UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

PhD thesis

Robin Kaarsgaard

The Logic of Reversible Computing

Theory and Practice

Academic advisors: Robert Glück (principal), Holger Bock Axelsen, Andrzej Filinski

Submitted: December 22, 2017

The Logic of Reversible Computing

Theory and Practice

Robin Kaarsgaard
DIKU, Department of Computer Science,
University of Copenhagen

December 22, 2017

This thesis has been submitted to the PhD School of the Faculty of Science at the University of Copenhagen.

To Sophie

Abstract

Reversible computing is the study of models of computation that exhibit both forward and backward determinism. While reversible computing initially gained interest through its potential to reduce the energy consumption of computing machinery, via a result from physics now known as *Landauer's principle*, a number of other applications in computer science have since been proposed, from syntax descriptions to model-based testing, debugging, and even robotics.

In spite of its numerous current (and potential future) applications, the established foundations for computation and programming, such as Turing machines, λ-calculi, and various categorical models, are largely ill equipped to handle reversible computing, as these often tacitly rely on irreversible operations to function. To set reversible computing on a foundation as solid as the one for conventional computing requires both a significant adaptation of existing techniques and the development of new ones.

In this thesis, we investigate reversible computing from a perspective of logic, broadly construed. To complement the operational point of view from which reversible computing is often studied, we offer a denotational account of reversibility in computation based on recent work in category theory. We propose two new techniques, founded in formal logic, for reasoning about reversible logic circuits. Further, we account for the behaviour of *fixed points* in certain proposed categorical models of reversible computing, and connect these results to the behaviour of recursive functions and data types in established reversible programming languages. In an application and extension of some of these results, we propose a uniform categorical foundation for a large class of reversible imperative programming languages known as *structured reversible flowchart languages*. We investigate the role of reversible effects in reversible functional programming, and show that a wide palette of these may be modelled as *arrows* (in the sense of Hughes) satisfying certain additional equations. Finally, we propose a brief vision for the future of the reversible functional programming language Rfun.

Dansk resumé

Reversibel beregning er studiet af beregningsmodeller der er både fremad- og baguddeterministiske. På trods af at reversibel beregning oprindeligt fik interesse igennem dets potentialle til at reducere beregningsmaskiners energiforbrug, via et resultat fra fysikken der nu kendes som *Landauer's princip*, har det senere fundet et antal andre anvendelser i datalogien, fra syntaksbeskrivelser til model-baseret testning, fejlfinding i programmel, og endda robotteknologi.

På trods af disse talrige nuværende (og potentielle fremtidige) anvendelser er mange grundlæggende modeller for beregning og programmerinbg, såsom Turingmaskiner, λ-kalkyler og visse kategoriske modeller, i høj grad dårligt rustede til at håndtere reversibel beregning, da disse ofte implicit afhænger af irreversible operationer for at virke. At kunne placere reversibel beregning på et lige så solidt grundlag som det for konventionel beregning kræver både betydelige tilpasninger af eksisterende teknikker, og udviklingen af nye.

I denne afhandling undersøger vi reversibel beregning fra et (i bred forstand) logisk perspektiv. Som supplement til det operationelle udgangspunkt der ofte benyttes til at studere reversible beregning giver vi en denotationel redegørelse af reversibilitet i beregning, baseret på nylige resultater fra kategoriteori. Vi fremsætter to nye teknikker, grundlagt i teknikker fra formel logik, til at ræsonnere om reversible logiske kredsløb. Endvidere redegør vi for *fikspunkters* opførsel i visse foreslåede kategoriske modeller for reversibel beregning, og forbinder denne redegørelse til rekursive funktioner og datatypers opførsel i etablerede reversible programmeringssprog. Ved brug og udvidelse af nogle af disse resultater foreslår vi et ensartet kategorisk grundlag for en betydelig mængde af imperative reversible programmeringssprog, de såkaldte *strukturerede reversible flowchartsprog*. Vi undersøger reversible effekters rolle i reversibel funktionsprogrammering, og viser, at en bred palet af disse kan modelleres som *pile* (i Hughes' fortolkning) der opfylder visse yderligere ligheder. Endelig fremsætter vi en kortfattet vision for fremtiden for det reversible funktionsprogrammeringssprog Rfun.

Contents

Preface

This manuscript constitutes the author's PhD thesis, an endeavour that could not have been completed without the support and encouragement from a tremendous number of people. First of all, I would like to thank my academic advisors Holger Bock Axelsen, Andrzej Filinski, and Robert Glück for their encouragement and sage advice, always ready to discuss a particularly puzzling topic (of which there are many) or to point me in the right direction. I would also like to thank my (current and previous) office mates and other fellow PhD students, with whom I've shared many discussions (academic and otherwise), thoughts, and beers. In no particular order: Ulrik Terp Rasmussen, Bjørn "Nillerbjørn" Bugge Grathwohl, Vivek Shah, Oleksandr Shturmov, Danil Annenkov, Abraham Wolk, Martin Dybdal. I'd also like to thank Fritz Henglein for many interesting discussions, for agreeing to chair my thesis committee, and for entrusting me early on with teaching duties in one of my favourite courses here at DIKU, *Logic in Computer Science*. I would also like to thank Hanna van Lee, who quickly became my partner in crime in teaching this course.

I would also like to thank those who have helped me broaden my horizons, and who have showed in interest in my work: Apart from those already mentioned, I'd like to thank Torben Mogensen, Marcos vaz Salles and the rest of the APL group here at DIKU; and Rasmus Møgelberg (who first introduced me to category theory) and Marco Paviotti for great discussions and great coffee. I also owe thanks to Bart Jacobs and the rest of the Nijmegen Quantum Logic Group at Radboud University for hosting me in the spring of 2016: Aleks Kissinger, Fabio Zanasi, Mathys Rennela, Bas Westerbaan, Bram Westerbaan, Kenta Chō, and Sander Uijlen. I would also like to thank Dexter Kozen, Michael Johnson, Robin Cockett, and Tarmo Uustalu for interesting discussions, and for showing interest in my work; and, the latter two, for agreeing to serve on my thesis committee. A special thanks goes to my (non-advisor) collaborators: Chris Heunen, Martti Karvonen, Mathias Soeken, Michael Kirkedal Thomsen. I am also grateful for the support offered by COST Action IC1405: *Reversible computing – extending horizons of computing.*

Finally, I would like to thank my friends and family for their support during the past three years; and Sophie, for everything.

Robin Kaarsgaard

Full name: Robin Kaarsgaard Jensen

"Sometimes I wish I knew how to go crazy. I forget how."
"It's a lost art," Hank said. "Maybe there's an instruction manual on it."
Philip K. Dick, *A Scanner Darkly*

1 Introduction

A famous parable from the *Rigveda*, familiar to many category theorists due to the truly awesome work of Peter Johnstone [74], is that of the blind men examining an elephant: One man, holding the trunk, says it is like a snake; another man, touching one of the elephant's legs, says it is like a pillar; a third man, holding the tail, says it is like a rope, etc. Similar sound but incomplete statements can be made of reversible computing: It is about computations that can be *undone*; it is about computers as *physical machines*; it is about computation free of *information effects* [72]; and so on.

In this chapter we will give an introduction to central concepts in reversible computing as they will be used later in the thesis. To set the stage for some of these developments, we will also present a new denotational view of reversible computing that highlights *compositionality* as a central principle of reversibility, clarify some concepts that are often left underspecified, and connect this to existing views on reversible computing presented in the literature during the past semicentury.

1.1 Reversibility and invertibility: A semantic perspective

Reversibility in computational processes is often presented as an operational property based on forward and backward determinism. We illustrate these concepts by means of Figure 1.1. A program is *locally forward deterministic* (or simply *deterministic*) if every computation state in the program has a unique *next* computation state. For example, programs written in programming languages such as Java or Haskell are all forward deterministic. More exotic is the notion of *local backward determinism*, requiring that every computation state has a unique *previous* state. This is typically *not* guaranteed by common programming languages: For example, in an imperative language, a program performing a destructive assignment such as

```
x := 2
```

is not backward deterministic, since there is generally no way of recovering the state of x before it was assigned to have the value 2. Likewise, in functional programming languages, branching expressions (such as conditionals or *case*-expressions) are common sources of backward nondeterminism, as two or more branches may produce identical outputs on distinct inputs.

The locality of forward and backward determinism is a crucial part of what is sometimes humorously called the *Copenhagen interpretation of reversibility*, and is often presented by the slogan that *reversibility is a local phenomenon*. Concretely, it requires not only that the entire program behaves in a way that is forward and backward deterministic (*i.e.*, inputs uniquely determine outputs and vice versa), but that any computation step along the way (no matter if it is a simple instruction or a complicated loop structure) behaves in this way as well. Put in another way, reversibility is a property of the program rather than the function it computes (*i.e.*, it is an intensional property). Without the emphasis on locality, it is not clear that one would be able to separate reversible programs from those merely computing injective functions (which is a distinction we wish to make). We summarise this view on reversibility in the following definition.

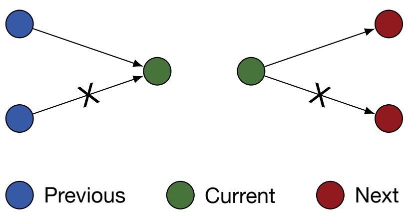


Figure 1.1: Forward and backward determinism.

Definition 1. A program is reversible if it is locally forward deterministic and locally backward deterministic.

A consequence of this view is that reversible programs can be uniquely assigned both forward and backward semantics, if they can be assigned semantics at all. As an example, consider a an operational semantics for an imperative language, *i.e.*, with a judgment form of $\sigma \vdash p \downarrow \sigma'$ taken to mean that evaluating any command p in the state σ yields the state σ'. Forward determinism states that for all states σ and commands p, there is at most one state σ' such that $\sigma \vdash p \downarrow \sigma'$. Symmetrically, backward determinism states that for all states σ' and commands p, there is at most one state σ such that $\sigma \vdash p \downarrow \sigma'$.

Since reversibility is an intensional property, it requires very careful program construction, and just as careful argumentation, to establish for a program. Even worse, it is difficult even to heuristically check for reversibility by means of testing, as traditional testing techniques are designed to test extensional properties (*e.g.*, I/O behaviour) rather than intensional ones. For this reason, the problem of guaranteeing reversibility for programs is one best solved through cautious design of programming languages, such that the burden of proof may be alleviated from the programmer and placed with the language designer instead. We will return to this idea in Section 1.4 where we consider reversible programming languages.

1.1.1 Reversibility in denotational semantics

While reversibility is, as mentioned, traditionally presented from an operational viewpoint as given above, the focus in this dissertation will be on an alternative, denotational account of this phenomenon. For a program p, we let $[\![p]\!]$ denote its meaning in an appropriate *domain of computation* (or *semantic domain*). This domain is usually taken to be an appropriate algebraic structure: rather uncontroversially, we choose the view of semantic domains as categories. To define reversibility in this setting, we require first a notion of *semantic invertibility* of programs:

Definition 2. A program p is *semantically invertible* if there exists a *unique* map $[\![p]\!]^\dagger$ in its semantic domain such that

$$[\![p]\!] \circ [\![p]\!]^\dagger \circ [\![p]\!] = [\![p]\!] \qquad \text{and} \qquad [\![p]\!]^\dagger \circ [\![p]\!] \circ [\![p]\!]^\dagger = [\![p]\!]^\dagger.$$

In this case, we call $[\![p]\!]^\dagger$ the *semantic inverse* of p.

A critique of this criterion commonly encountered by the author when discussing reversibility with others is that it is not strict enough: That semantic invertibility really ought to mean that $[\![p]\!]$ is an isomorphism, *i.e.*, that we should have $[\![p]\!] \circ [\![p]\!]^\dagger = \mathrm{id}$ and $[\![p]\!]^\dagger \circ [\![p]\!] = \mathrm{id}$ instead. Unfortunately, this stronger criterion fails to capture all functions that are computable using a *reversible Turing machine* [21] (briefly, a Turing machine for which the transition function is injective), since these are precisely the partial computable injective functions [16]. While partial injective functions may fail to satisfy the stronger criterion, they all satisfy the definition given above.

Semantic invertibility captures similar aspects of programs as *global* forward and backward determinism did in the operational account: Since the semantic inverse is uniquely given, it serves as the unique backward semantics for the given program. However, semantic invertibility alone does not capture the local aspects of reversibility. To address this, we turn to a central principle of denotational semantics, namely that *denotational semantics are compositional semantics* [118]. Compositionality, in natural language and programming languages alike, refers to the idea that *"the meaning of each complex expression [..] is determined by the meanings of the component expressions plus the way they are combined into the complex expression."*[103]. This is often made formal (see [94] and, *e.g.*, [8] in the context of programming languages) by requiring that the interpretation $[\![\cdot]\!]$ is a *homomorphism* of algebras from the syntactic algebra (*i.e.*, the algebra of syntax trees with syntactic operations to compose these) into the semantic algebra (*i.e.*, the algebra of values and interpretations of operations). Concretely, this means that for any syntactic operator C in the syntactic algebra (with interpretation C' in the semantic algebra) and expressions $e_1, \ldots, e_n$ (*i.e.*, elements of the syntactic algebra), we have

$$[\![C(e_1, \ldots e_n)]\!] = C'([\![e_1]\!], \ldots, [\![e_n]\!]) \ .$$

For this reason, a given denotational semantics may be reasonably expected to provide a notion of a *meaningful subprogram* for a program by the principle of compositionality. It is precisely this notion we need to capture the local aspect and define reversibility in the denotational setting.

Definition 3. A program is reversible if all of its meaningful subprograms are semantically invertible.

Note that we consider any meaningful program (*i.e.*, any program p for which $[\![p]\!]$ exists) to be a meaningful subprogram of itself. As such, the above definition specifically requires any reversible program to be semantically invertible.

One might reasonably object to this definition by asserting that the notion of "meaningful subprogram" is vaguely defined, even in the face of a compositional semantics. We argue that the denotational definition ought to be only as specific as the operational one, and that a similar argument could be made for the operational account as to what constitutes a *computation step*. That is to say, both definitions should be read as "meta-definitions", only to be fully reified once a concrete model of computation is chosen.

No matter if one prefers the operational account (using forward and backward determinism, computation states and computation steps), or the denotational account (using invertibility and meaningful subprograms), their overall view of what constitutes a reversible program is the same: A reversible program should be constructed from forward and backward deterministic (resp. invertible) parts, and should only be combined in a way that preserves forward and backward determinism (resp. invertibility). The semantics of a reversible program should really be dictated by the semantics of its constituent components and how they are combined: Global behaviour should be explainable solely as a combination of local behaviours, such that reversibility may be determined by the invertibility of local behaviours and the preservation of invertibility by combinators. We summarise this idea as follows.

Thesis 1. *Reversible programs have compositional semantics.*

While the denotational account of reversibility may be reasonable, determining the reversibility of a program in this view becomes a daunting task: Not only do we need to show that it is semantically invertible, we also need to show this to be the case for all of its meaningful subprograms. This would be the case were we to consider a category such as **DCPO** (of directed-complete partial orders and continuous functions) for our semantic domain, as it is usually done. To avoid this, we seek instead to push our notion of reversibility from the level of program semantics to the level of semantic domains. A solution to this problem, first suggested by B. G. Giles [46], is to consider a particular class of categories as semantic domains for reversible programs that guarantee semantic invertibility: Inverse categories.

1.1.2 Inverse categories

Inverse categories are categories that internalise a notion of *partial invertibility*, as inverse semigroups do for semigroups. These are defined as follows (see [79]).

Definition 4. A category $\mathscr{C}$ is said to be an *inverse category* if for every morphism $f : X \to Y$ of $\mathscr{C}$, there exists a *unique* morphism $g : Y \to X$, called the *partial inverse* of f, such that $f \circ g \circ f = f$.

As is the case for inverse semigroups, it can be shown that requiring partial inverses to be unique is equivalent to requiring all idempotents to commute. Further, it follows that partial inverses are symmetrically assigned, *i.e.*, if f is partial inverse to g then g is partial inverse to f. Before we move further, we consider a few important examples of inverse categories.

Example 5. The category **PInj** of sets and partial injective functions is an inverse category. For a partial injective function $f : X \to Y$, its partial inverse $f^{(-1)}$ is given by the partial inverse in the set-theoretic sense, *i.e.*, the partial function

$$f^{(-1)}(y) = \begin{cases} x & \text{if } f(x) = y \\ \text{undefined} & \text{if } y \notin \text{im}(f) \end{cases} .$$

Example 6. Any groupoid is an inverse category, with the partial inverse of a morphism f given by the usual inverse f^{-1}.

Example 7. A partial function $f : (X, \mathcal{T}_X) \to (Y, \mathcal{T}_Y)$ between topological spaces is said to be a *partial homeomorphism* if it is injective, continuous, open, and defined on an open subset of X. It follows directly from this that the image of f is open in Y as well. Since injectivity, continuity, and openness are preserved under composition, and since ordinary homeomorphisms (specifically identities) are all injective, continuous, open, and defined on an open subset, it follows that topological spaces and partial homeomorphisms form a category, **PHomeo**. This is an inverse category, with the partial inverse $f^{(-1)}$ given precisely as in the case for **PInj**.

The view of inverse categories (and presumably also the name) as the categorical extensions of inverse semigroups came from the semigroup community [79]. Curiously, as with inverse semigroups, there is no need for a specialised notion of "inverse functor" – any functor between inverse categories automatically preserves inverses. Here, we will explore some more recent views on inverse categories, namely as certain instances of *dagger categories* and *restriction categories*.

A dagger category is a category that is self-dual (*i.e.*, satisfies $\mathscr{C} \cong \mathscr{C}^{\text{op}}$) in a canonical way. These are defined as follows (see, *e.g.*, [59, 107]).

Definition 8. A category $\mathscr{C}$ is said to be a *dagger category* if it is equipped with a contravariant, involutive, identity-on-objects endofunctor, *i.e.*, a functor $(-)^\dagger : \mathscr{C}^{\mathrm{op}} \to \mathscr{C}$ satisfying $\mathrm{id}_X^\dagger = \mathrm{id}_X$, $f^{\dagger\dagger} = f$, and $(g \circ f)^\dagger = f^\dagger \circ g^\dagger$ for all objects X and composable morphisms f, g of $\mathscr{C}$.

Note that a dagger is a structure on a category, and as such, a given category may be a dagger category in several different ways. As such, when specifying a dagger category, we really ought to specify *which* dagger structure on the category we are referring to, though this is often left implicit. A functor $F : \mathscr{C} \to \mathscr{D}$ between dagger categories is a *dagger functor* if it preserves the dagger structure, *i.e.*, if $F(f^\dagger) = F(f)^\dagger$ for all f. Examples of dagger categories include **Rel** of sets and relations (with $f^\dagger$ given by the relational converse f^{-1} of f), **PInj** of sets and partial injective functions (with $f^\dagger$ given by the partial inverse $f^{(-1)}$ of f), and **FHilb** of finite dimensional Hilbert spaces and linear maps (with $f^\dagger$ given by the Hermitian conjugate of f, which is typically also denoted $f^\dagger$).

Restriction categories is an axiomatic approach to partiality in categories, expressed by assigning to each morphism $f : X \to Y$ a *restriction idempotent* $\overline{f} : X \to X$ that may intuitively be thought of as a partial identity defined precisely where f is defined. These are defined (see [31, 32, 33]) as follows.

Definition 9. A restriction category is a category equipped with a combinator

$$\frac{f : X \to Y}{\overline{f} : X \to X}$$

such that the following axioms are satisfied:

(R1) $f \circ \overline{f} = f$,

(R2) $\overline{g} \circ \overline{f} = \overline{f} \circ \overline{g}$ when $\mathrm{dom}(f) = \mathrm{dom}(g)$,

(R3) $\overline{g \circ \overline{f}} = \overline{g} \circ \overline{f}$ when $\mathrm{dom}(f) = \mathrm{dom}(g)$, and

(R4) $\overline{g} \circ f = f \circ \overline{g \circ f}$ when $\mathrm{cod}(f) = \mathrm{dom}(g)$

for all such morphisms f, g.

Like dagger categories, a restriction combinator satisfying these equations is a structure on the category, not a property of it. A functor $F : \mathscr{C} \to \mathscr{D}$ between restriction categories preserving this restriction structure, *i.e.*, satisfying $F(\overline{f}) = \overline{F(f)}$ for all morphisms f, is called a *restriction functor*.

A prototypical example of a restriction category is **Pfn** of sets and partial functions, with the restriction idempotent $\overline{f}$ for a partial function f given by

$$\overline{f}(x) = \begin{cases} x & \text{if } f \text{ is defined at } x \\ \text{undefined} & \text{otherwise} \end{cases}$$

Another example, with a similar restriction structure, is **PTop** of topological spaces and partial continuous functions defined on open sets. The Kleisli category of the $(\cdot) + 1$ monad over any *distributive category* (*i.e.*, a category with products and coproducts such that products distribute over coproducts) can also be outfitted with a restriction structure [31] by defining $\overline{f}$ for a morphism $f : X \to Y$ as the composition

$$X \xrightarrow{\langle \mathrm{id},\mathrm{id} \rangle} X \times X \xrightarrow{\mathrm{id} \times f} X \times (Y + 1) \xrightarrow{\delta} (X \times Y) + (X \times 1) \xrightarrow{\pi_1 + \pi_2} X + 1$$

For our purposes, a particularly interesting class of morphisms in restriction categories is the class of *partial isomorphisms* (or *restricted isomorphisms*). A partial isomorphism in a restriction category is a morphism $f : X \to Y$ for which there exists a $g : Y \to X$ such that $g \circ f = \overline{f}$ and $f \circ g = \overline{g}$. As the name suggests, this notion generalises ordinary isomorphisms in that any isomorphism is a partial isomorphism. For some concrete examples, partial isomorphisms in **Pfn** are precisely the partial injective functions (*i.e.*, the morphisms of **PInj**), while the partial isomorphisms of **PTop** are partial homeomorphisms (that is, morphisms of **PHomeo**). Indeed, just like any ordinary category $\mathscr{C}$ induces a groupoid $\mathrm{Core}(\mathscr{C})$ as a subcategory consisting only of its isomorphisms, any restriction category $\mathscr{C}$ induces a subcategory of partial isomorphisms, $\mathrm{Inv}(\mathscr{C})$, which is an inverse category by the following characterisation.

Proposition 1. *Let $\mathscr{C}$ be a category. The following are equivalent:*

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

(ii) $\mathscr{C}$ is a restriction category and each morphism of $\mathscr{C}$ is a partial isomorphism.

(iii) $\mathscr{C}$ is a dagger category and for each morphism f of $\mathscr{C}$, $f^\dagger$ is the unique morphism such that $f \circ f^\dagger \circ f = f$.

Proof. See [31], Theorem 2.20. □

Given that inverse categories internalise the notion of partial invertibility, they are a particularly good fit for reversible computation when seen through the denotational lens. Consider any program p that can be assigned a compositional semantics $[\![p]\!]$ as a morphism in an inverse category $\mathscr{C}$. Since $\mathscr{C}$ is an inverse category, $[\![p]\!]$ is partially invertible, *i.e.*, p is semantically invertible. However, since p has been assigned compositional semantics in $\mathscr{C}$, any meaningful subprogram p' of p must also have an interpretation as a morphism $[\![p']\!]$ of $\mathscr{C}$; which, since $\mathscr{C}$ is inverse, must also be partially invertible, such that p' is semantically invertible. But since this argument could be applied to any meaningful subprogram of p, p must be reversible by the denotational defintion. This can be summarised in the slogan that *inverse categories are semantic domains for reversible programs*.

1.2 Unifying operational and denotational reversibility

We have so far presented two different accounts of reversibility: One that defines reversible computations to be those that are locally forward and backward deterministic, and another that defines reversible computations as those for which any meaningful subcomputation is semantically invertible. In the following, we will argue that the two describe the same phenomenon, as expressed in the following thesis.

Thesis 2. *The operational and denotational definitions of reversibility are equivalent.*

Starting with the operational account, we must consider the notion (due to Bennett [21], though this presentation follows [16] more closely) of a reversible Turing machine. Like their ordinary counterparts, reversible Turing machines serve as the fundamental computation model to study computability theory in a reversible setting.

1.2.1 Reversible Turing machines

Like a regular Turing machine, a reversible Turing machine T is a tuple $(Q, \Sigma, \delta, b, q_s, q_f)$ where Q is a (non-empty) finite set of *states*, Σ is a finite set of *tape symbols*, $b \in \Sigma$ is a distinguished *blank symbol*, $q_s \in Q$ and $q_f \in Q$ are the distinguished *start* respectively *final* states, and $\delta : Q \times \Sigma \to Q \times \Sigma \times \{\leftarrow, \downarrow, \to\}$ is a partial function, the *transition function*. For a given state $q \in Q$ and symbol $s \in \Sigma$, we take $\delta(q, s) = (q', s', d)$ to mean that if the machine is in state q and the symbol s is being read by the tape head, the machine should write the symbol s', go to the state q', and move the tape head according to $d \in \{\leftarrow, \downarrow, \to\}$ (*i.e.*, left if d is $\leftarrow$, right if it is $\to$, and stay if it is $\downarrow$).

From this we get a usual notion of *configuration* as a pair $(c, (l, s, r))$ where c is the state of the Turing machine, l and r are the contents of the tape to the left respectively right of the tape head, and s is the symbol directly under the tape head. Likewise, we get from the transition function a notion of a *computation step* as a single transition of the Turing machine that takes a configuration $(c, (l, s, r))$ to a new configuration $(c', (l', s', r'))$: We write $(c, (l, s, r)) \rightsquigarrow (c', (l', s', r'))$ when this is the case, and call the induced relation the *computation step relation*. This relation is key to defining reversible Turing machines:

Definition 10. A reversible Turing machine is a Turing machine for which the induced computation step relation is a partial injective function.

This definition exposes the origins of the operational definition of reversibility: A Turing machine is *locally forward deterministic* when the transition relation is a partial function, and *locally backward deterministic* when its inverse relation is a partial function: Put the two together, and you get that it must be a partial injective function to satisfy both requirements simultaneously.

As for ordinary Turing machines, reversible Turing machines are associated with a function it computes. It can be shown (see [16]) that reversible Turing machines are in a one-to-one correspondence with the computable partial injective functions: Every reversible Turing machine computes a (necessarily computable) partial injective function, and for every partial injective function computable on a regular Turing machine, there exists a reversible one that computes it as well. Curiously, no real computational power is lost when considering only reversible Turing machines: Bennett showed [21] that it is always possible to "reversibilise" any ordinary Turing machine (call it T, and call the function it computes T_f), by fashioning a reversible Turing machine that computes the function $x \mapsto (x, T_f(x))$.

Since the identity function on any given countable set is computable, and the composition of computable functions is again computable, sets and computable partial injective functions form a category, **CPInj**. Further, since a partial injective function is computable if and only if its partial inverse is (see, *e.g.*, [16]), this is an inverse category.

To see that this model lives up to the denotational definition of reversibility, we must consider what "meaningful subprogram" means in terms of reversible Turing machines. It seems clear that any meaningful subprogram of a reversible Turing machine T must only do a part of what T does. In other words, a meaningful subprogram of a reversible Turing machine T must be another Turing machine T' for which the transition relation of T', $\cdot \rightsquigarrow_{T'} \cdot$ is a subset of that of T, $\cdot \rightsquigarrow_T \cdot$. But since $\cdot \rightsquigarrow_T \cdot$ is a partial injective function, so must $\cdot \rightsquigarrow_{T'} \cdot$ be, so T' must also be reversible. But then T' must compute a computable partial injective function, which again is a morphism of **CPInj**. In this way, the standard model of reversible Turing machines, from which the operational definition was extracted, satisfies the denotational definition of reversibility.

1.2.2 Support categories and deterministic maps

An argument for the proposition that the denotational account respects the operational one comes from the theory of *support categories*. A support category (see [30]) is a category that is *almost* a restriction category, save for the fact that it only satisfies a weaker form of axiom **(R4)** of restriction categories:

Definition 11. A *support category* is a category equipped with a combinator

$$\frac{f : X \to Y}{\overline{f} : X \to X}$$

such that the following axioms are satisfied:

(R1) $f \circ \overline{f} = f$,

(R2) $\overline{g} \circ \overline{f} = \overline{f} \circ \overline{g}$ when $\mathrm{dom}(f) = \mathrm{dom}(g)$,

(R3) $\overline{g \circ \overline{f}} = \overline{g} \circ \overline{f}$ when $\mathrm{dom}(f) = \mathrm{dom}(g)$, and

(wR4) $\overline{\overline{g} \circ f} = \overline{g \circ f}$ when $\mathrm{cod}(f) = \mathrm{dom}(g)$

for all such morphisms f, g.

The reason for this related concept of support categories is that restriction categories often fail to capture partiality in categories that are somehow nondeterministic. For example, the category **Rel** of sets and relations, with the support $\overline{R} : X \to X$ of a relation $R : X \to Y$ given by $(x, x) \in \overline{R}$ iff there exists $y \in Y$ such that $(x, y) \in R$, gives a support structure on **Rel**, but not a restriction structure as it fails to satisfy **(R4)**. For this reason, the axiom **(R4)** is sometimes called the *axiom of determinism*. It can be shown that the axiom **(wR4)** is weaker than **(R4)** in the presence of the remaining axioms, as it is satisfied in any restriction category. As such, any restriction category is a support category as well.

Following this terminology from [30], a morphism f in a support category is said to be *deterministic* if it satisfies **(R4)** for all other morphisms g; *i.e.*, if for all g with $\mathrm{cod}(f) = \mathrm{dom}(g)$, $\overline{g} \circ f = f \circ \overline{g \circ f}$. It can be shown [30] that the deterministic morphisms of a support category $\mathscr{C}$ form a subcategory of $\mathscr{C}$, and that the support structure inherited from $\mathscr{C}$ is a full restriction structure in this subcategory. For example, the deterministic maps in **Rel** are precisely the partial functions, and so the subcategory of deterministic maps of **Rel** is the category **Pfn** of sets and partial functions. As such, any morphism in a restriction category can be said to be deterministic in this sense.

