Join inverse rig categories for reversible functional programming, and beyond

Kaarsgaard, Robin; Rennela, Mathys

Published in:
Proceedings 37th Conference on Mathematical Foundations of Programming Semantics

DOI:
10.4204/EPTCS.351.10

Publication date:
2021

Document version:
Final published version

Join Inverse Rig Categories for Reversible Functional Programming, and Beyond

Robin Kaarsgaard
University of Edinburgh
Edinburgh, United Kingdom
robin.kaarsgaard@ed.ac.uk

Mathys Rennela
INRIA
Paris, France
mathys.rennela@inria.fr

Reversible computing is a computational paradigm in which computations are deterministic in both the forward and backward direction, so that programs have well-defined forward *and* backward semantics. We investigate the formal semantics of the reversible functional programming language Rfun. For this purpose we introduce join inverse rig categories, the natural marriage of join inverse categories and rig categories, which we show can be used to model the language Rfun, under reasonable assumptions. These categories turn out to be a particularly natural fit for reversible computing as a whole, as they encompass models for other reversible programming languages, notably Theseus and reversible flowcharts. This suggests that join inverse rig categories really are the categorical models of reversible computing.

1 Introduction

Since the early days of theoretical computer science, the quest for the mathematical description of (functional) programming languages has led to a substantial body of work. In reversible computing, every program is *reversible*, i.e., both *forward* and *backward* deterministic. But why study such a peculiar paradigm of computation at all? While the daily operations of our computers are irreversible, the physical devices which execute them are fundamentally reversible. In the paradigm of quantum computation, the physical operations performed by a scalable quantum computer intrinsically rely on quantum mechanics, which is reversible. Landauer [34] has famously argued, through what has later been coined *Landauer's principle*, that the erasure of a bit of information is inexorably linked to the dissipation of energy as heat (which has since seen both formal [1] and experimental [8] verification). On its own, this constitutes a reasonable argument for the study of reversible computing, as this model of computation sidesteps this otherwise inevitable energy dissipation by avoiding the erasure of information altogether.

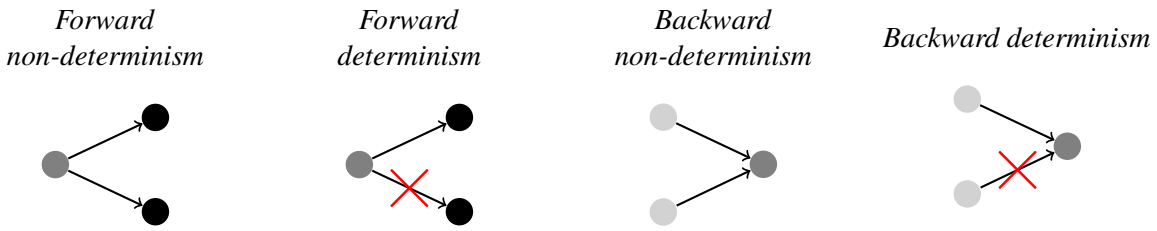
And although studying reversibility can be motivated by issues raised by the laws of thermodynamics which arguably constitute a theoretical limit of Moore's law [37], reversibility arises not only in quantum computing (see e.g., [2]), but also has its own circuit models [17, 44], Turing machines [4, 7] and automata [32, 33]. Moreover, the notion of reversibility has seen applications in areas spanning from high-performance computing [41] to process calculi [16] and robotics [42, 43], to name a few.

There is an increasing interest for the theoretical study of the reversible computing paradigm (see, e.g., [3]). The present work provides a building block in the study of reversible computing through the lens of category theory.

Ana Sokolova (Eds.): MFPS 2021
EPTCS 351, 2021, pp. 152–167, doi:10.4204/EPTCS.351.10

1.1 Reversible programming primer

Almost all programming languages in use today (with some notable exceptions) guarantee that programs are *forward deterministic* (or simply *deterministic*), in the sense that any current computation state uniquely determines the next computation state. Few such languages, however, guarantee that programs are *backward deterministic*, i.e. that any current computation state uniquely determines the *previous* computation state. For example, assigning a constant value to a variable in an imperative programming language is forward deterministic, but not backward deterministic (as one generally has no way of determining the value stored in the variable prior to this assignment).



Programming languages which guarantee both forward and backward determinism of programs are called *reversible*. Though there are numerous examples of such reversible programming languages (see, *e.g.*, [27,46]), we focus here on the reversible functional programming language Rfun [45].

Rfun is an untyped reversible functional programming language (see an example program, computing Fibonacci pairs, to the right, due to [45]) similar in style to the Lisp family of programming languages. As a consequence of reversibility, all functions in Rfun are *partial injections*, *i.e.*, whenever some function f is defined at points x and y, $f(x) = f(y)$ implies $x = y$. Being untyped, values in Rfun come in the form of Lisp-style symbols and constructors.

$$
\begin{array}{l}
\mathit{plus}\ \langle x,y\rangle \triangleq \textbf{case}\ y\ \textbf{of} \\
\qquad\begin{array}{lcl}
Z & \to & \lfloor\langle x\rangle\rfloor \\
S(u) & \to & \textbf{let}\ \langle x',u'\rangle = \mathit{plus}\ \langle x,u\rangle\ \textbf{in} \\
 & & \langle x',S(u')\rangle
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\mathit{fib}\ n \triangleq \textbf{case}\ n\ \textbf{of} \\
\qquad\begin{array}{lcl}
Z & \to & \langle S(Z),S(Z)\rangle \\
S(m) & \to & \textbf{let}\ \langle x,y\rangle = \mathit{fib}\ m\ \textbf{in} \\
 & & \textbf{let}\ z = \mathit{plus}\ \langle y,x\rangle\ \textbf{in}\ z
\end{array}
\end{array}
$$

Pattern matching and variable binding is supported by means of (slightly restricted) forms of case expressions and let bindings, and iteration by means of general recursion. These restriction ensure, essentially, that the inverse of a case expression is also a case expression.

$$
\begin{array}{l}
\mathit{plus}^{-1}\ z \triangleq \textbf{case}\ z\ \textbf{of} \\
\qquad\begin{array}{lcl}
\lfloor\langle x\rangle\rfloor & \to & \langle x,Z\rangle \\
\langle x',S(u')\rangle & \to & \textbf{let}\ \langle x,u\rangle = \mathit{plus}^{-1}\ \langle x',u'\rangle\ \textbf{in} \\
 & & \langle x,S(u)\rangle
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\mathit{fib}^{-1}\ z \triangleq \textbf{case}\ z\ \textbf{of} \\
\qquad\begin{array}{lcl}
\langle S(Z),S(Z)\rangle & \to & Z \\
z' & \to & \textbf{let}\ \langle y,x\rangle = \mathit{plus}^{-1}\ z'\ \textbf{in} \\
 & & \textbf{let}\ m = \mathit{fib}^{-1}\ \langle x,y\rangle\ \textbf{in}\ S(m)
\end{array}
\end{array}
$$

In order to guarantee backward determinism (and consequently reversibility), Rfun imposes a few restrictions on function definitions not usually present in irreversible functional programming languages. Firstly, while a given variable may only appear once in a pattern (as is also the case in, *e.g.*, Haskell and the ML family), it must also occur exactly once in the body. Secondly, the result of a function call must be bound in a let binder before use. Thirdly, the leaf expression of any branch in a case expression must not match *any* leaf expression in a branch preceding it: This is known as the *symmetric first-match policy*, and guarantees that case-expressions can be evaluated normally (i.e., in prioritised order from top to bottom) without impacting reversibility.

Together, these three restrictions guarantee reversibility. One might wonder whether these hinder

expressivity too much; fortunately, this is not so, as Rfun is r-Turing complete [45], that is: Rfun can simulate any reversible Turing machine [7].

