Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 336 (2018) 155–171

www.elsevier.com/locate/entcs

A Categorical Foundation for Structured Reversible Flowchart Languages

Robert Glück[2] Robin Kaarsgaard[3]

DIKU, Department of Computer Science, University of Copenhagen

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, as ordinary structured flowcharts allow for conventional languages. 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 *decision maps* to give a reversible representation of predicates and assertions. This leads to a categorical semantics for structured reversible flowcharts, from which we show that a program inverter can be extracted. Finally, we exemplify our approach by the development of a small structured reversible flowchart language, use our framework to both straightforwardly give it semantics and derive fundamental theorems about it, and discuss further applications of decisions in reversible programming.

Keywords: Reversible computing, flowchart languages, structured programming, denotational semantics, category theory

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.*, [23,22,25]), new programming models and techniques (*e.g.*, [27,11,19]), and the enhancement of software with reversibility (*e.g.*, [5]).

[1] The authors acknowledge the support given by *COST Action IC1405 Reversible computation: Extending horizons of computing.* We also thank the anonymous reviewers for their thoughtful and detailed comments.

[2] Email: glueck@acm.org

[3] Email: robin@di.ku.dk

https://doi.org/10.1016/j.entcs.2018.03.021

The semantics of reversible programming languages are usually formalized using traditional metalanguages such as structural operational semantics or denotational semantics based on complete partial orders. However, these are geared towards the definition of conventional programming languages. The fundamental properties of a reversible language are not naturally captured by these metalanguages and are to be shown individually for each semantic definition, such as the required backward determinism and the invertibility of object language programs.

This paper aims at providing a new categorical foundation specifically for formalizing reversible programming languages, in particular the semantics of reversible structured flowchart languages [24], which are the reversible counterpart of the structured programming languages used today. This formalization is based on join inverse categories with a developed notion of *extensivity* for inverse categories, which gives rise to natural representations of predicates and assertions, and consequently to models of reversible structured flowcharts. The goal is to provide a framework for modelling these languages, such that the reversible semantic properties of the object language are naturally ensured by the meta language.

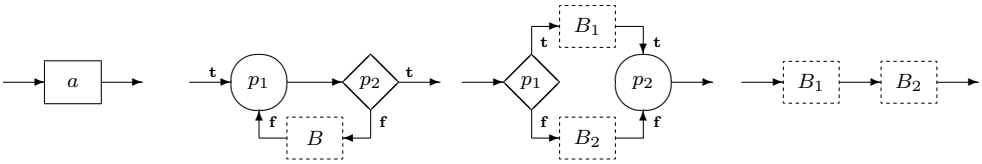
The semantic framework we are going to present in this paper covers the reversible structured languages regardless of their concrete formation, such as atomic operations, elementary predicates, and value domains. Reversible programming languages that are instances of this computation model include the imperative language Janus [27] without recursion, and the while languages `R-WHILE` and `R-CORE` with dynamic data structures [12,13]. Further, unstructured reversible flowchart languages, such as reversible assembly languages with jumps [9,2], can be transformed into structured ones thanks to the structured reversible program theorem [24].

Overview: In Section 2, we give an introduction to structured reversible flowchart languages, while Section 3 describes the restriction and inverse category theory used as backdrop in later sections. In Section 4, we warm up by developing a notion of extensivity for inverse categories, based on extensive restriction categories and its associated concept of *decisions*. Then, in Section 5, we put it all to use by showing how decisions may be used to model predicates and ultimately also reversible flowcharts, and use this to extract a program inverter. In Section 6, we develop a small language to exemplify our framework, and discuss other applications in reversible programming. Section 7 offers some concluding remarks.

2 Reversible structured flowcharts

Structured reversible flowcharts naturally model the control flow behavior of reversible (imperative) programming languages in a simple diagrammatic representation, as classical flowcharts do for conventional languages. A crucial difference is that atomic steps are limited to *partial injective functions* and they require an additional *assertion*, an explicit orthogonalizing condition, at join points in the control flow.

A structured reversible flowchart F is built from four blocks (Fig. 1): An *atomic step* that performs an elementary operation on a domain X specified by a partial injective function $a : X \rightharpoonup X$; a *while loop* over a block B with entry assertion



(a) Step (b) While loop (c) Selection (d) Sequence

Fig. 1. Structured reversible flowcharts.

$p_1 : X \to Bool$ and exit test $p_2 : X \to Bool$; a *selection* of block B_1 or B_2 with entry test $p_1 : X \to Bool$ and exit assertion $p_2 : X \to Bool$; and a *sequence* of blocks B_1 and B_2.

A structured reversible flowchart F consists of one main block. Blocks have unique entry and exit points, and can be nested any number of times to form more complex flowcharts. The interpretation of F consists of a given domain X (typically, a store) and a finite set of partial injective functions a and predicates $p : X \to Bool$. Computation starts at the entry point of F in an initial x_0 (the input), proceeds sequentially through the edges of F, and ends at the exit point of F in a final x_n (the output), if F is defined on the given input.

The assertion p_1 in a reversible while loop (marked by the circle) is a new flowchart operator: the predicate p_1 must be **true** when the control flow reaches the assertion along the **t**-edge, and **false** when it reaches the assertion along the **f**-edge; otherwise, the loop is undefined. The test p_2 (marked by a diamond) has the usual semantics. This means that B in a loop is repeated as long as p_1 and p_2 are **false**.

The selection has an assertion p_2, which must be **true** when the control flow reaches the assertion from B_1, and **false** when the control flow reaches the assertion from B_2; otherwise, the selection is undefined. As usual, the test p_1 selects B_1 or B_2. The assertion makes the selection reversible.

Despite their simplicity, reversible structured flowcharts are *reversibly universal* [1], which means that they are computationally as powerful as any reversible programming language can be. Given a suitable domain X for finite sets of atomic operations and predicates, there exists, for every injective computable function $f : X \to Y$, a reversible flowchart F that computes f.

Reversible structured flowcharts (Fig. 1) have a straightforward representation as program texts defined by the grammar

$$B ::= a \mid \texttt{from}\ p\ \texttt{loop}\ B\ \texttt{until}\ p \mid \texttt{if}\ p\ \texttt{then}\ B\ \texttt{else}\ B\ \texttt{fi}\ p \mid B\ \texttt{;}\ B\ .$$

Reversible structured flowcharts defined above corresponds to the reversible language `R-WHILE` [12], but their value domain, atomic functions and predicates are unspecified. As a minimum, a reversible flowchart needs blocks (a,b,d) because selection (c) can be simulated by combining while loops that conditionally skip the body block or execute it once. `R-CORE` [13] is an example of such a minimal language.