If we accept that the definition of determinism in a support category corresponds in a reasonable manner to determinism as we usually consider it, inverse categories – that is, semantic domains for reversible programs – are precisely categories in which both the forward semantics $[\![p']\!]$ and backward semantics $[\![p']\!]^\dagger$ of any meaningful subprogram p' of any program p' is deterministic.

1.3 Reversible computing as physical computing

While reversible computing has many applications in areas such as debugging, parser construction, fast discrete event simulation, and even robotics (see, *e.g.*, [29, 101, 104, 105, 106]), and may even be considered as more fundamental than classical (irreversible) computing [72, 111], one of the earliest motivations for studying reversible computing came from physics.

In his seminal 1961 paper [84], physicist Rolf Landauer established a connection between information and thermodynamics in what has since been dubbed *Landauer's principle*: The erasure of a single bit of information is associated with the dissipation of $kT \ln(2)$ joules of energy as heat (where k is Boltzmann's constant, and T is the temperature of the system in Kelvins). Since information in computing machinery must necessarily have a

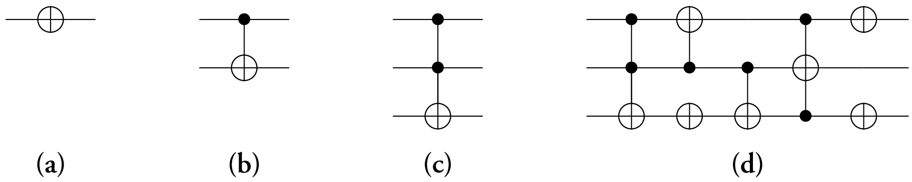


Figure 1.2: The NOT gate (a), CNOT gate (b), TOFFOLI gate (c), and a reversible circuit composed of NOT, CNOT, and TOFFOLI gates (d).

physical realisation, *logical irreversibility* of a computing process (such as resetting a bit to have a particular value), Landauer argues, inexorably leads to *physical irreversibility*. Since this physical irreversibility is associated with a decrease in entropy, which cannot occur in a closed system by the laws of thermodynamics, Landauer goes on to argue that this local decrease in entropy must be expressed as an increase in entropy in the environment, in the form of heat dissipation [84, p. 265]:

> *"Consider a statistical ensemble of bits in thermal equilibrium. If these are all reset to* ONE*, the number of states covered in the ensemble therefore has been reduced by* $k \ln(2) = 0{,}6931k$ *per bit. The entropy of a closed system,* e.g.*, a computer with its own batteries, cannot decrease; hence this entropy must appear elsewhere as a heating effect, supplying* $0{,}6931kT$ *per restored bit to the surroundings."*

On the other hand, logically reversible computations can have physically reversible realisations, which are, in turn, not associated with this heat dissipation in the ideal case.

As a physical principle, Landauer's principle has been a subject of much debate, notably in the *Norton-Ladyman controversy* [82, 83, 97] (see also [6]). Since then, the principle has seen redemption in the form of experimental validation [23, 98, 65] and even formal verification [6]. Aside from Landauer's principle, reversibility is also a key component in quantum computing: The quantum circuit model [40] relies on gates that are all *unitary*, *i.e.*, linear isomorphisms of Hilbert spaces that preserve the inner product.

1.3.1 Reversible gates and reversible circuits

While Landauer demonstrated the possibility for low-power computing via reversibility, it would take another twenty years before his ideas were fully formalised by Toffoli into a universal gate set for designing actual reversible computers. Though other gate sets have since appeared (notably the Fredkin gate set [44]), we focus here on the *Toffoli gate set* [114] (which we will use extensively in Chapter A).

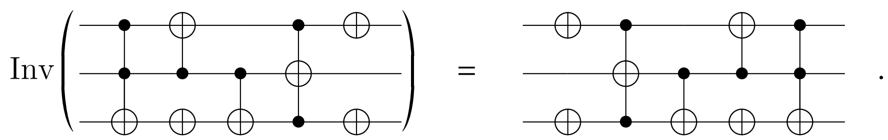
The Toffoli gate set consist of three gates, depicted in Figure 1.2: The NOT gate, the CNOT (or FEYNMAN) gate, and the TOFFOLI gate. The NOT gate works precisely as it does in irreversible computing, mapping 0 to 1 and vice versa. The CNOT gate (short for "controlled not") takes two inputs – one, marked with • is called the *control line*, while the other, marked with ⊕, is called the *target line* – and produces two outputs. The output of the control line always passes through unchanged, whereas a NOT gate is applied to the target line if the control line carries a value of 1 (if it is 0, the input of the target line passes through unchanged). In this way, the control line controls whether or not a NOT gate should be applied on the target line. This gives the truth table semantics shown in Figure 1.3 (where c_{in} and c_{out} denote the input respectively output on the control line, and t_{in} and t_{out} the input respectively output on the target line). Finally, the TOFFOLI gate (or CCNOT – "controlled controlled not" gate) is a generalisation of the CNOT gate to two control lines rather than just one. That is, a TOFFOLI gate has two control lines (marked with •) and single control line (marked with ⊕), and has the semantics that inputs pass through the control lines unchanged, while a NOT gate is applied to the target line only when *both* control lines carry a value of 1. If just one of the control lines carry a 0, the input of the target line passes through unchanged.

c_{in}	t_{in}		c_{out}	t_{out}
0	0		0	0
0	1	$\mapsto$	0	1
1	0		1	1
1	1		1	0

Figure 1.3: The truth table for the CNOT gate.

A reversible circuit may then be defined to be a finite network of NOT, CNOT, and TOFFOLI gates composed horizontally or vertically (corresponding to ordinary function composition respectively parallel composition) in which neither fan-in, fan-out nor loops occur. Figure 1.2 shows an example of such a circuit: Note that control lines needn't all be immediately above or below the target line, or even adjacent to it. As a result of the lack of fan-in and fan-out, all reversible circuits have the same number of input lines as output lines.

What makes these circuits reversible is the fact that the NOT, CNOT, and TOFFOLI gates are all invertible (in fact, they are *involutive*, *i.e.*, self-inverse), and that both horizontal and vertical composition preserves invertibility. As a result, a reversible circuit may be inverted simply by horizontally reversing the order in which gates appear in a circuit. For example, the inverse to the circuit shown in Figure 1.2 is the circuit

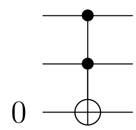


In the presence of *ancillae* (that is, input lines with a constant value assigned), the Toffoli gate set can be shown to be computationally universal with respect to finite Boolean bijections, in the sense that any finite Boolean bijective function can be expressed as a reversible circuit [114]. As such, when ancillary variables are allowed, it is common to generalise the Toffoli gate to allow an arbitrary number of control lines (not just two), as this has no effect on what functions can be expressed using this gate set, though it often simplifies presentation considerably. Even more, it is also *weakly universal* with respect to arbitrary finite Boolean functions: For any finite Boolean function $f : \mathbb{B}^n \to \mathbb{B}^m$, there exists a finite bijection $f' : \mathbb{B}^k \to \mathbb{B}^k$ (with $k \geq \max(n, m)$) computed by some reversible circuit, and a vector of constant inputs $(c_1, \dots, c_{k-n})$, such that for all $(x_1, \dots, x_n) \in \mathbb{B}^n$ with

$$f(x_1, \ldots, x_n) = (y_1, \ldots, y_m),$$

$$f'(c_1, \ldots, c_{k-n}, x_1, \ldots, x_n) = (g_1, \ldots, g_{k-m}, y_1, \ldots, y_m) \; .$$

In other words, any finite Boolean function may be computed by a reversible circuit if we accept the presence of additional ancillary inputs and *garbage* outputs [114]. For example, the usual AND gate cannot be represented "on the nose" as a reversible circuit since it is irreversible, but if we accept an additional input line with a constant value of 0, and two additional garbage outputs, we may straightforwardly represent it by the circuit



Here, the output of the target line gives precisely the conjunction of the two inputs, produced in the presence of garbage given by the outputs of the two control lines.

1.4 Reversibility and invertibility in programming languages

Though we consider reversibility to be a semantic property of programs, it is also one that requires very deliberate program construction and subsequent argumentation to assert. To avoid subtle bugs and errors related to reversibility, this problem is one best solved through language design.

A programming language is said to be reversible if all of its meaningful programs are. This statement, while accurate, is also deceptively simple in that it obscures *precisely what* must be done when it comes to language design in order to guarantee this property. Here, we will show this by example by considering the reversible general purpose programming language Janus [123, 119] in some detail. It must be stressed that many other reversible programming languages exist, *e.g.*, Theseus [73], Rfun [121, 112], `R-CORE` [52], `R-WHILE` [51], $\mathtt{RINT}_k$ (see [Paper C.1]), and others.

Janus is a *reversible flowchart language* [122] (with recursion), the reversible analogue of ordinary flowchart languages (see, *e.g.*, [75]), in that its control flow can be illustrated by means of flowcharts. It has a very simple scoping semantics: All variables are global (and may contain only integers), and all are assumed to carry the value `0` at the start of a program. Janus also supports procedures: However, unlike ordinary flowchart languages, procedures in Janus can not only be called, they can also be *uncalled*, *i.e.*, executed in reverse, since all Janus programs have a well-defined inverse semantics. We consider these structured reversible flowchart languages in much more detail in Chapter C.

Janus supports four different flowchart structures, as illustrated in Figure 1.4: Atomic steps, sequencing, conditionals, and loops. Atomic steps in Janus are instructions that reversibly modify a single global variable in the store: These are instructions such as `x +=`

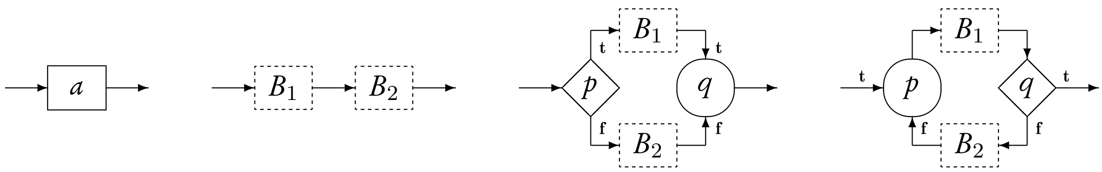


Figure 1.4: The reversible flowchart structures in Janus: Atomic steps, sequences, conditionals, and loops. Squares denote flowcharts, diamonds are *tests*, and circles are *assertions*. Figures adapted from [122].

`y` (add the contents of `y` to the contents of `x`), `x <=> y` (swap the contents of `x` and `y`), and so on. Sequences are simply that: Sequences of statements separated by carriage return. Conditionals, on the other hand, are a little more tricky: Unlike ordinary conditionals which has a single predicate that decides which branch to take, a Janus conditional takes the form `if` p `then` B_1 `else` B_2 `fi` q, where p and q are predicates, and B_1 and B_2 are statements. Here, the trailing predicate q must be fashioned in such a way as to be true whenever the *then*-branch is chosen by p, and false when the *else*-branch is chosen. This extra predicate is necessary to guarantee reversibility of the entire conditional, as it might not otherwise be possible to deduce which branch to choose in the backward direction.

Like conditionals, loops in Janus require an additional predicate in order to be reversible. That is, loops in Janus take the form `from` p `do` B_1 `loop` B_2 `until` q, where p is a predicate that must be true *before* any iterations of the loop, and *false* after one or more iterations. The semantics of such loops are the following: After checking the assertion p, the statement B_1 is executed, and the test q is executed: If it is true, the loop is exited; otherwise, the statement B_2 is executed and the program counter returns to the entry assertion to start anew. An example of a Janus program using loops is shown in Figure 1.5. Assuming that `x` and `y` are both zero cleared and that `n` contains some positive integer, after calling the `fib` procedure the pair of variables $(\mathtt{x}, \mathtt{y})$ will contain (f_n, f_{n+1}), where f_i is the i^{th} Fibonacci number. This procedure also an example of an *injectivisation* of a problem: The function mapping $n \mapsto f_n$ is not injective since the first and second Fibonacci numbers are both 1, but it can be made injective by changing its semantics slightly to compute Fibonacci pairs instead. Such small semantic changes to problems are common in practical reversible programming, as problems are often formulated in a non-injective way.

```
procedure fib
  y += 1
  from x = 0 do
    skip
  loop
    x += y
    x <=> y
    n -= 1
  until n = 0
```

Figure 1.5: The Fibonacci pair procedure in Janus.

Though Janus is a rather simple language compared to mainstream programming languages such as Java or Haskell, when augmented with a dynamic data structure such as a stack [119], it is as expressive as it can be reversibly, in the sense that it is *r-Turing com-*

plete [16, 119]: In other words, it is as computationally powerful as reversible Turing machines, which serve as the gold standard for reversible computability.

1.4.1 Program inversion and inverse interpretation

A feature of Janus (and other reversible languages) that is often highlighted is the fact that it comes equipped with a sound and complete *program inverter*. A program inverter is a program transformer Inv (that is, a program that takes program texts as inputs and produces them as outputs) satisfying for all program texts p that $[\![\mathrm{Inv}(\mathrm{p})]\!] = [\![\mathrm{p}]\!]^\dagger$. Though program inversion was first considered for general (irreversible) programming languages [36] (by hand, even), it has found a home in reversible programming, as program inverters are both very meaningful to have, and often straightforward to produce, for reversible languages.

Similar to program inverters are *inverse interpreter*. An inverse interpreter is a program InvInt satisfying for all program texts p and inputs x that $[\![\mathrm{InvInt}]\!](\mathrm{p}, x) = [\![\mathrm{p}]\!]^\dagger(x)$. Inverse interpretation goes back even further than program inversion (and was likewise originally considered for irreversible languages), as it was first studied when McCarthy developed his *generate-and-test* method in 1958. In Janus, such an inverse interpreter was constructed by remarkable means in [123]: Instead of implementing one directly, the authors instead implemented a self-interpreter for Janus (*i.e.*, a Janus interpreter written in Janus), and then showed that inverse interpretation could be achieved by using the *uncall*-functionality of Janus to uncall the self-interpreter.

Though this was not realised until much later [50] (see also [49]), inverse interpretation and program inversion are connected via the *Futamura projections* [45]. Given a program specialiser, a program inverter can be produced by specialising the specialiser with respect to the inverse interpreter. Note, however, that the language in which the inverse program produced by this program inverter is expressed is the target language of the specialiser. This is a subtle point regarding program inversion in reversible languages: While a deterministic inverse interpreter can always be fashioned for a reversible programming language $\mathcal{L}$ (though it may have to be written in another language), the language $\mathcal{L}$ may not be expressive enough to express its own inverse programs. We illustrate this by the following example.

Consider the following programming language, an imperative language with stores consisting of a single memory cell containing an integer, which we shall refer to as `Zinc` ("$\mathbb{Z}$ *with incrementation*"): The syntax is given by finite (possibly empty) lists of the symbol `+`, *i.e.*, ϵ, `'+'`, `'++'`, `'+++'`, and so on. Its semantics is given by the reduction relation $n \vdash p \downarrow n'$ (where p is a program, and n and n' are integers corresponding to the contents of the store before and after execution) defined as follows:

$$\frac{}{n \vdash \epsilon \downarrow n} \qquad \frac{n \vdash c_1 \downarrow n' \qquad n' \vdash c_2 \downarrow n''}{n \vdash c_1 c_2 \downarrow n''} \qquad \frac{}{n \vdash \texttt{'+'} \downarrow n+1}$$

For example, executing the program `'+++'` in a store containing the integer n results in a store containing the integer $n + 3$. This is a reversible language: For any `Zinc` command c, given any integer n there exists at most one n' such that $n \vdash c \downarrow n'$, and given any integer m there exists at most one m' such that $m' \vdash c \downarrow m$. However, since `Zinc` cannot express decrementation, inverse programs of `Zinc` programs are not expressible in `Zinc`: We say that `Zinc` (unlike Janus) is not *closed under program inversion*. Closure under program inversion should be considered a constructive property: Even if a reversible programming language is r-Turing complete (and, as such, strong enough to express its own inverse programs by the Church-Turing thesis, as a partial injective function is computable if and only if its inverse is), it should not be considered closed under program inversion until a (sound and complete) program inverter is produced.

1.5 Contributions

In this chapter, we have introduced the prevalent operational account of reversibility which goes back to the seminal work of Bennett [21], as well as a novel denotational account, which links up with pioneering work on inverse categories as models of reversible computing [46], that stresses the importance of *compositional semantics* for reversible programs. Further, using concepts from the literature on reversible computing and inverse category theory, we have argued why these two accounts should be seen as complimentary rather than competing. We have introduced Landauer's principle as the physical motivation for reversible computing, and Toffoli's gate set as a foundations for a practical realisation of these concepts. Finally, we have discussed reversibility as it applies to programming languages, and the role of program inversion and inverse interpretation in reversible programming.

The remaining part of this thesis is given as an appendix of manuscripts produced by the author (and others) during the course of the past three years. The appendix consists of seven papers split into three topical chapters, covering the areas of reversible circuit logic, foundations of reversible recursion, and the semantics of reversible programming languages. We outline below the main contributions of these papers.

1.5.1 Reversible circuit logic

The reversible circuit model, based on the Toffoli gate set (see Figure 1.2 on page 9), provides the mathematical foundation for the construction of real world reversible computers. Much like programs in high level languages, reversible circuits are not all created equal: Some are better than others at performing the same job, in that they require fewer gates or ancillary lines, or produce fewer garbage outputs.

Chapter A concerns itself with two approaches to the *equivalence problem* for reversible circuits: Given two reversible circuits C and $\mathcal{D}$, which thus compute finite bijective functions $f_C, f_{\mathcal{D}} : \mathbb{B}^k \to \mathbb{B}^k$, we wish to determine whether these two circuits are equivalent, *i.e.*, whether it is the case that $f_C(\mathbf{x}) = f_{\mathcal{D}}(\mathbf{x})$ for all $\mathbf{x} \in \mathbb{B}^k$. Unfortunately, as for the circuit equivalence problem for the conventional (irreversible) circuit model, this is a computationally intractable problem shown to be coNP-complete [76]. For this reason, the equivalence problem is one best solved by providing tools that aid humans in solving it for concrete instances.

One such tool is the hardware description language *Ricercar* presented in [Paper A.1], which comes equipped with an equational theory based on previous work by two of the authors [110]. The equational theory exploits the compositional nature of reversible circuits (*cf.* Thesis 1), and is similar to the template-based approach to reversible circuit optimisation that has become popular in recent years (see, *e.g.*, [12]).

Ricercar regards reversible circuits as constructed using only the IDENTITY and NOT gates as primitives: Circuits may be formed through the sequential composition of smaller circuits, or by letting their execution be *controlled* by an arbitrary Boolean formula. This view of *control* by a Boolean formula as an operation on circuits leads to a much wider applicability of the equational theory. Ricercar also introduces a notion of *scoped ancillary variables* as input variables which are guaranteed to be unchanged across the execution of the circuit, and also provides a set of equational rewriting rules for reasoning about circuits with scoped ancillae. Since ancillary lines are often used to temporarily store partial results from computations, assigning and maintaining scopes to ancillae facilitates reuse of ancillary lines, which may help in bringing down the overall circuit size. Unfortunately, though the Ricercar semantics and equational theory is sound with respect to the reversible circuit model, and though good results can often be obtained in practice, it is demonstrated to be incomplete (in that there exist two equivalent circuits that cannot be shown to be equivalent by means of the equational theory of Ricercar).

Where Ricercar adopts a view of circuits vaguely similar to that of *propositional dynamic logics* (see, *e.g.*, [42]), an approach to the equivalence problem much more reminiscent of classical propositional logic is chosen in [Paper A.2], through the development of the logic $\mathrm{LRS}_{\circledast}$. In this view, reversible circuits (much like ordinary, irreversible circuits in classical propositional logic) are expressible as propositions, and propositional equivalence of circuit representations correspond to the equivalence of represented circuits. The result is a two-level proof calculus that is sound and complete with respect to a semantics based on so-called *Toffoli lattices*, which, in turn, is shown to be equationally complete with respect to reversible logic circuits. This, finally, leads to a sound and complete equational theory for propositional representations of reversible circuits. However, this approach comes at the cost that propositions are not guaranteed to correspond to reversible circuits: Reversible circuits merely embed in the logic as propositions in an equivalence-preserving way.

1.5.2 Foundations of reversible recursion

Out of all language features, the ability to express (primitively or generally) recursive procedures is often what separates simple, toy-like languages from fully fledged programming languages capable of expressing sophisticated computations. In Chapter B of this thesis, we investigate the role of recursion in reversible programming languages and how to model it in a categorical setting.

In existing reversible programming languages, recursion presents itself in several different forms: Rfun [121, 112] allows generally recursive functions[1], while Theseus [72, 73] allows only tail recursive functions to be expressed by means of *iteration labels* (a syntactic sugarcoating of a sufficiently well-behaved *trace operator* [77, 73, 108]). Even more, Janus [123, 119] (see also Section 1.4) supports both general recursion and tail recursion in the form of recursive procedures respectively reversible *while*-loops.

In the irreversible case, recursion (be it recursively defined procedures or data types) is often studied through the lens of *domain theory* (see, *e.g.*, [7]), *i.e.*, by means of categories enriched in the category **DCPO** of directed complete partial orders and continuous maps. Following the mantra that inverse categories are models of reversible computing (as discussed in Section 1.1), we give a domain theoretic account of inverse categories outfitted with *joins* (in the sense of [53], but see also [115]) in [Paper B.1] (an extended version of a previously published conference paper and workshop abstract). In particular, it is shown that having at least countable joins leads to a **DCPO**-enrichment for which the canonical dagger is locally continuous. This gives a family of fixed point operators $\mathrm{fix} : (\mathscr{C}(X, Y) \to \mathscr{C}(X, Y)) \to \mathscr{C}(X, Y)$ in the join inverse category $\mathscr{C}$ for modelling generally recursive first-order functions, and (crucially, as far as reversibility is concerned) it is shown that each continuous map $\varphi : \mathscr{C}(X, Y) \to \mathscr{C}(X, Y)$ has a fixed point adjoint $\varphi_\ddagger : \mathscr{C}(Y, X) \to \mathscr{C}(Y, X)$ such that $(\mathrm{fix}\,\varphi)^\dagger = \mathrm{fix}\,\varphi_\ddagger$.

Essentially, this result states that inversion of recursive maps can be solved by local means alone, a result which is consistent with established procedures for inversion of recursively defined reversible programs (*e.g.*, in Rfun [121]). Using a representation theorem for join restriction categories from [53], it is further shown that each join inverse category can be embedded in one that is algebraically ω-compact for locally continuous (in particular, join-preserving) functors, which may be used to model recursively defined data types. Finally, it is shown that when outfitted with a *disjointness tensor* [46] that preserves joins, the join inverse category may be canonically regarded as a *unique decomposition category* [55], equipping it with a trace operator that is shown to be preserve the canonical dagger – in other words, modelling reversible tail recursion in the style of Theseus.

In [Paper B.2], we seek to generalise key results from [Paper B.1] by asking what features of a join inverse category are necessary for fixed point adjoints to exist, and if they can be made canonical in some way. For this reason, we study the more general dagger categories enriched in **DCPO**. Unlike inverse categories, these categories also model computations that are not reversible under the Copenhagen interpretation (see page 2), but still have adjoints[2]: Examples include relational, doubly-stochastic, and even certain quantum

[1] Which, to the initial astonishment of the author, works precisely as it does in the irreversible case: Using a call stack, no changes required.

[2] By an *adjoint* to some $f : X \to Y$ we simply mean a "partner" $f^\dagger : Y \to X$, with no additional requirements.

computations. It is shown that for a dagger category enriched in **DCPO** to have fixed point adjoints (as in [Paper B.1]), it is sufficient that the dagger is locally monotone, a result that even extends to *parametrised* fixed points (see, *e.g.*, [38]). The requirement of local monotony of the dagger can be seen as an instance of *the way of the dagger* [62], a tenet of dagger category theory stating that all structure in sight must cooperate with the dagger. We show that an induced category of *functionals* over such a **DCPO**-enriched dagger category is canonically an *involutive monoidal category* [70], a view enables us to explain seemingly unrelated notions such as fixed point adjoints (in the sense of [Paper B.1]), dagger-preserving natural transformations, and the ambidexterity of dagger adjunctions in terms of *conjugation* of functionals. In particular, there is a world beyond the strict Copenhagen interpretation of reversibility where reversible recursion is still well-behaved.

1.5.3 Semantics of reversible programming languages

In the final chapter of the thesis, Chapter C, we consider the applications of inverse category theory in giving semantics to (aspects of) reversible programming languages.

In [Paper C.1] (an extended version of a previously published conference paper), we develop categorical semantics for *structured reversible flowchart languages* (of which Janus without recursion is an example), *i.e.*, imperative reversible programming languages with a control flow that can be illustrated purely by means of flowchart structures (Figure 1.4 on page 10 shows the flowchart structures available in Janus without recursion). Since many of these flowchart structures involve *tests* (and *assertions*), this hinges on a reversible representation of these as predicates (and their inverses), which we provide in the form of *decisions* (inspired by the analogous notion in restriction category theory, see [33]). Using these decisions, we give semantics to a universal class of reversible flowchart structures in certain join inverse categories, using some of the machinery developed in [Paper A.1] to give semantics to reversible *while*-loops via the dagger trace operator. We likewise provide an operational semantics for the reversible flowchart structures and tests, and show that our categorical semantics are both sound and adequate with respect to the operational ones. We additionally provide a sufficient condition for *full abstraction* (see, *e.g.*, [8, 99]), and demonstrate how a program inverter can be extracted from the semantics. We also show how the developed techniques can be used to straightforwardly give semantics to a class of structured reversible flowchart languages called $\texttt{RINT}_k$ (a family of structured reversible flowchart languages inspired by *reversible counter machines* [95, 78]) in the category **PInj** of sets and partial injective functions.

In [Paper C.2], we give an account of *reversible effects* in reversible programming. Monads are the typically the weapon of choice for modelling effects in the irreversible case (see, *e.g.*, [93, 116]), and though these have an analogue in dagger categories as *Frobenius monads* [62], we were only able to produce trivial or otherwise contrived examples of Frobenius monads in inverse categories. For this reason, we develop instead reversible versions of *arrows* [67, 71] (which generalise monads) as *dagger arrows* and *inverse arrows*. These arrows correspond to arrows between dagger (respectively inverse) categories that preserve inversion. We extend the arrow laws with laws for dagger and inverse arrows, and demonstrate their applicability by a number of examples, notably stateful reversible computations and a reversible variant of the *Writer* monad which we call the *Rewriter* arrow as inverse arrows, and the CPM construction [107] used to lift pure quantum computations to mixed ones as a dagger arrow. Following the idea that arrows, like monads, are monoids [61] (here in the category of *profunctors*), we show that dagger arrows correspond precisely to *involutive monoids*, and inverse arrows to dagger arrows satisfying additional equations.

Finally, in the brief [Paper C.3], we present a vision for the future of the reversible functional programming language Rfun [121]. New features include support for *ancillary variables*, which can be seen as a slight generalisation of the *parametrised maps* found in Theseus [73], and the abolishment of the duplication/equality operator[3] in favor of a combined *relevant* (see, *e.g.*, [37]) and unrestricted type system (see also [100]) that allows for reversible duplication and deduplication of first-order data.

[3]A source of confusion for students and reviewers alike.

References

[1] S. M. Abramov and R. Glück. The universal resolving algorithm: inverse computation in a functional language. In R. Backhouse and J. N. Oliveira, editors, *Proceedings of the 5th International Conference on Mathematics of Program Construction (MPC 2000)*, volume 1837 of *Lecture Notes in Computer Science*, pages 187–212. Springer-Verlag, 2000.

[2] S. M. Abramov and R. Glück. Principles of inverse computation and the universal resolving algorithm. In T. Æ. Mogensen, D. A. Schmidt, and I. H. Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation*, volume 2566 of *Lecture Notes in Computer Science*, pages 269–295. Springer-Verlag, 2002.

[3] S. Abramsky. Retracing some paths in process algebra. In U. Montanari and V. Sassone, editors, *CONCUR '96*, pages 1–17. Springer, 1996.

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

[5] S. Abramsky, E. Haghverdi, and P. Scott. Geometry of interaction and linear combinatory algebras. *Mathematical Structures in Computer Science*, 12(5):625–665, 2002.

[6] S. Abramsky and D. Horsman. DEMONIC programming: a computational language for single-particle equilibrium thermodynamics, and its formal semantics. In C. Heunen, P. Selinger, and J. Vicary, editors, *Proceedings of the 12th International Workshop on Quantum Physics and Logic (QPL 2015)*, volume 195 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–16. Open Publishing Association, 2015.

[7] S. Abramsky and A. Jung. Domain Theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Oxford University Press, 1994.

[8] S. Abramsky and C.-H. L. Ong. Full abstraction in the lazy lambda calculus. *Information and Computation*, 105(2):159–267, 1993.

[9] J. Adámek. Recursive data types in algebraically ω-complete categories. *Information and Computation*, 118:181–190, 1995.

[10] J. Adámek and V. Koubek. Least fixed point of a functor. *Journal of Computer and System Sciences*, 19(2):163–178, 1979.

[11] T. Altenkirch, J. Chapman, and T. Uustalu. Monads need not be endofunctors. *Logical Methods in Computer Science*, 11(1), 2015.

[12] M. Arabzadeh, M. Saeedi, and M. S. Zamani. Rule-based optimization of reversible circuits. In *Proceedings of the 15th Asia and South-Pacific Design Automation Conference (ASP-DAC 2010)*, pages 849–854, 2010.