Another peculiarity of Rfun has to do with duplication of values. Even though values *can* be (de)duplicated reversibly, the linear use policy on variables hinders this. To allow for (de)duplication of values, a special *duplication/equality* operator $\lfloor\cdot\rfloor$ is introduced (see Figure 1). Note that this operator can be used both as a value and as a pattern; in the former use case, it is used to (de)duplicate values, and in the latter, to test whether values are identical.

As a reversible programming language, Rfun has the property that it is syntactically closed under inverses. That is to say, if p is an Rfun program, there exists another Rfun program p' such that p' computes the semantic inverse of p. This is witnessed by a *program inverter* [45]. For example, the inverses of the addition and Fibonacci pair functions shown earlier are given above.

1.2 Join inverse category theory

In Section 2, we focus on categorical structures in which one can conveniently model the mathematical foundations of reversible computing. A brainchild of Cockett & Lack [12, 13, 15], restriction categories are categories with an abstract notion of partiality, associating to each morphism $f : A \to B$ a "partial identity" $\overline{f} : A \to A$ satisfying $f \circ \overline{f} = f$ and other axioms. In particular, this gives rise to *partial isomorphisms*, which are morphisms $f : A \to B$ associated with partial inverses $f^\dagger : B \to A$ such that $f^\dagger \circ f = \overline{f}$ and $f \circ f^\dagger = \overline{f^\dagger}$. In this context, a *total* map is a morphism $f : A \to B$ such that $\overline{f}$ is the identity on A.

$$\lfloor\langle x\rangle\rfloor = \langle x, x\rangle$$
$$\lfloor\langle x, y\rangle\rfloor = \begin{cases} \langle x\rangle & \text{if } x = y \\ \langle x, y\rangle & \text{if } x \neq y \end{cases}$$

Figure 1: The operator $\lfloor\cdot\rfloor$.

A restriction category in which all morphisms are partial isomorphisms is called an *inverse category*. Such categories have been considered by the semigroup community for decades [24, 31, 35], though they have more recently been rediscovered in the framework of restriction category theory, and considered as models of reversible computing [5, 18, 21, 28]. The category **PInj** of sets and partial injections is the canonical example of an inverse category. In fact, every (locally small) inverse category can be faithfully embedded in it [12, 22].

In line with [5], we focus here on a particular class of domain-theoretic inverse categories, namely join inverse categories. Informally, join inverse categories are inverse categories in which joins (i.e. least upper bounds) of morphisms exist in such a way that the partial identity of a join is the join of the partial identities, among with other coherence axioms.

In this setting, join inverse rig categories are join inverse categories equipped with monoidal products $X \otimes Y$ and monoidal sums $X \oplus Y$ for every pair of objects X and Y, together with a isomorphisms which distribute products over sums and annihilate products with the additive unit, subject to preservation of joins and the usual coherence laws of rig categories. Every inverse category embeds in such a join inverse rig category via Wagner-Preston (see also Theorem 3.5).

1.3 Categorical models of reversible computing

The present work is predominantly concerned with the axiomatization of categorical models of reversible computing, following similar approaches based on presheaf-theoretic models, in the semantics of quantum computing [36, 38–40] and reversible computing [5].

In order to construct denotational models of the terms of the language Rfun, we adopt the categorical formalism of join inverse rig categories. Since morphisms in inverse categories are all partial isomorphisms, inverse categories have been suggested as models of reversible functional programming languages [18], and the presence of joins has been shown [5, 28] to induce fixed point operators for modelling reversible recursion.

In short, Rfun is an untyped first-order language in which the arguments of functions are organized in left expressions (patterns) given by the grammar

$$l ::= x \mid c(l_1, \ldots, l_n) \mid \lfloor l \rfloor$$

where variables x and constructors c are taken from denumerable alphabets $\mathcal{V}$ respectively S. In other words, a function definition $f\, l \triangleq l'$ takes a pattern l as argument and organizes its output as an expression l'.

Our work involves an unorthodox way of thinking about the denotational semantics of a function in a functional programming language to be explicitly constructed piecemeal by the (partial) denotations of its individual branches. Categorically, this means that the denumerable alphabet S is denoted by the (least) fix point of the functor $F : X \mapsto X \oplus 1$, which is given by algebraic compactness as the initial F-algebra [6] and corresponds to the denotation of the recursive type of natural numbers. Then, every value $c(l_1, \ldots, l_n)$ is naturally denoted by induction as a tree which has a root labelled by the symbol c, with a branch to every subtree l_i for $1 \leqslant i \leqslant n$. We detail this construction in Section 5.

Finally, in order to denote the duplication/equality operator and the case expressions of Rfun, we require the notions of *decidable equality* and *decidable pattern matching*. Recall that in set theory, a set is *decidable* (or *has decidable equality*) whenever any pair of elements is either equal or different. Using the interpretation of restriction idempotents as propositions, we express decidability through the presence of *complementary* restriction idempotents.

We conclude our study in Section 6 with a definition of categorical models of reversible computing as join inverse rig categories with decidable equality and decidable pattern matching.

2 Join inverse category theory

While we assume prior knowledge of basic category theory, including monoidal categories and string diagrams, we briefly introduce some of the less well-known material on restriction and inverse categories (see also, e.g., [12, 13, 15, 18, 21]).

Definition 2.1. A *restriction structure* on a category consists of an operator mapping each morphism $f : A \to B$ to a morphism $\overline{f} : A \to A$ (called *restriction idempotent*) such that the following properties are satisfied:

(i) $f \circ \overline{f} = f$,

(ii) $\overline{f} \circ \overline{g} = \overline{g} \circ \overline{f}$ for all $g : A \to C$,

(iii) $\overline{g \circ \overline{f}} = \overline{g} \circ \overline{f}$ for all $g : A \to C$,

(iv) $\overline{g} \circ f = f \circ \overline{g \circ f}$ for all $g : B \to C$.

A category with a restriction structure is called a *restriction category*. A *total* map is a morphism $f : A \to B$ such that $\overline{f} = \mathrm{id}_A$. Using an analogy from topology, the collection of restriction idempotents on an object A is denoted $\mathcal{O}(A)$, and is sometimes even called the *opens* on A. (Indeed, in the category **PTop** of topological spaces and partial continuous functions, the restriction idempotents coincide with the open sets.) We recall now some basic properties of restriction idempotents:

Lemma 2.2. *In any restriction category, we have for all suitable f and g that*

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

As a trivial example, any category is a restriction category when equipped with the trivial restriction structure mapping $\overline{f} = 1_A$ for all $f : A \to B$.

Definition 2.3. A morphism $f : A \to B$ in a restriction category is a partial isomorphism whenever there exists a morphism $f^\dagger : B \to A$, the partial inverse of f, such that $f^\dagger \circ f = \overline{f}$ and $f \circ f^\dagger = \overline{f^\dagger}$.

Note that the definite article – *the* partial inverse – is justified, as partial inverses are unique whenever they exist.

Definition 2.4. An *inverse category* is a restriction category in which every morphism is a partial isomorphism.

It is worth noting that this is not the only definition of an inverse category: historically, this mathematical structure has been defined as the categorical extension of inverse semigroups rather than as a particular class of restriction categories (see e.g. [24, 31, 35]).

Definition 2.5. A *zero object* in a restriction (or inverse) category is said to be a *restriction zero* iff $\overline{0_{A,A}} = 0_{A,A}$ for every zero endomorphism $0_{A,A}$.

Definition 2.6. Parallel morphisms $f, g : A \to B$ of an inverse category are said to be *inverse compatible*, denoted $f \asymp g$, if the following hold:

(i) $g \circ \overline{f} = f \circ \overline{g}$
(ii) $g^\dagger \circ \overline{f^\dagger} = f^\dagger \circ \overline{g^\dagger}$

By extension, one says that $S \subseteq \mathrm{Hom}(A,B)$ is inverse compatible if $s \asymp t$ for each $s,t \in S$. Note that *disjoint* morphisms (those satisfying $g \circ \overline{f} = 0_{A,B}$) are always inverse compatible.

