Reversible Programs Have Reversible Semantics

Robert Glück[1], Robin Kaarsgaard[1], and Tetsuo Yokoyama[2]

[1] DIKU, Department of Computer Science, University of Copenhagen, Copenhagen
[2] Dept. of SW, Nanzan University, Nagoya, Japan `tyokoyama@acm.org`

Abstract. During the past decade reversible programming languages have been formalized using various established semantics frameworks. Common to these semantics is that they are ineffective at specifying the distinct properties of reversible languages at the metalevel, even including the central question whether the defined language is reversible. In this paper, we build upon a metalanguage foundation for reversible languages based on the category of sets and partial injective functions. We exemplify our approach by a step-by-step development of the full semantics of an r-Turing complete reversible while-language with recursive procedures. This leads to a formalization of the semantics in which the reversibility of the language and its inverse semantics are immediate, as well as the inversion of programs written in the language. We discuss further applications and directions for reversible semantics.

Keywords: Formal semantics · Reversible languages · Recursion · Iteration · Partial injective functions.

1 Introduction

During the past decade reversible programming languages ranging from imperative to functional and object-oriented languages have been formalized using established semantics frameworks, such as state transition functions, structural operational semantics, and recently denotational semantics (*e.g.* [19, 18, 6, 7]). These frameworks, which have been used to give meaning to advanced language features and computation models, such as nondeterminism and parallelism, have turned out to be ineffective at specifying the distinct semantics properties of reversible languages. Even the central question whether a language is reversible cannot be answered immediately, likewise questions about the uniqueness of the inverse semantics and the inversion of programs in such languages.

In this paper, we build on a metalanguage foundation for reversible languages based on the category **PInj** of sets and partial injective functions. The philosophy behind this approach is straightforward: Interpretations of syntax are composed in ways that preserve their injectiveness. More specifically, interpretations of syntax are composed by sequential composition, cartesian product, disjoint union, and function inversion. For this we make use of the categorical foundation developed elsewhere (*e.g.* [5, 11]). Our approach exploits the fact that reversible programs have reversible semantics. We regard a program as (compositionally) reversible iff each of its meaningful subprograms are partially invertible.

This allows us to give a clean reversible semantics to a reversible programming language.

We demonstrate the idea by a step-by-step development of a full formal semantics of the r-Turing complete reversible while-language `R-WHILE` including recursive procedures. This leads to a formal semantics in which the reversibility of the language and its inverse semantics are immediate, as well as the inversion of programs. The reversibility of the language follows immediately from the semantics formalization. That the language is clean without hidden tracing can be seen from the signature of the semantics functions. This approach is independent of the specific language details and can be extended to other methods of composing semantics functions, provided their injectiveness is preserved.

`R-WHILE` is a reversible while-language with structured control-flow operators, recursive procedures, and dynamic data structures [6, 7].[3] The language is reversibly universal, which means that it is computationally as powerful as any reversible programming language can be. It has features representative of reversible imperative and functional languages, including reversible assignments, pattern matching, and access to the inverse semantics of procedures in programs.

The metalanguage used here has an interesting property familiar from reversible programming: It is not possible to define an irreversible (non-injective) language semantics. Conventional metalanguages require a discipline to ensure reversibility, *e.g.* a standard denotational semantics, or it is quite unclear how to restrict them, *e.g.* to reversible inference systems. A future direction of research can be the investigation of the extension of the metalanguage to capture others forms of composition and language features, which may include object-oriented features, combinators, and machine languages.

Overview: Section 2 introduces the elements of the formal semantics, while Section 3 describes the reversible language `R-WHILE`. In Section 4 the formal semantics of the language is developed step by step. Section 5 and Section 6 offer related work, concluding remarks and directions for future work. We assume that the reader is familiar with the basic notions of reversible languages (*e.g.*, [18]).

2 Elements of the Formal Semantics

This section concerns some details on sets and partial injective functions as they will be used in the sections which follow (compare, *e.g.*, [2, 4, 15]). While the constructions mentioned in this section all come from the study of the category **PInj** of sets and partial injective functions, *no categorical background is assumed* (though a basic understanding of sets, partial functions, and domain theory is).

2.1 Composition and Inversion

Partial functions are ordinary functions, save for the fact that they may be undefined on parts of their domain. To indicate that a partial function $X \xrightarrow{f} Y$ is undefined on some $x_0 \in X$ (*e.g.*, in the definition of a piecewise function), we

[3] An online interpreter for `R-WHILE` with procedures and the example program in this paper are available at `http://tetsuo.jp/ref/RPLA2019`.

use symbol $\uparrow$. A partial function is injective iff whenever $f(x)$ and $f(y)$ are both defined and $f(x) = f(y)$, it is also the case that $x = y$. Injectivity is preserved by composition (*i.e.*, if $X \xrightarrow{f} Y$ and $Y \xrightarrow{g} Z$ are both partial injective functions so is $X \xrightarrow{g \circ f} Z$), and each identity function $X \xrightarrow{\text{id}} X$ is trivially injective.

Partial injective functions can be inverted in a unique way: for every partial injective function $X \xrightarrow{f} Y$ there exists a unique partial injective function $Y \xrightarrow{f^\dagger} X$ which undoes whatever f does (how rude!), in the sense that $f \circ f^\dagger \circ f = f$, and, vice versa, $f^\dagger \circ f \circ f^\dagger = f^\dagger$.