[13] S. Awodey. *Category theory*, volume 52 of *Oxford Logic Guides*. Oxford University Press, second edition, 2010.

[14] H. B. Axelsen and R. Glück. A simple and efficient universal reversible turing machine. In A.-H. Dediu, S. Inenaga, and C. Martín-Vide, editors, *Proceedings of the 5th International Conference on Language and Automata Theory and Applications (LATA 2011)*, pages 117–128. Springer-Verlag, 2011.

[15] H. B. Axelsen and R. Glück. What do reversible programs compute? In M. Hofmann, editor, *Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 6604 of *Lecture Notes in Computer Science*, pages 42–56. Springer-Verlag, 2011.

[16] H. B. Axelsen and R. Glück. On reversible turing machines and their function universality. *Acta Informatica*, 53(5):509–543, 2016.

[17] H. B. Axelsen, R. Glück, and T. Yokoyama. Reversible machine code and its abstract processor architecture. In V. Diekert, M. V. Volkov, and A. Voronkov, editors, *Proceedings of the Second International Symposium on Computer Science in Russia (CSR 2007)*, volume 4649 of *Lecture Notes in Computer Science*, pages 56–69. Springer-Verlag, 2007.

[18] E. S. Bainbridge, P. J. Freyd, A. Scedrov, and P. J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70(1):35–64, 1990.

[19] M. Barr. Algebraically compact functors. *Journal of Pure and Applied Algebra*, 82(3):211–231, 1992.

[20] E. J. Beggs and S. Majid. Bar categories and star operations. *Algebras and Representation Theory*, 12(2):103–152, 2009.

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

[22] N. Benton and M. Hyland. Traced premonoidal categories. *Theoretical Informatics and Applications*, 37(4):273–299, 2003.

[23] A. Bérut, A. Arakelyan, A. Petrosyan, S. Ciliberto, R. Dillenschneider, and E. Lutz. Experimental verification of Landauer's principle linking information and thermodynamics. *Nature*, 483(7388):187–189, 2012.

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

[25] W. J. Bowman, R. P. James, and A. Sabry. Dagger traced symmetric monoidal categories and reversible programming. In A. De Vos and R. Wille, editors, *Preproceedings of the 3rd International Workshop on Reversible Computation (RC 2011)*, pages 51–56. Ghent University, 2011.

[26] A. Carboni, S. Lack, and R. F. C. Walters. Introduction to extensive and distributive categories. *Journal of Pure and Applied Algebra*, 84(2):145 – 158, 1993.

[27] J. Carette and A. Sabry. Computing with semirings and weak rig groupoids. In P. Thiemann, editor, *Proceedings of the 25th European Symposium on Programming (ESOP 2016)*, volume 9632 of *Lecture Notes in Computer Science*, pages 123–148. Springer-Verlag, 2016.

[28] C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Transactions on Modeling and Computer Simulation*, 9(3):224–253, 1999.

[29] S.-K. Chen, W. K. Fuchs, and J.-Y. Chung. Reversible debugging using program instrumentation. *IEEE Transactions on Software Engineering*, 27(8):715–727, 2001.

[30] J. R. B. Cockett, X. Guo, and P. Hofstra. Range categories i: General theory. *Theory and Applications of Categories*, 26(17):412–452, 2012.

[31] J. R. B. Cockett and S. Lack. Restriction categories I: Categories of partial maps. *Theoretical Computer Science*, 270(1–2):223–259, 2002.

[32] J. R. B. Cockett and S. Lack. Restriction categories II: Partial map classification. *Theoretical Computer Science*, 294(1–2):61–102, 2003.

[33] J. R. B. Cockett and S. Lack. Restriction categories III: Colimits, partial limits and extensivity. *Mathematical Structures in Computer Science*, 17(4):775–817, 2007.

[34] R. Cockett and R. Garner. Restriction categories as enriched categories. *Theoretical Computer Science*, 523:37–55, 2014.

[35] I. Cristescu, J. Krivine, and D. Varacca. A compositional semantics for the reversible π-calculus. In *Proceedings of the 28th Annual IEEE/ACM Symposium on Logic in Computer Science (LICS 2013)*, pages 388–397. IEEE Computer Society, 2013.

[36] E. W. Dijkstra. Program inversion. In F. L. Bauer and M. Broy, editors, *Program Construction, International Summer School*, pages 54–57. Springer-Verlag, 1979.

[37] J. M. Dunn and G. Restall. Relevance logic. In D. M. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic*, volume 6, pages 1–136. Kluwer Academic Publishers, 2002.

[38] Z. Ésik. Fixed point theory. In M. Droste, W. Kuich, and H. Vogler, editors, *Handbook of Weighted Automata*, pages 29–65. Springer-Verlag, 2009.

[39] Z. Ésik. Equational properties of fixed point operations in cartesian categories: An overview. In G. Italiano, G. Pighizzini, and D. Sannella, editors, *Proceedings of the 40th International Symposium on Mathematical Foundations of Computer Science (MFCS 2015), Part I*, pages 18–37. Springer-Verlag, 2015.

[40] R. P. Feynman. Quantum mechanical computers. *Optics News*, 11(2):11–20, 1985.

[41] M. P. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps*. PhD thesis, University of Edinburgh, 1994.

[42] M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194–211, 1979.

[43] M. P. Frank. *Reversibility for efficient computing*. PhD thesis, Massachusetts Institute of Technology, 1999.

[44] E. Fredkin and T. Toffoli. Conservative logic. *International Journal of Theoretical Physics*, 21(3-4):219–253, 1982.

[45] Y. Futamura. Partial computation of programs. In E. Goto, K. Furukawa, R. Nakajima, I. Nakata, and A. Yonezawa, editors, *RIMS Symposia on Software Science and Engineering, Proceedings*, pages 1–35. Springer-Verlag, 1983.

[46] B. G. Giles. *An Investigation of some Theoretical Aspects of Reversible Computing*. PhD thesis, University of Calgary, 2014.

[47] R. Glück and M. Kawabe. A program inverter for a functional language with equality and constructors. In A. Ohori, editor, *Proceedings of the First Asian Symposium on Programming Languages and Systems (APLAS 2003)*, volume 2895 of *Lecture Notes in Computer Science*, pages 246–264. Springer-Verlag, 2003.

[48] R. Glück and M. Kawabe. Derivation of deterministic inverse programs based on LR parsing. In Y. Kameyama and P. J. Stuckey, editors, *Proceedings of the 7th International Symposium on Functional and Logic Programming (FLOPS 2004)*, volume 2998 of *Lecture Notes in Computer Science*, pages 291–306. Springer-Verlag, 2004.

[49] R. Glück, Y. Kawada, and T. Hashimoto. Transforming interpreters into inverse interpreters by partial evaluation. In M. Leuschel, editor, *Proceedings of the 2003 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM '03)*, pages 10–19. ACM Press, 2003.

[50] R. Glück and M. H. Sørensen. A roadmap to metacomputation by supercompilation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation: International Seminar, Selected Papers*, pages 137–160. Springer-Verlag, 1996.

[51] R. Glück and T. Yokoyama. A linear-time self-interpreter of a reversible imperative language. *Computer Software*, 33(3):108–128, 2016.

[52] R. Glück and T. Yokoyama. A minimalist's reversible while language. *IEICE Transactions on Information and Systems*, E100-D(5):1026–1034, 2017.

[53] X. Guo. *Products, Joins, Meets, and Ranges in Restriction Categories*. PhD thesis, University of Calgary, 2012.

[54] E. Haghverdi. *A Categorical Approach to Linear Logic, Geometry of Proofs and Full Completeness*. PhD thesis, Carlton University and University of Ottawa, 2000.

[55] E. Haghverdi. Unique decomposition categories, Geometry of Interaction and combinatory logic. *Mathematical Structures in Computer Science*, 10(2):205–230, 2000.

[56] M. Hasegawa. Decomposing typed lambda calculus into a couple of categorical programming languages. In D. Pitt, D. E. Rydeheard, and P. Johnstone, editors, *Proceedings of the 6th International Conference on Category Theory and Computer Science (CTCS '95)*, volume 953 of *Lecture Notes in Computer Science*, pages 200–219. Springer-Verlag, 1995.

[57] M. Hasegawa. *Models of Sharing Graphs*. PhD thesis, University of Edinburgh, 1997.

[58] M. Hasegawa. Recursion from cyclic sharing: Traced monoidal categories and models of cyclic lambda calculi. In P. de Groote and J. R. Hindley, editors, *Proceedings of*

the Third International Conference on Typed Lambda Calculi and Applications (TLCA '97), volume 1210 of *Lecture Notes in Computer Science*, pages 196–213. Springer-Verlag, 1997.

[59] C. Heunen. *Categorical quantum models and logics*. PhD thesis, Radboud University Nijmegen, 2009.

[60] C. Heunen. On the functor ℓ^2. In *Computation, Logic, Games, and Quantum Foundations - The Many Facets of Samson Abramsky*, volume 7860 of *Lecture Notes in Computer Science*, pages 107–121. Springer-Verlag, 2013.

[61] C. Heunen and B. Jacobs. Arrows, like monads, are monoids. In S. D. Brookes and M. W. Mislove, editors, *Proceedings of the 22nd Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXII)*, volume 158 of *Electronic Notes in Theoretical Computer Science*, pages 219–236, 2006.

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

[63] P. M. Hines. *The Algebra of Self-Similarity and its Applications*. PhD thesis, University of Wales, Bangor, 1998.

[64] P. M. Hines and S. Braunstein. The structure of partial isometries. In S. Gay and I. Mackie, editors, *Semantic Techniques in Quantum Computation*, pages 361–388. Cambridge University Press, 2009.

[65] J. Hong, B. Lambson, S. Dhuey, and J. Bokor. Experimental test of Landauer's principle in single-bit operations on nanomagnetic memory bits. *Science Advances*, 2(3), 2016.

[66] N. Hoshino. A representation theorem for unique decomposition categories. In U. Berger and M. Mislove, editors, *Proceedings of the 28th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVIII)*, volume 286 of *Electronic Notes in Theoretical Computer Science*, pages 213–227. Elsevier, 2012.

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

[68] M. Hyland. Abstract and concrete models for recursion. In O. Grumberg, T. Nipkow, and C. Pfaller, editors, *Proceedings of the NATO Advanced Study Institute on Formal Logical Methods for System Security and Correctness*, pages 175–198. IOS Press, 2008.

[69] B. Jacobs. Semantics of weakening and contraction. *Annals of Pure and Applied Logic*, 69:73–106, 1994.

[70] B. Jacobs. Involutive categories and monoids, with a GNS-correspondence. *Foundations of Physics*, 42(7):874–895, 2012.

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

[72] R. P. James and A. Sabry. Information effects. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 73–84. ACM, 2012.

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

[74] P. T. Johnstone. *Sketches of an elephant: A topos theory compendium*, volume 1. Oxford University Press, 2002.

[75] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, 1993.

[76] S. P. Jordan. Strong equivalence of reversible circuits is coNP-complete. *Quantum Information and Computation*, 14(15-16):1302–1307, 2014.

[77] A. Joyal, R. Street, and D. Verity. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society*, 119(3):447–468, 1996.

[78] J. Kari and N. Ollinger. Periodicity and immortality in reversible computing. In E. Ochmański and J. Tyszkiewicz, editors, *Proceedings of the 33rd International Symposium on Mathematical Foundations of Computer Science (MFCS 2008)*, pages 419–430. Springer-Verlag, 2008.

[79] J. Kastl. Inverse categories. In H.-J. Hoehnke, editor, *Algebraische Modelle, Kategorien und Gruppoide*, volume 7 of *Studien zur Algebra und ihre Anwendungen*, pages 51–60. Akademie Verlag, 1979.

[80] M. Kawabe and R. Glück. The program inverter lrinv and its structure. In M. V. Hermenegildo and D. Cabeza, editors, *Proceedings of the 7th International Conference on Practical Aspects of Declarative Languages (PADL '05)*, volume 3350 of *Lecture Notes in Computer Science*, pages 219–234. Springer-Verlag, 2005.

[81] G. M. Kelly. *Basic Concepts of Enriched Category Theory*, volume 64 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, 1982.

[82] J. Ladyman, S. Presnell, A. J. Short, and B. Groisman. The connection between logical and thermodynamic irreversibility. *Studies in History and Philosophy of Science Part B: Studies in History and Philosophy of Modern Physics*, 38(1):58–79, 2007.

[83] J. Ladyman and K. Robertson. Landauer defended: Reply to norton. *Studies in History and Philosophy of Science Part B: Studies in History and Philosophy of Modern Physics*, 44(3):263–271, 2013.

[84] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5(3):183–191, 1961.

[85] M. L. Laplaza. Coherence for distributivity. In G. M. Kelly, M. L. Laplaza, G. Lewis, and S. Mac Lane, editors, *Coherence in Categories*, volume 281 of *Lecture Notes in Mathematics*, pages 29–65. Springer-Verlag, 1972.

[86] M. V. Lawson. *Inverse Semigroups: The Theory of Partial Symmetries*. World Scientific, 1998.

[87] S. Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, second edition, 1998.

[88] E. G. Manes and M. A. Arbib. *Algebraic approaches to program semantics*. Springer-Verlag, 1986.

[89] E. G. Manes and D. B. Benson. The inverse semigroup of a sum-ordered semiring. *Semigroup Forum*, 31(1):129–152, 1985.

[90] M. L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, Inc., 1967.

[91] T. Æ. Mogensen. Partial evaluation of the reversible language Janus. In *Proceedings of the 20th ACM SIGPLAN workshop on Partial Evaluation and Program Manipulation (PEPM '11)*, pages 23–32. ACM Press, 2011.

[92] T. Æ. Mogensen. Garbage collection for reversible functional languages. In J. Krivine and J.-B. Stefani, editors, *Proceedings of the 7th International Conference on Reversible Computation (RC 2015)*, volume 9138 of *Lecture Notes in Computer Science*, pages 79–94. Springer-Verlag, 2015.

[93] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[94] R. Montague. Universal grammar. *Theoria*, 36(3):373–398, 1970.

[95] K. Morita. Universality of a reversible two-counter machine. *Theoretical Computer Science*, 168(2):303–320, 1996.

[96] K. Nakata and T. Uustalu. Trace-based coinductive operational semantics for while. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *Lecture Notes in Computer Science*, pages 375–390. Springer-Verlag, 2009.

[97] J. D. Norton. Waiting for Landauer. *Studies in History and Philosophy of Science Part B: Studies in History and Philosophy of Modern Physics*, 42(3):184–198, 2011.

[98] A. O. Orlov, C. S. Lent, C. C. Thorpe, G. P. Boechler, and G. L. Snider. Experimental test of Landauer's principle at the sub-$k_b t$ level. *Japanese Journal of Applied Physics*, 51(6S):06FE10, 2012.

[99] G. Plotkin. Full abstraction, totality and pcf. *Mathematical. Structures in Comp. Sci.*, 9(1):1–20, 1999.

[100] J. Polakow. *Ordered linear logic and applications*. PhD thesis, Carnegie Mellon University, 2001.

[101] T. Rendel and K. Ostermann. Invertible syntax descriptions: Unifying parsing and pretty printing. *SIGPLAN Notices*, 45(11):1–12, 2010.

[102] E. Robinson and G. Rosolini. Categories of partial maps. *Information and Computation*, 79:95–130, 1988.

[103] G. Sandu and J. Hintikka. Aspects of compositionality. *Journal of Logic, Language and Information*, 10(1):49–61, 2001.

[104] M. Schordan, D. Jefferson, P. Barnes, T. Oppelstrup, and D. Quinlan. Reverse code generation for parallel discrete event simulation. In J. Krivine and J.-B. Stefani, editors, *Proceedings of the 7th International Conference on Reversible Computation (RC 2015)*, volume 9138 of *Lecture Notes in Computer Science*, pages 95–110. Springer-Verlag, 2015.

[105] U. P. Schultz, M. Bordignon, and K. Støy. Robust and reversible execution of self-reconfiguration sequences. *Robotica*, 29(1):35–57, 2011.

[106] U. P. Schultz, J. S. Laursen, L.-P. Ellekilde, and H. B. Axelsen. Towards a domain-specific language for reversible assembly sequences. In J. Krivine and J.-B. Stefani, editors, *Proceedings of the 7th International Conference on Reversible Computation (RC 2015)*, volume 9138 of *Lecture Notes in Computer Science*, pages 111–126. Springer-Verlag, 2015.

[107] P. Selinger. Dagger compact closed categories and completely positive maps. In *Proceedings of the 3rd International Workshop on Quantum Programming Languages (QPL 2005)*, volume 170 of *Electronic Notes in Theoretical Computer Science*, pages 139–163. Elsevier, 2007.

[108] P. Selinger. A survey of graphical languages for monoidal categories. In B. Coecke, editor, *New Structures for Physics*, volume 813 of *Lecture Notes in Physics*, pages 289–355. Springer-Verlag, 2011.

[109] M. B. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal on Computing*, 11(4):761–783, 1982.

[110] M. Soeken and M. K. Thomsen. White dots do matter: Rewriting reversible logic circuits. In G. W. Dueck and D. M. Miller, editors, *Proceedings of the 5th International Conference on Reversible Computation (RC 2013)*, volume 7948 of *Lecture Notes in Computer Science*, pages 196–208, 2013.

[111] Z. Sparks and A. Sabry. Superstructural reversible logic. Presented at the *3rd International Workshop on Linearity*, available at `https://www.cs.indiana.edu/~sabry/papers/superstructural.pdf`, 2014.

[112] M. K. Thomsen and H. B. Axelsen. Interpretation and programming of the reversible functional language rfun. In R. Lämmel, editor, *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages (IFL '15)*, pages 8:1–8:13. ACM Press, 2015.

[113] M. K. Thomsen, H. B. Axelsen, and R. Glück. A reversible processor architecture and its reversible logic design. In A. De Vos and R. Wille, editors, *Proceedings of the 3rd International Workshop on Reversible Computation (RC 2011)*, volume 7165 of *Lecture Notes in Computer Science*, pages 30–42. Springer-Verlag, 2012.

[114] T. Toffoli. Reversible computing. In J. W. de Bakker and J. van Leeuwen, editors, *Proceedings of the 7th International Colloquium on Automata, Languages and Programming (ICALP 1980)*, pages 632–644. Springer-Verlag, 1980.

[115] T. Uustalu and N. Veltri. The delay monad and restriction categories. In D. V. Hung and D. Kapur, editors, *Proceedings of the 14th International Colloquium on Theoretical Aspects of Computing (ICTAC 2017)*, volume 10580 of *Lecture Notes in Computer Science*, pages 32–50. Springer-Verlag, 2017.

[116] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *First International Spring School on Advanced Functional Programming Techniques, Tutorial Text*, pages 24–52. Springer-Verlag, 1995.

[117] M. Wand. Fixed point constructions in order-enriched categories. *Theoretical Computer Science*, 8(1):13–30, 1979.

[118] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

[119] T. Yokoyama, H. B. Axelsen, and R. Glück. Principles of a reversible programming language. In A. Ramírez, G. Bilardi, and M. Gschwind, editors, *Proceedings of the 5th Conference on Computing Frontiers (CF '08)*, pages 43–54. ACM Press, 2008.

[120] T. Yokoyama, H. B. Axelsen, and R. Glück. Reversible flowchart languages and the structured reversible program theorem. In L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, editors, *Proceedings of the 35th International Colloquium on Automata, Languages and Programming (ICALP 2008)*, volume 5126 of *Lecture Notes in Computer Science*, pages 258–270. Springer-Verlag, 2008.

[121] T. Yokoyama, H. B. Axelsen, and R. Glück. Towards a reversible functional language. In A. De Vos and R. Wille, editors, *Proceedings of the 3rd Workshop on Reversible Computation (RC 2011)*, volume 7165 of *Lecture Notes in Computer Science*, pages 14–29. Springer-Verlag, 2012.

[122] T. Yokoyama, H. B. Axelsen, and R. Glück. Fundamentals of reversible flowchart languages. *Theoretical Computer Science*, 611:87–115, 2016.

[123] T. Yokoyama and R. Glück. A reversible programming language and its invertible self-interpreter. In G. Ramalingam and E. Visser, editors, *PEPM '07: Proceedings of the 2007 ACM SIGPLAN Workshop on Partial evaluation and semantics-based program manipulation*, pages 144–153. ACM Press, 2007.

I have seen myself backward.

Philip K. Dick, *A Scanner Darkly*

A
Reversible circuit logic

This chapter contains two papers related to reversible circuit logic.

(**A1**) M. K. Thomsen, R. Kaarsgaard, and M. Soeken. Ricercar: A Language for Describing and Rewriting Reversible Circuits with Ancillae and its Permutation Semantics. In J. Krivine and J.-B. Stefani, editors, *Reversible Computation*, Lecture Notes in Computer Science volume 9138, pages 200–215, Springer Verlag, 2015.

(**A2**) H. B. Axelsen, R. Glück, and R. Kaarsgaard. A Classical Propositional Logic for Reasoning about Reversible Logic Circuits. In J. Väänänen, Å. Hirvonen, and R. de Queiroz, editors, *Logic, Language, Information, and Computation (WoLLIC)*, Lecture Notes in Computer Science volume 9803, pages 52–67, Springer Verlag, 2016.

Ricercar: A Language for Describing and Rewriting Reversible Circuits with Ancillae and Its Permutation Semantics

Michael Kirkedal Thomsen[1](✉), Robin Kaarsgaard[2], and Mathias Soeken[1]

[1] Group of Computer Architecture, University of Bremen, Bremen, Germany
{kirkedal,msoeken}@informatik.uni-bremen.de
[2] DIKU, Department of Computer Science,
University of Copenhagen, Copenhagen, Denmark
robin@di.ku.dk

Abstract. Previously, Soeken and Thomsen presented six basic semantics-preserving rules for rewriting reversible logic circuits, defined using the well-known diagrammatic notation of Feynman. While this notation is both useful and intuitive for describing reversible circuits, its shortcomings in generality complicates the specification of more sophisticated and abstract rewriting rules.

In this paper, we introduce *Ricercar*, a general textual description language for reversible logic circuits designed explicitly to support rewriting. Taking the not gate and the identity gate as primitives, this language allows circuits to be constructed using control gates, sequential composition, and ancillae, through a notion of *ancilla scope*. We show how the above-mentioned rewriting rules are defined in this language, and extend the rewriting system with five additional rules to introduce and modify ancilla scope. This treatment of ancillae addresses the limitations of the original rewriting system in rewriting circuits with ancillae in the general case.

To set Ricercar on a theoretical foundation, we also define a permutation semantics over symmetric groups and show how the operations over permutations as transposition relate to the semantics of the language.

Keywords: Reversible logic · Term rewriting · Ancillae · Circuit equivalence · Permutation

1 Introduction

In [14] two of the authors presented six elementary rules for rewriting reversible circuits using mixed-polarity multiple-control Toffoli gates. Building on this,

M.K. Thomsen—This work was partly funded by the *European Commission* under the *7th Framework Programme.*

M.K. Thomsen—A preliminary version of Ricercar was presented as work-in-progress at *6th Conference on Reversible Computation, 2014*.

J. Krivine and J.-B. Stefani (Eds.): RC 2015, LNCS 9138, pp. 200–215, 2015.
DOI: 10.1007/978-3-319-20860-2_13

more complex rules, such as moving and deletion rules, can be derived. Rewriting using such rules can be used not just to reduce the size and cost of reversible circuits, but also to analyse and explain other optimisation approaches for reversible circuits. As one example, the templates presented in [12] are all derivable from these rewriting rules.

The rewriting rules in [14] are based on the diagrammatic notation first introduced by Feynman. This notation gives a very intuitive description of reversible circuits and the presented rewriting rules inherit this benefit. However, one goal with rewriting is to provide computer aid to the design of reversible circuits, and just as intuitive as diagrammatic notation is to understand for humans, just as hard it is to model for computers. In particular, representing the more general rules poses a problem.

In this paper we introduce *Ricercar*, a description language for reversible logic circuits (Sect. 3.) inspired by work on a *reversible combinator language* [18] and the *logic of reversible structures* [11]. Its only basic atoms are the not gate and the identity gate (both with named wires) from which other circuits are constructed using control gates and sequential composition. After describing the syntax and semantics of the language, we show how to define the graphical rewriting rules of [14] as textual rewriting rules for Ricercar descriptions (Sect. 4.) To give a theoretical foundation for Ricercar, we also define a permutation semantics over symmetric groups (Sect. 2) and show how the operations over permutations as transposition relate to the semantics of the language (Sect. 3.3).

A notable feature of the language is that it directly supports ancillae. Since reversible circuit logic does not support arbitrary fan-out, ancillae are often used to store partial results from computations by means of reversible duplication. The concept of ancillae have, however, been used in many different ways, but in this work we take the most strict possible definition. By ancillae we mean a variable (or a line) that are, for all possible assignments of other defined variables, guaranteed to be unchanged over the execution of a circuit.

This definition is much more strict than what is normally characterised by temporary storage, but it *is* needed if one wants to ensure that information is leaked and, thus, the backwards semantics of the circuits can be used directly. It is, however, still very useful in both high-level programs as well as reversible circuit constructs. As an example, an n-bit binary adder of linear depth can be implemented without ancillae, but it requires the use of reversible gates that have n inputs. However, using just one ancilla line the linear depth V-shaped adder [5,20] is implemented using only gates with a constant number of inputs. Furthermore, all current designs for implementing sub-linear depth adders require a larger number of ancilla lines that is dependent on the input size [7,17,19]. Using a similar definition, the *restore* model [4] has been investigated with respect to is computational complexity limits.

In Sect. 5 we discuss the *ancilla scope* construct of Ricercar and show five basic rewriting rules for inserting and modifying ancilla wires (Sect. 5.1). This is interesting given that deciding if a wire is indeed an ancilla wire is difficult; it can generally be done using equivalence checking, which for reversible circuits has been shown to be coNP-complete [10]. Furthermore, we show how to derive more general rules (Sect. 5.2), and show a non-trivial and useful example of how these can be used to create reversible circuits with ancilla wires from ancilla-free circuits (Sect. 5.3). As a result, the proposed rewriting language can serve as a framework to formally analyse the trade-off between gate count and number of ancilla lines in reversible circuits. Such trade-offs have so far been investigated theoretically for Turing machines in *e.g.* [2,3] and experimentally for reversible circuit synthesis in [21]. We discuss further related work in Sect. 6.

The main contributions of the paper are the following:

1. An extension of the rewriting rules with rules for circuit rewriting using ancillae.
2. A textual language to describe rewriting which is more concise than the diagrammatic notation.
3. Semantics for the rewrite rules based on permutations that is useful to show soundness of the rules and to formally argue over them.

2 Symmetric Groups as a Theory of Reversible Logic

Every reversible function f computed by a reversible circuit of n input lines $x_1, \ldots, x_n$ and n output lines $y_1, \ldots, y_n$ can be represented by an element π_f of the symmetric group S_{2^n}, *i.e.*, a permutation of the set $\{0, \ldots, 2^n - 1\}$. We have $\pi_f(x) = y$ whenever $f(x_1, \ldots, x_n) = (y_1, \ldots, y_n)$, where x and y denote the natural number representations of the bits $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$, respectively. This duality has been used for reversible logic synthesis in the last decade [6,13], but has also seen use as a theoretical foundation for the analysis of reversible circuit logic [15,16].

Unlike the usual formulation of the symmetric group S_n, we will consider its elements to be permutations of the set $\{0, \ldots, n-1\}$ rather than $\{1, \ldots, n\}$. However, we will use the standard notation of writing explicit permutations using square brackets, *e.g.* $\pi = [0\,1\,3\,2]$, cycles using parentheses, *e.g.* $\pi = (2, 3)$, and π_e for the identity permutation. Under this interpretation, composition of gates corresponds to multiplication (*i.e.*, composition) of permutations.

The gate library we consider consists of only single-target gates, which are characterised by changing one circuit line based on a control function that argues over the variables of the remaining lines. Since all single-target gates are self-inverse, their respective permutations are involutions with cycle representations consisting of only transpositions and fixpoints. As pointed out in [15], all such transpositions are of the form (a, b) where the hamming distance of a and b is 1, *i.e.*, their binary expansions differ in exactly one position. We refer to the set of all such transpositions as

$$H_n = \{(a, b) \mid \nu(a \oplus b) = 1\} \tag{1}$$

where ν denotes the sideways sum. Note that each transposition (a, b) in H_n corresponds to one fully controlled Toffoli gate with positive and negative control lines, acting on line i, where i is the single index for which $a_i \neq b_i$. The polarity of the controls is chosen according to the other bits. Based on this observation, we partition the set H_n into n sets $H_{n,1}, H_{n,2}, \ldots, H_{n,n}$ such that

$$H_{n,i} = \{(a,b) \in H_n \mid a \oplus b = 2^{i-1}\} \tag{2}$$

contains all transpositions in which the components differ in their i-th bit. Single-target gates that act on the target line i are all permutations that consist of a subset of transpositions in $H_{n,i}$.

We call $g_n(f) \in S_{2^n}$ a *transposition generation function* which takes as argument an injective function $f : \{0, \ldots, 2^n - 1\} \hookrightarrow \{0, \ldots, 2^n - 1\}$ and returns the permutation

$$(0, f(0))(1, f(1)) \cdots (2^n - 1, f(2^n - 1)). \tag{3}$$

3 Ricercar: A Description Language for Reversible Logic

In this section, we will explain the description language, Ricercar, that is used to formulate the rewriting rules. We will first explain the syntax (Fig. 1) and then show two ways to describe the semantics.

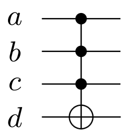
3.1 Syntax

Circuit wires (denoted by lower case Latin letters in the end of the alphabet: $\ldots, x, y, z$) are defined over a set of names Σ that includes both input/output wires and ancilla wires currently in scope. (For wires without specific names, we will use lower case Latin letters starting from a.) We define a circuit (denoted by upper case Latin letters) to be one of the following five forms:

- The identity gate on a wire x, written $\mathsf{Id}(x)$, where $x \in \Sigma$.
- The not gate applied to a wire x, written $\mathsf{Not}(x)$, where $x \in \Sigma$.
- Sequential composition of two circuits, written using the operator " ; ".

