From Reversible Programming Languages to Reversible Metalanguages

Robert Glück[a], Robin Kaarsgaard[b,*], Tetsuo Yokoyama[c]

[a]*DIKU, Department of Computer Science, University of Copenhagen, Denmark*
[b]*School of Informatics, University of Edinburgh, United Kingdom*
[c]*Department of Software Engineering, Nanzan University, Nagoya, Japan*

Abstract

During the past decade reversible programming languages have been formalized using various established semantics frameworks. However, these semantics fail to effectively specify the distinct properties of reversible languages at the metalevel, even including the central question of whether the defined language is reversible. In this paper, we build a metalanguage foundation for reversible languages from categorical principles, 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. The use of the metalanguage leads to a formalization of the reversible semantics. A language defined in the metalanguage is guaranteed to have reversibility and inverse semantics. Also, program inverters for this language are obtained for free. We further discuss applications and directions for reversible semantics.

1. Introduction

Over the last ten years, 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.* [1, 2, 3, 4]). These frameworks, which have been used to provide meaning to a wide range of 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

*Corresponding author
Email addresses: `glueck@acm.org` (Robert Glück), `robin.kaarsgaard@ed.ac.uk` (Robin Kaarsgaard), `tyokoyama@acm.org` (Tetsuo Yokoyama)

questions about the uniqueness of inverse semantics and the inversion of programs in reversible languages, and in particular the central question of whether a language is reversible or not, cannot be answered immediately.

In this paper, we build 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 injectivity. More specifically, interpretations of syntax are composed of sequential composition, Cartesian product, disjoint union, function inversion, iteration, and recursion. For this, we make use of the categorical foundation developed elsewhere (*e.g.* [5, 6, 7]; see also the fully abstract denotational semantics of Janus in [8] based on partial injective functions). Our approach exploits the fact that reversible programs have reversible semantics: We regard a program as (compositionally) reversible iff each of its meaningful subprograms is partially invertible. This allows us to provide a clean reversible semantics to a reversible language.

We demonstrate the aforementioned idea by a step-by-step development of a full formal semantics of the reversible procedural language `R-WHILE` including iteration and recursion. The use of the metalanguage leads to a formalization of the reversible semantics. Any language defined in the metalanguage is guaranteed to have reversibility and inverse semantics. The reversibility of the language follows immediately from its semantics, and it can be seen from the signatures of semantic functions that the language is clean and without any hidden tracing. This metalanguage approach is independent of the specific details of the defined language and can be extended to other ways of composing semantic functions, provided their injectivity is preserved. Also, program inverters for this language are obtained for free.

`R-WHILE` with procedures is a reversible while-language with structured control-flow operators, and dynamic data structures [3, 4].[1] The 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.*, when using conventional denota-

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

tional semantics. In the case of operational semantics, which permits the description of arbitrary relations, it is unclear how to restrict such an inference system to a purely reversible one without requiring an explicit proof of reversibility. A future direction of research can be to investigate extensions of the metalanguage to capture other 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, and Section 3 describes the reversible language `R-WHILE` with procedures. In Section 4, the formal semantics of the language is developed step-by-step. Section 5 demonstrates the extension of the language and equational reasoning. Section 6 and Section 7 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.*, [2]) and formal semantics (*e.g.*, [9]).

Extensions: This paper is an extended version of a paper [10] presented at the workshop on Reversibility in Programming, Languages, and Automata (RPLA 2019). Aside from a number of corrections and extensions throughout the paper serving to make exposition more clear (in Section 2, Section 3, and Section 4), we also explicate how one can extend the syntax and semantics of a reversible language with a number of useful features (Section 5) and perform equational reasoning when we use the presented specialized metalanguage $\mathcal{L}$ (Section 4.8).

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.*, [11, 12, 13]). 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).

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 $\uparrow$ and write $f(x_0) = \uparrow$. 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 $X \xrightarrow{\mathrm{id}_X} X$ 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, 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 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 we do *not* have projections (such as $X \otimes Y \xrightarrow{\pi_1} X$ 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. For any set X, we shall call the bijection witnessing that 1 is the unit of the Cartesian product on the left by $X \xrightarrow{\lambda_X} 1 \otimes X$, and on the right by $X \xrightarrow{\rho_X} X \otimes 1$.

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 ($\mathsf{inl}\ \cdot$) 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, $X \oplus X' \xrightarrow{f \oplus g} Y \oplus Y'$, 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 $\kappa_1^\dagger(\mathsf{inr}\ y) = \uparrow$.

Finally, note the interactions between the Cartesian product and disjoint union: $X \otimes 0$ is empty for all sets X, and analogous to the behavior 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 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 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 $\mathbf{PInj}(X, Y) \to \mathbf{PInj}(X, Y)$ has a least fixed point (where $\mathbf{PInj}(X, Y)$ denotes the set of partial injective functions between sets X and Y). By its definition, the least fix point 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, as well as the trace discussed below), so any function involving only these operations is guaranteed to be continuous. To this end, it is shown in [6] that continuity in **PInj** amounts to preserving *joins* of partial functions (*i.e.*, unions of function graphs), and continuity of sequential composition, partial inversion, and traces is shown. For parallel composition using Cartesian products and disjoint sums, the join characterisation of continuity means that this reduces to the well-known properties of set-theoretic union of $(X \cup X') \otimes Y = (X \otimes Y) \cup (X' \otimes Y)$ and $(X \cup X') \oplus Y = (X \oplus Y) \cup (X' \oplus Y)$.

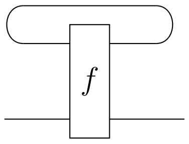
Finally, partial injective functions can also be tail-recursively described using the *trace* operator. Intuitively, the trace of a partial injective function $X \oplus U \xrightarrow{f} Y \oplus U$ is a function $X \xrightarrow{\mathrm{Tr}(f)} Y$ given as follows: If $f(\mathsf{inl}\ x) = \mathsf{inl}\ y$ 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 $\mathbf{PInj}(U \oplus X, U \oplus Y) \xrightarrow{\mathrm{Tr}} \mathbf{PInj}(X, Y)$. It is most easily defined using a tail-recursively described pretrace $\mathbf{PInj}(U \oplus X, U \oplus Y) \xrightarrow{pretrace} \mathbf{PInj}(U \oplus X, Y)$ given as follows:

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

With this, it is defined simply as $\mathrm{Tr}(f)(x) = pretrace(f)(\mathsf{inr}\ x)$.

The data flow of $\mathrm{Tr}(f)$ is typically illustrated using the diagram



in which the flow is from left to right and the feedback loop represents the repeated application of f to outputs of type U.

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 $U \oplus X \xrightarrow{f} U \oplus Y$. (Formally, the trace operator can also be defined as a fixed point using the *trace formula*, see [14].)

2.3. Summary of the Metalanguage

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

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

For any expression in $\mathcal{L}$, the least fixed point exists. The formal argument of the least fixed point ϕ is a partial injective function. $\mathcal{L}$ 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 expressive enough to fully formalize the semantics of reversibly-universal languages, as demonstrated below for `R-WHILE`.

An atomic function a can be any auxiliary partial injective function. In what follows, we use the following four injective helper functions: i) $X \xrightarrow{\rho_X} X \otimes 1$, defined above, ii) $1 \xrightarrow{const_v} \mathbb{V}$ defined by $const_v(\star) = v$ where $\star$ is

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

Figure 1: Syntax of the reversible language `R-WHILE` with procedures.

the unique element in 1 and v is any value, and iii) $(X \otimes Y) \otimes Z \xrightarrow{assocr_\otimes} X \otimes (Y \otimes Z)$ defined by $assocr_\otimes((x, y), z) = (x, (y, z))$. By abuse of notation, $X_i \xrightarrow{\kappa_i} X_1 \oplus (X_2 \oplus (\cdots (X_i \oplus \cdots (X_{n-1} \oplus X_n) \cdots))$ returns the nested injection:

$$
\kappa_i(x) = \begin{cases} \underbrace{\mathsf{inr}(\cdots(\mathsf{inr}}_{i-1}(\mathsf{inl}\ x))\cdots) & \text{if } 1 \le i < n \\ \underbrace{\mathsf{inr}(\cdots(\mathsf{inr}}_{n-1}\ x)\cdots) & \text{if } i = n \end{cases}
$$

3. R-WHILE with Reversible Recursion and Iteration

We describe the semantics of the reversible language `R-WHILE` with procedures informally, and illustrate it with a recursive program that translates infix expressions to Polish notation, 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 [4] is shown in Fig. 1. A *program* m is a sequence of procedures $p \cdots p$, where the first 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. Each procedure has a single argument and a single return value. Thus, it is convenient to compose and decompose input and output values by patterns. Figure 2 shows example procedures.

A *command* c is either 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 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). The assignment sets x to the value of e if the value of x is $\underline{nil}$ and sets x to $\underline{nil}$

if the values of x and e are equal; otherwise, the assignment is undefined. In other words, x is exclusively set or reset depending on its value. The assignment is only defined for these two cases. This definition ensures the reversibility of assignments. The hat $\widehat{\ }$ of assignment operator $\widehat{=}$ is reminiscent of an exclusive-or operator. For example, if x is $\underline{nil}$ then $x \mathrel{\widehat{=}} y$ sets x to the value of y. If x and y are equal then $x \mathrel{\widehat{=}} y$ resets x to $\underline{nil}$. In the first case, the value of y is duplicated; in the second case, the value of x is nil-cleared using the value of y. The variables that occur in the expression on the right side, here y, are never changed, only x is *updated reversibly* [15].

A reversible replacement $q_1 \Leftarrow q_2$ arranges values according to patterns q_1 and q_2. For example, $(y.x) \Leftarrow (x.y)$ swaps the values of variables x and y. In contrast to an assignment, no value can be duplicated by a reversible replacement. Before the value constructed by q_2 on the right side is *matched* with q_1 on the left side, all variables in q_2 are nil-cleared. This means that the same variables 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 in the language. For example, replacement $(t.y) \Leftarrow y$ decomposes the value of y on the right side by the pattern $(t.y)$ on the left side into head t and tail y provided the original value is indeed a pair and the value of t is initially $\underline{nil}$. Another example is $y \Leftarrow (t.y)$ that pairs the values of t and y by the pattern $(t.y)$ on the right side, nil-clears t and y, and binds the new pair to y on the left side. It is easy to see that sequence $y \Leftarrow (t.y)$; $(t.y) \Leftarrow y$ restores the original values of t and y.

