A Minimalist’s Reversible While Language

Robert GLück¹, Nonmember and Tetsuo Yokoyama¹(a), Member

SUMMARY The paper presents a small reversible language R-CORE, a structured imperative programming language with symbolic tree-structured data (S-expressions). The language is reduced to the core of a reversible language, with a single command for reversibly updating the store, a single reversible control-flow operator, a limited number of variables, and data with a single atom and a single constructor. Despite its extreme simplicity, the language is reversibly universal, which means that it is as powerful as any reversible language can be, while it is linear-time self-interpretable, and it allows reversible programming with dynamic data structures. The four-line program inverter for R-CORE is among the shortest existing program inverters, which demonstrates the conciseness of the language. The translator to R-CORE, which is used to show the formal properties of the language, is clean and modular, and it may serve as a model for related reversible translation problems. The goal is to provide a language that is sufficiently concise for theoretical investigations. Owing to its simplicity, the language may also be used for educational purposes.

key words: reversible programming language, reversible self-interpreter, while language, translation, program inversion

1. Introduction

Physical devices on a microscopic scale are inherently endowed with reversibility, which is deterministic both forward and backward in time. Hoisting the benefit of the physical reversibility to the logical level has been studied in a wide range of computing machineries, ranging from the design and synthesis of reversible circuits to computer architectures and programming languages (e.g., [1]–[3]).

Concise programming languages with formal semantics have been used for studies in computational complexity and computability as a shift away from the classic Turing machine approaches towards abstractions familiar from programming languages [4]. This approach can render computability and complexity results more accessible, and make theoretical results more applicable to practical problems.

This paper presents the small reversible language R-CORE, a structured imperative programming language with symbolic tree-structured data (S-expressions). The goal is to provide a concise and versatile language for use in a variety of theoretical studies, such as reversible language theory, reversible computational complexity, and computability, where the clearness and ease of meta-reasoning are advantageous. Owing to its simplicity, the language may also be useful for educational purposes. R-CORE is a proper subset of its companion language R-WHILE [5], which provides more programming conveniences such as pattern matching, and it is simplified as the core of a reversible language, as follows:

- a single command for reversibly updating the store,
- a single reversible control-flow operator,
- a limited number of variables, and
- tree-structured data comprising a single atom (n1).

Despite its extreme simplicity, the language is r-Turing complete, which means that it is computationally as powerful as any reversible language can be, while it is also linear-time self-interpretable, and it allows programming with dynamic resource consumption.

The formal properties of the language are shown by a clean translator from R-WHILE to R-CORE based on our recent results [5]. First, it can translate a simulator for reversible Turing machines (RTMs) into R-CORE. Hence, R-CORE is reversibly universal. Second, a linear-time reversible self-interpreter for a subset of R-WHILE, which includes R-CORE, can be translated to R-CORE without changing the interpreter’s asymptotic time complexity. Hence, R-CORE is linear-time self-interpretable.

The translator also shows that all R-WHILE programs can be translated directly to the reversible core language. The translator is clean and modular, and it may serve as a small toolbox for related translation problems. Interestingly, the four-line syntax-directed program inverter for R-CORE, which is used by the translator, is among the shortest existing program inverters* and this demonstrates the conciseness of R-CORE.

An important advantage of while languages is that they are well documented in the theoretical literature, which makes the results obtained for their reversible variants readily accessible and comparable. The R-CORE and R-WHILE languages together appear to provide a good mix of expressive power and formal simplicity. Instead of tapes, stacks, and Gödel numbers, tree-structured data is particularly well-suited for dealing with programs as data objects. Both languages are instances of the structured reversible flowchart paradigm [7].

Given that how we think is restricted by how we describe computation, the simplicity of computational mod-

---

*Reversible Turing machines have a four-line inverter, e.g. [6].
els is crucial, which is why simple universal computation models such as Turing machines have been used extensively. They have been simplified in terms of the numbers of tapes, symbols, and states, as well as in a reversible setting [8], [9].

In this paper, after introducing the syntax and semantics of R-CORE in Sect. 2, we define the translator from R-WHILE to R-CORE in Sect. 3. We assume that the reader is familiar with the basic notions of denotational semantics (e.g. [10]) and reversible programming languages (e.g. [7]). The research presented in this paper is part of our efforts to advance reversible programming language theory and practice (e.g. [5], [7], [11]) and inversion (e.g. [12], [13]).


2. A Small Reversible Programming Language

This section presents the small reversible language R-CORE, a structured imperative language with symbolic tree-structured data (S-expressions). It is a proper subset of R-WHILE, an irreversible program language, and then be mapped to R-CORE. More examples of reversible programming can be found in [5], [7]. From a theoretical perspective, R-CORE is sufficient as a core programming language model.