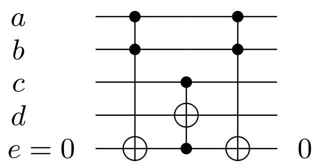
$A, B, C ::=$	$\mathsf{Id}(x) \mid \mathsf{Not}(x)$	Identity and not gate
	$\mid A \,;\, B$	Sequence of circuits
	$\mid \phi \bullet\!\!- A$	Controlled circuit
	$\mid \alpha x.A$	Scope of ancilla variable α; α is part of the syntax
$\phi, \psi, \pi ::=$	$x \mid \neg\phi \mid \phi \wedge \phi$	Boolean formulas
	$\mid \top \mid \bot \mid \phi \vee \phi \mid \phi \oplus \phi$	Derivable Boolean operators

Fig. 1. Syntax of Ricercar. Note that this grammar does not guarantee reversibility in itself. By x we mean that variables occurring in Boolean formulas must be elements from a predefined set of input/output wires or ancilla wires in scope.



$(a \wedge b \wedge c) \bullet\!\!- \mathsf{Not}(d)$

(a) Three-controlled Toffoli gate



$\alpha e.((a \wedge b) \bullet\!\!- \mathsf{Not}(e)\ ;\ (c \wedge e) \bullet\!\!- \mathsf{Not}(d)\ ;\ (a \wedge b) \bullet\!\!- \mathsf{Not}(e))$

(b) Sequence of Toffoli gates

Fig. 2. Two (weakly) equivalent reversible circuits and their descriptions in Ricercar. Here, e denotes an arbitrary ancilla wire.

$$\begin{array}{lll} \mathrm{inv}(\mathsf{Id}(x)) = \mathsf{Id}(x) & \mathrm{inv}(A\ ;\ B) = \mathrm{inv}(B)\ ;\ \mathrm{inv}(A) & \\ \mathrm{inv}(\mathsf{Not}(x)) = \mathsf{Not}(x) & \mathrm{inv}(\phi \bullet\!\!- A) = \phi \bullet\!\!- \mathrm{inv}(A) & \mathrm{inv}(\alpha x.A) = \alpha x.\,\mathrm{inv}(A) \end{array}$$

Fig. 3. The syntactic function inv(·) that defines the inverse of a Ricercar description

- A controlled circuit, denoted with the binary "$\bullet\!\!-$" operator, which contains a control function ϕ and a controlled circuit A.[1] The control function can be any Boolean formula.
- An ancilla scope for a circuit A, denoted with a functional lambda-style notation using the symbol α, and a variable denoting a wire which *must* be false both before and after A. Without loss of generality, we will assume that ancillae scopes always introduces fresh variable names.

For readability, we define control gates ($\bullet\!\!-$) to bind tighter than sequence (;) and the unary ($\alpha x.$).

Figure 2 shows two example circuits, defined using multiply controlled Toffoli gates, represented in the usual diagram notation due to Feynman, as well as in Ricercar.

As Ricercar should be reversible, we will define the straight-forward inverse of all the syntactic constructs. We have chosen *not* to include inversion as a basic construct, but will define it as a syntactic function; this simplifies both the language and the following rewriting rules. Figure 3 shows the inversion function inv(·).

3.2 Ancillae and Reversibly Well-Formed Properties

Ancillae hold a central place in Ricercar. We follow the idea that there are always as many ancilla wires available as needed. Consequently ancilla lines do not need to be declared in advance, but can be introduced on-the-fly. This is not an unrealistic assumption: remember that we define ancillae to be constant (false) at both input and output, which permits a large degree of reuse. Furthermore,

[1] The "$\bullet\!\!-$" notation is borrowed from [11], although with a different semantics.

$\mathrm{rwf}(\mathsf{Id}(x)) = \{x\}$	$\mathrm{rwf}(A\ ;\ B) = \mathrm{rwf}(A) \cup \mathrm{rwf}(B)$	$\mathrm{rwf}(\alpha x.A) = \mathrm{rwf}(A) \backslash \{x\}$
$\mathrm{rwf}(\mathsf{Not}(x)) = \{x\}$	$\mathrm{rwf}(\phi \bullet\!\!- A) = \mathrm{dom}(\phi) \uplus \mathrm{rwf}(A)$	

Fig. 4. A reversible circuit, A, is reversibly well-formed iff $\mathrm{rwf}(A)$ evaluates to a set. Here, $\uplus$ is the disjoint union and $\mathrm{dom}(\phi)$ is the domain of the Boolean function ϕ. We assume that the disjoint union is undefined whenever the operands are not disjoint.

the actual number of ancilla lines used is limited by the depth of the circuit, and cannot grow unboundedly. As an example, given that we know the upper bound of the depth to implement a reversible circuit without ancillae (*cf.* [1]), this also gives an upper bound on the number of (useful) ancilla any circuit with smaller depth can have.

The syntax presented in Fig. 1 does not guarantee reversibility by itself. One problem comes from the control gate, where we must enforce that the wires of the control function are disjoint from wires of the circuit being controlled. This is similar to the concept used in the reversible updates in Janus [22]. Figure 4 shows a function $\mathrm{rwf}(\cdot)$ that implements this check; we say that the circuit is *reversibly well-formed* if it upholds this restriction. Given a circuit description A, it returns the set of all used variable names *if and only if* A is reversibly well-formed. If a circuit A is *not* reversibly well-formed, the disjoint union operation will fail on the control gate operator, and the result of $\mathrm{rwf}(A)$ will thus be undefined.

However, even a reversibly well-formed circuit is not necessarily reversible.[2] To ensure that an ancilla variable within an ancilla scope does indeed have ancilla behaviour (guaranteed false at both input and output), we need a additional semantic check. However, a circuit *without* ancillae is reversibly well-formed if and only if it is reversible. In Sect. 5, we will show how this can be exploited to introduce ancillae in a way that guarentees reversibility.

3.3 Operational Semantics

The straightforward semantics of Ricercar is shown in Fig. 5; they follow, but also extend, the logic by Fredkin and Toffoli, and describe the mapping from a circuit description to a reversible circuit using the well-known gates. More concretely, this semantics can be used to show that Ricercar is actually reversible.

Theorem 1 (Reversibility). *For all mappings σ and circuits A there exists a mapping σ' and a circuit B such that*

$$\sigma \vdash A \rightarrow \sigma' \iff \sigma' \vdash B \rightarrow \sigma.$$

This theorem and the following two lemmas are easily proven by structural induction over the circuit A and reference to the operational semantics of Ricercar (Fig. 5).

[2] The other direction holds: all reversible circuits are reversibly well-formed.

$$\sigma : \Sigma \rightharpoonup \mathbb{B} \qquad\qquad \frac{}{\sigma \vdash \mathsf{Id}(x) \to \sigma}$$

$$\frac{\sigma \vdash \neg x \to b}{\sigma \vdash \mathsf{Not}(x) \to \sigma[x \mapsto b]} \qquad\qquad \frac{\sigma \vdash A \to \sigma'' \qquad \sigma'' \vdash B \to \sigma'}{\sigma \vdash A\ ;\ B \to \sigma'}$$

$$\frac{\sigma \vdash \phi \to 1 \qquad \sigma \vdash A \to \sigma'}{\sigma \vdash \phi \bullet\!\!- A \to \sigma'} \qquad\qquad \frac{\sigma \vdash \phi \to 0}{\sigma \vdash \phi \bullet\!\!- A \to \sigma}$$

$$\frac{\sigma \vdash x \to b \qquad \sigma[x \mapsto 0] \vdash A \to \sigma' \qquad \sigma' \vdash x \to 0}{\sigma \vdash \alpha x.A \to \sigma'[x \mapsto b]}$$

Fig. 5. The semantics of Ricercar. Here, σ is a partial function mapping variable names to Boolean values; any variable name that is not part of the input is assumed to be undefined in σ. The semantics uses two judgment forms, $\sigma \vdash A \to \sigma'$ for evaluating circuits, and $\sigma \vdash \phi \to b$ for evaluating Boolean formulae, both with respect to σ. The rules for judgments of the latter form are not shown, but are completely standard.

To ensure that the previously defined inversion (with sequence as composition function) is indeed inversion, we show the following.

Lemma 1 (Inversion). *For all circuits A and states σ,*

$$\sigma \vdash A\ ;\ \mathrm{inv}(A) \to \sigma \quad \textit{and} \quad \sigma \vdash \mathrm{inv}(A)\ ;\ A \to \sigma.$$

Later it will also be useful to know that the inversion function respects involution symmetry.

Lemma 2 (Involution Symmetry). *For all circuits A, and states σ and σ',*

$$\sigma \vdash A \to \sigma' \iff \sigma \vdash \mathrm{inv}(\mathrm{inv}(A)) \to \sigma'.$$

3.4 Permutation (Denotational) Semantics

In order to ease the formal analyses using this language, we also express the functional semantics in terms of permutations. The counterparts to Id, Not, and '$\bullet\!\!-$' are provided for this purpose. In contrast to the language, the permutation description requires an order of variables and therefore we assume a strict total order '$>$' on the variables in Σ for the following equations. If $x > y$, it means that the variable x corresponds to a more significant bit than y. For the identity and the not gate we have

$$\mathsf{Id}(x) = \pi_e \quad \text{and} \quad \mathsf{Not}(x) = (0, 1) \qquad \text{if } \Sigma = \{x\}. \tag{4}$$

For the following four equations, let $G(\pi, f)$ be the commutator $g_n(f) \circ \pi \circ g_n^{-1}(f)$ for a permutation $\pi \in S_{2^n}$ and a function f as in (3). Note that G is an endomorphism with respect to composition, since

$$\begin{aligned}G(\pi_1 \circ \pi_2, f) &= g_n(f) \circ \pi_1 \circ \pi_2 \circ g_n^{-1}(f) \\ &= g_n(f) \circ \pi_1 \circ g_n^{-1}(f) \circ g_n(f) \circ \pi_2 \circ g_n^{-1}(f) \\ &= G(\pi_1, f) \circ G(\pi_2, f).\end{aligned}$$

For some circuit A, let π_A be its permutation representation. Then one can "add a control line from the bottom," expressed as

$$\neg x \bullet\!- A = G(\pi_A, x \mapsto x) \quad \begin{array}{l}\text{if } \Sigma = \mathrm{rwf}(A) \cup \{x\} \\ \text{and } x > y \text{ for all } y \in \mathrm{rwf}(A)\end{array} \tag{5}$$

and

$$x \bullet\!- A = G(\pi_A, x \mapsto x + 2^n) \quad \begin{array}{l}\text{with } n = |\,\mathrm{rwf}(A)|, \text{ if } \Sigma = \mathrm{rwf}(A) \cup \{x\} \\ \text{and } x > y \text{ for all } y \in \mathrm{rwf}(A).\end{array} \tag{6}$$

Similarly, one can "add a control line from the top," expressed as

$$\neg x \bullet\!- A = G(\pi_A, x \mapsto 2x) \quad \begin{array}{l}\text{if } \Sigma = \mathrm{rwf}(A) \cup \{x\} \\ \text{and } x < y \text{ for all } y \in \mathrm{rwf}(A)\end{array} \tag{7}$$

and

$$x \bullet\!- A = G(\pi_A, x \mapsto 2x + 1) \quad \begin{array}{l}\text{if } \Sigma = \mathrm{rwf}(A) \cup \{x\} \\ \text{and } x < y \text{ for all } y \in \mathrm{rwf}(A).\end{array} \tag{8}$$

The above denotational semantics is not complete. Circuit sequence (;) can be defined by permutation composition after extending the two permutations to the same symmetric group, and scoped ancillae can be accommodated by imposing restrictions on the permutation for the more general circuit (*i.e.*, where the ancilla is considered as any other input line.) It is then not hard to prove equivalence between the operational and denotational semantics. The denotational semantics is reversible by construction.

4 Rewriting in Ricercar

In this section, we will recap the rewriting rules from [14], and define the rules with respect to Ricercar, as well as show soundness based on the permutation semantics.

First, however, note that gate composition is associative; that is, in a cascade of gates, the order in which we look at the gates does not matter, so in, *e.g.* Fig. 2(b), we are free to either look at the two first gates and perform rewriting on these, or start with the last two gates instead. The identity gate is the identity element for sequences:

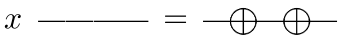
$$A = \mathsf{Id}(x) \;;\; A = A \;;\; \mathsf{Id}(x) \tag{ID}$$

Furthermore, note that we can always rewrite the controlling Boolean functions and, *e.g.* use identities from AND-EXOR decomposition:

$$\phi \bullet\!\!- \psi \bullet\!\!- A = (\phi \wedge \psi) \bullet\!\!- A \qquad \text{and}$$
$$\phi \bullet\!\!- A \; ; \; \psi \bullet\!\!- A = (\phi \oplus \psi) \bullet\!\!- A \qquad \text{if } A = \mathrm{inv}(A).$$

Finally, implicit to rules is that the circuits must always be reversibly well-formed both before and after a rewriting, and that in any given circuit we can rewrite any sub-circuit we like.

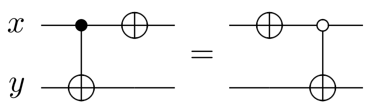
The first rule presented in [14] is for introducing and eliminating not gates, and states that we can always rewrite the identity function to two not gates.



$$\mathsf{Id}(x) = \mathsf{Not}(x) \; ; \; \mathsf{Not}(x) \tag{R1}$$

Soundness trivially follows from $\pi_e = (0,1) \circ (0,1)$.

The second rule states that we can "move" a not gate over a control if we negate the control line.



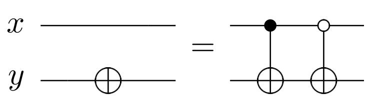
$$x \bullet\!\!- \mathsf{Not}(y) \; ; \; \mathsf{Not}(x) = \mathsf{Not}(x) \; ; \; \neg x \bullet\!\!- \mathsf{Not}(y) \tag{R2}$$

Similar to [14], we notice that its dual rule with negative control can be derived using this rule in combination with Rule R1:

$$\begin{aligned} \neg x \bullet\!\!- \mathsf{Not}(y) \; ; \; \mathsf{Not}(x) &\overset{(R1)}{=} \mathsf{Not}(x) \; ; \; \mathsf{Not}(x) \; ; \; \neg x \bullet\!\!- \mathsf{Not}(y) \; ; \; \mathsf{Not}(x) \\ &\overset{(R2)}{=} \mathsf{Not}(x) \; ; \; x \bullet\!\!- \mathsf{Not}(y) \; ; \; \mathsf{Not}(x) \; ; \; \mathsf{Not}(x) \\ &\overset{(R1)}{=} \mathsf{Not}(x) \; ; \; x \bullet\!\!- \mathsf{Not}(y). \end{aligned} \tag{R2'}$$

Soundness follows from Eqs. (5)–(8) and the identity $(a,b)(b,c) = (a,c)(a,b) = (a,c)(b,c)$.

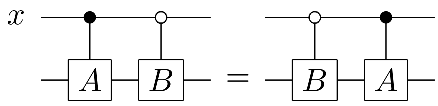
Third, we can extend a gate by copying it and adding once a positive and once a negative control line to it.



$$\mathsf{Id}(x) \; ; \; \mathsf{Not}(y) = x \bullet\!\!- \mathsf{Not}(y) \; ; \; \neg x \bullet\!\!- \mathsf{Not}(y) \tag{R3}$$

Soundness follows from Eqs. (7) and (8). In fact, in permutation notation, both controlled not gates are represented by a single transposition, and combining them results in the (permutation corresponding to the) not gate. Also, combining the equation of adding a negative and positive control yields an equation for adding an empty line.

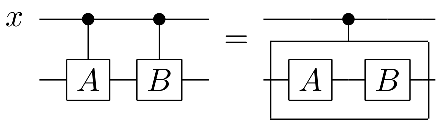
Next, two arbitrary adjacent gates can be interchanged whenever they have a common control line with different polarities. Notice how Ricercar captures the fact that two controlled circuits can have *any* circuit structure; something that is not well captured by the diagrammatic notation in [14].



$$x \bullet\!\!- A \;;\; \neg x \bullet\!\!- B = \neg x \bullet\!\!- B \;;\; x \bullet\!\!- A \qquad \text{(R4)}$$

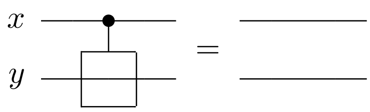
The permutation equations also reveal this property, since the transpositions resulting from $\neg x \bullet\!\!- A$ and $x \bullet\!\!- A$ are disjoint.

Whenever two gates share the same control variable with the same polarity, these two gates can be grouped together, where the group is controlled by that control line. Again, Ricercar allows for a precise formulation of the idea, compared to the diagrammatic notation.



$$x \bullet\!\!- A \;;\; x \bullet\!\!- B = x \bullet\!\!- (A \;;\; B) \qquad \text{(R5)}$$

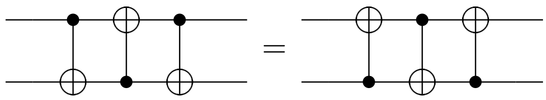
This property follows from G being an endomorphism. Finally, we have the rule for introducing and eliminating groups of wires.



$$x \bullet\!\!- \mathsf{Id}(y) = \mathsf{Id}(x) \;;\; \mathsf{Id}(y) \qquad \text{(R6)}$$

4.1 A Note on Completeness

A question raised in [14] regards the *completeness* of the above rules, in the sense that every circuit can be rewritten, in a finite number of steps, to any other equivalent circuit. In this strict sense, the rules are *not* complete. The counter example is the two-line swap gate, which can be represented in the following two ways:



Given the six rules, it is not possible to rewrite one to the other. This is, of course, not satisfactory, and a shortcoming that must be solved. The easy solution would be to add the above equation as a seventh rule, but the extent to which there exist other counter examples related to this problem is unknown, and the solution is only an incremental extension that will not add any interesting new insights.

However, this counter example is restricted in that it does not generalise to more lines. If we have a third line available (no matter its value), it can be used as an auxiliary line and thereby enable rewriting between the two swap gates. The question is now if the six rules are complete for circuits of more than two lines. But there is a possibility that two similar three line circuits exist and we need to assume a fourth auxiliary line to rewrite between them. The better solution, that we will follow in the next section, is thus to extend with rules for ancilla lines.

5 Ancillae and Rewriting

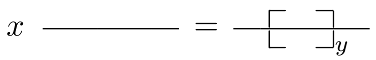
As mentioned earlier, ancillae is a powerful extension to a reversible language, but the power comes at a cost. Checking that some defined ancillae are indeed unchanged for all possible input vectors of an arbitrary description is hard; in general, one has to test all possible input vectors, which is undesirable. For a reversible programming language such as Janus [22], this is therefore implemented as a runtime check that checks the reversibility of a program *only* in relation to the executing input vector. This is also the case for the syntax described in Fig. 1.

For this reason, we will pursue a different approach. Given a description without ancillae, we can statically check reversibility using the rwf-function shown in Fig. 4. From a reversible description without ancillae, we will now define rewriting rules that can extend the given description with ancilla wires. Hence, instead of showing reversibility of a description with ancillae (which is hard), we only have to show that the rewriting rules do not interfere with the ancilla-property of the wires, and thereby with the reversibility of the circuit; this is much easier.

5.1 The Rewriting Rules

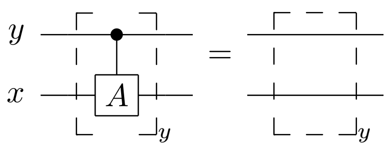
To be able to introduce and remove ancilla wires from a circuit, we have identified the need for five basic rules.

The first rule is for introducing and removing an ancilla scope. It states that we can always introduce a scope containing the identity circuit with a fresh (unused) ancilla wire name.



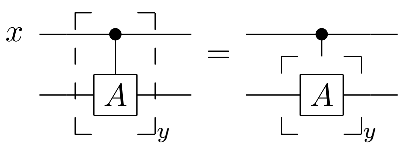
$$\mathsf{Id}(x) = \alpha y.\mathsf{Id}(x) \tag{A1}$$

The second rule states that a circuit in an ancilla scope can be removed (or added) if it is controlled by the ancilla wire. Recall that the ancilla variable is assumed to be assigned false outside of the ancilla scope, so the control is never active. For now, the gate must be the only gate within the scope, but we will show how this can be generalised later.



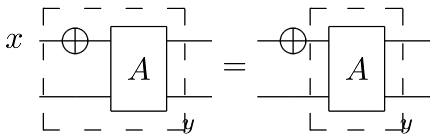
$$\alpha y.(y \bullet\!\!- A) = \alpha y.\mathsf{Id}(x), \qquad x \in \mathrm{rwf}(A) \tag{A2}$$

The third rule considers the case in which the controlling wire is not the ancilla wire of the scope. In this case, the control can be pulled out of the ancilla scope, and thereby control the scope containing the controlled circuit.



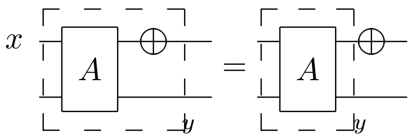
$$\alpha y.(x \bullet\!\!- A) = x \bullet\!\!- \alpha y.A, \qquad x \neq y \tag{A3}$$

The fourth rules states that a not gate on a non-ancilla wire that is positioned to the immediate left of a circuit it shares a scope with can be pulled out of the ancilla scope.



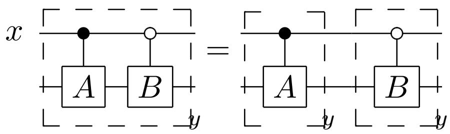
$$\alpha y.(\mathsf{Not}(x)\ ;\ A) = \mathsf{Not}(x)\ ;\ \alpha y.A, \qquad x \neq y \tag{A4}$$

In the case where the not gate is on the right, a similar rule can be derived from the Involution Symmetry Lemma with Rule A4.



$$\alpha y.(A\ ;\ \mathsf{Not}(x)) = (\alpha y.A)\ ;\ \mathsf{Not}(x), \qquad x \neq y \tag{A4'}$$

The fifth and final rule states that if (and only if) an ancilla scope contains a sequence of two circuits where the first is positively controlled, and the second is negatively controlled by the same wire, then this scope can be divided into two; or, in the other direction, merged. Note that x *can* be equal to y. This rule is likely the most powerful of the five, and it shows up in the proofs that extend and generalise the previous rules.



$$\alpha y.(x \bullet\!\!- A\ ;\ \neg x \bullet\!\!- B) = (\alpha y.x \bullet\!\!- A)\ ;\ (\alpha y.\neg x \bullet\!\!- B) \tag{A5}$$

That the first four rules (A1 to A4) do not interfere with the ancilla-property of a wire is clear, but the last rule (A5) requires an argument. Only either A or B (but not both) is performed as the control on x is exclusive. Thus assuming that $x \neq y$, any usage of y in A must have uncomputed y to zero again; similarly any usage of y in B must have assumed it to be zero. Therefore, we can divide the ancilla scope of y. If $x = y$ then y will always be unchanged (zero) as y is not used in B.

5.2 Generalisation of Ancilla Rules

Rule A2 has a twin-rule for the case where the gate is negatively controlled by the ancilla wire. We can derive that this is equal to the controlled gate in the following way:

$$\begin{aligned}
\alpha y.(\neg y \bullet\!\!- A) &\overset{(\mathrm{ID})}{=} \mathsf{Id}(x)\ ;\ (\alpha y.\neg y \bullet\!\!- A) && \overset{(\mathrm{A1})}{=} (\alpha y.\mathsf{Id}(x))\ ;\ (\alpha y.\neg y \bullet\!\!- A) \\
&\overset{(\mathrm{A2})}{=} (\alpha y.y \bullet\!\!- A)\ ;\ (\alpha y.\neg y \bullet\!\!- A) && \overset{(\mathrm{A5})}{=} \alpha y.(y \bullet\!\!- A\ ;\ \neg y \bullet\!\!- A) \\
&\overset{(\mathrm{R3})}{=} \alpha y.A.
\end{aligned} \tag{A2'}$$

This twin-rule can now be used to show the more general rule that if an ancilla wire controls a circuit in the beginning of a scope it can be removed entirely:

$$\begin{aligned}
\alpha y.(y \bullet\!\!- A \,;\, B) &\overset{(\text{R3})}{=} \alpha y.(y \bullet\!\!- A \,;\, y \bullet\!\!- B \,;\, \neg y \bullet\!\!- B) \\
&\overset{(\text{R5})}{=} \alpha y.(y \bullet\!\!- (A \,;\, B) \,;\, \neg \alpha \bullet\!\!- .B) \\
&\overset{(\text{A5})}{=} (\alpha y.y \bullet\!\!- (A \,;\, B)) \,;\, (\alpha y.\neg y \bullet\!\!- B) \\
&\overset{(\text{A2})}{=} (\alpha y.\mathsf{Id}(x)) \,;\, (\alpha y.\neg y \bullet\!\!- B) \quad \overset{(\text{A2'})}{=} \quad (\alpha y.\mathsf{Id}(x)) \,;\, (\alpha y.B) \\
&\overset{(\text{A1})}{=} \mathsf{Id}(x) \,;\, \alpha y.B \qquad\qquad\qquad \overset{(\text{ID})}{=} \quad \alpha y.B.
\end{aligned} \tag{D8}$$

Similarly, we can also generalise A3 to the case where the circuit in the ancilla scope contains more than one gate. Assuming that $x \neq y$, to extract x from the ancilla scope of y we can do

$$\begin{aligned}
\alpha y.x \bullet\!\!- A \,;\, B &\overset{(\text{R3})}{=} \alpha y.(x \bullet\!\!- A \,;\, x \bullet\!\!- B \,;\, \neg x \bullet\!\!- B) \\
&\overset{(\text{R5})}{=} \alpha y.(x \bullet\!\!- (A \,;\, B) \,;\, \neg x \bullet\!\!- B) \\
&\overset{(\text{A5})}{=} (\alpha y.x \bullet\!\!- (A \,;\, B)) \,;\, (\alpha y.\neg x \bullet\!\!- B) \\
&\overset{(\text{A3})}{=} x \bullet\!\!- (\alpha y.(A \,;\, B)) \,;\, \neg x \bullet\!\!- \alpha y.B.
\end{aligned} \tag{D9}$$

This duplicates B such that it is performed both when x is true, and when it is false. Assuming that the ancilla wire y does not occur in A (*i.e.* $y \notin \mathrm{rwf}(A)$), we can then use D9 to show by induction on the depth of the control that

$$\alpha y.(A \,;\, B) = A \,;\, \alpha y.B, \qquad y \notin \mathrm{rwf}(A). \tag{D10}$$

As a special case of this rule, specifically when $B = \mathsf{Id}(x)$ for any choice of $x \in \mathrm{rwf}(A) \cup \{y\}$, we get that

$$\alpha y.A = A, \qquad y \notin \mathrm{rwf}(A). \tag{D11}$$

As a closing derived rule, we will show how ancilla wires can be introduced to perform computations that were otherwise performed by an input wire. In other words, we can use the rules to introduce ancilla wires that are then used to control what was previously controlled by x.

$$\begin{aligned}
x \bullet\!\!- A &\overset{(\text{D11})}{=} \alpha y.(x \bullet\!\!- A \,;\, \mathsf{Id}(y)) \\
&\overset{(\text{D1})}{=} \alpha y.(x \bullet\!\!- A \,;\, x \bullet\!\!- y \,;\, x \bullet\!\!- y) \\
&\overset{(\text{D8})}{=} \alpha y.(y \bullet\!\!- A \,;\, x \bullet\!\!- A \,;\, x \bullet\!\!- y \,;\, x \bullet\!\!- y) \\
&\overset{(\text{D6})}{=} \alpha y.(x \bullet\!\!- A \,;\, y \bullet\!\!- A \,;\, x \bullet\!\!- y \,;\, x \bullet\!\!- y) \\
&\overset{(\text{D7})}{=} \alpha y.(x \bullet\!\!- y \,;\, y \bullet\!\!- A \,;\, x \bullet\!\!- y).
\end{aligned} \tag{D12}$$

Here D1, D6, and D7 refer to derived rules from [14]. This example increases the size and depth of the circuit, but if x controls several gates this can be used to reduce the depth of the circuit considering that gates can be put in parallel.

5.3 Practical Example of Application of Ricercar

As a final example we show how to derive the circuit depicted in Fig. 2(b) from the one in Fig. 2(a) using the rewriting rules. Again D1 and D7 refer to derived rules from [14].

$$
\begin{aligned}
&(a \wedge b \wedge c) \bullet\!\!- \mathsf{Not}(d) \\
&\quad \overset{\text{(D11)}}{=} \alpha e.((a \wedge b \wedge c) \bullet\!\!- \mathsf{Not}(d)) \\
&\quad \overset{\text{(D8)}}{=} \alpha e.((c \wedge \beta) \bullet\!\!- \mathsf{Not}(d) \,;\, (a \wedge b \wedge c) \bullet\!\!- \mathsf{Not}(d)) \\
&\quad \overset{\text{(D1)}}{=} \alpha e.((a \wedge b) \bullet\!\!- \mathsf{Not}(e) \,;\, (a \wedge b) \bullet\!\!- \mathsf{Not}(e) \,;\, (c \wedge e) \bullet\!\!- \mathsf{Not}(d) \,; \\
&\quad\qquad (a \wedge b \wedge c) \bullet\!\!- \mathsf{Not}(d)) \\
&\quad \overset{\text{(D7)}}{=} \alpha e.((a \wedge b) \bullet\!\!- \mathsf{Not}(e) \,;\, (c \wedge e) \bullet\!\!- \mathsf{Not}(d) \,;\, (a \wedge b) \bullet\!\!- \mathsf{Not}(e) \,; \\
&\quad\qquad (a \wedge b \wedge c) \bullet\!\!- \mathsf{Not}(d) \,;\, (a \wedge b \wedge c) \bullet\!\!- \mathsf{Not}(d)) \\
&\quad \overset{\text{(D1)}}{=} \alpha e.((a \wedge b) \bullet\!\!- \mathsf{Not}(e) \,;\, (c \wedge e) \bullet\!\!- \mathsf{Not}(d) \,;\, (a \wedge b) \bullet\!\!- \mathsf{Not}(e)).
\end{aligned}
$$

6 Related Work

This is not the first language that has been designed to describe the concepts of reversible logic; there exist description languages for both reversible and quantum circuits.

