A Linear-Time Self-Interpreter of a Reversible Imperative Language

Robert Glück　Tetsuo Yokoyama

A linear-time reversible self-interpreter in an r-Turing complete reversible imperative language is presented. The proposed imperative language has reversible structured control flow operators and symbolic tree-structured data (S-expressions). The latter data structures are dynamically allocated and enable reversible simulation of programs of arbitrary size and space consumption. As self-interpreters are used to show a number of fundamental properties in classic computability and complexity theory, the present study of an efficient reversible self-interpreter is intended as a basis for future work on reversible computability and complexity theory as well as programming language theory for reversible computing. Although the proposed reversible interpreter consumes superlinear space, the restriction of the number of variables in the source language leads to linear-time reversible simulation.

1 Introduction

Reversible computing is a relatively young area of computer science that studies the theory and practice of forward and backward computation on all levels of the computation stack ranging from logic circuits and computer architectures to programming languages and algorithms (e.g. [8][29][34]). Reversible programming languages compute only *injective functions* [2] owing to the forward and backward determinism of the underlying computation model. This means that there is a unique relation between the input and the output of a reversible program. The validity of several fundamental programming language theories, whose irreversible counterparts are often taken for granted, has not yet been examined in reversible computing. Examples of recently shown properties include the structured program theorem [32] and the universality of Turing machines [2].

This paper deals with the question of the existence of *linear-time self-interpretation* in reversible computing (i.e., *efficient* interpretation in the sense of Jones [16]). Linear-time interpreters are used in computability and complexity theory. Among others, they are used to show the existence of a linear-time complexity hierarchy along the lines of Jones [16][17] and of Levin's optimal universal search [18][13]. Although linear-time self-interpretation is a good property to have in any computation model, no such self-interpreter is known in some models (cf. a universal Turing machine as fast as that with a logarithmic-time overhead is known [19]), whereas it is for the pure untyped lambda calculus [21] and for Jones' imperative `I` language [16]. However, the existence of linear-time self-interpreters in irreversible computing does not imply the corresponding result for reversible computing.

We begin by introducing the language `R-WHILE`, a reversible version of Jones' `WHILE` language. The language has structured control flow operators and symbolic tree-structured data (S-expressions

---

可逆命令型言語の線形時間自己解釈系

ロバート・グリュック, コペンハーゲン大学計算機科学部, DIKU, Department of Computer Science, University of Copenhagen.

横山哲郎, 南山大学理工学部ソフトウェア工学科, Department of Software Engineering, Nanzan University.

コンピュータソフトウェア, Vol.33, No.3 (2016), pp.108–128.

[研究論文] 2015 年 7 月 30 日受付.

known from Lisp). The latter allows for the modeling of many familiar data structures in a simple way, which is more difficult in existing reversible imperative languages such as Janus, which only provide integers, arrays, and stacks. It was only recently shown how to represent and manipulate binary trees in the heap of a reversible random-access machine [3]. We shall see that a linear-time self-interpreter exists in R-WHILE-M, an r-Turing complete subset of the language R-WHILE in which the number of variables is bounded.

The reversible structured control flow operators of the reversible imperative reversible language R-WHILE-M are instances of those of the reversible flowchart language [34]. There are other r-Turing complete languages with reversible structured control flow operators such as Janus and R [9]. However, none of these languages have tree-structured data, and they explicitly limit the number of variables and constants.

Moreover, the reversibility of R-WHILE will be shown from a new viewpoint, namely by using its denotational semantics. This simplifies the proof of this defining property as compared to the previous approach used for the reversible language Janus [35].

The work presented here is part of a larger effort on advancing reversible computing systems on all levels, including the design and physical realization of reversible logic circuits [8] [29] [25], reversible computer architectures [4] [28], and programming languages and abstract machines [1] [20] [22].

To demonstrate our results, a complete online interpreter for R-WHILE, an implementation of the linear-time reversible self-interpreter in R-WHILE-M, and the program inverter are available from http://tetsuo.jp/rwhile-playground/.

2 Standard vs. Reversible Interpreters

We define the notion of a reversible interpreter and its efficiency. We find the fundamental differences between standard and reversible interpreters already on this abstract level and even more when we later consider the actual construction of reversible interpreters. We begin by recalling the standard definition of an interpreter and then discuss the properties of reversible interpreters.

We use the notation presented in [17]: Programs and their input and output are written in typewriter font. If $\mathtt{p}$ is a program written in the language S, then $[\![\mathtt{p}]\!]^{\mathtt{S}}$ denotes the meaning of $\mathtt{p}$, i.e., the input–output function that $\mathtt{p}$ computes. Thus, the output $\mathtt{d}' = [\![\mathtt{p}]\!]^{\mathtt{S}}(\mathtt{d})$ results from running $\mathtt{p}$ on the input $\mathtt{d}$. It is undefined if the program does not terminate.

We assume that programs compute with trees built from a finite set of atoms. The data domain $\mathbb{D}$ of a program is the smallest set that contains the finite set of atoms and all of the pairs $(\mathtt{d}_1.\mathtt{d}_2)$, where $\mathtt{d}_1, \mathtt{d}_2 \in \mathbb{D}$.

2.1 Standard Interpreter

An L-program int is an *interpreter* of a language S in the language L iff for any S-program p and S-data d and $\mathtt{d}'$[†1],

$$[\![\mathtt{int}]\!]^{\mathtt{L}}(\mathtt{p}.\mathtt{d}) = \mathtt{d}' \iff [\![\mathtt{p}]\!]^{\mathtt{S}}(\mathtt{d}) = \mathtt{d}' . \quad (1)$$

Following the convention [17], whenever a program is an input into another program, we regard it as the corresponding data representation. For example, p on the right-hand side is an S-program text, whereas p on the left-hand side is a representation of the text as L-data (e.g., as tree-structured data).

The language S is the *source language*, and L is the *implementation language* of int. If the source language and the implementation language are the same, S = L; then, int is a *self-interpreter*.

2.2 Reversible Interpreter

The usual definition of an interpreter in Eq. (1) does not carry over to interpreters written in a reversible language. Let us look at this surprising and important point more closely. Suppose that the implementation language L is a reversible language, which means that all L-programs compute *injective* functions. Thus, for all L-programs p and data $\mathtt{d}_1$, $\mathtt{d}_2$, we have

$$[\![\mathtt{p}]\!]^{\mathtt{L}}(\mathtt{d}_1) = [\![\mathtt{p}]\!]^{\mathtt{L}}(\mathtt{d}_2) \implies \mathtt{d}_1 = \mathtt{d}_2 . \quad (2)$$

That is, whenever the output of two applications of the same program is identical, so must be their input. In particular, consider the L-program int, which computes an injective function because of the reversibility of L. For all S-programs $\mathtt{p}_1$, $\mathtt{p}_2$ and data

†1 Following the notational convention [17], we omit the outermost brackets of data if it appears as the argument of a semantic functions, e.g., we write $[\![\mathtt{int}]\!]^{\mathtt{L}}(\mathtt{p}.\mathtt{d})$ instead of $[\![\mathtt{int}]\!]^{\mathtt{L}}((\mathtt{p}.\mathtt{d}))$.

d, we have

$$[\![\texttt{int}]\!]^{\texttt{L}}(\texttt{p}_1\,\texttt{.d}) = [\![\texttt{int}]\!]^{\texttt{L}}(\texttt{p}_2\,\texttt{.d}) \implies (\texttt{p}_1\,\texttt{.d}) = (\texttt{p}_2\,\texttt{.d})\,. \quad (3)$$

This implies that the program representation of the two S-programs is textually identical: $\texttt{p}_1 = \texttt{p}_2$. Using this implication and Eq. (1), we get $[\![\texttt{p}_1]\!]^{\texttt{S}}(\texttt{d}) = [\![\texttt{p}_2]\!]^{\texttt{S}}(\texttt{d}) \implies \texttt{p}_1 = \texttt{p}_2$ from Eq. (3). That is, S-programs that compute the same result for *some* d must have the *same* program representation as the L-data argument for int. However, source languages that do satisfy this condition are too restrictive and not very expressive. For example, either the injective function $f_1(x) = (x.\texttt{nil})$ or $f_2(x) = (\texttt{nil}.x)$ cannot be computed in such a language since $f_1(\texttt{nil}) = f_2(\texttt{nil})$, and the programs implementing f_1 and f_2 will not be syntactically equivalent.

The issue described above cannot be avoided when a universal reversible Turing machine (RTM) is considered, i.e., a self-interpreter for a reversible language [2]. A universal RTM needs to preserve the machine it simulates in the output [19]. These considerations lead to the following definition of a reversible interpreter.

Definition 1. Let L be a reversible language. Then, an L-program rint is a *reversible interpreter* of the language S in L iff for any S-program p and S-data d and d′,

$$[\![\texttt{rint}]\!]^{\texttt{L}}(\texttt{p.d}) = (\texttt{p.d}') \iff [\![\texttt{p}]\!]^{\texttt{S}}(\texttt{d}) = \texttt{d}'\,. \quad (4)$$

The reversible interpreter pairs p and d′ in the output, and it becomes symmetric: the input and output formats of the interpreter are the same.

The definition also implies that every S-program computes an *injective* function. Otherwise, an S-program p and the data $\texttt{d}_1$, $\texttt{d}_2$ exist s.t. $[\![\texttt{p}]\!]^{\texttt{S}}(\texttt{d}_1) = [\![\texttt{p}]\!]^{\texttt{S}}(\texttt{d}_2)$ and $\texttt{d}_1 \neq \texttt{d}_2$. By Eq. (4), we have

$$\begin{aligned} [\![\texttt{rint}]\!]^{\texttt{L}}(\texttt{p.d}_1) &= (\texttt{p.}[\![\texttt{p}]\!]^{\texttt{S}}(\texttt{d}_1)) \\ &= (\texttt{p.}[\![\texttt{p}]\!]^{\texttt{S}}(\texttt{d}_2)) \\ &= [\![\texttt{rint}]\!]^{\texttt{L}}(\texttt{p.d}_2)\,. \end{aligned}$$

However, $(\texttt{p.d}_1) \neq (\texttt{p.d}_2)$ because $\texttt{d}_1 \neq \texttt{d}_2$. Therefore, $[\![\texttt{rint}]\!]^{\texttt{L}}$ is not an injective function, and L cannot be reversible, which contradicts the assumption that L is reversible. Thus, the source language S must have a semantic function $[\![\cdot]\!]^{\texttt{S}}$ such that $[\![\texttt{p}]\!]^{\texttt{S}}$ is injective for any S-program p. Simulating irreversible languages in reversible languages requires an injectivization of the semantic function of the irreversible language. For example, this can be done by adding a trace to the output of the interpreter [24]. A reversible interpreter that does not require any output in addition to the representation of the program and its output is said to be functionally *clean*. Thus, an interpreter that returns an extra trace is not functionally clean.

2.3 Running Time Function

In analogy to the standard terminology, the efficiency of programs written in a reversible language L is defined by a *running time function*

$$time^{\texttt{L}} : \texttt{L-programs} \to (\texttt{L-data} \to \mathbb{N}_\perp), \quad (5)$$

where $time^{\texttt{L}}_{\texttt{p}}(\texttt{d})$ is the number of steps to compute p with the input d, provided the computation terminates; otherwise, undefined $\perp$. We denote $\mathbb{N} \cup \{\perp\}$ by $\mathbb{N}_\perp$ (see Sect. 3.3).

Finally, the definition of an efficient (i.e., linear-time) reversible interpreter in the sense of Jones coincides with the standard case [17]. The constant c in the definition may depend on the reversible interpreter rint but not on the source program p or data d.

Definition 2. A reversible interpreter rint of the language S in the language L is *linear-time* if there exists a constant c such that for any S-program p and S-data d,

$$time^{\texttt{L}}_{\texttt{rint}}(\texttt{p.d}) \leq c \cdot time^{\texttt{S}}_{\texttt{p}}(\texttt{d})\,. \quad (6)$$

3 A Structured Reversible Language

This section presents the concise *reversible language* R-WHILE, a structured imperative language with tree-structured data. We define its syntax (Fig. 1), denotational semantics (Fig. 2), and a syntax-directed program inverter (Fig. 4), which we will use later for the construction of the self-interpreter. The language is based on Jones' irreversible language WHILE [17]. We adopt denotational semantics for ease of comparison, as the semantics of WHILE is defined in a denotational manner. We assume that the reader is familiar with the basic notions of denotational semantics (e.g., [30]).