The present work focuses on join inverse categories. The definition of such categories relies on the fact that in a restriction category $\mathbf{C}$, every hom-set $\mathbf{C}(A,B)$ gives rise to a poset when equipped with the following partial order: $f \leqslant g$ if and only if $g \circ \overline{f} = f$.

Definition 2.7 ([21])**.** An inverse category is a (countable) *join inverse category* if it has a restriction zero object, and satisfies that for all (countable) inverse compatible subsets S of all hom sets $\mathrm{Hom}(A,B)$, there exists a morphism $\bigvee_{s\in S} s$ such that

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

On that matter, it is important to mention that there are significant mathematical results about join *inverse* categories. In particular, there is an adjunction between the categories of join restriction categories and join inverse categories [5].

In a join inverse category, every restriction idempotent $\overline{e}$ has a *pseudocomplement* $\overline{e}^\perp$ given by $\bigvee_{\overline{e'} \in C(\overline{e})} \overline{e'}$ where $C(\overline{e}) = \{\overline{e'} \in \mathscr{O}(X) \mid \overline{e'} \circ \overline{e} = 0_{X,X}\}$. Indeed, the opens $\mathscr{O}(X)$ on any object X form a frame (specifically a Heyting algebra) in any join inverse category (indeed, in any join restriction category, see [14]). Recall that in intuitionistic logic (which Heyting algebras model), a proposition p is *decidable* when $p \vee \neg p$ holds. For our purposes, restriction idempotents with this property turn out to be very useful, and so we define them analogously:

Definition 2.8. A restriction idempotent $\overline{e}$ is said to be *decidable* if $\overline{e} \vee \overline{e}^{\perp} = \mathrm{id}$.

Join inverse categories are canonically enriched in domains [5, 28]. This has the pleasant property that any *morphism scheme*, i.e., Scott-continuous function $\mathrm{Hom}(X,Y) \to \mathrm{Hom}(X,Y)$ has a fixed-point:

Theorem 2.9 ([5, 28])**.** *In a join inverse category, any morphism scheme* $\mathrm{Hom}(X,Y) \xrightarrow{\phi} \mathrm{Hom}(X,Y)$ *has a least fixed point* $X \xrightarrow{\mathrm{fix}\,\phi} Y$.

As one might imagine, this turns out to be useful in giving semantics to (systems of mutually) recursive functions. We note in this regard that with the canonical domain enrichment, both partial inversion $f \mapsto f^{\dagger}$ and the action of any join restriction functor on morphisms is continuous.

We conclude this section with a few examples: The category **PInj** of sets and partial injections is a canonical example of an inverse category (even further, by the categorical Wagner-Preston theorem [12], every (locally small) inverse category can be faithfully embedded in **PInj**). For a partial injection $f : A \to B$, define its restriction idempotent $\overline{f} : A \to A$ by $\overline{f}(x) = x$ if f is defined at x, and undefined otherwise. With this definition, every partial injection is a partial isomorphism. Moreover, the partial order on homsets corresponds to the usual partial order on partial functions: that is, for $f, g \in \mathbf{PInj}(A,B)$, $f \leqslant g$ if and only if, for every $x \in A$, f is defined at x implies that g is defined at x in such a way that $f(x) = g(x)$. Observe that *every* restriction idempotent in **PInj** is decidable.

Another example of a join inverse categories is the category **PHom** of topological spaces and partial homeomorphisms with open range and domain of definition. Restrictions and joins are given as in **PInj**, though showing that the join of partial homeomorphisms is again a partial homeomorphism requires use of the so-called *pasting lemma*. In this category, restriction idempotents on a topological space correspond precisely to its open sets. As a consequence, the decidable restriction idempotents in **PHom** correspond to the *clopen* sets, i.e., those simultaneously open and closed.

3 Join inverse rig categories

In this section, we provide the categorical foundations of join inverse rig categories, which are the basis of our model for reversible computing.

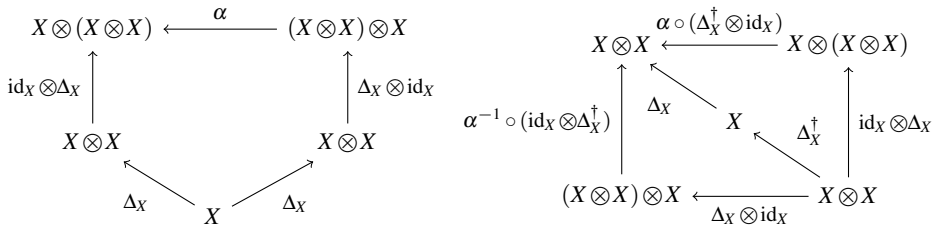
Definition 3.1 ([5, 18])**.** An (join) inverse category **C** with a restriction zero object 0 is said to have a (join-preserving) *disjointness tensor* if it is equipped with a symmetric monoidal (join-preserving) functor $- \oplus -$ (with left unitor $\lambda_{\oplus}$, right unitor $\rho_{\oplus}$, associator $\alpha_{\oplus}$, and commutator $\gamma_{\oplus}$) such that the restriction zero 0 is tensor unit, and the canonical injections given by

$$\coprod_1 = (1_A \oplus 0_{0,B}) \circ \rho_{\oplus}^{-1} : A \to A \oplus B \qquad \text{and} \qquad \coprod_2 = (0_{0,A} \oplus 1_B) \circ \lambda_{\oplus}^{-1} : B \to A \oplus B$$

are jointly epic.

At this point, to define the notion of an inverse product (which first appeared in [18]), we recall the definition of a †-Frobenius semialgebra (see, *e.g.*, [18]), that we later use to describe well-behaved products on inverse categories.

Definition 3.2. In a (symmetric) monoidal †-category, a †-*Frobenius semialgebra* is a pair (X, Δ_X) of an object X and a map $\Delta_X : X \to X \otimes X$ such that the diagrams below commute.



Formally, the leftmost diagram (and its dual) makes (X, Δ_X) a cosemigroup, and $(X, \Delta_X^\dagger)$ a semigroup, while the diagram to the right is called the *Frobenius condition*. One says that a †-Frobenius semialgebra (X, Δ_X) is *special* if $\Delta_X^\dagger \circ \Delta_X = \mathrm{id}_X$, and *commutative* if the monoidal category in which it lives is symmetric and $\gamma_{X,X} \circ \Delta_X = \Delta_X$ (where $\gamma_{X,X}$ is the symmetry of the monoidal category).

Note that this definition is slightly nonstandard, in that †-Frobenius algebras are more often defined in terms of a composition $X \to X \otimes X$ rather than a cocomposition $X \otimes X \to X$. This choice is entirely cosmetic, however, and has no bearing on the algebraic structure defined. Next, we recall the definition of an inverse product.

Definition 3.3. An inverse category $\mathbf{C}$ is said to have an *inverse product* [18] if it is equipped with a symmetric monoidal functor $- \otimes -$ (with left unitor $\lambda_\otimes$, right unitor $\rho_\otimes$, associator $\alpha_\otimes$, and commutator $\gamma_\otimes$) equipped with a natural transformation $X \xrightarrow{\Delta} X \otimes X$ such that the pair (X, Δ_X) is a special and commutative †-Frobenius semialgebra for any object X.

In the context of inverse products, we think of the cosemigroup composition $\Delta : X \to X \otimes X$ as a *duplication* map, and its inverse as a (partial) *equality test* map defined precisely on pairs of equal things. In this way, categories with inverse products are really inverse categories with robust notions of duplication and partial equality testing.