3 Restriction and inverse categories

The following section contains the background on restriction and inverse category theory necessary for our later developments. Unless otherwise specified, the definitions and results presented in this section can be found in introductory texts on the subject (*e.g.*, [10,14,6,7,8]).

Restriction categories [6,7,8] axiomatize categories of partial maps. This is done by assigning to each morphism f a *restriction idempotent* $\overline{f}$, which we think of as a partial identity defined precisely where f is. Formally, restriction categories are defined as follows.

Definition 3.1 A *restriction category* is a category $\mathscr{C}$ equipped with a combinator mapping each morphism $A \xrightarrow{f} B$ to a morphism $A \xrightarrow{\overline{f}} A$ satisfying

(i) $f \circ \overline{f} = f$,
(ii) $\overline{g} \circ \overline{f} = \overline{f} \circ \overline{g}$,
(iii) $\overline{f \circ \overline{g}} = \overline{f} \circ \overline{g}$, and
(iv) $\overline{g} \circ f = f \circ \overline{g \circ f}$

for all suitable g.

As an example, the category **Pfn** of sets and partial functions is a restriction category, with $\overline{f}(x) = x$ if f is defined at x, and undefined otherwise. Note that being a restriction category is a structure, not a property; a category may be a restriction category in several different ways (*e.g.*, assigning $\overline{f} = \mathrm{id}$ for each morphism f gives a trivial restriction structure to any category).

In restriction categories, we say that a morphism $A \xrightarrow{f} B$ is *total* if $\overline{f} = \mathrm{id}_A$, and a *partial isomorphism* if there exists a (necessarily unique) *partial inverse* $B \xrightarrow{f^\dagger} A$ such that $f^\dagger \circ f = \overline{f}$ and $f \circ f^\dagger = \overline{f^\dagger}$. Isomorphisms are then simply the total partial isomorphisms with total partial inverses. An inverse category can then be defined as a special kind of restriction category [4].

Definition 3.2 An *inverse category* is a restriction category where each morphism is a partial isomorphism.

Every restriction category $\mathscr{C}$ gives rise to an inverse category $\mathrm{Inv}(\mathscr{C})$, which has as objects all objects of $\mathscr{C}$, and as morphisms all of the partial isomorphisms of $\mathscr{C}$. As such, since partial isomorphisms in **Pfn** are partial injective functions, a canonical example of an inverse category is the category $\mathrm{Inv}(\mathbf{Pfn}) \cong \mathbf{PInj}$ of sets and partial injective functions.

Since each morphism in an inverse category has a unique partial inverse, as also suggested by our notation this makes inverse categories canonically *dagger categories* [20], in the sense that they come equipped with a contravariant endofunctor $(-)^\dagger$ satisfying $f = f^{\dagger\dagger}$ and $\mathrm{id}_A^\dagger = \mathrm{id}_A$ for each morphism f and object A.

Given two restriction categories $\mathscr{C}$ and $\mathscr{D}$, the well-behaved functors between them are *restriction functors*, *i.e.*, functors F satisfying $F(\overline{f}) = \overline{F(f)}$. Analogous to how

[4] This is a rather modern definition due to [6]. Originally, inverse categories were defined as the categorical extensions of inverse semigroups; see [18].

regular semigroup homomorphisms preserve partial inverses in inverse semigroups, when $\mathscr{C}$ and $\mathscr{D}$ are inverse categories, all functors between them are restriction functors; specifically they preserve the canonical dagger, *i.e.*, $F(f^\dagger) = F(f)^\dagger$.

3.1 Partial order enrichment and joins

A consequence of how restriction (and inverse) categories are defined is that hom sets $\mathscr{C}(A, B)$ may be equipped with a partial order given by $f \leq g$ iff $g \circ \overline{f} = f$ (this extends to an enrichment in the category of partial orders and monotone functions). Intuitively, this states that f is below g iff g behaves exactly like f when restricted to the points where f is defined. A sufficient condition for each $\mathscr{C}(A, B)$ to have a least element is that $\mathscr{C}$ has a *restriction zero*; a zero object 0 in the usual sense which additionally satisfies $A \xrightarrow{0_{A,A}} A = A \xrightarrow{\overline{0_{A,A}}} A$ for each endo-zero map $0_{A,A}$.

One may now wonder when $\mathscr{C}(A, B)$ has joins as a partial order. Unfortunately, $\mathscr{C}(A, B)$ has joins of all morphisms only in very degenerate cases. However, if instead of considering arbitrary joins we consider joins of maps that are somehow compatible, this becomes much more viable.

Definition 3.3 In a restriction category, say that parallel maps f and g are *disjoint* iff $f \circ \overline{g} = 0$; and *compatible* iff $f \circ \overline{g} = g \circ \overline{f}$.

It can be shown that disjointness implies compatibility, as disjointness is expectedly symmetric. Further, we may extend this to say that a set of parallel morphisms is disjoint iff each pair of morphisms is disjoint, and likewise for compatibility. This gives suitable notions of *join restriction categories*.

Definition 3.4 A restriction category $\mathscr{C}$ has compatible (disjoint) joins if it has a restriction zero, and satisfies that for each compatible (disjoint) subset S of any hom set $\mathscr{C}(A, B)$, there exists a morphism $\bigvee_{s \in S} s$ such that

(i) $s \leq \bigvee_{s \in S} s$ for all $s \in S$, and $s \leq t$ for all $s \in S$ implies $\bigvee_{s \in S} s \leq t$;
(ii) $\overline{\bigvee_{s \in S} s} = \bigvee_{s \in S} \overline{s}$;
(iii) $f \circ \left(\bigvee_{s \in S} s\right) = \bigvee_{s \in S} (f \circ s)$ for all $f : B \to X$; and
(iv) $\left(\bigvee_{s \in S} s\right) \circ g = \bigvee_{s \in S} (s \circ g)$ for all $g : Y \to A$.

For inverse categories, the situation is a bit more tricky, as the join of two compatible partial isomorphisms may not be a partial isomorphism. To ensure this, we need stronger relations:

Definition 3.5 In an inverse category, say that parallel maps f and g are *disjoint* iff $f \circ \overline{g} = 0$ and $f^\dagger \circ \overline{g^\dagger} = 0$; and *compatible* iff $f \circ \overline{g} = g \circ \overline{f}$ and $f^\dagger \circ \overline{g^\dagger} = g^\dagger \circ \overline{f^\dagger}$.

