Reversible effects as inverse arrows

Chris Heunen[1]

School of Informatics
University of Edinburgh
United Kingdom

Robin Kaarsgaard[2]

Datalogisk Institut
University of Copenhagen
Denmark

Martti Karvonen[3]

School of Informatics
University of Edinburgh
United Kingdom

Abstract

Reversible computing models settings in which all processes can be reversed. Applications include low-power computing, quantum computing, and robotics. It is unclear how to represent side-effects in this setting, because conventional methods need not respect reversibility. We model reversible effects by adapting Hughes' arrows to dagger arrows and inverse arrows. This captures several fundamental reversible effects, including serialization and mutable store computations. Whereas arrows are monoids in the category of profunctors, dagger arrows are involutive monoids in the category of profunctors, and inverse arrows satisfy certain additional properties. These semantics inform the design of functional reversible programs supporting side-effects.

Keywords: Reversible Effect; Arrow; Inverse Category; Involutive Monoid

1 Introduction

Reversible computing studies settings in which all processes can be reversed: programs can be run backwards as well as forwards. Its history goes back at least as far as 1961, when Landauer formulated his physical principle that logically irreversible manipulation of information costs work. This sparked the interest in developing reversible models of computation as a means to making them more energy efficient. Reversible computing has since also found applications in high-performance computing [28], process calculi [8], probabilistic computing [31], quantum computing [30], and robotics [29].

There are various theoretical models of reversible computations. The most well-known ones are perhaps Bennett's reversible Turing machines [4] and Toffoli's reversible circuit model [32]. There are also various other models of reversible automata [25,23] and combinator calculi [1,18].

We are interested in models of reversibility suited to functional programming languages. Functional languages are interesting in a reversible setting for two reasons. First, they are easier to reason and prove properties

[1] Email: chris.heunen@ed.ac.uk
[2] Email: robin@di.ku.dk
[3] Email: martti.karvonen@ed.ac.uk

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

about, which is a boon if we want to understand the logic behind reversible programming. Second, they are not stateful by definition, which eases reversing programs. It is fair to say that existing reversible functional programming languages [19,33] still lack various desirable constructs familiar from the irreversible setting.

Irreversible functional programming languages like Haskell naturally take semantics in categories. The objects interpret types, and the morphisms interpret functions. Functional languages are by definition not stateful, and their categorical semantics only models pure functions. However, sometimes it is useful to have non-functional side-effects, such as exceptions, input/output, or indeed even state. Irreversible functional languages can handle this elegantly using monads [24] or more generally arrows [16].

A word on terminology. We call a computation $a\colon X \to Y$ *reversible* when it comes with a specified partner computation $a^\dagger\colon Y \to X$ in the opposite direction. This implies nothing about possible side-effects. Saying that a computation is *partially invertible* is stronger, and requires $a \circ a^\dagger \circ a = a$. Saying that it is *invertible* is even stronger, and requires $a \circ a^\dagger$ and $a^\dagger \circ a$ to be identities. We call this partner of a reversible computation its *dagger*. In other words, reversible computing for us concerns dagger arrows on dagger categories, and is modeled using involutions [14]. In an unfortunate clash of terminology, categories of partially invertible maps are called inverse categories [6], and categories of invertible maps are called groupoids [10]. Thus, inverse arrows on inverse categories concern partially invertible maps.

We develop *dagger arrows* and *inverse arrows*, which are useful in two ways:

- We illustrate the reach of these notions by exhibiting many fundamental reversible computational side-effects that are captured (in Section 3), including: pure reversible functions, information effects, reversible state, serialization, vector transformations, dagger Frobenius monads [13,14], recursion [20], and superoperators. Because there is not enough space for much detail, we treat each example informally from the perspective of programming languages, but formally from the perspective of category theory.
- We prove that these notions behave well mathematically (in Section 4): whereas arrows are monoids in a category of profunctors [17], dagger arrows and inverse arrows are involutive monoids.

This paper aims to inform design principles of sound reversible programming languages. The main contribution is to match desirable programming concepts to precise category theoretic constructions. As such, it is written from a theoretical perspective. To make examples more concrete for readers with a more practical background, we adopt the syntax of a typed first-order reversible functional programming language with type classes. We begin with preliminaries on reversible base categories (in Section 2).

2 Dagger categories and inverse categories

This section introduces the categories we work with to model pure computations: dagger categories and inverse categories. Each has a clear notion of reversing morphisms. Regard morphisms in these base categories as pure, ineffectful maps.

Definition 2.1 A *dagger category* is a category equipped with a *dagger*: a contravariant endofunctor $\mathbf{C} \to \mathbf{C}$ satisfying $f^{\dagger\dagger} = f$ for morphisms f and $X^\dagger = X$ for objects X. A morphism f in a dagger category is:

- *positive* if $f = g^\dagger \circ g$ for some morphism g;
- a *partial isometry* if $f = f \circ f^\dagger \circ f$;
- *unitary* if $f \circ f^\dagger = \mathrm{id}$ and $f^\dagger \circ f = \mathrm{id}$.

A dagger functor is a functor between dagger categories that preserves the dagger, *i.e.* a functor F with $F(f^\dagger) = F(f)^\dagger$. A *(symmetric) monoidal dagger category* is a monoidal category equipped with a dagger making the coherence isomorphisms

$$\alpha_{X,Y,Z}\colon X \otimes (Y \otimes Z) \to (X \otimes Y) \otimes Z \qquad \rho_X\colon X \otimes I \to X$$
$$\lambda_X\colon I \otimes X \to X \qquad (\text{and } \sigma_{X,Y}\colon X \otimes Y \to Y \otimes X \text{ in the symmetric case})$$

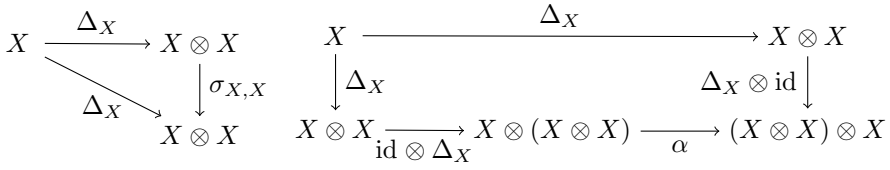
unitary and satisfying $(f \otimes g)^\dagger = f^\dagger \otimes g^\dagger$ for morphisms f and g. We will sometimes suppress coherence isomorphisms for readability.

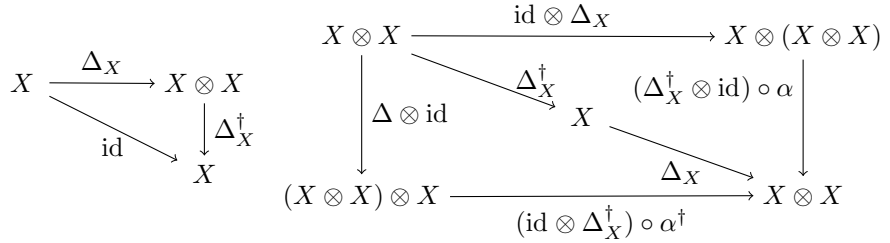
Any groupoid is a dagger category under $f^\dagger = f^{-1}$. Another example of a dagger category is **Rel**, whose objects are sets, and whose morphisms $X \to Y$ are relations $R \subseteq X \times Y$, with composition $S \circ R = \{(x,z) \mid \exists y \in Y\colon (x,y) \in R, (y,z) \in S\}$. The dagger is $R^\dagger = \{(y,x) \mid (x,y) \in R\}$. It is a monoidal dagger category under either Cartesian product or disjoint union.