R-CORE has a single command for reversibly updating the store (a reversible assignment) and a single control-flow operator (a reversible while loop). There is only one atom (nil) in the data domain and the number of variables is limited. The dynamic allocation of data allows the implementation of programs with arbitrary space consumption. Despite its minimalistic features, R-CORE is r-Turing complete.

Next, we define the syntax and formal semantics of R-CORE, and discuss its informal semantics with small examples. The goal of R-CORE is similar to that of the approach taken by Jones [4], who introduced an irreversible universal language I with a single variable and a single atom. However, R-CORE needs more variables because a variable cannot be reused by overwriting its old value with a destructive (irreversible) assignment. Tree-structured data with one atom was used for theoretical purposes [4], [14].

2.1 Syntax

A program P written in R-CORE has a unique entry and exit point (read, write), and a command C as its body (Fig. 1). The input and output of a program P is read into and written from a single variable X. The data domain D is the smallest set containing the atom nil and all of the pairs (d1, d2) ∈ D. If d1, d2 ∈ D, the single atom nil is sufficient to encode other atoms (e.g., by an unary encoding). The language has only a fixed number of variables. More variable values can be encoded in the values of these variables (e.g., by pairing the values of two variables). For the sake of readability, mnemonic names to the variables are used.

An expression is either a variable X, an atom nil, or the application of an operator to variables (the head and tail selectors hd and tl, constructor cons, and equality test =?). Expressions occur only on the right side of reversible assignments. A reversible assignment X ^= E is well-formed, if the variable X does not occur in the expression E, and one-operator form, if the expression E contains at most one operator. Every reversible assignment must be well formed and one-operator form.

A reversible while loop from X loop C until Y is the only control-flow operator in R-CORE.

2.2 Semantics

We define the denotational semantics of R-CORE and discuss its informal semantics with small examples. We begin with some notation and then present the semantic functions.

The undefined value is denoted by ⊥, and a set X ∪ {⊥} is denoted by X. We use D⊥ as the value domain. As usual, a store σ for P is a function from a set of variables in P to D⊥. The store σ ⊔ σ′ is the disjoint union of the bindings in the stores σ and σ′, which have no variables in domains in common. The set of all stores is denoted by Stores.

The three semantic functions of R-CORE are defined for expressions, commands, and programs (Fig. 2):¹

\[
\begin{align*}
E &: \text{Expressions} \rightarrow (\text{Stores} \rightarrow D⊥) \\
C &: \text{Commands} \rightarrow (\text{Stores} \leftarrow \text{Stores}) \\
P &: \text{Programs} \rightarrow (D \leftarrow D⊥)
\end{align*}
\]

The expression evaluation E is as expected. For readability, we write false for nil and true for (nil.nil).

\[
\begin{align*}
E[X]σ &= σ(X) \\
E[nil]σ &= nil \\
E[hd X]σ &= e \text{ if } σ(X) = (e.f) \\
E[tl X]σ &= f \text{ if } σ(X) = (e.f) \\
E[cons X Y]σ &= (σ(X).σ(Y)) \\
E[? X Y]σ &= \begin{cases} 
\text{true} & \text{if } σ(X) = σ(Y) \\
\text{false} & \text{otherwise} 
\end{cases} \\
C[X := E]σ &= σ \circ (X \mapsto d) = σ \circ (X \mapsto d \odot E[σ]) \\
C[C; D]σ &= C[D][C][σ] \\
C[loop C]σ &= σ′ \text{ if } E[X]σ = \text{true} \land σ′ = \text{false}(F)(σ) \\
\text{where } F(ψ) = \{(σ, σ′) | σ(Y) = \text{true} \lor ((σ, σ₂) | σ(Y) = \text{false}) \land σ₁(σ) = \text{false} \land σ₂ = ψ(σ₁)\} \\
P[P]D &= D′ \text{ if } C[C][σ₁(D)] = σ₂(D′) \\
\end{align*}
\]

Fig. 2 Denotational semantics of R-CORE.

¹A ← B denotes an injective mapping from A to B.
\( \mathcal{E} \) can evaluate to \( \bot \) because the values of \( \text{hd} \ X \) and \( \text{tl} \ X \) are undefined if the value of \( X \) is \( \text{nil} \).

We now consider the command evaluation \( C \). There are only three commands to consider in the semantics:

- reversible assignment (\( =^* \)),
- command composition (\( ; \)), and
- control-flow operator (\( \text{loop} \)).

