The value in register r_a, once we have computed the specific address in step 2, *must* be conserved over the evaluation of e_2 in step 3. It is only because we *know* that the register is later cleared that we may use it as a free register in future computations. In other words, no instruction in PISA can by itself be used to declare a register *dead*, making general register allocation in PISA non-trivial.

4.4 Control Flow Operators

Frank provides some (informal) guidelines for programming control structures in PISA [6, Ch. 9]. However, it is unclear how these can be used for a clean translation of Janus: There is little to no discussion of evaluation of conditionals, garbage handling, or any such concepts. Furthermore, the discussion of loops is largely limited to fixed iteration for-loops, which are much less general than Janus loops. However, Janus CFOs are fully implementable in PISA without generating any garbage data, as we shall demonstrate.

We shall use a bottom-up approach to explain our translation. It is easy to see that the Janus CFOs (Fig. 7) can be decomposed using the simpler reversible

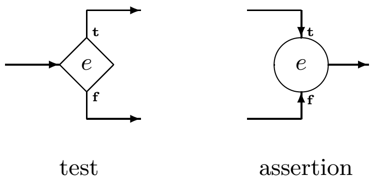


Fig. 8. General reversible flowchart nodes

constructs in Fig. 8. The *test* is simply a conventional conditional branch, while the *assertion* is a join of control flow, where the conditional expression must be *true* when entering the node via the edge labeled **t**, and *false* if entering via the edge labeled **f**, see [18]. We shall examine how each of these constructs can be translated cleanly to PISA and combined for the translation of Janus CFOs.

Translation of Tests. An initial attempt at a PISA translation of the test would be to follow the standard method: Evaluate the expression, placing the result in (zero-cleared) register r_e. Use a conditional jump on r_e for the control flow, jumping to another place in the code if the expression evaluated to zero (coding for *false*) and falling through if not (coding for *true*). In PISA this looks as follows.

```
              <code for r_e ← ⟦e⟧ >        ; Evaluate e
test        : BEQ r_e r_0 test_false        ; Jump if ⟦e⟧ = 0
              <code for true branch>
                        ⋮
test_false  : BRA test                     ; Receive jump
              <code for false branch>
                        ⋮
```

While functional, this approach poses several problems, the major of which is the massive generation of garbage: The value of the evaluation is left in register r_e after being used by the conditional branch, and the evaluation of e will likely leave garbage data in other registers and possibly in memory as well. The simplest way of removing the garbage data would be to push it onto a "garbage stack", but this would break the cleanness of the translation. Furthermore, since the test is a reversible flowchart, we expect to be able to simulate it in PISA without having to resort to the Landauer embedding.

We observe that we can use the inverse of the expression evaluation code for a Lecerf reversal [8]:

```
<code for r_e ← ⟦e⟧ >           ; Evaluate e
<inverse code for r_e ← ⟦e⟧ >   ; Unevaluate e
```

The total effect of this is to compute the identity. (We shall write $r_e \rightarrow [\![e]\!]$ as a shorthand for the *uncomputation*.) Because branch instructions can not alter the contents of r_e, we can place the uncomputation code at the start of both the *true* and *false* branch, and the total effect will be to compute the test without generating garbage.

```
               <code for r_e ← [[e]] >     ; Evaluate e
test         : BEQ r_e r_0 test_false      ; Jump if [[e]] = 0
               <code for r_e → [[e]] >     ; Unevaluate e
               <code for true branch>
                          ⋮
test_false   : BRA test                    ; Receive jump
               <code for r_e → [[e]] >     ; Unevaluate e
               <code for false branch>
                          ⋮
```

Note that the uncomputation code occurs twice in this translation. Because the expressions may be very large, this can have a significant impact on program size, with all its drawbacks, especially *wrt* debugging. This *code duplication* can be avoided per the following observation: Only the value of r_e has an effect on the control flow. Every other piece of garbage data from the expression evaluation may be safely uncomputed *before* the conditional jump. To clear r_e itself we require some knowledge of its value at the branches. If we jumped to the *false* branch then r_e is necessarily zero, and so already cleared. If we fell through to the *true* branch, then r_e was non-zero. Expression evaluation may in general return any integer result for r_e, so it seems that we have no way of deterministically clearing r_e in the *true* branch.