A *pattern* q is a variable x, a symbol $\underline{s}$, a pair of patterns $(q.q)$, or an invocation or inverse invocation of a procedure by call $f(q)$ or uncall $f(q)$. All 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, not in expressions. This will be formalized later. Examples can be seen in Fig. 2.

The conditional if...fi and the loop from...until are two control structures which are also found in other reversible flowchart languages (*e.g.*, [2]), and they work as follows: Compared to conventional control structures, they are equipped with assertions. The exit of a conditional, fi e, is an assertion that must evaluate to true after the then-branch and to false after the else-branch. The entry of a loop, from e, is an assertion that must evaluate to true before entering the loop and to false after each iteration. The control structures are undefined if their assertions do not evaluate as required.

Expressions are conventional. An *expression* e is either a variable x, a symbol $\underline{s}$, or the application of an operator, *i.e.*, constructor cons $(\cdot\,.\,\cdot)$, selectors head hd and tail tl, or equality test =?. An expression defines a

```
proc $in2prefix(t)$                    (* infix exp to Polish notation *)
  $y \Leftarrow$ call $pre((t.\underline{nil}))$;    (* call preorder traversal       *)
  return $y$;

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

proc $pre((t.y))$                      (* recursive preorder traversal *)
  if $=?\ t\ \underline{0}$ 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$(y)\ \underline{0}$;               (* head of list $y$ is a leaf?    *)
  return $y$;
```

Figure 2: Translation between infix expressions and Polish notation in `R-WHILE`.

partial function that is not necessarily injective (*e.g.*, hd, tl are not injective). Other operators can easily be added to expressions.

Variables in a program are denoted by small letters, such as l, d, r, and symbols are underlined, such as $\underline{nil}$, $\underline{0}$, $\underline{1}$.

Example 1. The translation of infix expressions into Polish notation, and vice versa, has many practical applications. 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

$$tree \quad ::= \quad \underline{0} \mid (tree\,.\,(\underline{1}\,.\,tree)) \enspace ,$$

where symbols $\underline{0}$ and $\underline{1}$ stand for an operand (leaf) and a binary operator (inner label) in an expression, respectively. To keep it simple, we will only use these two symbols in expressions. The corresponding expressions in Polish notation can be represented by proper lists

$$list \quad ::= \quad \underline{nil} \mid (\underline{0}\,.\,list) \mid (\underline{1}\,.\,list) \enspace ,$$

where $\underline{nil}$ stands for the empty list.

Figure 2 shows the recursive procedure *pre* that reversibly translates an infix expression into a prefix expression (Polish notation) by a preorder traversal of the full binary tree t. Procedure *pre* is called and uncalled in the two procedures *in2prefix* and *pre2infix* to translate to Polish notation, and vice versa. For example, the infix expression

$$t = ((\underline{\mathit{0}} \,.\, (\underline{\mathit{1}} \,.\, \underline{\mathit{0}})) \,.\, (\underline{\mathit{1}} \,.\, \underline{\mathit{0}}))$$

translates to Polish notation

$$y = (\underline{\mathit{1}} \,.\, (\underline{\mathit{1}} \,.\, (\underline{\mathit{0}} \,.\, (\underline{\mathit{0}} \,.\, (\underline{\mathit{0}} \,.\, \underline{\mathit{nil}}))))) \ .$$

Using syntactic sugar, this can be written as the translation of an infix expression $((\underline{\mathit{0}}\ \underline{\mathit{1}}\ \underline{\mathit{0}})\ \underline{\mathit{1}}\ \underline{\mathit{0}})$ to Polish notation $(\underline{\mathit{1}}\ \underline{\mathit{1}}\ \underline{\mathit{0}}\ \underline{\mathit{0}}\ \underline{\mathit{0}})$.

In *in2prefix* the translation is invoked by a call to *pre* (line 2)

$$y \Leftarrow \textsf{call}\ \mathit{pre}((t.\underline{\mathit{nil}}))\ ,$$

where the argument of the call is a pair $(t.\underline{\mathit{nil}})$ of t and the empty list $\underline{\mathit{nil}}$, and the result is matched with trivial pattern y, which binds it to y.

In *pre2infix* the inverse translation of *pre* is invoked by an uncall of *pre* (line 6)

$$(t.\underline{\mathit{nil}}) \Leftarrow \textsf{uncall}\ \mathit{pre}(y)\ ,$$

where y is the argument of the uncall and t is picked from the resulting pair. Given y, uncall *pre* computes the infix expression t. The uncall invokes the inverse computation of the recursive traversal implemented by *pre*.

The body of *pre* is a reversible conditional if...fi (lines 10–17) with the entry test $(=?\ t\ \underline{\mathit{0}})$ and the exit assertion $(=?\ \textsf{hd}(y)\ \underline{\mathit{0}})$. If t is a leaf $\underline{\mathit{0}}$ then 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 the label d are selected from t by $(l.(d.r)) \Leftarrow t$. List y is built from right to left, so d is added to y after both subtrees are translated (line 16). The exit assertion $(=?\ \textsf{hd}(y)\ \underline{\mathit{0}})$ is always true after the then-branch and always false after the else-branch provided t is a correct infix expression.

The arity of all procedures is one, so it is convenient to decompose the argument value by pattern $(t.y)$ already in the head of *pre* (line 9). Due to the reversibility of the language, *pre* performs a deterministic traversal in both directions when invoked by call and uncall (lines 2, 6). Inverting the traversal of binary trees, not only of full binary trees, is a classic programming problem for which related reversible solutions were given [4]. The reversible semantics that follows in the next section underpins all these solutions.

4. An Intrinsically Reversible Semantics

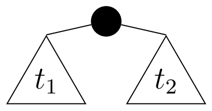
In this section, we illustrate the principle of reversible semantics by constructing a denotational semantics for `R-WHILE` with procedures using sets and partial injective functions. This is done first by constructing the domains of computation, and then constructing an interpretation of each syntactic construct, proceeding by 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.

In the following, we use standard notations of denotational semantics [9] including brackets for semantic functions $[\![\cdot]\!]$, mapping syntactic constructs to semantic ones. Note, in particular, the use of semantic functions *parameterized* by a state or program context – see, *e.g.*, patterns in Section 4.3). Such an interpretation should not be regarded as a single function, but rather as a *family* of functions indexed by the appropriate parameter (*e.g.*, a state or program context).

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 semantic representations of *symbols*, elements of which we denote using an overline, *e.g.*, $\overline{\mathit{0}}$, $\overline{\mathit{nil}}$, etc. Note, in particular, the distinction between *semantic* symbols (*e.g.*, $\overline{\mathit{0}}$, $\overline{\mathit{nil}}$) and their *syntactic representations* (*e.g.*, $\underline{\mathit{0}}$, $\underline{\mathit{nil}}$). As is usual, we will assume that these two sets are distinct but in bijective correspondence.

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, *i.e.*,



A state associates each variable with a value. The set of states Σ can be constructed as *colists* of values (*i.e.*, lists of infinite length), that is, $\Sigma = \mathbb{V} \otimes \mathbb{V} \otimes \cdots$ (explicitly, this is constructed as the greatest fixed point $\nu X.I \oplus (\mathbb{V} \otimes X)$ where I is identity). Owing to this intuition, we will write states as infinite vectors of values, *e.g.*, $(v_1, v_2, \dots)$. Note that colists arising from computational states will all be finitely supported, *i.e.*, they will only contain finitely many non-$\overline{nil}$ values. 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 shall write variables as x_1, x_2, x_3, etc. rather than (as is usual) as x, y, z, etc.

4.2. Expressions

In irreversible languages, expressions are usually interpreted as partial functions of the signature $\Sigma \to \mathbb{V}$. Because multiple states result in the same value, the 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 the signature:

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

Regardless of their concrete form, interpretations of expressions are 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 as the value of e in the state σ. Precisely, a family of the infinite number of the semantic functions $\mathcal{E}'[\![\cdot]\!]$ is indexed by the subscript 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. In other words, when a state σ is fixed, $\mathcal{E}[\![e]\!](\sigma, v)$ is only ever defined on two choices of v, namely $v = \overline{nil}$ and $v = \mathcal{E}'[\![e_1]\!]_\sigma$. In these two cases, it sends $(\sigma, \overline{nil})$ to $(\sigma, \mathcal{E}'[\![e_1]\!]_\sigma)$ and vice versa; thus, it is injective. Another way to say this is that the semantics function defines a *reversible update* [15] of the value argument with the state σ kept unchanged, which also implies that it is self-inverse. This is the general principle hidden here.

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

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

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 (resp., tail) of an expression takes the head (resp. tail) of its meaning, diverging if not of this form. The meaning of equality check $=?$ returns a distinct value depending on e_1 and e_2 having the same value. The definition uses the convention that $\overline{nil}$ is considered to be *false*, and any other value to be *true*.

It is obvious that more operators can be added to this list. Note that an element $v \in \mathbb{V}$ is uniquely associated with a (necessarily injective) function $1 \to \mathbb{V}$, which maps the unique element of 1 to v. This justifies the fact that the interpretation of an expression may diverge (as is the case for, *e.g.*, $\mathcal{E}'[\![\mathsf{hd}(e_1)]\!]_\sigma$).