Aside from sequential composition, partial injective functions can also be composed in parallel in two ways. The first is using the cartesian product of sets X and Y, which we denote $X \otimes Y$. If $X \xrightarrow{f} X'$ and $Y \xrightarrow{g} Y'$ are partial injective functions, we can form a new one on the cartesian product, $X \otimes Y \xrightarrow{f \otimes g} X' \otimes Y'$, by $(f \otimes g)(x, y) = (f(x), g(y))$. Note, however, that we do *not* have projections (such as $X \otimes Y \xrightarrow{\pi_1} Y$ given by $\pi_1(x, y) = x$) available, as these are never injective. We will denote the unit, up to bijective correspondence, of the cartesian product (any distinguished singleton set will do) by 1.

Another parallel composition is given on the disjoint union of sets X and Y, which we denote $X \oplus Y$. We think of elements of $X \oplus Y$ as being tagged with either left or right depending on their set of origin; for example, if $x \in X$ then $\mathsf{inl}\ x \in X \oplus Y$, and if $y \in Y$ then $\mathsf{inr}\ y \in X \oplus Y$. Up to bijective correspondence, the unit of disjoint union is the empty set $\emptyset$, which we will also denote as 0. The tagged union of partial injective functions $X \xrightarrow{f} X'$ and $Y \xrightarrow{g} Y'$ is then a partial injective function of tagged unions, $X \oplus X' \xrightarrow{f \oplus g} Y \oplus Y'$, performing a case analysis on the inputs and tagging outputs with their origin:

$$(f \oplus g)(x) = \begin{cases} \mathsf{inl}\ f(x') & \text{if } x = \mathsf{inl}\ x' \\ \mathsf{inr}\ g(x') & \text{if } x = \mathsf{inr}\ x' \end{cases}$$

While the cartesian product lost its projections in the setting of partial injective functions, the disjoint union retains its usual injections: There are injections $X \xrightarrow{\kappa_1} X \oplus Y$ and $Y \xrightarrow{\kappa_2} X \oplus Y$ given by $\kappa_1(x) = \mathsf{inl}\ x$ and $\kappa_2(y) = \mathsf{inr}\ y$. Note in particular that since we consider *partial* injective functions, these have partial inverses $\kappa_i^\dagger$ (sometimes called *quasiprojections*) which remove the tag, but are only defined for elements from the i'th part of the union. For example, $X \oplus Y \xrightarrow{\kappa_1^\dagger} X$ is given by $\kappa_1^\dagger(\mathsf{inl}\ x) = x$ and $\kappa_1^\dagger(\mathsf{inr}\ y) = \uparrow$.

Note finally the interactions between the cartesian product and disjoint union: $X \otimes 0$ is empty for all sets X, and analogous to the behaviour of addition and multiplication in a (semi)ring, there is a bijective correspondence (given by the so called *distributor*) between $X \otimes (Y \oplus Z)$ and $(X \otimes Y) \oplus (X \otimes Z)$ for all sets X, Y, Z.

2.2 Fixed Points and Iteration

Both sets and partial injective functions are well-behaved when it comes to recursive definitions. For sets, any recursive definition of a set involving only disjoint unions, cartesian products, and already defined sets (including 0 and 1) has a unique least and greatest solution: As is usual in domain theory, we use $\mu X \ldots$ for the least solution (the least fixed point) and $\nu X \ldots$ for the greatest solution (the greatest fixed point). For example, the set of lists with entries taken from a set A is given by the least fixed point $\mu X.1 \oplus (A \otimes X)$.

A useful property of partial functions, as opposed to total ones, is that the set of all partial functions with specified domain and target forms a directed complete partial order. This has useful consequences for the recursive description of partial injective functions: In particular, any continuous function $\mathbf{PInj}(X,Y) \to \mathbf{PInj}(X,Y)$ (where $\mathbf{PInj}(X,Y)$ denotes the set of partial injective functions between sets X and Y) has a least fixed point, which, by its definition, must be a partial injective function $X \to Y$ (*i.e.*, an element of $\mathbf{PInj}(X,Y)$). For the continuity requirement, we note that all previously presented operations on partial injective functions are continuous (*i.e.*, sequential composition, partial inversion, parallel composition using cartesian products and disjoint unions), so any function involving only these operations is guaranteed to be continuous.

Finally, partial injective functions can also be tail recursively described using the *trace* operator. Intuitively, the trace of a partial injective function $X \oplus U \xrightarrow{f} Y \oplus U$ is a function $X \xrightarrow{\mathrm{Tr}(f)} Y$ given as follows: If $f(\mathsf{inl}\ x) = \mathsf{inl}\ y$ for some y, this y is returned directly. Otherwise, if f is defined at $\mathsf{inl}\ x$, it must be the case that $f(\mathsf{inl}\ x) = \mathsf{inr}\ u$ for some u. If this is the case, this $\mathsf{inr}\ u$ is fed back into f, and the feedback loop continues until it either terminates to some $\mathsf{inl}\ y$, which is then returned, or does not, in which case the trace is undefined at x. This trace operator may be described as a function $\mathbf{PInj}(X \oplus U, Y \oplus U) \xrightarrow{\mathrm{Tr}} \mathbf{PInj}(X,Y)$. It is most easily described using a recursively described *pretrace* $\mathbf{PInj}(X \oplus U, Y \oplus U) \xrightarrow{\mathsf{pretrace}} \mathbf{PInj}(X \oplus U, Y)$ given as follows:

$$\mathsf{pretrace}(f)(x) = \begin{cases} y & \text{if } f(x) = \mathsf{inl}\ y \\ \mathsf{pretrace}(f)(\mathsf{inr}\ y) & \text{if } f(x) = \mathsf{inr}\ y \end{cases}$$

