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, Denmark
glueck@acm.org, robin@di.ku.dk

[2] Department of Software Engineering, Nanzan University, Nagoya, Japan
tyokoyama@acm.org

Abstract. During the past decade, reversible programming languages have been formalized using various established semantic frameworks. However, these semantics fail to effectively specify the distinct properties of reversible languages at the metalevel, and even neglect the central question of whether the defined language is reversible. In this paper, we build on a metalanguage foundation for reversible languages based on the category of sets and partial injective functions. We exemplify our approach through step-by-step development of the full semantics of an r-Turing complete reversible while-language with recursive procedures. This yields 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 further discuss applications and future research directions for reversible semantics.

1 Introduction

Over the past years, reversible programming languages ranging from imperative to functional and object-oriented languages have been formalized using established semantic frameworks, such as state transition functions, structural operational semantics and, recently, denotational semantics (*e.g.* [7,8,21,22]). These frameworks, which have been used to provide meaning to advanced language features and computation models, such as nondeterminism and parallelism, have turned out to be ineffective at specifying the distinct semantic properties of reversible languages. Immediate answers to questions regarding the uniqueness of the inverse semantics, the inversion of programs and, in particular, the central question of whether a language is reversible, are unavailable.

In this paper, we build on a metalanguage foundation for reversible languages based on the category **PInj** of sets and partial injective functions. The rationale behind this approach is straightforward: Interpretations of syntax are composed in ways that preserve their injectivity. More specifically, interpretations of syntax are composed of sequential composition, cartesian product, disjoint union, function inversion, iteration, and recursion. To achieve this, we make use of the categorical foundation developed elsewhere (*e.g.* [5,13]). Our approach exploits

© Springer Nature Switzerland AG 2020
E. Sekerinski et al. (Eds.): FM 2019 Workshops, LNCS 12233, pp. 413–427, 2020.
https://doi.org/10.1007/978-3-030-54997-8_26

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 provide a clean reversible semantics to a reversible language.

We demonstrate the aforementioned idea through step-by-step development of a full formal semantics for the reversible procedural language `R-WHILE`, which includes iteration and recursion. This leads to a formal semantics in which the reversibility of the language and its inverse semantics are immediate, along with program inversion. The reversibility of the language follows immediately from the formalization. It is apparent from the signatures of semantic functions that the language is clean and without any hidden tracing. Note that this approach is independent of the specific details of the language and can be extended to other ways of composing semantic functions, provided their injectivity is preserved.

`R-WHILE` with procedures is a reversible while-language with structured control-flow operators and dynamic data structures [7,8].[1] This language is reversibly universal (r-Turing complete), 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 inverse invocation of recursive procedures.

The metalanguage used here has a distinct property familiar from reversible programming: It is not possible to define an irreversible (non-injective) language semantics. To ensure reversibility, conventional metalanguages require discipline in the formalization, *e.g.*, a standard denotational semantics. In the case of operational semantics, it is quite unclear how to restrict an inference system to a purely reversible one. One possible future direction is an investigation of metalanguage extensions to capture other composition forms and language features, which may include object-oriented features, combinators, and machine languages.

Overview: Sect. 2 introduces the elements of the formal semantics and Sect. 3 describes the reversible language `R-WHILE` with procedures. In Sect. 4, the formal semantics of the language is developed in a step-by-step manner. Sections 5 and 6 present related work, concluding remarks, and directions for future work. We assume the reader is familiar with the basic concepts of reversible languages (*e.g.*, [21]) and formal semantics (*e.g.*, [19]).

2 Elements of the Formal Semantics

This section is concerned with some of the details of sets and partial injective functions as they will be used in the following sections (compare, *e.g.*, [2,4,17]). While the constructions mentioned in this section are extracted 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).

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

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 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 $X \xrightarrow{\mathrm{id}} X$ 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 method utilizes 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 partial injective function 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 projections (such as $X \otimes Y \xrightarrow{\pi_1} Y$ given by $\pi_1(x, y) = x$) are *unavailable*, as these are never injective. We denote the unit, up to bijective correspondence, of the cartesian product (any distinguished singleton set is acceptable) by 1.