Definition 2.2 A *(monoidal) inverse category* is a (monoidal) dagger category of partial isometries where positive maps commute: $f \circ f^\dagger \circ f = f$ and $f^\dagger \circ f \circ g^\dagger \circ g = g^\dagger \circ g \circ f^\dagger \circ f$ for all maps $f\colon X \to Y$ and $g\colon X \to Z$.

Every groupoid is an inverse category. Another example of an inverse category is **PInj**, whose objects are sets, and morphisms $X \to Y$ are partial injections: $R \subseteq X \times Y$ such that for each $x \in X$ there exists at most one $y \in Y$ with $(x, y) \in R$, and for each $y \in Y$ there exists at most one $x \in X$ with $(x, y) \in R$. It is a monoidal inverse category under either Cartesian product or disjoint union.

Definition 2.3 A dagger category is said to have *inverse products* [11] if it is a symmetric monoidal dagger category with a natural transformation $\Delta_X \colon X \to X \otimes X$ making the following diagrams commute:





These diagrams express cocommutativity, coassociativity, speciality and the Frobenius law.

Another useful monoidal product, here on inverse categories, is a disjointness tensor, defined in the following way (see [11]):

Definition 2.4 An inverse category is said to have a *disjointness tensor* if it is equipped with a symmetric monoidal tensor product $- \oplus -$ such that its unit 0 is a zero object, and the canonical *quasi-injections*

$$\amalg_1 = X \xrightarrow{\rho_X^{-1}} X \oplus 0 \xrightarrow{X \oplus 0_{0,Y}} X \oplus Y \qquad \amalg_2 = Y \xrightarrow{\lambda_Y^{-1}} 0 \oplus Y \xrightarrow{0_{0,X} \oplus Y} X \oplus Y$$

are jointly epic.

For example, **PInj** has inverse products $\Delta_X \colon X \to X \otimes X$ with $x \mapsto (x, x)$, and a disjointness tensor where $X \oplus Y$ is given by the tagged disjoint union of X and Y (the unit of which is $\emptyset$).

Inverse categories can also be seen as certain instances of restriction categories. Informally, a restriction category models partially defined morphisms, by assigning to each $f \colon A \to B$ a morphism $\bar{f} \colon A \to A$ that is the identity on the domain of definition of f and undefined otherwise. For more details, see [6].

Definition 2.5 A *restriction category* is a category equipped with an operation that assigns to each $f \colon A \to B$ a morphism $\bar{f} \colon A \to A$ such that:

- $f \circ \bar{f} = f$ for every f;
- $\bar{f} \circ \bar{g} = \bar{g} \circ \bar{f}$ whenever $\operatorname{dom} f = \operatorname{dom} g$;
- $\overline{g \circ \bar{f}} = \bar{g} \circ \bar{f}$ whenever $\operatorname{dom} f = \operatorname{dom} g$;
- $\bar{g} \circ f = f \circ \overline{g \circ f}$ whenever $\operatorname{dom} g = \operatorname{cod} f$.

A *restriction functor* is a functor F between restriction categories with $F(\bar{f}) = \overline{F(f)}$. A *monoidal restriction category* is a restriction category with a monoidal structure for which $\otimes \colon \mathbf{C} \times \mathbf{C} \to \mathbf{C}$ is a restriction functor.

A morphism f in a restriction category is a *partial isomorphism* if there is a morphism g such that $g \circ f = \bar{f}$ and $f \circ g = \bar{g}$. Given a restriction category $\mathbf{C}$, define $\operatorname{Inv}(\mathbf{C})$ to be the wide subcategory of $\mathbf{C}$ having all partial isomorphisms of $\mathbf{C}$ as its morphisms.

An example of a monoidal restriction category is **PFn**, whose objects are sets, and whose morphisms $X \to Y$ are partial functions: $R \subseteq X \times Y$ such that for each $x \in X$ there is at most one $y \in Y$ with $(x, y) \in R$. The restriction $\bar{R}$ is given by $\{(x, x) \mid \exists y \in Y \colon (x, y) \in R\}$.

Remark 2.6 Inverse categories could equivalently be defined as either categories in which every morphism f satisfies $f = f \circ g \circ f$ and $g = g \circ f \circ g$ for a unique morphism g, or as restriction categories in which all morphisms are partial isomorphisms [6, Theorem 2.20]. It follows that functors between inverse categories automatically preserve daggers and that $\mathrm{Inv}(\mathbf{C})$ is an inverse category.

It follows, in turn, that an inverse category with inverse products is a monoidal inverse category: because $X \otimes -$ and $- \otimes Y$ are endofunctors on an inverse category, they preserve daggers, so that by bifunctoriality $- \otimes -$ does as well.

3 Arrows as an interface for reversible effects

Arrows are a standard way to encapsulate computational side-effects in a functional (irreversible) programming language [15,16]. This section extends the definition to reversible settings, namely to dagger arrows and inverse arrows. We argue that these notions are "right", by exhibiting a large list of fundamental reversible side-effects that they model. We start by recalling irreversible arrows.

Definition 3.1 An *arrow* on a symmetric monoidal category $\mathbf{C}$ is a functor $A\colon \mathbf{C}^{\mathrm{op}} \times \mathbf{C} \to \mathbf{Set}$ with operations

$$\begin{aligned} \mathrm{arr} &: (X \to Y) \to A\ X\ Y \\ (\ggg) &: A\ X\ Y \to A\ Y\ Z \to A\ X\ Z \\ \mathrm{first}_{X,Y,Z} &: A\ X\ Y \to A\ (X \otimes Z)\ (Y \otimes Z) \end{aligned}$$

that satisfy the following laws:

$$(a \ggg b) \ggg c = a \ggg (b \ggg c) \tag{1}$$
$$\mathrm{arr}(g \circ f) = \mathrm{arr}\, f \ggg \mathrm{arr}\, g \tag{2}$$
$$\mathrm{arr}\,\mathrm{id} \ggg a = a = a \ggg \mathrm{arr}\,\mathrm{id} \tag{3}$$
$$\mathrm{first}_{X,Y,I}\, a \ggg \mathrm{arr}\,\rho_Y = \mathrm{arr}\,\rho_X \ggg a \tag{4}$$
$$\mathrm{first}_{X,Y,Z}\, a \ggg \mathrm{arr}(\mathrm{id}_Y \otimes f) = \mathrm{arr}(\mathrm{id}_X \otimes f) \ggg \mathrm{first}_{X,Y,Z}\, a \tag{5}$$
$$(\mathrm{first}_{X,Y,Z\otimes V}\, a) \ggg \mathrm{arr}\,\alpha_{Y,Z,V} = \mathrm{arr}\,\alpha_{X,Z,V} \ggg \mathrm{first}(\mathrm{first}\, a) \tag{6}$$
$$\mathrm{first}(\mathrm{arr}\, f) = \mathrm{arr}(f \otimes \mathrm{id}) \tag{7}$$
$$\mathrm{first}(a \ggg b) = (\mathrm{first}\, a) \ggg (\mathrm{first}\, b) \tag{8}$$

where we use the functional programming convention to write $A\ X\ Y$ for $A(X,Y)$ and $X \to Y$ for $\hom(X,Y)$ The *multiplicative fragment* consists of above data except first, satisfying all laws except those mentioning first; we call this a *weak arrow*.

Define $\mathrm{second}(a)$ by $\mathrm{arr}(\sigma) \ggg \mathrm{first}(a) \ggg \mathrm{arr}(\sigma)$, using the symmetry, so analogs of (4)–(8) are satisfied. Arrows makes sense for (nonsymmetric) monoidal categories if we add this operation and these laws.