We may now extend this to notions of disjoint sets and compatible sets of morphisms in inverse categories as before. This finally gives notions of *join inverse categories*:

Definition 3.6 An inverse category $\mathscr{C}$ has compatible (disjoint) joins if it has a restriction zero and satisfies that for all compatible (disjoint) subsets S of all hom

sets $\mathscr{C}(A,B)$, there exists a morphism $\vee_{s\in S} s$ satisfying (i) – (iv) of Definition 3.4.

A functor F between restriction (or inverse) categories with joins is said to be join-preserving when $F(\bigvee_{s\in S} s) = \bigvee_{s\in S} F(s)$.

3.2 Restriction coproducts, extensivity, and related concepts

While a restriction category may very well have coproducts, these are ultimately only well-behaved when all coproduct injections are total; if this is the case, we say that the restriction category has *restriction coproducts*. If a restriction category has all finite restriction coproducts, it also has a restriction zero serving as unit.

In [8], it is shown that the existence of certain maps, called *decisions*, in a restriction category $\mathscr{C}$ with restriction coproducts leads to the subcategory $\mathrm{Total}(\mathscr{C})$ of total maps being extensive (in the sense of, *e.g.*, [4]). This leads to the definition of an *extensive restriction category* [5].

Definition 3.7 A restriction category is said to be *extensive* (as a restriction category) if it has restriction coproducts and a restriction zero, and for each map $A \xrightarrow{f} B + C$ there is a unique *decision* $A \xrightarrow{\langle f\rangle} A + A$ satisfying

(D.1) $\nabla \circ \langle f\rangle = \overline{f}$ and **(D.2)** $(f+f)\circ\langle f\rangle = (\kappa_1+\kappa_2)\circ f$.

In the above, ∇ denotes the codiagonal $[\mathrm{id},\mathrm{id}]$. A consequence of these axioms is that each decision is a partial isomorphism; one can show that $\langle f\rangle$ must be partial inverse to $[\overline{\kappa_1^\dagger \circ f}, \overline{\kappa_2^\dagger \circ f}]$ (see [8]). Further, when a restriction category with restriction coproducts has finite joins, it is also extensive with $\langle f\rangle = \kappa_1 \circ \overline{\kappa_1^\dagger \circ f} \vee \kappa_2 \circ \overline{\kappa_2^\dagger \circ f}$. As an example, **Pfn** is extensive with $A \xrightarrow{\langle f\rangle} A + A$ for $A \xrightarrow{f} B + C$ given by

$$\langle f\rangle(x) = \begin{cases} \kappa_1(x) & \text{if } f(x) = \kappa_1(y) \text{ for some } y \in B \\ \kappa_2(x) & \text{if } f(x) = \kappa_2(z) \text{ for some } z \in C \\ \text{undefined} & \text{if } f(x) \text{ is undefined} \end{cases}.$$

While inverse categories only have coproducts (much less restriction coproducts) in very degenerate cases (see [10]), they may very well be equipped with a more general sum-like symmetric monoidal tensor, a disjointness tensor.

Definition 3.8 A *disjointness tensor* on a restriction category is a symmetric monoidal restriction functor $- \oplus -$ satisfying that its unit is the restriction zero, and that the canonical maps

$$\amalg_1 = A \xrightarrow{\rho^{-1}} A \oplus 0 \xrightarrow{\mathrm{id}\oplus 0} A \oplus B \qquad \amalg_2 = B \xrightarrow{\lambda^{-1}} 0 \oplus B \xrightarrow{0\oplus\mathrm{id}} A \oplus B$$

[5] The name is admittedly mildly confusing, as an extensive restriction category is not extensive in the usual sense. Nevertheless, we stay with the established terminology.

are jointly epic, where ρ respectively λ is the left respectively right unitor of the monoidal functor $- \oplus -$.

It can be straightforwardly shown that any restriction coproduct gives rise to a disjointness tensor. A useful interaction between compatible joins and a join-preserving disjointness tensor in inverse categories was shown in [3,17], namely that it leads to a †-trace (in the sense of [16,21]):

Proposition 3.9 *Let $\mathscr{C}$ be an inverse category with (at least countable) compatible joins and a join-preserving disjointness tensor. Then $\mathscr{C}$ has a trace operator given by*

$$\mathrm{Tr}^U_{A,B}(f) = f_{11} \vee \bigvee_{n \in \omega} f_{21} \circ f_{22} \circ f_{12}$$

satisfying $\mathrm{Tr}^U_{A,B}(f)^\dagger = \mathrm{Tr}^U_{A,B}(f^\dagger)$, where $f_{ij} = \amalg_j^\dagger \circ f \circ \amalg_i$.

4 Extensivity of inverse categories

As discussed earlier, extensivity of restriction categories hinges on the existence of certain partial isomorphisms – decisions – yet their axiomatization relies on the presence of a map that is not a partial isomorphism, the codiagonal.

In this section, we tweak the axiomatization of extensivity of restriction categories to one that is equivalent, but additionally transports more easily to inverse categories. We then give a definition of extensitivity for inverse categories, from which it follows that $\mathrm{Inv}(\mathscr{C})$ is an extensive inverse category when $\mathscr{C}$ is an extensive restriction category.

Recall that decisions satisfy the following two axioms:

(D.1) $\nabla \circ \langle f \rangle = \overline{f}$ and **(D.2)** $(f + f) \circ \langle f \rangle = (\kappa_1 + \kappa_2) \circ f$

As mentioned previously, an immediate problem with this is the reliance on the codiagonal. However, intuitively, what **(D.1)** states is simply that the decision $\langle f \rangle$ cannot do anything besides to tag its inputs appropriately. Using a disjoint join, we reformulate this axiom to the following:

(D'.1) $(\kappa_1^\dagger \circ \langle f \rangle) \vee (\kappa_2^\dagger \circ \langle f \rangle) = \overline{f}$

Note that this axiom also subtly states that disjoint joins of the given form always exist. Say that a restriction category is *pre-extensive* if it has restriction coproducts, a restriction zero, and a combinator mapping each map $A \xrightarrow{f} B + C$ to a *pre-decision* $A \xrightarrow{\langle f \rangle} A + A$ (with no additional requirements). We can then show the following:

Theorem 4.1 *Let $\mathscr{C}$ be a pre-extensive restriction category. The following are equivalent:*

(i) $\mathscr{C}$ is an extensive restriction category.

*(ii) Every pre-decision of $\mathscr{C}$ satisfies **(D.1)** and **(D.2)**.*

*(iii) Every pre-decision of $\mathscr{C}$ satisfies **(D'.1)** and **(D.2)**.*

Another subtle consequence of our amended first rule is that $\kappa_1^\dagger \circ \langle f \rangle$ is its own restriction idempotent (and likewise for $\kappa_2^\dagger$) since $\kappa_1^\dagger \circ \langle f \rangle \leq (\kappa_1^\dagger \circ \langle f \rangle) \vee (\kappa_2^\dagger \circ \langle f \rangle) = \overline{f} \leq \mathrm{id}$, as the maps below identity are precisely the restriction idempotents.