A reversible assignment \( X =^* E \) sets the variable \( X \) to the value of expression \( E \) if \( X \) is \( \text{nil} \) and it sets \( X \) to \( \text{nil} \) if the values of \( X \) and \( E \) are equal. In the former case, the value of \( E \) is duplicated, whereas in the latter case, the equality of the values of \( X \) and \( E \) is asserted.\(^1\) The store update by an assignment is formalized by using the update operator \( \odot \):

\[
 d \odot e = \begin{cases} 
 e & \text{if } d = \text{nil} \\
 \text{nil} & \text{if } d = e \neq \text{nil}.
\end{cases}
\] (1)

The store \( \sigma \), in which \( E \) is evaluated, contains no binding for \( X \) on the left-hand side (see Fig. 2). This is ensured by the disjoint union \( \sigma \sqcup \{ X \mapsto d \} \). Hence, if \( X \) occurs in \( E \), the assignment is undefined. If this were not the case, an invalid assignment \( X = E \) would set \( X \) to \( \text{nil} \) for any value of \( X \), and \( C \square \{ X = X \} \) would not be injective, which is a requirement for the reversibility of the assignment.

A reversible assignment is self-inverse: applying \( X = E \) twice in a row restores the original value of \( X \) provided that \( X = E \) is defined. This holds because for any \( d \) and \( e \), if \( d \odot e \) is defined, then \((d \odot e) \odot e = d \). An assignment can require time that depends on the unbounded size of \( d \) and \( e \) due to the equality test in the second case of Eq. (1).

The semantics of a command composition \( C; D \) is as expected: \( D \) is evaluated in the store resulting from \( C \).

The following examples show how reversible assignments can be used to (i) simulate a skip command, (ii) move a value from a variable to a variable set to \( \text{nil} \), and (iii) swap two variable values. It is often convenient to have a designated variable that is set to \( \text{nil} \) (here, \( N \)).

\[
\begin{align*}
\text{SKIP} & \quad \text{MOVE VALUE } X \leftrightarrow Y \\
(* \ N=\text{nil} \ *) & \quad (* \ X=d, \ Y=\text{nil} \ *) \\
N = \text{nil} & \quad Y = X; \\
(* \ N=\text{nil} \ *) & \quad X = Y \\
(* \ X=\text{nil}, \ Y=d \ *) & \quad (* \ X=\text{nil}, \ Y=d \ *)
\end{align*}
\]

SWAP VALUES \( X \leftrightarrow Y \)

\[
\begin{align*}
(* \ X=d, \ Y=e, \ N=\text{nil} \ *) & \quad (* \ MOVE \ X \mapsto N \ *) \\
N = X; \ X = N & \quad (* \ MOVE \ Y \mapsto X \ *) \\
X = Y; \ Y = X & \quad (* \ MOVE \ N \mapsto Y \ *) \\
Y = N; \ N = Y & \quad (* \ MOVE \ N \mapsto Y \ *)
\end{align*}
\]

(* \ X=e, \ Y=d, \ N=\text{nil} \ *)

A reversible while loop from \( X \) loop \( C \) until \( Y \) has an assertion \( X \) and a test \( Y \), which terminates the iteration of the command \( C \) in the body of the loop (Fig. 3). The informal semantics is as follows. At the entry of the loop, the value of \( X \) must be \( \text{true} \). If the value of \( Y \) is \( \text{true} \), then the loop terminates. If \( Y \) is \( \text{false} \), \( C \) is executed. Provided that \( X \) and \( Y \) are \( \text{false} \), the execution of \( C \) is repeated. The loop is a while loop: \( C \) may be executed zero or more times.

In the semantics, the stores before and after loop execution are related by the least fixed point of \( \mathcal{F} \). The function \( \mathcal{F}(\varphi) \) is defined by the union of the two sets, which correspond to the termination and iteration of the loop execution. If the value of \( X \) does not match the required value, the loop is undefined.

The loop is sufficient to simulate reversible conditionals. For example, the following loop executes \( C \) once if \( X \) is \( \text{false} \) and skips \( C \) if \( X \) is \( \text{true} \). This simulates a conditional with an empty then-branch and \( C \) in the else-branch. We assume the auxiliary variable \( Y \) is not modified by \( C \). Other reversible conditionals can be simulated in a similar manner.

\[
\begin{align*}
\text{LOOP SIMULATING A CONDITIONAL} & \quad (* \ X=\text{bool}, \ Y=\text{true} \ *) \\
& \quad \text{from } Y; \ \text{if } X; \ \text{loop } C; \ \text{SWAP}(X,Y) \ \text{= then } \text{SKIP} \ \text{until } X; \ \text{else } C \\
& \quad \text{SWAP}(X,Y) \ \text{fi } X \\
(* \ X=\text{bool}, \ Y=\text{true} \ *)
\end{align*}
\]

The semantics of a program \( P \) is defined by program evaluation \( \mathcal{P} \). To be reversible, all of the variable values must have well-defined values in the initial and final stores before and after evaluating the main command \( C \). All of the variables are required to be bound to \( \text{nil} \) except for the variable \( X \) in \( \text{read} \) and \( \text{write} \). The store \( \sigma_X(D) \) binds \( X_i \) to data \( D \) and all other variables to \( \text{nil} \): \( \sigma_X(D) = \{ X_0 \mapsto \text{nil}, \ldots, X_i \mapsto D, \ldots, X_9 \mapsto \text{nil} \} \).