When clear from the context, we omit the subscripts on unitors, associators, and commutators. We are finally ready to define join inverse rig categories.

Definition 3.4. A join inverse rig category is a join inverse category equipped with a join-preserving inverse product $(\otimes, 1)$ and a join-preserving disjointness tensor $(\oplus, 0)$, such that there are natural isomorphisms $X \otimes (Y \oplus Z) \xrightarrow{\delta_L} (X \otimes Y) \oplus (X \otimes Z)$ and $(X \oplus Y) \otimes Z \xrightarrow{\delta_R} (X \otimes Z) \oplus (Y \otimes Z)$ (the *distributors*) natural in X, Y, and Z, and natural isomorphisms $0 \otimes X \xrightarrow{\nu_L} 0$ and $X \otimes 0 \xrightarrow{\nu_R} 0$ (the *annihilators*) natural in X, such that $(\otimes, \oplus, 0, 1)$ form a rig category in the usual sense.

Recall that a category $\mathbf{C}$ is algebraically compact for a class $\mathscr{L}$ of endofunctors on $\mathbf{C}$ if every endofunctor F in the class $\mathscr{L}$ has a unique fixpoint $\mu X.FX$ (or shortly μF) given by the initial F-algebra.

Theorem 3.5. *Any locally small inverse category can be faithfully embedded in a category of which is*

(i) a join inverse rig category,

(ii) algebraically compact for join inverse-preserving functors.

Proof. **PInj** has all of these properties (see also [5]), and any locally small inverse category embeds faithfully into it by the categorical Wagner-Preston theorem [22, Prop. 3.11]. □

In light of this, one can assume without loss of generality that our join inverse rig categories from here on out are algebraically compact for restriction endofunctors, so that every restriction endofunctor F has a unique fixpoint. Note that this provides the first steps towards an interpretation of recursive types in Idealized Theseus [27].

On a final note, our definition of join inverse rig categories is fairly similar to the notion of distributive join inverse category in Giles' terminology [18, Sec. 9.2].

4 A primer in Rfun

The syntax of the programming language Rfun can be summarised by the following grammars:

$$\begin{aligned}
\text{Left Expressions } l &::= x \mid c(l_1,\ldots,l_n) \mid \lfloor l \rfloor \\
\text{Expressions } e &::= l \mid \textbf{let } l_{out} = f\, l_{in} \textbf{ in } e \\
&\quad \mid \textbf{rlet } l_{in} = f\, l_{out} \textbf{ in } e \\
&\quad \mid \textbf{case } l \textbf{ of } \{l_i \to e_i\}_{i=1}^{m} \\
\text{Definitions } d &::= f\, x \triangleq e \\
\text{Programs } q &::= d_1;\ldots;d_n
\end{aligned}$$

The *values* of Rfun are of the form $c(v_1,\ldots,v_n)$ for $n \geqslant 0$, where c is some constructor name, and each v_i is a value. A value of the form $c()$ is called a *symbol*, and is typically written simply as c. Tuples $\langle l_1,\ldots,l_n\rangle$ are also permitted, though these are taken to be syntactic sugar for $\langle\rangle(l_1,\ldots,l_n)$ where $\langle\rangle$ is a distinguished constructor name. Note that, contrary to the original presentation of the language, we make the simplifying assumption that the argument to a function definition is always a variable, and not a (more general) left expression. We recover the original expressivity of Rfun by introducing some syntactic sugar: definitions $f\, l \triangleq e$ stand for terms

$$f\, x \triangleq \textbf{case } x \textbf{ of } l \to e.$$

It is important to recall before going any deeper in the presentation of Rfun that:

- We assume three distinct, denumerable sorts for variables, constructor names, and functions.
- We suppose that programs in the same sequences of definitions have (pairwise) distinct functional identifiers.
- Variables may appear only once in left expressions, and may be used only once in expressions (linearity).
- Domains of substitutions are (pairwise) disjoint.

Now a presentation of Rfun's big step operational semantics is given, with expression judgement $\langle q,\sigma\rangle \vdash e \Downarrow v$ instead of the notation $\sigma \vdash_q e \hookrightarrow v$ from [45]. Concretely, the pair of a program q and a substitution (i.e. partial function) σ constitutes a programming context $\langle q,\sigma\rangle$. Then, the expression judgement $\langle q,\sigma\rangle \vdash e \Downarrow v$ means that the expression e evaluates to the value v in the context $\langle q,\sigma\rangle$. Let us write $\langle q,\sigma\rangle \vdash e \Downarrow$ when there is some value v such that $\langle q,\sigma\rangle \vdash e \Downarrow v$.

As for the pattern matching operations which guide the formation of subtitutions, we replace $v \lhd l \rightsquigarrow \sigma$ [45, Fig. 3, pp. 19] by the more restrictive statement $\langle q,\sigma\rangle \vdash l \Downarrow v$. The relation between those two expressions is given by the following correspondence:

$$\frac{v \lhd l \rightsquigarrow \sigma}{\forall q.\langle q,\sigma\rangle \vdash e \Downarrow v}$$

This leads us to the following operational semantics, which guarantees that computations are reversible (see [45]). Note in particular the distinction between **let** and **rlet**-expressions, which are used to call functions in the *forward* and *backward* directions respectively.

$$\frac{}{\langle q,\{x \mapsto v\}\rangle \vdash x \Downarrow v} \qquad \frac{\lfloor v \rfloor \downarrow = v' \qquad \langle q,\sigma\rangle \vdash l \Downarrow v'}{\langle q,\sigma\rangle \vdash \lfloor l \rfloor \Downarrow v}$$

$$\frac{\langle q,\sigma_1\rangle \vdash l_1 \Downarrow v_1 \quad \cdots \quad \langle q,\sigma_n\rangle \vdash l_n \Downarrow v_n}{\langle q, \uplus_{i=1}^{n} \sigma_i\rangle \vdash c(l_1,\ldots,l_n) \Downarrow c(v_1,\ldots,v_n)} \qquad \frac{f\,x_f \triangleq e_f \in q \quad \langle q,\sigma\rangle \vdash x \Downarrow v' \quad \langle q,\sigma_f\rangle \vdash x_f \Downarrow v' \quad \langle q,\sigma_f\rangle \vdash e_f \Downarrow v}{\langle q,\sigma\rangle \vdash f\,x \Downarrow v}$$

$$\frac{\langle q,\sigma_{\text{in}}\rangle \vdash f\,l_{\text{in}} \Downarrow v_{\text{out}} \qquad \langle q,\sigma_{\text{out}} \uplus \sigma_e\rangle \vdash e \Downarrow v \qquad \langle q,\sigma_{\text{out}}\rangle \vdash l_{\text{out}} \Downarrow v_{\text{out}}}{\langle q,\sigma_{\text{in}} \uplus \sigma_e\rangle \vdash \textbf{let}\, l_{\text{out}} = f\,l_{\text{in}}\ \textbf{in}\ e \Downarrow v}$$

$$\frac{\langle q,\sigma_{\text{out}}\rangle \vdash f\,l_{\text{out}} \Downarrow v_{\text{in}} \qquad \langle q,\sigma_{\text{out}} \uplus \sigma_e\rangle \vdash e \Downarrow v \qquad \langle q,\sigma_{\text{in}}\rangle \vdash l_{\text{in}} \Downarrow v_{\text{in}}}{\langle q,\sigma_{\text{in}} \uplus \sigma_e\rangle \vdash \textbf{rlet}\, l_{\text{in}} = f\,l_{\text{out}}\ \textbf{in}\ e \Downarrow v}$$

$$\frac{\langle q,\sigma_l\rangle \vdash l \Downarrow v' \qquad \langle q,\sigma_{l_j} \uplus \sigma_t\rangle \vdash e_j \Downarrow v \qquad j = \min\{i \mid \forall q.\langle q,\sigma_{l_i}\rangle \vdash l_i \Downarrow v'\} \qquad = \min\{i \mid \forall q.\, l' \in \text{leaves}(e_i) \wedge \langle q,-\rangle \vdash l' \Downarrow v\}}{\langle q,\sigma_l \uplus \sigma_t\rangle \vdash_q \textbf{case}\ l\ \textbf{of}\ \{l_i \to e_i\}_{i=1}^{m} \Downarrow v}$$

5 A categorical model of Rfun

To give a model of Rfun, we start with a join inverse rig category and provide

1. a construction of *values* as an object $T(S)$ over a given alphabet S (thought of as an alphabet of symbols),
2. an interpretation of open left expressions with k free variables as morphisms $T(S)^{\otimes k} \to T(S)$,
3. an interpretation of open expressions with k free variables as morphisms $T(S)^{\otimes k} \to T(S)$ in a *program context*,
4. an interpretation of function definitions as open terms with a single free variable, and
5. an interpretation of programs as a sum of function definitions wrapped in a fixed point (the *program context*).

