Categorical Semantics of Reversible Pattern-Matching

Kostia Chardonnet
LMF, Univ. Paris Saclay.
IRIF, Univ. Paris.
kostia@lri.fr

Louis Lemonnier
LMF, Univ. Paris Saclay.
lemonnier@lsv.fr

Benoît Valiron
LMF, CentraleSupélec, Univ. Paris Saclay
benoit.valiron@universite-paris-saclay.fr

This paper is concerned with categorical structures for reversible computation. In particular, we focus on a typed, functional reversible language based on Theseus. We discuss how join inverse rig categories do not in general capture pattern-matching, the core construct Theseus uses to enforce reversibility. We then derive a categorical structure to add to join inverse rig categories in order to capture pattern-matching. We show how such a structure makes an adequate model for reversible pattern-matching.

1 Introduction

In this paper, we are concerned with the semantics of reversible programming languages. The idea of reversible computation comes from Landauer and Bennett [28, 3] with the analysis of its expressivity, and the relationship between irreversible computing and dissipation of energy. This lead to an interest in reversible computation [4, 1], both with a low-level approach [6, 33, 31], and from a high-level perspective [29, 37, 36, 21, 20, 30, 35, 32, 19]. Reversible programming lies on the latter side of the spectrum, and two main approaches have been followed. Embodied by Janus [29, 37, 34, 36] and later R-CORE and R-WHILE [13], the first one focuses on imperative languages whose control flow is inherently reversible —the main issue with this aspect being tests and loops. The other approach is concerned with the design of functional languages with structured data and related case-analysis, or *pattern-matching* [35, 32, 20, 30, 19]. To ensure reversibility, strong constraints have to be established on the pattern-matching in order to maintain reversibility.

In general, reversible computation captures *partial injective maps* [13] from inputs to outputs, or, equivalently in this paper, *partial isomorphisms*. Indeed, from a computational perspective reversibility is understood as a time-local property: if each time-step of the execution of the computation can soundly be reversed, there is no overall condition on the global behavior of the computation. In particular, this does not say anything about termination: a computation seen as a map from inputs to outputs might very well be partial, as some inputs may trigger a (global) non-terminating behavior.

The categorical analysis of partial injective maps have been thoroughly analyzed since 1979, first by Kastl [27], and then by Cockett and Lack [8, 9, 10]. This led to the development of *inverse category*: a category equipped with an inverse operator in which all morphisms have partial inverses and are therefore reversible. The main aspect of this line of research is that partiality can have a purely algebraic description: one can introduce a restriction operator on morphisms, associating to a morphism a partial identity on its domain.

This categorical framework has recently been put to use to develop the semantics of specific reversible programming constructs and concrete reversible languages: analysis of recursion in the context of reversibility [2, 23, 26], formalization of reversible flowchart languages [12, 22], analysis of side-effects [17, 16], *etc*. Interestingly enough however, the adequacy of the developed categorical constructs

Ana Sokolova (Eds.): MFPS 2021
EPTCS 351, 2021, pp. 18–33, doi:10.4204/EPTCS.351.2

with reversible *functional* programming languages has been seldom studied. For instance, if Kaarsgaard *et al.* [24] mention Theseus as a potential use-case, they do not discuss it in details. So far, the semantics of functional and applicative reversible languages has always been done in *concrete* categories of partial isomorphisms [26, 25].

In particular, one important aspect that has not been addressed yet in detail is the categorical interpretation of pattern-matching. If pattern-matching can be added to reversible imperative languages [13], it is particularly relevant in the context of functional languages where it is one of the core construct needed for manipulating structured data. This is for instance emphasized by the several existing languages making use of it [35, 32, 20, 19, 30, 7]. In the literature, pattern-matching has either been considered in the context of a Set-based semantics [13], or more generally in categorical models making heavy use of rig structures [5] or co-products [26, 25] to represent it. If such rich structures are clearly enough to capture pattern-matching, we claim that they are too coarse, and that a weaker structure is enough for characterizing pattern-matching.

Contributions. In this paper, we make a proposal for a general categorical interpretation of pattern-matching in the context of inverse categories, without having to rely on external structure such as coproducts. More specifically, we study the categorical semantics of a typed, linear and reversible language in the style of Theseus [20]. We develop and discuss *pattern-matching categories*: a categorical construction shown to be sufficient to model the pattern-matching at the heart of the operational semantics of the language. In particular, in this category we can mimic the notion of clauses and values to be matched against them, without to have to rely on coproducts. We conclude the paper with a proof that pattern-matching categories make adequate models for the considered language, thus confirming the validity of the approach.

This paper is organized as follows: first, in Section 2 we present the language based on [30], its type system and rewriting system along with the usual safety properties. We then give some reminder on inverse categories and set the theoretical background for the rest of the paper in Section 3. Basic knowledge in category theory is assumed. Finally, sections 4 and 5 are focused on the categorical semantics of the language. The proofs are available in the appendix.

2 The language

In this section, we present a small, formal reversible functional programming language in the like of Theseus [20] and its later developments [30, 7]. The core feature of Theseus is to define reversible control-flow using pattern-matching. Doing so elegantly bridges functional programming and reversible computation in a typed manner: well-typed, terminating programs describe isomorphisms between types. In previous —untyped— approaches [35], pattern-matching was simply considered as one language feature among many, without any special attention devoted to it, and the reversible aspect was taken care of separately. On the contrary, Theseus uses pattern matching as the core feature to make the language reversible.

In the context of a typed pattern-matching, the compiler can easily verify two crucial properties: (i) non-overlapping patterns in clauses, ensuring deterministic behavior, and (ii) exhaustive coverage of a datatype with the patterns, ensuring totality. In [30], these properties are shown sufficient to produce a simple first-order reversible programming language. In this paper we relax the constraints on exhaustivity, making it possible to define partial functions.

2.1 Terms and Types

The language we focus on in this paper is a weakened version of the language presented in [30, Sec.2]. In particular, we allow non-exhaustive pattern-matching and enum types. The language is typed and consists of two layers: values and functions (called "isos" in [30]). It is parametrized by a set of enum types (spanned by α) and their constant values (spanned by c_α). The language whose only type α is the unit-type with one single unit constant will be called the *minimal language*.

$$\begin{array}{llll} \text{(Value types)} & a,b & ::= & \alpha \mid a \oplus b \mid a \otimes b \\ \text{(Iso types)} & T & ::= & a \leftrightarrow b \\ \text{(Values)} & v & ::= & c_\alpha \mid x \mid \mathtt{inj}_l\, v \mid \mathtt{inj}_r\, v \mid \langle v_1, v_2 \rangle \\ \text{(Functions)} & \omega & ::= & \{ \mid v_1 \leftrightarrow v_1' \mid \ldots \mid v_n \leftrightarrow v_n' \} \\ \text{(Terms)} & t & ::= & v \mid \omega\, t \end{array}$$

The language comes with two kinds of judgments, one for terms and one for functions (or *isos*). We denote typing contexts as Δ, they stand for sets of typed variables $x_1 : a_1, \ldots, x_n : a_n$. We then write $\Delta \vdash_v t : a$ for a well-typed term and $\vdash_\omega \omega : a \leftrightarrow b$ for a well-typed function.