With this, the trace is described simply as $\mathrm{Tr}(f)(x) = \mathsf{pretrace}(f)(\mathsf{inl}\ x)$. More formally, it can also be described as a fixed point using the *trace formula*, see [8]. While less general than the fixed point (which can be used to describe arbitrary recursion), this tail recursion operator is very well behaved with respect to inversion, as it satisfies $\mathrm{Tr}(f^\dagger) = \mathrm{Tr}(f)^\dagger$ for all partial injective functions $X \oplus U \xrightarrow{f} Y \oplus U$.

Now, we specify a metalanguage $\mathcal{M}$ for describing objects of $\mathbf{PInj}$:

$$f ::= a \mid \kappa_i \mid \mathrm{id} \mid \mu\phi.f \mid f \oplus f \mid f \otimes f \mid f \circ f \mid \mathrm{Tr}(f) \mid f^\dagger \mid \phi$$

All constructs appeared in this section except the atomic function a, which is a user-defined partial injective function. The formal argument of the least fixed point ϕ expects program contexts. For any expression in $\mathcal{M}$, the least fixed point exists. $\mathcal{M}$ is closed under inversion, and the inverse semantics of each expression is obtained for free.

$$
\begin{aligned}
e &::= x \mid \overline{s} \mid (e.e) \mid \mathsf{hd}(e) \mid \mathsf{tl}(e) \mid {=?}\ e\ e \\
q &::= x \mid \overline{s} \mid (q.q) \mid \mathsf{call}\ f(q) \mid \mathsf{uncall}\ f(q) \\
c &::= x \mathrel{\widehat{=}} e \mid q \Leftarrow q \mid c; c \mid \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c\ \mathsf{fi}\ e \mid \mathsf{from}\ e\ \mathsf{do}\ c\ \mathsf{loop}\ c\ \mathsf{until}\ e \\
p &::= \mathsf{proc}\ f(q)\ c; \mathsf{return}\ q; \\
m &::= p\ \cdots\ p
\end{aligned}
$$

Fig. 1. The syntax of `R-WHILE`.

3 R-WHILE with Reversible Recursion and Iteration

This section describes the syntax and informal semantics of `R-WHILE` and illustrates it with a program that translates infix expressions into Polish notation. The data domain of the language is tree-structured data (lists known from Lisp). Readers familiar with reversible programming can skip to the example.

The syntax of the recursive language [7] is shown in Fig. 1. An *expression* e is either a variable x, a symbol $\overline{s}$, or the application of an operator, *i.e.* constructor cons $(\cdot . \cdot)$, selectors head hd and tail tl, equality test $=?$. A *variable* x is denoted by small letters, such as t, and a *symbol* $\overline{s}$ is overlined, such as $\overline{nil}$.

A *pattern* q is a variable x, a symbol $\overline{s}$, a pair of patterns $(q.q)$, or a forward or backward invocation of a procedure by $\mathsf{call}\ f(q)$ or $\mathsf{uncall}\ f(q)$. All patterns are linear; procedure invocations can be nested. The meaning of a procedure uncall is the inverse semantics of a procedure call. Procedures can only be invoked in patterns, but patterns can appear in several places including the actual and formal arguments of procedures, return commands, and reversible replacements.

A *command* c is either a *reversible assignment* $x \mathrel{\widehat{=}} e$, a *reversible replacement* $q \Leftarrow q$, a *reversible conditional* $\mathsf{if}...\mathsf{fi}$, or a *reversible loop* $\mathsf{from}...\mathsf{until}$. The latter two are the same control structures as in reversible flowchart languages (*e.g.*, [18]). As usual, variable x in a reversible assignment $x \mathrel{\widehat{=}} e$ must not occur in e (*e.g.*, $x \mathrel{\widehat{=}} x$ is not well formed). It sets x to the value of e if x is $\overline{nil}$, to the value $\overline{nil}$ if the values of x and e are equal, or it is undefined. No value is copied by $q_1 \Leftarrow q_2$. Before the value constructed by q_2 is matched with q_1, all variables in q_2 are nil-cleared, so the same variable may occur on both sides of a replacement.

A *program* m is a sequence of procedures, where the topmost procedure is the main procedure. A *procedure* p has a name f, a formal argument q, a command c as its body, and a return pattern q. The language has no global variables.

Example 1. Figure 2 shows the reversible recursive procedure *pre* that translates infix expressions into prefix expressions (Polish notation) by a preorder traversal of a full binary tree representing the infix expression. The procedure *pre* is called and uncalled in procedures *infix2pre* and *pre2infix* to translate to and from Polish notation, respectively. An infix expression is represented by a full binary tree

$$tree ::= \overline{a} \mid (tree\,.\,(\overline{d}\,.\,tree))$$

where $\overline{a}$ and $\overline{d}$ are the operand (leaf) and the operator (inner label), respectively.