Our next snag in transporting this definition to inverse categories has to do with the restriction coproducts themselves, as it is observed in [10] that any inverse category with restriction coproducts is a preorder. Intuitively, the problem is not that unicity of coproduct maps cannot be guaranteed in non-preorder inverse categories, but rather that the coproduct map $A + B \xrightarrow{[f,g]} C$ in a restriction category is not guaranteed to be a partial isomorphism when f and g are.

For this reason, we will consider the more general disjointness tensor for sum-like constructions rather than full-on restriction coproducts, as inverse categories may very well have a disjointness tensor without it leading to immediate degeneracy. Notably, **PInj** has a disjointness tensor, constructed on objects as the disjoint union of sets (precisely as the restriction coproduct in **Pfn**, but without the requirement of a universal mapping property). This leads us to the following definition:

Definition 4.2 An inverse category with a disjointness tensor is said to be *extensive* when each map $A \xrightarrow{f} B \oplus C$ has a unique decision $A \xrightarrow{\langle f \rangle} A \oplus A$ satisfying

(D'.1) $(\amalg_1^\dagger \circ \langle f \rangle) \vee (\amalg_2^\dagger \circ \langle f \rangle) = \overline{f}$

(D'.2) $(f \oplus f) \circ \langle f \rangle = (\amalg_1 \oplus \amalg_2) \circ f$.

As an example, **PInj** is an extensive inverse category with the unique decision $A \xrightarrow{\langle f \rangle} A \oplus A$ for a partial injection $A \xrightarrow{f} B \oplus C$ given by

$$\langle f \rangle(x) = \begin{cases} \amalg_1(x) & \text{if } f(x) = \amalg_1(y) \text{ for some } y \in B \\ \amalg_2(x) & \text{if } f(x) = \amalg_2(z) \text{ for some } z \in C \\ \text{undefined} & \text{if } f(x) \text{ is undefined} \end{cases} .$$

Aside from a shift from coproduct injections to the quasi-injections of the disjointness tensor, a subtle change here is the notion of join. That is, for restriction categories with disjoint joins, any pair of maps f, g with $f \circ \overline{g} = 0$ has a join – but for inverse categories, we additionally require that their *inverses* are disjoint as well, *i.e.*, that $f^\dagger \circ \overline{g^\dagger} = 0$, for the join to exist. In this case, however, there is no difference between the two. As previously discussed, a direct consequence of this axiom is that each $\amalg_i^\dagger \circ \langle f \rangle$ must be its own restriction idempotent. Since restriction idempotents are self-adjoint (*i.e.*, satisfy $f = f^\dagger$), they are disjoint iff their inverses are disjoint.

Since restriction coproducts give rise to a disjointness tensor, we may straightforwardly show the following theorem.

Theorem 4.3 *When $\mathscr{C}$ is an extensive restriction category,* $\mathrm{Inv}(\mathscr{C})$ *is an extensive inverse category.*

Further, constructing the decision $\langle f \rangle$ as $(\amalg_1 \circ \overline{\amalg_1^\dagger \circ f}) \vee (\amalg_2 \circ \overline{\amalg_2^\dagger \circ f})$ (*i.e.*, mirroring the construction of decisions in restriction categories with disjoint joins),

we may show the following.

Theorem 4.4 *Let $\mathscr{C}$ be an inverse category with a disjointness tensor, a restriction zero, and finite disjoint joins. Then $\mathscr{C}$ is extensive as an inverse category.*

5 Modelling structured reversible flowcharts

In the following, let $\mathscr{C}$ be an inverse category with (at least countable) compatible joins and a join-preserving disjointness tensor. As disjoint joins are compatible, it follows that $\mathscr{C}$ is an extensive inverse category with a (uniform) †-trace operator.

In this section, we will show how this framework can be used model reversible structured flowchart languages. First, we will show how decisions in extensive inverse categories can be used to model predicates, and how this representation extends to give very natural semantics to reversible flowcharts corresponding to conditionals and loops. Then we will use the "internal program inverter" given by the canonical dagger functor on $\mathscr{C}$ to extract a program inverter for reversible flowcharts.

5.1 Predicates as decisions

In suitably equipped categories, one naturally considers predicates on an object A as given by maps $A \to 1 + 1$. In inverse categories, however, the mere idea of a predicate as a map of the form $A \to 1 \oplus 1$ is problematic, as only very degenerate maps of this form are partial isomorphisms. In the following, we show how decisions give rise to an unconventional yet ultimately useful representation of predicates. To our knowledge this representation is novel, motivated here by the necessity to model predicates in a reversible fashion, as decisions are always partial isomorphisms.

The simplest useful predicates are the predicates that are always true respectively always false. By convention, we represent these by the left respectively right injection (which are both their own decisions),

$$\begin{aligned} [\![\texttt{true}]\!] &= \amalg_1 \\ [\![\texttt{false}]\!] &= \amalg_2. \end{aligned}$$

Semantically, we may think of decisions as a separation of an object A into *witnesses* and *counterexamples* of the predicate it represents. In a certain sense, the axioms of decisions say that there is nothing more to a decision than how it behaves when postcomposed with $\amalg_1^\dagger$ or $\amalg_2^\dagger$. As such, given the convention above, we think of $\amalg_1^\dagger \circ \langle p \rangle$ as the witnesses of the predicate represented by the decision $\langle p \rangle$, and $\amalg_2^\dagger \circ \langle p \rangle$ as its counterexamples.