The second method of 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 ($\mathsf{inl}\ \cdot$) or right ($\mathsf{inr}\ \cdot$) 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, $X \oplus X' \xrightarrow{f \oplus g} Y \oplus Y'$, which performs a case analysis of the inputs and tags the outputs with their origins:

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

The cartesian product loses its projections in the setting of partial injective functions. However, the disjoint union retains its usual *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 $\kappa_2(y) = \mathsf{inr}\ y$, respectively. Note in particular that since we consider *partial* injective functions, these injections 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 $\kappa_1^\dagger(\mathsf{inr}\ y) = \uparrow$.

2.2 Fixed Points and Iteration

Both sets and partial injective functions are well-behaved as regards 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), respectively. For example, the set of flat 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 functions, is that the set of all partial functions with a 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). Thus, 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 $X \xrightarrow{\mathrm{Tr}(f)} Y$, which is 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 that 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 fails to do so. In the latter 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 defined using a tail-recursively described pretrace $\mathbf{PInj}(X \oplus U, Y \oplus U) \xrightarrow{\mathsf{pretrace}} \mathbf{PInj}(X \oplus U, Y)$, which is defined as follows:

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

Hence, it is defined simply as $\mathrm{Tr}(f)(x) = \mathsf{pretrace}(f)(\mathsf{inl}\ x)$. The data flow of $\mathrm{Tr}(f)$ is illustrated in Fig. 1, in which the flow is from left to right and the feedback loop represents the repeated application of f to elements of U.

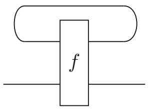


Fig. 1. Data flow of trace $\mathrm{Tr}(f)$ with feedback loop.

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 $X \oplus U \xrightarrow{f} Y \oplus U$. (Formally, the trace operator can also be defined as a fixed point using the *trace formula*, see [9].)

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

Fig. 2. Syntax of the reversible language `R-WHILE` with procedures.

2.3 Summary of the Metalanguage

Collecting the injective constructs for the formal semantics introduced above, we can specify a clean reversible metalanguage $\mathcal{L}$ to describe objects of **PInj**:

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

An atomic function a can be any auxiliary partial injective function, such as “swap” for cartesian products: $swap_\otimes(x, y) = (y, x)$. For any expression in $\mathcal{L}$, the least fixed point exists. The formal argument of the least fixed point ϕ expects a *program context*, *i.e.*, a disjoint union of partial injective functions. $\mathcal{L}$ is closed under inversion, and the inverse semantics of each expression is unique and immediate. Any language described by the metalanguage is (compositionally) reversible. $\mathcal{L}$ is sufficiently expressive for full formalization of the semantics of reversibly-universal languages. This is demonstrated below for `R-WHILE`.

3 R-WHILE with Reversible Recursion and Iteration

We informally describe the semantics of the reversible language `R-WHILE` with procedures, and illustrate it with a recursive program that translates infix expressions to Polish notation; this is a classic translation that is reversible. The data domain of the language is tree-structured data (lists known from Lisp and many modern languages). Readers familiar with reversible programming can skip to Example 1 below and return to the informal description later.

The syntax of the language [8] is shown in Fig. 2. A *program* m is a sequence of procedures $p \cdots p$, where the topmost procedure is the main procedure. A *procedure* p has a name f, an argument pattern q, a command c as its body, and a return pattern q. The input to and output from a procedure is through the argument and return patterns, respectively. All procedures have arity and coarity one. Thus, it is convenient to compose and decompose input and output values via patterns.

A *command* c is a *reversible assignment* $x \mathrel{\widehat{=}} e$, a *reversible replacement* $q \Leftarrow q$, a *reversible conditional* if...fi, or a *reversible loop* from...until. The latter are two control structures familiar from reversible flowchart languages (*e.g.*, [21]).