The use of a non-injective function in the definition of an injective function is often found 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

Since patterns may include procedure invocation, the meaning of a pattern depends on the program context ϕ in which it is interpreted. The program context ϕ is the disjoint union of the meaning of all procedures f_i in the program: $\phi = f_1 \oplus (f_2 \oplus (\cdots (f_{n-1} \oplus f_n)) \cdots)$. Program contexts are constructed at the level of programs as we shall see in Section 4.7. Patterns in a program context are all interpreted as partial injective functions with the signature

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

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. To more easily define the interpretation of patterns, we use injective helper functions: One is $\Sigma \xrightarrow{extract_i} \Sigma \otimes \mathbb{V}$ given by $extract_i(v_1, \dots, v_{i-1}, v_i, \dots) = ((v_1, \dots, v_{i-1}, \overline{nil}, \dots), v_i)$ (*extracting* the i'th value from the state) and $V \otimes V \xrightarrow{cons} V$ given by $cons(v_1, v_2) = v_1 \bullet v_2$. The interpretation of patterns is then defined as follows, depending on the form of q:

$$\begin{aligned}
\mathcal{Q}[\![x_j]\!]_\phi &= extract_j \\
\mathcal{Q}[\![\underline{s}]\!]_\phi &= (\mathrm{id}_\Sigma \otimes const_{\overline{s}}) \circ \rho_\Sigma \\
\mathcal{Q}[\![\mathsf{call}\ f_i(q_1)]\!]_\phi &= (\mathrm{id}_\Sigma \otimes (\kappa_i^\dagger \circ \phi \circ \kappa_i)) \circ \mathcal{Q}[\![q_1]\!]_\phi \\
\mathcal{Q}[\![\mathsf{uncall}\ f_i(q_1)]\!]_\phi &= (\mathrm{id}_\Sigma \otimes (\kappa_i^\dagger \circ \phi^\dagger \circ \kappa_i)) \circ \mathcal{Q}[\![q_1]\!]_\phi \\
\mathcal{Q}[\![(q_1.q_2)]\!]_\phi &= (\mathrm{id}_\Sigma \otimes cons) \circ assocr_\otimes \circ (\mathcal{Q}[\![q_2]\!]_\phi \otimes \mathrm{id}_\mathbb{V}) \circ \mathcal{Q}[\![q_1]\!]_\phi
\end{aligned}$$

The meaning of a variable, as a pattern, is to simultaneously extract its contents *and* clear it, as handled by the $extract_i$ function. The meaning of a symbol is given by the bijection witnessing on the right ρ_Σ, which yields a new state and the unique element $\star$, and then the $const_{\overline{s}}$ function, which takes the latter $\star$ and yields the value $\underline{s}$. A procedure call $\mathsf{call}\ f_i(q_1)$ is interpreted as passing the value of q_1 to the i'th component of the program context ϕ and extracting from the i'th component afterwards, which, as we will see in Section 4.8, corresponds precisely to invoking the i'th procedure. Uncall to a procedure is handled analogously, but using the *inverse* to the program context $\phi^\dagger$ instead. Finally, the meaning of a cons pattern $(q_1.q_2)$ is as a kind of sequential composition. First, the interpretation of q_2 results in a new state σ' and value v_2. Then, the interpretation of q_1 results in this new state σ' with v_2 kept unchanged by $\mathrm{id}_\mathbb{V}$, yielding a final state σ'' and value v_1. The injective helper function $assocr_\otimes$ defined in Section 2.3 rearranges the nested pair $((\sigma'', v_1), v_2)$ to $(\sigma'', (v_1, v_2))$. The two values v_1 and v_2 are then consed together using the injective helper function *cons*, finally resulting in the state σ'' and prepared value $v_1 \bullet v_2$. Recall that no variable occurs more than once in a pattern. Hence, it does not matter whether a cons pattern $(q_1.q_2)$ is interpreted from left to right, or vice versa.

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. We will see how to add the inverse procedures to ϕ in Section 4.8.

4.4. Predicates

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

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

We use the convention that $\overline{nil}$ is *false* and the other values are *true*. The predicate interpretation of an expression e is defined as follows:

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

As such, the predicate interpretation of e_1 sends control flow to the first component if e_1 is considered false in the given state, and to the second component if it is considered true. As we will see in Section 4.5, this style makes for a straightforward interpretation of *conditional execution* of commands (see also [5, 16]).

4.5. Commands

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

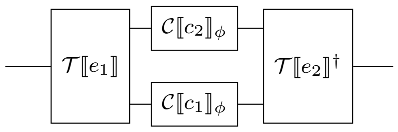
$$\Sigma \xrightarrow{\mathcal{C}[\![c]\!]_\phi} \Sigma \ .$$

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

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

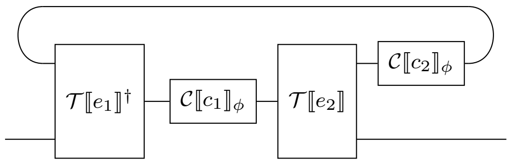
Note, in particular, the use of *inverses* to patterns and predicates in the definition above. The inverse to a predicate corresponds to its corresponding *assertion*, whereas the inverse to a pattern performs *state preparation* consuming (part of) a value (rather than, in the forward direction, *value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



$\mathcal{T}[\![e_1]\!]$
$\mathcal{C}[\![c_2]\!]_\phi$
$\mathcal{C}[\![c_1]\!]_\phi$
$\mathcal{T}[\![e_2]\!]^\dagger$

where the upper $\mathcal{C}[\![c_2]\!]_\phi$ and the lower $\mathcal{C}[\![c_1]\!]_\phi$ correspond to the meaning of else and then branches, respectively, and loops, illustrated with the diagram



$\mathcal{T}[\![e_1]\!]^\dagger$
$\mathcal{C}[\![c_1]\!]_\phi$
$\mathcal{T}[\![e_2]\!]$
$\mathcal{C}[\![c_2]\!]_\phi$

relies on predicate inverses: In both cases, they serve as conditional join points, corresponding to an assertion that e_2 (respectively e_1) 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 for the sake of simplicity `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

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

To define the interpretation of procedures, we use 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

$$\mathcal{P}[\![\mathsf{proc}\ f(q_1)\ c_1; \mathsf{return}\ q_2]\!]_\phi = \xi^\dagger \circ \mathcal{Q}[\![q_2]\!]_\phi \circ \mathcal{C}[\![c_1]\!]_\phi \circ (\mathcal{Q}[\![q_1]\!]_\phi)^\dagger \circ \xi \ .$$

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 $\vec{o}$, 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

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

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

$$\mathcal{M}[\![p_1 \cdots p_n]\!] = \kappa_1^\dagger \circ (\mu\phi.\mathcal{P}[\![p_1]\!]_\phi \oplus \cdots \oplus \mathcal{P}[\![p_n]\!]_\phi) \circ \kappa_1 \enspace .$$

Note the inner interpretation of procedures $p_1 \cdots p_n$ as a disjoint union $\mathcal{P}[\![p_1]\!]_\phi \oplus \cdots \oplus \mathcal{P}[\![p_n]\!]_\phi$: This gives one large partial injective function, which behaves just the partial injective functions $\mathcal{P}[\![p_i]\!]_\phi$ 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 explains the need for injections κ_i and quasiprojections $\kappa_i^\dagger$ in the definition of procedure calls in Section 4.3.

The interpretations $\mathcal{E}'[\![\cdot]\!]$ and $\mathcal{T}[\![\cdot]\!]$ are atomic functions in the metalanguage $\mathcal{L}$. The interpretation ($\mathcal{M}[\![\cdot]\!], \mathcal{P}[\![\cdot]\!]_\phi$, and $\mathcal{C}[\![\cdot]\!]_\phi$) maps syntax to injective (value, stores, ...) transformations (on stores, values). The injective (value, store, ...) transformations can be expressed in $\mathcal{L}$.

4.8. Applications of the Semantics

In conventional programming languages, programs are not guaranteed to be injective, program inversion usually needs a global program analysis, and inverse interpretation requires extra computation overhead. However, owing to the formalization, programs in object languages formalized in $\mathcal{L}$ are always injective, program inversion can be obtained by a recursive descendent transformation, and inverse interpretation often has no extra overhead. The intrinsic properties of the metalanguage are a great help 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

$$\begin{aligned}
\mathcal{I}[\![x \mathrel{\widehat{=}} e]\!] &\equiv x \mathrel{\widehat{=}} e \\
\mathcal{I}[\![q_1 \Leftarrow q_2]\!] &\equiv q_2 \Leftarrow q_1 \\
\mathcal{I}[\![c_1; c_2]\!] &\equiv \mathcal{I}[\![c_2]\!];\ \mathcal{I}[\![c_1]\!] \\
\mathcal{I}[\![\mathsf{if}\ e_1\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}\ e_2]\!] &\equiv \mathsf{if}\ e_2\ \mathsf{then}\ \mathcal{I}[\![c_1]\!]\ \mathsf{else}\ \mathcal{I}[\![c_2]\!]\ \mathsf{fi}\ e_1 \\
\mathcal{I}[\![\mathsf{from}\ e_1\ \mathsf{do}\ c_1\ \mathsf{loop}\ c_2\ \mathsf{until}\ e_2]\!] &\equiv \mathsf{from}\ e_2\ \mathsf{do}\ \mathcal{I}[\![c_1]\!]\ \mathsf{loop}\ \mathcal{I}[\![c_2]\!]\ \mathsf{until}\ e_1 \\[1ex]
\mathcal{I}_p[\![\mathsf{proc}\ f_i(q_1)\ c_1; \mathsf{return}\ q_2;]\!] &\equiv \mathsf{proc}\ f_i^{inv}(q_2)\ \mathcal{R}[\![\mathcal{I}[\![c_1]\!]]\!]; \mathsf{return}\ q_1; \\[1ex]
\mathcal{I}_m[\![p_1\ \cdots\ p_n]\!] &\equiv \mathcal{I}_p[\![p_1]\!]\ \cdots\ \mathcal{I}_p[\![p_n]\!]
\end{aligned}$$

Figure 3: A command inverter $\mathcal{I}$, a procedure inverter $\mathcal{I}_p$, and a program inverter $\mathcal{I}_m$ for `R-WHILE` with procedures. $\mathcal{R}$ renames each called/uncalled procedure f_i to its inverse f_i^{inv} without changing its meaning.

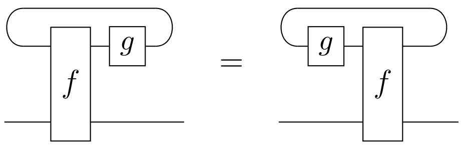
be mechanically obtained by properties of **PInj** [5]. For example, we have for a reversible replacement

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

and hence we obtain that the inverse semantics of replacement is

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

The right-hand sides of the semantic function of commands are mostly symmetric, and their inversion rules are obtained in a similar way (Fig. 3). The only inversion rule that requires some work to derive is the one for loops, which relies on the *dinaturality law* (see [17]) of traces, $\mathrm{Tr}((\mathrm{id} \oplus g) \circ f) = \mathrm{Tr}(f \circ (\mathrm{id} \oplus g))$, or graphically



A similar asymmetry appears in the operational semantics of Janus [1], in which the inference rule for the loop can be either right or left recursive.