The body of *pre* consists of a reversible conditional (lines 10–17) with an entry predicate $({=?}\ t\ \overline{a})$ and an exit assertion $({=?}\ \mathsf{hd}(y)\ \overline{a})$. If tree t is a leaf $\overline{a}$,

```
 1: proc $infix2pre(t)$                 (* infix exp to Polish notation *)
 2:   $y \Leftarrow$ call $pre((t.\overline{nil}))$;     (* call preorder traversal      *)
 3:   return $y$;
 4:
 5: proc $pre2infix(y)$                 (* Polish notation to infix exp *)
 6:   $(t.\overline{nil}) \Leftarrow$ uncall $pre(y)$;   (* uncall preorder traversal    *)
 7:   return $t$;
 8:
 9: proc $pre((t.y))$                   (* recursive preorder traversal *)
10:   if $=?\ t\ \overline{a}$ then               (* tree t is leaf?              *)
11:     $y \Leftarrow (t.y)$;                  (* add leaf to list y           *)
12:   else
13:     $(l.(d.r)) \Leftarrow t$;              (* decompose node               *)
14:     $y \Leftarrow$ call $pre((r.y))$;        (* traverse right subtree r     *)
15:     $y \Leftarrow$ call $pre((l.y))$;        (* traverse left  subtree l     *)
16:     $y \Leftarrow (d.y)$;                  (* add label d to list y        *)
17:   fi $=?$ hd$(y)\ \overline{a}$;             (* head of list y is leaf?      *)
18:   return $y$;
```

Fig. 2. Translation between infix expressions and Polish notation in R-WHILE.

t is added to the list y by the reversible replacement (line 11). Otherwise, in the else-branch, *pre* calls itself recursively on the right and left subtrees r and l with the current y (lines 14–15). List y is built from right to left, so d is added to y after both subtrees are translated. The arity of all procedures is one, so it is convenient to decompose the argument of *pre* by pattern $(t.y)$ (line 9).

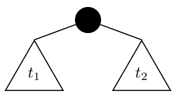
The inverse semantics of a procedure can be invoked by uncall. In procedure *pre2infix*, the reversible replacement $(t.\overline{nil}) \Leftarrow$ uncall $pre(y)$ is the inverse of $y \Leftarrow$ call $pre((t.\overline{nil}))$. For example, the infix expression $t = ((\overline{a}\,.\,(\overline{d}\,.\,\overline{a}))\,.\,(\overline{d}\,.\,\overline{a}))$ is translated into Polish notation $y = (\overline{d}\,.\,(\overline{d}\,.\,(\overline{a}\,.\,(\overline{a}\,.\,(\overline{a}\,.\,\overline{nil})))))$, and vice versa.

4 An Intrinsically Reversible Semantics

In this section, we illustrate the principle of reversible semantics by constructing a denotational semantics for R-WHILE with procedures using sets and partial injective functions. This is done first by constructing the domains of computation, and then constructing a denotation function for each syntactic category. While this results in a semantics for R-WHILE with procedures, we stress rather the use of abstract concepts (*e.g.*, cartesian products, disjoint unions, traces, and fixed points) to construct these semantics in reversible programming languages in general, rather than the concrete realization of R-WHILE with procedures.

4.1 States and Values

We begin by constructing the appropriate domains of computation for values and states. To do this, we assume that we are given an alphabet Λ of *symbols*, elements of which we denote using an overline, *e.g.*, $\overline{a}$, $\overline{nil}$, etc. The set of values $\mathbb{V}$ is then constructed as the set of binary trees with elements from Λ at the leaves. More formally, this set can be constructed by the least fixed point of sets $\mathbb{V} = \mu X.\Lambda \oplus (X \otimes X)$. If t_1 and t_2 are such binary trees, we will use the notation $t_1 \bullet t_2$ (read: "t_1 cons t_2") to mean the binary tree constructed from t_1 and t_2:



Using this definition, the set of states Σ can be constructed as *colists* (*i.e.*, lists of infinite length) of values, that is, $\Sigma = \mathbb{V}^\omega$ (explicitly, this is constructed as the largest fixed point $\nu X.I \oplus (\mathbb{V} \otimes X)$). In this semantics, we assume the number of non-nil values in a state is finite. By associating each distinct variable (of which there are countably many) with an index in ω, a state is then precisely a description of the contents of all variables. In keeping with this principle, we shall write variables as x_1, x_2, x_3, etc. rather than (as is usual) as x, y, z etc.

4.2 Expressions

Expressions are interpreted as partial injective functions with the signature:

$$\Sigma \otimes \mathbb{V} \xrightarrow{[\![e]\!]_{\text{exp}}} \Sigma \otimes \mathbb{V} \ .$$