However, for *conditional* evaluations we may reduce the expression result further to being either 0 (for *false*) or 1 (for *true*), since only two distinct values are actually necessary to make the jump work (we write $[\![e]\!]_c$ for such conditional evaluation). This is easily implemented in PISA: If r_x contains the integer value of $[\![e]\!]$, we may compute $[\![e]\!]_c$ in r_e as follows.

```
cond_top : BEQ r_x r_0 cond_bot  ; If [[e]] ≠ 0,
           XORI r_e 1            ; set r_e = [[e]]_c = 1,
cond_bot : BEQ r_x r_0 cond_top  ; else leave r_e = 0.
```

With this strategy, r_e contains either 0 or 1 after evaluation of the conditional expression. Copy this result into an (initially zero-cleared) register r_t. We can now clear r_e along with the other garbage in the uncomputation before the jump.[6] After the jump r_t can now be deterministically cleared in the *true* branch by a single `XORI` instruction, since we know it must contain a 1. The full translation of a test is shown in Fig. 9; it avoids code duplication, and generates no garbage data.[7] (We discuss the error check below.)

[6] Technically, we do not need the extra register r_t. We can organize the uncomputation to clear everything *except* r_e. This is more efficient, but the details depend on the evaluation strategy for conditional expressions; in particular, on the top-level operator in e.

[7] An alternative strategy is to implement the expression evaluation as a subroutine. This removes the need for the inverse code as well, as we can use an `RBRA` instruction for the uncomputation.

```
                BNE $r_t$ $r_0$ *error*                 ; Error check
                <*code for* $r_e \leftarrow [\![e]\!]_c$ >   ; Evaluate $e$
                XOR $r_t$ $r_e$                         ; Set $r_t = r_e$
                <*code for* $r_e \rightarrow [\![e]\!]_c$ >  ; Unevaluate $e$
test          : BEQ $r_t$ $r_0$ $test_{false}$          ; Jump if $[\![e]\!] = 0$
                XORI $r_t$ 1                            ; Clear $r_t$
                <*code for true branch*>
                          ⋮
$test_{false}$ : BRA *test*                             ; Receive jump
                <*code for false branch*>
                          ⋮
```

Fig. 9. Code template for translation of the conditional test in Fig. 8

```
                          ⋮
                <*code for true branch*>
                XORI $r_t$ 1                            ; Set $r_t = 1$
$assert_{true}$ : BRA *assert*                          ; Jump
                          ⋮
                <*code for false branch*>
assert        : BNE $r_t$ $r_0$ $assert_{true}$         ; Receive jump
                <*code for* $r_e \leftarrow [\![e]\!]_c$ >   ; Evaluate $e$
                XOR $r_t$ $r_e$                         ; Clear $r_t$
                <*code for* $r_e \rightarrow [\![e]\!]_c$ >  ; Unevaluate $e$
                BNE $r_t$ $r_0$ *error*                 ; Error check
```

Fig. 10. Code template for translation of the assertion in Fig. 8

Translation of Assertions. We observe that an assertion is actually the *inverse* of a test. Since we already have a garbage-free translation of tests, we can use this symmetry property on the translation template in Fig. 9 to get a similarly garbage-free translation of *assertions*, practically for free. A translation of assertions is shown in Fig. 10.