With this in mind, we turn to boolean combinators. The negation of a predicate-as-a-decision must simply swap witnesses for counterexamples (and vice versa). In other words, we obtain the negation of a decision by postcomposing with the commutator γ of the disjointness tensor,

$$[\![\texttt{not}\ p]\!] = \gamma \circ [\![p]\!]\,.$$

With this, it is straightforward to verify that, *e.g.*, $[\![\texttt{not true}]\!] = [\![\texttt{false}]\!]$, as

$$[\![\texttt{not true}]\!] = \gamma \circ \Pi_1 = \gamma \circ \mathrm{id} \oplus 0 \circ \rho^{-1} = 0 \oplus \mathrm{id} \circ \gamma \circ \rho^{-1} = 0 \oplus \mathrm{id} \circ \lambda^{-1} = \Pi_2 = [\![\texttt{false}]\!]\,.$$

For conjunction, we exploit that our category has (specifically) finite disjoint joins, and define the conjunction of predicates-as-decisions $[\![p]\!]$ and $[\![q]\!]$ by

$$[\![p \texttt{ and } q]\!] = (\Pi_1 \circ \overline{\Pi_1^\dagger \circ [\![p]\!]} \circ \overline{\Pi_1^\dagger \circ [\![q]\!]}) \vee (\Pi_2 \circ (\overline{\Pi_2^\dagger \circ [\![p]\!]} \vee \overline{\Pi_2^\dagger \circ [\![q]\!]})).$$

The intuition behind this definition is that the witnesses of a conjunction of predicates is given by the meet of the witnesses of the each predicate, while the counterexamples of a conjunction of predicates is the join of the counterexamples of each predicate. Noting that the meet of two restriction idempotents is given by their composition, this is precisely what this definition states. Similarly we define the disjunction of $[\![p]\!]$ and $[\![q]\!]$ by

$$[\![p \texttt{ or } q]\!] = (\Pi_1 \circ (\overline{\Pi_1^\dagger \circ [\![p]\!]} \vee \overline{\Pi_1^\dagger \circ [\![q]\!]})) \vee (\Pi_2 \circ (\overline{\Pi_2^\dagger \circ [\![p]\!]} \circ \overline{\Pi_2^\dagger \circ [\![q]\!]})),$$

as $[\![p \texttt{ or } q]\!]$ then has as witnesses the join of the witnesses of $[\![p]\!]$ and $[\![q]\!]$, and as counterexamples the meet of the counterexamples of $[\![p]\!]$ and $[\![q]\!]$. With these definitions, it can be shown that, *e.g.*, the De Morgan laws are satisfied.

That all of these are indeed decisions can be shown straightforwardly, as summarized in the following closure theorem.

Theorem 5.1 *Decisions in $\mathscr{C}$ are closed under Boolean negation, conjunction, and disjunction.*

5.2 Reversible structured flowcharts, categorically

To give a categorical account of structured reversible flowchart languages, we assume the existence of a suitable distinguished object Σ of stores, which we think of as the *domain of computation*, such that we may give denotations to structured reversible flowcharts as morphisms $\Sigma \to \Sigma$.

Since atomic steps (corresponding to elementary operations, *e.g.*, store updates) may vary from language to language, we assume that each such atomic step in our language has a denotation as a morphism $\Sigma \to \Sigma$. In the realm of reversible flowcharts, these atomic steps are required to be partial injective functions; here, we abstract this to require that their denotation is a partial isomorphism (though this is a trivial requirement in inverse categories).

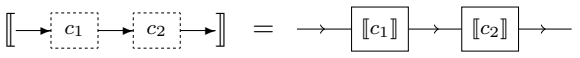
Likewise, elementary predicates (*e.g.*, comparison of values in a store) may vary from language to language, so we assume that such elementary predicates have denotations as well as decisions $\Sigma \to \Sigma \oplus \Sigma$. If necessary (as is the case for Janus [27]), we may then close these elementary predicates under boolean combinations as discussed in the previous section.

To start, we note how sequencing of flowcharts may be modelled trivially by

means of composition, *i.e.*,

$$[\![c_1 \ ; \ c_2]\!] = [\![c_2]\!] \circ [\![c_1]\!]$$

or, using the diagrammatic notation of flowcharts and the string diagrams for monoidal categories in the style of [21] (read left-to-right and bottom-to-top),

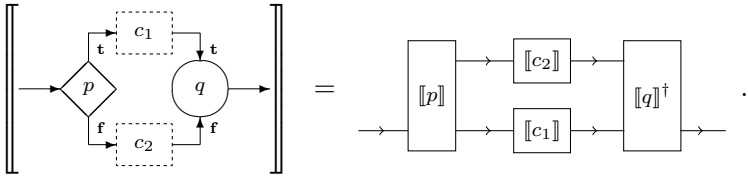


To extend this elementary model to one that additionally models reversible conditionals, we observe that the partial inverse to a decision is precisely its corresponding assertion. Intuitively, a decision separates an object into witnesses (in the first component) and counterexamples (in the second). As such, the partial inverse to a decision must be defined only on witnesses in the first component, and only on counterexamples in the second.

With this in mind, we achieve a denotation of reversible conditionals as

$$[\![\texttt{if}\ p\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \texttt{fi}\ q]\!] = [\![q]\!]^\dagger \circ [\![c_1]\!] \oplus [\![c_2]\!] \circ [\![p]\!]$$

or, as diagrams



It is straightforward to see that this has the usual operational behavior of reversible conditionals. For example, when $[\![p]\!] = [\![q]\!] = [\![\texttt{true}]\!]$ we get

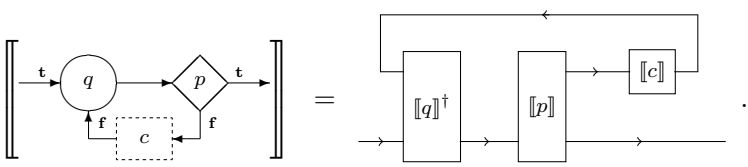
$$\begin{aligned}[\![\texttt{if}\ p\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \texttt{fi}\ q]\!] &= [\![q]\!]^\dagger \circ [\![c_1]\!] \oplus [\![c_2]\!] \circ [\![p]\!] \\ &= [\![\texttt{true}]\!]^\dagger \circ [\![c_1]\!] \oplus [\![c_2]\!] \circ [\![\texttt{true}]\!] \\ &= \Pi_1^\dagger \circ [\![c_1]\!] \oplus [\![c_2]\!] \circ \Pi_1 = [\![c_1]\!]\end{aligned}$$

and analogously for the symmetric case.

For reversible loops, we use the †-trace operator to obtain the denotation

$$[\![\texttt{from}\ q\ \texttt{loop}\ c\ \texttt{until}\ p]\!] = \mathrm{Tr}^{\Sigma}_{\Sigma,\Sigma}(\mathrm{id}_\Sigma \oplus [\![c]\!] \circ [\![p]\!] \circ [\![q]\!]^\dagger)$$

or diagrammatically



That this has the desired operational behavior follows from the fact that the †-trace operator is canonically constructed in join inverse categories as

$$\mathrm{Tr}^U_{X,Y}(f) = f_{11} \vee \bigvee_{n\in\omega} f_{21} \circ f_{22}^n \circ f_{12} \ .$$

Recall that $f_{ij} = \amalg_j^\dagger \circ f \circ \amalg_i$. As such, for our loop construct defined above, the f_{11}-cases correpond to cases where a given state bypasses the loop entirely; $f_{21} \circ f_{12}$ (that is, for $n = 0$) to cases where exactly one iteration is performed by a given state before exiting the loop; $f_{21} \circ f_{22} \circ f_{12}$ to cases where two iterations are performed before exiting; and so on. In this way, the given trace semantics contain all successive loop unrollings, as desired.