The closest related work is the *Reversible Combinator Language* (RCL) [18] that was also made to describe reversible logic; though it is more general than our work, there are still some common ideas. Taking inspiration from RCL, we use a similar sequence operator, and the conditional in RCL is (in its semantics) comparable to our control operator. However, being a combinator language, RCL does not have variables, but rather a type system in which circuits of arbitrary size with a given structure can be defined. Also it has more general combinators, such as a ripple circuit and parallel composition, as basic constructs. RCL also admits a number of rewriting rules, but compared to Ricercar, RCL's type system and larger set of atomic gates makes rewriting more cumbersome.

Although aiming to describe quantum circuits, it is worth mentioning *Quipper* [8,9]. Though Quipper also supports ancilla scopes, in order to uphold the ancilla-property, the Quipper synthesis results in a symmetric compute-use-uncompute "Bennett-style" structure of the ancilla wires. In contrast, the ancilla scopes in Ricercar are more general, but have to be built from the bottom up with rewriting to uphold the property. The interested reader can find further references for quantum description languages in the works above.

7 Conclusion

In this paper we have presented *Ricercar*, a language designed to describe reversible circuits. A main focus during the design process of the language has been rewriting, specifically that rewriting rules should be easy to both define and apply in the language. The previous approach to rewriting of reversible circuits was shown for the standard diagrammatic notation, but this notation neither captures the full intent of all of the six original rules, nor does it provide an optimal setting for a future computer aided system. Ricercar, with its simple symbolic description, both captures the complete intent of the original rules, and has a syntax that is directly implementable.

In addition, Ricercar has support for ancillae as a basic circuit construct in the form of a *scope*. Using this construct, we have extended the six original rules with five basic rules that applies when rewriting ancillae. We have shown how it is possible to use these rules to derive more general ones that also apply to ancillae, and as a final example, how to derive a rule that moves the control of a gate from an input wire to an ancilla wire.

Determining reversibility of a circuit that contains ancillae is generally hard, but with the presented rewriting rules, it is possible to take an ancillae-free circuit (for which it is easy to show reversibility) and rewrite it into a circuit that contains ancillae, and is guaranteed to be reversible. The key here is that the basic rules (and all of the derived rules) cannot break the ancilla-property of a wire and, thus, the reversibility of the circuit.

We hope that this approach can further help in the understanding of the trade-off between ancillae on the one hand, and the size and depth of a reversible circuit on the other.

Acknowledgments. The authors thank the anonymous reviewers from RC'14 for their useful comments on a preliminary version of this paper, presented as work-in-progress at *6th Conference on Reversible Computation, 2014* and the anonymous reviewers from RC'15 on the current version. Finally, we thank Holger Bock Axelsen for several discussion on work in this paper.

References

1. Abdessaied, N., Soeken, M., Thomsen, M.K., Drechsler, R.: Upper bounds for reversible circuits based on Young subgroups. Information Processing Letters **114**(6), 282–286 (2014)
2. Bennett, C.H.: Time/Space Trade-Offs for reversible computation. SIAM Journal on Computing **18**(4), 766–776 (1989)
3. Buhrman, H., Tromp, J., Vitányi, P.: Time and space bounds for reversible simulation. Journal of Physics A: Mathematical and General **34**(35), 6821–6830 (2001)
4. Chan, T., Munro, J.I., Raman, V.: Selection and sorting in the "restore" model. In: Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 995–1004 (2014)
5. Cuccaro, S.A., Draper, T.G., Kutin, S.A., Moulton, D.P.: A new quantum ripple-carry addition circuit. arXiv:quant-ph/0410184v1 (2005)
6. De Vos, A., Rentergem, Y.V.: Reversible computing: from mathematical group theory to electronical circuit experiment. In: Proceedings of the Second Conference on Computing Frontiers, 2005, Ischia, Italy, May 4–6, 2005, pp. 35–44 (2005)

7. Draper, T.G., Kutin, S.A., Rains, E.M., Svore, K.M.: A logarithmic-depth quantum carry-lookahead adder. arXiv:quant-ph/0406142 (2008)
8. Green, A.S., Lumsdaine, P.L.F., Ross, N.J., Selinger, P., Valiron, B.: An introduction to quantum programming in quipper. In: Dueck, G.W., Miller, D.M. (eds.) RC 2013. LNCS, vol. 7948, pp. 110–124. Springer, Heidelberg (2013)
9. Green, A.S., Lumsdaine, P.L., Ross, N.J., Selinger, P., Valiron, B.: Quipper: a scalable quantum programming language. In: Conference on Programming Language Design and Implementation, PLDI 2013, pp. 333–342. ACM (2013)
10. Jordan, S.P.: Strong equivalence of reversible circuits is coNP-complete. Quantum Information & Computation **14**(15–16), 1302–1307 (2014)
11. Kaarsgaard, R.: Towards a propositional logic for reversible logic circuits. In: de Haan, R. (ed.) Proceedings of the ESSLLI 2014 Student Session, pp. 33–41 (2014). http://www.kr.tuwien.ac.at/drm/dehaan/stus2014/proceedings.pdf
12. Miller, D.M., Maslov, D., Dueck, G.W.: A transformation based algorithm for reversible logic synthesis. In: Design Automation Conference, DAC, pp. 318–323 (2003)
13. Shende, V.V., Prasad, A.K., Markov, I.L., Hayes, J.P.: Synthesis of reversible logic circuits. IEEE Trans. on CAD of Integrated Circuits and Systems **22**(6), 710–722 (2003)
14. Soeken, M., Thomsen, M.K.: White dots *do* matter: rewriting reversible logic circuits. In: Dueck, G.W., Miller, D.M. (eds.) RC 2013. LNCS, vol. 7948, pp. 196–208. Springer, Heidelberg (2013)
15. Soeken, M., Thomsen, M.K., Dueck, G.W., Miller, D.M.: Self-inverse functions and palindromic circuits. arXiv 1502.05825 (2015)
16. Storme, L., De Vos, A., Jacobs, G.: Group theoretical aspects of reversible logic gates. J. UCS **5**(5), 307–321 (1999)
17. Takahashi, Y., Kunihiro, N.: A fast quantum circuit for addition with few qubits. Quantum Info. Comput. **8**(6), 636–649 (2008)
18. Thomsen, M.K.: Describing and optimising reversible logic using a functional language. In: Gill, A., Hage, J. (eds.) IFL 2011. LNCS, vol. 7257, pp. 148–163. Springer, Heidelberg (2012)
19. Thomsen, M.K., Axelsen, H.B.: Parallelization of reversible ripple-carry adders. Parallel Processing Letters **19**(1), 205–222 (2009)
20. Vedral, V., Barenco, A., Ekert, A.: Quantum networks for elementary arithmetic operations. Physical Review A **54**(1), 147–153 (1996)
21. Wille, R., Soeken, M., Miller, D.M., Drechsler, R.: Trading off circuit lines and gate costs in the synthesis of reversible logic. Integration, the VLSI Journal **47**(2), 284–294 (2014)
22. Yokoyama, T., Axelsen, H.B., Glück, R.: Principles of a reversible programming language. In: Conference on Computing Frontiers, CF, pp. 43–54. ACM Press (2008)

A Classical Propositional Logic for Reasoning About Reversible Logic Circuits

Holger Bock Axelsen, Robert Glück, and Robin Kaarsgaard(✉)

DIKU, Department of Computer Science,
University of Copenhagen, Copenhagen, Denmark
{funkstar,glueck,robin}@di.ku.dk

Abstract. We propose a syntactic representation of reversible logic circuits in their entirety, based on Feynman's control interpretation of Toffoli's reversible gate set. A pair of interacting proof calculi for reasoning about these circuits is presented, based on classical propositional logic and monoidal structure, and a natural order-theoretic structure is developed, demonstrated equivalent to Boolean algebras, and extended categorically to form a sound and complete semantics for this system. We show that all strong equivalences of reversible logic circuits are provable in the system, derive an equivalent equational theory, and describe its main applications in the verification of both reversible circuits and template-based reversible circuit rewriting systems.

1 Introduction

Reversible computing–the study of computing models deterministic in both the forward and backward directions–is primarily motivated by a potential to reduce the power consumption of computing processes, but has also seen applications in topics such as static average-case program analysis [17], unified descriptions of parsers and pretty-printers [16], and quantum computing [6]. The potential energy reduction was first theorized by Rolf Landauer in the early 1960s [12], and has more recently seen experimental verification [2,14]. Reaping these potential benefits in energy consumption, however, requires the use of a specialized gate set guaranteeing reversibility, when applied at the level of logic circuits.

Boolean logic circuits correspond immediately to propositions in classical propositional logic (CPL): This is done by identifying input lines with propositional atoms, and logic gates with propositional connectives, reducing the problem of reasoning about circuits to that of reasoning about arbitrary propositions in a classical setting. However, although Toffoli's gate set for reversible circuit logic is equivalent to the Boolean one in terms of what can be computed [22], it falls short of this immediate and pleasant correspondence. This article seeks

The authors acknowledge support from the Danish Council for Independent Research | Natural Sciences under the *Foundations of Reversible Computing* project, and partial support from COST Action IC1405 *Reversible Computation*.

Colors in electronic version.

J. Väänänen et al. (Eds.): WoLLIC 2016, LNCS 9803, pp. 52–67, 2016.
DOI: 10.1007/978-3-662-52921-8_4

to establish such a correspondence by proposing a standardized way of syntactically representing and reasoning about reversible logic circuits. This is done by considering a reformulation, and slight extension, of the toolset of classical propositional logic. The main contributions of this article are the following:

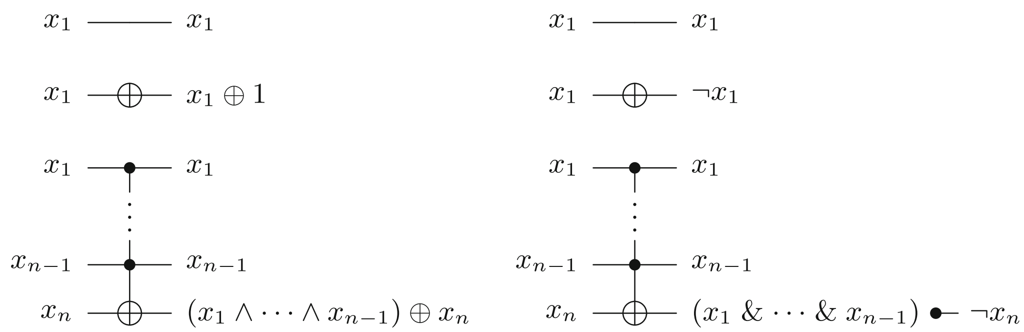
– A syntactic representation of entire reversible logic circuits as propositions, and a pair of proof calculi for reasoning about the semantics of thusly represented reversible logic circuits, sound and complete with respect to
– a categorical/algebraic semantics based on the free strict monoidal category over a *Toffoli lattice*, an order structure proven *equivalent to Boolean rings*,
– a proof that all strong equivalences of reversible logic circuits are provable, and
– an illustration of how the presented logic can be used to show strong equivalences of reversible circuits, and in particular to verify template-based reversible logic circuit rewriting systems.

The complexity of reversible circuits has been increasing while at the same time entirely new functional designs have been found (*e.g.* linear transforms [5], reversible microprocessors [21]). Established tools employing conventional Boolean logic are not geared towards the synthesis, transformation and verification of reversible circuits. Thus, it is important to find better ways of handling this new type of circuits, and some work has been approaching these problems from different angles (*e.g.* [4,23]). Our goal is to formally model the semantics of reversible circuits, and in particular to capture strong equivalence of such circuits as provable equivalence of propositions.

Overview: Sect. 2 introduces the syntax and intuitive interpretation of the connectives, and shows how reversible logic circuits can be represented as propositions by way of a simple annotation algorithm. Section 3 describes the proof calculi used to reason about circuits thus represented, and relates them to existing systems. Section 4 develops the concept of a *Toffoli lattice* as a semantics for the central proof calculus and extends it, via a categorical view on such a structure, into the final model category $\mathfrak{T}_\otimes$. Section 5 sketches the fundamental metatheorems of soundness, completeness and circuit completeness, Sect. 6 outlines the applications of the developed theory in reversible circuit rewriting, and Sect. 7 presents ideas for future work, and concludes on the results presented.

2 Circuits as Propositions

The correspondence between Boolean circuits and propositions, in all of its convenience to areas such as circuit design and computational complexity, did not happen by mistake: It is a well-known result that any Boolean function can be computed by a circuit composed of only NAND gates and constants, yet the Boolean gate set is still, in all of its redundancies, considered the *lingua franca* of logic circuit design, precisely due to this correspondence.



(a) Boolean ring semantics. (b) Propositional semantics.

Fig. 1. Toffoli's reversible gate set–consisting of, from top to bottom, the IDENTITY gate, the NOT gate, and the generalized TOFFOLI gate–annotated with their Boolean ring semantics, as well as our propositional semantics.

x_1	x_2	x_3		x_1	x_2	x_3
0	0	0		0	0	0
0	0	1		0	0	1
0	1	0		0	1	0
0	1	1	$\mapsto$	0	1	1
1	0	0		1	0	0
1	0	1		1	0	1
1	1	0		1	1	1
1	1	1		1	1	0

Reversible circuits are usually depicted as gate networks where computation flows from left to right. Here, we consider circuits composed of the gates in Toffoli's reversible gate set, shown in Fig. 1a. (This widely used gate set is known as the *Multiple-Control Toffoli* (MCT) library.) We provide a brief exposition, which the reader familiar with reversible circuit logic can safely skip.

The only gate that warrants particular explanation is the generalized TOFFOLI gate, since the remaining gates behave exactly as they do in Boolean circuit logic: This gate takes $n > 1$ input lines, of which $n - 1$ are *control lines* (marked with black dots), and the remaining one is the *target line* (marked with $\oplus$). If all control lines carry a value of 1, the value on the target line is negated – if not, the input of the target line simply passes through unchanged. As such, *the control lines control whether the* NOT *operation should be carried out on the target line*; in either case, the inputs to all control lines are carried through to the output unchanged (see also the truth table to the right for the generalized Toffoli gate where $n = 3$; x_1 and x_2 are control lines, x_3 is the target line). Circuits may be composed horizontally (*i.e.*, by ordinary function composition) and vertically (*i.e.*, by computation in parallel) so long as they remain finite in size and contain neither loops, fan-in, nor fan-out. Note also that even though the target line is placed at the bottom in Fig. 1a for purposes of illustration, it may be placed anywhere relative to the control lines.

Contrary to Toffoli's Boolean ring semantics for the gate set [22], our presentation embraces Feynman's control interpretation [6] not just in the intuitive explanation given above, but also directly in the formalism. Following Kaarsgaard [11], this is done by replacing exclusive disjunction (here, $\cdot \oplus \cdot$) with the connective $\cdot \bullet\!\!- \cdot$, read as *control*, and introducing the usual negation connective $\neg\cdot$ on the target. This results in the propositional semantics shown in Fig. 1b. In this case, the semantics of the target line for the generalized TOFFOLI gate

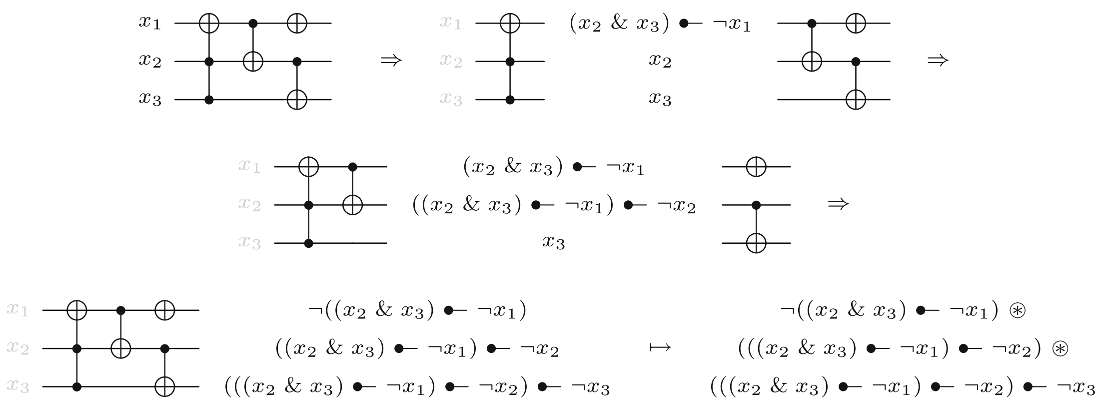


Fig. 2. An example of the annotation algorithm.

pleasingly reads as "x_1 and $\cdots$ and x_{n-1} control not x_n". While Soeken and Thomsen [20] have shown (with their box rules) that control is a general concept corresponding (roughly) to conditional execution of a subcircuit, it turns out that, at the level of individual circuit lines, control carries the same meaning as material bi-implication in CPL. We postpone the proof of equivalence of these two approaches to Sect. 5.

Although the target line of the generalized TOFFOLI gate is, in many ways, the heart of this gate's semantics, it only paints part of the picture. Since reversible circuits are, by definition, required to have the same number of output lines as input lines, parallelism plays a much larger role in reversible circuits than in Boolean ones: To capture the semantics of reversible logic circuits in their entirety, we need a way to capture this parallelism. We do this by introducing yet another connective, $\cdot \circledast \cdot$, read as *while*, with the meaning of $A \circledast B$ as the *multiplicative ordered conjunction* of propositions A and B, *i.e.* as string concatenation in a free monoid. Order is important: as stated earlier, we wish the provable equivalence relation to capture *strong equivalence* of reversible circuits (reversible circuits are strongly equivalent if they compute the same function up to function extensionality [8]), rather than equivalence up to arbitrary permutation of output lines.

Using these two new connectives, along with the usual connectives for conjunction (here, $\cdot \,\&\, \cdot$) and negation, we can produce a straightforward annotation algorithm for extracting the semantics of reversible logic circuits as a proposition in this syntax. As also done for Boolean circuits, we identify each input line with a (fresh) propositional atom, and then propagate the semantics (as given in Fig. 1b) through until the entire circuit has been annotated, at which point we terminate and return the multiplicative ordered conjunction of these propositions, from top to bottom. An example of the annotation algorithm can be found in Fig. 2.

As also noted by Kaarsgaard [11], the syntax of propositions for forming reversible logic circuits using Toffoli's gate set in Fig. 1b is more restrictive than, *e.g.*, CPL; that is, (ordinary) conjunctions only appear as subpropositions of controls. Further, linear ordered conjunctions only appear as a way of "glueing" the propositions of individual circuit lines together (see, *e.g.*, the final step in Fig. 2).

This structure suggests a syntactic hierarchy, which we will illustrate by means of color: Blue propositions will be those that correspond to the semantics of a single circuit line (perhaps of many in a circuit), red propositions correspond to the semantics of entire circuits (or subcircuits), and yellow (or recolorable) propositions will be those that can be either of these two. Formally, we define such propositions to be those produced by the grammars

$$
\begin{aligned}
A_B, B_B, C_B &:= A_Y \mid \neg A_B \mid A_B \mathbin{\&} B_B && \text{(Blue propositions)}\\
A_Y, B_Y, C_Y &:= a \mid \mathbf{0} \mid \mathbf{1} \mid \neg A_Y \mid A_B \bullet\!\!- B_Y \mid A_Y \bullet\!\!- B_B && \text{(Yellow propositions)}\\
A_R, B_R, C_R &:= A_Y \mid A_R \circledast B_R \mid \mathbf{e} && \text{(Red propositions)}
\end{aligned}
$$

where a denotes any propositional atom; we will assume that there is a denumerable set P of these. For readability, we adopt the convention that $\neg\cdot$ binds tighter than $\cdot \mathbin{\&} \cdot$, which binds tighter than $\cdot \bullet\!\!- \cdot$, which finally binds tighter than $\cdot \circledast \cdot$. Further, we will omit subscripts when the syntactic class is clear from the context.

Starting with blue and recolorable propositions, $\mathbf{0}$ and $\mathbf{1}$ represent the false respectively true proposition (corresponding to *ancillae*, lines of constant value, in circuit terms), $\neg A_Y$ the usual negation of a proposition, $A_B \bullet\!\!- B_Y$ and $A_Y \bullet\!\!- B_B$ as "A *control* B", and finally $A_B \mathbin{\&} B_B$ as the usual (additive) conjunction. Red propositions are interpreted as circuit structures, with $A_R \circledast B_R$ representing the ordered (parallel) structure made up of A_R and B_R, and $\mathbf{e}$ representing the empty structure (*i.e.*, the empty circuit). Further, we will denote the set of all such well-formed blue respectively red propositions by Φ_B respectively Φ_R.

In the same manner, well-formed blue and red contexts (a notion of a recolorable context is unnecessary) are those produced by the grammars

$$
\begin{aligned}
\Gamma_B, \Delta_B, \Pi_B &:= \cdot \mid \Gamma_B, A_B && \text{(Blue contexts)}\\
\Gamma_R, \Delta_R, \Pi_R &:= \cdot \mid \Gamma_R, A_R && \text{(Red contexts)}
\end{aligned}
$$

The distinction between the empty blue context and the empty red one is important, since the two types of contexts will be interpreted in two different ways; blue contexts are interpreted as an additive (blue) conjunction with $\mathbf{1}$ as unit, while red contexts are interpreted as an ordered multiplicative (red) conjunction with $\mathbf{e}$ as unit. As we did for propositions, we will denote the set of all well-formed blue respectively red contexts by Φ_B^* respectively Φ_R^*.

3 Proof Calculi

As the syntax presented in the previous section perhaps already alludes to, we will use not one but two proof calculi to reason about propositions thus formed. Figures 3 and 4 show the two proof calculi–the blue and the red fragment, respectively–that make up the logic which we shall call $\mathrm{LRS}_{\circledast}$.

Core rules

$$\frac{}{\Gamma, A \vdash_B A}\ (\text{BId}) \qquad \frac{\Gamma \vdash_B A \qquad \Gamma, A \vdash_B B}{\Gamma \vdash_B B}\ (\text{BCut})$$

Structural rules

$$\frac{\Gamma \vdash_B B}{\Gamma, A \vdash_B B}\ (\text{Wkn}) \qquad \frac{\Gamma, A, A \vdash_B B}{\Gamma, A \vdash_B B}\ (\text{Cnt})$$

$$\frac{\Gamma, A, \Delta, B, \Pi \vdash_B C}{\Gamma, B, \Delta, A, \Pi \vdash_B C}\ (\text{Exc})$$

Units

$$\frac{}{\Gamma \vdash_B \mathbf{1}}\ (\mathbf{1}\text{I}) \qquad \text{(no elimination for } \mathbf{1}\text{)}$$

$$\text{(no introduction for } \mathbf{0}\text{)} \qquad \frac{\Gamma \vdash_B \mathbf{0}}{\Gamma \vdash_B A}\ (\mathbf{0}\text{E})$$

Conjunction

$$\frac{\Gamma \vdash_B A \qquad \Gamma \vdash_B B}{\Gamma \vdash_B A \,\&\, B}\ (\&\text{I}) \qquad \frac{\Gamma \vdash_B A \,\&\, B}{\Gamma \vdash_B A}\ (\&\text{E}_1)$$

$$\frac{\Gamma \vdash_B A \,\&\, B}{\Gamma \vdash_B B}\ (\&\text{E}_2)$$

Control

$$\frac{\Gamma, A \vdash_B B \qquad \Gamma, B \vdash_B A}{\Gamma \vdash_B A \bullet\!\!- B}\ (\bullet\!\!-\text{I}) \qquad \frac{\Gamma \vdash_B A \bullet\!\!- B \qquad \Gamma \vdash_B B}{\Gamma \vdash_B A}\ (\bullet\!\!-\text{E}_1)$$

$$\frac{\Gamma \vdash_B A \bullet\!\!- B \qquad \Gamma \vdash_B A}{\Gamma \vdash_B B}\ (\bullet\!\!-\text{E}_2)$$

Negation

$$\frac{\Gamma, A \vdash_B \mathbf{0}}{\Gamma \vdash_B \neg A}\ (\neg\text{I}) \qquad \frac{\Gamma \vdash_B A \qquad \Gamma \vdash_B \neg A}{\Gamma \vdash_B \mathbf{0}}\ (\neg\text{E})$$

Classical rules

$$\frac{\Gamma, A \vdash_B B \qquad \Gamma, \neg A \vdash_B B}{\Gamma \vdash_B B}\ (\text{Lem}) \qquad \frac{\Gamma \vdash_B \neg\neg A}{\Gamma \vdash_B A}\ (\neg\neg\text{E})$$

Fig. 3. The blue fragment of $\text{LRS}_{\circledast}$. (Color figure online)

There are two judgment forms, $\Gamma_B \vdash_B \varphi_B$ and $\Gamma_R \vdash_R \varphi_R$, which differ not only by syntax, but also by the interpretation of the context: Blue contexts are understood to be an additive (ordinary) conjunction of its constituent propositions (as usual) with $\mathbf{1}$ as unit, while red contexts are understood as a multiplicative ordered conjunction of its constituent propositions with $\mathbf{e}$ as unit. This difference of interpretation is reflected directly in the core rules of the calculi; while the identity and cut rules for the red fragment use careful bookkeeping to ensure that order and linearity are not broken, the corresponding rules in the blue fragment display implicit use of the structural rules available in the blue fragment. More explicitly, the blue fragment contains the usual structural rules of CPL–weakening, contraction, and exchange–while the red fragment has *none* of these.

The blue fragment, largely similar to the sequent calculus of LRS [11], presents itself as a reformulation of CPL in which control (corresponding to material bi-implication) is taken as a fundamental connective, while implication and disjunction are omitted. In particular, the omission of disjunction as a connective presents a challenge for classical reasoning, as we can no longer express the law of the excluded middle axiomatically in a way which facilitates its easy use in derivations. To resolve this, we present the rule instead as an explicit case analysis, corresponding to a proof tree of the form

CORE RULES	$$\dfrac{}{A \vdash_R A}\,(\text{RId})$$	$$\dfrac{\Delta \vdash_R A \qquad \Gamma, A, \Pi \vdash_R B}{\Gamma, \Delta, \Pi \vdash_R B}\,(\text{RCut})$$
UNIT	$$\dfrac{}{\cdot \vdash_R \mathbf{e}}\,(\mathbf{e}\text{I})$$	$$\dfrac{\Delta \vdash_R \mathbf{e} \qquad \Gamma, \Pi \vdash_R A}{\Gamma, \Delta, \Pi \vdash_R A}\,(\mathbf{e}\text{E})$$
ORDERED CONJUNCTION	$$\dfrac{\Gamma \vdash_R A \qquad \Delta \vdash_R B}{\Gamma, \Delta \vdash_R A \circledast B}\,(\circledast\text{I})$$	$$\dfrac{\Delta \vdash_R A \circledast B \qquad \Gamma, A, B, \Pi \vdash_R C}{\Gamma, \Delta, \Pi \vdash_R C}\,(\circledast\text{E})$$
RECOLORING	$$(\dagger)\ \dfrac{A \vdash_B B}{A \vdash_R B}\,(\text{Rcl})$$	$\dagger$ Side condition: A and B are recolorable.

Fig. 4. The red fragment of $\text{LRS}_\circledast$. (Color figure online)

$$\dfrac{\dfrac{}{\Gamma \vdash A \vee \neg A}\,(\text{Lem}) \qquad \begin{matrix}\vdots \\ \Gamma, A \vdash B\end{matrix} \qquad \begin{matrix}\vdots \\ \Gamma, \neg A \vdash B\end{matrix}}{\Gamma \vdash B}\,(\vee\text{E})$$

in CPL, which not only presents the common use case of the law of the excluded middle, but is also strong enough to derive the double negation elimination rule in the straightforward way. (Proof theoretically inclined buyers beware: Though this rule is sufficiently powerful, it threatens the subformula property [3] even in the face of cut-elimination.) Note that as we are not aiming for minimality, both rules are included in the blue fragment.

The red fragment offers little in terms of rules, since the only structure we are interested in is the parallel structure of circuit lines, captured by the rules for ordered multiplicative conjunction – essentially corresponding to concatenation of strings (though our formulation follows the conjunctive fragment of Polakow's presentation [15] of the Lambek calculus), with $\mathbf{e}$ corresponding to the empty string.

In our setting, by far the most interesting rule of the red fragment is the recoloring rule, which states that any logical deduction from a single recolorable proposition can be inserted into a structure of unit length, as long as the succedent is likewise recolorable. Recall that the recolorable propositions are precisely those that are well-formed as both blue and red propositions, so this (purely syntactic) side condition is entirely reasonable. Figure 5 gives a larger example of an $\text{LRS}_\circledast$ derivation, showing $\neg x_1 \circledast x_1 \bullet\!\!- \neg x_2 \vdash_R \neg x_1 \circledast \neg\neg x_1 \bullet\!\!- \neg x_2$.