Regardless of their concrete form, interpretations of expressions are defined as

$$[\![e_1]\!]_{\text{exp}}(\sigma, v) = \begin{cases} (\sigma, [\![e_1]\!]^\sigma_{\text{exp}}) & \text{if } v = \overline{nil} \\ (\sigma, \overline{nil}) & \text{if } v = [\![e_1]\!]^\sigma_{\text{exp}} \neq \overline{nil} \end{cases}$$

where $[\![e]\!]^\sigma_{\text{exp}} \in \mathbb{V}$, given below, is understood as the interpretation of e in the state σ. Concretely, it is defined as follows, depending on the form of e:

$$\begin{aligned}
[\![x_i]\!]^\sigma_{\text{exp}} &= v_i \quad \text{where } \sigma = (v_1, v_2, \ldots, v_i, \ldots) \\
[\![\overline{s_1}]\!]^\sigma_{\text{exp}} &= \overline{s_1} \\
[\![\mathsf{cons}(e_1, e_2)]\!]^\sigma_{\text{exp}} &= [\![e_1]\!]^\sigma_{\text{exp}} \bullet [\![e_2]\!]^\sigma_{\text{exp}} \\
[\![\mathsf{hd}(e_1)]\!]^\sigma_{\text{exp}} &= \begin{cases} v_1 \text{ if } [\![e_1]\!]^\sigma_{\text{exp}} = v_1 \bullet v_2 \\ \uparrow \ \text{otherwise} \end{cases} \\
[\![\mathsf{tl}(e_1)]\!]^\sigma_{\text{exp}} &= \begin{cases} v_2 \text{ if } [\![e_1]\!]^\sigma_{\text{exp}} = v_1 \bullet v_2 \\ \uparrow \ \text{otherwise} \end{cases} \\
[\![=?\ e_1\ e_2]\!]^\sigma_{\text{exp}} &= \begin{cases} \overline{nil} \bullet \overline{nil} \text{ if } [\![e_1]\!]^\sigma_{\text{exp}} = [\![e_2]\!]^\sigma_{\text{exp}} \\ \overline{nil} \qquad \text{otherwise} \end{cases}
\end{aligned}$$

As such, the meaning of a variable in a state is given by its contents, and the meaning of a symbol is given by its direct representation in the alphabet Λ. The meaning of the cons of expressions is given by the cons of their meanings, while the head (and the tail) of an expression takes the head (respectively tail) of its meaning, diverging if not of this form.

The use of a non-injective function in part of an injective function is frequently found in the context of reversible computing and the one used above is generally referred to as the Bennett method.

4.3 Patterns

Since patterns may include procedure invocation, the meaning of a pattern depends on the program context ϕ in which it is interpreted. Patterns in a program context are all interpreted as partial injective functions with the signature

$$\Sigma \xrightarrow{[\![q]\!]^{\phi}_{\text{pat}}} \Sigma \otimes \mathbb{V} \ .$$

In particular, note that this signature allows patterns to alter the state. Indeed, patterns may have side effects (here, in the form of altering the store): They should be regarded as a means to prepare a given value in a state, in such a way that may alter the state it began with. The interpretation of patterns is defined as follows, depending on the form of p:

$$\begin{aligned}
[\![x_i]\!]^{\phi}_{\text{pat}}(\sigma) &= ((v_1, \dots, v_{i-1}, \overline{nil}, \dots), v_i) \quad \text{where } \sigma = (v_1, \dots, v_{i-1}, v_i, \dots) \\
[\![\mathsf{call}\ f_i(q_1)]\!]^{\phi}_{\text{pat}}(\sigma) &= (\sigma', (\kappa_i^{\dagger} \circ \phi \circ \kappa_i)(v)) \qquad \text{where } (\sigma', v) = [\![q_1]\!]^{\phi}_{\text{pat}}(\sigma) \\
[\![\mathsf{uncall}\ f_i(q_1)]\!]^{\phi}_{\text{pat}}(\sigma) &= (\sigma', (\kappa_i^{\dagger} \circ \phi^{\dagger} \circ \kappa_i)(v)) \quad \text{where } (\sigma', v) = [\![q_1]\!]^{\phi}_{\text{pat}}(\sigma) \\
[\![\mathsf{cons}(q_1, q_2)]\!]^{\phi}_{\text{pat}}(\sigma) &= (\sigma'', v_1 \bullet v_2) \\
&\qquad \text{where } (\sigma', v_1) = [\![q_1]\!]^{\phi}_{\text{pat}}(\sigma) \text{ and } (\sigma'', v_2) = [\![q_2]\!]^{\phi}_{\text{pat}}(\sigma')
\end{aligned}$$

The meaning of a variable, as a pattern, is to simultaneously extract its contents and clear it. A procedure call $\mathsf{call}\ f_i(q_1)$ is interpreted as essentially passing the meaning of q_1 to the i'th component of the program context and extracting from the i'th component afterwards, which, as we will see in Section 4.6, corresponds precisely to invoking the i'th procedure. Uncall to a procedure is handled analogously, but using the *inverse* to the program context instead. Finally, the meaning of a cons pattern is as a kind of sequential composition: First, the meaning of q_1 is executed, resulting in a new state σ' and value v_1. Then, the meaning of q_2 is executed in this new state σ', yielding a final state σ'' and value v_2. The two values are then consed together, finally resulting in the state σ'' and prepared value $v_1 \bullet v_2$.

Uncall to a procedure is handled using the inverse procedure instead of the inverse to the meaning of the procedure if the inverse procedure is in the procedure context ϕ. We will see how to add the inverse procedures to the procedure environment in Section 4.6.

