Reversible Programs Have Reversible Semantics

RPLA 2019

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

October 9, 2019

[1] DIKU, Department of Computer Science, University of Copenhagen, Denmark
[2] Department of Software Engineering, Nanzan University, Japan

robin@di.ku.dk

http://www.di.ku.dk/~robin

Reversible Programs Have Reversible Semantics

Logical Methods in Computer Science
Vol. 14(3:16)2018, pp. 1–38
https://lmcs.episciences.org/

Submitted Oct. 12, 2017
Published Sep. 05, 2018

A CATEGORICAL FOUNDATION FOR STRUCTURED REVERSIBLE FLOWCHART LANGUAGES: SOUNDNESS AND ADEQUACY

ROBERT GLÜCK AND ROBIN KAARSGAARD

DIKU, Department of Computer Science, University of Copenhagen, Denmark
e-mail address: glueck@acm.org
e-mail address: robin@di.ku.dk

Abstract. Structured reversible flowchart languages is a class of imperative reversible programming languages allowing for a simple diagrammatic representation of control flow built from a limited set of control flow structures. This class includes the reversible programming language Janus (without recursion), as well as more recently developed reversible programming languages such as `R-CORE` and `R-WHILE`.

In the present paper, we develop a categorical foundation for this class of languages based on inverse categories with joins. We generalize the notion of extensivity of restriction categories to one that may be accommodated by inverse categories, and use the resulting *decisions* to give a reversible representation of predicates and assertions. This leads to a categorical semantics for structured reversible flowcharts, which we show to be computationally sound and adequate, as well as equationally fully abstract with respect to the operational semantics under certain conditions.

1. Introduction

Reversible computing is an emerging paradigm that adopts a physical principle of reality into a *computation model without information erasure*. Reversible computing extends the standard forward-only mode of computation with the ability to execute in reverse as easily as forward. Reversible computing is a necessity in the context of quantum computing and some bio-inspired computation models. Regardless of the physical motivation, bidirectional determinism is interesting in its own right. The potential benefits include the design of innovative reversible architectures (*e.g.*, [34, 33, 37]), new programming models and techniques (*e.g.*, [40, 17, 28]), and the enhancement of software with reversibility (*e.g.*, [8]).

Today the semantics of reversible programming languages are usually formalized using traditional metalanguages, such as structural operational semantics or denotational semantics

Key words and phrases: Reversible computing, flowchart languages, structured programming, categorical semantics, category theory.

This is the extended version of an article presented at MFPS XXXIII [16], extended with proofs that previously appeared in the appendix, as well as new sections on soundness, adequacy, and full abstraction.

The authors acknowledge the support given by *COST Action IC1405 Reversible computation: Extending horizons of computing.* We also thank the anonymous reviewers of MFPS XXXIII for their thoughtful and detailed comments on a previous version of this paper.

LOGICAL METHODS IN COMPUTER SCIENCE
DOI:10.23638/LMCS-14(3:16)2018

Overview

1. The formalization problem
2. `RWHILE` with procedures
3. **PInj**: A reversible metalanguage
4. Aspects of the semantics
5. Concluding remarks

The formalization problem

Suppose that you're in the process of designing a new *reversible* programming language, $\mathcal{L}$.

To give $\mathcal{L}$-programs meaning, you define a simple operational semantics for $\mathcal{L}$.

$$\frac{\sigma \vdash b_1 \rightsquigarrow tt \qquad \sigma \vdash c_1 \downarrow \sigma' \qquad \sigma' \vdash b_2 \rightsquigarrow tt}{\sigma \vdash \textbf{if } b_1 \textbf{ then } c_1 \textbf{ else } c_2 \textbf{ fi } b_2 \downarrow \sigma'}$$

How do you show that $\mathcal{L}$ is reversible? *You prove a theorem.*

Proving reversibility of (imperative) reversible programming languages usually amounts to proving two lemmas:

Lemma (Forward determinism): *For every command* $\mathtt{c}$ *and store* σ, *there exists* at most *one store* σ' *such that* $\sigma \vdash c \downarrow \sigma'$.

Proof. By structural induction on derivations.

Lemma (Backward determinism): *For every command* $\mathtt{c}$ *and store* σ', *there exists* at most *one store* σ *such that* $\sigma \vdash c \downarrow \sigma'$.

Proof. By structural induction on derivations.

It takes a bit of work to prove this, and though it's pretty tedious, you finally get your proof of reversibility. Not too bad, right?

The formalization problem

What happens if you decide to change $\mathcal{L}$? *You need to change the proofs.*

What happens when you decide to design a new and improved reversible language, $\mathcal{L}'$? *You need to prove this all over again!*

There is a *disconnect* in the properties of our object language and meta language: We want the object language to guarantee reversibility, but the meta language is completely oblivious to this property.

The formalization problem

More to the point, there is a *disconnect* between object-level constructs and meta-level constructs.

Operational semantics describe commands as *relations*.
Reversible commands are *partial injective functions*.

Operational semantics are, in a sense, *too general*. This leads to us having to prove properties of object languages, in this case reversibility, in an *ad hoc, case-by-case* manner.