Definition 3.2 A *dagger arrow* is an arrow on a monoidal dagger category with an additional operation $\mathrm{inv} : A\ X\ Y \to A\ Y\ X$ satisfying the following laws:

$$\mathrm{inv}(\mathrm{inv}\, a) = a \tag{9}$$
$$\mathrm{inv}\, a \ggg \mathrm{inv}\, b = \mathrm{inv}(b \ggg a) \tag{10}$$
$$\mathrm{arr}(f^\dagger) = \mathrm{inv}(\mathrm{arr}\, f) \tag{11}$$
$$\mathrm{inv}(\mathrm{first}\, a) = \mathrm{first}(\mathrm{inv}\, a) \tag{12}$$

A *inverse arrow* is a dagger arrow on a monoidal inverse category such that:

$$(a \ggg \mathrm{inv}\, a) \ggg a = a \tag{13}$$
$$(a \ggg \mathrm{inv}\, a) \ggg (b \ggg \mathrm{inv}\, b) = (b \ggg \mathrm{inv}\, b) \ggg (a \ggg \mathrm{inv}\, a) \tag{14}$$

The *multiplicative fragment* consists of above data except first, satisfying all laws except those mentioning first.

Remark 3.3 There is some redundancy in the definition of an inverse arrow: (13) and (14) imply (11) and (12); and (11) implies $\mathrm{inv}(\mathrm{arr}\,\mathrm{id}) = \mathrm{arr}\,\mathrm{id}$.

Like the arrow laws (1)–(8), in a programming language with inverse arrows, the burden is on the programmer to guarantee (9)–(14) for their implementation. If that is done, the language guarantees arrow inversion.

Remark 3.4 Now follows a long list of examples of inverse arrows, described in a typed first-order reversible functional pseudocode with type classes, inspired by Theseus [19,18], the revised version of Rfun (briefly described in [21]), and Haskell. Type classes are a form of interface polymorphism: A type class is defined by a class specification containing the signatures of functions that a given type must implement in order to be a member of that type class (often, type class membership also informally requires the programmer to ensure that certain equations are required of their implementations). For example, the *Functor* type class (in Haskell) is given by the class specification

class *Functor f* **where**
  $fmap : (a \to b) \to f\ a \to f\ b$

with the additional informal requirements that $fmap\ id = id$ and $fmap\ (g \circ f) = (fmap\ g) \circ (fmap\ f)$ must be satisfied for all instances. For example, lists in Haskell satisfy these equations when defining *fmap* as the usual *map* function, *i.e.*:

instance *Functor List* **where**
  $fmap\ \ \ \ \ \ \ \ : (a \to b) \to List\ a \to List\ b$
  $fmap\ f\ []\ \ \ \ \ = []$
  $fmap\ f\ (x{::}xs) = (f\ x){::}(fmap\ f\ xs)$

While higher-order reversible functional programming is fraught, aspects of this can be mimicked by means of parametrized functions. A parametrized function is a function that takes parts of its input statically (*i.e.*, no later than at compile time), in turn lifting the first-order requirement on these inputs. To separate static and dynamic inputs from one another, two distinct function types are used: $a \to b$ denotes that a must be given statically, and $a \leftrightarrow b$ (where a and b are first-order types) denotes that a is passed dynamically. As the notation suggests, functions of type $a \leftrightarrow b$ are reversible. For example, a parametrized variant of the reversible map function can be defined as a function $map : (a \leftrightarrow b) \to ([a] \leftrightarrow [b])$. Thus, *map* itself is *not* a reversible function, but given statically any reversible function $f : a \leftrightarrow b$, the parametrized $map\ f : ([a] \leftrightarrow [b])$ is.

Given this distinction between static and dynamic inputs, the signature of arr becomes $(X \leftrightarrow Y) \to A\ X\ Y$. Definition 3.1 uses the original signature, because this distinction is not present in the irreversible case. Fortunately, the semantics of arrows remain the same whether or not this distinction is made.

Example 3.5 *(Pure functions)* A trivial example of an arrow is the identity arrow $\hom(-,+)$ which adds no computational side-effects at all. This arrow is not as boring as it may look at first. If the identity arrow is an inverse arrow, then the programming language in question is both *invertible* and *closed under program inversion*: any program p has a semantic inverse $[\![p]\!]^\dagger$ (satisfying certain equations), and the semantic inverse coincides with the semantics $[\![\mathrm{inv}(p)]\!]$ of another program $\mathrm{inv}(p)$. As such, inv must be a sound and complete *program inverter* (see also [22]) on pure functions; not a trivial matter at all.

Example 3.6 *(Information effects)* James and Sabry's *information effects* [18] explicitly expose creation and erasure of information as effects. This type-and-effect system captures irreversible computation inside a pure reversible setting.

We describe the languages from [18] categorically, as there is no space for syntactic details. Start with the free dagger category $(\mathbf{C}, \times, 1)$ with finite products (and hence coproducts), where products distribute over coproducts by a unitary map. Objects interpret types of the reversible language Π of bijections, and morphisms interpret terms. The category $\mathbf{C}$ is a monoidal inverse category.

The category $\mathbf{C}$ carries an arrow, where $A(X,Y)$ is the disjoint union of $\hom(X \times H, Y \times G)$ where G and H range over all objects, and morphisms $X \times H \to Y \times G$ and $X \times H' \to Y \times G'$ are identified when they are equal up to coherence isomorphisms. This is an inverse arrow, where $\mathrm{inv}(a)$ is simply $a^\dagger$. It supports the following additional operations:

$$erase = [\,\pi_H : X \times H \to H\,]_\simeq \in A(X,1),$$
$$create_X = [\,\pi_H^\dagger : H \to X \times H\,]_\simeq \in A(1,X).$$

James and Sabry show how a simply-typed first order functional irreversible language translates into a reversible one by using this inverse arrow to build implicit communication with a global heap H and garbage dump G.

Example 3.7 *(Reversible state)* Perhaps the prototypical example of an effect is computation with a mutable store of type S. In the irreversible case, such computations are performed using the state monad $\mathit{State}\ S\ X\ =\ S \multimap (X \otimes S)$, where $S \multimap -$ is the right adjoint to $- \otimes S$, and can be thought of as a function type. Morphisms in the corresponding Kleisli category are morphisms of the form $X \to S \multimap (Y \otimes S)$ in the ambient monoidal closed category. In this formulation, the current state is fetched by $\mathit{get} : \mathit{State}\ S\ S$ defined as $\mathit{get}\ s\ =\ (s, s)$, while the state is (destructively) updated by $\mathit{put} : S \to \mathit{State}\ S\ 1$ defined as $\mathit{put}\ x\ s\ =\ ((), x)$.

Such arrows can not be used as-is in inverse categories, however, as canonical examples (such as **PInj**) fail to be monoidal closed. To get around this, note that it follows from monoidal closure that $\hom(X, S \multimap (Y \otimes S)) \simeq \hom(X \otimes S, Y \otimes S)$, so that $\hom(- \otimes S, - \otimes S)$ is an equivalent arrow that does not depend on closure. With this is mind, we define the *reversible state arrow* with a store of type S:

$$\textbf{type}\ \mathit{RState}\ S\ X\ Y\ =\ X \otimes S \leftrightarrow Y \otimes S$$

$$\begin{aligned}
&\textbf{instance}\ \mathit{Arrow}\ (\mathit{RState}\ S)\ \textbf{where}\\
&\quad \mathit{arr}\ f\ (x, s) &&= (f\ x,\ s)\\
&\quad (a \ggg b)\ (x, s) &&= b\ (a\ (x, s))\\
&\quad \mathit{first}\ a\ ((x, z), s) &&= \textbf{let}\ (x', s') = a\ (x, s)\ \textbf{in}\ ((x', z), s')\\
&\textbf{instance}\ \mathit{InverseArrow}\ (\mathit{RState}\ S)\ \textbf{where}\\
&\quad \mathit{inv}\ a\ (y, s) &&= a^{\dagger}\ (y, s)
\end{aligned}$$