3.1 Example Program: List Reversal

A short example program reverse for list reversal illustrates several language features[†2].

```
read X;  (* reverse list X *)
  from  (=? Y nil)
```

†2 Here, we omit the do branch of the loop. See Sect. 3.2.

```
Expressions  ∋  E, F  ::=  X | d | cons E F | hd E | tl E | =? E F
Patterns     ∋  Q, R  ::=  X | d | cons Q R
Commands     ∋  C, D  ::=  X ^= E
                        |  Q <= R
                        |  C; D
                        |  if E then C else D fi F
                        |  from E do C loop D until F
Programs     ∋  P     ::=  read X; C; write Y
```

Fig. 1 Syntax of R-WHILE.

```
    loop  (Z.X) <= X;
          Y <= (Z.Y)
    until (=? X nil);
  write Y
```

The input of the program is read into the variable `X`, and the output is written from the variable `Y`. All variables are initially set to `nil`. The *reversible loop* `from...until` is entered by asserting that `Y` is `nil` and exited when `X` is `nil`. The commands in the loop body are repeated as long as the entry assertion and the exit test are false. The first command in the loop, a *reversible replacement*, deconstructs the value of `X` into its head and tail components; the second reversible replacement constructs a value by pairing the values of `Z` and `Y`. A reversible replacement sets the variables on the right-hand side to `nil` before binding the original value of the right-hand side to the variables on the left-hand side (e.g., `Z` is `nil` after the replacement `Y <= (Z.Y)`). All variables except the output variable `Y` must finally be `nil`. The application of the program to reverse a list is denoted by $[\![\texttt{reverse}]\!]^{\text{R-WHILE}}(\texttt{'a.('b.('c.nil))}) = \texttt{('c.('b.('a.nil)))}$, where the list elements `'a`, `'b`, and `'c` are atoms. Because of the entry assertion and reversible replacements, the program is reversible.

3.2 Syntax

The syntax of the language is simple (Fig. 1). A *program* `P` has a unique entry and exit point (`read`, `write`) and a command `C` as its body. The *input* and *output* of a program is read into and written from the variables in the entry and exit points.

The data domain $\mathbb{D}$ of a program is the smallest set that contains the atom `nil` and all pairs $(\texttt{d}_1\texttt{.d}_2) \in \mathbb{D}$ if $\texttt{d}_1, \texttt{d}_2 \in \mathbb{D}$. `Vars` is an infinite set of variable names. We use the conventions $\texttt{d}, \texttt{e}, \texttt{f}, \ldots \in \mathbb{D}$ and $\texttt{X}, \texttt{Y}, \texttt{Z}, \ldots \in$ `Vars`. For the sake of readability, we will use more than one atom in the programs. For example, the three atoms `'a`, `'b`, `'c` can be encoded in $\mathbb{D}$ as `nil`, `(nil.nil)`, `(nil.(nil.nil))`.

An *expression* is either a variable `X`, a constant `d`, or the application of an operator (the selectors `hd` and `tl`, constructor `cons`, and equality test `=?`). *Patterns* are a subset of expressions: a variable `X`, a data element `d`, or a pair of patterns `cons Q R`. For the sake of readability, we denote a pair of patterns `cons Q R` by `(Q.R)`. Patterns must be linear, i.e., not contain repeated variables. The language has no local variables.

In a *reversible assignment* `X ^= E`, the variable `X` on the left-hand side must not occur in the expression `E` on the right-hand side. A *reversible replacement* `Q <= R` updates the variables in `Q` using the values of the variables in `R`. In contrast to an assignment, a variable can appear on both sides of a replacement in the linear patterns `Q` and `R` (e.g., the variable `Y` in the replacement `Y <= (Z.Y)`).

The two structured control-flow operators of the language are *conditional* `if E then C else D fi F` and *loop* `from E do C loop D until F`. Loops have an additional *entry assertion* `E`, and conditionals have an additional *exit assertion* `F`. Tests and assertions have the same syntax, and both are evaluated at run time. For readability, we often omit the branches of conditionals and loops that contain only skip commands (e.g., `nil <= nil`).

As usual, we write the *list*

$$(\texttt{d}_1\ \texttt{d}_2\ \cdots\ \texttt{d}_n)$$

for $(\texttt{d}_1\texttt{.}(\texttt{d}_2\texttt{.}(\cdots\texttt{.}(\texttt{d}_n\texttt{.nil})\cdots)))$, and the improper list

$$(\texttt{d}_1\ \cdots\ \texttt{d}_{n-1}\texttt{.d}_n)$$

for $(\texttt{d}_1\texttt{.}(\cdots\texttt{.}(\texttt{d}_{n-1}\texttt{.d}_n)\cdots))$.

Similar to the notation for lists, we write the *pat-*

$$\mathcal{E}[\![\texttt{d}]\!]\sigma = \texttt{d}$$
$$\mathcal{E}[\![\texttt{X}]\!]\sigma = \sigma(\texttt{X})$$
$$\mathcal{E}[\![\texttt{hd E}]\!]\sigma = \texttt{e} \quad \text{if } \mathcal{E}[\![\texttt{E}]\!]\sigma = \texttt{(e.f)}$$
$$\mathcal{E}[\![\texttt{tl E}]\!]\sigma = \texttt{f} \quad \text{if } \mathcal{E}[\![\texttt{E}]\!]\sigma = \texttt{(e.f)}$$
$$\mathcal{E}[\![\texttt{cons E F}]\!]\sigma = (\mathcal{E}[\![\texttt{E}]\!]\sigma.\mathcal{E}[\![\texttt{F}]\!]\sigma)$$
$$\mathcal{E}[\![\texttt{=? E F}]\!]\sigma = \begin{cases}\texttt{true} & \text{if } \mathcal{E}[\![\texttt{E}]\!]\sigma = \mathcal{E}[\![\texttt{F}]\!]\sigma \\ \texttt{false} & \text{otherwise}\end{cases}$$

$$\mathcal{Q}[\![\texttt{d}]\!]\sigma = (\texttt{d}, \sigma)$$
$$\mathcal{Q}[\![\texttt{X}]\!](\sigma \uplus \{\texttt{X} \mapsto \texttt{d}\}) = (\texttt{d}, \sigma \uplus \{\texttt{X} \mapsto \texttt{nil}\})$$
$$\mathcal{Q}[\![\texttt{cons Q R}]\!]\sigma = ((\texttt{d}_1.\texttt{d}_2), \sigma_2) \text{ where } (\texttt{d}_1, \sigma_1) = \mathcal{Q}[\![\texttt{Q}]\!]\sigma \ \wedge\ (\texttt{d}_2, \sigma_2) = \mathcal{Q}[\![\texttt{R}]\!]\sigma_1$$

$$\mathcal{C}[\![\texttt{X ^= E}]\!](\sigma \uplus \{\texttt{X} \mapsto \texttt{d}\}) = \sigma \uplus \{\texttt{X} \mapsto \texttt{d} \odot \mathcal{E}[\![\texttt{E}]\!]\sigma\}$$
$$\mathcal{C}[\![\texttt{Q <= R}]\!]\sigma = \mathcal{Q}[\![\texttt{Q}]\!]^{-1}(\mathcal{Q}[\![\texttt{R}]\!]\sigma)$$
$$\mathcal{C}[\![\texttt{C; D}]\!]\sigma = \mathcal{C}[\![\texttt{D}]\!](\mathcal{C}[\![\texttt{C}]\!]\sigma)$$
$$\mathcal{C}\left[\!\!\left[\begin{array}{l}\texttt{if E then C}\\ \texttt{else D fi F}\end{array}\right]\!\!\right]\sigma = \begin{cases}\sigma' & \text{if } \mathcal{E}[\![\texttt{E}]\!]\sigma = \texttt{true} \ \wedge\ \sigma' = \mathcal{C}[\![\texttt{C}]\!]\sigma \ \wedge\ \mathcal{E}[\![\texttt{F}]\!]\sigma' = \texttt{true} \\ \sigma' & \text{if } \mathcal{E}[\![\texttt{E}]\!]\sigma = \texttt{false} \ \wedge\ \sigma' = \mathcal{C}[\![\texttt{D}]\!]\sigma \ \wedge\ \mathcal{E}[\![\texttt{F}]\!]\sigma' = \texttt{false}\end{cases}$$
$$\mathcal{C}\left[\!\!\left[\begin{array}{l}\texttt{from E do C}\\ \texttt{loop D until F}\end{array}\right]\!\!\right]\sigma = \sigma' \quad \text{if } \mathcal{E}[\![\texttt{E}]\!]\sigma = \texttt{true} \ \wedge\ \sigma' = \mathit{fix}(F)(\sigma)$$
$$\text{where } F(\varphi) = \{\,(\sigma, \sigma_1) \mid \sigma_1 = \mathcal{C}[\![\texttt{C}]\!]\sigma \ \wedge\ \mathcal{E}[\![\texttt{F}]\!]\sigma_1 = \texttt{true}\,\} \cup \{\,(\sigma, \sigma_3) \mid \sigma_1 = \mathcal{C}[\![\texttt{C}]\!]\sigma \ \wedge\ \mathcal{E}[\![\texttt{F}]\!]\sigma_1 = \texttt{false} \ \wedge\ \sigma_2 = \mathcal{C}[\![\texttt{D}]\!]\sigma_1 \ \wedge\ \mathcal{E}[\![\texttt{E}]\!]\sigma_2 = \texttt{false} \ \wedge\ \sigma_3 = \varphi(\sigma_2)\,\}$$

$$\mathcal{P}[\![\texttt{P}]\!]D = D' \quad \text{if P is } \texttt{read X; C; write Y} \ \wedge\ \mathcal{C}[\![\texttt{C}]\!](\sigma^{\texttt{P}}_{\texttt{X}}(D)) = \sigma^{\texttt{P}}_{\texttt{Y}}(D')$$

Fig. 2 Denotational semantics of R-WHILE.

tern list

$$(\texttt{Q}_1\ \texttt{Q}_2\ \cdots\ \texttt{Q}_\texttt{n})$$

for $(\texttt{Q}_1.(\texttt{Q}_2.(\cdots.(\texttt{Q}_\texttt{n}.\texttt{nil})\cdots)))$, and the improper pattern list

$$(\texttt{Q}_1\ \cdots\ \texttt{Q}_{\texttt{n-1}}.\texttt{Q}_\texttt{n})$$

for $(\texttt{Q}_1.(\cdots.(\texttt{Q}_{\texttt{n-1}}.\texttt{Q}_\texttt{n})\cdots))$.

The expression $(\texttt{list E}_1\ \cdots\ \texttt{E}_\texttt{n})$ will be used as shorthand for the expression $(\texttt{cons E}_1 \cdots (\texttt{cons E}_\texttt{n}\ \texttt{nil})\cdots)$, and the expression $(\texttt{cons*}\ \texttt{E}_1 \cdots \texttt{E}_{\texttt{n-1}}\ \texttt{E}_\texttt{n})$ is used for the expression $(\texttt{cons E}_1 \cdots (\texttt{cons E}_{\texttt{n-1}}\ \texttt{E}_\texttt{n})\cdots)$.

3.3 Semantics

We define the denotational semantics for the language R-WHILE. We begin with some notation and then explain the four semantic functions in Fig. 2.

The sets of variables occurring in a program P and an expression E are denoted by $\mathit{Var}(\texttt{P})$ and $\mathit{Var}(\texttt{E})$, respectively. Boolean values are denoted by two distinct elements $\texttt{false}, \texttt{true} \in \mathbb{D}$ for readability. The undefined value is denoted by $\bot$, and a set $X \cup \{\bot\}$ is denoted by $X_\bot$. We use $\mathbb{D}_\bot$ as the value domain.

A store σ for P is a function from $\mathit{Var}(\texttt{P})$ to $\mathbb{D}_\bot$. The set of all stores for P is denoted by $\texttt{Stores}^{\texttt{P}}$. The store $\sigma\backslash\texttt{X}$ is identical to σ, except that it maps the variable X to $\bot$. The store $\sigma \uplus \sigma'$ is the disjoint union of the bindings in the stores σ and σ', which have no variables in common. The undefined store, which returns $\bot$ for any variable, is also written as $\bot$.

The store $\sigma^{\texttt{P}}_{\texttt{X}}(D)$ binds the variable X to D and all other variables in $\mathit{Var}(\texttt{P}) = \{\texttt{X}, \texttt{Y}_1, \ldots, \texttt{Y}_\texttt{n}\}$ to nil:

$$\sigma^{\texttt{P}}_{\texttt{X}}(D) = [\texttt{X} \mapsto D, \texttt{Y}_1 \mapsto \texttt{nil}, \ldots, \texttt{Y}_\texttt{n} \mapsto \texttt{nil}]\ . \quad (7)$$

The semantic functions are defined for expressions, patterns, commands, and programs (Fig. 2)[†3]:

$\mathcal{E}$: $\texttt{Expressions} \to (\texttt{Stores}^{\texttt{P}} \hookrightarrow \mathbb{D}_\bot)$
$\mathcal{Q}$: $\texttt{Patterns} \to (\texttt{Stores}^{\texttt{P}} \hookrightarrow \mathbb{D}_\bot \times \texttt{Stores}^{\texttt{P}}_\bot)$
$\mathcal{C}$: $\texttt{Commands} \to (\texttt{Stores}^{\texttt{P}} \hookrightarrow \texttt{Stores}^{\texttt{P}}_\bot)$
$\mathcal{P}$: $\texttt{Programs} \to (\mathbb{D} \hookrightarrow \mathbb{D}_\bot)$

A program p denotes the partial function $\mathcal{P}[\![\texttt{p}]\!]$. We sometimes write $[\![\texttt{p}]\!]^{\text{R-WHILE}}$ instead to make explicit that p is a program in the language R-WHILE.

The *expression evaluation* $\mathcal{E}$ evaluates an expression E in a store σ to a value in $\mathbb{D}_\bot$. The evaluation

†3 An injective mapping from A to B is denoted by $A \hookrightarrow B$.

of a variable X, constant d and the application of an operator such as head and tail selectors hd and tl, the pairing cons, and the equality test =? are as expected. $\mathcal{E}$ can evaluate to $\perp$ because the values of hd E and tl E are undefined if the value of E is nil.