This suggests the need for a *more specialized metalanguage*.

RWHILE WITH PROCEDURES

RWHILE is a simple reversible imperative programming language with dynamic data, pattern matching, and reversible control structures, introduced by Glück and Yokoyama in 2015.

More recently, it was extended with support for *procedures* and procedure *calls* and *uncalls*, including (mutually) recursive procedure systems.

With or without procedures, RWHILE is r-Turing complete.

```
proc infix2pre(t)                   (* infix exp to Polish notation *)
  y ⇐ call pre((t.nil));            (* call preorder traversal      *)
  return y;

proc pre2infix(y)                   (* Polish notation to infix exp *)
  (t.nil) ⇐ uncall pre(y);          (* uncall preorder traversal    *)
  return t;

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

PInj: A reversible metalanguage

Unlike the metalanguage of operational semantics, **PInj** is a metalanguage which *guarantees* reversibility – no theorems needed!

PInj is a *category*, the category of sets and partial injective functions.

However, *no categorical background is assumed!*

There are no tricks up my sleeve: The meta-language was *not* designed with `RWHILE` (or any other) particular language in mind.

In **PInj** we can, among other things …

- form simple partial injective functions $f(x) = \ldots$, so long as it is immediately clear that they are injective,
- compose partial injective functions $X \xrightarrow{f} Y$ and $Y \xrightarrow{g} Z$ to form their *composite* $X \xrightarrow{g \circ f} Z$, $(g \circ f)(x) = g(f(x))$,
- invert a partial injective function $X \xrightarrow{f} Y$ to form its *partial inverse* $Y \xrightarrow{f^\dagger} X$.

In other words, partial injective functions are closed under composition and inversion.

Inversion and composition interact: $(g \circ f)^\dagger = f^\dagger \circ g^\dagger$.

Sets and partial injective functions

In **PInj** we can, among other things …

- form the *cartesian product* $X \otimes Y$ of sets X and Y, *and* of partial injective functions: Given $X \xrightarrow{f} Y$ and $X' \xrightarrow{g} Y'$, we define a partial injection $X \otimes X' \xrightarrow{f \otimes g} Y \otimes Y'$ by

$$(f \otimes g)(x, x') = (f(x), g(x'))$$

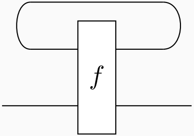
- form the *tagged union* of sets $X \oplus Y$ and partial injective functions $X \xrightarrow{f} Y$ and $X' \xrightarrow{g} Y'$. Elements of $X \oplus Y$ are of the form $\mathrm{inl}(x)$ for $x \in X$, and $\mathrm{inr}(y)$ for $y \in Y$, and $X \oplus X' \xrightarrow{f \oplus g} Y \oplus Y'$ is defined by

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

Note that $(f \otimes g)^\dagger = f^\dagger \otimes g^\dagger$ and $(f \oplus g)^\dagger = f^\dagger \oplus g^\dagger$.

In **PInj** we can, among other things …

- form the *trace* $X \xrightarrow{\mathrm{Tr}(f)} Y$ of a partial injective function $X \oplus U \xrightarrow{f} Y \oplus U$



  Note that this satisfies $\mathrm{Tr}(f)^\dagger = \mathrm{Tr}(f^\dagger)$.

- (We can also construct sets and partial injective functions as *fixed points*, though we're not going to worry about that here.)

Aspects of the semantics

Basic principles of denotational semantics

To produce a denotational semantics for a language, we generally need to

- Construct an object Σ, the *semantic domain* (here: the set of *states*).
- For each syntactic class (here: expressions, patterns, predicates, commands, procedures, and programs), construct a *denotation* $[\![t]\!]$ of each term t of that syntactic class as a morphism (here: partial injective function).

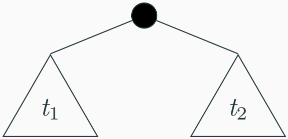
The semantic domain

To construct the set of states, we first need to consider *values*.

Values in `RWHILE` with procedures are binary trees with *symbols* (over some fixed but unspecified alphabet) as external nodes. Assume that a set Λ^* of symbols is given.