Beside the linearity aspect (used to ensure injectivity), the typing rules for terms are standard. The typing judgment of a term is valid if it can be derived from the following rules.

$$\frac{}{\vdash_v c_\alpha : \alpha}, \qquad \frac{}{x : a \vdash_v x : a}, \qquad \frac{\Delta_1 \vdash_v v_1 : a \quad \Delta_2 \vdash_v v_2 : b}{\Delta_1, \Delta_2 \vdash_v \langle v_1, v_2 \rangle : a \otimes b.}$$

$$\frac{\Delta \vdash_v v : a}{\Delta \vdash_v \mathtt{inj}_l\, v : a \oplus b}, \qquad \frac{\Delta \vdash_v v : b}{\Delta \vdash_v \mathtt{inj}_r\, v : a \oplus b}, \qquad \frac{\vdash_v t : a \quad \vdash_\omega \omega : a \leftrightarrow b}{\vdash_v \omega\, t : b}$$

In particular, while a value can have free variables, a term is always closed. This difference is explained by the fact that values are used as patterns in clauses. The typing rule for isos is as follows.

$$\frac{\begin{array}{ccc} \Delta_1 \vdash_v v_1 : a & & \Delta_n \vdash_v v_n : a \quad \forall i \neq j, v_i \perp v_j \\ \Delta_1 \vdash_v v_1' : b & \cdots & \Delta_n \vdash_v v_n' : b \quad \forall i \neq j, v_i' \perp v_j' \end{array}}{\vdash_\omega \{ \mid v_1 \leftrightarrow v_1' \mid \ldots \mid v_n \leftrightarrow v_n' \} : a \leftrightarrow b,} \tag{1}$$

The rule relies on the condition that both left- and right-hand-side patterns in clauses are orthogonal, enforcing non-overlapping. The rules for deriving orthogonality of values (or patterns) are the following.

$$\frac{c_\alpha \neq d_\alpha}{c_\alpha \perp d_\alpha} \qquad \frac{}{\mathtt{inj}_l\, v_1 \perp \mathtt{inj}_r\, v_2} \qquad \frac{}{\mathtt{inj}_r\, v_1 \perp \mathtt{inj}_l\, v_2}$$

$$\frac{v_1 \perp v_2}{\mathtt{inj}_l\, v_1 \perp \mathtt{inj}_l\, v_2} \qquad \frac{v_1 \perp v_2}{\mathtt{inj}_r\, v_1 \perp \mathtt{inj}_r\, v_2} \qquad \frac{v_1 \perp v_2}{\langle v, v_1 \rangle \perp \langle v', v_2 \rangle} \qquad \frac{v_1 \perp v_2}{\langle v_1, v \rangle \perp \langle v_2, v' \rangle}$$

Left and right injections generates disjoint subsets of values, and distinct constants of an enum type α are orthogonal.

2.2 Operational semantics

The language is equipped with a simple call-by-value operational semantics on terms based on matching and substitution. We recall the formalization proposed in [30], with the notion of valuation: partial map

from a finite set of variables (the support) to a set of values. We denote the matching of a value w against a pattern v and its associated valuation σ as $\sigma[v] = w$. It is defined as follows.

$$\frac{}{\sigma[c_\alpha] = c_\alpha} \quad \frac{\sigma = \{x \mapsto v\}}{\sigma[x] = v} \quad \frac{\sigma[v] = w}{\sigma[\mathtt{inj}_l\, v] = \mathtt{inj}_l\, w} \quad \frac{\sigma[v] = w}{\sigma[\mathtt{inj}_r\, v] = \mathtt{inj}_r\, w}$$

$$\frac{\sigma_2[v_1] = w_1 \quad \sigma_1[v_2] = w_2 \quad \mathrm{supp}(\sigma_1) \cap \mathrm{supp}(\sigma_2) = \emptyset \quad \sigma = \sigma_1 \cup \sigma_2}{\sigma[\langle v_1, v_2\rangle] = \langle w_1, w_2\rangle}$$

Whenever σ is a valuation whose support contains the variables of v, we write $\sigma(v)$ for the value where the variables of v have been replaced with the corresponding values in σ, as follows:

- $\sigma(c_\alpha) = c_\alpha$,
- $\sigma(x) = v$ if $\{x \mapsto v\} \subseteq \sigma$,
- $\sigma(\mathtt{inj}_l\, v) = \mathtt{inj}_l\, \sigma(v)$,
- $\sigma(\mathtt{inj}_r\, v) = \mathtt{inj}_r\, \sigma(v)$,
- $\sigma(\langle v_1, v_2\rangle) = \langle \sigma(v_1), \sigma(v_2)\rangle$.

Definition 2.1 (Reduction)**.** *The reduction $\to$ is then defined as the smallest relation such that $\omega\, t \to \omega\, t'$ whenever $t \to t'$ and such that, provided that $\sigma[v_i] = v$, the redex*

$$\{ \mid v_1 \leftrightarrow v_1' \mid \ldots \mid v_n \leftrightarrow v_n' \}\ v$$

reduces to $\sigma(v_i')$. As usual, we write $s \to t$ to say that s rewrites in one step to t and $s \to^* t$ to say that s rewrites to t in 0 or more steps.

2.3 Properties

The language satisfies usual safety properties, and, although not necessarily total, functions are indeed reversible. In this section we formalize these results.

Remark 2.2. *The reduction is deterministic: if $s \to t$ and $s \to t'$ then $t = t'$.*

Because of the conditions set on patterns in the typing rule of isos, the rewrite system is deterministic. Note that, since we do not impose exhaustivity, the reduction might get stuck. Nonetheless, the following properties hold [30].

Lemma 2.3 (Subject reduction)**.** *If $\vdash_v s : a$ and $s \to t$ then $\vdash_v t : a$.* □

Lemma 2.4 (Termination)**.** *If $\vdash_v s : a$ then there exists a term t that does not reduce such that $s \to^* t$.* □

The small language presented in this section can be called *reversible* for the following reason. Given an iso

$$\omega = \{ \mid v_1 \leftrightarrow v_1' \mid \ldots \mid v_n \leftrightarrow v_n' \},$$

we can syntactically define the *inverse* ω^{-1} as the iso

$$\{ \mid v_1' \leftrightarrow v_1 \mid \ldots \mid v_n' \leftrightarrow v_n \}.$$

This inverse satisfies the following property.

Lemma 2.5 (Inverse)**.** *If $\vdash_\omega \omega : a \leftrightarrow b$ and $\vdash_v v : a$, and if $\omega\, v \to^* w$ with w a value, then $\vdash_\omega \omega^{-1} : b \leftrightarrow a$ and we have that $\omega^{-1}\, w \to^* v$.* □

3 Inverse categories

The simple reversible language of Section 2 can easily be modeled within the category PInj of sets and partial injective maps. Indeed, suppose that we write $\{|a|\}$ for the set of closed values of type a, that is, the set of values v such that $\vdash_v v : a$. Because of Lemmas 2.3 and 2.4, for each well-typed function $\vdash_\omega \omega : a \leftrightarrow b$ one can define an injective set-map $\{|\omega|\} : \{|a|\} \to \{|b|\}$ as follows: $\{|\omega|\}(v) = w$ whenever $\omega\, v \to^* w$. Many of the properties of PInj can be reflected in the language, for instance partiality and union of functions with disjoint domain.