4.4 Predicates

The predicate interpretation provides a different way of interpreting expressions which are to be used to determine branching of control flow. They are interpreted as partial injective functions with the signature

$$\Sigma \xrightarrow{[\![e]\!]_{\text{pred}}} \Sigma \oplus \Sigma \ .$$

The definition is based on the convention that an expression interpreted as $\overline{nil}$ in a state σ is considered to be *false* in σ, and *true* otherwise. The predicate interpretation of an expression e is defined as follows:

$$[\![e_1]\!]_{\text{pred}}(\sigma) = \begin{cases} \mathsf{inl}\ \sigma & \text{if } [\![e_1]\!]^{\sigma}_{\text{exp}} \neq \overline{nil} \\ \mathsf{inr}\ \sigma & \text{otherwise} \end{cases}$$

As such, the predicate interpretation of e_1 sends control flow to the first component if e_1 is considered true in the given state, and to the second component otherwise. As we will see in Section 4.5, this style makes for a straightforward interpretation of *conditional execution* of commands. Here, inl corresponds to true and inr to false in the semantics level.

4.5 Commands

Commands in `R-WHILE` with procedures are interpreted as invertible state transformations, *i.e.*, as partial injective functions with signature

$$\Sigma \xrightarrow{[\![c]\!]^{\phi}_{\text{cmd}}} \Sigma \ .$$

The interpretation of commands is defined as follows, depending on the syntactic form of c:

$$\begin{aligned}
[\![c_1; c_2]\!]^{\phi}_{\text{cmd}} &= [\![c_2]\!]^{\phi}_{\text{cmd}} \circ [\![c_1]\!]^{\phi}_{\text{cmd}} \\
[\![x_i \mathrel{\widehat{=}} e_1]\!]^{\phi}_{\text{cmd}} &= ([\![x_i]\!]^{\phi}_{\text{pat}})^{\dagger} \circ [\![e_1]\!]_{\text{exp}} \circ [\![x_i]\!]^{\phi}_{\text{pat}} \\
[\![q_1 \Leftarrow q_2]\!]^{\phi}_{\text{cmd}} &= ([\![q_1]\!]^{\phi}_{\text{pat}})^{\dagger} \circ [\![q_2]\!]^{\phi}_{\text{pat}} \\
[\![\mathsf{if}\ e_1\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}\ e_2]\!]^{\phi}_{\text{cmd}} &= [\![e_2]\!]^{\dagger}_{\text{pred}} \circ ([\![c_1]\!]^{\phi}_{\text{cmd}} \oplus [\![c_2]\!]^{\phi}_{\text{cmd}}) \circ [\![e_1]\!]_{\text{pred}} \\
[\![\mathsf{from}\ e_1\ \mathsf{do}\ c_1\ \mathsf{loop}\ c_2\ \mathsf{until}\ e_2]\!]^{\phi}_{\text{cmd}} &= \mathrm{Tr}\left(([\![c_2]\!]^{\phi}_{\text{cmd}} \oplus \mathrm{id}) \circ [\![e_2]\!]_{\text{pred}} \circ [\![c_1]\!]^{\phi}_{\text{cmd}} \circ [\![e_1]\!]^{\dagger}_{\text{pred}}\right)
\end{aligned}$$

Note in particular the use of *inverses* to patterns and predicates in the definition above. The inverse to a predicate corresponds to its corresponding *assertion*, whereas the inverse to a pattern performs *state preparation* consuming (part of) a value (rather than, in the forward direction, *value preparation* consuming part of a state).

Pattern inverses are illustrated in both reversible assignments and pattern matching, each consisting of a value preparation (indeed, the expression interpretation can be seen as *side-effect free* value preparation), using the interpretation of patterns, followed by a state preparation using the inverse. Similarly, the interpretation of conditionals and loops relies on predicate inverses: In both cases, they serve as conditional join points, corresponding to an assertion that e_2 is expected to be true when coming from the then branch of the conditional (respectively from the *outside* of the loop), and false when coming from the *else* branch (respectively from the *inside* of the loop).

4.6 Procedures

Since `R-WHILE` with procedures only uses local state, procedure definitions are interpreted (in a program context) as partial injective value transformations, *i.e.*, partial injective functions of the form

$$\mathbb{V} \xrightarrow{[\![f]\!]^{\phi}_{\text{proc}}} \mathbb{V} \ .$$

To define the interpretation of procedures, we need an injective helper function $\mathbb{V} \xrightarrow{\xi} \Sigma \otimes \mathbb{V}$ given by

$$\xi(v) = (\vec{\sigma}, v) \ ,$$

where $\vec{\sigma} = (\overline{nil}, \overline{nil}, \dots)$ is the state in which all variables are cleared (*i.e.*, contain $\overline{nil}$). This *canonical state* is the initial computation state in which all procedures are executed. A procedure definition in the program context ϕ is interpreted as

$$[\![\mathsf{proc}\ f(q_1); c; \mathsf{return}\ q_2]\!]^{\phi}_{\mathrm{proc}} = \xi \circ [\![q_2]\!]_{\mathrm{pat}} \circ [\![c]\!]^{\phi}_{\mathrm{cmd}} \circ ([\![q_1]\!]^{\phi}_{\mathrm{pat}})^{\dagger} \circ \xi^{\dagger} \ .$$

This definition should be read as follows: In the canonical state $\vec{\sigma}$, the state described by the inverse interpretation of the input pattern q_1 is first prepared. Then, the body of the procedure is executed, resulting in a new state which is then used to prepare a value as specified by interpretation of the output pattern q_2. At this point, the system *must* again be in the canonical state $\vec{\sigma}$, which, if this is the case, can then be discarded, leaving only the output value.