In the semantic function of patterns, the inverse semantics $\phi^\dagger$ of the program context defines the meaning of a procedure uncall. The inverse semantics of procedures is equal to the semantics of inverted procedures. This leads to 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 $\mathcal{Q}[\![\mathsf{uncall}\ f_i(q_1)]\!]_\phi(\sigma)$ (Section 4.3) can be replaced by $\kappa_{n+i}^\dagger \circ \phi \circ \kappa_{n+i}$, *i.e.*, accessing the $n+i$'th function.

Program Equivalences. Another application of the semantics, perhaps more familiar to readers with a background in classical programming languages, is for deriving semantic equivalences of program fragments. One such equivalence is the equivalence of the replacement commands

$$q_2 \Leftarrow \mathsf{call}\ f_i(q_1) \ \equiv\ \mathsf{uncall}\ f_i(q_2) \Leftarrow q_1$$

for all patterns q_1, q_2 and any procedure f_i, as it holds for any program context ϕ that

$$\begin{aligned}
\mathcal{C}[\![q_2 \Leftarrow \mathsf{call}\ f_i(q_1)]\!]_\phi &= (\mathcal{Q}[\![q_2]\!]_\phi)^\dagger \circ \mathcal{Q}[\![\mathsf{call}\ f_i(q_1)]\!]_\phi \\
&= (\mathcal{Q}[\![q_2]\!]_\phi)^\dagger \circ (\mathrm{id}_\Sigma \oplus (\kappa_i^\dagger \circ \phi \circ \kappa_i)) \circ \mathcal{Q}[\![q_1]\!]_\phi \\
&= (\mathcal{Q}[\![q_2]\!]_\phi)^\dagger \circ (\mathrm{id}_\Sigma \oplus (\kappa_i^\dagger \circ \phi \circ \kappa_i))^{\dagger\dagger} \circ \mathcal{Q}[\![q_1]\!]_\phi \\
&= (\mathcal{Q}[\![q_2]\!]_\phi)^\dagger \circ (\mathrm{id}_\Sigma \oplus (\kappa_i^\dagger \circ \phi^\dagger \circ \kappa_i))^\dagger \circ \mathcal{Q}[\![q_1]\!]_\phi \\
&= ((\mathrm{id}_\Sigma \oplus (\kappa_i^\dagger \circ \phi^\dagger \circ \kappa_i)) \circ \mathcal{Q}[\![q_2]\!]_\phi)^\dagger \circ \mathcal{Q}[\![q_1]\!]_\phi \\
&= (\mathcal{Q}[\![\mathsf{uncall}\ f_i(q_2)]\!]_\phi)^\dagger \circ \mathcal{Q}[\![q_1]\!]_\phi \\
&= \mathcal{C}[\![\mathsf{uncall}\ f_i(q_2) \Leftarrow q_1]\!]_\phi \ .
\end{aligned}$$

Similarly, we have the equivalence of the replacement commands

$$q_2 \Leftarrow \mathsf{uncall}\ f_i(q_1) \ \equiv\ \mathsf{call}\ f_i(q_2) \Leftarrow q_1 \ .$$

By applying these equivalences, we see in action the familiar idea that uncalling nested procedure calls happens in reverse procedure call order, *i.e.*,

$$\begin{aligned}
& q_2 \Leftarrow \mathsf{call}\ g(\mathsf{call}\ f(q_1)) \\
\equiv\quad & \mathsf{uncall}\ g(q_2) \Leftarrow \mathsf{call}\ f(q_1) \\
\equiv\quad & \mathsf{uncall}\ f(\mathsf{uncall}\ g(q_2)) \Leftarrow q_1 \ .
\end{aligned}$$

and we obtain various other equivalences for replacements, such as

$$\begin{aligned}
& q_2 \Leftarrow \mathsf{uncall}\ g(\mathsf{call}\ f(q_1)) \\
\equiv\quad & \mathsf{call}\ g(q_2) \Leftarrow \mathsf{call}\ f(q_1) \\
\equiv\quad & \mathsf{uncall}\ f(\mathsf{call}\ g(q_2)) \Leftarrow q_1 \ .
\end{aligned}$$

It is easy to see the general principle that calls correspond to uncalls when they change sides in a replacement and that the order of nested calls and uncalls is reversed. This shows that the semantics is useful for deriving new equivalences that are not known in irreversible programming languages. Even further, since `R-WHILE` is a relatively simple first-order language, given a suitable operational semantics, we expect soundness, computational adequacy, and full abstraction to all be provable (see analogous results in [5, 8]). If this does turn out to be the case, the denotational semantics could also be used to derive contextual equivalences.

5. Extensions to `R-WHILE` and Their Semantics

In this section, we consider some syntactic extensions to `R-WHILE`, and show how to give semantics to these using the presented metalanguage. While they are presented for `R-WHILE`, it should be stressed that these are general reversible language constructs, and ought to fit the vast majority of imperative reversible programming languages.

5.1. Skip, Simple Loops, and Conditional Execution

Arguably the simplest extension to `R-WHILE` we can consider is the skip command, which does absolutely nothing—it behaves as the identity. Introducing this requires extending the syntax of commands:

$$c \quad ::= \quad \cdots \mid \mathsf{skip} \enspace .$$

Though it may not be obvious, skip can be considered an alias for $\underline{s} \Leftarrow \underline{s}$ for any choice of symbol $\underline{s}$. We then have

$$\mathcal{C}[\![\mathsf{skip}]\!]_\phi = \mathcal{C}[\![\underline{s} \Leftarrow \underline{s}]\!]_\phi = (\mathcal{Q}[\![\underline{s}]\!]_\phi)^\dagger \circ \mathcal{Q}[\![\underline{s}]\!]_\phi$$

and to support our claim, it suffices to show that the last term is equal to the identity:

$$\begin{aligned} & ((\mathrm{id}_\Sigma \otimes \mathit{const}_{\overline{s}}) \circ \rho_\Sigma)^\dagger \circ (\mathrm{id}_\Sigma \otimes \mathit{const}_{\overline{s}}) \circ \rho_\Sigma \\ = \; & \rho_\Sigma^\dagger \circ (\mathrm{id}_\Sigma^\dagger \otimes \mathit{const}_{\overline{s}}^\dagger) \circ (\mathrm{id}_\Sigma \otimes \mathit{const}_{\overline{s}}) \circ \rho_\Sigma \\ = \; & \rho_\Sigma^\dagger \circ ((\mathrm{id}_\Sigma^\dagger \circ \mathrm{id}_\Sigma) \otimes (\mathit{const}_{\overline{s}}^\dagger \circ \mathit{const}_{\overline{s}})) \circ \rho_\Sigma \\ = \; & \rho_\Sigma^\dagger \circ \rho_\Sigma \\ = \; & \mathrm{id}_\Sigma \enspace . \end{aligned}$$

Since both $\mathit{const}_{\overline{s}}$ and id_Σ are total, it follows that $(\mathcal{Q}[\![\underline{s}]\!]_\phi)^\dagger \circ \mathcal{Q}[\![\underline{s}]\!]_\phi = \mathrm{id}_\Sigma$, and hence $\mathcal{C}[\![\mathsf{skip}]\!]_\phi = \mathrm{id}_\Sigma$, as expected.

Using skip as the empty command of R-WHILE allows us to derive some simpler versions of the flow control constructs presented in the previous sections. For example, the one-armed conditional (similar to *control* in reversible circuit logic) can now be expressed as syntactic sugar for the conditional,

$$\mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{fi}\ e' \quad\equiv\quad \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else\ skip\ fi}\ e' \ ,$$

and the simpler *while*-loop [3, 5] can be seen as syntactic sugar for the loop,

$$\mathsf{from}\ e\ \mathsf{loop}\ c\ \mathsf{until}\ e' \quad\equiv\quad \mathsf{from}\ e\ \mathsf{do\ skip\ loop}\ c\ \mathsf{until}\ e' \ .$$

The abort command cannot even “do nothing” such as the skip command, it always fails to reach a next state. Adding such a construct to R-WHILE requires extending the syntax of commands:

$$c ::= \cdots \ \mid \mathsf{abort} \ .$$

The command can be considered an alias for an impossible match $\underline{s} \Leftarrow \underline{s'}$ where $s \neq s'$ (*e.g.*, let abort be an alias for $\underline{\mathit{0}} \Leftarrow \underline{\mathit{1}}$). Hence, we have the interpretation $\mathcal{C}[\![\mathsf{abort}]\!]_\phi(\sigma) = \uparrow$ for any store σ, as expected.

5.2. Multiconditionals

Another simple extension to R-WHILE is the introduction of reversible *multiconditionals* or *case*-constructs: Where an ordinary (*i.e.*, binary) reversible conditional $\mathsf{if}\ e_1\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}\ e_1'$ has two branches and two predicates to distinguish them, an n-armed reversible multiconditional has n branches and $2n$ predicates to distinguish them, extending the syntax of commands:

$$c ::= \cdots \ \mid \mathsf{case}\ e_1 : c_1 : e_1'; \cdots ; e_n : c_n : e_n'\ \mathsf{esac} \ .$$

With this syntax in place, the interpretation of an n-armed reversible multiconditional can be specified by extending the semantics function for commands using the recursive definition

$$\mathcal{C}[\![\mathsf{case}\ e_1 : c_1 : e_1'; \cdots ; e_n : c_n : e_n'\ \mathsf{esac}]\!]_\phi = \begin{cases} \mathcal{T}[\![e_1']\!]^\dagger \circ (\mathcal{C}[\![c_1]\!]_\phi \oplus \mathcal{C}[\![\mathsf{case}\ e_2 : c_2 : e_2'; \cdots ; e_n : c_n : e_n'\ \mathsf{esac}]\!]_\phi) \circ \mathcal{T}[\![e_1]\!] & \text{if } n \geq 1 \\ \mathcal{C}[\![\mathsf{abort}]\!]_\phi & \text{if } n = 0 \,. \end{cases}$$

The reader may note that multiconditionals have the same semantics as simply nested conditionals

$$\mathsf{if}\ e_1\ \mathsf{then}\ c_1\ \mathsf{else}\ (\mathsf{if}\ e_2\ \mathsf{then}\ c_2\ \mathsf{else}\ (\cdots (\mathsf{if}\ e_n\ \mathsf{then}\ c_n\ \mathsf{else\ abort\ fi}\ e_n') \cdots)\ \mathsf{fi}\ e_2')\ \mathsf{fi}\ e_1' \ .$$