Example 3.1. *Let ω_1 and ω_2 be respectively defined as*

$$\vdash_\omega \{ \mid \mathtt{inj}_l\, x \leftrightarrow \mathtt{inj}_r\, x \,\} : a \oplus b \leftrightarrow b \oplus a,$$
$$\vdash_\omega \{ \mid \mathtt{inj}_r\, x \leftrightarrow \mathtt{inj}_l\, x \,\} : a \oplus b \leftrightarrow b \oplus a.$$

They correspond to two partial set-functions $\{|\omega_1|\}$ and $\{|\omega_2|\}$ defined on the disjoint union $\{|a|\} \uplus \{|b|\}$: $\{|\omega_1|\}$ is defined on the $\{|a|\}$-component and $\{|\omega_2|\}$ on the $\{|b|\}$-component. Defined as

$$\{ \mid \mathtt{inj}_l\, x \leftrightarrow \mathtt{inj}_r\, x \mid \mathtt{inj}_r\, x \leftrightarrow \mathtt{inj}_l\, x \,\},$$

the iso ω of type $a \oplus b \leftrightarrow b \oplus a$ can be regarded as the union *of ω_1 and ω_2 —and in the set-model, their denotation is such a union: $\{|\omega|\} = \{|\omega_1|\} \uplus \{|\omega_2|\}$. This works only because the functions $\{|\omega_1|\}$ and $\{|\omega_2|\}$ are* compatible *on their common domain of definition.*

The category PInj has been extensively studied and its structure analyzed within the framework of *inverse categories* [27, 8, 9, 10, 11, 14]. These categories formalize the notion of *partial inverse* morphisms and also conveys a natural definition of joins (without relying on coproducts) :—least upper bounds—, which shall be shown as the best way to denote the pattern-matching, as discussed in Example 3.1.

This section therefore aims at a rapid introduction to inverse categories: it contains the necessary material needed to read and understand the remaining sections: it is far from stating all the results and interests of restriction and inverse categories. A reader interested in the subject may for instance refer to [8, 9, 10, 11, 14] for further information.

3.1 Restriction category

The definition of a proper *partial function* requires a formal way to write the "domain" of a morphism f, through a *partial identity* $\overline{f}$ called *restriction*.

Definition 3.2 (Restriction [8])**.** *A restriction structure is an operator that maps each morphism $f : A \to B$ to a morphism $\overline{f} : A \to A$ such that*

$$f \circ \overline{f} = f \quad (2) \qquad \overline{f} \circ \overline{g} = \overline{g} \circ \overline{f} \quad (3) \qquad \overline{f \circ \overline{g}} = \overline{f} \circ \overline{g} \quad (4) \qquad \overline{h} \circ f = f \circ \overline{h \circ f} \quad (5)$$

A morphism f is said to be total if $\overline{f} = 1_A$. A category with a restriction structure is called a restriction category.

Remark 3.3. *When unambiguous, we write gf for the composition $g \circ f$.*

Example 3.4. *Any category can be given a restriction structure by saying that all morphisms are total, but this is definitely not interesting as a model. The standard non trivial example of restriction category is Pfn, the category of sets and partial functions. Given a partial function $f : A \to B$, we can define its restriction operator as $\overline{f}(x) = x$ on the domain of f and undefined otherwise.*

Throughout this paper, we manipulate functors. These are often required to keep the restriction structure intact; hence the following definition.

Definition 3.5 (Restriction functor [24]). *A functor $F : \mathscr{C} \to \mathscr{D}$ is a* restriction functor *if $\overline{F(f)} = F(\overline{f})$ for all morphism f of $\mathscr{C}$. The definition is canonically extended to bifunctors.*

3.2 Inverse category

Our goal is to denote reversible operations: this requires some form of inverse. The restriction operator gives a notion of domain of morphisms, and moreover, $\overline{f}$ is meant as an identity function on this domain; thus the composition of f with its presumed inverse should be equal to its restriction. Hence the next definition. Note that inverse categories were invented [27] before restriction categories, but the order of definitions used here is believed more convenient by the authors.

Definition 3.6 (Inverse category [24]). *An* inverse category *is a restriction category where all morphisms are partial isomorphisms; meaning that for $f : A \to B$, there exists a unique $f^\circ : B \to A$ such that $f^\circ \circ f = \overline{f}$ and $f \circ f^\circ = \overline{f^\circ}$.*

The canonical example of inverse category is PInj, the category of sets and partial injective functions. It is actually more than canonical:

Theorem 3.7 ([27]). *Every locally small inverse category is isomorphic to a subcategory of PInj.* □

Example 3.8. *Let us fix a non-empty set S. We then define PId_S as the category with one object $*$ and with morphisms all of the sets $Y \subseteq S$. Composition is defined as intersection and the identity is S. Intersection also gives us a monoidal structure with unit S. Note that the category PId_S can be regarded as a subcategory of PInj, where each morphism $Y \subseteq S$ corresponds to a partial identity defined on Y. The category PId_S can be endowed with a structure of inverse category by defining morphisms as their own restriction and partial inverse.*

From PId_S we can define $\mathrm{PId}_S^{\oplus}$ as the inverse category whose objects are (generic) sets, and morphisms are defined as follows: $f : A \to B$ is a pair $f = \left(\sigma_f, \{X_a^f\}_{a \in dom(\sigma_f)}\right)$ where σ_f is a partial injective map between sets $A \to B$, and where each X_a^f is a morphism of PId_S. Composition in $\mathrm{PId}_S^{\oplus}$ is done pairwise with classical composition of functions for σ and the composition in PId_S for the rest. The identity over A is the pair made of the set-identity on A together with S for each $a \in A$. The restriction and partial inverse are generated from those of PInj and PId_S for each element in the pair.

3.3 Compatibility

To give a denotation to isos in the programming language considered in Section 2, it is compulsory to find a way to combine terms such as ω_1 and ω_2 in Example 3.1. More generally speaking, we need to combine morphisms of the same type $A \to B$ to make a "join" morphism. First, we have to make sure that the morphisms are *compatible*: f and g are compatible if they are alike on their common "domain", and can behave however they like when the other does not apply. Since we aim at building a model for (partial) bijections, compatibility of the partial inverse is also checked.

Definition 3.9 (Restriction compatible [24]). *Two morphisms $f, g : A \to B$ in a restriction category $\mathscr{C}$ are restriction compatible if $f\overline{g} = g\overline{f}$. The relation is written $f \smile g$. If $\mathscr{C}$ is an inverse category, they are inverse compatible if $f \smile g$ and $f^\circ \smile g^\circ$, noted $f \asymp g$. A set S of morphisms of the same type $A \to B$ is restriction compatible (resp. inverse compatible) if all elements of S are pairwise restriction compatible (resp. inverse compatible).*

Example 3.10. *In PInj, let us consider $f,g : \{a,b,c\} \to \{a,b,c\}$ defined as identities on their domains where: $dom(f) = \{a,b\}$, $dom(g) = \{b,c\}$. It is pretty clear that $f \asymp g$. However, if we consider $h : \{a,b,c\} \to \{a,b,c\}$ defined on $\{b,c\}$ as $h(b) = a$ and $h(c) = c$, then f and h are not compatible.*

3.4 Joins and ordering on morphisms

When considering partial set-functions, a natural notion of order can be built on functions by considering domain-inclusion: we can say that $f \leq g$ if g is defined on the whole "domain" of f and both behave the same there. When considering models of reversible languages, joins are used to model pattern-matching [11, 25]: each clause is a partial function, and the complete pattern-matching can then be represented with the *join* of the clauses. This notion can be extended to restriction categories as follows.

Definition 3.11 (Partial order [8])**.** *Let $f,g : A \to B$ be two morphisms in a restriction category. We then define $f \leq g$ as $g\overline{f} = f$.*

Example 3.12. *Consider PId_S from Example 3.8: we have $X \leq Y$ iff $X \subseteq Y$. For $PId_S^{\oplus}$ we have $f \leq g$ iff $dom(\sigma_f) \subseteq dom(\sigma_g)$ and, for all $a \in dom(\sigma_f)$, $\sigma_f(a) = \sigma_g(a)$, and, for all $a \in dom(\sigma_f)$, $X_a^f \subseteq X_a^g$.*

Property 3.13. *Let us consider $f,g : A \to B$ such that $f \leq g$. Then, whenever $h : B \to C$, we have $hf \leq hg$. Similarly, whenever $h : C \to A$ we have $fh \leq gh$.* □

As discussed in Example 3.1, the isos ω_1 and ω_2 can be combine to form the "join" morphism ω. This notion of join is defined in the context of restriction categories as follows.