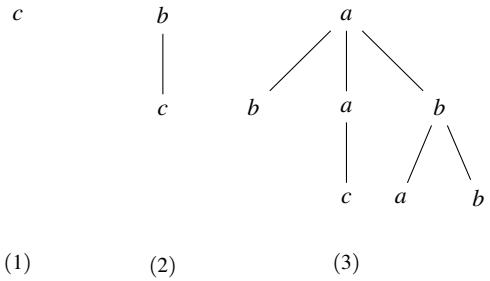
5.1 Values

We start by constructing a denumerable object S of *symbols*, each identified by a unique morphism $1 \to S$ (where 1 is unit of the inverse product). Since the sort of symbols is denumerable by assumption, provided that we can construct such an object, we can uniquely identity a symbol s with a morphism $1 \to S$. Straightforwardly, we define S to be the least fixed point of the join restriction functor $N(X) = 1 \oplus X$, via algebraic compactness: This yields an isomorphism $S \xrightarrow{\text{unfold}_S} S \oplus 1$ (by Lambek's lemma) with inverse $S \oplus 1 \xrightarrow{\text{fold}_S} S$. This allows us to identity the first symbol s_1 with $1 \xrightarrow{\coprod_1} 1 \oplus S \xrightarrow{\text{fold}_S} S$, the second symbol s_2 with $1 \xrightarrow{\coprod_1} 1 \oplus (1 \oplus S) \xrightarrow{\text{id} \oplus \text{fold}_S} 1 \oplus S \xrightarrow{\text{fold}_S} S$, and so on. For this reason, we will simply write $1 \xrightarrow{s} S$ for the morphism corresponding to the symbol s. Note that this has a partial inverse $S \xrightarrow{s^\dagger} 1$, which we think of as a corresponding *assertion* that a given symbol is precisely s.

With this in hand, we can construct the object $T(S)$ of Rfun values, where the functor T is defined as follows:

$$T(X) = \mu K.X \otimes L(K) \qquad L(X) = \mu K.1 \oplus (X \otimes K)$$

Intuitively, L maps an object X to that of lists of X, while T maps an object X to nonempty finite trees with X-values at each node. In **PInj**, a few examples of elements of $T(X)$ (for some set X and $a,b,c \in X$) are shown in the figure below.



The object $T(S)$ allows us to represent terms in Rfun very naturally, namely by their syntax trees. As such, (1) above corresponds to the value c, (2) to $b(c)$, and (3) to the value $a(b,a(c),b(a,b))$. Though this is often how untyped programming languages are modelled, we do not formally require $T(S)$ to be an universal object of the category, as long as it is rich enough for $\mathrm{Hom}(1,T(S))$ to uniquely encode all values.

As in the case for S, we have isomorphisms $L(X) \xrightarrow{\mathrm{unfold}_L} 1 \oplus (X \otimes L(X))$ (with inverse fold_L) and $T(X) \xrightarrow{\mathrm{unfold}_T} X \otimes L(T(X))$ (with inverse fold_T) by Lambek's lemma. The object $L(X)$ can be thought of as lists over X: The morphism $1 \xrightarrow{\amalg_1} 1 \oplus (X \otimes L(X)) \xrightarrow{\mathrm{fold}_L} L(X)$ is thought of as the empty list $[]$, and $X \otimes L(X) \xrightarrow{\amalg_2} 1 \oplus (X \otimes L(X)) \xrightarrow{\mathrm{fold}_L} L(X)$ as the usual *cons* operation. By combining these, we obtain an inductive family of morphisms $X^{\otimes n} \xrightarrow{\mathrm{pack}_n} L(X)$ mapping n-ary tuples into lists, with pack_0 given by $1 \xrightarrow{[]} L(X)$, and pack_{n+1} by $X^{\otimes n+1} \xrightarrow{\cong} X \otimes X^{\otimes n} \xrightarrow{\mathrm{id}\otimes\mathrm{pack}_n} X \otimes L(X) \xrightarrow{\mathrm{cons}} L(X)$. Their partial inverses, $L(X) \xrightarrow{\mathrm{unpack}_n} X^{\otimes n}$, can be thought of as unpacking lists of length precisely n into an n-ary tuple, being undefined on lists of any other length. In particular, one can show that unpack_n and unpack_m are disjoint, so that $\overline{\mathrm{unpack}_n} \circ \overline{\mathrm{unpack}_m} = 0_{L(X),L(X)}$ whenever $n \neq m$.

5.2 Left expressions and patterns

In order to continue with the construction of left expressions, we first need to make one of two assumptions about decidability. Say that an object has *decidable equality* if the restriction idempotent $\overline{\Delta_X^\dagger}$ is decidable.

Assumption 5.1. $T(S)$ has decidable equality.

We justify this terminology by the fact that the cocomposition $X \xrightarrow{\Delta_X} X \otimes X$ is thought of as *duplication*. As such, $\Delta_X^\dagger$ is only ever defined for *results* of duplication, i.e., points which are equal. This assumption allows us to define the *duplication/equality* operator on $T(S)$ as shown in Figure 2. The morphism is described using string diagrams read from bottom to top, with parallel wires representing inverse products. The three morphisms that join to form the definition of $T(S) \xrightarrow{\mathrm{dupeq}} T(S)$ correspond to the three cases in the definition of duplication/equality in Figure 1: The first corresponds to the case where $\lfloor \langle x,y \rangle \rfloor = \langle x \rangle$ when $x = y$, the second to $\lfloor \langle x,y \rangle \rfloor = \langle x,y \rangle$ when $x \neq y$, and the third to $\lfloor \langle x \rangle \rfloor = \langle x,x \rangle$. That this join exists at all follows by the fact that these morphisms are pairwise disjoint (the first and second morphism are both disjoint from the third since unpack_1 and unpack_2 are disjoint; and the two first morphisms are disjoint since $\Delta^\dagger$ and $\overline{\Delta^\dagger}^\perp$ are disjoint). Note the use of the symbol $\langle\rangle$, representing the fact that tuples $\langle l_1,\ldots,l_n \rangle$ in Rfun are simply syntactic sugar for $\langle\rangle(l_1,\ldots,l_n)$ with $\langle\rangle$ a distinguished symbol. Interestingly, from this definition, it is straightforward to show that $\mathrm{dupeq}^\dagger = \mathrm{dupeq}$.