That the semantics function and the interpretation of the nested conditionals are equivalent can be seen by the following derivation. When $n \geq 1$ we have the derivation

$$\begin{aligned}
&\mathcal{C}[\![\mathsf{case}\ e_1 : c_1 : e'_1;\ \cdots\ ;\ e_n : c_n : e'_n\ \mathsf{esac}]\!]_\phi \\
&= \mathcal{C}[\![\mathsf{if}\ e_1\ \mathsf{then}\ c_1\ \mathsf{else}\ (\mathsf{if}\ e_2\ \mathsf{then}\ c_2\ \mathsf{else}\ (\cdots \\
&\qquad\qquad (\mathsf{if}\ e_n\ \mathsf{then}\ c_n\ \mathsf{else}\ \mathsf{abort}\ \mathsf{fi}\ e'_n) \cdots)\ \mathsf{fi}\ e'_2)\ \mathsf{fi}\ e'_1]\!]_\phi \\
&= \mathcal{T}[\![e'_1]\!]^\dagger \circ (\mathcal{C}[\![c_1]\!]_\phi \oplus \mathcal{C}[\![\mathsf{if}\ e_2\ \mathsf{then}\ c_2\ \mathsf{else}\ (\cdots \\
&\qquad\qquad (\mathsf{if}\ e_n\ \mathsf{then}\ c_n\ \mathsf{else}\ \mathsf{abort}\ \mathsf{fi}\ e'_n) \cdots)\ \mathsf{fi}\ e'_2]\!]_\phi) \circ \mathcal{T}[\![e_1]\!] \\
&= \mathcal{T}[\![e'_1]\!]^\dagger \circ (\mathcal{C}[\![c_1]\!]_\phi \oplus \mathcal{C}[\![\mathsf{case}\ e_2 : c_2 : e'_2;\ \cdots\ ;\ e_n : c_n : e'_n\ \mathsf{esac}]\!]_\phi) \circ \mathcal{T}[\![e_1]\!]
\end{aligned}$$

and when $n = 0$ multiconditionals behave as abort.

Multiconditionals follow the *symmetric first match* policy introduced in the reversible functional programming language RFUN [18]. We test $e_1, e_2, \ldots, e_n$ in sequence until we find the firstly satisfied expression e_i. Then, the i'th branch is selected and c_i is executed. The computation continues if assertion e'_i holds and all previous assertions $e'_1, e'_2, \ldots, e'_{i-1}$ do not hold.

Under this assumption, the default branch like in the switch statements in C can be realized by making both e_n and e'_n have a non-nil value, e.g., $e_n \equiv e'_n \equiv (\underline{nil}.\underline{nil})$, which is interpreted as true. These non-nil expressions can be hidden by syntactic sugar:

$$\begin{aligned}
&\mathsf{case}\ e_1 : c_1 : e'_1;\ \cdots\ ;\ e_{n-1} : c_{n-1} : e'_{n-1}\ \mathsf{else}\ c_n\ \mathsf{esac}\ \equiv \\
&\mathsf{case}\ e_1 : c_1 : e'_1;\ \cdots\ ;\ e_{n-1} : c_{n-1} : e'_{n-1};\ (\underline{nil}.\underline{nil}) : c_n : (\underline{nil}.\underline{nil})\ \mathsf{esac}
\end{aligned}$$

In particular, this new case command with a single branch is just interpreted as an if conditional:

$$\mathsf{case}\ e_1 : c_1 : e'_1\ \mathsf{else}\ c_2\ \mathsf{esac}\ \equiv\ \mathsf{if}\ e_1\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}\ e'_1\ .$$

The correctness of those equivalences can be checked by simple derivations using the presented metalanguage.

5.3. Rewriting

A reversible rewrite construct turns out to be very useful for implementing rewriting systems [19]. First, we illustrate the construct with an iterative translation of infix expressions into reverse Polish notation, a representation popular in stack-oriented programming languages. Then, we formalize the rewrite extension.

```
1: proc $post(t)$                                        (* iter.postorder traversal *)
2:   $s \Leftarrow (t.\underline{nil})$;                 (* init stack $s$ to tree $t$     *)
3:   from $=?\ y\ \underline{nil}$ loop                  (* list $y$ empty at entry?  *)
4:     rewrite $(s.y)$ by                                (* rewrite $s$ and $y$           *)
5:       $((\underline{\mathit{0}}.s) \quad .y) \Rightarrow (s \quad .(\underline{\mathit{0}}.y))$;   (* stack top is leaf          *)
6:       $(((l.(\underline{\mathit{1}}.r)).s).y) \Rightarrow ((r.(l.s)).(\underline{\mathit{1}}.y))$  (* stack top is inner node *)
7:     etirwer                                           (* end rewrite               *)
8:   until $=?\ s\ \underline{nil}$;                     (* stack $s$ empty at exit?  *)
9: return $y$;
```

Figure 4: Translation between infix expressions and reverse Polish notation using rewriting.

Example 2. The procedure in Fig. 4 translates an infix expression t to reverse Polish notation (RPN) y by iteratively rewriting a stack of trees s and adding leaves and labels to y (lines 3–8). Stack s is initialized to t (line 2). The representation of t is the same as in Example 1. The reversible iteration is implemented by a while-loop (see Section 5.1). Initially, y is empty; the loop terminates when s is empty. The body of the loop consists of a rewrite statement rewrite...etirwer (lines 4–7) that tests the value of $(s.y)$ against two patterns and rewrites it once. If a leaf is on top of the stack as required by $(\underline{\mathit{0}}.s)$ in the first pattern (line 5) then $\underline{\mathit{0}}$ is popped from the stack and consed onto y. Otherwise, if an inner node is on top of the stack as required by $(l.(\underline{\mathit{1}}.r))$ in the second pattern (line 6), the subtrees l and r are pushed onto the stack and label $\underline{\mathit{1}}$ is consed onto y. The two patterns on the left side are disjoint as are the two patterns on the right side, which make the rewrite statement reversible. Hence, the procedure is reversible, which means uncalling *post* translates expressions written in RPN back to infix notation.

Extending `R-WHILE` with such a construct requires extending the syntax of commands:

$$c ::= \cdots \mid \textsf{rewrite}\ q\ \textsf{by}\ q_1 \Rightarrow q_1';\ \cdots\ ; q_n \Rightarrow q_n'\ \textsf{etirwer}\ .$$

The intended semantics of a rewrite block is as follows: The pattern q is tested against the patterns $q_1, \dots, q_n$ until a match is found — the patterns $q_1, \dots, q_n$ are assumed to be *disjoint* such that q can match at most one of these, as are $q_1', \dots, q_n'$ to ensure reversibility. Once a match q_i is found, q is rewritten by the pattern on the right-hand side, q_i': Knowing that q_i and q match, performing this rewriting amounts to performing the command $q_i \Leftarrow q; q \Leftarrow q_i'$.

Recall that the reversible replacement $q_1 \Leftarrow q_2$ acts as an *assertion with side effects*: If the two patterns q_1 and q_2 match, the replacement is performed, and if they do not, the replacement $q_1 \Leftarrow q_2$ is undefined. As such, determining whether two patterns q_1 and q_2 *match* in a given state (with program context ϕ) amounts to checking whether $\mathcal{Q}[\![q_1 \Leftarrow q_2]\!]_\phi$ is *defined* at that state. For example, the patterns $\underline{foo}$ and x *match* in all states where the variable x evaluates to $\underline{foo}$ — which is the same as saying that $\mathcal{Q}[\![\underline{foo} \Leftarrow x]\!]_\phi$ is *defined* at all states where x evaluates to $\underline{foo}$.

In this way, checking that two patterns match in a given state can be reduced to checking whether a partial function is defined at a given point. For a given partial injection $X \xrightarrow{f} Y$, we define the *classifying predicate* $X \xrightarrow{\chi_f} X \oplus X$ (related, but not equivalent, to the idea of *classified partiality*, see, *e.g.*, [20, 21]) defined as follows:

$$\chi_f(x) = \begin{cases} \mathsf{inr}\ x & \text{if } f \text{ defined at } x \\ \mathsf{inl}\ x & \text{otherwise} \end{cases} .$$

In the convention that left injection corresponds to falsehood, and right injections to truth, χ_f is *true* at x if and only if when f is *defined* at x. This allows us to give semantics to $\mathsf{rewrite}$ blocks in the following way:

$$\mathcal{C}[\![\mathsf{rewrite}\ q\ \mathsf{by}\ q_1 \Rightarrow q_1';\ \cdots\ ;\ q_n \Rightarrow q_n'\ \mathsf{etirwer}]\!]_\phi(\sigma) =$$
$$\begin{cases} f(\mathcal{C}[\![\mathsf{rewrite}\ q\ \mathsf{by}\ q_2 \Rightarrow q_2';\ \cdots\ ;\ q_n \Rightarrow q_n'\ \mathsf{etirwer}]\!]_\phi)(\sigma) & \text{if } n \geq 1 \\ \mathcal{C}[\![\mathsf{abort}]\!]_\phi(\sigma) & \text{if } n = 0 \end{cases}$$
$$\text{where } f(x) = (\chi_{\mathcal{C}[\![q_1' \Leftarrow q]\!]_\phi})^\dagger \circ (x \oplus \mathcal{C}[\![q_1 \Leftarrow q; q \Leftarrow q_1']\!]_\phi) \circ \chi_{\mathcal{C}[\![q_1 \Leftarrow q]\!]_\phi} \ .$$

The right-hand side of the equation is expanded as

$$\underbrace{f(f(\cdots(f}_{n}(\mathcal{C}[\![\mathsf{abort}]\!]_\phi))\cdots))(\sigma) \ .$$

As such, this is essentially an n-armed multiconditional using the classifying predicate $\chi_{\mathcal{C}[\![q_i \Leftarrow q]\!]_\phi}$ to test whether the i'th branch should be chosen, and using the inverse classifying predicate $(\chi_{\mathcal{C}[\![q_i' \Leftarrow q]\!]_\phi})^\dagger$ as the exit assertion of the i'th branch. Since $q_i' \Leftarrow q$ is inverse to $q \Leftarrow q_i'$ (see Section 4.8), it *must* be defined after performing $q \Leftarrow q_i'$ if this is defined at all, and since $q_1', \ldots, q_n'$ are assumed to be disjoint, *at most one* of the classifying predicates $\chi_{\mathcal{C}[\![q_i' \Leftarrow q]\!]_\phi}$ is true at any one time. Hence, the exit assertion property is satisfied.