Definition 3.14 (Joins [14])**.** *A restriction category $\mathscr{C}$ is equipped with joins if for all restriction compatible sets S of morphisms $A \to B$, there exists $\bigvee_{s\in S} s : A \to B$ morphism of $\mathscr{C}$ such that, whenever $t : A \to B$ and whenever for all $s \in S$, $s \leq t$,*

$$s \leq \bigvee_{s\in S} s \ \ (6), \quad \bigvee_{s\in S} s \leq t \ \ (7), \quad \overline{\bigvee_{s\in S} s} = \bigvee_{s\in S} \overline{s} \ \ (8), \quad f \circ \left(\bigvee_{s\in S} s\right) = \bigvee_{s\in S} fs \ \ (9), \quad \left(\bigvee_{s\in S} s\right) \circ g = \bigvee_{s\in S} sg \ \ (10).$$

Such a category is called a join restriction category. *An inverse category with joins is called a* join inverse category.

Example 3.15. *Consider the morphisms f,g from Example 3.10. They are compatible, and $f \vee g = 1_{\{a,b,c\}}$. It is easy to verify that e.g. $f \leq 1_{\{a,b,c\}}$ and $g \leq 1_{\{a,b,c\}}$. Let us consider $h,k,l : \{a,b,c\} \to \{a,b,c\}$ such that $h(a) = b$ and undefined otherwise, $k(b) = c$ and undefined otherwise, and l defined as $l(a) = b$, $l(b) = c$ and $l(c) = a$. We have $h \leq l$ and $k \leq l$. Besides, $h \vee k$ is the function whose domain is $\{a,b\}$ and sending $a \mapsto b$ and $b \mapsto c$. We therefore have $h \vee k \leq l$.*

The join operator admits a unit, as follows.

Definition 3.16 (Zero [24])**.** *Since $\emptyset \subseteq Hom_{\mathscr{C}}(A,B)$, and since all of its elements are restriction compatible, there exists a morphism $0_{A,B} \doteq \bigvee_{s\in\emptyset} s$, called* zero.

Lemma 3.17 ([24])**.** *Whenever well-typed, the zero satisfies the following equations: $f0 = 0$, $0g = 0$, $0^{\circ}_{A,B} = 0_{B,A}$, $\overline{0_{A,B}} = 0_{A,A}$.* □

4 Semantics of isos

In this section, we turn to the question of representing the terms and isos of the language presented in Section 2. Instead of using the concrete category PInj as suggested in the header of Section 3, we aim at using general, inverse categories.

4.1 Representing choice and pairing

One of the problem to overcome is the denotation of pairing and sums. If a standard categorical denotation for the former is a monoidal structure, the latter is usually represented with coproducts. Such a notion is however slightly insufficient. Indeed, for being able to join independent clauses in pattern-matching (as shown e.g. in Example 3.1), it has to interact well with the restriction structure. This lead to the development of *disjointness tensor*. If Giles [11] was the first one to introduce the notion of disjointness tensors, in this paper we rely on the definition of [24].

Definition 4.1 (Disjointness tensor [24])**.** *An inverse category $\mathscr{C}$ is said to have a disjointness tensor if it is equipped with a restriction bifunctor $. \oplus . : \mathscr{C} \times \mathscr{C} \to \mathscr{C}$, with a unit object 0 and morphisms $\iota_l : A \to A \oplus B$ and $\iota_r : B \to A \oplus B$ that are total and such that $\overline{\iota_l^\circ}\ \overline{\iota_r^\circ} = 0_{A\oplus B}$.*

To model the types of Section 2, we therefore essentially need a disjointness tensor and a monoidal structure. Similar to what happen in the category PInj, they need to relate through distributivity.

Definition 4.2 ([25])**.** *Let us consider a join inverse category equipped with a symmetric monoidal tensor product $(\otimes, 1)$ and a disjointness tensor $(\oplus, 0)$ that are join preserving, and such that there is an isomorphism $\delta_{A,B,C} : A \otimes (B \oplus C) \to (A \otimes B) \oplus (A \otimes C)$. This is called a* join inverse rig category.

Example 4.3. *The category $\mathrm{PId}_S^{\oplus}$ is a join inverse rig category with the following structure.*

- *$A \otimes B = A \times B$, the usual cartesian product. If $f : A \to B$ and $g : C \to D$ then we define $\sigma_{f\otimes g}(a,c) \doteq (\sigma_f(a), \sigma_g(c))$ and $X^{f\otimes g}_{(a,c)} \doteq X^f_a \cap X^g_c$. The unit 1 is the singleton set.*
- *$A \oplus B = A \uplus B$, the disjoint union. If $f : A \to B$ and $g : C \to D$ we define $\sigma_{f\uplus g}(\iota_l(a)) \doteq \iota_l(\sigma_f(a))$, $\sigma_{f\uplus g}(\iota_r(c)) \doteq \iota_r(\sigma_g(c))$, $X^{f\uplus g}_{\iota_l(a)} \doteq X^f_a$, $X^{f\uplus g}_{\iota_r(c)} \doteq X^g_c$. The zero is the empty set.*

Note how in $\mathrm{PId}_S^{\oplus}$ if S is of cardinality at least 2 there are several morphisms $1 \to 1$. In particular a morphism $1 \to 1 \oplus 1$ is not necessarily an injection.

4.2 Orthogonality between morphisms

In the description of the language, the definition of isos heavily relies on orthogonality. In this section we define this notion over morphisms in an inverse category.

General definitions of orthogonality in inverse categories exist, but we have chosen to manipulate a more practical one. Our notion introduced below verifies all the axioms set in [11] for orthogonality of morphisms. We shall then see in this section that orthogonal values produce *disjoint* morphisms.

Definition 4.4 (Disjointness)**.** *$f : A \to B,\ g : A \to C$ are disjoint iff $\overline{f}\ \overline{g} = 0$.*

To picture it, one may say that disjoint morphisms have an empty common domain of definition. We can then prove that some of the axioms of morphism orthogonality stated in [11] hold for disjointness, expressed with the lemmas below.

Lemma 4.5 (Left-composed disjointness)**.** *Let $f_1 : A \to B_1,\ f_2 : A \to B_2,\ g_1 : B_1 \to C_1,\ g_2 : B_2 \to C_2$ be morphisms. If $\overline{f_1}\ \overline{f_2} = 0$, then $\overline{g_1 f_1}\ \overline{g_2 f_2} = 0$.* □

Lemma 4.6 (Right-composed disjointness)**.** *Let $f : A \to B,\ g : A \to C,\ h : D \to A$ be morphisms. If $\overline{f}\ \overline{g} = 0$ then $\overline{fh}\ \overline{gh} = 0$.* □

Lemma 4.7 (Disjoint compatibility)**.** *If $\overline{f_1}\ \overline{f_2} = 0$ then $f_1 \smile f_2$.* □

This last lemma is very natural: if two morphisms apply on strictly different domains, they are compatible. It also allows to underline that disjointness is a right choice to picture orthogonality.

4.3 Semantics of types and values

In order to model the types of the language presented in Section 2, the most natural is to consider a *join* inverse *rig* category to capture the tensor and the sum-type.