The *pattern evaluation* $\mathcal{Q}$ evaluates a pattern Q in a store σ to a value-store pair. Every variable used in Q is bound to nil in the resulting store. This moves every value used in constructing the resulting value out of the store. We have $\mathcal{Q}[\![\mathtt{Q}]\!]\sigma = (\mathcal{E}[\![\mathtt{Q}]\!]\sigma, \sigma')$ for some σ. For any pattern Q, the denotation $\mathcal{Q}[\![\mathtt{Q}]\!]$ is an injective function. Thus, the inverse function, $\mathcal{Q}[\![\mathtt{Q}]\!]^{-1}$, is unique. The *inverse pattern evaluation*, $\mathcal{Q}[\![\mathtt{Q}]\!]^{-1}(\mathtt{d}, \sigma')$, takes a value-store pair to a new store in which all of the variables in Q are bound to the corresponding component of the value d, provided those variables were bound to nil in the store σ'. We have $\mathcal{Q}[\![\mathtt{Q}]\!]^{-1}(\mathcal{Q}[\![\mathtt{Q}]\!]\sigma) = \sigma$ if $\mathcal{Q}$ is defined for Q and σ. Pattern evaluation and its inverse will be used for defining the semantics of reversible replacements.

We now turn to the *command evaluation* $\mathcal{C}$ and explain some design choices. Assignments in an irreversible language are destructive. They overwrite the value of the variable on the left-hand side, and the original value cannot be reconstructed after the assignment. Thus, they cannot be used in a reversible language (unless one logs the original values to injectivize the computation). Instead, we use a *reversible assignment* X ^= E that sets X to the value of E if X is nil and sets X to nil if the values of X and E are equal. In the former case, the value of E is duplicated, and in the latter case, the equality of the values of X and E is asserted.

We formalize the semantics by defining an update operator $\odot$:

$$\mathtt{d} \odot \mathtt{e} = \begin{cases} \mathtt{e} & \text{if } \mathtt{d} = \mathtt{nil} \\ \mathtt{nil} & \text{if } \mathtt{d} = \mathtt{e} \neq \mathtt{nil}\,. \end{cases} \tag{8}$$

A reversible assignment can then be defined by

$$\mathcal{C}[\![\mathtt{X \ \hat{}= \ E}]\!](\sigma \uplus [\mathtt{X} \mapsto \mathtt{d}]) = \sigma \uplus [\mathtt{X} \mapsto \mathtt{d}'], \tag{9}$$

where $\mathtt{d}' = \mathtt{d} \odot \mathcal{E}[\![\mathtt{E}]\!]\sigma$. Note that σ, in which E is evaluated, contains *no* binding for X on the left-hand side. This is ensured by the disjoint union $\sigma \uplus [\mathtt{X} \mapsto \mathtt{d}]$. Hence, if X occurs in E, the assignment is undefined. If this were not the case, an invalid assignment X ^= X would set X to nil for any value of X, and $\mathcal{C}[\![\mathtt{X \ \hat{}= \ X}]\!]$ would not be injective.

Applying a reversible assignment X ^= E twice in a row restores the original value of X. This holds because for any d and e, if $\mathtt{d} \odot \mathtt{e}$ is defined, then

$$(\mathtt{d} \odot \mathtt{e}) \odot \mathtt{e} = \mathtt{d}\,. \tag{10}$$

A reversible assignment is an instance of a *reversible update* [31]. The operator $\odot$ is injective in its first argument; it is not commutative.

Remark: The reversible assignment operator ^= is reminiscent of the bitwise exclusive-or assignment operator in Janus [35], which zero-clears the value of the variable on the left-hand side if it is equal to the value of the right-hand side expression. Unlike the Janus assignment operator, which always returns a valid store in constant time given the value of the right-hand-side expression (the values are fixed-size integers), the R-WHILE assignment operator can return an undefined store $\perp$ and take time depending on the unbounded size of the tree-structured data owing to the equality test.

A *reversible replacement* Q <= R updates the variables in the linear patterns Q and R on both sides of the replacement. $\mathcal{C}[\![\mathtt{Q \ <= \ R}]\!]\sigma$ is defined by first evaluating $\mathcal{Q}[\![\mathtt{R}]\!]\sigma$, which results in $(\mathtt{d}, \sigma')$, where $\mathtt{d} = \mathcal{E}[\![\mathtt{R}]\!]\sigma$ and σ' is σ, except that all variables in $\mathit{Var}(\mathtt{R})$ are nil. This is followed by the inverse evaluation $\mathcal{Q}[\![\mathtt{Q}]\!]^{-1}(\mathtt{d}, \sigma')$, which yields σ''—the final store of the reversible replacement—in which all variables in $\mathit{Var}(\mathtt{Q})$, which must be nil in σ', are bound to parts of d such that $\mathtt{d} = \mathcal{E}[\![\mathtt{Q}]\!]\sigma''$.

The computational structure defining a reversible replacement resembles the original Bennett method [6] without copying where d is the intermediate garbage. However, the forward computation $\mathcal{Q}[\![\mathtt{R}]\!]$ and backward uncomputation $\mathcal{Q}[\![\mathtt{Q}]\!]^{-1}$ are not inverses of each other. On the other hand, the value of R before the operation matches the value of Q after the operation. Specifically, for any σ and σ'',

$$\mathcal{C}[\![\mathtt{Q \ <= \ R}]\!]\sigma = \sigma'' \iff \mathcal{E}[\![\mathtt{R}]\!]\sigma = \mathcal{E}[\![\mathtt{Q}]\!]\sigma''\,. \tag{11}$$

We note that a reversible replacement can be simulated in R-WHILE by using several reversible assignments and auxiliary variables. However, reversible replacements are convenient for programming and lead to shorter programs.

At a *reversible conditional* if E then C else D fi F, the control flow branches depending on the test E. If true, the command C is executed, and the assertion F must be true. If false, the command D is executed, and the assertion F must be false. If the value of F at the exit does not correspond to

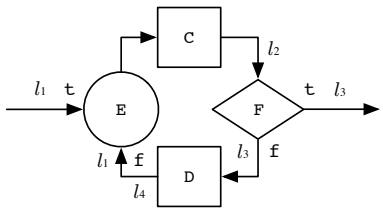

Fig. 3 A flowchart for a reversible while loop.

the value of `E` at the entry, the conditional is undefined. The assertion `F` ensures that the execution of a conditional is backward deterministic. Both expressions, `E` and `F`, are evaluated at run time. The definition of the semantics is straightforward.

A *reversible loop* `from E do C loop D until F` can be illustrated by a reversible flowchart (Fig. 3) [32]. The informal semantics is as follows. At the entry of the loop, the assertion `E` must be `true`, indicated by `t` in the flowchart, and the command `C` is executed. Then, if the test `F` is `true`, the loop terminates. If `F` is `false`, the command `D` is executed, and the assertion `E` must be `false`, indicated by `f` in the flowchart. If the value of `E` does not match the required value, the loop is undefined. Different types of loops can be programmed with this general loop construct. For example, when `C` is a skip command, the loop becomes a `while` loop; when `D` is a skip command, the loop becomes a `do-until` loop. In the semantics, the stores before and after loop execution are related by the least fixed point of F. The function $F(\varphi)$ is defined by the union of the two sets, which correspond to the termination and iteration of the loop execution.

Labels l_1–l_4 in Fig. 3 will be used later in Sect. 4 to simulate loops in the reversible interpreter.

3.4 A Tool: A Program Inverter

A *program inverter* is usually not available as a programming tool in a conventional language (cf. [11] [12]) but is straightforward to define for `R-WHILE` (cf. [34]). It is useful for the construction of the programs in this paper.

We immediately have the inverse program in a reversible language. A simple example is a replacement that pairs `nil` with the value of `X`. If `X` is a unary number (consisting of a list of `nil`), this increments `X`. Then, the inverse of the replacement decrements `X`.

```
X <= (nil.X)  (* increment X *)
(nil.X) <= X  (* decrement X *)
```

A *syntax-directed program inverter* $\mathcal{I}$ for `R-WHILE` is defined in Fig. 4. Each command is inverted *locally* by a recursive descent over the program structure. Reversible assignments are self-inverse; therefore, no change is needed. The patterns in a reversible replacement switch sides. The inverse of a command sequence is the reversed sequence of its inverted commands. Tests and assertions change roles in control-flow operators. A program is inverted by swapping the input and output variables and inverting the body.

For example, we obtain the *inverse program* of our example program by $\mathcal{I}[\![\texttt{reverse}]\!]$:

```
read Y;  (* inverse of list reversal *)
  from  (=? X nil)
  loop  (Z.Y) <= Y;
        X <= (Z.X)
  until (=? Y nil);
write X
```

The inverse program performs the same function as the original program, which is not surprising because list reversal is self-inverse. The inverse program is textually identical to `reverse`, except that the variables `X` and `Y` switched places. (Clearly, this is not always the case for an inverse program.) The inversion of programs, such as `reverse`, is straightforward, and we will use it extensively for building the reversible interpreter in the next section.

Owing to the syntactic nature of $\mathcal{I}$, applying it twice returns the original program $\texttt{P} = \mathcal{I}[\![\mathcal{I}[\![\texttt{P}]\!]]\!]$. A program and its inverse take the same number of computation steps. For any `P` and `d`, we have

$$time_{\texttt{P}}(\texttt{d}) = time_{\mathcal{I}[\![\texttt{P}]\!]}([\![\texttt{P}]\!](\texttt{d}))\,. \qquad (12)$$

We prove the correctness of $\mathcal{I}$ by showing the correctness of command inversion and then that of program inversion. The following lemma implies that $\mathcal{I}$ is a program inverter for `R-WHILE`.

Lemma 1. For any `R-WHILE` command `C` and stores σ and σ', if $\sigma' = \mathcal{C}[\![\texttt{C}]\!]\sigma$, then $\sigma = \mathcal{C}[\![\mathcal{I}[\![\texttt{C}]\!]]\!]\sigma'$.

Proof. If $\sigma' = \bot$, the statement holds vacuously. Otherwise, we use induction over the number of times used to obtain σ' by unfolding the semantic function $\mathcal{C}$. We use an induction hypothesis in which all occurrences of $\mathcal{C}$ on the right-hand side in Fig. 2 satisfy the statement and show for each case that the left-hand side satisfies the statement.

$$
\begin{array}{lcl}
\mathcal{I}[\![\texttt{X ^= E}]\!] & = & \texttt{X ^= E} \\
\mathcal{I}[\![\texttt{Q <= R}]\!] & = & \texttt{R <= Q} \\
\mathcal{I}[\![\texttt{C; D}]\!] & = & \mathcal{I}[\![\texttt{D}]\!]\texttt{;}\ \mathcal{I}[\![\texttt{C}]\!] \\
\mathcal{I}[\![\texttt{if E then C else D fi F}]\!] & = & \texttt{if F then}\ \mathcal{I}[\![\texttt{C}]\!]\ \texttt{else}\ \mathcal{I}[\![\texttt{D}]\!]\ \texttt{fi E} \\
\mathcal{I}[\![\texttt{from E do C loop D until F}]\!] & = & \texttt{from F do}\ \mathcal{I}[\![\texttt{C}]\!]\ \texttt{loop}\ \mathcal{I}[\![\texttt{D}]\!]\ \texttt{until E} \\
\mathcal{I}[\![\texttt{read X; C; write Y}]\!] & = & \texttt{read Y;}\ \mathcal{I}[\![\texttt{C}]\!]\texttt{;}\ \texttt{write X}
\end{array}
$$

Fig. 4 A syntax-directed program inverter for R-WHILE.

(The case for loops uses induction on the natural numbers. This simplifies the correctness proof of a program inverter [35], which uses induction on proof trees constructed by operational rules.)

In the case of a reversible assignment, assume that $\sigma' = \mathcal{C}[\![\texttt{X ^= E}]\!]\sigma$ holds. Then, $\mathcal{C}[\![\mathcal{I}[\![\texttt{X ^= E}]\!]]\!]\sigma' = \sigma[\texttt{X} \mapsto (\sigma(\texttt{X}) \odot \mathcal{E}[\![\texttt{E}]\!](\sigma\backslash\texttt{X})) \odot \mathcal{E}[\![\texttt{E}]\!](\sigma\backslash\texttt{X})] = \sigma$, where the last equality holds by Eq. (10). In the case of a reversible replacement, by the definition of $\mathcal{C}$, we have

$$
\begin{array}{rcl}
\sigma_{\texttt{Q}} = \mathcal{C}[\![\texttt{Q <= R}]\!]\sigma_{\texttt{R}} & \Longleftrightarrow & \sigma_{\texttt{Q}} = (\mathcal{Q}[\![\texttt{Q}]\!])^{-1}(\mathcal{Q}[\![\texttt{R}]\!]\sigma_{\texttt{R}}) \\
& \Longleftrightarrow & \sigma_{\texttt{R}} = (\mathcal{Q}[\![\texttt{R}]\!])^{-1}(\mathcal{Q}[\![\texttt{Q}]\!]\sigma_{\texttt{Q}}) \\
& \Longleftrightarrow & \sigma_{\texttt{R}} = \mathcal{C}[\![\texttt{R <= Q}]\!]\sigma_{\texttt{Q}}.
\end{array}
$$

This implies $\sigma' = \mathcal{C}[\![\texttt{Q <= R}]\!]\sigma$ iff $\sigma = \mathcal{C}[\![\texttt{R <= Q}]\!]\sigma'$.