While it may seem like a small point, the mere existence of a categorical semantics in inverse categories for a reversible programming language has some immediate benefits. In particular, that a programming language is reversible can be rather complicated to show by means of operational semantics (see, *e.g.*, [27, Sec. 2.3]), yet it follows directly in our categorical semantics, as all morphisms in inverse categories have a unique partial inverse. Additionally, reversible loops are significantly easier to work with categorically as †-traces than operationally; the operational semantics of Janus [27, Fig. 4] demonstrate adequately just how difficult it can be to give an operational semantics for reversible loops.

5.3 Extracting a program inverter

A desirable syntactic property for reversible programming languages is to be closed under program inversion, in the sense that for each program p, there is another program $\mathcal{I}[p]$ such that $[\![\mathcal{I}[p]]\!] = [\![p]\!]^\dagger$. Janus, `R-WHILE`, and `R-CORE` [27,12,13] are all examples of reversible programming languages with this property. This is typically witnessed by a *program inverter*, that is, a procedure mapping the program text of a program to the program text of its inverse program [6].

Suppose that we are given a language where elementary operations are closed under program inversion (*i.e.*, where each elementary operation b has an inverse $\mathcal{I}[b]$ such that $[\![\mathcal{I}[b]]\!] = [\![b]\!]^\dagger$). We can extend this to a program inverter for reversible conditionals and loops as follows, by structural induction with the hypothesis that $[\![\mathcal{I}[c]]\!] = [\![c]\!]^\dagger$. Given some conditional statement `if` p `then` c_1 `else` c_2 `fi` q, we notice that

$$\begin{aligned}
[\![\texttt{if } p \texttt{ then } c_1 \texttt{ else } c_2 \texttt{ fi } q]\!]^\dagger &= ([\![q]\!]^\dagger \circ [\![c_1]\!] \oplus [\![c_2]\!] \circ [\![p]\!])^\dagger \\
&= [\![p]\!]^\dagger \circ [\![c_1]\!]^\dagger \oplus [\![c_2]\!]^\dagger \circ [\![q]\!]^{\dagger\dagger} \\
&= [\![p]\!]^\dagger \circ [\![c_1]\!]^\dagger \oplus [\![c_2]\!]^\dagger \circ [\![q]\!] \\
&= [\![p]\!]^\dagger \circ [\![\mathcal{I}[c_1]]\!] \oplus [\![\mathcal{I}[c_2]]\!] \circ [\![q]\!] \\
&= [\![\texttt{if } q \texttt{ then } \mathcal{I}[c_1] \texttt{ else } \mathcal{I}[c_2] \texttt{ fi } p]\!]
\end{aligned}$$

[6] While semantic inverses *are* unique, their program texts generally are not. As such, a programming language may have many different sound and complete program inverters, though they will all be equivalent up to program semantics.

which yields the inversion rule

$$\mathcal{I}[\texttt{if } p \texttt{ then } c_1 \texttt{ else } c_2 \texttt{ fi } q] = \texttt{if } q \texttt{ then } \mathcal{I}[c_1] \texttt{ else } \mathcal{I}[c_2] \texttt{ fi } p \ .$$

Fortunately, this is precisely the usual inversion rule for reversible conditionals (see, *e.g.*, [12,13]). For reversible loops, we have

$$\begin{aligned}
[\![\texttt{from } q \texttt{ loop } c \texttt{ until } p]\!]^\dagger &= \mathrm{Tr}^{\Sigma}_{\Sigma,\Sigma}(\mathrm{id}_\Sigma \oplus [\![c]\!] \circ [\![p]\!] \circ [\![q]\!]^\dagger)^\dagger \\
&= \mathrm{Tr}^{\Sigma}_{\Sigma,\Sigma}((\mathrm{id}_\Sigma \oplus [\![c]\!] \circ [\![p]\!] \circ [\![q]\!]^\dagger)^\dagger) \\
&= \mathrm{Tr}^{\Sigma}_{\Sigma,\Sigma}([\![q]\!] \circ [\![p]\!]^\dagger \circ \mathrm{id}_\Sigma \oplus [\![c]\!]^\dagger) \\
&= \mathrm{Tr}^{\Sigma}_{\Sigma,\Sigma}(\mathrm{id}_\Sigma \oplus [\![c]\!]^\dagger \circ [\![q]\!] \circ [\![p]\!]^\dagger) \\
&= \mathrm{Tr}^{\Sigma}_{\Sigma,\Sigma}(\mathrm{id}_\Sigma \oplus [\![\mathcal{I}[c]]\!] \circ [\![q]\!] \circ [\![p]\!]^\dagger) \\
&= [\![\texttt{from } p \texttt{ loop } \mathcal{I}[c] \texttt{ until } q]\!]
\end{aligned}$$

where the fact that it is a †-trace allows us to move the dagger inside the trace, and dinaturality of the trace in the second component allows us to move $\mathrm{id}_\Sigma \oplus [\![c]\!]^\dagger$ from the very right to the very left. This gives us the inversion rule

$$\mathcal{I}[\texttt{from } q \texttt{ loop } c \texttt{ until } p] = \texttt{from } p \texttt{ loop } \mathcal{I}[c] \texttt{ until } q$$

which matches the usual inversion rule for reversible loops [13]. We summarize this in the following theorem:

Theorem 5.2 *If a reversible structured flowchart language is syntactically closed under inversion of elementary operations, it is also closed under inversion of reversible conditionals and loops.*

6 Applications

In this section, we briefly cover some applications of the developed theory: We introduce a small reversible flowchart language and use the results from the previous sections to give it semantics, and discuss how decisions may be used as a programming technique to naturally represent predicates in a reversible functional language.