Due to programmer errors, an assertion might *fail*: e might evaluate to *true* when coming from the *false* branch, and vice versa. In the given translations, such an error will manifest itself in the value of r_t: A failed assertion means that r_t will *not* be zero-cleared after the assertion code is executed. Rather than let the machine continue with erroneous (though reversible) behaviour, we can catch failed assertions by *dynamic error checks.* This is accomplished by checking the final value of r_t. If it is non-zero then the assertion failed, and we jump to an error handling routine *error.* Translated code may be executed in reverse (*e.g.*, in procedure uncalls). In that case a test acts as an assertion, so we need dynamic error checks in the translation of tests as well. Hence the seemingly superfluous check at the beginning of the translated test.

```
                BNE $r_t$ $r_0$ $error$                      ; Error check
                <$code$ $for$ $r_e \leftarrow [\![e_1]\!]_c$ >   ; Evaluate $e_1$
                XOR $r_t$ $r_e$                              ; Set $r_t = r_e$
                <$code$ $for$ $r_e \rightarrow [\![e_1]\!]_c$ >  ; Unevaluate $e_1$
$test$          : BEQ $r_t$ $r_0$ $test_{false}$             ; Jump if $[\![e]\!]_c = 0$
                XORI $r_t$ 1                                 ; Clear $r_t$
                <$code$ $for$ $true$ $branch$>
                XORI $r_t$ 1                                 ; Set $r_t = 1$
$assert_{true}$ : BRA $assert$                               ; Jump
$test_{false}$  : BRA $test$                                 ; Receive jump
                <$code$ $for$ $false$ $branch$>
$assert$        : BNE $r_t$ $r_0$ $assert_{true}$            ; Receive jump
                <$code$ $for$ $r_e \leftarrow [\![e_2]\!]_c$ >   ; Evaluate $e_2$
                XOR $r_t$ $r_e$                              ; Clear $r_t$
                <$code$ $for$ $r_e \rightarrow [\![e_2]\!]_c$ >  ; Unevaluate $e_2$
                BNE $r_t$ $r_0$ $error$                      ; Error check
```

Fig. 11. Translation of a Janus `if` e_1 `then` s_1 `else` s_2 `fi` e_2

Complete Translation of CFOs. The translations for the reversible control flow nodes in Figs. 9 and 10 can now be combined to yield the complete translation of a Janus conditional selection, see Fig. 11. The principles carry over directly to the the translation of a reversible loop, where the placement of code is a little more intricate, see Fig. 12.

The only Janus CFO left is the *sequence* operator. The other translated CFOs are structured with only one entry and exit point, so this amounts to simple code concatenation. Thus, all of Janus' control structures (and statements in general) are implementable in PISA with garbage-free translations.

4.5 Expression Evaluation

Janus expression evaluation is irreversible: different stores may evaluate a given expression to the same value. This means that we cannot, in any way, implement expression evaluation *cleanly* in a reversible language: Evaluating an expression necessarily leaves garbage. This was recognized in the translation of reversible assignments and CFOs, where we chose to *un*evaluate the expression immediately after the use of the expression value, to clear any such garbage generated. This makes the translation of statements clean, which is the best we can achieve.

The problem of reversible expression evaluation then reduces to finding a way of forwards evaluating expressions in general using reversible instructions, generating garbage as necessary. Because expressions are uncomputed after use, any reversibilization will work, but we should still aim for efficiency.

Expressions in Janus are trees, so we can use simple post-order traversal to generate code (*Maximal Munch*, *cf.* [1, Ch. 9]). A leaf node on the tree (representing a constant or variable) is simply copied into a free register. An internal

```
         XORI $r_t$ 1                          ; Set $r_t = 1$
entry  : BEQ $r_t$ $r_0$ assert                ; Receive jump
         <code for $r_e \leftarrow [\![e_1]\!]_c$ >  ; Evaluate $e_1$
         XOR $r_t$ $r_e$                       ; Clear $r_t = [\![e_1]\!]_c$
         <code for $r_e \rightarrow [\![e_1]\!]_c$ >  ; Unevaluate $e_1$
         BNE $r_t$ $r_0$ error                 ; Error check
         <code for $s_1$ >
         BNE $r_t$ $r_0$ error                 ; Error check
         <code for $r_e \leftarrow [\![e_2]\!]_c$ >  ; Evaluate $e_2$
         XOR $r_t$ $r_e$                       ; Set $r_t = [\![e_2]\!]_c$
         <code for $r_e \rightarrow [\![e_2]\!]_c$ >  ; Unevaluate $e_2$
test   : BNE $r_t$ $r_0$ exit                  ; Exit if $[\![e_2]\!]_c = 1$
         <code for $s_2$ >
assert : BRA entry                             ; Jump to top
exit   : BRA test                              ; Receive exit jump
         XORI $r_t$ 1                          ; Clear $r_t$
```

Fig. 12. Translation of a Janus `from` e_1 `do` s_1 `loop` s_2 `until` e_2 loop

node (an expression $e_1 \otimes e_2$, ignoring unary operators) is (recursively) translated, yielding the following code structure.