The variable x in a reversible assignment $x \mathrel{\widehat{=}} e$ must not occur in expression e, which calculates a value (*e.g.*, $x \mathrel{\widehat{=}} x$ is not well formed). It is important

```
proc $infix2pre(t)$                  (* infix exp to Polish notation *)
 $y \Leftarrow$ call $pre((t.\overline{nil}))$;      (* call preorder traversal      *)
 return $y$;

proc $pre2infix(y)$                  (* Polish notation to infix exp *)
 $(t.\overline{nil}) \Leftarrow$ uncall $pre(y)$;     (* uncall preorder traversal    *)
 return $t$;

proc $pre((t.y))$                    (* recursive preorder traversal *)
 if $=?\ t\ \overline{0}$ then                  (* tree t is a leaf?            *)
    $y \Leftarrow (t.y)$                      (* add leaf to list y           *)
 else
    $(l.(d.r)) \Leftarrow t$;                  (* decompose tree t             *)
    $y \Leftarrow$ call $pre((r.y))$;          (* traverse right subtree r     *)
    $y \Leftarrow$ call $pre((l.y))$;          (* traverse left  subtree l     *)
    $y \Leftarrow (d.y)$                      (* add label d to list y        *)
 fi $=?$ hd$(y)\ \overline{0}$;                 (* head of list y is a leaf?    *)
 return $y$;
```

Fig. 3. Translation between infix expressions and Polish notation in R-WHILE.

to note that the assignment sets x to the value of e if the value of x is $\overline{nil}$, and sets x to $\overline{nil}$ if the values of x and e are equal; otherwise, it is undefined. In other words, a variable is updated or cleared depending on the original value of x. This definition ensures the reversibility of assignments.

No value is duplicated by a reversible replacement $q_1 \Leftarrow q_2$. Before the value constructed by q_2 is *matched* with q_1, all variables in q_2 are nil-cleared. Thus, the same variable may occur on both sides of a replacement (unlike an assignment).

Patterns play a central role in the construction and deconstruction of values, and are used in both ways (*e.g.*, reversible replacements). A *pattern* q is a variable x, a symbol $\overline{s}$, a pair of patterns $(q.q)$, or an invocation or inverse invocation of a procedure by call $f(q)$ or uncall $f(q)$, respectively. Patterns are linear (no variable occurs more than once in a pattern). The semantics of a procedure uncall is the inverse semantics of a procedure call. Procedures can only be invoked in patterns.

Expressions are conventional. An *expression* e is 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, or equality test $=?$. The variables in a program are denoted by small letters, such as l, d, r, and symbols are overlined, such as $\overline{nil}$.

Example 1. There are many practical applications of translating infix expressions to Polish notation, and vice versa. Because this function is injective, it can be programmed cleanly in a reversible language and run in both directions.

In R-WHILE, infix expressions can be represented by full binary trees

$$tree\ ::= \overline{0} \mid (tree\,.\,(\overline{1}\,.\,tree)),$$

where symbols $\overline{0}$ and $\overline{1}$ stand for an operand (leaf) and an operator (inner label) in an expression, respectively. For simplicity, we only use these two symbols.

Figure 3 shows the recursive procedure *pre*, which reversibly translates an infix expression to a prefix expression (Polish notation) via a preorder traversal of the full binary tree t representing the infix expression. Procedure *pre* is called and uncalled in the two procedures *infix2pre* and *pre2infix* for translating to Polish notation, and vice versa. For example, the infix expression $t = ((\overline{0} \,.\, (\overline{1} \,.\, \overline{0})) \,.\, (\overline{1} \,.\, \overline{0}))$ translates to Polish notation $y = (\overline{1} \,.\, (\overline{1} \,.\, (\overline{0} \,.\, (\overline{0} \,.\, (\overline{0} \,.\, \overline{nil})))))$.