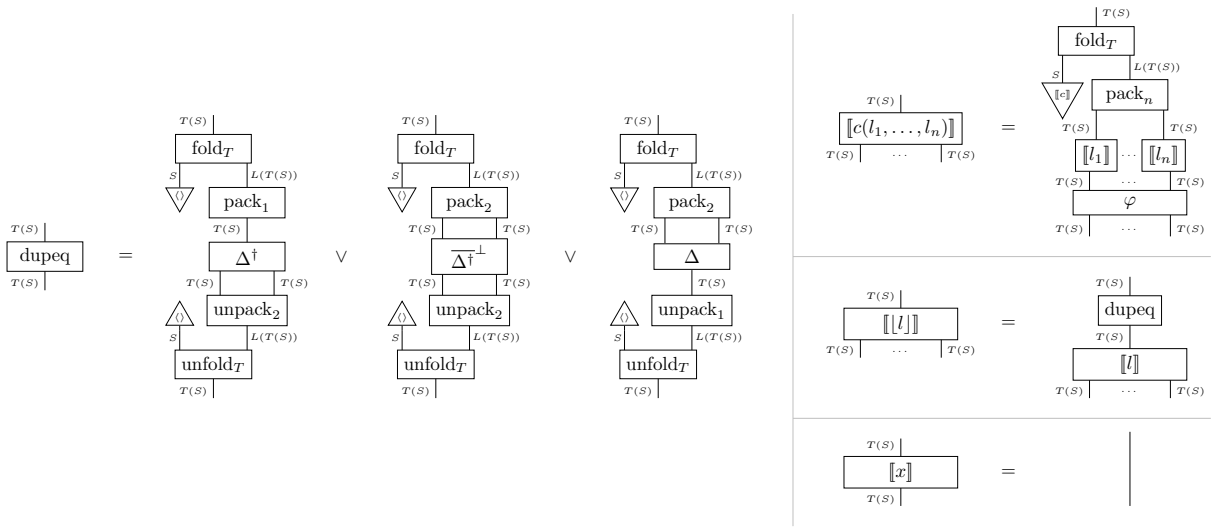


Figure 2: The semantics of left expressions.

To give a semantics to constructed terms and variables, we start with the idea that the free variables of a (left) expression are interpreted as wires of $T(S)$ type going into the denotation, and that the result of a denotation is again of $T(S)$ type. As such, a (left) expression with k free variables is interpreted as a morphism $T(S)^{\otimes k} \to T(S)$.

The semantics of open left expressions is shown in Figure 2. Note the permutation φ of variables wires at the beginning of $[\![c(l_1,\ldots,l_n)]\!]$. This is necessary since free variables may not be used in the order they are given (consider, e.g., the program that takes any tuple $\langle x,y\rangle$ as input and returns $\langle y,x\rangle$ as output), so some reordering must take place first. Since a variable expression x contains precisely one free variable, and nothing is required to further prepare it for use, its semantics is simply the identity. As expected, the semantics of a duplication/equality expression is simply given by handing off the semantics of the inner left expression to the duplication/equality operator previously constructed.

The semantics of left expressions shown in Figure 2 concern their use as *expressions*, but left expressions can also be used as *patterns* in pattern matching case expressions. Fortunately, the interpretation of a left expression as a pattern is simply the partial inverse to its interpretation as an expression. Partiality is key to this insight: Since a left expression describes the formation of a value of a given form, its partial inverse will only ever be *defined* on values of that form. Further, while an open expression consumes (parts of) variable bindings in order to produce a value, patterns do the opposite, consuming a value to produce a variable binding which binds subvalues to free variables. This is reflected in the fact that the types of a left expression as a pattern is then $T(S) \to T(S)^{\otimes k}$, i.e., a partial map which, if it succeeds, splits a value into subvalues accordingly.

5.3 Expressions

Before we're able to proceed with a semantics for expressions, we first need to make our second and final assumption regarding decidability, here involving pattern matching. We say that $T(S)$ has *decidable pattern matching* if for any left expression l the restriction idempotent $\overline{[\![l]\!]}^\dagger$ is decidable.

Assumption 5.2. $T(S)$ has decidable pattern matching.

We now turn to the semantics for expressions which, unlike those for left expressions, need to be parametrised by a *program context*. This is necessary since the expressions include (inverse) function calls, the semantics of which will naturally differ according to the definition of the function being invoked. Similar to [20], we take a program context of n functions to be a morphism $T(S)^{\oplus n} \to T(S)^{\oplus n}$, and think of it as a sum $[\![f_1]\!] \oplus \cdots \oplus [\![f_n]\!]$ of interpretations of constituent functions.

Given such a program context ξ, we will use ξ_i as shorthand for $\coprod_i^\dagger \circ \xi \circ \coprod_i$, and since $(\xi^\dagger)_i = (\xi_i)^\dagger$ we may also use $\xi_i^\dagger$ unambiguously. The semantics of expressions in a program context are shown in Figure 3. As can be seen, function calls to the i^{th} function in the program context are handled by evaluating the input before handing it off to the i^{th} component of the program context. The **let** part of these expressions is handled by first permuting (reflecting the fact that some of the free variables may be used in l_{in} and others in e), and then passing the result to the semantics of e as the contents of a fresh variable. Inverse function invocation (using an **rlet** binder) is handled analogously, though using the *partial inverse* to the program context, rather than the program context itself. Left expressions are handled by passing them on to the earlier definition in Figure 2.

The semantics of **case** expressions require special attention: As with function invocation, since l may only use some of the free variables, we must first select the ones used by l using the permutation φ, and then pass the rest on to the bodies of each branch (which, by linearity, are each required to use the remaining variables). Then, after evaluating l, for each branch $l_i \to e_i$, we need to ensure that only values that did not match any of the previous branches are fed to this branch. This makes sure that branches are tried in the given order, and is why we must compose with $\overline{[\![l_{i-1}]\!]^\dagger}^\perp \circ \cdots \circ \overline{[\![l_1]\!]^\dagger}^\perp$ before trying to match using $[\![l_i]\!]^\dagger$. Should the match succeed, the resulting binding is passed to the semantics of the branch body e_i. In this way, each branch of the **case** expression is constructed as a partial map performing its own pattern matching and evaluation, and the meaning of the entire case expression is simply given by gluing these partial maps together using the join. That the join exists follows by the fact that each branch is made explicitly disjoint from all of the previous ones, by composition with $\overline{[\![l_{i-1}]\!]^\dagger}^\perp \circ \cdots \circ \overline{[\![l_1]\!]^\dagger}^\perp$ (not unlike Gram-Schmidt orthogonalisation).

5.4 Programs

We are finally ready to take on the semantics of function definitions and programs in Rfun. This is comparatively much simpler. Like expressions, the semantics of function definitions is given parametrised by a program context ξ. Since we made the simplifying assumption that any function definition is of the form $f\,x \triangleq e$ for a single variable x, e alone is an expression with exactly one free variable. As such, we simply define the meaning of a function definition to be given by the semantics of its body,

$$[\![f\,x \triangleq e]\!]^\xi = [\![e]\!]^\xi \quad .$$

Finally, a program is simply a list of function definitions, but unlike functions, their semantics should be self-contained and not depend on context. To solve this bootstrapping problem, we use the fixed point operator on continuous morphism schemes $\mathrm{Hom}(T(S)^{\oplus n}, T(S)^{\oplus n}) \to \mathrm{Hom}(T(S)^{\oplus n}, T(S)^{\oplus n})$ given by the canonical enrichment in domains. This, importantly, also allows (systems of mutually) recursive functions to be defined.

$$[\![d_1; \ldots, d_n]\!] = \mathrm{fix}(\xi \mapsto [\![d_1]\!]^\xi \oplus \cdots \oplus [\![d_n]\!]^\xi) \quad .$$

This definition requires us to verify that the function $\xi \mapsto [\![d_1]\!]^\xi \oplus \cdots \oplus [\![d_n]\!]^\xi$ is always continuous, but this follows straightforwardly from the observation that only continuous operations (in particular vertical and horizontal composition) are ever performed on the program context ξ.