Note that since $q_1, \ldots, q_n$ and $q_1', \ldots, q_n'$, respectively, are required to be disjoint, in principle, we are free to change the order of branches. This approach is more conservative than the *symmetric first match* policy used in the reversible functional programming language RFUN [18].

Alternatively, we can define $\chi_{\mathcal{C}[\![q_1 \Leftarrow q]\!]_\phi}$ in the presented metalanguage $\mathcal{L}$:

$$\chi_{\mathcal{C}[\![q_1 \Leftarrow q]\!]_\phi} = ((\mathcal{Q}[\![q]\!]_\phi)^\dagger \oplus (f^\dagger \circ \kappa_2)) \circ f$$

where $f = \mathcal{Q}'[\![q_1]\!]_\phi \circ \mathcal{Q}[\![q]\!]_\phi$ and $\Sigma \times \mathbb{V} \xrightarrow{\mathcal{Q}'[\![q]\!]} (\Sigma \times \mathbb{V}) \oplus \Sigma$. Essentially, $\mathcal{Q}'[\![q]\!]_\phi$ behaves like the inverse of $\mathcal{Q}[\![q]\!]_\phi$ and its result is with the tag inr. But $\mathcal{Q}'[\![q]\!]_\phi$ can also detect pattern matching failures. When pattern matching fails, the result is the original input with the tag inl. Specifically, we can represent the functionality of $\mathcal{Q}'[\![q]\!]_\phi$ as follows:

$$\mathcal{Q}'[\![q]\!]_\phi(\sigma, v) = \begin{cases} \mathsf{inr}\ ((\mathcal{Q}[\![q]\!]_\phi)^\dagger(\sigma, v)) & \text{if } (\mathcal{Q}[\![q]\!]_\phi)^\dagger \text{ is defined at } (\sigma, v) \\ \mathsf{inl}\ (\sigma, v) & \text{otherwise} \end{cases} .$$

The exact definition of this function in the presented metalanguage $\mathcal{L}$ is in Appendix C.

6. Related Work

Formal meaning has been given to reversible programming languages using well-established formalisms such as operational semantics to the imperative language Janus [1], the functional language RFUN [18], and to concurrent languages [22], small-step operational semantics to the assembler language PISA [15], transition functions to the flowchart language RFCL [2], and denotational semantics to `R-WHILE` [3, 4]. The reversibility of a language is not directly expressed by these formalisms. It is up to the language designer's discipline to capture reversibility and to show the inversion properties for each language individually. Additionally, the semantics of `R-WHILE` was first expressed irreversibly [1]. The type and effects systems were studied for reversible languages [23] (see also [24]).

In this paper, the reversible elements of `R-WHILE` are composed by the metalanguage $\mathcal{L}$ in a manner that preserves 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 [2]. The categorical approach to reversible structured flowchart semantics was pioneered by [5]. Similarly, reversible Turing machines were built from reversible rotary elements [25].

The category **PInj** of sets and partial injective functions has a rich history of study in relation to various reversible computation models (e.g., [26, 12]), but perhaps in particular those arising from the Geometry of Interaction (see [27] for an overview). An interesting recent example concerns the denotational semantics of Janus based on partial injective functions [8], and given that this approach is based on the same category as our metalanguage $\mathcal{L}$ is, there are many similarities, though presentations differ. To make similarities clearer, we show in Figure 5 the representation of (certain) $\mathcal{L}$ terms

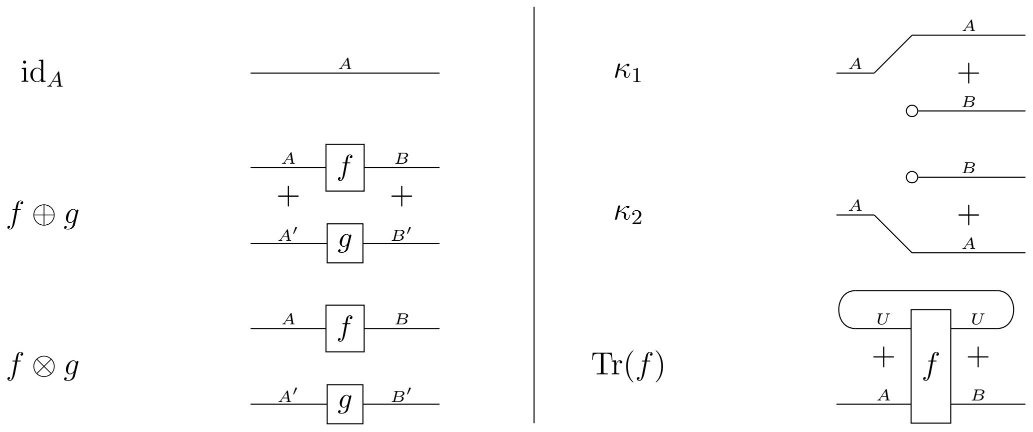


Figure 5: An overview of (certain) $\mathcal{L}$ terms and their graphical representation in the style of [8].

in the graphical language employed by [8] (see also [13] regarding graphical languages).

To give meaning to reversible languages by translators and interpreters is another operational approach to a semantics. Examples include reversible interpreters [1, 3], translation of the high-level language R to the reversible assembler language PISA [28], mapping hardware descriptions in SyReC to reversible circuits [29], and processor architectures [30]. A different approach is the reversibilization of irreversible languages by extending the operational semantics by tracing to undo program runs [31]. Alternatively, irreversible programs can be inverted using different program inversion approaches, *e.g.*, [32, 33, 34]. 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 [35].

7. Conclusion

Reversible systems have reversible semantics. In the present paper, we built on a metalanguage foundation intended to describe the semantics of reversible programming languages, which we demonstrated by the full development of a formal semantics of the reversibly-universal language `R-WHILE`. It 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 in giving formal reversible semantics. A language defined in the metalanguage is guaranteed to have reversibility, which means it requires no explicit proof of reversibility. We argued that this metalanguage approach serves as a strong basis for understanding and reasoning about reversible programs.

It could be interesting to further explore how advanced object-oriented structures, combinators, or features for concurrency are best described and the metalanguage features that may be useful. Characterizing reversible heap allocation and concurrent reversible computations are some of those challenges. However, its practical feasibility and relationship to advanced reversible automata including nondeterminism, *e.g.*, [36] remains to be explored.

Acknowledgments. The authors would like to thank Markus Holzer and Martin Kutrib for editing this special issue, and the anonymous reviewers for their useful feedback. The second author is supported by the Independent Research Fund Denmark under DFF-International Postdoc Fellowship No. 0131-00025B. The third author is supported by JSPS KAKENHI Grant Number 18K11250 and Nanzan University Pache Research Subsidy I-A-2 for the 2021 academic year.

References

[1] T. Yokoyama, R. Glück, A reversible programming language and its invertible self-interpreter, in: Partial Evaluation and Semantics-Based Program Manipulation. Proceedings, ACM Press, 2007, pp. 144–153.

[2] T. Yokoyama, H. B. Axelsen, R. Glück, Fundamentals of reversible flowchart languages, Theor. Comput. Sci. 611 (2016) 87–115.

[3] R. Glück, T. Yokoyama, A minimalist's reversible while language, IEICE Transactions on Information and Systems E100-D (5) (2017) 1026–1034.

[4] R. Glück, T. Yokoyama, Constructing a binary tree from its traversals by reversible recursion and iteration, Inf. Process. Lett. 147 (2019) 32–37.

[5] R. Glück, R. Kaarsgaard, A categorical foundation for structured reversible flowchart languages: Soundness and adequacy, Log. Methods Comput. Sci. 14 (3) (2018).

[6] R. Kaarsgaard, H. B. Axelsen, R. Glück, Join inverse categories and reversible recursion, J. Log. Algebr. Methods 87 (2017) 33–50.

[7] R. Kaarsgaard, Inversion, iteration, and the art of dual wielding, in: M. K. Thomsen, M. Soeken (Eds.), Reversible Computation. Proceedings, Vol. 11497 of LNCS, Springer, 2019, pp. 34–50.

[8] L. Paolini, M. Piccolo, L. Roversi, A certified study of a reversible programming language, in: T. Uustalu (Ed.), 21st International Conference on Types for Proofs and Programs (TYPES 2015), Vol. 69 of Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2018, pp. 7:1–7:21.

[9] G. Winskel, The Formal Semantics of Programming Languages: An Introduction, MIT Press, 1993.

[10] R. Glück, R. Kaarsgaard, T. Yokoyama, Reversible programs have reversible semantics, in: E. Sekerinski, et al. (Eds.), Formal Methods. FM 2019 International Workshops. Proceedings, Vol. 12233 of LNCS, Springer, 2020, pp. 413–427.

[11] R. Cockett, S. Lack, Restriction categories III: Colimits, partial limits and extensivity, Math. Struct. Comput. Sci. 17 (4) (2007) 775–817.

[12] B. Giles, An investigation of some theoretical aspects of reversible computing, Ph.D. thesis, University of Calgary (2014).

[13] P. Selinger, A survey of graphical languages for monoidal categories, in: B. Coecke (Ed.), New Structures for Physics, Vol. 813 of LNP, Springer, 2010, pp. 289–355.

[14] E. Haghverdi, A categorical approach to linear logic, geometry of proofs and full completeness, Ph.D. thesis, Carlton University and University of Ottawa (2000).

[15] H. B. Axelsen, R. Glück, T. Yokoyama, Reversible machine code and its abstract processor architecture, in: V. Diekert, M. V. Volkov, A. Voronkov (Eds.), Computer Science - Theory and Applications. Proceedings, Vol. 4649 of LNCS, Springer, 2007, pp. 56–69.

[16] R. Kaarsgaard, Condition/decision duality and the internal logic of extensive restriction categories, Electronic Notes in Theoretical Computer Science 347 (2019) 179–202.

[17] A. Joyal, R. Street, D. Verity, Traced monoidal categories, Math. Proc. Camb. Phil. Soc. 119 (3) (1996) 447–468.

[18] T. Yokoyama, H. B. Axelsen, R. Glück, Towards a reversible functional language, in: A. De Vos, R. Wille (Eds.), Reversible Computation. Proceedings, Vol. 7165 of LNCS, Springer, 2012, pp. 14–29.

[19] R. Glück, T. Yokoyama, A linear-time self-interpreter of a reversible imperative language, Computer Software 33 (3) (2016) 108–128.