In the case of a reversible conditional, a simple case analysis proves the correctness.

In the case of a reversible loop, assume that $\sigma' = \mathcal{C}[\![\texttt{from E do C loop D until F}]\!]\sigma$ holds. Then, by unfolding, we have the entry condition $\mathcal{E}[\![\texttt{E}]\!]\sigma = \texttt{true}$, the exit condition $\mathcal{E}[\![\texttt{F}]\!]\sigma' = \texttt{true}$, and the store update by the loop body $\sigma' = \mathit{fix}(F)\sigma$. By induction on n (and the induction hypothesis on the number of unfoldings of the semantic function $\mathcal{C}$), if $\sigma_2 = F^n(\emptyset)(\sigma_1)$, then $\sigma_1 \in F'^n(\emptyset)(\sigma_2)$ for all σ_1, σ_2. Here, F' is defined by F, in which C and D are replaced with $\mathcal{I}[\![\texttt{C}]\!]$ and $\mathcal{I}[\![\texttt{D}]\!]$, respectively. Therefore, if $\sigma' = \mathit{fix}(F)\sigma$, then $\sigma = \mathit{fix}(F)\sigma'$. By the definition of $\mathcal{I}$ and the evaluation of E and F above, we have $\sigma = \mathcal{C}[\![\mathcal{I}[\![\texttt{from E do C loop D until F}]\!]]\!]\sigma'$. □

The correctness of the program inverter $\mathcal{I}$ directly follows from the lemma.

Corollary 1. For any R-WHILE program P and data d and d′, if $\texttt{d}' = \mathcal{P}[\![\texttt{P}]\!](\texttt{d})$, then $\texttt{d} = \mathcal{P}[\![\mathcal{I}[\![\texttt{P}]\!]]\!](\texttt{d}')$.

Proof. Let P be `read X; C; write Y`. We assume that $\texttt{d}' = \mathcal{P}[\![\texttt{P}]\!](\texttt{d})$ holds. We have $\mathcal{C}[\![\texttt{C}]\!](\sigma^{\texttt{P}}_{\texttt{X}}(\texttt{d})) = \sigma^{\texttt{P}}_{\texttt{Y}}(\texttt{d}')$, and by Lemma 1, $\mathcal{C}[\![\mathcal{I}[\![\texttt{C}]\!]]\!](\sigma^{\texttt{P}}_{\texttt{Y}}(\texttt{d}')) = \sigma^{\texttt{P}}_{\texttt{X}}(\texttt{d})$. By $\sigma^{\mathcal{I}[\![\texttt{P}]\!]}_{\texttt{Y}} = \sigma^{\texttt{P}}_{\texttt{Y}}$, we have $\mathcal{P}[\![\mathcal{I}[\![\texttt{P}]\!]]\!](\texttt{d}') = \texttt{d}$. □

3.5 Reversibility of R-WHILE

For a language to be reversible, it should be injective at a level of certain atomic constructs so that computational sequences of atomic constructs are locally forward and backward deterministic. This atomic level of reversible execution differs for concrete languages. We will show that R-WHILE is reversible at the level of commands, which is the natural level of execution for an imperative language. The following theorem shows that the execution of any command is injective, even though expression evaluation is not injective. The injectivity of commands implies that of programs.

Theorem 1. For any R-WHILE command C and any stores σ_1 and σ_2, if $\mathcal{C}[\![\texttt{C}]\!]\sigma_1 = \mathcal{C}[\![\texttt{C}]\!]\sigma_2$, then $\sigma_1 = \sigma_2$.

Proof. By Lemma 1, we have the desired equality

$$\sigma_1 = \mathcal{C}[\![\mathcal{I}[\![\texttt{C}]\!]]\!](\mathcal{C}[\![\texttt{C}]\!]\sigma_1) = \mathcal{C}[\![\mathcal{I}[\![\texttt{C}]\!]]\!](\mathcal{C}[\![\texttt{C}]\!]\sigma_2) = \sigma_2\,.$$

□

Corollary 2. For any R-WHILE program P and any data d and d′, if $\mathcal{P}[\![\texttt{P}]\!](\texttt{d}) = \mathcal{P}[\![\texttt{P}]\!](\texttt{d}')$, then $\texttt{d} = \texttt{d}'$.

Proof. Let P be `read X; C; write Y`. Assume $\mathcal{P}[\![\texttt{P}]\!](\texttt{d}) = \mathcal{P}[\![\texttt{P}]\!](\texttt{d}')$. By definition, $\mathcal{C}[\![\texttt{C}]\!](\sigma^{\texttt{P}}_{\texttt{X}}(\texttt{d})) = \mathcal{C}[\![\texttt{C}]\!](\sigma^{\texttt{P}}_{\texttt{X}}(\texttt{d}'))$, and by Theorem 1, $\sigma^{\texttt{P}}_{\texttt{X}}(\texttt{d}) = \sigma^{\texttt{P}}_{\texttt{X}}(\texttt{d}')$. Since $\sigma^{\texttt{P}}_{\texttt{X}}$ is injective, the equality $\texttt{d} = \texttt{d}'$ holds. □

3.6 Rewrite Rules as Syntactic Sugar

Programs often become more readable by using a reversible version of the *rewrite rules* [17] that simultaneously update several variables $\texttt{X}_1, \ldots, \texttt{X}_\texttt{n}$ as syntactic sugar:

`rewrite [X1,...,Xn] by Rule1 ... Rulem.`

Rule $\texttt{Rule}_\texttt{j}$ is either

1. $\texttt{[Q}_1\texttt{,...,Q}_\texttt{n}\texttt{]} \Rightarrow \texttt{[E}_1\texttt{,...,E}_\texttt{n}\texttt{]}$, or
2. $\texttt{[Q}_1\texttt{,...,Q}_\texttt{n}\texttt{]} \Rightarrow \texttt{C;}$.

The intended meaning is as follows. If the values of $\texttt{X}_1, \ldots, \texttt{X}_\texttt{n}$ match the patterns $\texttt{Q}_1, \ldots, \texttt{Q}_\texttt{n}$ of a rule $\texttt{Rule}_\texttt{j}$, which are tried sequentially, the variables in $\texttt{Q}_\texttt{i}$ are bound. Then, depending on the form of $\texttt{Rule}_\texttt{j}$, either $\texttt{X}_1, \ldots, \texttt{X}_\texttt{n}$ are bound to the

Reversible rewriting:

```
from (=? T′ nil) loop
  rewrite [ T, T′, N ]
    [ ('(l).T),           T′,  N ] ⇒ [ T,                cons '(l) T′, cons nil N ]
    [ (('n L.R).T),       T′,  N ] ⇒ [ cons* L R '(m) T, cons '(m) T′,          N ]
    [ ('(m).T), (R L '(m).T′), N ] ⇒ [ T,      cons (cons* 'n L R) T′,          N ]
until (=? T nil)
```

R-WHILE:

```
from (=? T′ nil) loop
  if (=? '(l) (hd T)) then
    ('(l).T) <= T;       (* leaf *)
    T′ <= ('(l).T′);
    N <= (nil.N)
  else                   (* node *)
    if (=? 'n (hd (hd T))) then
      (('n L.R).T) <= T;
      T <= (L R '(m).T);
      T′ <= ('(m).T′)
    else               (* ascent *)
      ('(m).T) <= T;
      (R L '(m).T′) <= T′;
      T′ <= (('n L.R).T′)
    fi (=? '(m) (hd T′))
  fi (=? '(l) (hd T′))
until (=? T nil)
```

A binary tree and its representation:

```
    / \
  (l) / \
    (l) (l)

t = ('n '(l).('n '(l).'(l)))
```

Rewriting steps for `t` (3 leaves):

```
[ (('n '(l).('n '(l).'(l)))),          nil,    nil ] ⇒
[ ('(l) ('n '(l).'(l)) '(m)),      ('(m)),    nil ] ⇒
[ (('n '(l).'(l)) '(m)),      ('(l) '(m)), (nil¹) ] ⇒
[ ('(l) '(l) '(m) '(m)), ('(m) '(l) '(m)), (nil¹) ] ⇒
[ ('(l) '(m) '(m)), ('(l) '(m) '(l) '(m)), (nil²) ] ⇒
[ ('(m) '(m)), ('(l) '(l) '(m) '(l) '(m)), (nil³) ] ⇒
[ ('(m)),      (('n '(l).'(l)) '(l) '(m)), (nil³) ] ⇒
[ nil,         (('n '(l).('n '(l).'(l)))), (nil³) ]
```

Fig. 5 Example: Counting the leaves in a binary tree. The reversible rewriting rules are syntactic sugar that is expanded into R-WHILE. The steps of an example run are shown.

values of $E_1, \ldots, E_n$, correspondingly, or the command C is executed, which may also update X_i. Patterns with repeated variables match if the variables match with the same value.

Similar to the *orthogonality condition* of case expressions [10], the range of the output of a rule must statically differ from the output of all previous rules in order for the rewriting to be injective and thus must be implementable by expansion into nested reversible conditionals. We do not require a general expansion mechanism. We only use rules as syntax sugar where atoms make the left and right sides statically orthogonal and thus easily expandable.

Example rewriting: tree traversal

The example in Fig. 5 illustrates R-WHILE and the use of the reversible rewrite rules for counting the leaves in a *binary tree* by performing an in-order tree traversal. The same reversible programming method can be used to traverse more complicated tree structures and to perform more involved calculations. In the reversible interpreter, we will use this method to compute, among others, the value of an expression given as an *abstract syntax tree*. We now describe the reversible rewrite rules and their implementation in R-WHILE.

A rewrite tuple [T,T′,N] is a stack of trees T, a stack of processed trees T′, and a counter N. The initial tuple is [(t),nil,nil], and the final tuple is [nil,(t),n], where t is the representation of a binary tree, as shown in Fig. 5, and n is the number of leaves in t (a unary number). The iteration of the rules is implemented by a reversible loop.

There are three cases. If a leaf '(l) is on top of T, the counter N is incremented, and the leaf is moved to T′. When an internal node tagged 'n is on top of T, its two subtrees L and R and a marker '(m) are placed on T. The marker is also placed on T′ to make the right side disjoint from the other rules. After the subtrees are traversed, the marker

`'(m)` is on top of `T`, and the node is reassembled on `T'`. The rewrite rules are orthogonal on both sides: on the left side, the tags on top of `T` are disjoint, and on the right side, the tags on top of `T'` are disjoint. The rules define a reversible state transition system.

Owing to their orthogonality, the rewrite rules are easily expanded into nested reversible conditionals (Fig. 5). The three cases are guarded by the tests and assertions given by the rules. There is no error checking; all trees are assumed to be well-formed.

4 Building a Reversible Interpreter

In this section, we introduce the Bennett compute–uncompute method and two variants—the reversible store management of the interpreter and the data representation of `R-WHILE` programs.

4.1 Bennett Reversible Simulation and a Generalization

In a reversible computation, no information can be deleted, neither data nor control-flow information. By preserving the information lost in an irreversible computation step, e.g., by tracing the value of a variable overwritten by an assignment, a *reversible simulation* of an irreversible computation can be obtained. When we want the result of the reversibilized computation but wish to undo all side effects it had on the store, e.g., the trace, we use the *Bennett method* [6], a method for clean reversible programming. The original method consists of three parts, where the third part undoes all changes made to the store by the first part:

1. the reversibilized computation `C`,
2. saving of the result by `COPY`, and
3. the computation of the inverse $\mathcal{I}[\![\texttt{C}]\!]$.

To abstract from this frequently recurring compute–uncompute program pattern, the macro `BENNETT(C,COPY)` (Fig. 6) is used. We use such C-like function macros to keep our programs short and readable. The parameterized macros are expanded before program execution. A macro is textually replaced by the macro body, in which formal arguments are replaced by actual arguments and inversion of commands is carried out. The actual arguments are completely expanded before macro expansion. Self-referential macros are not allowed so that macro expansion terminates.

```
macro BENNETT(C,COPY)        ≡ C; COPY; I[[C]]
macro COMP-INV(C,D)          ≡ C;       I[[D]]
macro COMP-CP-INV(C,COPY,D)  ≡ C; COPY; I[[D]]
```

Fig. 6 Bennett method and two variants.

By allowing different reversible computations, `C` and `D`, in the first and third part, we can cover a wider range of program patterns. In Fig. 6, we define the generalized composition macro `COMP-CP-INV(C,COPY,D)` for this program pattern and its copy-free version `COMP-INV(C,D)` in the case where no `COPY` is needed between the command `C` and the inverse command $\mathcal{I}[\![\texttt{D}]\!]$. The Bennett method `BENNETT` and the copy-free composition `COMP-INV` are instances of the generalized composition `COMP-CP-INV`:

```
BENNETT(C,COPY) = COMP-CP-INV(C,COPY,C)
COMP-INV(C,D)   = COMP-CP-INV(C,SKIP,D)
```

where the macro `SKIP` has no computational effect. If `COPY` is self-inverse, the original `BENNETT(C,COPY)` is also self-inverse. However, the copy-free and generalized variants are not. In the next subsections, we will see how the macros are used in reversible programming.