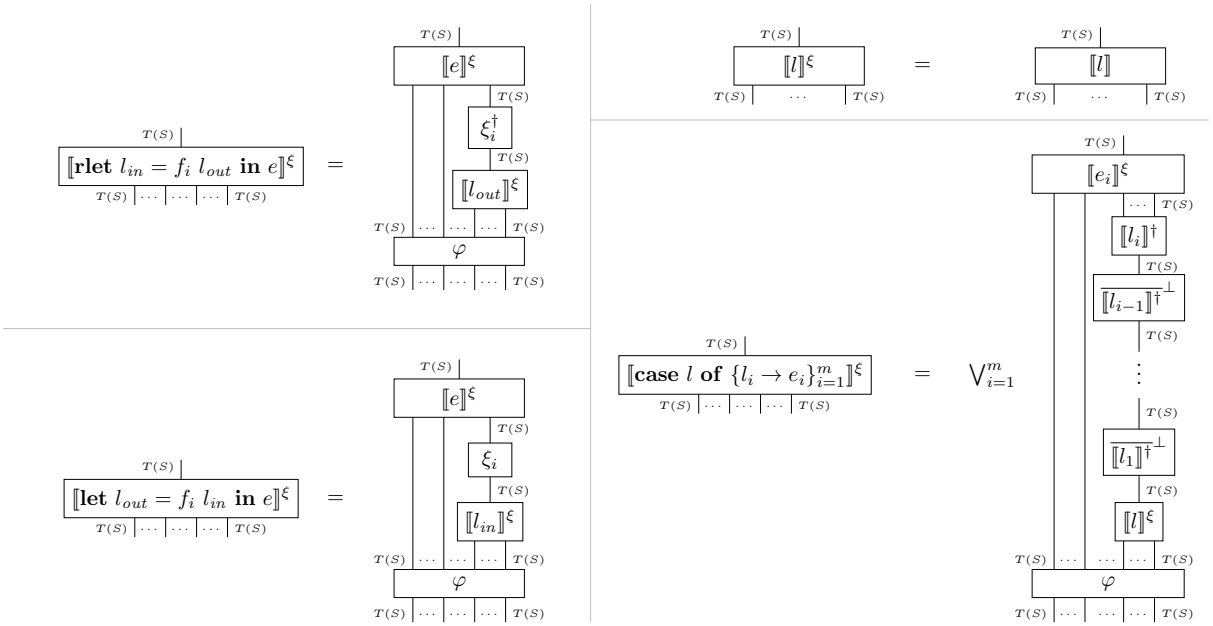


Figure 3: The semantics of expressions in a program context ξ.

5.5 Other reversible languages

Join inverse categorie, as a model for reversible computing, also outline models for other reversible languages than Rfun. We have already noted in Section 3 that Theorem 3.5 leads to an interpretation of recursive types in Theseus [27]. Noting that Theseus is built on the reversible combinator calculus Π^0 [26], and that it can be straightforwardly shown that join inverse rig categories are algebraically compact over join restriction functors and that they are examples of †-traced ω-continuous rig categories (in the sense of Karvonen [30]), it follows that join inverse rig categories are models of Theseus. The fact that join inverse rig categories are †-traced was established in [28]:

Proposition 5.3. *Every join inverse category with a join-preserving disjointness tensor (specifically any join inverse rig category) has a uniform trace operator*

$$\mathrm{Tr}^U_{A,B} : \mathrm{Hom}(A \oplus U, B \oplus U) \to \mathrm{Hom}(A, B)$$

which satisfies $\mathrm{Tr}^U_{A,B}(f)^\dagger = \mathrm{Tr}^U_{B,A}(f^\dagger)$.

Join inverse rig categories also turn out to constitute a model for structured reversible flowcharts, as studied in [19]. The two crucial elements in the interpretation of reversible flowcharts in inverse categories [19] are: (inverse) *extensivity*, which holds for any join inverse category with a join-preserving disjointness tensor (specifically any join inverse rig category) and gives semantics to *reversible control flow*; and the presence of the †-trace constructed previously, which describes *reversible tail recursion*.

6 Concluding remarks

In summary, we have introduced join inverse categories and constructed the categorical semantics of the expressions of the language Rfun. We have also argued that our categorical framework fits neatly in

other reversible languages (Theseus, reversible flowcharts). With arguably weak categorical assumptions, we showcase the strengths of join inverse category theory in the study of the semantics of reversible programming.

Rig categories and groupoids have previously been considered in connection with reversible computing. Notably, the Π family [9, 26] of reversible programming languages, as well as the language CoreFun [25], are essentially term languages for dagger rig categories. Many extensions of Π have since been considered (see, e.g., [10, 11, 23, 29]). The notion of a *distributive* inverse category [18] is also strongly related to our approach.

As future work, we consider the categorical treatment of languages for reversible circuits, for example in the context of abstract embedded circuit-description languages such as EWire [39]. Such a study would be particularly relevant in the context of the development of verification and optimisation tools for Field-Programmable Gate Array (FPGA) circuits, but also quantum circuits.

References

[1] Samson Abramsky & Dominic Horsman (2015): *DEMONIC programming: a computational language for single-particle equilibrium thermodynamics, and its formal semantics*. In Chris Heunen, Peter Selinger & Jamie Vicary, editors: *Proceedings 12th International Workshop on Quantum Physics and Logic*, pp. 1–16, doi:10.4204/EPTCS.195.1.

[2] Thorsten Altenkirch & Jonathan Grattage (2005): *A Functional Quantum Programming Language*. In: *20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, IL, USA, Proceedings*, pp. 249–258, doi:10.1109/LICS.2005.1.

[3] Bogdan Aman, Gabriel Ciobanu, Robert Glück, Robin Kaarsgaard, Jarkko Kari, Martin Kutrib, Ivan Lanese, Claudio Antares Mezzina, Łukasz Mikulski, Rajagopal Nagarajan et al. (2020): *Foundations of reversible computation*. In Irek Ulidowski, Ivan Lanese, Ulrik Pagh Schultz & Carla Ferreira, editors: *Reversible Computation: Extending Horizons of Computing*, Springer, pp. 1–40, doi:10.1016/j.tcs.2015.07.046.

[4] Holger Bock Axelsen & Robert Glück (2011): *What do reversible programs compute?* In Martin Hofmann, editor: *FoSSaCS 2011*, *LNCS* 6604, Springer, pp. 42–56, doi:10.1007/978-3-642-19805-2.

[5] Holger Bock Axelsen & Robin Kaarsgaard (2016): *Join Inverse Categories as Models of Reversible Recursion*. In: *Foundations of Software Science and Computation Structures 2016*, *LNCS* 9634, Springer, pp. 73–90, doi:10.1007/978-3-642-29517-1.

[6] Michael Barr (1992): *Algebraically compact functors*. *Journal of Pure and Applied Algebra* 82(3), pp. 211–231, doi:10.1016/0022-4049(92)90169-G.

[7] 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.

[8] Antoine Bérut, Artak Arakelyan, Artyom Petrosyan, Sergio Ciliberto, Raoul Dillenschneider & Eric Lutz (2012): *Experimental verification of Landauer's principle linking information and thermodynamics*. *Nature* 483(7388), pp. 187–189, doi:10.1143/JPSJ.66.3326.

[9] William J. Bowman, Roshan P. James & Amr Sabry (2011): *Dagger traced symmetric monoidal categories and reversible programming*. Work-in-progress report presented at the 3rd International Workshop on Reversible Computation.

[10] Jacques Carette & Amr Sabry (2016): *Computing with semirings and weak rig groupoids*. In: *Proceedings of the 25th European Symposium on Programming (ESOP 2016)*, Springer, pp. 123–148, doi:10.1007/978-3-662-49498-1.

[11] Chao-Hong Chen & Amr Sabry (2021): *A Computational Interpretation of Compact Closed Categories: Reversible Programming with Negative and Fractional Types*. *Proc. ACM Program. Lang.* 5(POPL), doi:10.1016/j.tcs.2015.07.046.