In *infix2pre*, the translation is invoked by a call to *pre* (line 2)

$$y \Leftarrow \mathsf{call}\ pre((t.\overline{nil})),$$

where the argument of the call is a singleton list $(t.\overline{nil})$ containing t, and the result is matched with the trivial pattern y, which binds the value to y. In *pre2infix*, the inverse computation of *pre* is invoked by an uncall of *pre* (line 6)

$$(t.\overline{nil}) \Leftarrow \mathsf{uncall}\ pre(y),$$

where y is the argument and t is picked from the resulting singleton list.

The body of *pre* is a reversible conditional $\mathsf{if}...\mathsf{fi}$ (lines 10–17) with an entry test $=?\ t\ \overline{0}$ and an exit assertion $=?\ \mathsf{hd}(y)\ \overline{0}$. If t is a leaf $\overline{0}$, t is added to list y by $y \Leftarrow (t.y)$ (line 11). Otherwise, in the else-branch, *pre* calls itself recursively on the right and left subtrees r and l with the current list y (lines 14–15). The two subtrees and label d are selected from t by $(l.(d.r)) \Leftarrow t$. List y is constructed from right to left; thus, d is added after both subtrees are translated (line 16). The arity of all procedures is one; therefore, it is convenient to decompose the argument value by pattern $(t.y)$ already in the head of *pre* (line 9).

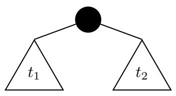
4 An Intrinsically Reversible Semantics

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. First, the domains of computation are constructed, followed by a semantic function for each syntactic category. While this approach yields a semantics for R-WHILE with procedures, for construction of such semantics for reversible programming languages in general, we stress the use of abstract concepts (*e.g.*, cartesian products, disjoint unions, traces, and fixed points), rather than the concrete realization of R-WHILE with procedures.

In the following, we use standard notation of denotational semantics [19], including brackets for semantic functions $[\![\cdot]\!]$, which show that the domain of the argument is syntax.

4.1 States and Values

We begin by constructing appropriate domains of computation for values and states. To achieve this, we assume that we are given an alphabet Λ of *symbols*, elements of which we denote using an overline, *e.g.*, $\overline{0}$, $\overline{1}$, and $\overline{nil}$. 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 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. A state associates each variable with a value. The set of states Σ can be constructed as finitely supported *colists* (*i.e.*, lists of infinite length) of values; that is, $\Sigma = \mathbb{V} \otimes \mathbb{V} \otimes \cdots$ (explicitly, this is constructed as the largest fixed point $\nu X.I \oplus (\mathbb{V} \otimes X)$). By associating each variable in the language (of which there are countably many) with a distinct index, a state is then precisely a description of the contents of all variables. In keeping with this principle, we write variables as x_1, x_2, x_3, etc. rather than as x, y, z etc. Note that the number of non-nil values in a state of a given program is finite.



4.2 Expressions

In irreversible languages, expressions are usually interpreted as partial functions of the signature $\Sigma \to \mathbb{V}$. Obviously, because there are multiple states resulting in the same value, such a function is not injective and cannot be an atomic function a in the metalanguage $\mathcal{L}$. Instead, expressions are interpreted as partial injective functions with signature

$$\Sigma \otimes \mathbb{V} \xrightarrow{\mathcal{E}[\![e]\!]} \Sigma \otimes \mathbb{V}.$$

Regardless of their syntactic form, expression interpretation is defined as

$$\mathcal{E}[\![e_1]\!](\sigma, v) = \begin{cases} (\sigma, \mathcal{E}'[\![e_1]\!]\sigma) & \text{if } v = \overline{nil} \\ (\sigma, \overline{nil}) & \text{if } v = \mathcal{E}'[\![e_1]\!]\sigma \neq \overline{nil} \\ \uparrow & \text{otherwise} \end{cases}$$

where $\mathcal{E}'[\![e]\!]\sigma \in \mathbb{V}$, given below, is understood to be the value of e in the state σ. When v in $\mathcal{E}[\![e_1]\!](\sigma, v)$ is $\overline{nil}$, the value of e_1 in σ is obtained. When v is equal to the value of e_1 in σ, $\overline{nil}$ is obtained. In both cases, σ is left unchanged. Otherwise, the meaning is undefined. The semantic function defines a *reversible update* [1] of the value argument, which also implies that it is self-inverse.