4.2 Store Management

We use a list `vl` bound to `Vl` to hold the values of the variables in a source program. The `i`-th element of `vl` is the value of the `i`-th variable $\texttt{X}_\texttt{i}$ (the first element in the list is indexed by one). Initially, `vl` is a list of `nil` of length n, which is the number of variables in a program. Without loss of generality, we assume that all of the variables in a program are named $\texttt{X}_\texttt{i}$, where $\texttt{i} \geq 1$. Every program has at least one variable that is read and written. We access the values in `Vl` using four reversible macros: `UPDATE`, `LOOKUP`, `GETV`, and `PUTV`.

First, we specify the functionality of `UPDATE` and `LOOKUP`. For any value $\sigma(\texttt{Vl}) = (\texttt{d}_1\ \texttt{d}_2\ \ldots\ \texttt{d}_\texttt{n})$, we require, for any index `I` ranging over 1 to n and variable `X`, that

$$\mathcal{C}[\![\texttt{LOOKUP(I,X)}]\!]\sigma = \sigma[\texttt{X} \mapsto \sigma(\texttt{X}) \odot \texttt{d}_\texttt{i}] \tag{13}$$

$$\mathcal{C}[\![\texttt{UPDATE(I,X)}]\!]\sigma = \sigma[\texttt{Vl} \mapsto (\texttt{d}_1\ \ldots\ (\texttt{d}_\texttt{i} \odot \sigma(\texttt{X}))\ \ldots\ \texttt{d}_\texttt{n})] \tag{14}$$

where $\sigma(\texttt{I}) = \texttt{i}$. The macro `LOOKUP(I,X)` reversibly

$$\mathcal{D}[\![\texttt{read X}_\texttt{i}\texttt{; C; write X}_\texttt{j}]\!] = ((\texttt{'var.}\mathcal{D}_{nat}[\![\texttt{i}]\!])\ \mathcal{D}_{cmd}[\![\texttt{C}]\!]\ (\texttt{'var.}\mathcal{D}_{nat}[\![\texttt{j}]\!]))$$

$$\begin{aligned}
\mathcal{D}_{cmd}[\![\texttt{X}_\texttt{i}\ \texttt{\^{}= E}]\!] &= (\texttt{'asn}\ (\texttt{'var.}\mathcal{D}_{nat}[\![\texttt{i}]\!]).\mathcal{D}_{exp}[\![\texttt{E}]\!])\\
\mathcal{D}_{cmd}[\![\texttt{Q <= R}]\!] &= (\texttt{'rep}\ \mathcal{D}_{exp}[\![\texttt{Q}]\!].\mathcal{D}_{exp}[\![\texttt{R}]\!])\\
\mathcal{D}_{cmd}[\![\texttt{C; D}]\!] &= (\texttt{'seq}\ \mathcal{D}_{cmd}[\![\texttt{C}]\!].\mathcal{D}_{cmd}[\![\texttt{D}]\!])\\
\mathcal{D}_{cmd}[\![\texttt{if E then C else D fi F}]\!] &= (\texttt{'cond}\ \mathcal{D}_{exp}[\![\texttt{E}]\!]\ \mathcal{D}_{cmd}[\![\texttt{C}]\!]\ \mathcal{D}_{cmd}[\![\texttt{D}]\!]\ \mathcal{D}_{exp}[\![\texttt{F}]\!])\\
\mathcal{D}_{cmd}[\![\texttt{from E do C loop D until F}]\!] &= (\texttt{'loop}\ \mathcal{D}_{exp}[\![\texttt{E}]\!]\ \mathcal{D}_{cmd}[\![\texttt{C}]\!]\ \mathcal{D}_{cmd}[\![\texttt{D}]\!]\ \mathcal{D}_{exp}[\![\texttt{F}]\!])
\end{aligned}$$

$$\begin{aligned}
\mathcal{D}_{exp}[\![\texttt{d}]\!] &= (\texttt{'val.d}) & \mathcal{D}_{exp}[\![\texttt{cons E F}]\!] &= (\texttt{'cons}\ \mathcal{D}_{exp}[\![\texttt{E}]\!].\mathcal{D}_{exp}[\![\texttt{F}]\!])\\
\mathcal{D}_{exp}[\![\texttt{X}_\texttt{i}]\!] &= (\texttt{'var.}\mathcal{D}_{nat}[\![\texttt{i}]\!]) & \mathcal{D}_{exp}[\![\texttt{=? E F}]\!] &= (\texttt{'eq}\ \mathcal{D}_{exp}[\![\texttt{E}]\!].\mathcal{D}_{exp}[\![\texttt{F}]\!])\\
\mathcal{D}_{exp}[\![\texttt{hd E}]\!] &= (\texttt{'hd.}\mathcal{D}_{exp}[\![\texttt{E}]\!]) & & \\
\mathcal{D}_{exp}[\![\texttt{tl E}]\!] &= (\texttt{'tl.}\mathcal{D}_{exp}[\![\texttt{E}]\!]) & \mathcal{D}_{nat}[\![\texttt{i}]\!] &= (\texttt{nil}_1\ \cdots\ \texttt{nil}_\texttt{i})
\end{aligned}$$

Fig. 7 Mapping R-WHILE programs to their data representation.

```
macro LOOKUP(J,X) ≡ BENNETT(POS(J,Y),X ^= Y)
macro UPDATE(J,X) ≡ BENNETT(POS(J,Y),Y ^= X)
macro GETV(J,X)   ≡ BENNETT(POS(J,Y),X <= Y)
macro PUTV(J,X)   ≡ BENNETT(POS(J,Y),Y <= X)

macro POS(J,Y) ≡
  from (=? JJ nil) loop
    (U.Vl) <= Vl;   (* Move head    *)
    Vr <= (U.Vr);   (*  of Vl to Vr *)
    JJ <= (nil.JJ)  (* JJ + 1       *)
  until (=? JJ J);
  (Y.Vr) <= Vr
```

Fig. 8 Store management macros.

updates the value of X in σ by the i-th value of vl. The macro UPDATE(I,X) reversibly updates the i-th value of the list bound to Vl by the value of the variable X. The operator $\odot$ is the reversible update defined in Eq. 10. LOOKUP and UPDATE are self-inverse, a property which we will exploit as a reversible programming technique to reset values. (Other data structures could be used for the value list vl, but the list representation does not affect the asymptotic time complexity of the interpreter if the number of variables in R-WHILE is fixed.)

Fig. 8 defines the two parameterized macros LOOKUP(J,X) and UPDATE(J,X) using the original Bennett method BENNETT parameterized by the auxiliary macro POS(J,Y) in Fig. 8, where the temporary variable JJ traverses the index of vl, and the temporary variable U is used to move the head elements bound to Vl to the head bound to Vr. We require that all temporary variables, i.e., all variables that are neither macro parameters nor the store variable Vl, are initially and finally nil-cleared. Under this requirement, Eq. 13 and Eq. 14 hold. Because the store management must be reversible, both lookup and update first move forward and then backward on the value list. Since these two movements are inverses of each other, *one* implementation by POS combined with the Bennett method suffices. Between the forward and backward movements, the value in Y is used as desired.

The macro GETV(J,X) places the J-th value of the value list vl in the variable X, which must be nil-cleared. Afterwards, the J-th value of the value list bound to Vl is nil. For any value $\sigma(\texttt{Vl}) = (\texttt{d}_1\ \texttt{d}_1\ \ldots\ \texttt{d}_\texttt{n})$, we require

$$\begin{aligned}&\mathcal{C}[\![\texttt{GETV(J,X)}]\!](\sigma[\texttt{X} \mapsto \texttt{nil}]) =\\ &\quad \sigma[\texttt{X} \mapsto \texttt{d}_\texttt{j}][\texttt{Vl} \mapsto (\texttt{d}_1\ \ldots\ \texttt{d}_{\texttt{j-1}}\ \texttt{nil}\ \texttt{d}_{\texttt{j+1}}\ \ldots\ \texttt{d}_\texttt{n})],\end{aligned} \tag{15}$$

where $\sigma(\texttt{J}) = \texttt{j}$. The macro GETV(J,X) has the same functionality as a sequence of lookup and update, LOOKUP(J,X); UPDATE(J,X) (a copy-free composition COMP-INV(LOOKUP(J,X),UPDATE(J,X))). However, the original method BENNETT(POS(J,Y),X <= Y) with a reversible replacement is a more direct and efficient implementation in this case (Fig. 8).

The macro PUTV(J,X) is the inverse of the macro GETV(J,X):

$$\texttt{PUTV(J,X)} \equiv \mathcal{I}[\![\texttt{GETV(J,X)}]\!].$$

Although reversible programming does not allow discarding of the values left in an "unorderly state" but requires their restoration to a well-defined state, as in the case of the value list vl

after a variable lookup, this does not necessarily require writing the program text twice. The Bennett method and the invertibility of the programming language reduce this to a minimum, and somewhat surprisingly, we gain a new form of modularity, as illustrated by these macros for store management.

4.3 Program Representation

The mapping of R-WHILE programs to their data representation is straightforward (Fig. 7): each expression and command is mapped to a tagged list so that they can be distinguished easily. Let {'var, 'seq, 'asn, 'rep, 'cond, 'loop, 'val, 'cons, 'hd, 'tl, 'eq} be the set of tags. The natural numbers i and j that index variables are transformed into unary numbers represented by lists of nil. They are used to access the value list representing the store. The mapping is injective. In the rest of the paper, we assume that the largest index is used for the variable for the input and output in the program representation. It is easy to transform any program representation to this form without changing its asymptotic time complexity.

Example data representation: List reversal

We show the data representation of the list reversal program

```
(x ('seq
    ('loop
     ('eq  y . ('val . nil))
     ('rep ('val . nil) . ('val . nil))
     ('seq ('rep ('cons z . x) . x) .
           ('rep y . ('cons z . y)))
     ('eq  x . ('val . nil))) .
    ('rep x . y))
 x)
```

where the data representation of the variables are as follows:

```
x = ('var . (nil nil nil))
y = ('var . (nil nil))
z = ('var . (nil))
```

The program is based on the list reversal program in Sect. 3.1, but the reversible replacement is inserted at the last line of the body in order for the variable X used for the input and output to have the largest index in its data representation x. We assume the sequence of the list reversal program is left associative, and the structure in the data representation is maintained as the left associative 'seq. Because the do branch of the list reversal program is omitted (cf. Sect. 3.2), the data representation of the skip command nil <= nil is used.

5 Main Program: Program Execution

We now build the *reversible self-interpreter* for R-WHILE. The main challenge is to organize the interpreter to be fully reversible. No values may be discarded, including any part of the source program. To this end, we utilize several programming techniques: the Bennett method [6] and the variants we introduced above, the reversible tree traversal and the rewriting rules as syntactic sugar (Sect 3.6), the reversible flowchart simulation [32], and the reversible programming techniques of the Janus interpreter [35]. Together, they play a crucial role in minimizing the reversible programming effort. We will now describe the reversible interpreter in more detail.

5.1 Disassembly–reassembly of programs

A conventional interpreter switches between decomposing and simulating parts of the source program. Typically, while interpreting a part (e.g., an expression), that part is consumed to compute its effect (e.g., the value of the expression). If the same part is needed again later, it is taken from a "backup copy" of the source program. However, this classic *use–delete* interpretation strategy cannot be used in a reversible interpreter. (Deletion as in an irreversible language is not possible.)

Recall from Def. 1 that a reversible interpreter has a *symmetric* input and output format: it reads and writes program–data pairs. The data d in the input pair (p.d) are transformed into the data d′ by simulating p, which is returned unchanged in the output pair (p.d′). Thus, even though p needs to be decomposed, one cannot just dump its used parts somewhere but has to handle them so that p can be reassembled. (Again, a naive solution is not possible: make a copy of p; after interpretation, delete all dumped parts and return the copy.)

The way the disassembly and reassembly of a source program is handled depends on the implementation language. The method employed in R-WHILE is the one illustrated in Fig. 5, i.e., move all parts onto a stack during the descent over an *abstract syntax tree* (expression, pattern, command) and reassemble the tree from that stack during the

```
read PD;  (* input  (p.d)  *)
 BENNETT(INIT(Cd,PD),EVAL-CMD(Cd,Cd′));
write PD  (* output (p.d') *)

macro INIT(Cd,PD) ≡
 (Pgm.D) <= PD;
 (('var.I′) Cd ('var.J′)) <= Pgm;
 Vl ^= I′;
 PUTV(I′,D)
```

Fig. 9 Reversible self-interpreter `ri` for `R-WHILE`.

ascent. In a reversible language with RAM, such as arrays in Janus, one can instead reversibly position and reposition pointers in the syntactic structure [35].

5.2 Main program of the interpreter

The main program of the reversible interpreter `ri` is shown in Fig. 9. It consists of a macro `BENNETT` that invokes the command simulation `EVAL-CMD` and administers the initial and final stores of the program. It is a reversible implementation of the semantic function $\mathcal{P}$ in Fig. 2: $[\![\texttt{ri}]\!]^{\texttt{R-WHILE}}(\texttt{p.d}) = \mathcal{P}[\![\texttt{p}]\!](\texttt{d})$.