[12] J. R. B. Cockett & Stephen Lack (2002): *Restriction categories I: Categories of partial maps*. Theoretical Computer Science 270(1–2), pp. 223–259, doi:10.1016/S0304-3975(00)00382-0.

[13] 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.

[14] Robin Cockett & Richard Garner (2014): *Restriction categories as enriched categories*. Theoretical Computer Science 523, pp. 37–55, doi:10.1016/j.tcs.2013.12.018.

[15] Robin Cockett & Stephen Lack (2007): *Restriction categories III: Colimits, partial limits and extensivity*. Mathematical Structures in Computer Science 17(04), pp. 775–817, doi:10.1016/S1571-0661(05)80303-2.

[16] Ioana Cristescu, Jean Krivine & Daniele Varacca (2013): *A compositional semantics for the reversible p-calculus*. In: Logic in Computer Science (LICS), 2013 28th Annual IEEE/ACM Symposium on, IEEE, pp. 388–397, doi:10.1109/LICS.2013.45.

[17] Edward Fredkin & Tommaso Toffoli (1982): *Conservative logic*. International Journal of Theoretical Physics 21(3-4), pp. 219–253, doi:10.1007/BF01857727.

[18] Brett Gordon Giles (2014): *An investigation of some theoretical aspects of reversible computing*. Ph.D. thesis, University of Calgary.

[19] Robert Glück & Robin Kaarsgaard (2018): *A categorical foundation for structured reversible flowchart languages: Soundness and adequacy*. Logical Methods in Computer Science Volume 14, Issue 3, doi:10.23638/LMCS-14(3:16)2018.

[20] Robert Glück, Robin Kaarsgaard & Tetsuo Yokoyama (2019): *Reversible Programs Have Reversible Semantics*. In: Formal Methods. FM 2019 International Workshops, Lecture Notes in Computer Science 12233, pp. 413–427, doi:10.1016/j.tcs.2015.07.046.

[21] Xiuzhan Guo (2012): *Products, Joins, Meets, and Ranges in Restriction Categories*. Ph.D. thesis, University of Calgary.

[22] Chris Heunen (2013): *On the functor ℓ^2*. In: Computation, Logic, Games, and Quantum Foundations. The Many Facets of Samson Abramsky, Springer, pp. 107–121, doi:10.1016/S0022-4049(02)00141-X.

[23] Chris Heunen & Robin Kaarsgaard (2021): *Quantum Information Effects*. ArXiv:2107.12144.

[24] Peter Mark Hines (1998): *The Algebra of Self-Similarity and its Applications*. Ph.D. thesis, University of Wales, Bangor.

[25] Petur Andrias Højgaard Jacobsen, Robin Kaarsgaard & Michael Kirkedal Thomsen (2018): *CoreFun: A Typed Functional Reversible Core Language*. In: International Conference on Reversible Computation, Springer, pp. 304–321, doi:10.1016/j.tcs.2015.07.046.

[26] Roshan P. James & Amr Sabry (2012): *Information Effects*. In: Principles of Programming Languages 2012, Proceedings, ACM, pp. 73–84, doi:10.1145/2103656.2103667.

[27] Roshan P. James & Amr Sabry (2014): *Theseus: A High Level Language for Reversible Computing*. Reversible Computing 2014.

[28] 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.

[29] Robin Kaarsgaard & Niccolò Veltri (2019): *En Garde! Unguarded Iteration for Reversible Computation in the Delay Monad*. In: Proceedings of the 13th International Conference on Mathematics of Program Construction (MPC 2019), Springer, pp. 366–384, doi:10.1007/978-3-030-33636-3.

[30] Martti Karvonen (2019): *The Way of the Dagger*. Ph.D. thesis, University of Edinburgh.

[31] J. Kastl (1979): *Inverse categories*. In Hans-Jürgen Hoehnke, editor: Algebraische Modelle, Kategorien und Gruppoide, Akademie Verlag, Berlin, pp. 51–60.

[32] Martin Kutrib & Andreas Malcher (2010): *Reversible pushdown automata*. In A.-H. Dediu, H. Fernau & C. Martín-Vide, editors: *LATA 2010*, *LNCS* 6031, Springer-Verlag, pp. 368–379, doi:10.1007/978-3-642-13089-2.

[33] Martin Kutrib & Matthias Wendlandt (2015): *Reversible limited automata*. In J. Durand-Lose & B. Nagy, editors: *MCU 2015*, *LNCS* 9288, Springer, pp. 113–128, doi:10.1007/978-3-319-23111-2.

[34] 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.

[35] Mark V Lawson (1998): *Inverse Semigroups: The Theory of Partial Symmetries*. World Scientific, doi:10.1142/3645.

[36] Octavio Malherbe, Philip Scott & Peter Selinger (2013): *Presheaf models of quantum computation: an outline*. In: *Computation, Logic, Games, and Quantum Foundations. The Many Facets of Samson Abramsky*, Springer, pp. 178–194, doi:10.1007/978-3-540-78499-9.

[37] Gordon E Moore (2006): *Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp. 114 ff.* *IEEE Solid-State Circuits Newsletter* 3(20), pp. 33–35, doi:10.1109/N-SSC.2006.4785860.

[38] Mathys Rennela & Sam Staton (2015): *Complete positivity and natural representation of quantum computations*. In: *MFPS XXXI*, 319, Electronic Notes in Theoretical Computer Science, pp. 369–385, doi:10.1016/j.entcs.2015.12.022.

[39] Mathys Rennela & Sam Staton (2018): *Classical control and quantum circuits in enriched category theory*. *Electronic Notes in Theoretical Computer Science* 336, pp. 257–279, doi:10.1016/j.entcs.2018.03.027.

[40] Mathys Rennela, Sam Staton & Robert Furber (2017): *Infinite-Dimensionality in Quantum Foundations: W*-algebras as Presheaves over Matrix Algebras*. In: *QPL'16*, *Electronic Proceedings in Theoretical Computer Science* 236, Open Publishing Association, pp. 161–173, doi:10.4204/EPTCS.236.11.

[41] Markus Schordan, David Jefferson, Peter Barnes, Tomas Oppelstrup & Daniel Quinlan (2015): *Reverse Code Generation for Parallel Discrete Event Simulation*. In Jean Krivine & Jean-Bernard Stefani, editors: *RC 2015*, *LNCS* 9138, Springer, pp. 95–110, doi:10.1007/978-3-319-20860-2.

[42] Ulrik Schultz, Mirko Bordignon & Kasper Stoy (2011): *Robust and reversible execution of self-reconfiguration sequences*. *Robotica* 29(01), pp. 35–57, doi:10.1145/345910.345920.

[43] Ulrik Pagh Schultz, Johan Sund Laursen, Lars-Peter Ellekilde & Holger Bock Axelsen (2015): *Towards a Domain-Specific Language for Reversible Assembly Sequences*. In: *Reversible Computation*, Springer, pp. 111–126, doi:10.1147/rd.456.0807.

[44] Tommaso Toffoli (1980): *Reversible Computing*. In: *Proceedings of the Colloquium on Automata, Languages and Programming*, Springer Verlag, pp. 632–644, doi:10.1007/3-540-10003-2.

[45] Tetsuo Yokoyama, Holger Bock Axelsen & Robert Glück (2012): *Towards a reversible functional language*. In Alexis De Vos & Robert Wille, editors: *RC 2011*, *LNCS* 7165, Springer, pp. 14–29, doi:10.1007/978-3-642-29517-1.

[46] Tetsuo Yokoyama & Robert Glück (2007): *A Reversible Programming Language and Its Invertible Self-interpreter*. In: *PEPM '07, Proceedings*, ACM, pp. 144–153, doi:10.1145/1244381.1244404.