A program \( P \) denotes the partial function \( \mathcal{P}[P] \). We sometimes abbreviate \( \mathcal{P} \) when it is clear from the context, and write \([X]_L \) instead of \([X]_L \) to make explicit that \( X \) is a program, a command, or an expression in the language \( L \).

2.3 A Tiny Syntax-Directed Program Inverter

We employ a program inverter in the translator described in the following. A syntax-directed program inverter \( I \) for \( \text{R-CORE} \) is defined in Fig. 4, which can immediately obtain the inverse of any \( \text{R-CORE} \) program. Each command is inverted locally by a recursive descent over the program. Reversible assignments are self-inverse; therefore, no change is needed. The inverse of a command sequence is the reversed sequence of its inverted commands. The two vari-

\[
\begin{align*}
I[X =^* E] &= X =^* E \\
I[C; D] &= I[D]; I[C] \\
I[\text{from } X \text{ loop } C \text{ until } Y] &= \text{from } Y \text{ loop } I[C] \text{ until } X \\
I[\text{read } X; C; \text{ write } X] &= \text{read } X; I[C]; \text{ write } X
\end{align*}
\]

\( I[ \ ) \]

Fig. 3 Reversible while loop in \( \text{R-CORE} \).

Fig. 4 A syntax-directed program inverter for \( \text{R-CORE} \).
R-WHILE by the inverter. The inverter is a special case of that for and that the run time and space consumption is not changed.

First, we present the three components of the translator

### 3.1 Three Small Translators

To show that the language is linear-time self-interpretable, we define a translator \( \mathcal{T}_{\text{asn}} \) from R-WHILE to R-CORE. The fact that every R-WHILE program can be translated into an R-CORE program shows that R-CORE is \( r \)-Turing complete.

Several transformations must be performed because R-CORE, unlike R-WHILE, has no reversible replacements, no conditionals, and only one-branch while loops. Expressions in R-CORE contain one operator at most and the number of variables is limited. In addition, the target program must be fully reversible and not retain additional data in the output. These challenges make the translation of R-WHILE to R-CORE different from the translation schemes for irreversible languages. No code optimizations are performed.

The full syntax and semantics of R-WHILE are defined in [5]. So that the presentation is self-contained, we explain the language features where necessary.

### 3.2 Translation from R-WHILE to R-CORE

#### 3.2.1 Translation to one-operator form

The following example shows the inversion of a push operation, which moves a value from a variable \( X \) to the head of a list \( Y \), thereby simulating a stack. The inverse operation pops the head of \( Y \) into \( X \). Again, it is convenient to assume a variable \( N \) set to nil by default. \( I \) inverts push and pop respectively by reversing the command sequence.

### 3.2.2 Translation of reversible assignments from multi-operator to one-operator form using the inverter \( I \)

The semantic function of expressions is not injective, as required for a reversible implementation. For example, the evaluation of the expression \( \text{hd} \ E \) removes the tail of the value of \( E \). To obtain a reversible implementation of a nested expression in R-CORE, the intermediate values must be preserved in temporary variables. Because these values are irrelevant subsequently, and we do not want to accumulate them in the program, all of the temporary variables are nil-cleared by the inverse command sequence after updating \( X \). The result is a local Bennett-type reversible simulation in the target program, i.e., the compute–copy–uncompute method. This technique is often used in the simulation of reversible programs (e.g., by the interpreter [5]).

The following example illustrates the translation of an assignment with a nested expression (the intermediate values are stored temporarily in \( Y \) and \( Z \), and then cleared):
by another reversible assignment simulation. The translator $T_{an}$ recursively generates code. No transformation is necessary, if an expression only contains a variable. Otherwise, an assignment sequence is generated to compute the value of a subexpression and reversibly copies the result to a temporary variable $Y$, which is fresh during each recursion. If the expression is a pair of structured data (e.g., $E$), the assignment is expanded into a sequence of assignments to construct the value. (The number of temporary variables is reduced later.)