4.7 Programs

Finally at the level of programs, these are interpreted as the meaning of their topmost defined procedure, and, as such, are interpreted as partial injective functions of the signature

$$\mathbb{V} \xrightarrow{[\![m]\!]_{\mathrm{prg}}} \mathbb{V} \ .$$

Since procedures may be defined to invoke themselves as well as other procedures, we need to wrap them in a fixed point, passing the appropriate program context ϕ to each procedure interpretation. This yields the interpretation

$$[\![f_1 \cdots f_n]\!]_{\mathrm{prg}} = \kappa_1^{\dagger} \circ (\mu\phi.[\![f_1]\!]^{\phi}_{\mathrm{proc}} \oplus \cdots \oplus [\![f_n]\!]^{\phi}_{\mathrm{proc}}) \circ \kappa_1 \ .$$

Note the inner interpretation of procedures $f_1 \cdots f_n$ as a disjoint union $[\![f_1]\!]^{\phi}_{\mathrm{proc}} \oplus \cdots \oplus [\![f_n]\!]^{\phi}_{\mathrm{proc}}$: This gives one large partial injective function, which behaves just the partial injective functions $[\![f_i]\!]^{\phi}_{\mathrm{proc}}$ when inputs are injected into the i'th component, save for the fact that outputs (if any) are also placed in the i'th component. This explains the need for injections κ_i and quasiprojections $\kappa_i^{\dagger}$ in the definition of procedure calls in Section 4.3.

Proposition 1. *`R-WHILE` with procedures is compositionally reversible.*

Proof (Sketch). We will prove the proposition by structural induction on syntax elements. For the base step, the meanings of atomic functions $[\![x_i]\!]^{\phi}_{\mathrm{pat}}$, $[\![e_1]\!]_{\mathrm{pred}}$, and ξ are straightforwardly verified to be partial injective functions (so invertible). For the induction step, the meanings of function compositions are given purely using means that preserve invertibility (trace, fixed points, composition, inversion, cartesian product, disjoint union). □

The interpretation ($[\![\cdot]\!]_{\mathrm{prg}}$, $[\![\cdot]\!]^{\phi}_{\mathrm{proc}}$, and $[\![\cdot]\!]^{\phi}_{\mathrm{cmd}}$) maps syntax to injective (value, store, ...) transformations (on stores, values). The injective (value, store, ...) transformations can be expressed in $\mathcal{M}$.

4.8 Use of Inverse Semantics

In conventional programming languages, the programs are not guaranteed to be injective, obtaining the inverse programs needs global analysis, and the inverse interpretation requires extra overhead. However, owing to the formalization, the programs in the object languages expressed in $\mathcal{M}$ are always injective, their inversion can be obtained by recursive descendent transformation, and the access to their inverse semantics tends to require only the constant time overhead. Moreover, our metalanguage is useful to derive the inverse program. For any command c, the inverse of the semantics $(\llbracket c \rrbracket^{\phi}_{\mathrm{cmd}})^{\dagger}$ can be a composition of the semantics of its components and traces, which can be mechanically obtained by properties of **PInj** [5]. For example, we have $(\llbracket q_1 \Leftarrow q_2 \rrbracket^{\phi}_{\mathrm{cmd}})^{\dagger} = ((\llbracket q_1 \rrbracket^{\phi}_{\mathrm{pat}})^{\dagger} \circ \llbracket q_2 \rrbracket^{\phi}_{\mathrm{pat}})^{\dagger} = (\llbracket q_2 \rrbracket^{\phi}_{\mathrm{pat}})^{\dagger} \circ \llbracket q_1 \rrbracket^{\phi}_{\mathrm{pat}}$, and hence we obtain the inverse program $(\llbracket q_1 \Leftarrow q_2 \rrbracket^{\phi}_{\mathrm{cmd}})^{\dagger} = \llbracket q_2 \Leftarrow q_1 \rrbracket^{\phi}_{\mathrm{cmd}}$. The right hand sides of the semantic function of commands are mostly symmetric. Therefore, the inversion rules for them are mechanically obtained in the similar way. The only exception is the loop, which requires an extra identity $\mathrm{Tr}((f_1 \oplus \mathrm{id}) \oplus f_2) = \mathrm{Tr}(f_2 \oplus (\mathrm{id} \oplus f_1))$ to obtain the inversion rule. The similar anti-symmetry appears in the operational semantics in Janus [19], in which the rule of the loop can be either right or left recursive.

In the above pattern evaluation, the inverse semantics $\phi^{\dagger}$ of the program context is used to realize uncall of procedures. The inverse semantics of procedures is equal to the semantics of inverted procedures. This leads to an alternative realization of the same meaning. First, we add the inverses of procedures in the program context:

$$\mu\phi.\ \llbracket f_1 \rrbracket^{\phi}_{\mathrm{proc}} \oplus \cdots \oplus \llbracket f_n \rrbracket^{\phi}_{\mathrm{proc}} \oplus (\llbracket f_1 \rrbracket^{\phi}_{\mathrm{proc}})^{\dagger} \oplus \cdots \oplus (\llbracket f_n \rrbracket^{\phi}_{\mathrm{proc}})^{\dagger}$$

Then we uncall a procedure i in both forward and backward evaluation is handled by invoking the $(i + n \mod 2n)$-th procedure.