This satisfies the inverse arrow laws. To access the state, we use reversible duplication of values (categorically, this requires the monoidal product to have a natural diagonal $\Delta_X : X \to X \otimes X$, as inverse products do). Syntactically, this corresponds to the following arrow:

$$\begin{aligned}
&\mathit{get} &&: \mathit{RState}\ S\ X\ (X \otimes S)\\
&\mathit{get}\ (x, s) &&= ((x, s), s)
\end{aligned}$$

The inverse to this arrow is $\mathit{assert} : \mathit{RState}\ S\ (X \otimes S)\ X$, which asserts that the current state is precisely what is given in its second input component; if this fails, the result is undefined. For changing the state, while we cannot destructively update it reversibly, we *can* reversibly update it by a given reversible function with signature $S \leftrightarrow S$. This gives:

$$\begin{aligned}
&\mathit{update} &&: (S \leftrightarrow S) \to \mathit{RState}\ S\ X\ X\\
&\mathit{update}\ f\ (x, s) &&= (x,\ f\ s)
\end{aligned}$$

This is analogous to how variable assignment works in the reversible programming language Janus [34]: Since destructive updating is not permitted, state is updated by means of built-in reversible update operators, *e.g.*, updating a variable by adding a constant or the contents of another variable to it, etc.

Example 3.8 *(Computation in context)* Related to computation with a mutable store is computation with an immutable one; that is, computation within a larger context that remains invariant across execution. In an irreversible setting, this job is typically handled by the *reader monad* (with context of type C), defined as $\mathit{Reader}\ C\ X\ =\ C \Rightarrow X$. This approach is fundamentally irreversible, however, as the context is "forgotten" whenever a value is computed by supplying it with a context. Even further, it relies on the reversibly problematic notion of monoidal closure.

A reversible version of this idea is one that remembers the context, giving us the reversible Reader arrow:

$$\textbf{type}\ \mathit{Reader}\ C\ X\ Y\ =\ X \otimes C \leftrightarrow Y \otimes C$$

This is precisely the same as the state arrow – indeed, the instance declarations for *arr*, $(\ggg)$, *first*, and *inv* are the same – save for the fact that we additionally require *all Reader* arrows r to satisfy $c = c'$ whenever $r\ (x, c)\ =\ (y, c')$. We notice that $\mathit{arr}\ f$ satisfies this property for all f, whereas $(\ggg)$, *first*, and *inv* all preserve it. This resembles the "slice" construction on inverse categories with inverse products; see [11, Sec. 4.4].

As such, while we can provide access to the context via a function defined exactly as *get* for the reversible state arrow, we cannot provide an update function without (potentially) breaking this property – as intended. In practice, the property that the context is invariant across execution can be aided by appropriate interface hiding, *i.e.* exposing the *Reader* type and appropriate instance declarations and helpers (such as *get* and *assert*) but leaving the *constructor* for *Reader* arrows hidden.

Example 3.9 *(Rewriter)* A particularly useful special case of the reversible state arrow is when the store S forms a group. While group multiplication if seen as a function $G \otimes G \leftrightarrow G$ is invertible only in degenerate cases, we can use parametrization to fix the first argument of the multiplication, giving it a much more reasonable signature of $G \to (G \leftrightarrow G)$. In this way, groups can be expressed as instances of the type class

$$\begin{aligned}&\textbf{class } \mathit{Group}\ G \textbf{ where}\\ &\quad \mathit{gunit} : G\\ &\quad \mathit{gmul} : G \to (G \leftrightarrow G)\\ &\quad \mathit{ginv}\ : G \leftrightarrow G\end{aligned}$$

subject to the usual group axioms. This gives us an arrow of the form

$$\textbf{type } \mathit{Rewriter}\ G\ X\ Y\ =\ X \otimes G \leftrightarrow Y \otimes G$$

with instance declarations identical to that of $\mathit{RState}\ G$, save that we require G to be an instance of the *Group* type class. With this, adding or removing elements from state of type G can then be performed by

$$\begin{aligned}&\mathit{rewrite} \quad : \ G \to \mathit{Rewriter}\ G\ X\ X\\ &\mathit{rewrite}\ a\ (x,\ b) = (x,\ \mathit{gmul}\ a\ b)\end{aligned}$$

which "rewrites" the state by the value a of type G. Note that while the name of this arrow was chosen to be evocative of the *Writer* monad known from irreversible functional programming, as it may be used for similar practical purposes, its construction is substantially different (*i.e.*, irreversible *Writer* arrows are maps of the form $X \to Y \times M$ where M is a monoid).

Example 3.10 *(Vector transformation)* Vector transformations, that is, functions on lists that preserve the length of the list, form another example of inverse arrows. The *Vector* arrow is defined as follows:

$$\textbf{type } \mathit{Vector}\ X\ Y\ =\ [X] \leftrightarrow [Y]$$

$$\begin{aligned}&\textbf{instance } \mathit{Arrow}\ (\mathit{Vector}) \textbf{ where}\\ &\quad \mathit{arr}\ f\ xs \quad\ = \mathit{map}\ f\ xs\\ &\quad (a \ggg b)\ xs = b\ (a\ xs)\\ &\quad \mathit{first}\ a\ ps \quad\ = \textbf{let } (xs, zs)\ =\ \mathit{zip}^\dagger\ ps \textbf{ in } \mathit{zip}\ (a\ xs,\ zs)\\ &\textbf{instance } \mathit{InverseArrow}\ (\mathit{Vector}) \textbf{ where}\\ &\quad \mathit{inv}\ a\ ys = a^\dagger\ ys\end{aligned}$$

The definition of *first* relies on the usual *map* and *zip* functions, which are defined as follows:

$$\begin{aligned}&\mathit{map} \quad\ : \ (a \leftrightarrow b) \to ([a] \leftrightarrow [b])\\ &\mathit{map}\ f\ [] \quad\ = []\\ &\mathit{map}\ f\ (x{::}xs) = (f\ x){::}(\mathit{map}\ f\ xs)\end{aligned} \qquad \begin{aligned}&\mathit{zip} \quad\ : \ ([a], [b]) \leftrightarrow [(a, b)]\\ &\mathit{zip}\ ([], []) \quad\ = []\\ &\mathit{zip}\ (x{::}xs, y{::}ys) = (x, y){::}(\mathit{zip}\ (xs, ys))\end{aligned}$$

Notice that preservation of length is required for *first* to work: if the arrow a does not preserve the length of xs, then $\mathit{zip}\ (a\ xs,\ zs)$ is undefined. However, since *arr* lifts a pure function f to a *map* (which preserves length), and $(\ggg)$ and *inv* are given by the usual composition and inversion, the interface maintains this property.

Example 3.11 *(Reversible error handling)* An inverse *weak arrow* comes from reversible computation with a possibility for failure. The weak *Error* arrow is defined using disjointness tensors as follows:

$$\textbf{type } \mathit{Error}\ E\ X\ Y\ =\ X \oplus E \leftrightarrow Y \oplus E$$

$$\begin{aligned}&\textbf{instance } \mathit{WeakArrow}\ (\mathit{Error}\ E) \textbf{ where}\\ &\quad \mathit{arr}\ f\ (\mathit{InL}\ x) = \mathit{InL}\ (f\ x)\\ &\quad \mathit{arr}\ f\ (\mathit{InR}\ e) = \mathit{InR}\ e\\ &\quad (a \ggg b)\ x \quad\ = b\ (a\ x)\\ &\textbf{instance } \mathit{InverseWeakArrow}\ (\mathit{Error}\ E) \textbf{ where}\\ &\quad \mathit{inv}\ a\ y = a^\dagger\ y\end{aligned}$$

In this definition, we think of the type E as the type of *errors* that could occur during computation. As such, a pure function f lifts to a weak arrow which always succeeds with value $f(x)$ when given a nonerroneous input of x, and always propagates errors that may have occured previously.