Let us then fix such a join inverse rig category $\mathscr{C}$. Each type a is given an interpretation as an object in the category $\mathscr{C}$, written $[\![a]\!]$. If a type α has n constants, it is necessary to choose $[\![\alpha]\!]$ such that there are n morphisms $f^i_\alpha : 1 \to [\![\alpha]\!]$ $(i = 1 \dots n)$ that are total and pairwise disjoint, following Definition 4.4. The set of these morphisms will be written S_α.

A sequence of types or of terms is denoted with the tensor product of the interpretations: $[\![\Delta]\!] = [\![a_1]\!] \otimes \dots \otimes [\![a_n]\!]$ whenever $\Delta \doteq x_1 : a_1, \dots, x_n : a_n$. Then we define the semantics of values and terms by induction on their definition: we set $[\![c^i_\alpha]\!] = f^i_\alpha \in S_\alpha$. The typing judgment of variables gives the following denotation: $[\![x : a \vdash x : a]\!] \doteq 1_{[\![a]\!]} : [\![a]\!] \to [\![a]\!]$. Now let us consider $f = [\![\Delta \vdash v : a]\!]$. We have the following denotation: $[\![\Delta \vdash \mathtt{inj}_l\, v : a \oplus b]\!] \doteq \iota_l \circ f$, and similarly for the right-projection. If $f = [\![\Delta_1 \vdash v_1 : a_1]\!]$ and $g = [\![\Delta_2 \vdash v_2 : a_2]\!]$, we can define $[\![\Delta_1, \Delta_2 \vdash \langle v_1, v_2 \rangle : a_1 \otimes a_2]\!] \doteq f \otimes g$. When Δ_1, Δ_2 are empty, $f \otimes g : 1 \otimes 1 \to [\![a_1]\!] \otimes [\![a_2]\!]$ will be identified with the type $1 \to [\![a_1]\!] \otimes [\![a_2]\!]$ since there is a *total* isomorphism between 1 and $1 \otimes 1$.

Remark 4.8. *By abuse of notation, we shall write $[\![v]\!]$ in place of $[\![\Delta \vdash v : a]\!]$ when the context is clear.*

The semantics of values v_1, v_2 of type a within a context Δ are morphisms $[\![\Delta]\!] \to [\![a]\!]$. The interesting part in terms of orthogonality is the codomain of these morphisms: intuitively, if they were sets, we would want them to be disjoint. This explains why we are manipulating morphisms such as $\overline{[\![v]\!]^\circ} : [\![a]\!] \to [\![a]\!]$ in the following lemma.

Lemma 4.9 (Orthogonality)**.** *If $v_1 \perp v_2$, then $\overline{[\![v_1]\!]^\circ}\ \overline{[\![v_2]\!]^\circ} = 0$.* □

Lemma 4.9 is proven by induction on the definition of orthogonality over values. It heavily relies on the fact that $\overline{\iota_l^\circ}\ \overline{\iota_r^\circ} = 0$, property of our category $\mathscr{C}$ that we have settled with Definition 4.1.

4.4 Iso semantics

The aim of this section is to build the denotation of isos:

$$\vdash_\omega \{ \mid v_1 \leftrightarrow v'_1 \mid \dots \mid v_n \leftrightarrow v'_n \} : a \leftrightarrow b.$$

Such a denotation should be a morphism $[\![a]\!] \to [\![b]\!]$. It should directly depend on the denotation of the individual clauses, as hinted in Example 3.1. Within the inverse rig category $\mathscr{C}$, we aim at showing that $[\![\omega]\!]\,[\![v_i]\!]$ is equal to $[\![v'_i]\!]\,[\![v_i]\!]^\circ\,[\![v_i]\!]$. This shows that the morphisms that we need to join are of the form $[\![v'_i]\!] \circ [\![v_i]\!]^\circ$: we thus need them to be compatible. Their compatibility is a direct conclusion of Lemmas 4.5, 4.7 and 4.9.

Lemma 4.10. *If $v_1 \perp v_2$ and $v'_1 \perp v'_2$, then $[\![v'_1]\!] \circ [\![v_1]\!]^\circ \asymp [\![v'_2]\!] \circ [\![v_2]\!]^\circ$.* □

Definition 4.11. *For a well-typed iso $\{ \mid v_1 \leftrightarrow v'_1 \mid \dots \mid v_n \leftrightarrow v'_n \}$, thanks to the orthogonality constraints on clauses the morphisms $[\![v'_i]\!] \circ [\![v_i]\!]^\circ$ form a family of pairwise compatible morphisms. We can rely on it to build the semantics of isos as*

$$[\![\vdash_\omega \{ \mid v_1 \leftrightarrow v'_1 \mid v_2 \leftrightarrow v'_2 \ \dots \} : a \leftrightarrow b]\!] = \bigvee_i [\![v'_i]\!] \circ [\![v_i]\!]^\circ : [\![a]\!] \to [\![b]\!]\,. \tag{11}$$

We can finally define the denotational semantics of the remaining terms of the language. Let F be the denotation $[\![\vdash_\omega \omega : a \leftrightarrow b]\!]$ and g be $[\![\vdash_v t : a]\!]$. Then $[\![\vdash_v \omega t : b]\!]$ is $F \circ g$.

4.5 Towards soundness

In order to validate the semantics, a standard expected result is soundness: the fact that whenever $t \to t'$ we also have $[\![t]\!] = [\![t']\!]$. The main difficulty lies in the denotation $[\![\omega t]\!]$ in Definition 4.11. Indeed, suppose that $\omega : a \leftrightarrow b$ is defined as $\{ \mid v_1 \leftrightarrow v_1' \mid v_2 \leftrightarrow v_2' \}$. Whenever a closed term t of type a reduces to w, and that $\omega\ w$ reduces, thanks to the safety properties we know that there exists some i and a substitution σ such that $\sigma[v_i] = w$ and $\omega t \to^* \sigma(v_i')$. We therefore need $[\![\omega t]\!]$ to be equal to $[\![\sigma(v_i')]\!]$, that is, since we should have $[\![t]\!] = [\![w]\!]$ if we had soundness,

$$\left(\left([\![v_1']\!] \circ [\![v_1]\!]^\circ\right) \vee \left([\![v_2']\!] \circ [\![v_2]\!]^\circ\right)\right) \circ [\![w]\!] = [\![v_i']\!] \circ [\![v_i]\!]^\circ \circ [\![w]\!]\,. \tag{12}$$

This amounts to ask that the pattern matching carries over joins in the category.

In the literature [15, 18, 11, 2, 25], the problem is solved by capitalizing on sum-like monoidal tensors, in particular the disjointness tensor of Giles [11]. As sketched in [25], one can follow the following strategy. First, one has to make sure that the denotation maps all types to objects of the form $1 \oplus \cdots \oplus 1$. Then, capitalizing on the structure of language, one shows that the only possible morphisms of type $1 \to 1 \oplus \cdots \oplus 1$ representable in the language are coproduct injections. The behavior of the pattern-matching is then categorically mimicked, getting Equation (12) directly from the properties of the injections.

However, as for instance in the category $\mathrm{PId}_S^{\oplus}$ of Example 4.3, in general join inverse rig categories morphisms of type $1 \to 1 \oplus \cdots \oplus 1$ are not necessarily injections. This calls for a weaker condition, independent from the coproduct. This is the subject of the next section.

5 Pattern-matching categories

As discussed in Section 4.5, the denotation of iso is a join over several morphisms, each mapping a particular pattern to its image through the iso. In the operational semantics, the "choice" of the pattern is expressed over the notion of orthogonality of values: indeed, if a value matches a pattern, this value is necessarily orthogonal to the other patterns. This is ensured by the straightforward definition of orthogonality. We have presented in Section 4.5 a way to transfer this notion of orthogonality to morphisms in join inverse rig categories, based on the structure of disjointness tensor. In this section, we discuss a notion of categorical pattern-matching independently from disjointness tensors, in particular generalizing the approach of [11, 25].