$$e_1 \otimes e_2 \quad \Longrightarrow \quad \begin{array}{l} 1.\ <\textit{code for } r_{e_1} \leftarrow [\![e_1]\!]> \\ 2.\ <\textit{code for } r_{e_2} \leftarrow [\![e_2]\!]> \\ 3.\ <\textit{code for } r_e \leftarrow r_{e_1}\ [\![\otimes]\!]\ r_{e_2}> \end{array}$$

Here, r_{e_1}, r_{e_2} and r_e are zero-cleared registers. This mimics the conventional translation of expressions, but in the reversible setting we are faced with two interesting problems. How do we allocate registers? How can we translate irreversible operators?

Register Allocation for Expression Trees. In irreversible languages this can be done optimally using the Sethi-Ullman algorithm. This is not available to us in reversible languages, as scratch registers cannot be indiscriminately overwritten. The Sethi-Ullman algorithm assumes them to be *dead*, but in this translation they are still live, as they will be reversibly cleared in uncomputation. In any case, overwriting a register simply is not possible in PISA. We thus also expect register pressure to be somewhat higher in reversible machine code. It is therefore important to know how we can free registers.

Instead of the usual categories of *live* and *dead* registers, we partition the register file into the following sets.

- *Free registers.* We know these to be zero-cleared, and may use them freely.
- *Commit registers.* These contain values that we shall need at a future point in the expression evaluation.
- *Garbage registers.* These contain values that are no longer needed for the computation.

In the translation of $e_1 \otimes e_2$ above, r_{e_1} is a *free* register before step 1. During step 2 it is a *commit* register as we need it for step 3, after which it becomes a *garbage* register. By maintaining the partitioning explicitly during the code generation for the expression, we can use several strategies to free registers.

- *Pebbling.* Garbage registers can be locally uncomputed. This is space-wise efficient, but can be very costly *wrt* time, if done recursively.
- *Garbage spills.* Garbage registers can be pushed unto the stack. This requires space, but reduces the number of executed instructions needed for an evaluation.
- *Commit spills.* We can push a *context* of commit registers onto the stack, and restore them when they are needed.

In the implemented compiler a mixture of all three strategies is used: At leaf nodes in the expression tree *pebbling* is used as the uncomputations at leafs are extremely short. For inner nodes *garbage spills* are used, with *commit spills* only as a last resort. With global variables allocated to memory, and no local variables, the compiler uses explicit register allocation throughout, without the use of virtual registers.

Operator Translation. With very few exceptions (unary minus, bitwise negation) the operators in Janus expressions are irreversible. However, most operators are still supported in PISA in various guises. Assume that we want to evaluate $x \otimes y$, with the values of x and y stored in r_x and r_y, respectively.

Addition, subtraction and exclusive-or are directly supported. For example, x `+` y can be evaluated by `ADD` $r_x\ r_y$. This has the added advantage of reusing r_x as the commit register for the total expression, leaving only r_y as garbage. Other operators, such as bitwise disjunction, x `|` y, have expanding support which consumes a free register (r_e): `ORX` $r_e\ r_x\ r_y$.

The most interesting, however, are those with no clear support in PISA, expanding or otherwise. As an example, we shall look at the equality test x `=` y. We shall need the `SLTX` (*set-less-than-xor*) instruction,

$$[\![\texttt{SLTX}\ r_d\ r_s\ r_t]\!] = r_d \leftarrow r_d \oplus ((r_s < r_t)\ ?\ 1\ :\ 0)\,,$$

where $\oplus$ is (bitwise) exclusive-or. We can use simple logical identities to reduce equality to the *less-than* comparisons. An obvious choice would seem to be

$$x = y \;\Leftrightarrow\; \neg(x < y \vee y < x)$$

However, both the logical NOR operation and *less-than* are only available as expanding instructions: Naïvely using this identity requires *three* free registers (here r_s, r_t and r_e) to compute x `=` y as follows.

$$\begin{array}{l} \texttt{SLTX}\ r_s\ r_x\ r_y \\ \texttt{SLTX}\ r_t\ r_y\ r_x \\ \texttt{NORX}\ r_e\ r_s\ r_t \end{array}$$