Finally, it is worth noting that the syntax of red propositions is not strong enough to ensure that only reversible circuits can be represented. For example, the red proposition $x_1 \circledast x_1 \,\&\, x_2 \bullet\!\!- \neg x_3$ is perfectly well-formed, but does not correspond directly to a reversible circuit. On the other hand, every reversible circuit *can* adequately, and with minimal work, be represented as a red proposition, as we saw in Sect. 2. This turned out to result in an interesting tradeoff in the proof calculi: Naturally, it would be desirable if we could guarantee that every red proposition corresponded precisely to a reversible circuit–however, by not guaranteeing this property, we may consider the semantics of a single line or group of lines in isolation, without having to take the overall structure of the circuit into account at every step of a derivation, making for simpler overall logic.

4 Semantics

Given the obvious similarities between CPL and the blue fragment of $\mathrm{LRS}_{\circledast}$, it would seem highly natural to adopt truth-functional semantics here as well. While this approach certainly works when considering the blue fragment in isolation, extending this approach to the red fragment runs into the problem of defining a single truth value–and for good reason, since truth should be interpreted relative to a circuit structure, taking order and resource use (*i.e.*, viewing circuit lines as ordered resources) into consideration.

For this reason, we will instead take the algebraic approach to semantics by considering what we call a *Toffoli lattice*, an order structure with obvious similarities to the blue fragment of $\mathrm{LRS}_{\circledast}$. This approach has the immediate benefit that order structures can very easily be interpreted as categories, giving us a whole suite of tools to extend the semantics to the red fragment. We define Toffoli lattices, and their corresponding homomorphisms, as follows:

Definition 1. *A Toffoli lattice $\mathfrak{A} = (A, \leq, \top, \bot, \wedge, \leftrightharpoons, \overline{\cdot})$ consists of a partially ordered set $(A, \leq)$ furnished with the following operations and conditions:*

(i) There is a greatest element $\top$ such that $x \leq \top$ for any element x.
(ii) There is a least element $\bot$ such that $\bot \leq x$ for any element x.
(iii) Given elements a, b there is an element $a \wedge b$ such that $x \leq a \wedge b$ iff $x \leq a$ and $x \leq b$.
(iv) Given elements a, b there is an element $a \leftrightharpoons b$, the relative equivalence *of a and b, such that $x \leq a \leftrightharpoons b$ iff $x \wedge a \leq b$ and $x \wedge b \leq a$.*
(v) Given an element a, there is an element $\overline{a}$ satisfying $x \leq \overline{a}$ iff $x \wedge a \leq \bot$, $a \wedge \overline{a} \leq \bot$, and if $x \wedge a \leq b$ and $x \wedge \overline{a} \leq b$ then $x \leq b$.

As is often done, we will use $|\mathfrak{A}|$ to denote the carrier set A.

Definition 2. *Let $\mathfrak{A}$ and $\mathfrak{B}$ be Toffoli lattices. A Toffoli lattice homomorphism is a function $h : |\mathfrak{A}| \to |\mathfrak{B}|$ that preserves all lattice operations and constants, i.e., $h(\top) = \top$, $h(\bot) = \bot$, $h(a \wedge b) = h(a) \wedge h(b)$, $h(a \leftrightharpoons b) = h(a) \leftrightharpoons h(b)$, and $h(\overline{a}) = \overline{h(a)}$ for all $a, b \in |\mathfrak{A}|$.*

From this definition, the truth-functional semantics appear by considering the set $\{0, 1\}$:

Example 1. The set $\{0, 1\}$ equipped with the usual partial order is a Toffoli lattice: Assigning the usual truth table semantics to $\top$, $\bot$, $\wedge$, and complement, and defining

$$0 \leftrightharpoons 0 = 1 \qquad 0 \leftrightharpoons 1 = 0 \qquad 1 \leftrightharpoons 0 = 0 \qquad 1 \leftrightharpoons 1 = 1$$

it is straightforward to verify that this yields a Toffoli lattice.

Though no explicit join operation is given, joins may be defined using meets and complements–*i.e.*, analogously to Boolean algebras, one can show that $\overline{\overline{x} \wedge \overline{y}}$ is the least upper bound of x and y.

Lemma 1. *Let $h : \mathfrak{A} \to \mathfrak{B}$ be a Toffoli lattice homomorphism. Then h is specifically monotonic, i.e. if $a \leq b$ then $h(a) \leq h(b)$.*

Like so many other structured sets, these definitions lead us, without much trouble, to show that Toffoli lattices with homomorphisms between them form a concrete category; a useful feature which we will use shortly to characterize the free Toffoli lattice.

Theorem 1. *The class of all Toffoli lattices with Toffoli lattice homomorphisms between them forms a category,* ***TL***.

Careful inspection of the definition of a Toffoli lattice reveals a correspondence with the blue fragment of $\mathrm{LRS}_{\circledast}$ – of course, this is entirely by design, though this correspondence is missing one part, namely the propositional atoms (recall the assumption that these form a denumerable set P). To account for these, we observe that Toffoli lattices may be freely constructed, and apply this free construction to the set of propositional atoms P to form an order theoretic model of the blue fragment.

Theorem 2. *Toffoli lattices may be freely constructed,* i.e., *the forgetful functor $U : \mathbf{TL} \to \mathbf{Sets}$ has a left adjoint $F : \mathbf{Sets} \to \mathbf{TL}$.*

Using this theorem, we take $\mathfrak{T} = FP$ (where P is the set of propositonal atoms) to be our model of the blue fragment. This allows us to define blue denotation and entailment:

Definition 3. *The denotation of a blue proposition $\varphi_B \in \Phi_B$, denoted $[\![\varphi_B]\!]$, is given by the function $[\![\cdot]\!] : \Phi \to |\mathfrak{T}|$ defined as follows:*

$$[\![\mathbf{1}]\!] = \top \qquad [\![a]\!] = a \qquad [\![A_B \,\&\, B_B]\!] = [\![A_B]\!] \wedge [\![B_B]\!]$$

$$[\![\mathbf{0}]\!] = \bot \qquad [\![\neg A_B]\!] = \overline{[\![A_B]\!]} \qquad [\![A_B \bullet\!\!- B_B]\!] = [\![A_B]\!] \leftrightharpoons [\![B_B]\!]$$

where a denotes any propositional atom in P. Further, the denotation of a blue context $\Gamma_B \in \Phi_B^*$ is given by the overloaded function $[\![\cdot]\!] : \Phi_B^* \to |\mathfrak{T}|$ defined by

$$[\![\cdot]\!] = \top \qquad [\![\Gamma'_B, A_B]\!] = [\![\Gamma'_B]\!] \wedge [\![A_B]\!]$$

Definition 4 (Blue entailment). *Let Γ be a well-formed blue context, and φ be a well-formed blue formula. Then we define the blue entailment relation by $\Gamma \vDash_B \varphi$ iff $[\![\Gamma]\!] \leq [\![\varphi]\!]$ in $\mathfrak{T}$.*

In the same manner as for any other partially ordered set, we can regard a single Toffoli lattice $\mathfrak{A}$ as a (skeletal preorder) category by considering each element of $|\mathfrak{A}|$ as an object of the category, and placing a morphism between objects X and Y iff $X \leq Y$ in $\mathfrak{A}$. This allows us to extend our model lattice $\mathfrak{T}$ by categorical means to form a model of the red fragment. A key insight in this regard is the role of monoidal categories in modelling linear logic [18]; in particular, a strict monoidal category is sufficient to model the red fragment. This leads to the following construction:

Definition 5. *Let $\mathfrak{T}_\otimes$ denote the* free strict monoidal category *over $\mathfrak{T}$. That is, $\mathfrak{T}_\otimes$ has as objects all strings $X_1X_2\dots X_n$ where all X_i are objects of $\mathfrak{T}$, and as morphisms all strings of morphisms $f_1f_2\dots f_n : X_1X_2\dots X_n \to Y_1Y_2\dots Y_n$ for morphisms $f_i : X_i \to Y_i$ of $\mathfrak{T}$. It has a monoidal tensor $-\otimes- : \mathfrak{T}_\otimes \times \mathfrak{T}_\otimes \to \mathfrak{T}_\otimes$ defined by concatenation; thus it is strictly associative and has a strict unit i, denoting the empty string.*

See, *e.g.*, Joyal and Street [9,10] for the construction of the free strict monoidal category (or, in their nomenclature, free strict tensor category) over a given category $\mathbf{C}$; it simply amounts to be the coproduct of all functor categories of the form $\mathbf{C}^n$, where n is the discrete category of n objects. This allows us to characterize $\mathfrak{T}_\otimes$ by means of a (Grothendieck) fibration (see, *e.g.*, Jacobs [7]) into the discrete category $\Delta_\mathbb{N}$ which has, as objects, all natural numbers[1]:

Theorem 3. *The functor $\Psi : \mathfrak{T}_\otimes \to \Delta_\mathbb{N}$ defined by mapping objects to their lengths as strings, and morphisms to the corresponding identities is a Grothendieck fibration and a monoidal functor. Specifically, each inverse image $\Psi^{-1}(k)$ for k in $(\Delta_\mathbb{N})_0$ is a full subcategory of $\mathfrak{T}_\otimes$.*

Proof. (Proof sketch). Since $\Delta_\mathbb{N}$ is discrete, for any object $X_1X_2\dots X_n$ of $\mathfrak{T}_\otimes$, the only possible morphism in $\Delta_\mathbb{N}$ of the form $u : K \to \Psi(X_1X_2\dots X_n)$ is the identity $1_{\Psi(X_1X_2\dots X_n)}$, which the identity morphism $1_{X_1X_2\dots X_n}$ is trivially cartesian over.

To see that Ψ is a strict monoidal functor, we note the obvious tensor product in $\Delta_\mathbb{N}$ given by addition, *i.e.*, by mapping objects $A\otimes B$ to their sum (as natural numbers) $A+B$, and likewise morphisms $1_A\otimes 1_B$ to 1_{A+B}. From this, it follows directly that $\Psi(A\otimes B) = \Psi(A)\otimes\Psi(B)$. $\square$

This approach is closely related to the theory of PROs, PROPs, and operads (see, *e.g.*, Leinster [13])–indeed, $\mathfrak{T}_\otimes$ is a PRO–but we will avoid relying on this theory for the sake of a more coherent presentation.

In order to define denotation and entailment in the red propositions, we need one last lemma, stating the obvious isomorphism between $\Psi^{-1}(1)$ (the subcategory of strings of objects of $\mathfrak{T}$ of length precisely 1) and $\mathfrak{T}$:

Lemma 2. *There exist functors $I : \mathfrak{T} \to \Psi^{-1}(1)$ and $J : \Psi^{-1}(1) \to \mathfrak{T}$ witnessing $\Psi^{-1}(1) \cong \mathfrak{T}$.*

Definition 6. *The denotation of a red proposition $\varphi_R \in \Phi_R$, denoted $[\![\varphi_R]\!]$, is given by the function $[\![\cdot]\!] : \Phi_R \to (\mathfrak{T}_\otimes)_0$ defined as follows:*

$$[\![\mathbf{e}]\!] = i \quad [\![A_R \circledast B_R]\!] = [\![A_R]\!] \otimes [\![B_R]\!] \quad [\![A_R]\!] = I([\![A_B]\!]) \quad \textit{if } A_R \textit{is a recolorable.}$$

As we did for blue propositions, we overload the denotation function to apply to (in this case, red) contexts as well, by defining the function $[\![\cdot]\!] : \Phi_R^* \to (\mathfrak{T}_\otimes)_0$ as

$$[\![\cdot]\!] = i \qquad\qquad [\![\Gamma_R, A_R]\!] = [\![\Gamma_R]\!] \otimes [\![A_R]\!]$$

[1] We use the notation $\Delta_\mathbb{N}$ for the discrete category specifically to avoid confusion with the ordinal category ω, which some authors denote $\mathbb{N}$.

Definition 7 (Red entailment). *Let Γ be a well-formed red context, and φ be a well-formed red proposition. We define red entailment by $\Gamma \vDash_R \varphi$ iff $[\![\Gamma]\!] \leq [\![\varphi]\!]$, i.e. iff there exists a morphism between the objects $[\![\Gamma]\!]$ and $[\![\varphi]\!]$ in $\mathfrak{T}_\otimes$.*

5 Metatheorems

With a semantics for both the blue and red fragments, we are ready to take on the fundamental metatheorems of soundness and completeness. The hierarchical structure of the proof calculi (and their semantics) gives a natural separation of work, as the soundness and completeness of the red fragment depends directly, via the recoloring rule, on the corresponding theorems for the blue fragment.

Theorem 4 (Soundness). *If $\Gamma \vdash_B \varphi$ then $\Gamma \vDash_B \varphi$; and if $\Gamma \vdash_R \varphi$ then $\Gamma \vDash_R \varphi$.*

Both parts follow straightforwardly by induction; the only interesting case is recoloring, which follows by Lemma 2 and soundness of the blue fragment. The completeness theorems require a little more work; blue completeness relies on the Lindenbaum-Tarski method (*i.e.*, by taking the set of blue propositions quotiented by blue provable equivalence, $\Phi_B/\dashv\vdash_B$, and showing that this is isomorphic to $\mathfrak{T}$), while red completeness uses the characterization of objects of $\mathfrak{T}_\otimes$ given by Theorem 3 as an induction principle for objects of $\mathfrak{T}_\otimes$.

Theorem 5 (Completeness). *If $\Gamma \vDash_B \varphi$ then $\Gamma \vdash_B \varphi$; and if $\Gamma \vDash_R \varphi$ then $\Gamma \vdash_R \varphi$.*

We are finally ready to tackle our previous obligation to show our propositional semantics equivalent to Toffoli's Boolean ring semantics. The first step is to show that Boolean rings are equivalent to Toffoli lattices:

Theorem 6 (Universality). *$\mathfrak{A}$ is a Toffoli lattice iff it is a Boolean ring.*

Proof (Proof sketch). If $\mathfrak{A}$ is a Toffoli lattice, we define the constants and operations of a ring by

$$0 = \bot \qquad 1 = \top \qquad a \cdot b = a \wedge b \qquad a \oplus b = a \leftrightharpoons \overline{b}$$

for all elements a and b of $\mathfrak{A}$. From this, it is straightforwardly shown that $\mathfrak{A}$ forms an abelian group under addition (with each a as its own additive inverse, and 0 as unit), and a monoid under multiplication (with 1 as unit) which further distributes over addition; thus it is a ring, and that it is Boolean follows directly by the idempotence of meets.

In the other direction, suppose $\mathfrak{A}$ is a Boolean ring; then it is also a Boolean algebra [19], so it suffices to show that a Boolean algebra is also a Toffoli lattice. But then we can construct relative equivalences by $a \leftrightharpoons b = (\overline{a} \vee b) \vee (\overline{b} \vee a)$ for all elements $a, b \in |\mathfrak{A}|$; that $\mathfrak{A}$ is then a Toffoli lattice follows by algebraic manipulation. $\square$

We now extend this result to the full generality of entire reversible circuits. Let the order of a reversible circuit denote its number of input (equivalently output) lines; having the same order is thus a trivial requirement for two reversible circuits to be strongly equivalent, as the functions they compute (denote this function $f_{\mathcal{C}}$ for a circuit $\mathcal{C}$) will otherwise differ fundamentally by domain and codomain. Further, we will use $\mathfrak{B} = (\{0,1\}, 0, 1, \oplus, \cdot)$ to denote the Boolean ring on the set $\{0,1\}$ with exclusive disjunction as addition, and conjunction as multiplication, and $\mathfrak{B}^n$ to be the direct product of $\mathfrak{B}$ with itself n times. Using Toffoli's Boolean ring semantics (as presented in Sect. 2, Fig. 1a), we will develop a semantic preorder on reversible circuits – but to do this, we need a way to handle ancillae (lines of constant value) in a clean way. This is done by the ancilla restriction on a circuit, defined as follows:

Definition 8. *Let $\mathcal{C}$ be a reversible circuit of order n, and $\mathbf{x} \in |\mathfrak{B}^n|$. We define the* ancilla restriction *on $\mathbf{x}$ with respect to $\mathcal{C}$ to be $\mathbf{x}_{|\mathcal{C}} = (c_1, c_2, \ldots, c_n)$ where each c_i is given by*

$$c_i = \begin{cases} k & \text{if the } i^{th} \text{ input of } \mathcal{C} \text{ is an ancilla of value } k \\ \pi_i(\mathbf{x}) & \text{otherwise} \end{cases}$$

This allows the following preorder on the set of reversible logic circuits, and in turn, the category induced by this preorder:

Lemma 3. *The relation on reversible circuits defined by $\mathcal{C} \leq_R \mathcal{D}$ iff $f_{\mathcal{C}}(\mathbf{x}_{|\mathcal{C}}) \leq f_{\mathcal{D}}(\mathbf{x}_{|\mathcal{D}})$ for all $\mathbf{x} \in |\mathfrak{B}^n|$ and circuits $\mathcal{C}, \mathcal{D}$ of equal order n, where the order relation $\cdot \leq \cdot$ denotes the usual (component-wise) ordering on Boolean vectors of length n, is a preorder.*

Definition 9. *Let* **RC** *denote the category which has reversible circuits as objects, and a single morphism between circuits $\mathcal{C}$ and $\mathcal{D}$ iff $\mathcal{C} \leq_R \mathcal{D}$.*

Note particularly from this definition that objects $\mathcal{C}$ and $\mathcal{D}$ of **RC** are isomorphic (i.e., $\mathcal{C} \leq_R \mathcal{D}$ and $\mathcal{D} \leq_R \mathcal{C}$) precisely when they are strongly equivalent. This allows us to show that all strong equivalences of reversible logic circuits are contained in $\mathfrak{T}_{\otimes}$:

Theorem 7 (Embedding of RC). *There exists a functor $H : \mathbf{RC} \to \mathfrak{T}_{\otimes}$ which constitutes an embedding of* **RC** *in $\mathfrak{T}_{\otimes}$, i.e., it is fully faithful; in particular $H(\mathcal{C}) \cong H(\mathcal{D})$ iff $\mathcal{C} \cong \mathcal{D}$.*

Proof. We define $H : \mathbf{RC} \to \mathfrak{T}_{\otimes}$ on objects by taking circuits to their annotation, as given by the annotation algorithm (see Sect. 2 and the example in Fig. 2), and on morphisms by taking $\mathcal{C} \leq \mathcal{D}$ to the morphism $H(\mathcal{C}) \leq H(\mathcal{D})$: That this morphism exists in $\mathfrak{T}$ follows by induction on the order of $\mathcal{C}$ (equivalently $\mathcal{D}$) by Theorem 6, since the order on the outputs is an order on Boolean ring terms, which are equivalent to Toffoli lattice terms via

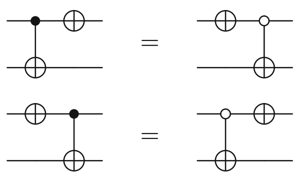
$$a \cdot b = a \wedge b \qquad a \oplus b = a \leftrightharpoons \overline{b} \qquad a \oplus 1 = a \leftrightharpoons \overline{\top} = a \leftrightharpoons \bot = \overline{a}$$

which shows, by soundness, completeness and the denotation of the propositional semantics, the exact correspondence between Toffoli's Boolean ring semantics and our propositional semantics (see Sect. 2, Figs. 1a and 1b). That $H(\mathcal{C}) \cong H(\mathcal{D})$ iff $\mathcal{C} \cong \mathcal{D}$ (equivalently, that H is fully faithful) follows likewise by induction on the order of $\mathcal{C}$ (equivalently $\mathcal{D}$) using Theorem 6. $\square$

6 Applications

Above, we have shown that the logic of $\mathrm{LRS}_{\circledast}$ is sound and complete with respect to a semantics that includes all strong equivalences of reversible logic circuits. This property suggests, as an obvious first application, a general method for proving such strong equivalences: Use the annotation algorithm of Sect. 2 to extract propositional representations of each circuit, and then use $\mathrm{LRS}_{\circledast}$ to show that their propositional representations are provably equivalent.

This approach can be applied directly in the optimization of reversible circuits. When used on very large circuits, the annotation algorithm may produce propositional representations that are infeasibly large to work with, however. Where the approach really shines is in the development and verification of template-based reversible circuit rewriting systems (see, *e.g.*, [1,20]). Template-based rewriting works by identifying certain forms of sub-circuits, allowing these to be substituted with equivalent ones.



Since such templates are typically quite modest in size, one can often extract corresponding propositions from templates with only a few steps of the annotation algorithm. A concrete example of such a template-based rewriting rule is Soeken and Thomsen's rule R2, shown on the right. Annotating these two circuits with our algorithm, the rule states precisely the equivalences

$$\neg x_1 \circledast x_1 \bullet\!- \neg x_2 \dashv\vdash_R \neg x_1 \circledast \neg\neg x_1 \bullet\!- \neg x_2 \tag{1}$$

and

$$\neg x_1 \circledast \neg x_1 \bullet\!- \neg x_2 \dashv\vdash_R \neg x_1 \circledast \neg x_1 \bullet\!- \neg x_2 \ . \tag{2}$$

which are both, indeed, provable. Note that (2) follows directly by red identity, as the annotation the two circuits resulted in syntactically identical propositions. One of the two derivations proving the (1) is shown in Fig. 5.

Using diagrammatic notation for such rewriting systems is both convenient and intuitive to use for humans. Although this has provided real insights into the rewriting behavior of reversible circuits, showing completeness (with respect to reversible circuits) for such rewriting systems has proven difficult.

Because $\mathrm{LRS}_{\circledast}$ provides sound and complete proof calculi for reasoning about reversible circuits, we can go the other way around and extract an equational theory from this that is sound and complete with respect to reversible circuits. Further, since the blue fragment of $\mathrm{LRS}_{\circledast}$ is sound and complete with respect to

$$\mathcal{D}_1 = \dfrac{\dfrac{\dfrac{}{x_1 \bullet\!\!- \neg x_2 \vdash_B x_1 \bullet\!\!- \neg x_2}\,(\text{BId})}{x_1 \bullet\!\!- \neg x_2, \neg\neg x_1 \vdash_B x_1 \bullet\!\!- \neg x_2}\,(\text{Wkn}) \qquad \dfrac{\dfrac{}{x_1 \bullet\!\!- \neg x_2, \neg\neg x_1 \vdash_B \neg\neg x_1}\,(\text{BId})}{x_1 \bullet\!\!- \neg x_2, \neg\neg x_1 \vdash_B x_1}\,(\neg\neg\text{E})}{x_1 \bullet\!\!- \neg x_2, \neg\neg x_1 \vdash_B \neg x_2}\,(\bullet\!\!-\text{E}_2)$$

$$\mathcal{D}_2 = \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x_1 \bullet\!\!- \neg x_2 \vdash_B x_1 \bullet\!\!- \neg x_2}\,(\text{BId})}{x_1 \bullet\!\!- \neg x_2, \neg x_2 \vdash_B x_1 \bullet\!\!- \neg x_2}\,(\text{Wkn}) \qquad \dfrac{}{x_1 \bullet\!\!- \neg x_2, \neg x_2 \vdash_B \neg x_2}\,(\text{BId})}{x_1 \bullet\!\!- \neg x_2, \neg x_2 \vdash_B x_1}\,(\bullet\!\!-\text{E}_1)}{x_1 \bullet\!\!- \neg x_2, \neg x_2, \neg x_1 \vdash_B x_1}\,(\text{Wkn}) \qquad \dfrac{}{x_1 \bullet\!\!- \neg x_2, \neg x_2, \neg x_1 \vdash_B \neg x_1}\,(\text{BId})}{x_1 \bullet\!\!- \neg x_2, \neg x_2, \neg x_1 \vdash_B \mathbf{0}}\,(\neg\text{E})}{x_1 \bullet\!\!- \neg x_2, \neg x_2 \vdash_B \neg\neg x_1}\,(\neg\text{I})$$

$$\mathcal{D}_0 = \dfrac{\dfrac{}{\neg x_1 \vdash_R \neg x_1}\,(\text{RId}) \qquad \dfrac{\dfrac{\dfrac{\mathcal{D}_1}{x_1 \bullet\!\!- \neg x_2, \neg\neg x_1 \vdash_B \neg x_2} \qquad \dfrac{\mathcal{D}_2}{x_1 \bullet\!\!- \neg x_2, \neg x_2 \vdash_B \neg\neg x_1}}{x_1 \bullet\!\!- \neg x_2 \vdash_B \neg\neg x_1 \bullet\!\!- \neg x_2}\,(\bullet\!\!-\text{I})}{x_1 \bullet\!\!- \neg x_2 \vdash_R \neg\neg x_1 \bullet\!\!- \neg x_2}\,(\text{RCl})}{\neg x_1, x_1 \bullet\!\!- \neg x_2 \vdash_R \neg x_1 \circledast \neg\neg x_1 \bullet\!\!- \neg x_2}\,(\circledast\text{I})$$

$$\dfrac{\dfrac{}{\neg x_1 \circledast x_1 \bullet\!\!- \neg x_2 \vdash_R \neg x_1 \circledast x_1 \bullet\!\!- \neg x_2}\,(\text{RId}) \qquad \dfrac{\mathcal{D}_0}{\neg x_1, x_1 \bullet\!\!- \neg x_2 \vdash_R \neg x_1 \circledast \neg\neg x_1 \bullet\!\!- \neg x_2}}{\neg x_1 \circledast x_1 \bullet\!\!- \neg x_2 \vdash_R \neg x_1 \circledast \neg\neg x_1 \bullet\!\!- \neg x_2}\,(\circledast\text{E})$$

Fig. 5. Derivation in $\text{LRS}_{\circledast}$ for verifying the first direction of Soeken and Thomsen's rule R2.

Toffoli lattices, we can instead extract an equational theory for the blue fragment from the definition of a Toffoli lattice, using the following lemma:

Lemma 4. *In any Toffoli lattice, $a \leq b$ iff $a \wedge \overline{b} = \bot$.*

This lemma allows us to straightforwardly recast the definition of a Toffoli lattice in purely equational terms (although the result is not exactly elegant). What this *does* give us, is a set of equations that must hold for all Toffoli lattices, and which any other complete equational theory must therefore be equivalent to, and the means to show such an equivalence by converting equalities to statements about the underlying order structure and vice versa.

Figure 6 shows a more pleasing equational theory for the blue fragment, presented in the syntax of $\text{LRS}_{\circledast}$ (the intrinsic properties of equality, *i.e.*, reflexivity, symmetry, transitivity, and congruences are not shown,) proven equivalent (and therefore sound and complete) exactly in the way outlined above (by the power of boring algebra). Deriving an equational theory for the red fragment is simpler, as it is sound and complete with respect to the free monoidal part of $\mathfrak{T}_{\otimes}$, which is already expressed in equational terms. As such, the equational theory for the red fragment shown in Fig. 6 is sound and complete by definition, though congruences applied in the red fragment are syntactically restricted by recolorability; that is, we can only replace recolorable propositions by recolorable propositions.

The usefulness of such an equational theory is evident in that we can, *e.g.*, now prove the soundness of the R2 rules directly by applying equation (B9) in Fig. 6. Such equational theories can themselves also be used to develop new rewriting systems for reversible circuits, in particular to suggest new templates.

$$\varphi \circledast (\psi \circledast \chi) \overset{(R1)}{=} (\varphi \circledast \psi) \circledast \chi \qquad (\varphi \mathbin{\&} \psi) \mathbin{\&} \chi \overset{(B4)}{=} \varphi \mathbin{\&} (\psi \mathbin{\&} \chi)$$

$$\varphi \circledast \mathbf{e} \overset{(R2)}{=} \varphi \qquad \varphi \bullet\!\!- \psi \overset{(B5)}{=} \psi \bullet\!\!- \varphi$$

$$\mathbf{e} \circledast \varphi \overset{(R3)}{=} \varphi \qquad (\varphi \bullet\!\!- \psi) \bullet\!\!- \chi \overset{(B6)}{=} \varphi \bullet\!\!- (\psi \bullet\!\!- \chi)$$

$$\varphi \mathbin{\&} \neg(\psi \mathbin{\&} \chi) \overset{(B7)}{=} \neg(\neg(\varphi \mathbin{\&} \neg\psi) \mathbin{\&} \neg(\varphi \mathbin{\&} \neg\chi))$$

$$\varphi \mathbin{\&} \mathbf{1} \overset{(B1)}{=} \varphi \qquad \varphi \bullet\!\!- \psi \overset{(B8)}{=} \neg(\varphi \mathbin{\&} \neg\psi) \mathbin{\&} \neg(\psi \mathbin{\&} \neg\varphi)$$

$$\varphi \mathbin{\&} \mathbf{0} \overset{(B2)}{=} \mathbf{0} \qquad \neg\neg\varphi \overset{(B9)}{=} \varphi$$

$$\varphi \mathbin{\&} \psi \overset{(B3)}{=} \psi \mathbin{\&} \varphi \qquad \varphi \mathbin{\&} \neg\varphi \overset{(B10)}{=} \mathbf{0}$$

Fig. 6. Sound and complete equational theories for the two calculi. (Color figure online)

7 Conclusion and Future Work

In this article, we have presented a syntactic representation of reversible logic circuits centered around the control interpretation of Toffoli's reversible gate set, and shown, via two proof calculi of natural deduction, that a variant of classical propositional logic extended with ordered multiplicative conjunction is sufficient to reason about these. We have developed an algebraic and categorical semantics, shown that the proof calculi are sound and complete with respect to these, and that this model subsumes the established notion of strong equivalence of reversible logic circuits. Finally, we have shown how our work can be used to prove strong equivalences of reversible logic circuits, to verify existing systems of reversible logic circuit rewriting, and to develop new such rewriting systems.

The approach has been successful in enabling reasoning about reversible logic circuits, but it is not quite on even footing with the template-based approaches to reversible circuit rewriting, as these use a graphical circuit notation which, by definition, asserts circuit reversibility on every rewriting step. Although our approach faithfully models circuit semantics, it is not currently clear when looking at an arbitrary proposition whether it corresponds to a reversible circuit or not. On the other hand, by decoupling the propositions from the graphical representations, the current logic may allow for much shorter rewritings than if each step must yield representations which directly translate to circuits in this way.