Raising an error reversibly requires more work than in the irreversible case, as the effectful program that produces an error must be able to *recover* from it in the converse direction. In this way, a reversible *raise* requires two pieces of data: a function of type $X \leftrightarrow E$ that transforms problematic inputs into appropriate errors; and a choice function of type $E \leftrightarrow E \oplus E$ that decides if the error came from this site, injecting it to the left if it did, and to the right if it did not. The latter choice function is critical, as in the converse direction it decides whether the error should be handled immediately or later. Thus we define *raise* as follows:

$$\begin{aligned} &\mathit{raise} \quad : \ (X \leftrightarrow E) \to (E \leftrightarrow E \oplus E) \to \mathit{Error}\ E\ X\ Y \\ &\mathit{raise}\ f\ p\ x = \mathit{InR}\ (p^\dagger\ (\mathit{arr}\ f\ x))) \end{aligned}$$

The converse of *raise* is *handle*, an (unconditional) error handler that maps matching errors back to succesful output values. Since unconditional error handling is seldom required, this can be combined with control flow (see Example 3.15) to perform conditional error handling, *i.e.* to only handle errors if they occur.

Example 3.12 *(Serialization)* When restricting our attention, as we do here, to only first-order reversible functional programming languages, another example of inverse arrows arises in the form of *serializers*. A serializer is a function that transforms an internal data representation into one more suitable for storage, or for transmission to other running processes. To transform serialized data back into an internal representation, a suitable deserializer is used.

When restricting ourselves to the first-order case, it seems reasonable to assume that all types are serializable, as we thus avoid the problematic case of how to serialize data of function type. As such, assuming that all types X admit a function $\mathit{serialize} : X \leftrightarrow \mathit{Serialized}\ X$ (where $\mathit{Serialized}\ X$ is the type of serializations of data of type X), we define the *Serializer* arrow as follows:

$$\begin{aligned} &\textbf{type}\ \mathit{Serializer}\ X\ Y\ = X \leftrightarrow \mathit{Serialized}\ Y \\ \\ &\textbf{instance}\ \mathit{Arrow}\ (\mathit{Serializer})\ \textbf{where} \\ &\quad \mathit{arr}\ f\ x \quad\quad = \mathit{serialize}\ (f\ x) \\ &\quad (a \ggg b)\ x\ = b\ (\mathit{serialize}^\dagger\ (a\ x)) \\ &\quad \mathit{first}\ a\ (x, z) = \mathit{serialize}\ (\mathit{serialize}^\dagger\ (a\ x),\ z) \\ &\textbf{instance}\ \mathit{InverseArrow}\ (\mathit{Serializer})\ \textbf{where} \\ &\quad \mathit{inv}\ a\ y = \mathit{serialize}\ (a^\dagger\ (\mathit{serialize}\ y)) \end{aligned}$$

Notice how $\mathit{serialize}^\dagger : \mathit{Serialized}\ X \leftrightarrow X$ takes the role of a (partial) deserializer, able to recover the internal representation from serialized data as produced by the serializer. A deserializer of the form $\mathit{serialize}^\dagger$ will often only be partially defined, since many serialization methods allow many different serialized representations of the same data (for example, many textual serialization formats are whitespace insensitive). In spite of this shortcoming, partial deserializers produced by inverting serializers are sufficient for the above definition to satisfy the inverse arrow laws.

Example 3.13 *(Dagger Frobenius monads)* Monads are also often used to capture computational side-effects. Arrows are more general. If T is a strong monad, then $A = \hom(-, T(+))$ is an arrow: arr is given by the unit, $\ggg$ is given by Kleisli composition, and first is given by the strength maps. What happens when the base category is a dagger or inverse category modelling reversible pure functions?

A monad T on a dagger category is a *dagger Frobenius monad* when it satisfies $T(f^\dagger) = T(f)^\dagger$ and $T(\mu_X) \circ \mu^\dagger_{T(X)} = \mu_{T(X)} \circ T(\mu^\dagger_X)$. The Kleisli category of such a monad is again a dagger category [14, Lemma 6.1], giving rise to an operation inv satisfying (9)–(10). A dagger Frobenius monad is strong when the strength maps are unitary. In this case (11)–(12) also follow. If the underlying category is an inverse category, then $\mu \circ \mu^\dagger \circ \mu = \mu$, whence $\mu \circ \mu^\dagger = \mathrm{id}$, and (13)–(14) follow. Thus, if T is a strong dagger Frobenius monad on a dagger/inverse category, then A is a dagger/inverse arrow. The Frobenius monad $T(X) = X \otimes \mathbb{C}^2$ on the category of Hilbert spaces captures measurement in quantum computation [13], giving a good example of capturing an irreversible effect in a reversible setting. For more examples see [14].

Example 3.14 *(Restriction monads)* There is a notion in between the dagger and inverse arrows of the previous example. A *(strong) restriction monad* is a (strong) monad on a (monoidal) restriction category whose underlying endofunctor is a restriction functor. The Kleisli-category of a restriction monad T has a

natural restriction structure: just define the restriction of $f\colon X \to T(Y)$ to be $\eta_X \circ \bar{f}$. The functors between the base category and the Kleisli category then become restriction functors. If T is a strong restriction monad on a monoidal restriction category $\mathbf{C}$, then $\mathrm{Inv}(\mathbf{C})$ has an inverse arrow $(X, Y) \mapsto (\mathrm{Inv}(\mathcal{K}\ell(T)))(X, Y)$.

Example 3.15 *(Control flow)* While only trivial inverse categories have coproducts [11], less structure suffices for reversible control structures. When the domain and codomain of an inverse arrow both have disjointness tensors (see Definition 2.4), it can often be used to implement *ArrowChoice*. For a simple example, the pure arrow on an inverse category with disjointness tensors implements $\mathit{left} : A\ X\ Y \to A\ (X \oplus Z)\ (Y \oplus Z)$ as

$$\mathit{left}\ f\ (x,\ z)\ =\ (f\ x,\ z)$$

The laws of *ArrowChoice* [15] simply reduce to $- \oplus -$ being a bifunctor with natural quasi-injections. More generally, the laws amount to preservation of the disjointness tensor. For the reversible state arrow (Example 3.7), this hinges on $\otimes$ distributing over $\oplus$.

The splitting combinator $(+\!\!+\!\!+)$ is unproblematic for reversiblity, but the fan-in combinator $(|||)$ cannot be defined reversibly, as it explicitly deletes information about which branch was chosen. Reversible conditionals thus require two predicates: one determining the branch to take, and one asserted to join the branches after execution. The branch-joining predicate must be chosen carefully to ensure that it is always true after the *then*-branch, and false after the *else*-branch. This is a standard way of handling branch joining reversibly [34,33,12].

Example 3.16 *(Superoperators)* Quantum information theory has to deal with environments. The basic category **FHilb** is that of finite-dimensional Hilbert spaces and linear maps. But because a system may be entangled with its environment, the only morphisms that preserve states are the so-called superoperators, or *completely positive* maps [30,7]: they are not just positive, but stay positive when tensored with an arbitrary ancillary object. In a sense, information about the system may be stored in the environment without breaking the (reversible) laws of nature. This leads to the so-called CPM construction. It is infamously known *not* to be a monad. But it *is* a dagger arrow on **FHilb**, where $A\ X\ Y$ is the set of completely positive maps $X^* \otimes X \to Y^* \otimes Y$, $\mathrm{arr}\, f = f_* \otimes f$, $a \ggg b = b \circ a$, $\mathrm{first}_{X,Y,Z}\, a = a \otimes \mathrm{id}_{Z^* \otimes Z}$, and $\mathrm{inv}\, a = a^\dagger$.