Concretely, $\mathcal{E}'$ is defined as follows, depending on the form of e:

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

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 two expressions is given by the cons of their meanings, while the head (tail) of an expression takes the head (tail) of its meaning, diverging if not of this form. The meaning of equality test returns distinct values indicating whether the two expressions have the same value. Obviously, more operators can be added to this list.

A non-injective function is often used in the definition of an injective function in the context of reversible computation. Above, $\mathcal{E}[\![e]\!]$, a reversible update defined using non-injective $\mathcal{E}'[\![e]\!]$, is injective for any e. Because $\mathcal{E}[\![e]\!]$ is not defined exclusively in terms of the metalanguage, we regard it as defining an atomic function a of $\mathcal{L}$.

4.3 Patterns

As 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 signature

$$\Sigma \xrightarrow{\mathcal{Q}[\![q]\!]\phi} \Sigma \otimes \mathbb{V}.$$

In particular, note that this signature allows patterns to perform state alternations. 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 manner that may alter the state it began with. Pattern interpretation is defined as follows, depending on the form of q:

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

The meaning of a variable, as a pattern, is simultaneous extraction of its contents *and* clearing of the variable. A procedure call call $f_i(q_1)$ is interpreted as passing the value of q_1 to the i'th component of the program context ϕ, followed by extraction from the i'th component. As we discussed in Sect. 4.6, this corresponds precisely to invoking the i'th procedure. Uncalling of a procedure is handled analogously, but the *inverse* of the program context is used instead. Finally, the meaning of a cons pattern $(q_1.q_2)$ is as a kind of sequential composition: First, q_1 is executed, yielding a new state σ' and value v_1. Then, q_2 is executed in σ', yielding a final state σ'' and value v_2. The two values are then consed together, finally yielding the state σ'' and prepared value $v_1 \bullet v_2$. Recall that no variable occurs more than once in a pattern.

Alternatively, uncall can be defined using the inverted procedures instead of the inverse to the program context, $\phi^\dagger$, provided the inverted procedures are in ϕ. Addition of the inverse procedures to ϕ is discussed in Sect. 4.6.

4.4 Predicates

The predicate interpretation provides a different means of interpreting expressions for determining branching of control flow. They are interpreted as partial injective functions with signature

$$\Sigma \xrightarrow{\mathcal{T}[\![e]\!]} \Sigma \oplus \Sigma.$$

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:

$$\mathcal{T}[\![e_1]\!](\sigma) = \begin{cases} \mathsf{inl}\ \sigma \text{ if } \ \mathcal{E}'[\![e_1]\!]\sigma \neq \overline{nil} \\ \mathsf{inr}\ \sigma \text{ otherwise} \end{cases}$$

As such, the predicate interpretation of e_1 sends the control flow to the first component if e_1 is considered true in the given state, and to the second component otherwise. As discussed in Sect. 4.5, this style allows straightforward interpretation of the *conditional execution* of commands. Here, inl and inr correspond to true and false in the semantics level, respectively.

4.5 Commands

Commands are interpreted as invertible state transformations, *i.e.*, as partial injective functions with signature

$$\Sigma \xrightarrow{\mathcal{C}[\![c]\!]\phi} \Sigma.$$

Command interpretation is defined as follows, depending on the form of c:

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

Note the use of *inverses* to patterns and predicates in the above definition. 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 regarded 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