The macro `EVAL-CMD(Cd,Cd′)`, which we explain later, simulates a list of commands. Initially, the variable `Cd` holds a singleton list of the command in the body of the source program representation. The command is disassembled on `Cd` and reassembled on `Cd′` so that, in the end, the original command can be taken from the singleton list `Cd′` and returned to `Cd`.

The initialization macro `INIT(Cd,PD)` decomposes the program–data pair `(p.d)` read into `PD`. The program in `Pgm` is further decomposed into the input and output variables and the command in its body. The value list bound to `Vl` is initialized to be a list of `nil` of length n, the number of variables in the source program (recall that the index `I′` as well as the index `J′` obtained from the input program `p` is the largest index occurring in `p` and that all indices are represented by unary numbers). The input data `d` are moved to the `I′`-th position in the value list bound to `Vl` by the store-management macro `PUTV` (Fig. 8). The inverse of `INIT(Cd,PD)` performs post-processing. If all elements in the value list are nil except for the `I′`-th (i.e., `J′`-th) position, the non-nil value is paired with the program representation and set to `PD`, and `Vl` is nil-cleared.

```
macro EVAL-EXP(E,V,H) ≡
 LOOP(E,E′,EVAL-EXP-STEP(E,E′,St,H));
 (V.nil) <= St

macro EVAL-PAT(Q,V) ≡
 LOOP(Q,Q′,EVAL-PAT-STEP(Q,Q′,St));
 (V.nil) <= St

macro EVAL-CMD(Cd,Cd′) ≡
 LOOP(Cd,Cd′,EVAL-CMD-STEP(Cd,Cd′))

macro LOOP(L,L′,Body) ≡
 L <= (L.nil);
 from (=? L′ nil) loop
   Body
 until (=? L nil);
 (L.nil) <= L′
```

Fig. 10 Macros for expression, pattern, and command evaluation.

5.3 Three mini-interpreters

The main components of `ri` are the reversible interpreters for the three sublanguages of `R-WHILE`: expressions, patterns, and commands. They implement the semantic functions $\mathcal{E}$, $\mathcal{Q}$, and $\mathcal{C}$ in Fig. 2. Just like any reversible interpreter (cf. Def. 1), they preserve the interpreted syntactic object in their result. Their implementation follows the same programming principles and uses rewrite rules to define the traversal of the abstract syntax trees. They assume that the variable `Vl` holds the values of the variables. The macros that iterate the rewrite rules are shown in Fig. 10. We begin with the mini-interpreter for pattern evaluation, the smallest of the three interpreters.

Pattern evaluation

The evaluation of a pattern on the right side of a reversible replacement builds a new value and never discards a value (there are no selectors `hd` and `tl` in a pattern). Because $\mathcal{Q}[\![\texttt{q}]\!]$ is injective, it suffices to preserve the pattern `q` in the result to obtain a reversible implementation.

The rewrite rules of the macro `EVAL-PAT-STEP` in Fig. 11 define a reversible state transition system

that simulates $\mathcal{Q}$ without the use of recursion. A rewrite tuple `[Q,Q',St]` consists of a stack of patterns `Q`, a stack of processed patterns `Q'`, and a value stack `St`. The rewriting rules test the next pattern on `Q` and update the state accordingly.

The traversal of a pattern works in essentially the same way as the in-order traversal of a binary tree in Fig. 5. A pattern on `Q` is disassembled during the descent over its syntactic structure and reassembled on `Q'` during the ascent that builds the final value from the intermediate values on `St`. The initial tuple is `[(q),nil,nil]`, and the final tuple is `[nil,(q),(v)]`, where `q` is the pattern to evaluate, and `v` is its value. The store in which `q` is evaluated is the value list `vl`. The value of a variable (`'var.J`) in a pattern is moved from `Vl` to `V` by the macro `GETV` (Sect. 4.2) and pushed on `St`. The macro `GETV` swaps values. Thus, `Vl` is updated by the evaluation, as required by the semantic function.

Pattern evaluation by iterating the rewrite rules in Fig. 11 is based on the following invariant:

$$\texttt{[(q.qs),qs',st]} \Rightarrow^* \texttt{[qs,(q.qs'),(v.st)]} \quad (16)$$

iff $\mathcal{Q}[\![\texttt{Q}]\!]\sigma = (\texttt{v}, \sigma')$, where $\sigma \simeq \texttt{vl}$ and $\sigma' \simeq \texttt{vl}'$ hold for the value lists before and after applying the rewrite rules to evaluate the representation `q` of `Q`. Suppose that `q` is on top of `qs`, and `qs'` and `st` are some values. After a number of rule iterations, `q` will be on top of `qs'` and `v` on top of `st`, and `vl` is updated. The store equivalence $\sigma \simeq (\texttt{d}_1\ \texttt{d}_2 \ldots \texttt{d}_n)$ is such that $\sigma(\texttt{X}_i) = \texttt{d}_i$ for any `i`.

The iteration of the rewrite rules is implemented by the macro `EVAL-PAT` in Fig. 10. The rewriting terminates because the pattern stack eventually becomes `nil`. The rewriting is reversible because the tags on `Q` and `Q'` differ on both sides.

Expression evaluation

The semantic function $\mathcal{E}$ for expression evaluation is non-injective. For example, the evaluation of an expression `hd E` removes the tail of the value of `E`. To obtain a reversible implementation of $\mathcal{E}$ in `R-WHILE`, we preserve all values, which would be lost if we use an irreversible evaluator, on a separate history stack `H`. That is, we push the tail value to `H`. Because this value is irrelevant for the final value of `hd E`, it is so-called garbage data, which we uncompute afterwards using the Bennett method.

The rewrite rules of the macro `EVAL-EXP-STEP` in Fig. 12 define a reversible state transition system that simulates $\mathcal{E}$. A rewrite tuple `[Ex,Ex',St,H]` consists of a stack of expressions `Ex`, a stack of processed expressions `Ex'`, a stack of intermediate values `St`, and a history stack `H`. The rewriting rules test the next expression on `Ex` and update the state accordingly. The value of a variable in an expression is copied from `Vl` to `V` by the macro `LOOKUP` (Sect. 4.2) and pushed on `St` (unlike a pattern, where the value is moved). The initial rewrite tuple is `[(e),nil,nil,nil]`, where `e` is the expression to evaluate. The rewrite rules are applied until the tuple has the form `[nil,(e),(v),h]`, where `v` is the value of `e`, and `h` is a history stack. Except for maintaining a history stack, the rules work in essentially the same way as the rules for pattern evaluation in Fig. 11.

The iteration of the rewrite rules is implemented by the macro `EVAL-EXP` in Fig. 10. The history stack is uncomputed by the original Bennett method: `BENNETT(EVAL-EXP(E,V,H),...)`. The reader can see this in all five places in Fig. 13, where the expressions are evaluated (assignments, conditionals, loops).

Expression evaluation by iterating the rewrite rules in Fig. 12 is based on the following invariant:

$$\texttt{[(e.ex),ex',st,h]} \Rightarrow^* \texttt{[ex,(e.ex'),(v.st),h']} \quad (17)$$

iff $\mathcal{E}[\![\texttt{E}]\!]\sigma = \texttt{v}$, where the value list in which the representation `e` of `E` is evaluated is $\sigma \simeq \texttt{vl}$. Suppose that `e` is on top of `ex`, and `ex'`, `st`, and `h` are some values. After a number of rule iterations, `e` will be on top of `ex'` and `v` on top of `st`. The list `vl` is unchanged; the history `h` is updated.

The rules are orthogonal on both sides, as required for reversible rewriting (Sect. 3.6). On the left side, all tags on top of `Ex` are disjoint (`'var`, `'hd`, etc.), and on the right side, all tags on top of `Ex'` are disjoint (`'var`, `'hdm`, etc.). The markers that trigger the evaluation of an operation (`'hdm`, `'tlm`, `'consm`, `'eqm`) are placed in singleton lists; thus, all rules have a uniform format. The evaluation halts after a finite number of rewrite steps.

Command execution

What remains is to define is the semantic function $\mathcal{C}$ for command execution. As before, the reversible simulation will preserve the command in the result. The rewrite rules in Fig. 13 define a reversible state transition system that simulates $\mathcal{C}$. A rewrite tuple consists of a command stack `Cd` and

```
macro EVAL-PAT-STEP(Q,Q′,St) ≡
rewrite [ Q, Q′, St ] by
 [ (('val.V).Q),    Q′, St ] ⇒ [ Q,                cons (cons 'val V) Q′,          cons V St ]
 [ (('var.J).Q),    Q′, St ] ⇒ { GETV(J,V);           Q′ <= (('var.J).Q′);       St <= (V.St)}
 [ (('cons E.F).Q), Q′, St ] ⇒ [ cons* E F '(consm) Q, cons '(consm) Q′,                St ]
 [ ('(consm).Q), (F E '(consm).Q′), (W V.St) ]
                             ⇒ [ Q,           cons (cons* 'cons E F) Q′, cons (cons V W) St ]
```

Fig. 11 Evaluation step of a pattern on the right side of a reversible replacement.

```
macro EVAL-EXP-STEP(Ex,Ex′,St,H) ≡
rewrite [ Ex, Ex′, St, H ] by
 [ (('val.V).   Ex), Ex′, St, H ] ⇒ [ Ex, cons (cons 'val V) Ex′,              cons V St, H ]
 [ (('var.J).   Ex), Ex′, St, H ] ⇒ { LOOKUP(J,V);   Ex′ <= (('var.J).Ex′);   St <= (V.St)   }
 [ (('hd.E).    Ex), Ex′, St, H ] ⇒ [ cons* E '(hdm) Ex, cons '(hdm) Ex′,             St, H ]
 [ ('(hdm).     Ex), (E '(hdm).Ex′), ((V.W).St), H ]
                                  ⇒ [ Ex, cons (cons 'hd E) Ex′,        cons V St, cons W H ]
 [ (('tl.E).    Ex), Ex′, St, H ] ⇒ [ cons* E '(tlm) Ex, cons '(tlm) Ex′,             St, H ]
 [ ('(tlm).     Ex), (E '(tlm).Ex′), ((V.W).St), H ]
                                  ⇒ [ Ex, cons (cons 'tl E) Ex′,        cons W St, cons V H ]
 [ (('cons E.F).Ex), Ex′, St, H ] ⇒ [ cons* E F '(consm) Ex, cons '(consm) Ex′,       St, H ]
 [ ('(consm).   Ex), (F E '(consm).Ex′), (W V.St), H ]
                                  ⇒ [ Ex, cons (cons* 'cons E F) Ex′, cons (cons V W) St, H ]
 [ (('eq E.F).  Ex), Ex′, St, H ] ⇒ [ cons* E F '(eqm) Ex,   cons '(eqm) Ex′,         St, H ]
 [ ('(eqm).     Ex), (F E '(eqm).  Ex′), (W V.St), H ]
                                  ⇒ { Ex′ <= (('eq E.F).Ex′);  U ^= (=? V W);  St <= (U.St);
                                                                  H <= (V W.H)}
```

Fig. 12 Expression evaluation step using the history H.

a stack of processed commands Cd′. The effect of command execution is to update the value list vl. The initial tuple is [(c),nil], and the final tuple is [nil,(c)], where c is the command to execute with the value list vl. The rewriting rules work in essentially the same way as the other rewriting rules, except that the rules for simulating loops are more involved. The rules make use of the macros for expression and pattern evaluation.

The iteration of the rewrite rules is implemented by the macro EVAL-CMD in Fig. 10. The macro is used in the main program of the interpreter in Fig. 9 to evaluate the body of the source program. All rewriting rules are pairwise orthogonal on both sides. The simulation of a loop by the rewrite rules may not terminate.

Command execution by iterating the rewrite rules in Fig. 13 is based on the following invariant:

$$\texttt{[(c.cd),cd']} \Rightarrow^{*} \texttt{[cd,(c.cd')]} \qquad (18)$$

iff $\mathcal{C}[\![\texttt{C}]\!]\sigma = \sigma'$, where $\sigma \simeq$ vl and $\sigma' \simeq$ vl′ hold for the value list before and after applying the rewrite rules to execute the representation c of C.

We now explain the rewrite rules in more detail. A *reversible assignment* Xk ^= E is represented by ('asn ('var.k′).e), where k′ is the unary number of k, and e is the representation of E. The evaluation of E is simulated by the original Bennett reversible simulation using the macro EVAL-EXP. In the Bennett simulation, the macro EVAL-EXP computes the value of E, the macro UPDATE reversibly updates the k-th position of Vl by the value, and the inverse of the macro EVAL-EXP clears the value of E set on V.

A *reversible replacement* Q <= R is represented by ('rep q.r), where q and r are the representations of Q and R. The execution of reversible replacements is performed by a copy-free Bennett simulation. First, the macro EVAL-PAT simulates the evaluation of R, clearing all variables in Vl that occur in R and setting V to the value of R. Second, the inverse of the macro EVAL-PAT updates all variables in Vl that occur in Q so that the value of Q in the updated value list is equal to V and clears V.