6.1 Example: A reversible flowchart language

Consider the following family of (neither particularly useful nor particularly useless) reversible flowchart languages for reversible computing with integer data, $\texttt{RINT}_k$. $\texttt{RINT}_k$ has precisely k variables available for storage, denoted $\texttt{x}_1$ through $\texttt{x}_k$ (of which $\texttt{x}_1$ is designated by convention as the input/output variable), and its only atomic operations are addition and subtraction of variables, as well as addition with a constant. Variables are used as elementary predicates, with zero designating truth and non-zero values all designating falsehood. For control structures we have

$$\Sigma = \mathbb{Z}^k$$
$$[\![\mathtt{x_i}]\!](a_1,\ldots,a_k) = \begin{cases} \amalg_1(a_1,\ldots,a_k) & \text{if } a_i = 0 \\ \amalg_2(a_1,\ldots,a_k) & \text{otherwise} \end{cases}$$
$$[\![\mathtt{x_i} \mathrel{+}= \mathtt{x_j}]\!](a_1,\ldots,a_k) = (a_1,\ldots,a_{i-1},a_i + a_j,\ldots,a_k)$$
$$[\![\mathtt{x_i} \mathrel{+}= \overline{n}]\!](a_1,\ldots,a_k) = (a_1,\ldots,a_{i-1},a_i + n,\ldots,a_k)$$
$$[\![\mathtt{x_i} \mathrel{-}= \mathtt{x_j}]\!](a_1,\ldots,a_k) = (a_1,\ldots,a_{i-1},a_i - a_j,\ldots,a_k)$$

Fig. 2. The object of stores and semantics of elementary operations and predicates of $\mathtt{RINT}_k$ in **PInj**.

reversible conditionals and loops, and sequencing as usual. This gives the syntax:

$$\begin{aligned} p &::= \mathtt{true} \mid \mathtt{false} \mid \mathtt{x_i} \mid p\ \mathtt{and}\ p \mid \mathtt{not}\ p && \text{(Tests)} \\ c &::= c\ ;\ c \mid \mathtt{x_i} \mathrel{+}= \mathtt{x_j} \mid \mathtt{x_i} \mathrel{-}= \mathtt{x_j} \mid \mathtt{x_i} \mathrel{+}= \overline{n} \\ &\quad \mid \mathtt{if}\ p\ \mathtt{then}\ c\ \mathtt{else}\ c\ \mathtt{fi}\ p \\ &\quad \mid \mathtt{from}\ p\ \mathtt{loop}\ c\ \mathtt{until}\ p && \text{(Commands)} \end{aligned}$$

Here, $\overline{n}$ is the syntactic representation of an integer n. In the cases for addition and subtraction, we impose the additional syntactic constraints that $1 \leq i \leq k$, $1 \leq j \leq k$, and $i \neq j$, the latter to guarantee reversibility. Subtraction by a constant is not included as it may be derived straightforwardly from addition with a constant. A program in $\mathtt{RINT}_k$ is then simply a command.

We may now give semantics to this language in our framework. For a concrete model, we select the category **PInj** of sets and partial injections, which is a join inverse category with a join-preserving disjointness tensor (given on objects by the disjoint union of sets), so it is extensive in the sense of Definition 4.2 by Theorem 4.4. By our developments previously in this section, to give a full semantics to $\mathtt{RINT}_k$ in **PInj**, it suffices to provide an object (*i.e.*, a set) of stores Σ, denotations of our three classes of elementary operations (addition by a variable, addition by a constant, and subtraction by a variable) as morphisms (*i.e.*, partial injective functions) $\Sigma \to \Sigma$, and denotations of our class of elementary predicates (here, testing whether a variable is zero or not) as decisions $\Sigma \to \Sigma \oplus \Sigma$. These are all shown in Fig. 2. It is uncomplicated to show that all of these are partial injective functions, and that the denotation of each predicate $[\![\mathtt{x_i}]\!]$ is a decision, so that this is, in fact, a model of $\mathtt{RINT}_k$ in **PInj**.

We can now reap the benefits in the form of a reversibility theorem for free:

Theorem 6.1 (Reversibility) *Every* $\mathtt{RINT}_k$ *program* p *is semantically reversible in the sense that* $[\![p]\!]$ *is a partial isomorphism.*

Further, since we can straightforwardly show that $[\![\mathtt{x_i} \mathrel{+}= \mathtt{x_j}]\!]^\dagger = [\![\mathtt{x_i} \mathrel{-}= \mathtt{x_j}]\!]$ and $[\![\mathtt{x_i} \mathrel{+}= \overline{n}]\!]^\dagger = [\![\mathtt{x_i} \mathrel{+}= \overline{-n}]\!]$, we can use the technique from Sec. 5.3 to obtain a sound and complete program inverter.

Theorem 6.2 (Program inversion) $\mathtt{RINT}_k$ *has a (sound and complete) program inverter. In particular, for every* $\mathtt{RINT}_k$ *program* p *there exists a program* $\mathcal{I}[p]$ *such that* $[\![\mathcal{I}[p]]\!] = [\![p]\!]^\dagger$.

$$
\begin{array}{lll}
\mathit{pnot} & :: \mathit{PBool}\ \alpha \leftrightarrow \mathit{PBool}\ \alpha & \\
\mathit{pnot}\ (\mathit{True}\ x) & = \mathit{False}\ x & \\
\mathit{pnot}\ (\mathit{False}\ x) & = \mathit{True}\ x &
\end{array}
\qquad
\begin{array}{lll}
\mathit{peven} & :: \mathit{Nat} \leftrightarrow \mathit{PBool}\ \mathit{Nat} & \\
\mathit{peven}\ 0 & = \mathit{True}\ 0 & \\
\mathit{peven}\ (n+1) & = \mathit{fmap}\ (+1)\ (\mathit{pnot}\ (\mathit{peven}\ n)) &
\end{array}
$$

Fig. 3. The definition of the *even*-predicate as a decision on natural numbers.

6.2 Decisions as a programming technique

Decisions offer a solution to the awkwardness in representing predicates reversibly. On the programming side, the reversible *duplication/equality operator* [11] (see also [26]) can be seen as a distant ancestor to predicates-as-decisions, in that it provides an ad-hoc solution to the problem of checking whether two values are equal in a reversible manner.

Decisions offer a more systematic approach: They suggest that one ought to define Boolean values in reversible functional programming not in the usual way, but rather by means of the polymorphic datatype

$$\textbf{data}\ \mathit{PBool}\ \alpha\ =\ \mathit{True}\ \alpha\ \mid\ \mathit{False}\ \alpha$$

storing not only the *result*, but also *what* was tested to begin with. With this definition, negation on these polymorphic Booleans (*pnot*) may be defined straightforwardly as shown in Figure 3. In turn, this allows for more complex predicates to be expressed in a largely familiar way. For example, the decision for testing whether a natural number is even (*peven*) is also shown in Figure 3, with *fmap* given in the straightforward way on polymorphic Booleans. For comparison, the corresponding irreversible predicate is typically defined as follows, with *not* the usual negation of Booleans

$$
\begin{array}{lll}
\mathit{even} & :: \mathit{Nat} \to \mathit{Bool} & \\
\mathit{even}\ 0 & = \mathit{True} & \\
\mathit{even}\ (n+1) & = \mathit{not}\ (\mathit{even}\ n) & .
\end{array}
$$

As such, the reversible implementation as a decision is nearly identical, the only difference being the use of *fmap* in the definition of *peven* to recover the input value once the branch has been decided.