As procedures use the local state only, procedure definitions are interpreted (in a program context) as partial injective value transformations, *i.e.*, partial injective functions of the form

$$\mathbb{V} \xrightarrow{\mathcal{P}[\![f]\!]\phi} \mathbb{V}.$$

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

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

where $\vec{o} = (\overline{nil}, \overline{nil}, \dots)$ is the state in which all variables are cleared (*i.e.*, they 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

$$\mathcal{P}[\![\mathsf{proc}\ f(q_1)\ c; \mathsf{return}\ q_2]\!]\phi = \xi^\dagger \circ \mathcal{Q}[\![q_2]\!] \circ \mathcal{C}[\![c]\!]\phi \circ (\mathcal{Q}[\![q_1]\!]\phi)^\dagger \circ \xi.$$

This definition should be read as follows: in the canonical state $\vec{o}$, the state described by the inverse interpretation of the input pattern q_1 is first prepared. Then, the body of the procedure is executed, yielding a new state that 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 $\vec{o}$. If this is the case, $\vec{o}$ can then be discarded, leaving only the output value.

4.7 Programs

Finally, programs are interpreted as the meaning of their topmost defined procedure and, thus, are interpreted as partial injective functions of signature

$$\mathbb{V} \xrightarrow{\mathcal{M}[\![m]\!]} \mathbb{V}.$$

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

$$\mathcal{M}[\![f_1 \cdots f_n]\!] = \kappa_1^\dagger \circ (\mu\phi.\mathcal{P}[\![f_1]\!]\phi \oplus \cdots \oplus \mathcal{P}[\![f_n]\!]\phi) \circ \kappa_1.$$

Note the inner interpretation of procedures $f_1 \cdots f_n$ as a disjoint union $\mathcal{P}[\![f_1]\!]\phi \oplus \cdots \oplus \mathcal{P}[\![f_n]\!]\phi$: This gives one large partial injective function, which behaves identically to the partial injective functions $\mathcal{P}[\![f_i]\!]\phi$ 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 behavior explains the need for injections κ_i and quasiprojections $\kappa_i^\dagger$ in the definition of procedure calls in Sect. 4.3.

The interpretation ($\mathcal{M}[\![\cdot]\!]$, $\mathcal{P}[\![\cdot]\!]\phi$, and $\mathcal{C}[\![\cdot]\!]\phi$) maps syntax to injective (value, store, ...) transformations (on stores, values). The injective (value, store, ...) transformations can be expressed in $\mathcal{L}$.

4.8 Use of Inverse Semantics

In conventional programming languages, programs are not guaranteed to be injective, program inversion requires a global analysis, and inverse interpretation requires additional overhead. However, owing to the formalization, programs in object languages formalized in $\mathcal{L}$ are always injective, program inversion can be achieved through a recursive descendent transformation, and inverse interpretation often has a constant time overhead only. The intrinsic properties of the metalanguage are extremely helpful in deriving rules for program inversion. For any command c, the *inverse semantics* $(\mathcal{C}[\![c]\!]\phi)^\dagger$ can be a composition of the semantics of its components and traces, which can be mechanically obtained from the properties of **PInj** [5]. For example, we have

$$(\mathcal{C}[\![q_1 \Leftarrow q_2]\!]\phi)^\dagger = ((\mathcal{Q}[\![q_1]\!]\phi)^\dagger \circ \mathcal{Q}[\![q_2]\!]\phi)^\dagger = (\mathcal{Q}[\![q_2]\!]\phi)^\dagger \circ \mathcal{Q}[\![q_1]\!]\phi,$$

for a reversible replacement and, hence, we obtain the inverse replacement

$$(\mathcal{C}[\![q_1 \Leftarrow q_2]\!]\phi)^\dagger = \mathcal{C}[\![q_2 \Leftarrow q_1]\!]\phi.$$

The right-hand sides of the semantic function of commands are mostly symmetric. Therefore, their inversion rules are obtained in a similar way. The only exception is the loop, which requires an additional identity $\mathrm{Tr}((f_1 \oplus \mathrm{id}) \oplus f_2) = \mathrm{Tr}(f_2 \oplus (\mathrm{id} \oplus f_1))$ in order to yield the inversion rule. A similar anti-symmetry appears in the operational semantics of the reversible language Janus [22], in which the inference rule for the loop can be either right or left recursive.