A *command sequence* C; D is represented by ('seq c.d), where c and d are the representations

```
macro EVAL-CMD-STEP(Cd,Cd′) ≡
rewrite [ Cd, Cd′ ] by
 (* Assignment, replacement, sequence *)
 [ (('asn ('var.K).E).Cd),          Cd′  ] ⇒ { BENNETT(EVAL-EXP(E,V,H),UPDATE(K,V));
                                               Cd′ <=                    (('asn ('var.K).E).Cd′)}
 [ (('rep Q.R).Cd),                 Cd′  ] ⇒ { COMP-INV(EVAL-PAT(R,V),EVAL-PAT(Q,V));
                                               Cd′ <=                           (('rep Q.R).Cd′)}
 [ (('seq C.D).Cd),                 Cd′  ] ⇒ [ cons* C D '(seqm) Cd,          cons '(seqm) Cd′ ]
 [ ('(seqm).Cd),       (D C '(seqm).Cd′) ] ⇒ [ Cd,                   cons (cons* 'seq C D) Cd′ ]

 (* Conditional *)
 [ (('cond E C D F).Cd),            Cd′  ] ⇒ { BENNETT(EVAL-EXP(E,V,H),W ^= V);
                                               if W then C′ ^= C else C′ ^= D fi W;
                                               Cd  <=                  (C′ ('condm E C D F).Cd);
                                               Cd′ <=                           (('condm W).Cd′)}
 [ (('condm E C D F).Cd), (C′ ('condm W).Cd′) ]
                                          ⇒ { if W then C′ ^= C else C′ ^= D fi W;
                                               BENNETT(EVAL-EXP(F,V,H),W ^= V)
                                               Cd′ <=                      (('cond E C D F).Cd′)}
 (* Loop *)
 [ (('loop E C D F).Cd),            Cd′  ] ⇒ [ cons (list 'l1 E C D F) Cd,  cons '(l1 true) Cd′ ]
 [ (('l1   E C D F).Cd),   (('l1 W).Cd′) ] ⇒ { BENNETT(EVAL-EXP(E,V,H),W ^= V);
                                               Cd  <=                    (C '(l2)          .Cd);
                                               Cd′ <=                      (('l2   E C D F).Cd′)}
 [ ('(l2).Cd),     (C ('l2 E C D F).Cd′) ] ⇒ { BENNETT(EVAL-EXP(F,V,H),W ^= V);
                                               Cd  <=                      (('l3   W)      .Cd);
                                               Cd′ <=                      (('l3   E C D F).Cd′)}
 [ ('(l3 false).Cd), (('l3 E C D F).Cd′) ] ⇒ [ cons* D '(l4) Cd,  cons (list 'l4   E C D F) Cd′ ]
 [ ('(l3 true) .Cd), (('l3 E C D F).Cd′) ] ⇒ [ Cd,                cons (list 'loop E C D F) Cd′ ]
 [ ('(l4).Cd),     (D ('l4 E C D F).Cd′) ] ⇒ [ cons (list 'l1 E C D F) Cd, cons '(l1 false) Cd′ ]
```

Fig. 13 Command execution step using expression and pattern evaluation.

of the commands `C` and `D`. Two rules are used for the execution. The first rule decomposes the sequence and pushes `c` and `d` onto `Cd` together with the marker `'(seqm)` to mark the end of the sequence. The marker is also pushed on `Cd′`. The two commands `c` and `d` are then processed by repeated application of the macro `EVAL-CMD-STEP`. After the execution of `c` and `d`, the marker `'(seqm)` appears on top of `Cd`, and the second rule reconstructs the original sequence `('seq c.d)` on `Cd′`. The following rules use markers in a similar fashion.

A *conditional* `if E then C else D fi F` is represented by `('cond e c d f)`, where `e`, `c`, `d`, and `f` are the representations of `E`, `C`, `D`, and `F`, respectively. First, the value of the test expression `E` is computed by the original Bennett method: setting the value of `E` on `V` by `EVAL-EXP` with the garbage set on `H`, the value `V` is copied to `W`, and the value `V` and the garbage `H` are cleared by the inverse of `EVAL-EXP`. Depending on `W`, the result of the test—either the then-branch `c` or the else-branch `d`—is pushed onto `Cd`, and the value of `W` is kept on `Cd′`. In the second rule, the code of the processed branch $\texttt{Cd}_1$ is cleared depending on the value of `W`, and the value of `W` is cleared by the value of the assertion `F` (if it is correct), which is also computed by the original Bennett simulation of expression evaluation. If this value is different from the value of the assertion `F`, the computation abnormally halts at the reversible assignment `W ^= V`. The codes of the conditionals are reconstructed and moved onto the processed codes `Cd′`. The palindromic structure appears on the right-hand sides of the two rules, although the expressions to be evaluated (`E` and `F`) are different in the first and last Bennett simulations.

A *loop* `from E do C loop D until F` is represented by `('loop e c d f)`, where `e`, `c`, `d`, and `f` are the representations of `E`, `C`, `D`, and `F`, respectively. For the interpretation of a loop, we follow

```
[ ((’seq (’loop _ _ _ _) (’rep _ . _))),                                    nil ] ⇒
[ ((’loop _ _ _ _) (’rep _ . _) ’(seqm)),                                (’(seqm)) ] ⇒
[ ((’l1 _ _ _ _) (’rep _ . _) ’(seqm)),                  (’(l1   true) . ’(seqm)) ] ⇒
[ (’(l2) (’rep _ . _) ’(seqm)),                          ((’l2 _ _ _ _) . ’(seqm)) ] ⇒
[ (’(l3.false) (’rep _ . _) ’(seqm)),                    ((’l3 _ _ _ _) . ’(seqm)) ] ⇒
[ ((’asn y . (’val . nil)) ’(l4) (’rep _ . _) ’(seqm)), ((’l4 _ _ _ _) . ’(seqm)) ] ⇒
[ (’(l4) (’rep _ . _) ’(seqm)), ((’asn y . (’val . nil)) ’l4 _ _ _ _) . ’(seqm)) ] ⇒
[ ((’l1 _ _ _ _) (’rep _ . _) ’(seqm)),                  (’(l1  false) . ’(seqm)) ] ⇒

... the above five lines repeat ...

[ (’(l2) (’rep _ . _) ’(seqm)),                          ((’l2 _ _ _ _) . ’(seqm)) ] ⇒
[ (’(l3.true) (’rep _ . _) ’(seqm)),                     ((’l3 _ _ _ _) . ’(seqm)) ] ⇒
[ ((’rep _ . _) ’(seqm)),                               ((’loop _ _ _ _) . ’(seqm)) ] ⇒
[ (’(seqm)),                              ((’rep _ . _) (’loop _ _ _ _) . ’(seqm)) ] ⇒
[ nil,                                      ((’seq (’loop _ _ _ _) (’rep _ . _))) ]
```

Fig. 14 Rewriting the trace of a command sequence of the list reversal program (underscores are “don’t care” elements).

the flowchart in Fig. 3. The tags `’l1`, `’l2`, `’l3`, and `’l4` are associated with the labels l_1 before the assertion `E` (two places), l_2 before the test `F`, l_3 after the test `F` (two places), and l_4 after the command `D`. Expression evaluations of tests and assertions are performed by the original Bennett simulation. An assertion `E` is placed in the circle that merges the two control flows labeled l_1. If a control flow comes from the edge labeled `t` (`f`, resp.), the evaluation of `E` must be `true` (`false`, resp.). This is simulated by the rule `’l1`, in which the expected value of the assertion `E` clears `W` by the value of `V`. The rule pushes the representation of the do-branch `C` and tag `’l2` onto `Cd` and the tag `’l2` onto the processed code `Cd′`. A test `F` is placed in the lozenge that divides the control flow into two branches. If control moves to the edge labeled `t` (`f`, resp.), the evaluation of `F` must be `true` (`false`, resp.). (Otherwise, the computation abnormally halts at the reversible assignment `W ^= V`.) This is simulated by one rule for `’l2` and two rules for `’l3`. The rule `’l2` pushes a pair of tags `’l3` and the value of `F` onto the code to process `Cd`. At the rule `’l3`, if `’l3` is paired with `true`, the representation of the loop-branch `D` is pushed onto the code to process `Cd`. When `’l3` is paired with `false`, it is the end of the loop. Then, `D` is moved to the processed code `Cd′`. It should be noted that the rules `’l2` and `’l4` have nonlinear input patterns, and the first `’l3` rule has a nonlinear output pattern. The rule `’l4` is used to clear the code of the loop branch and moves control to the test of the loop with the expected assertion `false` paired with `’l1`.

The simulation of an unstructured flowchart by a structured flowchart was used to give an alternative global proof for the structured program theorem by Cooper [7]. In his construction, the position of the control in the source program is stored in a sequence of Boolean variables. Similarly, we use tags to simulate the control in the source programs.

Example: Trace of list reversal

To illustrate the macro `EVAL-CMD-STEP`, we show a tracing of rewriting a part of the list reversal program in Fig. 14, whose program representation was shown in Sect. 4. 3. Which rewrite rule to apply depends on the tags in the head of the commands to be processed.

6 Efficiency of the Interpreter

In this study, we use a simple cost model for the language `R-WHILE`. The unfolding of a semantic function is counted as a step ($\mathcal{E}$, $\mathcal{Q}$, $\mathcal{C}$, and $\mathcal{P}$ in Fig. 2), and an equality test for data, $\mathtt{d}_1 = \mathtt{d}_2$, is counted as $\min(|\mathtt{d}_1|, |\mathtt{d}_2|)$ steps[†4]. Unfolding $(\mathcal{Q}[\![\mathtt{Q}]\!])^{-1}$ takes the same number of steps as unfolding $\mathcal{Q}[\![\mathtt{Q}]\!]$.

The reversible interpreter `ri` for `R-WHILE` is not linear-time according to Def. 2 because the variable access by the store management macros `LOOKUP`, `UPDATV`, `PUTV`, and `GETV` takes time proportional to

†4 We denote the size of data `d` by $|\mathtt{d}|$ and the size of the data in the variable `X` by $|\mathtt{X}|$.

the number of variables in the source program. To show the existence of a *linear-time* reversible self-interpreter, consider an r-Turing complete subset of R-WHILE, in which the number of variables is bounded.

Let m be the maximum number of variables that occurs in the reversible interpreter ri and in an RTM interpreter implemented in R-WHILE. Define a subset R-WHILE-M of R-WHILE with at most m variables. Since R-WHILE is an instance of the reversible flowchart language framework [34] and has dynamic allocation of data, we can construct an RTM interpreter in R-WHILE. By definition, this program is also an R-WHILE-M program. Hence, R-WHILE-M is r-Turing complete. As a consequence of limiting the number of variables, variable access in ri is constant time, i.e., $\mathrm{O}(m)$. Therefore, ri is a linear-time reversible interpreter for R-WHILE-M.

Theorem 2. The program ri is a linear-time reversible self-interpreter for an r-Turing complete language R-WHILE-M: there exists a constant c such that for any R-WHILE-M program p and data d, we have

$$time_{\mathtt{ri}}^{\text{R-WHILE-M}}(\mathtt{p.d}) \leq c \cdot time_{\mathtt{p}}^{\text{R-WHILE-M}}(\mathtt{d})\,. \qquad (19)$$

Proof. A simple induction on the structure of patterns shows that the direct evaluation of patterns, $\mathcal{Q}[\![\mathtt{Q}]\!]$, takes time proportional to $|\mathtt{Q}|$. This also holds for $\mathcal{Q}[\![\mathtt{Q}]\!]^{-1}$. Therefore, the command execution $\mathcal{C}[\![\mathtt{Q \ <= \ R}]\!]$ takes time proportional to $|\mathtt{Q}| + |\mathtt{R}|$.

The macros LOOKUP, UPDATE, GETV, and PUTV take time proportional to the size of the store, m, because of the loop of their common macro POS.

It is easy to see that the time for interpreting a pattern by the macro EVAL-PAT is bounded by the time for direct pattern evaluation. The macro EVAL-PAT-STEP is performed in constant time because of branching and merging of the control flow, and all other operations on the right side of the rewrite rules are constant time. The size of the data representation of a pattern is proportional to the size of the pattern, and each application of EVAL-PAT-STEP decrements the size of the pattern stack in the rewrite tuple. Hence, the macro EVAL-PAT(Q,V) takes time proportional to the size of the pattern represented by Q.

An analogous analysis shows that the time for interpreting expressions by the macro EVAL-EXP is bounded by the time for directly evaluating expressions. It should be noted that the time for interpreting an equality test =? is bounded by the time of the equality test in direct evaluation.

For the case of commands, an analogous analysis holds in addition to an induction on the number of steps. We assume as an induction hypothesis that the time for interpreting the branches of conditionals and loops is bounded by that of the corresponding direct execution. With the results for expressions and patterns above, we obtain that the time for interpreting commands by the macro EVAL-CMD is bounded by the time for directly evaluating commands.

Finally, we show that the total time for running ri with the input (p.d) is $\mathrm{O}(|\mathtt{d}|)$. The time of the macro BENNETT is bounded by the maximum of the times to run its two arguments. The initialization macro INIT and the reversible replacement are performed in constant time. Hence, the total time of running ri is bounded by the time required for interpreting the command in the body of p with d. □

The result that a linear-time reversible self-interpreter exists in the language R-WHILE-M is robust in that it is not affected by the time to access variables in the cost model of the language. Any reasonable access timing that we choose is constant time owing to the bounded number of variables. The existence of a linear-time reversible self-interpreter in the language R-WHILE depends on the assumption of the time to access variables. For example, if we assume a unit cost for each variable access, simulating the access to the unbounded number of variables in the source program by the bounded number in the interpreter is impossible in constant time. Therefore, under this assumption, ri for R-WHILE is not linear-time. The cause of the nonlinearity of this interpreter is the variable access time, and this question is independent of the reversibility of the language.