7 Concluding remarks

In the present paper, we have built on the work on extensive restriction categories to derive a related concept of extensivity for inverse categories. We have used this concept to give a novel reversible representation of predicates and their corresponding assertions in (specifically extensive) join inverse categories with a disjointness tensor, and in turn used these to model the fundamental control structures of reversible loops and conditionals in structured reversible flowchart languages. This approach allowed us to derive a program inversion theorem for structured reversible flowchart languages, and we illustrated our approach by developing a family of structured

reversible flowchart languages and using our framework to give it denotational semantics, with theorems regarding reversibility and program inversion for free.

The idea to represent predicates by decisions was partially inspired by the *instruments* associated with predicates in Effectus theory [15]. Given that *side effect free* instruments ι satisfy a similar rule, $\nabla \circ \iota = \mathrm{id}$, and that Boolean effecti are extensive, it could be interesting to explore the connections between extensive restriction categories and Boolean effecti, especially as regards their internal logic.

Finally, on the programming language side, it could be interesting to further explore how decisions can be used in reversible programming, *e.g.*, to do the heavy lifting involved in pattern matching and branch joining. As our focus has been on the representation of predicates, our approach may be easily adapted to other reversible flowchart structures, *e.g.*, Janus-style loops [27].

References

[1] Axelsen, H. B. and R. Glück, *What do reversible programs compute?*, in: M. Hofmann, editor, *Foundations of Software Science and Computation Structures. Proceedings*, Lecture Notes in Computer Science **6604** (2011), pp. 42–56.

[2] Axelsen, H. B., R. Glück and T. Yokoyama, *Reversible machine code and its abstract processor architecture*, in: V. Diekert, M. V. Volkov and A. Voronkov, editors, *Computer Science – Theory and Applications. Proceedings*, Lecture Notes in Computer Science **4649** (2007), pp. 56–69.

[3] Axelsen, H. B. and R. Kaarsgaard, *Join inverse categories as models of reversible recursion*, in: B. Jacobs and C. Löding, editors, *FOSSACS 2016, Proceedings* (2016), pp. 73–90.

[4] Carboni, A., S. Lack and R. F. C. Walters, *Introduction to extensive and distributive categories*, Journal of Pure and Applied Algebra **84** (1993), pp. 145 – 158.

[5] Carothers, C. D., K. S. Perumalla and R. M. Fujimoto, *Efficient optimistic parallel simulations using reverse computation*, ACM Trans. Model. Comput. Simul. **9** (1999), pp. 224–253.

[6] Cockett, J. R. B. and S. Lack, *Restriction categories I: Categories of partial maps*, Theoretical Computer Science **270** (2002), pp. 223–259.

[7] Cockett, J. R. B. and S. Lack, *Restriction categories II: Partial map classification*, Theoretical Computer Science **294** (2003), pp. 61–102.

[8] Cockett, R. and S. Lack, *Restriction categories III: Colimits, partial limits and extensivity*, Mathematical Structures in Computer Science **17** (2007), pp. 775–817.

[9] Frank, M. P., "Reversibility for efficient computing," Ph.D. thesis, EECS Dept., Massachusetts Institute of Technology (1999).

[10] Giles, B. G., "An Investigation of some Theoretical Aspects of Reversible Computing," Ph.D. thesis, University of Calgary (2014).

[11] Glück, R. and M. Kawabe, *A program inverter for a functional language with equality and constructors*, in: A. Ohori, editor, *Programming Languages and Systems. Proceedings*, Lecture Notes in Computer Science **2895** (2003), pp. 246–264.

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

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

[14] Guo, X., "Products, Joins, Meets, and Ranges in Restriction Categories," Ph.D. thesis, University of Calgary (2012).

[15] Jacobs, B., *New directions in categorical logic, for classical, probabilistic and quantum logic*, Logical Methods in Computer Science **11** (2015), pp. 1–76.

[16] Joyal, A., R. Street and D. Verity, *Traced monoidal categories*, Mathematical Proceedings of the Cambridge Philosophical Society **119** (1996), pp. 447–468.

[17] Kaarsgaard, R., H. B. Axelsen and R. Glück, *Join inverse categories and reversible recursion*, Journal of Logical and Algebraic Methods in Programming **87** (2017), pp. 33–50.

[18] Kastl, J., *Inverse categories*, in: H.-J. Hoehnke, editor, *Algebraische Modelle, Kategorien und Gruppoide*, Studien zur Algebra und ihre Anwendungen **7**, Akademie-Verlag, 1979 pp. 51–60.

[19] Mogensen, T. Æ., *Partial evaluation of the reversible language Janus*, in: *Partial Evaluation and Program Manipulation. Proceedings* (2011), pp. 23–32.

[20] Selinger, P., *Dagger compact closed categories and completely positive maps*, Electronic Notes in Theoretical Computer Science **170** (2007), pp. 139–163.

[21] Selinger, P., *A survey of graphical languages for monoidal categories*, in: B. Coecke, editor, *New Structures for Physics* (2011), pp. 289–355.

[22] Thomsen, M. K., H. B. Axelsen and R. Glück, *A reversible processor architecture and its reversible logic design*, in: A. De Vos and R. Wille, editors, *Reversible Computation. Proceedings*, Lecture Notes in Computer Science **7165** (2012), pp. 30–42.

[23] Thomsen, M. K., R. Glück and H. B. Axelsen, *Reversible arithmetic logic unit for quantum arithmetic*, Journal of Physics A: Mathematical and Theoretical **43** (2010), p. 382002.

[24] Yokoyama, T., H. B. Axelsen and R. Glück, *Reversible flowchart languages and the structured reversible program theorem*, in: L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir and I. Walukiewicz, editors, *International Colloquium on Automata, Languages and Programming. Proceedings*, Lecture Notes in Computer Science **5126** (2008), pp. 258–270.

[25] Yokoyama, T., H. B. Axelsen and R. Glück, *Optimizing clean reversible simulation of injective functions*, Journal of Multiple-Valued Logic and Soft Computing **18** (2012), pp. 5–24.

[26] Yokoyama, T., H. B. Axelsen and R. Glück, *Towards a reversible functional language*, in: A. De Vos and R. Wille, editors, *RC 2011*, Lecture Notes in Computer Science **7165** (2012), pp. 14–29.

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