This is the main contribution of the paper.

Example 5.1. *To illustrate our approach, consider a language only consisting of (1) tensors and (2) enum-types $\alpha = \{c_\alpha\}$ and $\beta = \{c_\beta^1, c_\beta^2, c_\beta^3\}$. Let us then consider the full subcategory of PInj, with objects consisting of the sets of cardinality power of 3. This sub-category can serve as a sound model of the language, including the pattern-matching. Note how the handling of the pattern-matching is independent from any disjointness tensor as the category does not feature coproducts.*

5.1 Non decomposability and pattern-matching categories

Concretely, in order to get pattern-matching, we manipulate expressions of the form $(f \vee g)h$ and the idea would be to find a property over h which would lead to $(f \vee g)h$ being equal either to fh or to gh, without having to rely on coproduct injections. For this matter, we introduce several definitions of *non decomposability* – roughly meaning that the morphism cannot be written with a join.

Definition 5.2 (Strongly non decomposable)**.** *A morphism $h : A \to B$ in a join inverse category is called strongly non decomposable (snd) when if $h = f \vee g$ with $f, g : A \to B$, then $f = 0$ or $g = 0$.*

Definition 5.3 (Linearly non decomposable)**.** *A morphism $h : A \to B$ in a join inverse category is called linearly non decomposable (lnd) when if $f \leq h$ and $g \leq h$ with $f, g : A \to B$, then $f \leq g$ or $g \leq f$.*

Definition 5.4 (Weakly non decomposable)**.** *A morphism $h : A \to B$ in a join inverse category is called weakly non decomposable (wnd) when if $h = f \vee g$ with $f, g : A \to B$, then $f \leq g$ or $g \leq f$.*

Remark 5.5. *Let us observe that snd $\Rightarrow$ lnd $\Rightarrow$ wnd.*

Even if they look alike, these definitions imply large differences. A strongly non decomposable morphism does not have any morphism strictly less than itself, except 0. On the other hand, linear non decomposability allows a certain order of morphisms, with a linear constraint. Finally the weak form gives very few information, as shown in the following example.

Example 5.6 (Weakly non decomposable morphism)**.** *Consider a subcategory of PInj with one object $\{a, b, c\}$ and with the identity and the four partial isos: f of domain $\{a, b\}$, g of domain $\{a\}$, h of domain $\{b\}$ and 0 of empty domain. This subcategory is a join inverse category. In this subcategory, the identity is weakly non decomposable: it cannot be written as a join without itself in it. Nonetheless, the morphism f is decomposable: the property of weakly non decomposability is not downwards closed, contrary to the two other notions. Weak non-decomposability then lead to very different results from the other notions.*

Definition 5.7 (Pattern-matching)**.** *A weakly (resp. linearly, strongly) pattern-matching category $\mathscr{C}$ is a join inverse category in which for all $f, g \in Hom_{\mathscr{C}}(A, B)$ and $h \in Hom_{\mathscr{C}}(C, A)$, if $f \smile g$ and h is weakly (resp. linearly, strongly) non decomposable, then*

$$(f \vee g)h = fh \quad \text{or} \quad (f \vee g)h = gh.$$

Note that we do not need the compatibility of the inverses to define the pattern-matching.

This straightforward definition of what we would like to have in our category is, however, difficult to grasp and manipulate; therefore we introduce an equivalent presentation: the correspondence between the two definitions is shown in Theorem 5.9.

Definition 5.8 (Consistency)**.** *A join inverse category is said weakly (resp. linearly, strongly) consistent when $h : A \to B$ weakly (resp. linearly, strongly) non decomposable implies that for all morphism $f : B \to C$, the morphism fh is weakly (resp. linearly, strongly) non decomposable.*

Theorem 5.9. *A weakly (resp. linearly, strongly) consistent join inverse category is a weakly (resp. linearly, strongly) pattern-matching category.* □

Indeed, if a join $f \vee g$ is composed on the left with a non decomposable morphism h, consistency ensures that $fh \vee gh$ stays non decomposable. This means that it can be only be one of the morphisms that form the join, which is exactly pattern-matching.

Theorem 5.10. *A weakly pattern-matching category is weakly consistent.* □

This theorem validates the choices of definition made above, and it underlines a strong link between pattern-matching and non decomposability.

Notwithstanding, an inverse category is not necessarily a weakly pattern-matching category:

Example 5.11 (Not weakly consistent)**.** *Consider the subcategory of PInj presented in Example 5.6. Although $id_{\{a,b,c\}}$ is weakly non decomposable, the composition $(g \vee h)id_{\{a,b,c\}}$ is equal to $g \vee h = f$, different from both g and h: the subcategory is not weakly consistent.*

Thus, if we were willing to work with weak non decomposability, not every inverse category can be involved. As shown below, the notions of linear and strong non decomposability let us consider any inverse category. Of course, the complexity does not disappear: it simply moves from the definition of the category to the number of morphisms that we can consider, since linear and strong non decomposability are far more demanding properties than the weak one.

Theorem 5.12. *A join inverse category is strongly consistent.* □

Even if it is not a surprise, this result is interesting. Strongly non decomposability is a very demanding property; but since it implies some morphisms being equal to 0, composing on the left naturally does not change the property.

Example 5.13. *A morphism in a join inverse category can be linearly non decomposable without being strongly non decomposable. Indeed, consider the (join inverse) sub-category of PInj whose only object is $\{a,b\}$, and with morphisms 0, id and f, the partial identity whose domain is $\{a\}$. Then we have $0 < f < id_{\{a,b\}}$. While it is linearly non-decomposable, the identity is not strongly non-decomposable.*

Fortunately, we still have the following result:

Theorem 5.14. *A join inverse category is linearly consistent.* □

Remark 5.15. *Linear decomposability is then the best property to consider for pattern-matching. In the context of pattern-matching categories, linear non decomposable maps then essentially correspond to "pure" values waiting to be matched against a set of "orthogonal" clauses —this orthogonality being captured by the compatibility of the morphisms representing the clauses.*

5.2 The case of our language

The previous subsection being an attempt to develop a categorical theory of pattern-matching, we can now take a look at how it would apply to the case of the minimal language of Section 2, whose only base type is the `unit` type with one single constant. Its semantics in a join inverse rig category $\mathscr{C}$ is straightforwardly given by the identity id_1 over 1, the unit of the monoidal structure. We are in the context of Section 4.5, where all closed values are denoted with morphisms $1 \to 1 \oplus \cdots \oplus 1$, but where in the category, in general morphisms of this type are not necessarily injections —and there are more than one morphism $1 \to 1$.

Theorem 5.16 (Denotation of closed values)**.** *In a join inverse rig category $\mathscr{C}$, if id_1 is weakly (resp. linearly, strongly) non decomposable, then the denotation of a closed value v of the minimal language is weakly (resp. linearly, strongly) non decomposable.* □

Then in any join inverse category with id_1 linearly —or strongly— non decomposable, in the context of Sec. 4 the previous results allow us to apply pattern-matching: $(\bigvee_i [\![v'_i]\!]\,[\![v_i]\!]^\circ)\,[\![v]\!] = [\![v'_{i_0}]\!]\,[\![v_{i_0}]\!]^\circ\,[\![v]\!]$.