The translation $T_{an}$ preserves the semantics of assignments. For any reversible assignment $C$ and store $\sigma$, we have $C[[T_{an}]] = C[[R-WHILE\sigma]]$. The length of the generated assignment sequence is linear in the size of the expression $E$ in $X \Leftarrow E$. If the original assignment is well formed, i.e., the variable $X$ on the left side does not occur in $E$, then the generated sequence is also well-formed. A simple optimization of $T_{an}$ is to return an assignment that is already in one-operator form without modification.

Expansion of reversible replacements

A reversible replacement $Q \Leftarrow R$ with linear patterns $Q$ and $R$ on both sides is convenient for programming in $R-WHILE$. In contrast to a reversible assignment, a variable can appear on both sides of a reversible replacement. For example, the effect of the replacement $(Y.X) \Leftarrow (X.Y)$ is to swap the values of $X$ and $Y$. Patterns comprise subsets of the set of nested expressions that do not contain repeated variables, selectors and equality tests:

Patterns $\ni Q$, $R ::= X | \text{nil} | (Q.R)$

The value of a pattern $R$ under the store $\sigma$ is obtained by replacing every variable in $R$ with its value in $\sigma$.

The execution of a reversible replacement $Q \Leftarrow R$ updates the store. Let $\sigma$ (resp., $\sigma'$) be the stores immediately before (resp., after) the replacement $Q \Leftarrow R$. The variables that occur in $Q$ are updated using the value of $R$ such that the value of $R$ evaluated in $\sigma$ is equal to the value of $Q$ evaluated in $\sigma'$. We impose no further restrictions on the values of $Q$ and $R$.

The reversible command sequence generated by the translator $T_{ce}$ (Fig. 6) for $Q \Leftarrow R$ has two parts. First, a temporary variable $Y$ is set to the value of $R$, and all variables in $R$ are nil-cleared, ready for an update by a reversible assignment. Second, the code generated by $T$ performs the inverse of the replacement $Y \Leftarrow Q$. The variables in $Q$ are set such that the value of $Q$ in the new store is equal to the value of $Y$, and the temporary variable $Y$ is nil-cleared.

The sub-translator $T_{ce}[[X \Leftarrow Q]]$ generates code, which assigns the value computed by pattern $Q$ to the variable $X$. Before the execution of the code, $X$ is assumed to be nil. The temporary variable $Y$ introduced by the sub-translator is fresh in each recursion of $T_{ce}$. If the pattern is $\text{nil}$, then the generated code asserts that the value of $X$ is equal to $\text{nil}$; otherwise, execution halts abnormally at the assertion $\text{ASSERT}(Y)$, which is implemented by from $Y \text{loop SKIP until } Y$ (this command is defined only if $Y$ is true). Then $Y$, which is always true after the assertion, is nil-cleared. If the pattern is a variable $Z$, then the value of $Z$ is moved to $X$. If the pattern is a pair of patterns $(Q.R)$, then the values of the patterns $Q$ and $R$ are moved to $Y$ and $X$, respectively, and the values are paired by pushing $Y$ onto $X$. Given actual arguments, the macros $\text{MOVE}(X,Y)$ and $\text{PUSH}(X,Y)$ are expanded to the corresponding codes in Sect. 2.2 and Sect. 2.3, respectively.

The reversible replacement that swaps two variable values is translated into the following assignment sequence (after inlining the trivial assignments produced by the translator), which is another way to write example $\text{SWAP}$ (Sect. 2.2).

$$
\begin{align*}
T &= \text{cons } X \ Y; (* 1.\text{comp } *) \\
X &= \text{hd } T; (* 2.\text{clear } *) \\
Y &= \text{tl } T; (* 2.\text{clear } *) \\
Y &= \text{hd } T; (* 3.\text{update } *) \\
X &= \text{tl } T; (* 3.\text{update } *) \\
T &= \text{cons } Y \ X (* 4.\text{uncomp } *)
\end{align*}
$$

Reducing the number of variables

The last pass of the translator involves reducing the number of variables in a program, which is achieved by packing all of the variable values in an $R-WHILE$ program into a single list $Vl$. This follows the variable reduction [4], but requires fully reversible store management. The number of variables in an $R-WHILE$ program in one-operator form is reduced to at most ten by the translator $R$ (Fig. 8). (The update and lookup of the packed values in $Vl$ require temporary variables in a reversible language.)