As regards the semantic function of patterns, the inverse semantics of the program context, $\phi^\dagger$, defines the meaning of a procedure uncall. The inverse semantics of procedures is equal to the semantics of inverted procedures. This yields an alternative formalization of the same meaning. First, the inverted procedures are added to the program context in addition to the original procedures:

$$\mu\phi.\ \mathcal{P}[\![f_1]\!]\phi \oplus \cdots \oplus \mathcal{P}[\![f_n]\!]\phi \oplus (\mathcal{P}[\![f_1]\!]\phi)^\dagger \oplus \cdots \oplus (\mathcal{P}[\![f_n]\!]\phi)^\dagger.$$

Given such an extended program context ϕ, the access to the inverse semantics $\kappa_i^\dagger \circ \phi^\dagger \circ \kappa_i$ in the pattern execution $\mathcal{Q}[\![\mathsf{uncall}\ f_i(q_1)]\!]\phi(\sigma)$ (Sect. 4.3) can be replaced by $\kappa_{i+n}^\dagger \circ \phi \circ \kappa_{i+n}$; *i.e.*, the $n+i$'th function is accessed.

5 Related Work

Formal meaning has been assigned to reversible programming languages using well-established formalisms, such as operational semantics to the imperative language Janus [22], the functional language RFUN [20], and the concurrent languages [15], small-step operational semantics to the assembler language PISA [1], transition functions to the flowchart language RFCL [21], and denotational semantics to `R-WHILE` [7,8]. However, the reversibility of a language is not directly expressed by these formalisms. It is the language designer's responsibility to guarantee the reversibility and to show the inversion properties of each language. Notably, the semantics of `R-WHILE` was first expressed irreversibly [22]. Note also that the type and effects systems have been studied for reversible languages [12].

In this paper, the reversible elements of `R-WHILE` were composed via the metalanguage $\mathcal{L}$ in a manner that preserved their reversibility. In previous work, compositional approaches to reversibility were applied in various guises, *e.g.*, in the diagrammatic composition of reversible circuits from reversible logic gates and reversible structured flowcharts from reversible control-flow operators [21]. Similarly, reversible Turing machines have been constructed from reversible rotary elements [16].

To give meaning to reversible languages by interpreters and translators is another operational approach to a semantics. Examples of similar applications include the realization of reversible interpreters [7,22], translation of the high-level language R to the reversible assembler language PISA [3], and mapping hardware descriptions in SyReC to reversible circuits [18]. A different approach involves the reversibilization of irreversible languages by extending the operational semantics via tracing, so as to undo program runs [10]. Alternatively, irreversible programs can be inverted by program inverters, *e.g.*, [6]. Reversible cellular automata may have non-injective local maps; However, if the local map is injective, the update by the global map is guaranteed to be reversible [14].

6 Conclusion

Reversible systems have reversible semantics. In this study, we built upon a semantic foundation intended to describe the semantics of reversible programming languages. Our approach was demonstrated through the full development of a formal semantics for the reversibly universal language `R-WHILE`. This allowed us to concisely formalize features representative of many reversible languages, including iteration, recursion, pattern matching, dynamic data structures, and access to a program's inverse semantics. The intrinsic properties of the metalanguage were essential for achieving formal reversible semantics. Hence, we argued that this approach provides a strong basis for understanding and reasoning about reversible programs.

Further exploration of the best description of advanced object-oriented structures, combinators, or features for concurrency, and the potentially useful metalanguage features, may be interesting. Some related challenges include the characterization of reversible heap allocation and concurrent reversible computations. However, further explanation of the practical feasibility of the metalanguage and its relationship to advanced reversible automata including nondeterminism, *e.g.*, [11], is necessary.

Acknowledgments. Support in the form of EU COST Action IC1405 is acknowledged. The third author is supported by JSPS KAKENHI Grant Number 18K11250.