Example 5.17. *Example 4.3 is a nice way to picture a id_1 that is non-trivial. Suppose that S contains at least two elements. Then id_1 in $\mathrm{PId}_S^{\oplus}$ is not even weakly non decomposable; if we consider the sub-category $\mathrm{SubPId}^{\oplus}_{\{a,b,c\}}$ with $S = \{a,b,c\}$, and with corresponding morphisms on PId_S $\{a\}$, $\{a,b\}$ and $\{a,b,c\}$ only, then id_1 is linearly non decomposable but not strongly so.*

Note how $\mathrm{SubPId}^{\oplus}_{\{a,b,c\}}$ is indeed a join inverse rig category, as the tensor is derived from the intersection on sets.

5.3 Soundness and adequacy

From Theorem 5.16 we can derive the fact that any join inverse rig category where id_1 is linearly non-decomposable forms a sound and adequate model for the minimal language. The remaining results assume such an ambient semantics.

Lemma 5.18 (Substitution). *In the minimal language, for any value $\Delta \vdash_v v_{i_0} : a$, $\Delta \vdash_v v'_{i_0} : b$ and $\vdash_v v : a$, we have $[\![\sigma(v'_{i_0})]\!] = [\![v'_{i_0}]\!]\,[\![v_{i_0}]\!]^\circ\,[\![v]\!]$.* □

Corollary 5.19 (Soundness). *If $t \to t'$, then $[\![t]\!] = [\![t']\!]$.* □

Definition 5.20 (Operational equivalence). *Let's consider two terms t_1, t_2. We define $t_1 \equiv_{op} t_2$ as both $t_1 \to^* v$ and $t_2 \to^* v$.*

Theorem 5.21 (Adequacy). *If t_1 and t_2 are closed, well-typed terms, then $t_1 \equiv_{op} t_2$ iff $[\![t_1]\!] = [\![t_2]\!]$.* □

6 Discussion and conclusion

Consider again Example 5.17. Following the same technique presented in the literature [25], one could be able extract from the minimal language of Section 2 a sub-category of $\mathrm{PId}^{\oplus}_{\{a,b,c\}}$ able to capture a notion of pattern-matching. However, this approach does not tell us anything about the categorical structures required to capture it: the machinery presented in Section 5 aims at answering this question.

In particular, we discussed in Example 5.17 a subcategory of $\mathrm{PId}^{\oplus}_{\{a,b,c\}}$: in $\mathrm{SubPId}^{\oplus}_{\{a,b,c\}}$, the morphism id_1 is linearly non-decomposable; Lemma 5.18 then shows that Equation (12) of Section 4.5 is correct in $\mathrm{SubPId}^{\oplus}_{\{a,b,c\}}$ while Theorem 5.21 guarantees that this subcategory is actually adequate: it strictly sits in between the image of the language and the whole category $\mathrm{PId}^{\oplus}_{\{a,b,c\}}$.

In general, although the notions of pattern-matching and linear non-decomposability we presented in this paper are sufficient to recover a notion of pattern-matching, a question that is however left open is whether they are necessary. This is left as future work.

Acknowledgments

This work was supported in part by the French National Research Agency (ANR) under the research project SoftQPRO ANR-17-CE25-0009-02, by the DGE of the French Ministry of Industry under the research project PIA-GDN/QuantEx P163746-484124 and by the STIC-AmSud project Qapla' 21-SITC-10.

References

[1] Bogdan Aman, Gabriel Ciobanu, Robert Glück, Robin Kaarsgaard, Jarkko Kari, Martin Kutrib, Ivan Lanese, Claudio Antares Mezzina, Lukasz Mikulski, Rajagopal Nagarajan, Iain C. C. Phillips, G. Michele Pinna, Luca Prigioniero, Irek Ulidowski & Germán Vidal (2020): *Foundations of Reversible Computation*. In Irek Ulidowski, Ivan Lanese, Ulrik Pagh Schultz & Carla Ferreira, editors: *Reversible Computation: Extending Horizons of Computing - Selected Results of the COST Action IC1405*, *Lecture Notes in Computer Science* 12070, Springer, pp. 1–40, doi:10.1007/978-3-030-47361-7_1.

[2] Holger Bock Axelsen & Robin Kaarsgaard (2016): *Join Inverse Categories as Models of Reversible Recursion*. In Bart Jacobs & Christof Löding, editors: *Proceedings of the 19th International Conference on Foundations of Software Science and Computation Structures (FOSSACS'16)*, *Lecture Notes in Computer Science* 9634, Springer, Eindhoven, The Netherlands, pp. 73–90, doi:10.1007/978-3-662-49630-5_5.

[3] Charles H Bennett (1973): *Logical reversibility of computation*. *IBM Journal of Research and Development* 17(6), pp. 525–532, doi:10.1147/rd.176.0525.

[4] Charles H. Bennett (2000): *Notes on the history of reversible computation*. *IBM Journal of Research and Development* 44(1), pp. 270–278, doi:10.1147/rd.441.0270.

[5] Jacques Carette & Amr Sabry (2016): *Computing with Semirings and Weak Rig Groupoids*. In Peter Thiemann, editor: *Proceedings of the 25th European Symposium on Programming Languages and Systems (ESOP'16)*, *Lecture Notes in Computer Science* 9632, Springer, Eindhoven, The Netherlands, pp. 123–148, doi:10.1007/978-3-662-49498-1_6.

[6] Michael Kirkedal Carøe (2012): *Design of Reversible Computing Systems*. Ph.D. thesis, University of Copenhagen, Denmark.

[7] Kostia Chardonnet, Alexis Saurin & Benoît Valiron (2020): *Toward a Curry-Howard Equivalence for Linear, Reversible Computation - Work-in-Progress*. In Ivan Lanese & Mariusz Rawski, editors: *Proceedings of the 12th International Conference on Reversible Computation (RC 2020)*, *Lecture Notes in Computer Science* 12227, Springer, pp. 144–152, doi:10.1007/978-3-030-52482-1_8.

[8] J. Robin B. Cockett & Stephen Lack (2002): *Restriction Categories I: Categories of Partial Maps*. *Theoretical Computer Science* 270(1), pp. 223–259, doi:10.1016/S0304-3975(00)00382-0.

[9] J. Robin B. Cockett & Stephen Lack (2003): *Restriction categories II: partial map classification*. *Theoretical Computer Science* 294(1), pp. 61–102, doi:10.1016/S0304-3975(01)00245-6.

[10] Robin Cockett & Stephen Lack (2007): *Restriction Categories III: Colimits, Partial Limits and Extensivity*. *Mathematical Structures in Computer Science* 17(4), pp. 775–817, doi:10.1017/S0960129507006056.

[11] Brett Gordon Giles (2014): *An Investigation of Some Theoretical Aspects of Reversible Computing*. Ph.D. thesis, University of Calgary, doi:10.11575/PRISM/24917.

[12] Robert Glück & Robin Kaarsgaard (2018): *A categorical foundation for structured reversible flowchart languages: Soundness and adequacy*. *Log. Methods Comput. Sci.* 14(3), doi:10.23638/LMCS-14(3:16)2018.

[13] Robert Glück, Robin Kaarsgaard & Tetsuo Yokoyama (2019): *Reversible Programs Have Reversible Semantics*. In Emil Sekerinski, Nelma Moreira, José N. Oliveira, Daniel Ratiu, Riccardo Guidotti, Marie Farrell, Matt Luckcuck, Diego Marmsoler, José Campos, Troy Astarte, Laure Gonnord, Antonio Cerone, Luis Couto, Brijesh Dongol, Martin Kutrib, Pedro Monteiro & David Delmas, editors: *Formal Methods. FM 2019 International Workshops - Porto, Portugal, October 7-11, 2019, Revised Selected Papers, Part II*, *Lecture Notes in Computer Science* 12233, Springer, pp. 413–427, doi:10.1007/978-3-030-54997-8_26.