5 Related Work

Formal meaning has been given to reversible programming languages using well-established formalisms such as operational semantics to the imperative language Janus [19], the functional language RFUN [17] and the concurrent languages [13], small-step operational semantics to the assembler language PISA [1], transition functions to the flowchart language RFCL [18], and denotational semantics to `R-WHILE` [6, 7]. The reversibility of a language is not directly expressed by these formalisms. It is up to the language designer's discipline to express reversibility and to show the inversion properties for each language individually. Also, the semantics of `R-WHILE` was first expressed irreversibly [19]. Type and effects systems were studied for reversible languages [10].

The approach taken in this paper is to compose reversible elements by the metalanguage $\mathcal{M}$ in ways that preserve their reversibility. Compositional approaches to reversibility have been used in various disguises including the diagrammatic composition of reversible circuits from reversible logic gates and reversible structured flowcharts from reversible control-flow operators [18]. Similarly, reversible Turing machines were built from reversible rotary elements [14].

Implementing a language by translation and interpretation is another operational approach to a reversible semantics. Examples include reversible interpreters [6], translation of the high-level language R to the assembler language PISA [3], and mapping hardware descriptions in SyReC to reversible circuits [16]. Reversible cellular automata may have non-injective local maps, but if the local map is injective, the update by the global map is guaranteed to be reversible [12].

6 Conclusion

Reversible systems have reversible semantics. In the present paper, we build upon a reversible semantics foundation intended to describe the semantics of reversible languages, which we demonstrate by the full development of a semantics for the reversible language `R-WHILE`. That the semantics foundation can formalize a language that is as computationally as powerful as any reversible programming language can be, also means that in general its expressiveness suffices for describing reversible programming languages. The metalanguage allowed us to concisely formalize features representative of many reversible programming languages, including iteration, recursion, pattern matching, dynamic data structures, and the access to a program's inverse semantics.

It could be interesting to further explore how object-oriented structures, combinators, or features for concurrency are best described and which metalanguage features may be useful. Characterizing reversible heap allocation and concurrent reversible computation are some of those challenges. However, its practical feasibility and relationship to advanced reversible automata including nondeterminism, *e.g.*, [9] remains to be explored.

Acknowledgments. The third author is supported by JSPS KAKENHI Grant Number 18K11250.

References

1. H. B. Axelsen, R. Glück, T. Yokoyama. Reversible machine code and its abstract processor architecture. *CSR*, *LNCS*, Vol. 4649, 56–69. Springer-Verlag, 2007.
2. R. Cockett, S. Lack. Restriction categories III: Colimits, partial limits and extensivity. *Mathematical Structures in Computer Science*, 17(4):775–817, 2007.
3. M. P. Frank. *Reversibility for Efficient Computing.* PhD thesis, MIT, 1999.
4. B. G. Giles. *An Investigation of some Theoretical Aspects of Reversible Computing.* PhD thesis, University of Calgary, Canada, 2014.
5. R. Glück, R. Kaarsgaard. A categorical foundation for structured reversible flowchart languages: Soundness and adequacy. *Log. Meth. Comp. Sci.*, 14(3), 2018.
6. R. Glück, T. Yokoyama. A minimalist's reversible while language. *IEICE Transactions on Information and Systems*, E100-D(5):1026–1034, 2017.
7. R. Glück, T. Yokoyama. Constructing a binary tree from its traversals by reversible recursion and iteration. *IPL*, 147:32–37, 2019.
8. E. Haghverdi. *A Categorical Approach to Linear Logic, Geometry of Proofs and Full Completeness.* PhD thesis, Carlton University and University of Ottawa, 2000.

9. M. Holzer, M. Kutrib. Reversible nondeterministic finite automata. In I. Phillips, H. Rahaman (eds.), *RC*, *LNCS*, Vol. 10301, 35–51. Springer, 2017.
10. R. P. James, A. Sabry. Information effects. In *POPL*, 73–84. ACM Press, 2012.
11. R. Kaarsgaard, H. B. Axelsen, R. Glück. Join inverse categories and reversible recursion. *J. Log. Algebr. Methods*, 87:33–50, 2017.
12. J. Kari. Reversible cellular automata: from fundamental classical results to recent developments. *New Gener. Comput.*, 36(3):145–172, 2018.
13. S. Kuhn, I. Ulidowski. A calculus for local reversibility. In S. Devitt, I. Lanese (eds.), *RC*, *LNCS*, Vol. 9720, 20–35. Springer-Verlag, 2016.
14. K. Morita. Reversible computing and cellular automata — A survey. *Theor. Comput. Sci.*, 395(1):101–131, 2008.
15. P. Selinger. A survey of graphical languages for monoidal categories. In B. Coecke (ed.), *New Structures for Physics*, *LNP*, Vol. 813, 289–355. Springer-Verlag, 2011.
16. R. Wille, et al. SyReC: A hardware description language for the specification and synthesis of reversible circuits. *Integration*, 53:39–53, 2016.
17. T. Yokoyama, H. B. Axelsen, R. Glück. Towards a reversible functional language. In A. De Vos, R. Wille (eds.), *RC*, *LNCS*, Vol. 7165, 14–29. Springer-Verlag, 2012.
18. T. Yokoyama, H. B. Axelsen, R. Glück. Fundamentals of reversible flowchart languages. *Theor. Comput. Sci.*, 611:87–115, 2016.
19. T. Yokoyama, R. Glück. A reversible programming language and its invertible self-interpreter. In *PEPM*, 144–153. ACM Press, 2007.