References

1. Arabzadeh, M., Saeedi, M., Zamani, M.S.: Rule-based optimization of reversible circuits. In: Proceedings of the ASP-DAC 2010, pp. 849–854. IEEE (2010)
2. Bérut, A., Arakelyan, A., Petrosyan, A., Ciliberto, S., Dillenschneider, R., Lutz, E.: Experimental verification of Landauer's principle linking information and thermodynamics. Nature **483**(7388), 187–189 (2012)
3. Buss, S.R.: Handbook of Proof Theory. Elsevier, Amsterdam (1998)
4. De Vos, A.: Reversible Computing. Wiley-VCH, Weinheim (2010)

5. De Vos, A., Burignat, S., Glück, R., Mogensen, T.Æ., Axelsen, H.B., Thomsen, M.K., Rotenberg, E., Yokoyama, T.: Designing garbage-free reversible implementations of the integer cosine transform. ACM J. Emerg. Tech. Com. **11**(2), 11:1–11:15 (2014)
6. Feynman, R.P.: Quantum mechanical computers. Found. Phys. **16**(6), 507–531 (1986)
7. Jacobs, B.: Categorical Logic and Type Theory. Elsevier, Amsterdam (1999)
8. Jordan, S.P.: Strong equivalence of reversible circuits is coNP-complete. Quantum Inf. Comput. **14**(15–16), 1302–1307 (2014)
9. Joyal, A., Street, R.: The geometry of tensor calculus I. Adv. Math. **88**(1), 55–112 (1991)
10. Joyal, A., Street, R.: Braided tensor categories. Adv. Math. **102**(1), 20–78 (1993)
11. Kaarsgaard, R.: Towards a propositional logic for reversible logic circuits. In: Proceedings of the ESSLLI 2014 Student Session, pp. 33–41 (2014). http://www.kr.tuwien.ac.at/drm/dehaan/stus2014/proceedings.pdf
12. Landauer, R.: Irreversibility and heat generation in the computing process. IBM J. Res. Dev. **5**(3), 261–269 (1961)
13. Leinster, T.: Higher Operads, Higher Categories. London Mathematical Society Lecture Note Series, vol. 298. Cambridge University Press, Cambridge (2004)
14. Orlov, A.O., Lent, C.S., Thorpe, C.C., Boechler, G.P., Snider, G.L.: Experimental test of Landauer's principle at the sub-$k_b t$ level. Japan. J. Appl. Phys. **51**, 06FE10 (2012)
15. Polakow, J.: Ordered linear logic and applications. Ph.D. thesis, CMU (2001)
16. Rendel, T., Ostermann, K.: Invertible syntax descriptions: unifying parsing and pretty printing. ACM SIGPLAN Notices, vol. 45, No. 11, pp. 1–12 (2010)
17. Schellekens, M.P.: MOQA: unlocking the potential of compositional static average-case analysis. J. Log. Algebr. Program. **79**(1), 61–83 (2010)
18. Seely, R.A.G.: Linear logic, *-autonomous categories and cofree coalgebras. Contemp. Math. **92**, 371–382 (1989)
19. Sikorski, R.: Boolean Algebras. Springer, Heidelberg (1969)
20. Soeken, M., Thomsen, M.K.: White dots *do* matter: rewriting reversible logic circuits. In: Dueck, G.W., Miller, D.M. (eds.) RC 2013. LNCS, vol. 7948, pp. 196–208. Springer, Heidelberg (2013)
21. Thomsen, M.K., Glück, R., Axelsen, H.B.: Reversible arithmetic logic unit for quantum arithmetic. J. Phys. A Math. Theor. **43**(38), 382002 (2010)
22. Toffoli, T.: Reversible computing. In: de Bakker, J., van Leeuwen, J. (eds.) Automata, Languages and Programming. LNCS, vol. 85, pp. 632–644. Springer, Heidelberg (1980)
23. Wille, R., Drechsler, R.: Towards a Design Flow for Reversible Logic. Springer, Heidelberg (2010)

A psychologist said, "They used to talk about seeing only 'reflections' of reality. Not reality itself. The main thing wrong with a reflection is not that it isn't real, but that it's reversed."

Philip K. Dick, *A Scanner Darkly*

B
Foundations of reversible recursion

This chapter contains two papers related to the categorical foundations of reversible recursion.

(**B1**) R. Kaarsgaard, H. B. Axelsen, and R. Glück. Join Inverse Categories and Reversible Recursion. *Journal of Logical and Algebraic Methods in Programming*, volume 87, pages 33–50, February 2017.

(**B2**) R. Kaarsgaard. Inversion, Fixed Points, and the Art of Dual-Wielding. In S. Alves and R. Wassermann, editors, *Proceedings of the 12th Workshop on Logical and Semantic Frameworks with Applications (LSFA 2017)*, pages 94–109, available at `http://lsfa2017.cic.unb.br/LSFA2017.pdf`, 2017.

Paper (**B1**) is the extended version of a conference paper, which, in turn, was extended from an abstract presented at a workshop:

- H. B. Axelsen and R. Kaarsgaard. Join inverse categories as models of reversible recursion. In B. Jacobs and C. Löding, editors, *Foundations of Software Science and Computation Structures (FoSSaCS)*, Lecture Notes in Computer Science volume 9634, pages 73–90, Springer Verlag, 2016.

- R. Kaarsgaard. Join inverse categories and reversible recursion. In L. Aceto, I. Fábregas, Á. García-Perez, A. Ingolfsdottir, editors, *Proceedings of the 27th Nordic Workshop on Programming Theory*, Technical Report RUTR-SCS16001, pages 69–71, Reykjavík University, 2015.

Journal of Logical and Algebraic Methods in Programming 87 (2017) 33–50

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Join inverse categories and reversible recursion ☆

Robin Kaarsgaard *, Holger Bock Axelsen, Robert Glück

DIKU, Department of Computer Science, University of Copenhagen, Denmark

ARTICLE INFO

Article history:
Received 31 January 2016
Received in revised form 14 August 2016
Accepted 16 August 2016
Available online 24 August 2016

Keywords:
Reversible computing
Recursion
Categorical semantics
Enriched category theory

ABSTRACT

Recently, a number of reversible functional programming languages have been proposed. Common to several of these is the assumption of totality, a property that is not necessarily desirable, and certainly not required in order to guarantee reversibility. In a categorical setting, however, faithfully capturing partiality requires handling it as additional structure. Recently, Giles studied inverse categories as a model of partial reversible (functional) programming. In this paper, we show how additionally assuming the existence of countable joins on such inverse categories leads to a number of properties that are desirable when modeling reversible functional programming, notably morphism schemes for reversible recursion, a †-trace, and algebraic ω-compactness. This gives a categorical account of reversible recursion, and, for the latter, provides an answer to the problem posed by Giles regarding the formulation of recursive data types at the inverse category level.

1. Introduction

Reversible computing, that is, the study of computations that exhibit both forward and backward determinism, originally grew out of the thermodynamics of computation. Landauer's principle states that computations performed by some physical system (thermodynamically) dissipate heat when information is erased, but that no dissipation is entailed by information-preserving computations [3]. This has motivated a long study of diverse reversible computation models, such as logic circuits [4], Turing machines [5,6], and many forms of restricted automata models [7,8]. Reversibility concepts are important in quantum computing, but are increasingly seen to be of interest in other areas as well, including high-performance computing [9], process calculi [10], and even robotics [11,12].

In this paper we concern ourselves with the categorical underpinnings of reversible functional programming languages. At the programming language level, reversible languages exhibit interesting program properties, such as easy program inversion [13]. Now, most reversible languages are stateful, giving them a fairly straightforward semantic interpretation [14]. While functional programs are usually easier to reason about at the meta-level, they do not have the concept of state that imperative languages do, making their semantics interesting objects of study.

Further, many reversible functional programming languages (such as Theseus [15] and the Π-family of combinator calculi [16]) come equipped with a tacit assumption of totality, a property that is neither required [6] nor necessarily desirable as far as guaranteeing reversibility is concerned. Shedding ourselves of the "tyranny of totality," however, requires us to handle partiality explicitly as additional categorical structure.

☆ This is an extended version of an abstract presented at NWPT 2015 [1] and a paper presented at FoSSaCS 2016 [2], elaborated with full proofs, additional examples, and more comprehensive background.

* Corresponding author.
E-mail addresses: robin@di.ku.dk (R. Kaarsgaard), funkstar@di.ku.dk (H.B. Axelsen), glueck@di.ku.dk (R. Glück).

http://dx.doi.org/10.1016/j.jlamp.2016.08.003

One approach which does precisely that is inverse categories, as studied by Cockett and Lack [17] as a specialization of restriction categories, which have recently been suggested and developed by Giles [18] as models of reversible (functional) programming. In this paper, we will argue that assuming ever slightly more structure on these inverse categories, namely the presence of *countable joins* of parallel morphisms [19], gives rise to a number of additional properties useful for modeling reversible functional programming. Notably, we obtain two different notions of reversible recursion (exemplified in the two different reversible languages RFUN and Theseus), and an account of recursive data types (via algebraic ω-compactness with respect to structure-preserving functors), which are not present in the general case. This is done by adopting two different, but complementary, views on inverse categories with countable joins as enriched categories – as **DCPO**-categories, and as (specifically Σ**Mon**-enriched) strong unique decomposition categories [20,21].

Overview. We give a brief introduction to reversible functional programming, specifically to the languages of RFUN [22] and Theseus [15], in Section 2, and present the necessary background on restriction and inverse categories in Section 3. In Section 4 we show that inverse categories with countable joins are **DCPO**-enriched, which allows us to demonstrate the existence of (reversible!) fixed points of both morphism schemes and structure-preserving functors. In Section 5 we show that inverse categories with countable joins and a join-preserving disjointness tensor are (strong) unique decomposition categories equipped with a uniform †-trace. Section 6 gives conclusions and directions for future work.

2. On reversible functional programming

In this section, we give a brief introduction to reversible functional programming, specifically to the languages of RFUN and Theseus. For more comprehensive accounts of these languages, including syntax, semantics, program inversion, further examples, and so on, see [22] respectively [15].

Reversible programming deals with the construction of programs that are deterministic not just in the forward direction (as any other deterministic program), but also in the backward direction. A central consequence of this property is that well-formed programs must have both uniquely defined forward and backward semantics, with backward semantics given either directly or indirectly (*e.g.*, as is often done, by providing a textual translation of terms into terms which carry their inverse semantics; this approach is related to program inversion [23,24]). In the case of reversible functional programming, reversibility is accomplished by guaranteeing local (forward and backward) determinism of evaluation – which, in turn, leads to global (forward and backward) determinism. Though reversible functions are injective [6], injectivity itself (a *global* property) is not enough to guarantee reversibility (a *local* property) – specifically, locally reversible control structures are necessary [22].

One such reversible functional programming language is RFUN, developed in recent years by Yokoyama, Axelsen, and Glück [22]. RFUN is an untyped language that uses Lisp-style symbols and constructors for data representation. Programs in RFUN are first-order functions, in which bound variables must be linearly used (though patterns are not required to be exhaustive). To account for the fact that data duplication *can* be performed reversibly, a *duplication-equality* operator [25], defined as follows, is used:

$$\lfloor \langle x \rangle \rfloor = \langle x, x \rangle$$

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

In the first case, the application of $\lfloor \cdot \rfloor$ to the unary tuple $\langle x \rangle$ yields the binary tuple $\langle x, x \rangle$, that is, the value x is duplicated. In the second case, when $x = y$, the application to $\langle x, y \rangle$ joins two identical values into $\langle x \rangle$; otherwise, the two values are returned unchanged (two different values cannot have been obtained by duplication of one value). Using an explicit operator simplifies reverse computation because the duplication of a value in one direction requires an equality check in the other direction, and *vice versa*. Instead of using a variable twice to duplicate a value, the duplication is made explicit. The operator is self-inverse, *e.g.*, $\lfloor \lfloor \langle x \rangle \rfloor \rfloor = \langle x \rangle$ and $\lfloor \lfloor \langle x, y \rangle \rfloor \rfloor = \langle x, y \rangle$.

The only control structure available in RFUN is a reversible case-expression employing the *symmetric first-match policy*: The control expression is matched against the patterns in the order they are given (as in, *e.g.*, the ML-family of languages), but, for the case-expression to be defined, once a match is found, any value produced by the matching branch must *not* match patterns that could have been produced by a previous branch. This policy guarantees reversibility. Perhaps surprising is the fact that recursion works in RFUN completely analogously to the way it works irreversibly; *i.e.*, using a call stack. In particular, inversion of recursive functions is handled simply by replacing the recursive call with a call to the inverse, and inverting the remainder of the function body. As such, the inverse of a recursive function is, again, a recursive function. This point will prove important later on.

An example of an RFUN program for computing Fibonacci-pairs is shown in Fig. 1 [22,25]: Given a natural number n encoded in unary, *fib*(n) produces the pair $\langle f_{n+1}, f_{n+2} \rangle$ where f_i is the unary encoding of the i'th Fibonacci number. Notice the use of the duplication operator in the definition of *plus*: The duplication-equality operator on the right-hand side of the first branch of *plus* duplicates $\langle x \rangle$ into $\langle x, x \rangle$ in the forward direction, and checks the equality of two values $\langle x, y \rangle$ in the backward direction. This accounts for the fact that the first branch of *plus* always returns two identical values, while the second branch always returns two different values. The first-match policy of RFUN described above guarantees the reversibility of the auxiliary function *plus*, which is defined by *plus* $\langle x, y \rangle = \langle x, x + y \rangle$.

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

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

Fig. 1. The Fibonacci-pair program and its helper function $\mathit{plus}\ \langle x, y\rangle = \langle x, x+y\rangle$ in RFUN.

$$
\begin{array}{l}
\textbf{type}\ \underline{\mathrm{Bool}} = \mathrm{False} \mid \mathrm{True} \\
\textbf{type}\ \underline{\mathrm{Nat}}\ \ = 0 \mid \mathrm{Succ}\ \underline{\mathrm{Nat}} \\
\\
\mathit{not}\ :: \ \underline{\mathrm{Bool}} \leftrightarrow \underline{\mathrm{Bool}} \\
\ \mid \ \ \mathrm{False} \ \ \leftrightarrow \ \ \mathrm{True} \\
\ \mid \ \ \mathrm{True} \ \ \leftrightarrow \ \ \mathrm{False} \\
\\
\mathit{parity}\ :: \ \underline{\mathrm{Nat}} \times \underline{\mathrm{Bool}} \leftrightarrow \underline{\mathrm{Nat}} \times \underline{\mathrm{Bool}} \\
\ \mid \ \ (n, b) \qquad\qquad\qquad \leftrightarrow \ \ \mathit{iter}\ (n, 0, b) \\
\ \mid \ \ \mathit{iter}\ (\mathrm{Succ}\ n, m, b) \ \ \leftrightarrow \ \ \mathit{iter}\ (n, \mathrm{Succ}\ m, \mathrm{not}\ b) \\
\ \mid \ \ \mathit{iter}\ (0, m, b) \qquad\ \ \leftrightarrow \ \ (m, b) \\
\qquad \textbf{where}\ \mathit{iter} :: \underline{\mathrm{Nat}} \times \underline{\mathrm{Nat}} \times \underline{\mathrm{Bool}} \leftrightarrow \underline{\mathrm{Nat}} \times \underline{\mathrm{Nat}} \times \underline{\mathrm{Bool}}
\end{array}
$$

Fig. 2. The parity program in Theseus and its type definitions and helper function $\mathit{not} :: \mathrm{Bool} \leftrightarrow \mathrm{Bool}$.

A different approach to reversible functional programming is given by Theseus, a language developed recently by James and Sabry [15] on top of the Π^0 reversible combinator calculus [16,26]. Unlike RFUN, Theseus features a simple type system with products and sums as well as the unit and empty type, and supports user-defined (isorecursive) data types (declared in a style similar to Haskell and languages in the ML-family). Like RFUN, bound variables must be linearly used, but unlike RFUN, pattern clauses must be *exhaustive* and *non-overlapping*, properties that when combined makes both guaranteeing local reversibility and producing inverse programs straightforward.

Though Theseus is, like RFUN, a first-order language, an elegant feature with a flavor of higher-order programming is its support for *parametrized maps*, *i.e.*, functions that depend statically on data of a given type in order to produce a reversible function. For example, though ordinary function composition cannot be performed reversibly without production of garbage, if we know the functions to compose no later than at compile time, we can certainly produce their composition at compile time without additional garbage. This is handled in the Theseus type system by allowing both irreversible and (first order) reversible arrow types, with the proviso that all irreversible arrows be discharged at compile time. In this way, a parametrized function $\Omega :: (a \leftrightarrow b) \to (a \leftrightarrow b)$ is *not* a Theseus program, but if $f :: a \leftrightarrow b$ is a first-order reversible function, then so is $\Omega\ f :: a \leftrightarrow b$.

Recursion in Theseus is achieved using *typed iteration labels* (a feature unique to Theseus) that specify the type and behavior of intermediate, tail-recursive computations. This feature is perhaps best illustrated by an example: Fig. 2 shows a Theseus-program (courtesy of [15]) that recursively computes the parity of a natural number (encoded in unary). Though this approach sacrifices the possibility for expressing programs with nested recursion, the benefit is that all thus specified recursive reversible functions can be expressed as a non-recursive function that a †-trace operator is applied to (*cf.* [15, 16,26] for details regarding this operator specifically in the context of Theseus and Π^0). That is to say, if $f :: a \leftrightarrow b$ is a recursive Theseus program with an iteration label of type u, we may construct a (non-recursive) function $f' :: a + u \leftrightarrow b + u$ such that the trace of f' (tracing out u, the type of the iteration label) is precisely f. This eases the process of obtaining inverse semantics greatly, as we only need to take the trace of the inverse of the (non-recursive, so straightforward to invert) function f' to obtain the inverse of the recursive function f.

3. Background

This section gives an introduction to restriction and inverse categories (with joins), dagger categories, and categories of partial maps as they will be used in the remainder of this text. Unless otherwise stated, the material described in this section can be found in numerous texts on restriction and inverse category theory (*e.g.*, Cockett and Lack [17,27,28], Giles [18], Guo [19]).

3.1. Restriction and inverse categories

We begin by recalling the definition of *restriction structures* and *restriction categories*.

Definition 1 *(Cockett and Lack).* A *restriction structure* on a category consists of an operator $\overline{(\cdot)}$ on morphisms mapping each morphism $f : A \to B$ to a morphism $\overline{f} : A \to A$ (the *restriction idempotent* of f) such that

(i) $f \circ \overline{f} = f$ for all morphisms $f : A \to B$,

(ii) $\overline{f} \circ \overline{g} = \overline{g} \circ \overline{f}$ whenever $\mathrm{dom}(f) = \mathrm{dom}(g)$,
(iii) $\overline{f \circ \overline{g}} = \overline{f} \circ \overline{g}$ whenever $\mathrm{dom}(f) = \mathrm{dom}(g)$, and
(iv) $\overline{h} \circ f = f \circ \overline{h \circ f}$ whenever $\mathrm{cod}(f) = \mathrm{dom}(h)$.

A category with a restriction structure is called a *restriction category*.

As a trivial example, any category can be equipped with a restriction structure given by setting $\overline{f} = 1_A$ for every morphism $f : A \to B$. However, there are also many useful and nontrivial examples of restriction categories (see, *e.g.*, [17, Sec. 2.1.3]), the canonical one being the category **Pfn** of sets and partial functions. In this category, the restriction idempotent $\overline{f} : A \to A$ for a partial function $f : A \to B$ is given by the partial identity function

$$\overline{f}(x) = \begin{cases} x & \text{if } f \text{ is defined at } x, \\ \text{undefined} & \text{otherwise.} \end{cases} \tag{1}$$

A related example is the category **PTop** of topological spaces and partial continuous functions (see also [18]). A partial function between topological spaces $f : X \to Y$ is continuous if

(i) the domain of definition of f (the set of points $X' \subseteq X$ for which f is defined) is open in X, and
(ii) f is continuous in the usual sense, *i.e.*, the set $f^{-1}(V) \subseteq X$ is open when $V \subseteq Y$ is.

Under this definition, **PTop** is a restriction category, with restriction idempotents given precisely as in Eq. (1); that a partial continuous function f is defined on an open set ensures precisely that $\overline{f}$ is continuous as well (and defined on an open set). Other examples include the category **DCPO** of pointed directed-complete partial orders and strict continuous maps, slice categories $\mathscr{C}/A$ for an object A of a restriction category $\mathscr{C}$ [17], and any inverse monoid (see also [29]) viewed as a (one-object) category.

Since we take restrictions as additional structure, we naturally want a notion of functors that preserve this structure.

Definition 2. A functor $F : \mathscr{C} \to \mathscr{D}$ between restriction categories $\mathscr{C}$ and $\mathscr{D}$ is a *restriction functor* if $\overline{F(f)} = F(\overline{f})$ for all morphisms f of $\mathscr{C}$.

A morphism $f : A \to B$ of a restriction category is said to be *total* if $\overline{f} = 1_A$. Given a restriction category $\mathscr{C}$, we can form the category $\mathrm{Total}(\mathscr{C})$, consisting of all objects and only the total morphisms of $\mathscr{C}$, which embeds in $\mathscr{C}$ via a faithful restriction functor. Restriction categories with restriction functors form a category, **rCat**.

Moving on to inverse categories, in order to define these[1] we first need the notion of a partial isomorphism:

Definition 3. In a restriction category $\mathscr{C}$, we say that a morphism $f : A \to B$ is a *partial isomorphism* if there exists a unique morphism $f^\circ : B \to A$ of $\mathscr{C}$ (the *partial inverse* of f) such that $f^\circ \circ f = \overline{f}$ and $f \circ f^\circ = \overline{f^\circ}$.

Definition 4. A restriction category $\mathscr{C}$ is said to be an *inverse category* if all morphisms of $\mathscr{C}$ are partial isomorphisms.

In this manner, if we accept an intuition of restriction categories as "categories with partiality," inverse categories are "groupoids with partiality" – and, indeed, the category **PInj** of sets and partial injective functions is *the* canonical example of an inverse category. In fact, the Wagner–Preston representation theorem (see, *e.g.*, [29]) for inverse monoids can be extended to show that every locally small inverse category can be faithfully embedded in **PInj** (see the two independent proofs by Kastl [30] and Heunen [31], or Cockett and Lack [17] for the special case of small inverse categories).

The analogy with groupoids goes even further; similar to how we can construct a groupoid $\mathrm{Core}(\mathscr{C})$ by taking only the isomorphisms of $\mathscr{C}$, every restriction category $\mathscr{C}$ has a subcategory $\mathrm{Inv}(\mathscr{C})$ that is an inverse category with the same objects as $\mathscr{C}$, and all partial isomorphisms of $\mathscr{C}$ as morphisms.

More generally, inverse categories are *dagger categories* (sometimes also called *categories with involution*):

Definition 5. A category $\mathscr{C}$ is said to be a *dagger category* if it is equipped with a contravariant endofunctor $(-)^\dagger : \mathscr{C}^{\mathrm{op}} \to \mathscr{C}$ such that $1_A^\dagger = 1_A$ and $f^{\dagger\dagger} = f$ for all morphisms f and identities 1_A.

Note that this definition in particular implies that a dagger functor must act as the identity on objects.[2]

[1] Strictly speaking, inverse categories predate restriction categories – see Kastl [30] for the first published article on inverse categories to the knowledge of the authors. Though we will use the axiomatization following from restriction categories, inverse categories can equivalently be defined as the categorical extension of inverse monoids, *i.e.*, as categories where all morphisms have a regular inverse, and all idempotents commute.

[2] Though it may look as if we are superfluously demanding preservation of identities at first glance, what we are stating is something stronger, namely that the dagger functor must map identities in $\mathscr{C}^{\mathrm{op}}$ to *themselves* in $\mathscr{C}$. That is, we are requiring $1_A^\dagger = 1_A$ rather than $1_A^\dagger = 1_{A^\dagger}$.

Proposition 1. *Every inverse category $\mathscr{C}$ is a dagger category with the dagger functor given by $A^\dagger = A$ on objects, and $f^\dagger = f^\circ$ on morphisms.*

As is conventional, we will call $f^\dagger$ the *adjoint* of f, and say that f is *self-adjoint* (or *hermitian*) if $f = f^\dagger$, and *unitary* if $f^\dagger = f^{-1}$. In inverse categories, unitary morphisms thus correspond precisely to (total) isomorphisms. For the remainder of this text, we will use this induced dagger-structure when referring to the partial inverse of a morphism (and write, *e.g.*, $f^\dagger$ rather than f°).

A useful feature of this definition of inverse categories is that we do not need an additional notion of an "inverse functor" as a functor that preserves partial inverses; restriction functors suffice.

Proposition 2. *Every restriction functor $F : \mathscr{C} \to \mathscr{D}$ between inverse categories preserves the canonical dagger structure of $\mathscr{C}$,* i.e.*, $F(f)^\dagger = F(f^\dagger)$ for all morphisms f of $\mathscr{C}$.*

A simple argument for this proposition is the fact that partial isomorphisms are defined purely in terms of composition and restriction idempotents, both of which are preserved by restriction functors. Inverse categories with restriction functors form a category, **invCat**.

3.2. Split restriction categories and categories of partial maps

When working in a category with a distinguished class of idempotents, it is often desirable that they split.[3] In restriction categories, such a distinguished class is given by the class of restriction idempotents, leading us to the straightforward definition of a split restriction category:

Definition 6. A restriction category in which every restriction idempotent splits is called a *split restriction category*.

For example, when equipped with the usual restriction structure, **Pfn** and **PTop** are both split restriction categories, though it straightforward to come up with subcategories of either that are not. It follows, by way of the Karoubi envelope, that every restriction category $\mathscr{C}$ can be embedded in a split restriction category $\mathrm{Split}(\mathscr{C})$ via a fully faithful restriction functor (though this requires us to show that $\mathrm{Split}(\mathscr{C})$ inherits the restriction structure from $\mathscr{C}$, and that the fully faithful functor into this category preserves restrictions; see Prop. 2.26 of [17] for details). Prime examples of split restriction categories are categories of partial maps.

Categories of partial maps provide a synthetic approach to partiality in a categorical setting [32]. To form a category of partial maps, we consider a *stable system of monics*: In a category $\mathscr{C}$, a collection $\mathcal{M}$ of monics of $\mathscr{C}$ is said to be a stable system of monics if it contains all isomorphisms of $\mathscr{C}$ and is closed under composition and pullbacks (in the sense that the pullback m' of an $m : X \to B$ in $\mathcal{M}$ along any $f : A \to B$ exists and $m' \in \mathcal{M}$). Given such a stable system of monics $\mathcal{M}$ in a category $\mathscr{C}$, we can form the category of partial maps as follows:

Proposition 3. *Given a category $\mathscr{C}$ and a stable system of monics $\mathcal{M}$ of $\mathscr{C}$, we form the category of partial maps $\mathrm{Par}(\mathscr{C}, \mathcal{M})$ by choosing the objects to be the objects of $\mathscr{C}$, and placing a morphism $(m, f) : A \to B$ for every pair (m, f) where $m : A' \to A \in \mathcal{M}$ and $f : A' \to B$ is a morphism of $\mathscr{C}$, as in*

$$\begin{tikzcd} & A' \arrow[dl, hook', "m"'] \arrow[dr, "f"] & \\ A & & B \end{tikzcd}$$

factored out by the equivalence relation $\cdot \sim \cdot$ in which $(m, f) \sim (m', f')$ if there exists an isomorphism $\alpha : A' \to A''$ such that $m' \circ \alpha = m$ and $f' \circ \alpha = f$. Composition of morphisms $(m, f) : A \to B$ and $(m', g) : B \to C$ is given by $(m \circ m'', g \circ f') : A \to C$ where m'' and f' arise from the pullback

$$\begin{tikzcd} & & A'' \arrow[dl, hook', "m''"'] \arrow[dr, "f'"] & & \\ & A' \arrow[dl, hook', "m"'] \arrow[dr, "f"'] & & B' \arrow[dl, hook', "m'"] \arrow[dr, "g"] & \\ A & & B & & C \end{tikzcd}$$

where $m'' \circ m \in \mathcal{M}$ precisely by $\mathcal{M}$ closed under composition and pullbacks.

[3] Recall that a splitting of an idempotent $e : A \to A$ consists of an object A' and morphisms $m : A \to A'$ and $r : A' \to A$ such that $r \circ m = e$ and $m \circ r = \mathrm{id}_{A'}$.

Categories of partial maps are prime examples of restriction categories; in fact, of split restriction categories. Even further, every split restriction category is isomorphic to a category of partial maps [17]. As previously noted, every restriction category can be fully and faithfully embedded in a split restriction category, and consequently, in a category of partial maps.

3.3. Partial order enrichment, joins, and compatibility

A useful feature of restriction categories, and one we will exploit throughout this article, is that hom-sets can be equipped with a partial order, defined as follows:

Proposition 4. *In a restriction category $\mathscr{C}$, every hom-set $\mathrm{Hom}_{\mathscr{C}}(A, B)$ can be equipped with the structure of a partial order where we let $f \le g$ iff $g \circ \overline{f} = f$. Further, every restriction functor F is locally monotone with respect to this order, in the sense that $f \le g$ implies $F(f) \le F(g)$.*

In **Pfn**, this corresponds to the usual partial order on partial functions: For $f, g : A \to B$, $f \le g$ if, for all $x \in A$, f is defined at x implies that g is defined at x and $f(x) = g(x)$.

A natural question to ask is when this partial order has a least element: A sufficient condition for this is when the restriction category has a *restriction zero*.

Definition 7. A restriction category $\mathscr{C}$ has a *restriction zero* object 0 iff for all objects A and B, there exists a *unique* morphism $0_{A,B} : A \to B$ that factors through 0 and satisfies $\overline{0_{A,A}} = 0_{A,A}$.