[20] A. Kock, Algebras for the partial map classifier monad, in: A. Carboni, M. C. Pedicchio, G. Rosolini (Eds.), Category Theory. Proceedings, Vol. 1488 of LNM, Springer, 1991, pp. 262–278.

[21] J. Cockett, S. Lack, Restriction categories II: partial map classification, Theor. Comput. Sci. 294 (1) (2003) 61–102.

[22] S. Kuhn, I. Ulidowski, A calculus for local reversibility, in: S. Devitt, I. Lanese (Eds.), Reversible Computation. Proceedings, Vol. 9720 of LNCS, Springer, 2016, pp. 20–35.

[23] R. P. James, A. Sabry, Information effects, in: Principles of Programming Languages. Proceedings, ACM Press, 2012, pp. 73–84.

[24] R. Kaarsgaard, N. Veltri, En garde! Unguarded iteration for reversible computation in the delay monad, in: G. Hutton (Ed.), Mathematics of Program Construction. Proceedings, Vol. 11825 of LNCS, Springer, 2019, pp. 366–384.

[25] K. Morita, Reversible computing and cellular automata — A survey, Theor. Comput. Sci. 395 (1) (2008) 101–131.

[26] S. Abramsky, A structural approach to reversible computation, Theor. Comput. Sci. 347 (3) (2005) 441–464.

[27] E. Haghverdi, P. Scott, Geometry of interaction and the dynamics of proof reduction: a tutorial, in: B. Coecke (Ed.), New Structures for Physics, Vol. 813 of LNP, Springer, 2010, pp. 357–417.

[28] M. P. Frank, Reversibility for efficient computing, Ph.D. thesis, MIT (1999).

[29] R. Wille, E. Schönborn, M. Soeken, R. Drechsler, SyReC: A hardware description language for the specification and synthesis of reversible circuits, Integration 53 (2016) 39–53.

[30] M. K. Thomsen, H. B. Axelsen, R. Glück, A reversible processor architecture and its reversible logic design, in: A. De Vos, R. Wille (Eds.), Reversible Computation. Proceedings, Vol. 7165 of LNCS, Springer, 2012, pp. 30–42.

[31] J. Hoey, I. Ulidowski, S. Yuen, Reversing parallel programs with blocks and procedures, in: J. A. Pérez, S. Tini (Eds.), Expressiveness in Concurrency, Structural Operational Semantics. Proceedings, Vol. 276 of Electronic Proceedings in Theoretical Computer Science, 2018, pp. 69–86.

[32] R. Glück, M. Kawabe, Revisiting an automatic program inverter for Lisp, SIGPLAN Not. 40 (5) (2005) 8–17.

[33] M. Kawabe, R. Glück, The program inverter LRinv and its structure, in: H. Manuel, D. Cabeza (Eds.), Practical Aspects of Declarative Languages. Proceedings, Vol. 3350 of LNCS, Springer, 2005, pp. 219–234.

[34] R. Glück, V. F. Turchin, Application of metasystem transition to function inversion and transformation, in: International Symposium on Symbolic and Algebraic Computation. Proceedings, ACM Press, 1990, pp. 286–287.

[35] J. Kari, Reversible cellular automata: from fundamental classical results to recent developments, New Gener. Comput. 36 (3) (2018) 145–172.

[36] M. Holzer, M. Kutrib, Reversible nondeterministic finite automata, in: I. Phillips, H. Rahaman (Eds.), Reversible Computation. Proceedings, Vol. 10301 of LNCS, Springer, 2017, pp. 35–51.

Appendix A. The Correctness of Command Inverter $\mathcal{I}$

As in [5, Section 8], the correctness of the command inverter $\mathcal{I}$ defined in Fig. 3 are shown as follows. We prove that for any command c there is a command $\mathcal{I}[\![c]\!]$ such that $\mathcal{C}[\![\mathcal{I}[\![c]\!]]\!]_\sigma = (\mathcal{C}[\![c]\!]_\sigma)^\dagger$ for any σ. To prove it by structural induction on command c, it is sufficient to show the following derivations:

$$\begin{aligned}
&\mathcal{C}[\![\mathcal{I}[\![c_1; c_2]\!]]\!]_\phi \\
&= \mathcal{C}[\![\mathcal{I}[\![c_2]\!];\ \mathcal{I}[\![c_1]\!]]\!]_\phi \\
&= \mathcal{C}[\![\mathcal{I}[\![c_2]\!]]\!]_\phi \circ \mathcal{C}[\![\mathcal{I}[\![c_1]\!]]\!]_\phi \\
&= (\mathcal{C}[\![c_2]\!]_\phi)^\dagger \circ (\mathcal{C}[\![c_1]\!]_\phi)^\dagger \\
&= (\mathcal{C}[\![c_1]\!]_\phi \circ \mathcal{C}[\![c_2]\!]_\phi)^\dagger \\
&= (\mathcal{C}[\![c_1;\ c_2]\!]_\phi)^\dagger
\end{aligned}$$

$$\begin{aligned}
&\mathcal{C}[\![\mathcal{I}[\![x_i \mathrel{\widehat{=}} e_1]\!]]\!]_\sigma \\
&= \mathcal{C}[\![x_i \mathrel{\widehat{=}} e_1]\!]_\sigma
\end{aligned}$$

$$
\begin{aligned}
&= (\mathcal{Q}[\![x_i]\!]_\phi)^\dagger \circ \mathcal{E}[\![e_1]\!] \circ \mathcal{Q}[\![x_i]\!]_\phi \\
&= ((\mathcal{Q}[\![x_i]\!]_\phi)^\dagger \circ (\mathcal{E}[\![e_1]\!])^\dagger \circ (\mathcal{Q}[\![x_i]\!]_\phi)^{\dagger\dagger})^\dagger \\
&= ((\mathcal{Q}[\![x_i]\!]_\phi)^\dagger \circ \mathcal{E}[\![e_1]\!] \circ \mathcal{Q}[\![x_i]\!]_\phi)^\dagger \\
&= (\mathcal{C}[\![x_i \mathrel{\widehat{=}} e_1]\!]_\sigma)^\dagger
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{C}[\![\mathcal{I}[\![q_1 \Leftarrow q_2]\!]]\!]_\phi \\
&= \mathcal{C}[\![q_2 \Leftarrow q_1]\!]_\phi \\
&= (\mathcal{Q}[\![q_2]\!]_\phi)^\dagger \circ \mathcal{Q}[\![q_1]\!]_\phi \\
&= ((\mathcal{Q}[\![q_1]\!]_\phi)^\dagger \circ \mathcal{Q}[\![q_2]\!]_\phi)^\dagger \\
&= (\mathcal{C}[\![q_1 \Leftarrow q_2]\!]_\phi)^\dagger
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{C}[\![\mathcal{I}[\![\mathsf{if}\ e_1\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}\ e_2]\!]]\!]_\phi \\
&= \mathcal{C}[\![\mathsf{if}\ e_2\ \mathsf{then}\ \mathcal{I}[\![c_1]\!]\ \mathsf{else}\ \mathcal{I}[\![c_2]\!]\ \mathsf{fi}\ e_1]\!]_\phi \\
&= \mathcal{T}[\![e_2]\!]^\dagger \circ (\mathcal{C}[\![\mathcal{I}[\![c_2]\!]]\!]_\phi \oplus \mathcal{C}[\![\mathcal{I}[\![c_1]\!]]\!]_\phi) \circ \mathcal{T}[\![e_1]\!] \\
&= (\mathcal{T}[\![e_2]\!]^\dagger \circ ((\mathcal{C}[\![\mathcal{I}[\![c_2]\!]]\!]_\phi)^\dagger \oplus (\mathcal{C}[\![\mathcal{I}[\![c_1]\!]]\!]_\phi)^\dagger) \circ \mathcal{T}[\![e_1]\!])^\dagger \\
&= (\mathcal{T}[\![e_2]\!]^\dagger \circ (\mathcal{C}[\![c_2]\!]_\phi \oplus \mathcal{C}[\![c_1]\!]_\phi) \circ \mathcal{T}[\![e_1]\!])^\dagger \\
&= (\mathcal{C}[\![\mathsf{if}\ e_2\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}\ e_1]\!]_\phi)^\dagger
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{C}[\![\mathcal{I}[\![\mathsf{from}\ e_1\ \mathsf{do}\ c_1\ \mathsf{loop}\ c_2\ \mathsf{until}\ e_2]\!]]\!]_\phi \\
&= \mathcal{C}[\![\mathsf{from}\ e_2\ \mathsf{do}\ \mathcal{I}[\![c_1]\!]\ \mathsf{loop}\ \mathcal{I}[\![c_2]\!]\ \mathsf{until}\ e_1]\!]_\phi \\
&= \mathrm{Tr}\left((\mathrm{id}_\Sigma \oplus \mathcal{C}[\![\mathcal{I}[\![c_2]\!]]\!]_\phi) \circ \mathcal{T}[\![e_1]\!] \circ \mathcal{C}[\![\mathcal{I}[\![c_1]\!]]\!]_\phi \circ \mathcal{T}[\![e_2]\!]^\dagger\right) \\
&= \mathrm{Tr}\left(\mathcal{T}[\![e_1]\!] \circ \mathcal{C}[\![\mathcal{I}[\![c_1]\!]]\!]_\phi \circ \mathcal{T}[\![e_2]\!]^\dagger \circ (\mathrm{id}_\Sigma \oplus \mathcal{C}[\![\mathcal{I}[\![c_2]\!]]\!]_\phi)\right) \\
&= \left(\mathrm{Tr}\left((\mathrm{id}_\Sigma \oplus (\mathcal{C}[\![\mathcal{I}[\![c_2]\!]]\!]_\phi)^\dagger) \circ \mathcal{T}[\![e_2]\!] \circ (\mathcal{C}[\![\mathcal{I}[\![c_1]\!]]\!]_\phi)^\dagger \circ \mathcal{T}[\![e_1]\!]^\dagger\right)\right)^\dagger \\
&= \left(\mathrm{Tr}\left((\mathrm{id}_\Sigma \oplus \mathcal{C}[\![c_2]\!]_\phi) \circ \mathcal{T}[\![e_2]\!] \circ \mathcal{C}[\![c_1]\!]_\phi \circ \mathcal{T}[\![e_1]\!]^\dagger\right)\right)^\dagger \\
&= (\mathcal{C}[\![\mathsf{from}\ e_1\ \mathsf{do}\ c_1\ \mathsf{loop}\ c_2\ \mathsf{until}\ e_2]\!]_\phi)^\dagger
\end{aligned}
$$

Because for any σ, $\mathcal{C}[\![\mathcal{I}[\![\mathcal{I}[\![c]\!]]\!]]\!]_\sigma = (\mathcal{C}[\![\mathcal{I}[\![c]\!]]\!]_\sigma)^\dagger = (\mathcal{C}[\![c]\!]_\sigma)^{\dagger\dagger}$, we have $\mathcal{I}[\![\mathcal{I}[\![c]\!]]\!] = c$.