Aside from these, other examples do fit the interface of inverse arrows, though they are less syntactically interesting as they must essentially be "built in" to a particular programming language. These include reversible IO, which functions very similarly to irreversible IO, and reversible recursion, which could be used to give a type-level separation between terminating and potentially non-terminating functions, by only allowing fixed points of parametrized functions between arrows rather than between (pure) functions.

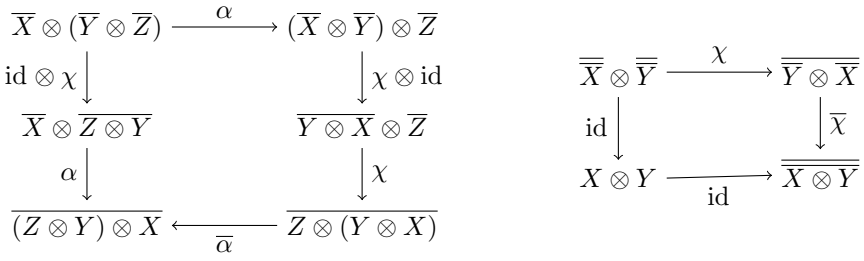
4 Inverse arrows, categorically

This section explicates the categorical structure of inverse arrows. Arrows on $\mathbf{C}$ can be modelled categorically as monoids in the functor category $[\mathbf{C}^{\mathrm{op}} \times \mathbf{C}, \mathbf{Set}]$ [17]. They also correspond to certain identity-on-objects functors $J\colon \mathbf{C} \to \mathbf{D}$. The category $\mathbf{D}$ for an arrow A is built by $\mathbf{D}(X, Y) = A\ X\ Y$, and arr provides the functor J. We will only consider the multiplicative fragment. The operation first can be incorporated in a standard way using strength [17,3], and poses no added difficulty in the reversible setting.

Clearly, dagger arrows correspond to $\mathbf{D}$ being a dagger category and J a dagger functor, whereas inverse arrows correspond to both $\mathbf{C}$ and $\mathbf{D}$ being inverse categories and J a (dagger) functor. This section takes the first point of view: which monoids correspond to dagger arrows and inverse arrows? In the dagger case, the answer is quite simple: the dagger makes $[\mathbf{C}^{\mathrm{op}} \times \mathbf{C}, \mathbf{Set}]$ into an involutive monoidal category, and then dagger arrows correspond to involutive monoids. Inverse arrows furthermore require certain diagrams to commute.

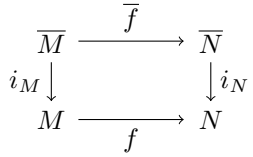
Definition 4.1 An *involutive monoidal category* is a monoidal category $\mathbf{C}$ equipped with an *involution*: a functor $\overline{(\ )}\colon \mathbf{C} \to \mathbf{C}$ satisfying $\overline{\overline{f}} = f$ for all morphisms f, together with a natural isomorphism $\chi_{X,Y}\colon \overline{X} \otimes \overline{Y} \to$

$\overline{Y \otimes X}$ that makes the following diagrams commute [4]:



Just like monoidal categories are the natural setting for monoids, involutive monoidal categories are the natural setting for involutive monoids. Any involutive monoidal category has a canonical isomorphism $\phi\colon I \to \overline{I}$ [9, Lemma 2.3]. Moreover, any monoid M with multiplication m and unit u induces a monoid on $\overline{M}$ with multiplication $\overline{m} \circ \chi_{M,M}$ and unit $\overline{u} \circ \phi$. This monoid structure on $\overline{M}$ allows us to define involutive monoids.

Definition 4.2 An *involutive monoid* is a monoid (M, m, u) together with a monoid homomorphism $i\colon \overline{M} \to M$ satisfying $i \circ \overline{i} = \mathrm{id}$. A *morphism* of involutive monoids is a monoid homomorphism $f\colon M \to N$ making the following diagram commute:



Our next result lifts the dagger on $\mathbf{C}$ to an involution on the category $[\mathbf{C}^{\mathrm{op}} \times \mathbf{C}, \mathbf{Set}]$ of profunctors. First we recall the monoidal structure on that category. It categorifies the dagger monoidal category **Rel** of relations of Section 2 [5].

Definition 4.3 If $\mathbf{C}$ is small, then $[\mathbf{C}^{\mathrm{op}} \times \mathbf{C}, \mathbf{Set}]$ has a monoidal structure

$$F \otimes G(X, Z) = \int^{Y} F(X, Y) \times G(Y, Z);$$

concretely, $F \otimes G(X,Z) = \coprod_{Y \in \mathbf{C}} F(X,Y) \times G(Y,Z) / \approx$, where $\approx$ is the equivalence relation generated by $(y, F(f, \mathrm{id})(x)) \approx (G(\mathrm{id}, f)(y), x)$, and the action on morphisms is given by $F \otimes G(f,g) := [y,x]_{\approx} \mapsto [F(f,\mathrm{id})x, G(\mathrm{id},g)y]$. The unit of the tensor product is $\hom_{\mathbf{C}}$.

Proposition 4.4 *If $\mathbf{C}$ is a dagger category, then $[\mathbf{C}^{\mathrm{op}} \times \mathbf{C}, \mathbf{Set}]$ is an involutive monoidal category when one defines the involution on objects F by $\overline{F}(X,Y) = F(Y,X)$, $\overline{F}(f,g) = F(g^\dagger, f^\dagger)$ and on morphisms $\tau\colon F \to G$ by $\overline{\tau}_{X,Y} = \tau_{Y,X}$.*

Proof. First observe that $\overline{(\ )}$ is well-defined: For any natural transformation of profunctors τ, $\overline{\tau}$ is natural, and $\tau \mapsto \overline{\tau}$ is functorial. Define $\chi_{F,G}$ by the following composite of natural isomorphisms:

$$\begin{aligned}
\overline{F} \otimes \overline{G}(X,Z) &\cong \textstyle\int^Y \overline{F}(X,Y) \times \overline{G}(Y,Z) \text{ by definition of } \otimes \\
&= \textstyle\int^Y F(Y,X) \times G(Z,Y) \text{ by definition of } \overline{(\ )} \\
&\cong \textstyle\int^Y G(Z,Y) \times F(Y,X) \text{ by symmetry of } \times \\
&\cong G \otimes F(Z,X) \text{ by definition of } \otimes \\
&= \overline{G \otimes F}(X,Z) \text{ by definition of } \overline{(\ )}
\end{aligned}$$

Checking that χ make the relevant diagrams commute is routine. □

[4] There is a more general definition allowing a natural isomorphism $\overline{\overline{X}} \to X$ (see [9] for details), but we only need the strict case.

Theorem 4.5 *If* $\mathbf{C}$ *is a dagger category, the multiplicative fragments of dagger arrows on* $\mathbf{C}$ *correspond exactly to involutive monoids in* $[\mathbf{C}^{\mathrm{op}} \times \mathbf{C}, \mathbf{Set}]$.

Proof. It suffices to show that the dagger on an arrow corresponds to an involution on the corresponding monoid F. But this is easy: an involution on F corresponds to giving, for each X, Y a map $F(X,Y) \to F(Y,X)$ subject to some axioms. That this involution is a monoid homomorphism amounts to it being a contravariant identity-on-objects-functor, and the other axiom amounts to it being involutive. □

Remark 4.6 If the operation first is modeled categorically as (internal) strength, axiom (12) for dagger arrows can be phrased in $[\mathbf{C}^{\mathrm{op}} \times \mathbf{C}, \mathbf{Set}]$ as follows: for each object Z of $\mathbf{C}$, and each dagger arrow M, the profunctor $M_Z = M((-) \otimes Z, (+) \otimes Z)$ is also a dagger arrow, and $\mathrm{first}_{-,+,Z}$ is a natural transformation $M \Rightarrow M_Z$. The arrow laws (7) and (8) imply that it is a monoid homomorphism, and the new axiom just states that it is in fact a homomorphism of involutive monoids. For inverse arrows this law is not needed, as any functor between inverse categories is automatically a dagger functor and thus every monoid homomorphism between monoids corresponding to inverse arrows preserves the involution.