There has been a discussion, independent of the reversibility, of what is a "fair charge" for store access in a language with an unbounded number of variables or constants because some cost models have the counterintuitive consequence (Blum's speedup theorem) that there are functions, which for any program computing such functions, another program that is significantly faster exists.

Limiting the number of variables and constants is known to avoid the question about the fairness of a model. For example, to make his complexity results independent of this discussion, Jones used a language with a single variable and single constant for constructing a linear-time (irreversible) self-interpreter [16] [17].

7 Related Work

`R-WHILE-M` is a simple imperative reversible language that only has tree-structured data and a limited number of variables and atoms, yet it is expressive enough to be r-Turing complete. These properties were all indispensable for constructing a linear-time reversible self-interpreter in this study. To the best of the authors' knowledge, there is no report on the existence of linear-time self-interpretation in r-Turing complete reversible imperative languages. Linear-time self-interpretation was neither reported for the high-level imperative reversible languages R [9] and Gries' invertible language [12] nor for the low-level assembler languages PISA [9] and BoB [27]. None of these languages have tree-structured data. Reversible languages with tree-structured data include the reversible linear Ψ-Lisp [5], the reversible functional languages INV [23] and RFUN [33], and a family of combinator languages [14]. Self-interpretation with a superlinear overhead was reported for other reversible languages and reversible computational models such as a reversible metacircular evaluator [15] for the reversible combinator language Π^o [14], an RFUN self-interpreter [26], and an RTM [2]. None of these languages explicitly limit the number of variables and constants.

It was shown that for any finite set of Janus programs, there is a nontrivial linear-time reversible interpreter in Janus that can simulate the programs without garbage data [35]. Because the storage available to an interpreter is fixed by the static storage allocation of Janus, no interpreter can simulate all Janus programs because they can have any size. Janus was extended with dynamically allocated stacks to be r-Turing complete [31], but whether this data structure allows linear-time self-interpretation is not known.

8 Conclusion and Future Work

We extended the definition of standard interpreters to reversible interpreters and showed for the first time that a linear-time reversible self-interpreter exists in an r-Turing complete reversible imperative language. The language `R-WHILE` has reversible structured control-flow operators and symbolic tree-structured data (S-expressions) and is well-suited for symbolic program manipulation. This new reversible language is as computationally powerful as any reversible language can get.

When building the reversible interpreter in a reversible imperative language, we identified and confirmed the importance of several unconventional programming techniques such as a family of Bennett-type reversible simulations, reversible control loops, the usage of biorthogonal rewrite rules, and the program inversion of reversibilized programs. We have seen that these techniques are effective in practice to minimize the reversible programming effort. They may also inspire new approaches in conventional programming language theory. It is future work to properly abstract reusable programming patterns and components and point out the applications of these programming techniques in reversible programming language design.

As we saw in Section 2, the extension of the familiar (but irreversible) interpreter definition to a new reversible definition is a necessity and not an option in reversible computing. A direction for future work will be to investigate its consequences and which properties hold in a reversible setting, which we take for granted in a irreversible setting. For example, the constant hierarchy theorem [16] [17] denies the existence of programs that are arbitrarily faster than a given program by constant time, and Levin's optimal universal search programs [18] [13] are optimal to within a constant factor for any (semi-)decidable relations. Although the existence of linear-time self-interpreters plays important roles in the proofs of these theorems, whether reversible self-interpreters play important roles in the counterparts of these theorems is an open question.

There is room to reduce the number of variables required to construct a linear-time reversible self-interpreter in a reversible language with a single

atom. One could optimize the interpreter to be more economical with the use of variables. Of course, one could also add more complex constructs to the language, which may even reduce the number of variables to one, as in the irreversible setting [16][17].

Acknowledgments

A preliminary version of this paper was presented at the 17th JSSST Workshop on Programming and Programming Languages (PPL2015). The authors would like to thank the anonymous reviewers for their detailed and constructive comments. The first author has the great pleasure to thank Masami Hagiya, Akihiko Takano, and Zhenjiang Hu for their invaluable support in Japan.

This work was partly supported by JSPS No. RC 21413101, JSPS KAKENHI Grant No. 25730049, and Danish DFF Grant No. FI 12-126689.

References

[1] Abramsky, S.: A Structural Approach to Reversible Computation, *Theoretical Computer Science*, Vol. 347, No. 3(2005), pp. 441–464.

[2] Axelsen, H. B. and Glück, R.: What Do Reversible Programs Compute?, *Foundations of Software Science and Computation Structures. Proceedings*, Hofmann, M.(ed.), Lecture Notes in Computer Science, Vol. 6604, Springer-Verlag, 2011, pp. 42–56.

[3] Axelsen, H. B. and Glück, R.: Reversible Representation and Manipulation of Constructor Terms in the Heap, *Reversible Computation. Proceedings*, Dueck, G. W. and Miller, D. M.(eds.), Lecture Notes in Computer Science, Vol. 7948, Springer-Verlag, 2013, pp. 96–109.

[4] Axelsen, H. B., Glück, R. and Yokoyama, T.: Reversible Machine Code and Its Abstract Processor Architecture, *Computer Science – Theory and Applications. Proceedings*, Diekert, V., Volkov, M. V. and Voronkov, A.(eds.), Lecture Notes in Computer Science, Vol. 4649, Springer-Verlag, 2007, pp. 56–69.

[5] Baker, H. G.: NREVERSAL of Fortune — The Thermodynamics of Garbage Collection, *International Workshop on Memory Management. Proceedings*, Bekkers, Y. and Cohen, J.(eds.), Lecture Notes in Computer Science, Vol. 637, Springer-Verlag, 1992, pp. 507–524.

[6] Bennett, C. H.: Logical Reversibility of Computation, *IBM Journal of Research and Development*, Vol. 17, No. 6(1973), pp. 525–532.

[7] Cooper, D. C.: Böhm and Jacopini's Reduction of Flow Charts, *Communications of the ACM*, Vol. 10, No. 8(1967), p. 463, p. 473.

[8] De Vos, A.: *Reversible Computing: Fundamentals, Quantum Computing, and Applications*, Wiley-VCH, 2010.

[9] Frank, M. P.: *Reversibility for Efficient Computing*, PhD Thesis, Massachusetts Institute of Technology, 1999.

[10] Glück, R. and Kawabe, M.: A Program Inverter for a Functional Language with Equality and Constructors, *Programming Languages and Systems. Proceedings*, Ohori, A.(ed.), Lecture Notes in Computer Science, Vol. 2895, Springer-Verlag, 2003, pp. 246–264.

[11] Glück, R. and Kawabe, M.: Derivation of Deterministic Inverse Programs Based on LR Parsing, *Functional and Logic Programming. Proceedings*, Kameyama, Y. and Stuckey, P. J.(eds.), Lecture Notes in Computer Science, Vol. 2998, Springer-Verlag, 2004, pp. 291–306.

[12] Gries, D.: *The Science of Programming*, Texts and Monographs in Computer Science, Springer-Verlag, 1981, chapter 21 Inverting Programs, pp. 265–274.

[13] Hutter, M.: The Fastest and Shortest Algorithm for All Well-Defined Problems, *International Journal of Foundations of Computer Science*, Vol. 13, No. 3(2002), pp. 431–443.

[14] James, R. P. and Sabry, A.: Information Effects, *Principles of Programming Languages. Proceedings*, ACM Press, 2012, pp. 73–84.

[15] James, R. P. and Sabry, A.: Isomorphic Interpreters from Logically Reversible Abstract Machines, *Reversible Computation. Proceedings*, Glück, R. and Yokoyama, T.(eds.), Lecture Notes in Computer Science, Vol. 7581, Springer-Verlag, 2013, pp. 57–71.

[16] Jones, N. D.: Constant Time Factors Do Matter, *ACM Symposium on Theory of Computing*, ACM Press, 1993, pp. 602–611.

[17] Jones, N. D.: *Computability and Complexity: From a Programming Perspective*, MIT Press, 1997. Revised version, available from http://www.diku.dk/~neil/Comp2book.html.

[18] Levin, L. A.: Universal Sequential Search Problems, *Problems of Information Transmission*, Vol. 9, No. 3(1975), pp. 265–266. Translated from *Probl. Peredachi Inf.*, Vol.9, No.3(1973), pp. 115–116.

[19] Li, M. and Vitányi, P. M. B.: *An Introduction to Kolmogorov Complexity and Its Applications*, Springer-Verlag, 3rd edition, 2008.

[20] Matos, A. B.: Linear Programs in a Simple Reversible Language, *Theoretical Computer Science*, Vol. 290, No. 3(2003), pp. 2063–2074.

[21] Mogensen, T. Æ.: Linear-Time Self-Interpretation of the Pure Lambda Calculus, *Higher-Order and Symbolic Computation*, Vol. 13, No. 3(2000), pp. 217–237.

[22] Morita, K.: Reversible Computing and Cellular

Automata — A Survey, *Theoretical Computer Science*, Vol. 395, No. 1(2008), pp. 101–131.
[23] Mu, S.-C., Hu, Z. and Takeichi, M.: An Injective Language for Reversible Computation, *Mathematics of Program Construction. Proceedings*, Kozen, D.(ed.), Lecture Notes in Computer Science, Vol. 3125, Springer-Verlag, 2004, pp. 289–313.
[24] Perumalla, K. S.: *Introduction to Reversible Computing*, CRC Press, 2013.
[25] Saeedi, M. and Markov, I. L.: Synthesis and Optimization of Reversible Circuits — A Survey, *ACM Computing Survey*, Vol. 45, No. 2(2013), pp. 21:1–21:34.
[26] Thomsen, M. K. and Axelsen, H. B.: Interpretation and Programming of the Reversible Functional Language RFUN, *IFL Pre-Proceedings*, 2015.
[27] Thomsen, M. K., Axelsen, H. B. and Glück, R.: A Reversible Processor Architecture and Its Reversible Logic Design, *Reversible Computation. Proceedings*, De Vos, A. and Wille, R.(eds.), Lecture Notes in Computer Science, Vol. 7165, Springer-Verlag, 2012, pp. 30–42.
[28] Thomsen, M. K., Glück, R. and Axelsen, H. B.: Reversible Arithmetic Logic Unit for Quantum Arithmetic, *Journal of Physics A: Mathematical and Theoretical*, Vol. 43, No. 38(2010), 382002 (10pp.).
[29] Wille, R. and Drechsler, R.: *Towards a Design Flow for Reversible Logic*, Springer-Verlag, 2010.
[30] Winskel, G.: *The Formal Semantics of Programming Languages: An Introduction*, MIT Press, 1993.
[31] Yokoyama, T., Axelsen, H. B. and Glück, R.: Principles of a Reversible Programming Language, *Computing Frontiers. Proceedings*, ACM Press, 2008, pp. 43–54.
[32] Yokoyama, T., Axelsen, H. B. and Glück, R.: Reversible Flowchart Languages and the Structured Reversible Program Theorem, *International Colloquium on Automata, Languages and Programming. Proceedings*, Aceto, L., et al. (eds.), Lecture Notes in Computer Science, Vol. 5126, Springer-Verlag, 2008, pp. 258–270.
[33] Yokoyama, T., Axelsen, H. B. and Glück, R.: Towards a Reversible Functional Language, *Reversible Computation. Proceedings*, De Vos, A. and Wille, R.(eds.), Lecture Notes in Computer Science, Vol. 7165, Springer-Verlag, 2012, pp. 14–29.
[34] Yokoyama, T., Axelsen, H. B. and Glück, R.: Fundamentals of Reversible Flowchart Languages, *Theoretical Computer Science*, Vol. 611(2016), pp. 87–115.
[35] Yokoyama, T. and Glück, R.: A Reversible Programming Language and its Invertible Self-Interpreter, *Partial Evaluation and Semantics-Based Program Manipulation. Proceedings*, ACM Press, 2007, pp. 144–153.

Robert Glück

Robert Glück received his PhD degree and post-PhD (Habilitation) in Austria, and is an Associate Professor in Computer Science at the University of Copenhagen, Denmark. He received twice the Erwin-Schrödinger-Fellowship of the Austrian Science Foundation, was an Invited Fellow of JSPS, and funded by the PRESTO21 basic research program of JST. He co-chaired meetings in North America, Europe and Asia, and is a member of several international computer societies. His main research interests are the theory and practice of programming languages and software systems. His last works relate to reversible computing, partial evaluation, and metacomputation.

Tetsuo Yokoyama

Tetsuo Yokoyama is an Associate Professor at the Department of Software Engineering, the Faculty of Science and Engineering, Nanzan University. He received his Ph.D. degree in Information Science and Technology from the University of Tokyo in 2006. He visited the University of Copenhagen, Department of Computer Science, supported by JSPS Research Fellowships for Young Scientists, from 2006 to 2007. He was a Researcher at the Center for Embedded Computing Systems, the Graduate School of Information Science, Nagoya University from 2007 to 2009, and was an Assistant Professor from 2009 to 2011 at the Department of Software Engineering, the Faculty of Information Sciences and Engineering ; He is a member of ACM, IPSJ, IEICE and JSSST.