Binary trees $\mathbb{V}$ can then be constructed inductively as:

- If $\overline{s} \in \Lambda^*$, $\overline{s} \in \mathbb{V}$ (base case), and
- if $t_1, t_2 \in \mathbb{V}$ then $t_1 \bullet t_2 \in \mathbb{V}$ (inductive case).



The set of states Σ is then constructed as (finitely supported) *colists* over Λ^*.

In other words, states σ are streams $(v_1, v_2, \ldots)$ of values such that only finitely many v_i are non-$\overline{nil}$.

Intuition: Assign to each variable x (of which there are denumerably many) a *distinct* natural number n. A state $\sigma = (v_1, v_2, \ldots)$ then specifies precisely the *contents* of each variable.

For this reason, we will write x_i for the variable corresponding to the i'th component of a state.

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

Intuition: An expression takes a state and extracts a value from it, returning also the original state (for reversibility).

$$[\![e]\!]_{\text{exp}}(\sigma) = (\sigma, [\![e]\!]^{\sigma}_{\text{exp}})$$

$$\begin{aligned}
[\![x_i]\!]^{\sigma}_{\text{exp}} &= v_i \qquad \text{where } \sigma = (v_1, v_2, \dots) \\
[\![e_1 . e_2]\!]^{\sigma}_{\text{exp}} &= [\![e_1]\!]^{\sigma}_{\text{exp}} \bullet [\![e_2]\!]^{\sigma}_{\text{exp}} \\
[\![hd(e_1)]\!]^{\sigma}_{\text{exp}} &= \begin{cases} v_1 & \text{if } [\![e_1]\!]^{\sigma}_{\text{exp}} = v_1 \bullet v_2 \\ \uparrow & \text{otherwise} \end{cases}
\end{aligned}$$

Patterns

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

Intuition: A pattern extracts a *value* from a *state*, returning the *residual state* as a byproduct (for reversibility).

$$\begin{aligned}
[\![x_i]\!]_{\text{pat}}(\sigma) &= (v_1, v_2, \ldots, v_{i-1}, \overline{nil}, v_{i+1}, \ldots, v_i) \quad \text{where } \sigma = (v_1, v_2, \ldots) \\
[\![q_1 \cdot q_2]\!]_{\text{pat}}(\sigma) &= (\sigma'', v_1 \bullet v_2) \quad \text{where } (\sigma', v_1) = [\![q_1]\!]_{\text{pat}}(\sigma) \\
&\qquad\qquad\qquad\qquad\qquad\quad (\sigma'', v_2) = [\![q_2]\!]_{\text{pat}}(\sigma'),
\end{aligned}$$

Note that we're only worrying about right-patterns here. This is because left patterns are their formal duals, i.e., the corresponding left-pattern for $[\![q]\!]_{\text{pat}}$ is precisely $[\![q]\!]^{\dagger}_{\text{pat}}$.

Predicates

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

Intuition: A predicate *directs control flow* depending on its truth or falsehood in a given state.

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

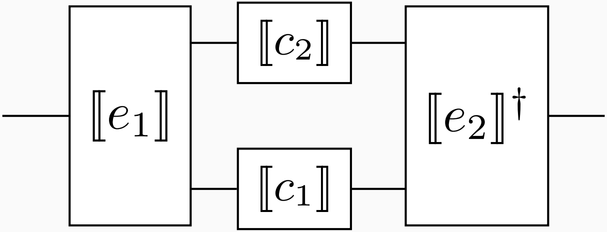
That is, a predicate sends control flow to the left if e is true (i.e., evaluates to a non-$\overline{nil}$ value) in the given state, and to the right otherwise.

Commands

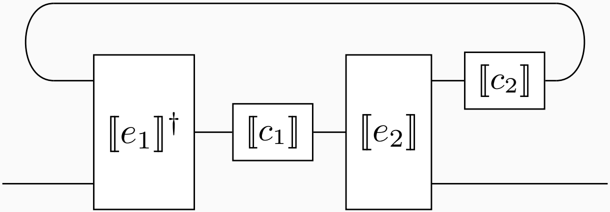
$$\Sigma \xrightarrow{[\![c]\!]_{\text{cmd}}} \Sigma$$

Intuition: Commands are *state transformations*.

$$[\![c_1; c_2]\!]_{\text{cmd}} = [\![c_2]\!]_{\text{cmd}} \circ [\![c_1]\!]_{\text{cmd}}$$
$$[\![q_1 \Leftarrow q_2]\!]_{\text{cmd}} = [\![q_1]\!]^{\dagger}_{\text{pat}} \circ [\![q_2]\!]_{\text{pat}}$$
$$[\![\mathit{if}\ e_1\ \mathit{then}\ c_1\ \mathit{else}\ c_2\ \mathit{fi}\ e_2]\!]_{\text{cmd}} = [\![e_2]\!]^{\dagger}_{\text{pred}} \circ ([\![c_1]\!]_{\text{cmd}} \oplus [\![c_2]\!]_{\text{cmd}}) \circ [\![e_1]\!]_{\text{pred}}$$
$$[\![\mathit{from}\ e_1\ \mathit{do}\ c_1\ \mathit{loop}\ c_2\ \mathit{until}\ e_2]\!]_{\text{cmd}} = \mathrm{Tr}(([\![c_2]\!]_{\text{cmd}} \oplus \mathrm{id}_{\Sigma}) \circ [\![e_2]\!]_{\text{pred}} \circ [\![c_1]\!]_{\text{cmd}} \circ [\![e_1]\!]^{\dagger}_{\text{pred}})$$


⟦c_2⟧
⟦e_1⟧
⟦e_2⟧†
⟦c_1⟧

Loops



Concluding remarks

Concluding remarks

Denotational semantics in **PInj**

- are intrinsically reversible,
- independent of concrete languages and paradigms, and
- allow the language designer to exploit *dualities* already present in the semantics.

Didn't get to all the details in this talk, so please see paper or ask me if you're interested.