Next we set out to characterize which involutive monoids correspond to inverse arrows. Given an involutive monoid M, the obvious approach would be to just state that the map $M \to M$ defined by $a \mapsto a \circ a^\dagger \circ a$ is the identity. However, there is a catch: for an arbitrary involutive monoid, the map $a \mapsto a \circ a^\dagger \circ a$ is not natural transformation and therefore not a morphism in $[\mathbf{C}^{\mathrm{op}} \times \mathbf{C}, \mathbf{Set}]$. To circumvent this, we first require some conditions guaranteeing naturality. These conditions concern endomorphisms, and to discuss them we introduce an auxiliary operation on $[\mathbf{C}^{\mathrm{op}} \times \mathbf{C}, \mathbf{Set}]$.

Definition 4.7 Let $\mathbf{C}$ be a dagger category. Given a profunctor $M \colon \mathbf{C}^{\mathrm{op}} \times \mathbf{C} \to \mathbf{Set}$, define $LM \colon \mathbf{C}^{\mathrm{op}} \times \mathbf{C} \to \mathbf{Set}$ by

$$\begin{aligned} LM(X,Y) &= M(X,X), \\ LM(f,g) &= f^\dagger \circ (-) \circ f. \end{aligned}$$

If M is an involutive monoid in $[\mathbf{C}^{\mathrm{op}} \times \mathbf{C}, \mathbf{Set}]$, define a subprofunctor of LM:

$$L^+M(X,Y) = \{a^\dagger \circ a \in M(X,X) \mid a \in M(X,Z) \text{ for some } Z\}.$$

Remark 4.8 The construction L is a functor $[\mathbf{C}^{\mathrm{op}} \times \mathbf{C}, \mathbf{Set}] \to [\mathbf{C}^{\mathrm{op}} \times \mathbf{C}, \mathbf{Set}]$. There is an analogous construction $RM(X,Y) = M(Y,Y)$ and R^+M, and furthermore $RM = \overline{LM}$. For any monoid M in $[\mathbf{C}^{\mathrm{op}} \times \mathbf{C}, \mathbf{Set}]$, LM is a right M-module (and RM a left M-module). Compare Example 3.16.

For the rest of this section, assume the base category $\mathbf{C}$ to be an inverse category. This lets us multiply positive arrows by positive pure morphisms. If M is an involutive monoid in $[\mathbf{C}^{\mathrm{op}} \times \mathbf{C}, \mathbf{Set}]$, then the map $LM \times L^+(\hom_{\mathbf{C}}) \to LM$ defined by $(a, g^\dagger \circ g) \mapsto a \circ g^\dagger \circ g$ is natural:

$$\begin{aligned} & LM \times L^+(\hom)(f, \mathrm{id}_Y)(a, g^\dagger \circ g) \\ &= (f^\dagger \circ a \circ f, f^\dagger \circ g^\dagger \circ g \circ f) \\ &\mapsto f^\dagger \circ a \circ f \circ f^\dagger \circ g^\dagger \circ g \circ f \\ &= f^\dagger \circ a \circ g^\dagger \circ g \circ f \circ f^\dagger \circ f && \text{because } \mathbf{C} \text{ is an inverse category} \\ &= f^\dagger \circ a \circ g^\dagger \circ g \circ f && \text{because } \mathbf{C} \text{ is an inverse category} \\ &= LM(f, \mathrm{id}_Y)(a \circ g^\dagger \circ g) \end{aligned}$$

Similarly there is a map $L^+(\hom) \times LM \to LM$ defined by $(g^\dagger \circ g, a) \mapsto g^\dagger \circ g \circ a$. Now the category corresponding to M satisfies $a^\dagger \circ a \circ g^\dagger \circ g = g^\dagger \circ g \circ a^\dagger \circ a$ for all a and pure g if and only if the following diagram commutes:

$$\begin{array}{ccccc} L^+M \times L^+(\hom) & & \longrightarrow & & LM \times L^+(\hom) \\ \sigma \downarrow & & & & \downarrow \\ L^+(\hom) \times L^+M & \longrightarrow & L^+(\hom) \times LM & \longrightarrow & LM \end{array} \tag{15}$$

If this is satisfied for an involutive monoid M in $[\mathbf{C}^{\mathrm{op}} \times \mathbf{C}, \mathbf{Set}]$, then positive arrows multiply. In other words, the map $L^+M \times L^+M \to LM$ defined by $(a^\dagger \circ a, b^\dagger \circ b) \mapsto a^\dagger \circ a \circ b^\dagger \circ b$ is natural:

$$\begin{aligned}
&D_M(f,g)(a,a^\dagger,a) \\
&= (g \circ a \circ f, f^\dagger \circ a^\dagger \circ g^\dagger, g \circ a \circ f) \\
&\mapsto g \circ a \circ f \circ f^\dagger \circ a^\dagger \circ g^\dagger \circ g \circ a \circ f \\
&= g \circ a \circ a^\dagger \circ g^\dagger \circ g \circ a \circ f \circ f^\dagger \circ f && \text{by (15)} \\
&= g \circ a \circ a^\dagger \circ g^\dagger \circ g \circ a \circ f && \text{because } \mathbf{C} \text{ is an inverse category} \\
&= g \circ g^\dagger \circ g \circ a \circ a^\dagger \circ a \circ f && \text{by (15)} \\
&= g \circ a \circ a^\dagger \circ a \circ f && \text{because } \mathbf{C} \text{ is an inverse category} \\
&= M(f,g)(a \circ a^\dagger \circ a)
\end{aligned}$$

This multiplication is commutative iff the following diagram commutes:

$$\begin{array}{ccc} L^+M \times L^+M & \xrightarrow{\sigma} & L^+M \times L^+M \\ & \searrow & \downarrow \\ & & LM \end{array} \tag{16}$$

Finally, let $D_M \hookrightarrow M \times \overline{M} \times M$ be the diagonal $D_M(X,Y) = \{(a, a^\dagger, a) \mid a \in M(X,Y\}$. If M satisfies (15), then the map $D_M \to M$ defined by $(a, a^\dagger, a) \mapsto a \circ a^\dagger \circ a$ is natural:

$$\begin{aligned}
&D_M(f,g)(a,a^\dagger,a) \\
&= (g \circ a \circ f, f^\dagger \circ a^\dagger \circ g^\dagger, g \circ a \circ f) \\
&\mapsto g \circ a \circ f \circ f^\dagger \circ a^\dagger \circ g^\dagger \circ g \circ a \circ f \\
&= g \circ a \circ a^\dagger \circ g^\dagger \circ g \circ a \circ f \circ f^\dagger \circ f && \text{by (15)} \\
&= g \circ a \circ a^\dagger \circ g^\dagger \circ g \circ a \circ f && \text{because } \mathbf{C} \text{ is an inverse category} \\
&= g \circ g^\dagger \circ g \circ a \circ a^\dagger \circ a \circ f && \text{by (15)} \\
&= g \circ a \circ a^\dagger \circ a \circ f && \text{because } \mathbf{C} \text{ is an inverse category} \\
&= M(f,g)(a \circ a^\dagger \circ a)
\end{aligned}$$

Thus M satisfies $a \circ a^\dagger \circ a = a$ if and only if the following diagram commutes:

$$\begin{array}{ccc} M & \longrightarrow & D_M \\ & \searrow^{\mathrm{id}} & \downarrow \\ & & M \end{array} \tag{17}$$

Hence we have established the following theorem.