The $i$-th element of $Vl$ is the value of the $i$-th variable $X_i$ in the interpreted program. The value list $Vl$ is accessed only by the function macros $\text{UPDATE}$ and $\text{LOOKUP}$ where for any $\sigma(Vl) = (d_1 d_2 \cdots d_n)^\dagger$:

$$
\begin{align*}
\mathbb{C}[\text{LOOKUP } i X \sigma] &= \sigma[X \mapsto \sigma(X) \odot d_i], & (2) \\
\mathbb{C}[\text{UPDATE } i E \sigma] &= \sigma[Vl \mapsto (d_1 d_2 \cdots (d_i \odot E \odot E[\sigma(Vl)] \cdots d_n)]]. & (3)
\end{align*}
$$

The index $i$ is equal to the unary number $i$. The macro $\text{LOOKUP } i X$ reversibly copies the $i$-th value of $Vl$ to the nil-variable $X_i$ in $\sigma$. The macro $\text{UPDATE } i E$ reversibly

\begin{footnote}{The store $\sigma[X \mapsto D]$ is identical to $\sigma$ except that it maps $X$ to $D$, and the store $\sigma'X$ is identical to $\sigma$ except that it maps $X$ to $\bot$.}
\end{footnote}
updates the $i$-th value of $V_l$ by the value of the expression $E$, which is in one-operator form. The update operator $\odot$ is defined in Eq. (1). $\text{LOOKUP}$ and $\text{UPDATE}$ are self-inverse, which is a property that we use extensively as a reversible programming technique to reset values. Other data structures could be used for $V_l$, but the list representation does not affect the asymptotic time complexity of the linear-time reversible interpretation (the number of variables in $\text{R-CORE}$ is fixed).

Figure 7 defines the macros $\text{LOOKUP}$ and $\text{UPDATE}$ using three auxiliary macros. $\text{Macro SHIFT}$ sets the variable $T$ to the $j$-th value of the value list $V_l$ by shifting the values from $V_l$ to $V_r$ in a loop. Variable $K$ in the loop counts the number of iterations. The variables $I$ and $J$ in the test and assertion of the loop are set by test macro $\text{TST}$, where $I$ is set to true if $K$ is equal to zero ($\text{nil}$) and $J$ is set to true if $K$ is equal to $j$. Variable $V_r$ holds the already traversed part of the value list $V_l$. The recursive macro $\text{CST}$ expands to a command sequence that pushes $\text{nil}$ $j$ times on $I$ (see Sect. 2.3). This constructs a constant in the program. The expansion of $j$ is necessary because $\text{R-CORE}$ does not have nested expressions. If $\bar{\Pi}$ is a unary number represented by a list of $n$ $\text{nil}$'s, then $\text{CST}\;\bar{\Pi}\;I$ sets a nil-cleared $I$ to $\bar{\Pi}$.

The translator $\mathcal{R}$ in Fig. 8 preserves the semantics for any proper input (the input must not include conditionals, loops with do branches, and nested expressions). Let relation $\sigma = (d_1, d_2, \ldots, d_n)$ hold iff for any $i$, $\sigma(X_i) = d_i$. Given a proper command $C$, for any stores $\sigma_1$, $\sigma_2$, $\sigma_1'$, and $\sigma_2'$ such that $\sigma_1 \simeq \sigma_1'(V_l)$, $\sigma_2 \simeq \sigma_2'(V_l)$, and $\sigma_1'(X) = \sigma_2'(X) = \text{nil}$, where $X$ ranges over the temporary variables $I$, $J$, $K$, $T$, $N$, and $V_r$,

$$\sigma_1' = \mathcal{C}[\mathcal{C}] \sigma_1 \quad \text{iff} \quad \sigma_2' = \mathcal{C}[\mathcal{R}][\mathcal{C}] \sigma_2.$$  

This is proved by structural induction on command $C$. The first six cases are trivial by the invariants of $\text{LOOKUP}$ and $\text{UPDATE}$ (Eqs. (2) and (3)). Excluding the case where $X \simeq \text{nil}$, the generated code comprises three parts: data lookup, which sets the values of the variables $V$ and $W$, update of the $i$-th position in $V_l$, and clearing the variables $V$ and $W$. The case for a sequence $C; D$ directly follows the induction hypothesis. The last loop case also directly follows the induction hypothesis because the stores before and after the application of $\mathcal{R}[\mathcal{C}]$ satisfy the above invariant. The variables $V$ and $W$ are set to $\text{nil}$ before and after the recursion of $\mathcal{R}$, so they can be reused during each recursion.

### 3.2 The Complete Translator

We have not yet discussed the translation of control-flow operators, and how the three translator components are used in the translator from $\text{R-WHILE}$ to $\text{R-CORE}$ in Fig. 9.

The translator preserves the meaning of an $\text{R-WHILE}$ program $P$: for any data $d$ and $d' = \mathcal{R}[\text{R-WHILE}](d)$ iff $d' = \mathcal{R}[\text{R-CORE}](d)$. The translation is clean in the sense that it generates target programs without adding extra output, i.e., the input and output format of $P$ is not modified by the translator.