[14] Xiuzhan Guo (2012): *Products, Joins, Meets, and Ranges in Restriction Categories*. Ph.D. thesis, University of Calgary, doi:10.11575/PRISM/4745.

[15] Esfandiar Haghverdi (2000): *A Categorical Approach to Linear Logic, Geometry of Proofs and Full Completeness*. Ph.D. thesis, University of Ottawa, doi:10.20381/ruor-16218.

[16] Chris Heunen, Robin Kaarsgaard & Martti Karvonen (2018): *Reversible Effects as Inverse Arrows*. In Sam Staton, editor: *Proceedings of the 34th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIV)*, *Electronic Notes in Theoretical Computer Science* 341, Elsevier, Dalhousie University, Halifax, Canada, pp. 179–199, doi:10.1016/j.entcs.2018.11.009.

[17] Chris Heunen & Martti Karvonen (2015): *Reversible Monadic Computing*. In Dan Ghica, editor: *Proceedings of the 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI)*, *Electronic Notes in Theoretical Computer Science* 319, Nijmegen, The Netherlands, pp. 217–237, doi:10.1016/j.entcs.2015.12.014.

[18] Naohiko Hoshino (2012): *A Representation Theorem for Unique Decomposition Categories*. In Ulrich Berger & Michael W. Mislove, editors: *Proceedings of the 28th Conference on the Mathematical Foundations of Programming Semantics, MFPS 2012, Bath, UK, June 6-9, 2012*, *Electronic Notes in Theoretical Computer Science* 286, Elsevier, pp. 213–227, doi:10.1016/j.entcs.2012.08.014.

[19] Petur Andrias Højgaard Jacobsen, Robin Kaarsgaard & Michael Kirkedal Thomsen (2018): *CoreFun : A Typed Functional Reversible Core Language*. In Jarkko Kari & Irek Ulidowski, editors: *Reversible Computation - 10th International Conference, RC 2018, Leicester, UK, September 12-14, 2018, Proceedings*, *Lecture Notes in Computer Science* 11106, Springer, pp. 304–321, doi:10.1007/978-3-319-99498-7_21.

[20] Rosham P. James & Amr Sabry (2014): *Theseus: A High-Level Language for Reversible Computing*. Draft, available at https://legacy.cs.indiana.edu/~sabry/papers/theseus.pdf.

[21] Roshan P. James & Amr Sabry (2012): *Information effects*. In John Field & Michael Hicks, editors: *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, ACM, Philadelphia, Pennsylvania, USA, pp. 73–84, doi:10.1145/2103656.2103667.

[22] Robin Kaarsgaard (2019): *Condition/Decision Duality and the Internal Logic of Extensive Restriction Categories*. In Barbara König, editor: *Proceedings of the 35th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXV)*, *Electronic Notes in Theoretical Computer Science* 347, London, UK, pp. 179–202, doi:10.1016/j.entcs.2019.09.010.

[23] Robin Kaarsgaard (2019): *Inversion, Iteration, and the Art of Dual Wielding*. In Michael Kirkedal Thomsen & Mathias Soeken, editors: *Proceedings of the 11th International Conference on Reversible Computation (RC 2019)*, *Lecture Notes in Computer Science* 11497, Springer, Lausanne, Switzerland, pp. 34–50, doi:10.1007/978-3-030-21500-2_3.

[24] Robin Kaarsgaard, Holger Bock Axelsen & Robert Glück (2017): *Join inverse categories and reversible recursion*. *Journal of Logical and Algebraic Methods in Programming* 87, pp. 33–50, doi:10.1016/j.jlamp.2016.08.003.

[25] Robin Kaarsgaard & Mathys Rennela (2021): *Join inverse rig categories for reversible functional programming, and beyond*. Draft, available at arXiv:2105.09929.

[26] Robin Kaarsgaard & Niccolò Veltri (2019): *En Garde! Unguarded Iteration for Reversible Computation in the Delay Monad*. In Graham Hutton, editor: *Proceedings of the 13th International Conference on Mathematics of Program Construction (MPC 2019)*, *Lecture Notes in Computer Science* 11825, Springer Verlag, Porto, Portugal, pp. 366–384, doi:10.1007/978-3-030-33636-3_13.

[27] J. Kastl (1979): *Inverse Categories*. In: *Algebraische Modelle, Kategorien und Gruppoide*, Studien zur Algebra und ihre Anwendungen, Band 7, Berlin, Akademie-Verlag, pp. 51–60.

[28] Rolf Landauer (1961): *Irreversibility and Heat Generation in the Computing Process*. *IBM Journal of Research and Development* 5(3), pp. 183–191, doi:10.1147/rd.53.0183.

[29] Christopher Lutz (1986): *Janus: a time-reversible language*. Letter to Rolf Landauer, posted online by Tetsuo Yokoyama on http://www.tetsuo.jp/ref/janus.html.

[30] Amr Sabry, Benoît Valiron & Juliana Kaizer Vizzotto (2018): *From Symmetric Pattern-Matching to Quantum Control*. In Christel Baier & Ugo Dal Lago, editors: *Proceedings of the 21st International Conference on Foundations of Software Science and Computation Structures (FOSSACS'18)*, *Lecture Notes in Computer Science* 10803, Springer, Thessaloniki, Greece, pp. 348–364, doi:10.1007/978-3-319-89366-2_19.

[31] Mehdi Saeedi & Igor L. Markov (2013): *Synthesis and Optimization of Reversible Circuits - a Survey*. *ACM Computing Surveys* 45(2), pp. 21:1–21:34, doi:10.1145/2431211.2431220.

[32] Michael Kirkedal Thomsen & Holger Bock Axelsen (2015): *Interpretation and programming of the reversible functional language RFUN*. In Ralf Lämmel, editor: *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages, IFL 2015, Koblenz, Germany, September 14-16, 2015*, ACM, pp. 8:1–8:13, doi:10.1145/2897336.2897345.

[33] Robert Wille, Eleonora Schönborn, Mathias Soeken & Rolf Drechsler (2016): *SyReC: A hardware description language for the specification and synthesis of reversible circuits*. *Integration, the VLSI Journal* 53, pp. 39–53, doi:10.1016/j.vlsi.2015.10.001.

[34] Tetsuo Yokoyama (2010): *Reversible Computation and Reversible Programming Languages*. In Irek Ulidowski, editor: *Proceedings of the Workshop on Reversible Computation (RC'09)*, *Electronic Notes in Theoretical Computer Science* 253(6), Elsevier, York, UK, pp. 71–81, doi:10.1016/j.entcs.2010.02.007.

[35] Tetsuo Yokoyama, Holger Bock Axelsen & Robert Glück (2012): *Towards a Reversible Functional Language*. In Alexis De Vos & Robert Wille, editors: *Revised Papers of the Third International Workshop on Reversible Computation (RC'11)*, *Lecture Notes in Computer Science* 7165, Springer, Gent, Belgium, pp. 14–29, doi:10.1007/978-3-642-29517-1_2.

[36] Tetsuo Yokoyama, Holger Bock Axelsen & Robert Glück (2016): *Fundamentals of reversible flowchart languages*. *Theoretical Computer Science* 611, pp. 87–115, doi:10.1016/j.tcs.2015.07.046.

[37] Tetsuo Yokoyama & Robert Glück (2007): *A reversible programming language and its invertible self-interpreter*. In G. Ramalingam & Eelco Visser, editors: *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2007, Nice, France, January 15-16, 2007*, pp. 144–153, doi:10.1145/1244381.1244404.