Theorem 4.9 *Let* $\mathbf{C}$ *be an inverse category. Then the multiplicative fragments of inverse arrows on* $\mathbf{C}$ *correspond exactly to involutive monoids in* $[\mathbf{C}^{\mathrm{op}} \times \mathbf{C}, \mathbf{Set}]$ *making the diagrams* (15)–(17) *commute.* □

5 Applications and related work

As we have seen, inverse arrows capture a variety of fundamental reversible effects. An immediate application of our results would be to retrofit existing typed reversible functional programming languages (*e.g.*, Theseus [19]) with inverse arrows to accommodate reversible effects while maintaining a type-level separation between pure and effectful programs. Another approach could be to design entirely new such programming languages, taking

inverse arrows as the fundamental representation of reversible effects. While the Haskell approach to arrows uses typeclasses [15], these are not a priori necessary to reap the benefits of inverse arrows. For example, special syntax for defining inverse arrows could also be used, either explicitly, or implicitly by means of an effect system that uses inverse arrows "under the hood".

To aid programming with ordinary arrows, a handy notation due to Paterson [26,27] may be used. If the underlying monoidal dagger category has natural coassociative diagonals, for example when it has inverse products, a similar `do`-notation can be implemented for inverse and dagger arrows.

A pleasant consequence of the semantics of inverse arrows is that inverse arrows are safe: as long as the inverse arrow laws are satisfied, fundamental properties guaranteed by reversible functional programming languages (such as invertibility and closure under program inversion) are preserved. In this way, inverse arrows provide reversible effects as a conservative extension to pure reversible functional programming.

A similar approach to invertibility using arrows is given by bidirectional arrows [2]. However, while the goal of inverse arrows is to add effects to already invertible languages, bidirectional arrows arise as a means to add invertibility to an otherwise uninvertible language. As such, bidirectional arrows have different concerns than inverse arrows, and notably do not guarantee invertibility in the general case.

Acknowledgements
This work was supported by COST Action IC1405, the Oskar Huttunen Foundation, and EPSRC Fellowship EP/L002388/1. We thank Robert Furber and Robert Glück for discussions.

References

[1] S. Abramsky. A structural approach to reversible computation. *Theoretical Computer Science*, 347(3):441–464, 2005.

[2] A. Alimarine, S. Smetsers, A. van Weelden, M. van Eekelen, and R. Plasmeijer. There and back again: Arrows for invertible programming. In *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 86–97, 2005.

[3] K. Asada. Arrows are strong monads. In *Proceedings of the third ACM SIGPLAN workshop on Mathematically Structured Functional Programming*, pages 33–42. ACM, 2010.

[4] C. H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(6):525–532, 1973.

[5] F. Borceux. *Handbook of categorical algebra*. Cambridge University Press, 1994.

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

[7] B. Coecke and C. Heunen. Pictures of complete positivity in arbitrary dimension. *Information and Computation*, 250:50–58, 2016.

[8] I. Cristescu, J. Krivine, and D. Varacca. A compositional semantics for the reversible π-calculus. In *Logic in Computer Science*, pages 388–397. IEEE Computer Society, 2013.

[9] J. M. Egger. On involutive monoidal categories. *Theory and Applications of Categories*, 25(14):368–393, 2011.

[10] M. J. Gabbay and P. H. Kropholler. Imaginary groups: lazy monoids and reversible computation. *Mathematical Structures in Computer Science*, 23(5):1002–10031, 2013.

[11] B. G. Giles. *An investigation of some theoretical aspects of reversible computing*. PhD thesis, University of Calgary, 2014.

[12] R. Glück and R. Kaarsgaard. A categorical foundations for structured reversible flowchart languages. In *Mathematical Foundations of Program Semantics XXXIII, Proceedings*, 2017. to appear.

[13] C. Heunen and M. Karvonen. Reversible monadic computing. In *Mathematical Foundations of Programming Semantics (MFPS)*, volume 319 of *Electronic Notes in Theoretical Computer Science*, pages 217–237, 2015.

[14] C. Heunen and M. Karvonen. Monads on dagger categories. *Theory and Applications of Categories*, 31(35):1016–1043, 2016.

[15] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, 2000.

[16] J. Hughes. *Advanced Functional Programming*, volume 3622 of *Lecture Notes in Computer Science*, chapter Programming with Arrows, pages 73–129. Springer, 2005.

[17] B. Jacobs, C. Heunen, and I. Hasuo. Categorical semantics for arrows. *Journal of Functional Programming*, 19(3-4):403–438, 2009.

[18] R. P. James and A. Sabry. Information effects. In *Principles of Programming Languages*, pages 73–84. ACM, 2012.

[19] R. P. James and A. Sabry. Theseus: A high level language for reversible computing, 2014. Work-in-progress report at RC 2014, available at https://www.cs.indiana.edu/~sabry/papers/theseus.pdf.

[20] R. Kaarsgaard, H. B. Axelsen, and R. Glück. Join inverse categories and reversible recursion. *Journal of Logical and Algebraic Methods in Programming*, 87, 2017.

[21] R. Kaarsgaard and M. K. Thomsen. Rfun revisited. In M. Waldén, editor, *Proceedings of the 29th Nordic Workshop on Programming Theory*, volume 27 of *TUCS Lecture Notes*, pages 65–67. Turku Centre for Computer Science, 2017.

[22] M. Kawabe and R. Glück. The program inverter lrinv and its structure. In M. V. Hermenegildo and D. Cabeza, editors, *Practical Aspects of Declarative Languages*, volume 3350 of *Lecture Notes in Computer Science*, pages 219–234. Springer, 2005.

[23] M. Kutrib and M. Wendlandt. Reversible limited automata. In *Machines, Computations and Universality*, volume 9288 of *Lecture Notes in Computer Science*, pages 113–128, 2015.

[24] E. Moggi. Computational lambda-calculus and monads. *Logic in Computer Science*, 1989.

[25] K. Morita. Two-way reversible multihead automata. *Fundamenta Informaticae*, 110(1–4):241–254, 2011.

[26] R. Paterson. A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240, 2001.

[27] R. Paterson. Arrows and computation. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, pages 201–222. Palgrave, 2003.

[28] M. Schordan, D. Jefferson, P. Barnes, T. Oppelstrup, and D. Quinlan. Reverse code generation for parallel discrete event simulation. In *Reversible Computing*, volume 9138 of *Lecture Notes in Computer Science*, pages 95–110. Springer, 2015.

[29] U. P. Schultz, M. Bordignon, and K. Støy. Robust and reversible execution of self-reconfiguration sequences. *Robotica*, 29(1):35–37, 2011.

[30] P. Selinger. Dagger compact closed categories and completely positive maps. In *Quantum Programming Languages*, volume 170 of *Electronic Notes in Theoretical Computer Science*, pages 139–163. Elsevier, 2007.

[31] B. Stoddart and R. Lynas. A virtual machine for supporting reversible probabilistic guarded command languages. In *Reversible Computing*, volume 253 of *Electronic Notes in Theoretical Computer Science*, pages 33–56. Elsevier, 2010.

[32] T. Toffoli. Reversible computing. In *International Colloquium on Automata, Languages and Programming*, pages 632–644, 1980.

[33] T. Yokoyama, H. B. Axelsen, and R. Glück. Towards a reversible functional language. In A. De Vos and R. Wille, editors, *Reversible Computation 2011*, volume 7165 of *LNCS*, pages 14–29. Springer, 2012.

[34] T. Yokoyama and R. Glück. A reversible programming language and its invertible self-interpreter. In *Partial Evaluation and Semantics-Based Program Manipulation. Proceedings*, pages 144–153. ACM Press, 2007.