The first case is the translation of the main program, which may contain different input and output variables. The variable $V_l$ stores the values of all the variables that appear

![Fig. 8 Reduction of the number of variables in commands. The variables $V$ and $W$ are global; $i$, $j$, and $k$ are unary numbers.](image-url)
Fig. 9  Translator __ from R-WHILE to R-CORE.

in the R-WHILE program. V1 is initialized by macro CST (Fig. 7) to a list of nil’s of length n, where n is the number of variables in the program. The inverse of macro CST clears the value of V1 assuming it is a list of nil’s. The value in the input and output variable V is guaranteed to be cleared before and after the execution of the translated main command R[C].

Commands in R-WHILE include reversible replacements, reversible conditionals, and reversible loops with two branches. All expressions in R-WHILE can be nested.

The individual commands are translated as follows. Reversible replacements Q <= R and reversible assignments Xi := E are translated by T_Q and T_Xi, respectively. The translation of a command sequence C; D is a sequence of translated commands C; D.

Reversible conditionals with only a then branch are transformed into loops with one branch, where the assertions and tests are variables. The variable U is used as a flag for the assertion and T for the test. If the values of test E and assertion F do not match, the evaluation halts abnormally at the assertion ASSERT(T). To assure the behavior of the conditional with one branch, it is sufficient to check the four cases of the values of the Boolean expressions E and F. Before and after the evaluation of the branch clause C, the temporary variables T and U are nil-cleared and can be reused. A similar technique is used in the code generated for reversible conditionals and reversible loops with both branches. Reversible conditionals with both branches are first transformed into two reversible conditionals each with one branch. The results are further translated by the rule for reversible conditionals with one branch.

The translation of a reversible loop in R-WHILE is more involved because it uses a technique for transforming unstructured to structured reversible flowcharts [7]. The generated code simulates the computation of the general loop:

We usually omit an empty branch (a skip command performing the identity function) in a loop or conditional.

The individual commands are translated as follows. Reversible replacements Q <= R and reversible assignments Xi := E are translated by T_Q and T_Xi, respectively. The translation of a command sequence C; D is a sequence of translated commands C; D.

Reversible conditionals with only a then branch are transformed into loops with one branch, where the assertions and tests are variables. The variable U is used as a flag for the assertion and T for the test. If the values of test E and assertion F do not match, the evaluation halts abnormally at the assertion ASSERT(T). To assure the behavior of the conditional with one branch, it is sufficient to check the four cases of the values of the Boolean expressions E and F. Before and after the evaluation of the branch clause C, the temporary variables T and U are nil-cleared and can be reused. A similar technique is used in the code generated for reversible conditionals and reversible loops with both branches. Reversible conditionals with both branches are first transformed into two reversible conditionals each with one branch. The results are further translated by the rule for reversible conditionals with one branch.

The translation of a reversible loop in R-WHILE is more involved because it uses a technique for transforming unstructured to structured reversible flowcharts [7]. The generated code simulates the computation of the general loop:

We usually omit an empty branch (a skip command performing the identity function) in a loop or conditional.
The assertion that urtm applies to the RTM interpreter is determined by conditional if Xi then U = Xi else U = Xj fi Xi. The two occurrences in the code are executed when U = nil resp. U = Xi || Xj to set resp. clear U, and do not halt abnormally.

Each variable access that is simulated when evaluating P has linear-time overhead relative to that when evaluating P. The time overhead can be reduced to logarithmic, if we use a balanced tree to store the simulated variables. Because this choice is not relevant for the argument in Sect. 4, we use the simpler list implementation.

4. **R-CORE** Properties

First, we show the computational power of R-CORE. A programming language in which a universal reversible Turing machine (URTM) can be implemented is r-Turing complete, i.e., computationally as powerful as any reversible language can be [6]. An RTM interpreter urtm written in R-WHILE, i.e., an implementation of a URTM, can be translated into R-CORE by the translator _ in Sect. 3, which implies the following theorem.

**Theorem 1:** The language R-CORE is r-Turing complete.

**Proof:** The meaning of P is the same as that of P, which also applies to the RTM interpreter urtm written in R-WHILE. The assertion that urtm is an RTM interpreter in R-CORE means that R-CORE is r-Turing complete.

A reversible self-interpreter ri can be constructed for R-WHILE as shown in publication [5]. We obtain an interpreter for R-WHILE in R-CORE by translating ri into R-CORE. For any R-WHILE program p and data d, we have

\[ [ri]^{R-CORE}(p, d) = [ri]^{R-WHILE}(p, d). \tag{5} \]

The space used during the evaluation of p is proportional to the space used during its interpretation by ri.