References

1. Axelsen, H.B., Glück, R., Yokoyama, T.: Reversible machine code and its abstract processor architecture. In: Diekert, V., Volkov, M.V., Voronkov, A. (eds.) CSR 2007. LNCS, vol. 4649, pp. 56–69. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74510-5_9
2. Cockett, R., Lack, S.: Restriction categories III: colimits, partial limits and extensivity. Math. Struct. Comput. Sci. **17**(4), 775–817 (2007)
3. Frank, M.P.: Reversibility for efficient computing. Ph.D. thesis, MIT (1999)
4. Giles, B.: An investigation of some theoretical aspects of reversible computing. Ph.D. thesis, University of Calgary (2014)
5. Glück, R., Kaarsgaard, R.: A categorical foundation for structured reversible flowchart languages: Soundness and adequacy. Log. Methods Comput. Sci. **14**(3) (2018)
6. Glück, R., Kawabe, M.: Revisiting an automatic program inverter for Lisp. SIGPLAN Not. **40**(5), 8–17 (2005)
7. Glück, R., Yokoyama, T.: A minimalist's reversible while language. IEICE Trans. Inf. Syst. E100-D **100**(5), 1026–1034 (2017)
8. Glück, R., Yokoyama, T.: Constructing a binary tree from its traversals by reversible recursion and iteration. IPL **147**, 32–37 (2019)
9. Haghverdi, E.: A categorical approach to linear logic, geometry of proofs and full completeness. Ph.D. thesis, Carlton Univ. and Univ. Ottawa (2000)
10. Hoey, J., Ulidowski, I., Yuen, S.: Reversing parallel programs with blocks and procedures. In: Pérez, J.A., Tini, S. (eds.) Expressiveness in Concurrency, Structural Operational Semantics. Electronic Proceedings in TCS, vol. 276, pp. 69–86 (2018)
11. Holzer, M., Kutrib, M.: Reversible nondeterministic finite automata. In: Phillips, I., Rahaman, H. (eds.) RC 2017. LNCS, vol. 10301, pp. 35–51. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59936-6_3
12. James, R.P., Sabry, A.: Information effects. In: POPL, pp. 73–84. ACM (2012)
13. Kaarsgaard, R., Axelsen, H.B., Glück, R.: Join inverse categories and reversible recursion. J. Log. Algebr. Methods **87**, 33–50 (2017)
14. Kari, J.: Reversible cellular automata: from fundamental classical results to recent developments. New Gener. Comput. **36**(3), 145–172 (2018)
15. Kuhn, S., Ulidowski, I.: A calculus for local reversibility. In: Devitt, S., Lanese, I. (eds.) RC 2016. LNCS, vol. 9720, pp. 20–35. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40578-0_2
16. Morita, K.: Reversible computing and cellular automata – a survey. Theor. Comput. Sci. **395**(1), 101–131 (2008)
17. Selinger, P.: A survey of graphical languages for monoidal categories. In: Coecke, B. (ed.) New Structures for Physics. LNP, vol. 813, pp. 289–355. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-12821-9_4
18. Wille, R., Schönborn, E., Soeken, M., Drechsler, R.: SyReC: a hardware description language for the specification and synthesis of reversible circuits. Integration **53**, 39–53 (2016)
19. Winskel, G.: The Formal Semantics of Programming Languages: An Introduction. MIT Press, Cambridge (1993)
20. Yokoyama, T., Axelsen, H.B., Glück, R.: Towards a reversible functional language. In: De Vos, A., Wille, R. (eds.) RC 2011. LNCS, vol. 7165, pp. 14–29. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29517-1_2

21. Yokoyama, T., Axelsen, H.B., Glück, R.: Fundamentals of reversible flowchart languages. Theor. Comput. Sci. **611**, 87–115 (2016)
22. Yokoyama, T., Glück, R.: A reversible programming language and its invertible self-interpreter. In: PEPM, pp. 144–153. ACM (2007)