Since registers are a scarce commodity, we want to do better, and indeed we can. Note that at most one of $x < y$ or $y < x$ can be true ($<$ is antisymmetric).

$$x = y \;\Leftrightarrow\; \neg(x < y \;\oplus\; y < x)\,.$$

Exclusive-or is directly supported, and logical negation is reversible, so we can evaluate x = y using only one free register:

SLTX r_e r_x r_y
SLTX r_e r_y r_x
XORI r_e 1

Some operators (such as logical conjunction) still require three registers. Finally, there are also operators in Janus that have no short implementation in PISA, *e.g.*, multiplication or modulus, which can require executing hundreds of PISA instructions. For these one can inline and specialize a reversible simulation of the operator, or use the reversible simulations as subroutines, with proper parameter handling. In the latter case, it is important that the callee subroutine does not perturb any registers other than those of the arguments.

5 Implementation

A compiler based on the above translation methods was implemented in ML (Moscow ML, version 2.01), in approximately 1500 lines of code. Target programs were tested on the PendVM Pendulum simulator[8] and compared with source runs in a Janus interpreter. All tests corroborated the correctness and efficiency of the translation, so target programs do not leave garbage data in neither registers nor memory.

The largest program translated was a PISA interpreter written in Janus, specialized to a PISA program running a simple physical simulation of a falling object. Weighing in at slightly more than 500 lines of Janus code, this is possibly the largest reversible program ever written, and was certainly the most complex available to the author. The compiled code is *ca.* 12K PISA instructions long. This program had still negligible compile and execution times, so we omit timing statistics. In general, target programs were about 10–20 times larger (in lines of code) than their source programs.

6 Related Work

The only other work on compilers for reversible languages known to the author is Frank's R-to-PISA compiler [6]. The R-compiler is described mainly by commented code in [6, App. D], and is not being maintained, which makes it difficult to discern how the compiler is intended to work abstractly, and verify its correctness and/or efficiency.

R is a prototype reversible procedural language developed specifically for compilation to PISA. R shares some features with Janus, but also has a number of significant differences. For example, R's control flow operators (CFOs) are fairly weak. R-loops are made for definite iteration, and there is no if-then-else CFO. The R if-then CFO uses just a single conditional expression as both if- and fi-conditional. Also, all subexpressions of the conditional must be conserved across

[8] C. R. Clark, *The Pendulum Virtual Machine.* Available at `http://www.cise.ufl.edu/research/revcomp/users/cclark/pendvm-fall2001/`

the branch body. In particular, this means that no variables occurring in the conditional expression can be updated in the body, severely limiting the expressiveness of R. On the other hand, R does have some advanced features that Janus does not, like direct access to memory, and input/output facilities. However, the emerging picture is still that R is somewhat limited in its expressiveness as a programming language, compared to Janus.

We believe such restrictions were imposed on R to simplify compilation: The R-compiler can leave any garbage values used for conditional expressions *in place* across the branch bodies, computing and uncomputing the expression (which removes the garbage) only once, and only *after* the conditional is exited. While the analogous translation in Janus requires *two* computations and uncomputations, this also implies that the translation of R is not intentionally clean. This strategy is furthermore *not* applicable to Janus translation, where if- and fi-conditional expressions are allowed to be different, and variables occuring therein are allowed to be updated freely in the branch bodies. (However, our translation could easily be applied to R.) Finally, by not clearing the garbage value before entering the branch body, target programs can generate unbounded garbage data at run-time in recursive procedure calls, which breaks efficiency.

7 Conclusion and Future Work

We presented a correct and efficient translation for compiling the reversible high-level programming language Janus to the reversible low-level machine language PISA. Target programs produced using this translation conserve both the semantics (correctness) and space/time complexities (efficiency) of the source programs. We achieved this by making the translation *intensionally* (as well as extensionally) clean: Reversibility in Janus bottoms out at the statement level, and the compilation reflects this by translating each *individual* statement in the source program cleanly. This has the effect that at no point in the execution of any translated program do we accumulate more than a constant amount of temporary garbage data.