**Theorem 2:** The program ri in Eq. (5) is a linear-time reversible self-interpreter for R-CORE.

**Proof:** Eq. (5) holds for any R-CORE program p because R-CORE is a proper subset of R-WHILE.

Next, we argue that ri is only a constant factor slower than ri. For every R-WHILE program, the time overhead introduced by the translator _ (Fig. 9) can be bounded by a constant. We can show this by structural induction. For a given program, the overhead introduced by R to access the variables is bounded by a constant (the size of V1 is fixed). If the code generated by the translator for a command is a straightline command sequence, then the time overhead is a constant factor. A loop in the code generated from a conditional is iterated at most once. Thus, the time overhead can be bounded by a constant. In a loop simulating a loop, a test, a command of either the do or loop branches, or an assertion is interpreted in each iteration. The induction hypothesis implies that the runtime for each iteration can be bound by a constant factor. Thus, some constant c1 exists for ri such that for any R-CORE program p and data d,\(^1\)

\[ time_{ri}^{R-CORE}(p, d) \leq c_1 \cdot time_{ri}^{R-WHILE}(p, d). \tag{6} \]

Note that the quantifier for c1 appears before that for p and d, so c1 does not depend on a particular p or d.

The original ri is linear time for any (infinite) subset of R-WHILE that has a maximum number m of variables [5, Thm. 2]. Since R-CORE only has ten variables, there is some constant c2, such that for any R-CORE program p and data d: time_{ri}^{R-CORE}(p, d) \leq c_2 \cdot time_{p}^{R-CORE}(d). This and Eq. (6) lead to time_{ri}^{R-CORE}(p, d) \leq c_1 \cdot c_2 \cdot time_{p}^{R-CORE}(d). Therefore, ri is a linear-time reversible self-interpreter of R-CORE.

One could obtain a shorter interpreter for R-CORE by trimming down the self-interpreter for R-WHILE [5], translating it to R-CORE, and performing a similar argument as in the proof above. For the purposes of this section, the existing interpreter suffices.

5. Conclusion

One of the main advantages of the reversible language R-CORE is its simplicity, yet it is computationally as powerful as any reversible language can be. R-CORE is minimal in terms of the numbers of store update commands and control flow operators. The reversible while loop of R-CORE can simulate reversible conditionals and various types of loops. The single atom (nil) and the binary constructor (cons) are sufficient to work with complex dynamic data structures. The number of variables is limited in the language. The cost model of R-CORE is fair in the sense that it is reasonable to assume that it requires constant time to access variables and atoms. The syntactic and semantic simplicity of the language are useful for meta-reasoning.

From an architectural viewpoint, R-CORE may also serve as the language for a list-processing reversible abstract machine with a limited number of registers, which operates at a higher abstraction level than other reversible machine models that usually only employ integer data. The support for high-level languages and compilers, such as the translator from R-WHILE to R-CORE described in this paper, can compensate for the terseness of the core language.

From the viewpoint of semantics, the denotational semantics makes the language accessible to a well established theoretical tool set. The simulation of R-CORE in other lan-

\(^1\) time_{p}^{R-CORE}(d) is the number of steps to compute p given input d if the computation terminates; otherwise, \(\perp\).
guages is not difficult. Playing with a small r-Turing complete language with lists as data may be instructive for newcomers to the area of reversible computing. A number of reversible programming patterns were shown in the examples and the translators. It is also easy to enrich the language in various ways for particular studies.

The list-processing state-full language presented above is an instance of the larger framework of reversible flowchart languages and can hopefully serve as a solid basis for further studies of reversible language theory and practice.

Acknowledgments

The first author has the great pleasure to thank Akihiko Takano, Masami Hagiya, Zhenjiang Hu, and Kanae Tsushima for their invaluable support in Japan. The authors would like to thank the anonymous reviewers for their constructive comments and an improvement of the replacement translation. A preliminary version of R-CORE and the translator was presented at the 17th Workshop on Programming and Programming Languages (PPL 2015). This work was supported partly by Nanzan University Pache Research Subsidy I-A-2 for the 2016 academic year. The support by COST Action IC1405 is acknowledged.

References


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

Tetsuo Yokoyama is an Associate Professor at the Department of Software Engineering, Nanzan University. He received his Ph.D. in Information Science and Technology from the University of Tokyo in 2006. He was a researcher at the Center for Embedded Computing Systems, Nagoya University from 2007 to 2009; an Assistant Professor from 2009 to 2011 at the Department of Software Engineering, Nanzan University.