Appendix B. Renaming Procedure Names $\mathcal{R}$

The translator $\mathcal{R}$ replaces each procedure f_i with its inverse f_i^{inv} with preserving the meaning. Note that call/uncall patterns are flipped to uncall/call patterns, respectively.

$$\begin{aligned}
\mathcal{R}[\![x_j]\!] &\equiv x_j \\
\mathcal{R}[\![\underline{s}]\!] &\equiv \underline{s} \\
\mathcal{R}[\![\mathsf{call}\ f_i(q_1)]\!] &\equiv \mathsf{uncall}\ f_i^{inv}(\mathcal{R}[\![q_1]\!]) \\
\mathcal{R}[\![\mathsf{uncall}\ f_i(q_1)]\!] &\equiv \mathsf{call}\ f_i^{inv}(\mathcal{R}[\![q_1]\!]) \\
\mathcal{R}[\![(q_1.q_2)]\!] &\equiv (\mathcal{R}[\![q_1]\!].\mathcal{R}[\![q_2]\!])
\end{aligned}$$

$$\begin{aligned}
\mathcal{R}[\![x \mathrel{\widehat{=}} e]\!] &\equiv x \mathrel{\widehat{=}} e \\
\mathcal{R}[\![q_1 \Leftarrow q_2]\!] &\equiv \mathcal{R}[\![q_1]\!] \Leftarrow \mathcal{R}[\![q_2]\!] \\
\mathcal{R}[\![c_1; c_2]\!] &\equiv \mathcal{R}[\![c_1]\!];\ \mathcal{R}[\![c_2]\!] \\
\mathcal{R}[\![\mathsf{if}\ e_1\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}\ e_2]\!] &\equiv \mathsf{if}\ e_1\ \mathsf{then}\ \mathcal{R}[\![c_1]\!]\ \mathsf{else}\ \mathcal{R}[\![c_2]\!]\ \mathsf{fi}\ e_2 \\
\mathcal{R}[\![\mathsf{from}\ e_1\ \mathsf{do}\ c_1\ \mathsf{loop}\ c_2\ \mathsf{until}\ e_2]\!] &\equiv \mathsf{from}\ e_1\ \mathsf{do}\ \mathcal{R}[\![c_1]\!]\ \mathsf{loop}\ \mathcal{R}[\![c_2]\!]\ \mathsf{until}\ e_2
\end{aligned}$$

Appendix C. The helper semantic function $\mathcal{Q}'[\![q]\!]_\phi$

$$\mathcal{Q}'[\![x_j]\!]_\phi(\sigma, v) = \begin{cases} \mathsf{inr}\ (\mathit{extract}_j^\dagger(\sigma, v)) & \text{if } \pi_j(\sigma) = v \\ \mathsf{inl}\ (\sigma, v) & \text{otherwise} \end{cases}$$

$$\mathcal{Q}'[\![\underline{s}]\!]_\phi(\sigma, v) = \begin{cases} \mathsf{inr}\ \sigma & \text{if } \overline{s} = v \\ \mathsf{inl}\ (\sigma, v) & \text{otherwise} \end{cases}$$

$$\mathcal{Q}'[\![\mathsf{call}\ f_i(q_1)]\!]_\phi(\sigma, v) = \begin{cases} \mathsf{inr}\ \sigma_1 & \text{if } \mathcal{Q}'[\![q_1]\!]_\phi(\sigma, v_1) = \mathsf{inr}\ \sigma_1 \text{ and } \kappa_i^\dagger(\phi^\dagger(\kappa_i(v_1))) = v \\ \mathsf{inl}\ (\sigma, v) & \text{if } \mathcal{Q}'[\![q_1]\!]_\phi(\sigma, v) = \mathsf{inl}\ x \end{cases}$$

$$\mathcal{Q}'[\![\mathsf{uncall}\ f_i(q_1)]\!]_\phi(\sigma, v) = \begin{cases} \mathsf{inr}\ \sigma_1 & \text{if } \mathcal{Q}'[\![q_1]\!]_\phi(\sigma, v_1) = \mathsf{inr}\ \sigma_1 \text{ and } \kappa_i^\dagger(\phi(\kappa_i(v))) = v_1 \\ \mathsf{inl}\ (\sigma, v) & \text{if } (\mathcal{Q}'[\![q_1]\!]_\phi(\sigma, v_1) = \mathsf{inl}\ x \text{ and } \kappa_i^\dagger(\phi(\kappa_i(v))) = v_1) \\ & \quad \text{or } \mathcal{Q}'[\![q_1]\!]_\phi(\sigma, v) = \mathsf{inl}\ x \end{cases}$$

$$\mathcal{Q}'[\![(q_1.q_2)]\!]_\phi(\sigma, v_1 \bullet v_2) = \begin{cases} \mathsf{inr}\ \sigma_1 & \text{if } \mathcal{Q}'[\![q_2]\!]_\phi(\sigma, v_2) = \mathsf{inr}\ \sigma_2 \text{ and } \mathcal{Q}'[\![q_1]\!]_\phi(\sigma_2, v_1) = \mathsf{inr}\ \sigma_1 \\ \mathsf{inl}\ (\sigma, v_1 \bullet v_2) & \text{if } (\mathcal{Q}'[\![q_2]\!]_\phi(\sigma, v_2) = \mathsf{inr}\ \sigma_2 \text{ and } \mathcal{Q}'[\![q_1]\!]_\phi(\sigma_2, v_1) = \mathsf{inl}\ x) \\ & \quad \text{or } \mathcal{Q}'[\![q_2]\!]_\phi(\sigma, v_2) = \mathsf{inl}\ x \end{cases}$$

Appendix D. Conditionals as Syntax Sugar

This is an example equational reasoning using $\mathcal{L}$.
We can swap two values $\mathit{swap}(x_1, x_2)$ by using the zeroed variable x_3.

$$\mathit{swap}(x_1, x_2) \equiv x_3 \Leftarrow x_1;\ x_1 \Leftarrow x_2;\ x_2 \Leftarrow x_3$$

The one-armed loop is sufficient to simulate reversible conditionals as demonstrated in [3]:

$$\mathsf{if}\ x_1\ \mathsf{else}\ c_1\ \mathsf{fi}\ x_1 \equiv \mathsf{from}\ x_2\ \mathsf{loop}\ c_1;\ \mathit{swap}(x_1, x_2)\ \mathsf{until}\ x_1;\ \mathit{swap}(x_1, x_2)$$

This law is proved by the following derivation. We assume $\mathcal{T}[\![x_2]\!](\sigma) = \mathsf{inr}\ \sigma$. Let $f = (\mathcal{C}[\![c_1;\ \mathit{swap}(x_1, x_2)]\!]_\phi \oplus \mathrm{id}_\Sigma) \circ \mathcal{T}[\![x_1]\!] \circ \mathcal{T}[\![x_2]\!]^\dagger$.

$$\begin{aligned}
&\mathcal{C}[\![\mathsf{from}\ x_2\ \mathsf{loop}\ c_1;\ \mathit{swap}(x_1, x_2)\ \mathsf{until}\ x_1]\!]_\phi(\sigma)\\
&= \mathrm{Tr}(f)(\sigma)\\
&= \mathit{pretrace}(f)(\mathsf{inr}\ \sigma)\\
&= \begin{cases} \sigma & \text{if } \mathcal{T}[\![x_1]\!](\sigma) = \mathsf{inr}\ \sigma \\ \mathit{pretrace}(f)(\mathsf{inl}\ (\mathcal{C}[\![c_1;\ \mathit{swap}(x_1, x_2)]\!]_\phi(\sigma))) & \text{if } \mathcal{T}[\![x_1]\!](\sigma) = \mathsf{inl}\ \sigma \end{cases}\\
&= \begin{cases} \sigma & \text{if } \mathcal{T}[\![x_1]\!](\sigma) = \mathsf{inr}\ \sigma \\ \mathcal{C}[\![c_1;\ \mathit{swap}(x_1, x_2)]\!]_\phi(\sigma) & \text{if } \mathcal{T}[\![x_1]\!](\sigma) = \mathsf{inl}\ \sigma \end{cases}
\end{aligned}$$

Hence, we have

$$\begin{aligned}
&\mathcal{C}[\![\mathsf{from}\ x_2\ \mathsf{loop}\ c_1;\ \mathit{swap}(x_1, x_2)\ \mathsf{until}\ x_1;\ \mathit{swap}(x_1, x_2)]\!]_\phi(\sigma)\\
&= \begin{cases} \mathcal{C}[\![\mathit{swap}(x_1, x_2)]\!]_\phi(\sigma) & \text{if } \mathcal{T}[\![x_1]\!](\sigma) = \mathsf{inr}\ \sigma \\ \mathcal{C}[\![c_1]\!]_\phi(\sigma) & \text{if } \mathcal{T}[\![x_1]\!](\sigma) = \mathsf{inl}\ \sigma \end{cases}\\
&= \begin{cases} \sigma & \text{if } \mathcal{T}[\![x_1]\!](\sigma) = \mathsf{inr}\ \sigma \\ \mathcal{C}[\![c_1]\!]_\phi(\sigma) & \text{if } \mathcal{T}[\![x_1]\!](\sigma) = \mathsf{inl}\ \sigma \end{cases}\\
&= (\mathcal{T}[\![x_1]\!] \circ (\mathrm{id}_\Sigma \oplus \mathcal{C}[\![c_1]\!]_\phi) \circ \mathcal{T}[\![x_1]\!]^\dagger)(\sigma)\\
&= \mathcal{C}[\![\mathsf{if}\ x_1\ \mathsf{else}\ c_1\ \mathsf{fi}\ x_1]\!]_\phi(\sigma)
\end{aligned}$$