By breaking down the control flow operators of Janus into simpler reversible flowchart nodes, we found that we could exploit the symmetry properties of reversible flowcharts to simplify the translation. We also eliminated the need for code duplication in the translation. The developed translation methods are generic, and will work for all languages with control flow describable by reversible flowcharts and statements describable by reversible updates.

The aim here was to produce a working compiler that demonstrates the fundamental structure of a correct and efficient translation between reversible languages, leaving plenty of opportunities for development and future research. General register allocation methods for reversible assembly languages must be developed, and might benefit from the novel partitioning of registers we use for register allocation for expression trees. The heavy use of uncomputation of expression evaluations in both reversible assignments and conditionals suggest that novel as well as conventional optimizations (such as common subexpression elimination) could be very useful in compilers for reversible languages.

Acknowledgments. A preliminary version of this work was presented at the 2nd Workshop on Reversible Computing in Bremen, July 2010.

References

1. Appel, A.W.: Modern Compiler Implementation in ML. Camb. Uni. Press, New York (1998)
2. Axelsen, H.B., Glück, R.: What do reversible programs compute? In: Hofmann, M. (ed.) FOSSACS 2011. LNCS, vol. 6604, pp. 42–56. Springer, Heidelberg (2011)
3. Axelsen, H.B., Glück, R., Yokoyama, T.: Reversible machine code and its abstract processor architecture. In: Diekert, V., Volkov, M.V., Voronkov, A. (eds.) CSR 2007. LNCS, vol. 4649, pp. 56–69. Springer, Heidelberg (2007)
4. Bennett, C.H.: Logical reversibility of computation. IBM Journal of Research and Development 17, 525–532 (1973)
5. De Vos, A.: Reversible Computing: Fundamentals, Quantum Computing and Applications. WILEY-VCH, Weinheim (2010)
6. Frank, M.P.: Reversibility for Efficient Computing. PhD thesis, MIT (1999)
7. Landauer, R.: Irreversibility and heat generation in the computing process. IBM Journal of Research and Development 5(3), 183–191 (1961)
8. Lecerf, Y.: Machines de Turing réversibles. Récursive insolubilité en $n \epsilon \mathbf{N}$ de l'équation $u = \theta^n u$, oú θ est un "isomorphisme de codes". Comptes Rendus Hebdomadaires 257, 2597–2600 (1963)
9. Lutz, C.: Janus: a time-reversible language. Letter written to R. Landauer (1986), `http://tetsuo.jp/ref/janus.html`
10. Mu, S.-C., Hu, Z., Takeichi, M.: An algebraic approach to bi-directional updating. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 2–20. Springer, Heidelberg (2004)
11. Schellekens, M.: MOQA; unlocking the potential of compositional static average-case analysis. Journal of Logic and Algebraic Programming 79(1), 61–83 (2010)
12. Thomsen, M.K., Axelsen, H.B.: Parallelization of reversible ripple-carry adders. Parallel Processing Letters 19(2), 205–222 (2009)
13. Thomsen, M.K., Glück, R., Axelsen, H.B.: Towards designing a reversible processor architecture (work-in-progress). In: Reversible Computation. Preliminary Proceedings, pp. 46–50 (2009)
14. Thomsen, M.K., Glück, R., Axelsen, H.B.: Reversible arithmetic logic unit for quantum arithmetic. J. of Phys. A: Math. and Theor. 42(38), 2002 (2010)
15. van de Snepscheut, J.L.A.: What computing is all about. Springer, Heidelberg (1993)
16. Vieri, C.J.: Reversible Computer Engineering and Architecture. PhD thesis, MIT (1999)
17. Yokoyama, T., Axelsen, H.B., Glück, R.: Principles of a reversible programming language. In: Proceedings of Computing Frontiers, pp. 43–54. ACM Press, New York (2008)
18. Yokoyama, T., Axelsen, H.B., Glück, R.: Reversible flowchart languages and the structured reversible program theorem. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 258–270. Springer, Heidelberg (2008)
19. Yokoyama, T., Glück, R.: A reversible programming language and its invertible self-interpreter. In: Proceedings of Partial Evaluation and Program Manipulation, pp. 144–153. ACM Press, New York (2007)