If such a restriction zero object exists, it is unique up to (total) isomorphism (as it is a zero object in the usual sense). When a given restriction category has such a restriction zero, the zero map $0_{A,B} : A \to B$ is precisely the least element of $\mathrm{Hom}_{\mathscr{C}}(A, B)$. Note that it may seem more natural to require instead that $\overline{0_{A,B}} = 0_{A,A}$ for all A and B. This is equivalent to requiring $\overline{0_{A,A}} = 0_{A,A}$:

Lemma 1. *When a restriction category has the zero object, $\overline{0_{A,A}} = 0_{A,A}$ if and only if $\overline{0_{A,B}} = 0_{A,A}$.*

Proof. Supposing $\overline{0_{A,B}} = 0_{A,A}$ for all A and B, this directly implies $\overline{0_{A,A}} = 0_{A,A}$ for all A. In the other direction, we observe by the universal mapping property of the zero object that $0_{A,B} = 0_{A,B} \circ 0_{A,A}$, so

$$\overline{0_{A,B}} = \overline{0_{A,B} \circ 0_{A,A}} = \overline{0_{A,B} \circ \overline{0_{A,A}}} = \overline{0_{A,B}} \circ \overline{0_{A,A}} = \overline{0_{A,B}} \circ 0_{A,A} = 0_{A,A}$$

by the assumption of $\overline{0_{A,A}} = 0_{A,A}$ and the universal mapping property of the zero object. □

Given that hom-sets of restriction (and, by extension, inverse) categories are partially ordered, one may wonder when this partial order has joins. It turns out, however, that it does not in the general case, and that only very simple restriction categories have joins for arbitrary parallel morphisms. However, we *can* define a meaningful notion of joins for parallel morphisms if this operation is not required to be total, but only be defined for *compatible* morphisms. Nevertheless, these partial joins turn out to be tremendously useful, and will prove key in many of the constructions in the following sections. For restriction categories, this compatibility relation is defined as follows:

Definition 8. Parallel morphisms $f, g : A \to B$ of a restriction category $\mathscr{C}$ are said to be *restriction compatible* if $g \circ \overline{f} = f \circ \overline{g}$; if this is the case, we write $f \smile g$. By extension, a set $S \subseteq \mathrm{Hom}_{\mathscr{C}}(A, B)$ is restriction compatible, or $\smile$-compatible, if all morphisms of S are pairwise restriction compatible.

This compatibility relation can be extended to apply to inverse categories by requiring that morphisms be compatible in both directions:

Definition 9. Parallel morphisms $f, g : A \to B$ of an inverse category $\mathscr{C}$ are said to be *inverse compatible* if $f \smile g$ and $f^\dagger \smile g^\dagger$; if this is the case, we write $f \asymp g$. A set $S \subseteq \mathrm{Hom}_{\mathscr{C}}(A, B)$ is inverse compatible, or $\asymp$-compatible, if all morphisms of S are pairwise inverse compatible.

The familiar reader will notice that this definition differs in its statement from Guo's [19, p. 98], who defined $f \asymp g$ in an inverse category $\mathscr{C}$ if $f \smile g$ holds in both $\mathscr{C}$ and $\mathscr{C}^{\mathrm{op}}$ (relying on the observation that inverse categories are simultaneously restriction categories and *co*restriction categories). To avoid working explicitly with corestriction categories, however, we will use this equivalent definition instead.

We define (countable) restriction joins and (countable) join restriction categories as follows:

Definition 10 *(Guo).* Say that a restriction category $\mathscr{C}$ is outfitted with (countable) $\sim$*-joins* if for all (countable) $\sim$-compatible subsets S of all hom sets $\mathrm{Hom}_{\mathscr{C}}(A, B)$ (for a compatibility relation $\cdot \sim \cdot$), there exists a morphism $\bigvee_{s \in S} s$ such that

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

Definition 11. A restriction category is said to be a (countable) *join restriction category* if it has (countable) $\smile$-joins.

In addition, we say that a restriction functor that preserves all thus constructed joins is a *join restriction functor*. Notice that the join of the empty set (which is vacuously compatible for any compatibility relation), which we will tellingly denote $0_{A,B} : A \to B$, is always the least element with respect to the partial order on hom-sets. When a join restriction category has a restriction zero object, empty joins and zero maps coincide.

As a concrete example, **Pfn** has joins of all restriction compatible sets; here, $f \smile g$ iff whenever f and g are both defined at some point x, $f(x) = g(x)$, and the join of a set of restriction compatible partial functions F is given by

$$\left(\bigvee_{f \in F} f\right)(x) = \begin{cases} f'(x) & \text{if there exists an } f' \in F \text{ such that } f' \text{ is defined at } x, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Notice that the compatibility relation ensures precisely that the result is a partial function. This construction extends to the category **PTop** of topological spaces and partial continuous functions defined on open sets: That the resulting function is defined on an open set follows by the fact that arbitrary unions are open, and that it is continuous follows by openness and the *gluing* (or *pasting*) *lemma*. As such, **PTop** has all joins as well.

This, finally, allows us to define join inverse categories by narrowing the definition above to only require the existence of joins of inverse compatible (sets of) morphisms:

Definition 12. An inverse category is said to be a (countable) *join inverse category* if has (countable) $\asymp$-joins.

Analogously to **Pfn**, the category **PInj** is a join inverse category with joins given precisely as in **Pfn**, since the additional requirement that $f^\dagger \smile g^\dagger$ ensures that the resulting partial function is injective.

4. As DCPO-categories

In the present section, we will show that inverse categories with countable joins are intrinsically **DCPO**-enriched. This observation leads to two properties that are highly useful for modeling reversible functional programming, namely the existence of fixed points for both morphism schemes for recursion (that is, continuous endomorphisms on hom-objects) and for locally continuous functors. The former can be applied to model reversible recursive functions, and the latter to model recursive data types [33]. Further, we will show that the partial inverse of the fixed point of a morphism scheme for recursion can be computed as the fixed point of an *adjoint* morphism scheme, giving a style of recursion similar to RFUN as discussed in Section 2.

Recall that a category is **DCPO**-enriched (or simply a **DCPO**-category) if all hom-sets are pointed directed complete partial orders (*i.e.*, they have a least element and satisfy that each directed subset has a supremum), and composition is continuous and strict (recall that continuous functions are monotone by definition). For full generality, though the countable case will suffice for our applications,[4] for some cardinal κ we let $\mathbf{DCPO}_\kappa$ denote the category of pointed directed κ-complete partial orders (*i.e.*, partially ordered sets with a least element satisfying that every directed subset of cardinality at most κ has a supremum) with strict and continuous maps. To begin, we will need the lemma below:

Lemma 2. *In any inverse category, partial inversion is monotone with respect to the natural partial order in the sense that $f \leq g$ implies $f^\dagger \leq g^\dagger$.*

Proof. Suppose $f \leq g$, *i.e.*, $g \circ \overline{f} = f$ by definition. Then

$$\begin{aligned} g^\dagger \circ \overline{f^\dagger} &= g^\dagger \circ f \circ f^\dagger = g^\dagger \circ g \circ \overline{f} \circ f^\dagger = \overline{g} \circ \overline{f} \circ f^\dagger = \overline{g \circ \overline{f}} \circ f^\dagger \\ &= \overline{f} \circ f^\dagger = f^\dagger \circ f \circ f^\dagger = f^\dagger \circ \overline{f^\dagger} = f^\dagger \end{aligned}$$

so $f^\dagger \leq g^\dagger$ as well, as desired. $\square$

[4] Strictly speaking, enrichment in ω-CPOs is sufficient to show all results that follow from Theorem 4, *i.e.*, Corollary 5, Theorem 7, and Theorem 14. Notably, only countable joins (rather than arbitrary joins) are needed to obtain these results.

We also recall some basic properties of least elements of hom-sets in join inverse categories:

Lemma 3. *Let $\mathscr{C}$ be an inverse (or restriction) category with (at least empty) joins, and let $0_{A,B}$ be the least element (*i.e.*, the empty join) of $\mathrm{Hom}_{\mathscr{C}}(A,B)$. Then*

(i) $f \circ 0_{A,B} = 0_{A,Y}$ *for all* $f : B \to Y$,
(ii) $0_{A,B} \circ g = 0_{X,B}$ *for all* $g : X \to A$, *and*
(iii) $0_{A,B}$ *has a partial inverse* $0^{\dagger}_{A,B} = 0_{B,A}$.

Proof. For (i), since $0_{A,B}$ is the empty join, it follows that $f \circ 0_{A,B} = f \circ \bigvee_{s\in\emptyset} s = \bigvee_{s\in\emptyset}(f \circ s) = 0_{A,Y}$, and completely analogously for (ii). For (iii), let $0_{B,A}$ be the least element in $\mathrm{Hom}_{\mathscr{C}}(A,B)$. Then

$$0_{B,A} \circ 0_{A,B} = \left(\bigvee_{s\in\emptyset} s\right) \circ \left(\bigvee_{t\in\emptyset} t\right) = \left(\bigvee_{s\in\emptyset} s\right) \circ 0_{A,B} = \bigvee_{s\in\emptyset}(s \circ 0_{A,B}) = 0_{A,A} = \bigvee_{t\in\emptyset} \bar{t} = \overline{\bigvee_{t\in\emptyset} t} = \overline{0_{A,B}},$$

and by entirely analogous argument, $0_{A,B} \circ 0_{B,A} = 0_{B,B} = \overline{0_{B,A}}$. □

These lemmas allow us to show $\mathbf{DCPO}_{\kappa}$-enrichment (for some cardinal κ) of inverse categories with joins of directed sets of parallel morphisms with cardinality at most κ:

Theorem 4. *Every inverse (or restriction) category satisfies that if it has joins of compatible sets of cardinality at most κ respectively all joins of compatible sets, it is enriched in $\mathbf{DCPO}_{\kappa}$ respectively* **DCPO**.

Proof. Let A, B be objects of $\mathscr{C}$, and let $F \subseteq \mathrm{Hom}_{\mathscr{C}}(A,B)$ be directed (with respect to the canonical partial ordering) – of cardinality at most κ, if required. Let $f, g : A \to B$ be in F. Since F is directed, there exists an $h : A \to B$ in F such that $f \le h$ and $g \le h$, i.e., $h \circ \overline{f} = f$ and $h \circ \overline{g} = g$. But then $g \circ \overline{f} = h \circ \overline{g} \circ \overline{f} = h \circ \overline{f} \circ \overline{g} = f \circ \overline{g}$ so $f \smile g$; by Lemma 2 we have $f^{\dagger} \le h^{\dagger}$ and $g^{\dagger} \le h^{\dagger}$ as well, so $f^{\dagger} \smile g^{\dagger}$ follows entirely analogously. Thus $f \asymp g$, so F is $\asymp$-compatible, allowing us to form the supremum of F by

$$\sup F = \bigvee_{f\in F} f$$

which is the supremum of this directed set directly by definition of the join.

Monotony of compositions holds in all restriction categories, not just inverse categories with countable joins: Supposing $f \le g$ then $g \circ \overline{f} = f$, and for $h : B \to X$,

$$h \circ g \circ \overline{h \circ f} = h \circ g \circ \overline{h \circ g \circ \overline{f}} = h \circ g \circ \overline{h \circ g} \circ \overline{f} = h \circ g \circ \overline{f} = h \circ f$$

so $h \circ f \le h \circ g$ in $\mathrm{Hom}_{\mathscr{C}}(A,X)$; the argument is analogous for postcomposition. That composition is continuous follows by monotony and definition of joins, as we have

$$h \circ \sup\{f\}_{f\in F} = h \circ \bigvee_{f\in F} f = \bigvee_{f\in F}(h \circ f) = \sup\{h \circ f\}_{f\in F}$$

for all $h : B \to X$, and analogously for postcomposition. That composition is strict follows by Lemma 3. □

Recall that a functor $F : \mathscr{C} \to \mathscr{D}$ between **DCPO**-categories is *locally continuous* iff each $F_{A,B} : \mathrm{Hom}_{\mathscr{C}}(A,B) \to \mathrm{Hom}_{\mathscr{D}}(FA,FB)$ is continuous (so specifically, all locally continuous functors are locally monotone). Note that since all restriction functors preserve the partial order on hom-sets, and since suprema are defined in terms of joins, join restriction functors are in particular locally continuous.

4.1. Reversible fixed points of morphism schemes

In the following, let $\mathscr{C}$ be an inverse category with countable joins – so, by Theorem 4, enriched in $\mathbf{DCPO}_{\aleph_0}$. We will use the term *morphism scheme* to refer to a continuous function $f : \mathrm{Hom}_{\mathscr{C}}(A,B) \to \mathrm{Hom}_{\mathscr{C}}(X,Y)$ – note that such schemes are morphisms of $\mathbf{DCPO}_{\aleph_0}$ and not of the inverse category $\mathscr{C}$, so they are specifically *not* required to have inverses. Enrichment in $\mathbf{DCPO}_{\aleph_0}$ then has the following immediate corollary by Kleene's fixed point theorem:

Corollary 5. *Every morphism scheme of the form $f : \mathrm{Hom}_{\mathscr{C}}(A,B) \to \mathrm{Hom}_{\mathscr{C}}(A,B)$ has a least fixed point $\mathrm{fix}\, f : A \to B$ in $\mathscr{C}$.*

Proof. Define $\mathrm{fix}\, f = \sup\,\{f^n(0_{A,B})\}_{n\in\omega}$; this is an ω-chain, so specifically a directed set of cardinality at most $\aleph_0$. That this is the least fixed point follows by Kleene's fixed point theorem, as $0_{A,B}$ is the least element in $\mathrm{Hom}_{\mathscr{C}}(A, B)$. □

Morphism schemes on their own are useful for modeling parametrized reversible functions, as discussed in relation to Theseus in Section 2. Under this interpretation, recursive reversible functions can be seen as the least fixed points of self-parametrized reversible functions.

Given that we can thus model reversible recursive functions via least fixed points of morphism schemes, a prudent question to ask is if the inverse of a least fixed point can be computed as the least fixed point of another morphism scheme. We will answer this in the affirmative, but to do this, we need to show that the induced dagger functor is locally continuous.

Lemma 6. *The canonical dagger functor* $\dagger : \mathscr{C}^{\mathrm{op}} \to \mathscr{C}$ *is locally continuous.*

Proof. Let $F \subseteq \mathrm{Hom}_{\mathscr{C}}(A, B)$ be directed with respect to the canonical partial order. As local monotony was already shown in Lemma 2, it suffices to show that suprema are preserved. Since $f \le \bigvee_{f\in F} f$ for each $f \in F$ by Definition 12, we have $f^\dagger \le \left(\bigvee_{f\in F} f\right)^\dagger$ for all $f \in F$ by monotony of $\dagger$, and so

$$\sup\,\{f^\dagger\}_{f\in F} = \bigvee_{f\in F} f^\dagger \quad \le \quad \left(\bigvee_{f\in F} f\right)^\dagger = \sup\,\{f\}^\dagger_{f\in F}$$

by Definition 12. In the other direction, we have $f^\dagger \le \bigvee_{f\in F} f^\dagger$ for all $f \in F$ by Definition 12, so by monotony of $\dagger$, $f = f^{\dagger\dagger} \le \left(\bigvee_{f\in F} f^\dagger\right)^\dagger$ for all $f \in F$. But then $\bigvee_{f\in F} f \le \left(\bigvee_{f\in F} f^\dagger\right)^\dagger$ by Definition 12, and so by monotony of $\dagger$, we finally get

$$\sup\,\{f\}^\dagger_{f\in F} = \left(\bigvee_{f\in F} f\right)^\dagger \quad \le \quad \left(\bigvee_{f\in F} f^\dagger\right)^{\dagger\dagger} = \bigvee_{f\in F} f^\dagger = \sup\,\{f^\dagger\}_{f\in F}$$

as desired. □

With this lemma, we are able to show that the inverse of a least fixed point of a morphism scheme can be computed as the least fixed point of an adjoint morphism scheme:

Theorem 7. *Every morphism scheme of the form* $f : \mathrm{Hom}_{\mathscr{C}}(A, B) \to \mathrm{Hom}_{\mathscr{C}}(A, B)$ *has an adjoint morphism scheme* $f_\ddagger : \mathrm{Hom}_{\mathscr{C}}(B, A) \to \mathrm{Hom}_{\mathscr{C}}(B, A)$ *such that* $(\mathrm{fix}\, f)^\dagger = \mathrm{fix}\, f_\ddagger$.

Proof. Let $\iota_{A,B} : \mathrm{Hom}_{\mathscr{C}}(A, B) \to \mathrm{Hom}_{\mathscr{C}}(B, A)$ denote the family of functions defined by $\iota_{A,B}(f) = f^\dagger$; each of these are continuous by Lemma 6, and an isomorphism (with inverse $\iota_{B,A}$) by $f^{\dagger\dagger} = f$. Given a morphism scheme $f : \mathrm{Hom}_{\mathscr{C}}(A, B) \to \mathrm{Hom}_{\mathscr{C}}(A, B)$, we define $f_\ddagger = \iota_{A,B} \circ f \circ \iota_{B,A}$ – this is continuous since it is a (continuous) composition of continuous functions. But since

$$f_\ddagger^n = (\iota_{A,B} \circ f \circ \iota_{B,A})^n = \iota_{A,B} \circ f^n \circ \iota_{B,A}$$

by $\iota_{B,A}$ an isomorphism with inverse $\iota_{A,B}$, and since $0^\dagger_{A,B} = 0_{B,A}$ by Lemma 3, we get

$$\begin{aligned} \mathrm{fix}\, f_\ddagger &= \sup\,\{f_\ddagger^n(0_{B,A})\}_{n\in\omega} = \sup\,\{(\iota_{A,B} \circ f^n \circ \iota_{B,A})(0_{B,A})\}_{n\in\omega} = \sup\,\{f^n(0^\dagger_{B,A})^\dagger\}_{n\in\omega} \\ &= \sup\,\{f^n(0_{A,B})^\dagger\}_{n\in\omega} = \sup\,\{f^n(0_{A,B})\}^\dagger_{n\in\omega} = (\mathrm{fix}\, f)^\dagger \end{aligned}$$

as desired. □

In modeling recursion in reversible functional programming, this theorem states precisely that the partial inverse of a recursive reversible function is, itself, a recursive reversible function, and that it can be obtained by inverting the function body and replacing recursive calls with recursive calls to the thus constructed inverse: This is *precisely* the inverse semantics of recursive reversible functions in RFUN, as discussed in Section 2.

4.2. Algebraic ω-compactness for free!

A pleasant property of **DCPO**-categories is that algebraic ω-compactness – the property that every locally continuous functor has a canonical fixed point – is relatively easy to check, thanks to the fixed point theorem due to Adámek [33] and Barr [34]:

Theorem 8 *(Adámek and Barr). Let $\mathscr{C}$ be a* (**D**)**CPO***-category with an initial object. If $\mathscr{C}$ has colimits of ω-sequences of embeddings, then $\mathscr{C}$ is algebraically ω-compact over* (**D**)**CPO***.*

Note that this theorem was originally stated for **CPO**-categories; categories in which every hom-set is an ω-CPO (in the sense that every ω-chain of parallel morphisms has a supremum), and composition is continuous and strict. However, since ω-chains are but specific examples of directed sets, every **DCPO**-category is a **CPO**-category, and every functor that is locally continuous with respect to **DCPO**-enrichment is locally continuous with respect to **CPO**-enrichment as well. By the same argument, noting that ω-chains are directed sets of cardinality at most $\aleph_0$, it suffices to be $\mathbf{DCPO}_{\aleph_0}$-enriched.

Recall that an *embedding* in a (**D**)**CPO**-category is a morphism $e : A \to B$ with a *projection* $p : B \to A$ such that $p \circ e = 1_A$ and $e \circ p \sqsubseteq 1_B$ (as such, with the canonical (**D**)**CPO**-enrichment, embeddings in join restriction categories are specifically total – in fact, they correspond to *restriction monics* in the sense of [17]).

Canonical fixed points of functors are of particular interest in modeling functional programming, since they can be used to provide interpretations for recursive data types. In the following, we will couple this theorem with a join-completion theorem for restriction categories to show that every inverse category can be faithfully embedded in an algebraically ω-compact inverse category with joins. That this succeeds rests on the following lemmas:

Lemma 9. *There is an adjunction*

$$\begin{tikzcd} \mathbf{rCat} \arrow[r, bend left, "\mathrm{Inv}"] \arrow[r, phantom, "\top"] & \mathbf{invCat} \arrow[l, bend left, "U"] \end{tikzcd}$$

between the forgetful functor $U : \mathbf{invCat} \to \mathbf{rCat}$ and the functor $\mathrm{Inv} : \mathbf{rCat} \to \mathbf{invCat}$ that maps a restriction category to its subcategory of partial isomorphisms.

Proof. Let $F : U(\mathscr{C}) \to \mathscr{D}$ be a restriction functor between some inverse category $\mathscr{C}$ and restriction category $\mathscr{D}$. As $\mathscr{C}$ is an inverse category, the restriction category $U(\mathscr{C})$ contains only partial isomorphisms, which F has to preserve (as it is a restriction functor); as such, the image of F in $\mathscr{D}$ lies in the inverse subcategory $\mathrm{Inv}(\mathscr{D})$, so we may simply define the functor $\mathscr{C} \to \mathrm{Inv}(\mathscr{D})$ to act precisely as F on both objects and morphisms.

In the other direction, given $G : \mathscr{C} \to \mathrm{Inv}(\mathscr{D})$, we define the restriction functor $U(\mathscr{C}) \to \mathscr{D}$ by letting it act as G on both objects and morphisms; nothing further is required, as inverse categories are extensionally restriction categories.

That is determines a natural isomorphism follows immediately by the fact that neither direction actually alters how the given functor acts on objects or morphisms. □

An immediate consequence of this adjunction is that the inverse subcategory $\mathrm{Inv}(\mathscr{D})$ of $\mathscr{D}$ is uniquely determined (up to canonical isomorphism) as the largest inverse subcategory of $\mathscr{D}$, in the sense that for any other inverse category $\mathscr{C}$ with a restriction functor $G : U(\mathscr{C}) \to \mathscr{D}$, there is a unique functor $F : \mathscr{C} \to \mathrm{Inv}(\mathscr{D})$ such that the following diagram commutes

$$\begin{tikzcd} \mathrm{Inv}(\mathscr{D}) & U(\mathrm{Inv}(\mathscr{D})) \arrow[r, "\epsilon_{\mathscr{D}}"] & \mathscr{D} \\ \mathscr{C} \arrow[u, dashed, "F"] & U(\mathscr{C}) \arrow[u, "U(F)"] \arrow[ur, "G"'] & \end{tikzcd}$$

where the counit $\epsilon_{\mathscr{C}}$ is the obvious faithful inclusion functor. For our purposes, it gives the following corollary:

Corollary 10. *If an inverse category $\mathscr{C}$ embeds faithfully in a restriction category $\mathscr{D}$, it also embeds faithfully in* $\mathrm{Inv}(\mathscr{D})$*.*

Proof. By Lemma 9, it suffices to show that $F : \mathscr{C} \to \mathrm{Inv}(\mathscr{D})$ is faithful when $G : U(\mathscr{C}) \to \mathscr{D}$ is. Suppose G is faithful; by the universal mapping property, $G = \epsilon_{\mathscr{C}} \circ U(F)$ for a unique $F : \mathscr{C} \to \mathrm{Inv}(\mathscr{D})$. Since G is faithful by assumption, $U(F)$ must be faithful as well, in turn implying that F is faithful (as U is the forgetful functor). □

Lemma 11. *The functor* Inv : **rCat** $\to$ **invCat** *takes join restriction categories to join inverse categories (and preserves join restriction functors).*

Proof. It suffices to show that if S is a set of inverse compatible parallel partial isomorphisms, then $\bigvee_{s\in S} s$ is a partial isomorphism – this follows directly by continuity of partial inversion (see Lemma 6). □

The latter of these lemmas was also shown by Guo [19, Lemma 3.1.27]. As for the completion theorem, in order to ease presentation we make the following notational shorthand.

Convention 13. For a restriction category $\mathscr{C}$, let $\overline{\mathscr{C}}$ denote the category of presheaves $\mathbf{Set}^{\mathrm{Total}(\mathrm{Split}(\mathscr{C}))^{\mathrm{op}}}$.

Note that $\overline{\mathscr{C}}$ is cocomplete and all colimits are stable under pullback (since colimits in $\overline{\mathscr{C}}$ are constructed object-wise in **Set**). This is used in the completion theorem for join restriction categories, due to Cockett and Guo [19,35].

Theorem 12 *(Cockett and Guo). Every restriction category $\mathscr{C}$ can be faithfully embedded in a join restriction category of the form* $\mathrm{Par}(\overline{\mathscr{C}}, \widehat{\mathcal{M}_{gap}})$.

The stable system of monics $\widehat{\mathcal{M}_{\mathrm{gap}}}$ relates to the join-completion for restriction categories via $\mathcal{M}$-gaps (see Cockett and Guo [35] or Guo [19, Sec. 3.2.2] for details).

Lemma 13. *For a split restriction category* $\mathrm{Par}(\mathscr{C}, \mathcal{M})$*, the following are equivalent:*

(i) *Every hom-set of* $\mathrm{Par}(\mathscr{C}, \mathcal{M})$ *has a least element,*
(ii) $\mathscr{C}$ *has a strict initial object* 0 *and each morphism* $!_A : 0 \to A$ *is in* $\mathcal{M}$*, and*
(iii) $\mathrm{Par}(\mathscr{C}, \mathcal{M})$ *has a restriction zero object.*

Proof. (i) $\Leftrightarrow$ (ii): See Guo [19], Lemmas 3.3.1 and 3.3.2.

(ii) $\Rightarrow$ (iii): We will show that 0 is a restriction zero in $\mathrm{Par}(\mathscr{C}, \mathcal{M})$ with the zero map $0_{A,B} : A \to B$ given by the span $A \xleftarrow{!_A} 0 \xrightarrow{!_B} B$. Suppose that some morphism $(m, f) : A \to B$ factors through 0 in $\mathrm{Par}(\mathscr{C}, \mathcal{M})$ as $(m_2, f_2) \circ (m_1, f_1)$, *i.e.*, we are in a situation indicated by the bottom part of the diagram below in $\mathscr{C}$, where $(m_1 \circ m_2', f_2 \circ f_1') \sim (m, f)$ (*i.e.*, there exists an isomorphism α in $\mathscr{C}$ such that $m_1 \circ m_2' \circ \alpha = m$ and $f_2 \circ f_1' \circ \alpha = f$).

But since $f_1 \circ m_2' = m_2 \circ f_1' : C \to 0$ is a morphism into the strict initial object 0, it must be an isomorphism, with only possible inverse the unique map $1_C : 0 \to C$. Since $!_A = !_C \circ m_2' \circ m_1$ and $!_B = !_C \circ f_1' \circ f_2$ the universal mapping property of the initial object, it follows that the isomorphism $!_C$ witnesses $(!_A, !_B) \sim (m_1 \circ m_2', f_2 \circ f_1') \sim (m, f)$, so 0 is the zero object in $\mathrm{Par}(\mathscr{C}, \mathcal{M})$. That it is the restriction zero follows by the fact that the restriction structure on $\mathrm{Par}(\mathscr{C}, \mathcal{M})$ is given by $\overline{(m, f)} = (m, m)$, so $0_{A,A} = (!_A, !_A) = \overline{0_{A,A}}$.

$$\begin{tikzcd}
& & 0 \arrow[d, "!_C"'] \arrow[dddll, bend right, "!_A"'] \arrow[dddrr, bend left, "!_B"] & & \\
& & C \arrow[dl, hook', "m_2'"'] \arrow[dr, "f_1'"] & & \\
& A' \arrow[dl, hook', "m_1"'] \arrow[dr, "f_1"] & & B' \arrow[dl, hook', "m_2"'] \arrow[dr, "f_2"] & \\
A & & 0 & & B
\end{tikzcd}$$

(iii) $\Rightarrow$ (i): Let $0_{A,B} : A \to B$ be the unique zero map for arbitrarily chosen objects A and B, and let $f : A \to B$ be any other morphism. By Lemma 1, $\overline{0_{A,B}} = 0_{A,A}$ for a restriction zero, so $f \circ \overline{0_{A,B}} = f \circ 0_{A,A} = 0_{A,B}$ by the universal mapping property of the zero object; thus $0_{A,B} \leq f$, and hence $0_{A,B}$ is the least element in $\mathrm{Hom}(A, B)$. □

We can now show the algebraic ω-compactness theorem for restriction categories:

Theorem 14. *If $\mathscr{C}$ is a restriction category then* $\mathrm{Par}(\overline{\mathscr{C}}, \widehat{\mathcal{M}_{gap}})$ *is algebraically ω-compact over* **DCPO***, and thus over join restriction functors.*

Proof. Let $\mathscr{C}$ be a restriction category. By Theorem 12, $\mathrm{Par}(\overline{\mathscr{C}}, \widehat{\mathcal{M}_{\mathrm{gap}}})$ is a join restriction category, and since it has all joins (specifically all empty joins), it follows that it has a restriction zero object 0 (which is specifically initial) by Lemma 13. By

Adámek and Barr's fixed point theorem and the fact that restriction categories with arbitrary joins are **DCPO**-enriched (by Theorem 4), it suffices to show that $\mathrm{Par}(\overline{\mathscr{C}}, \widehat{\mathcal{M}_{\mathrm{gap}}})$ has colimits of ω-diagrams of embeddings. Let $D : \omega \to \mathrm{Par}(\overline{\mathscr{C}}, \widehat{\mathcal{M}_{\mathrm{gap}}})$ be such a diagram of embeddings. This corresponds to the diagram

$$\begin{tikzcd}
& A \arrow[dl, hook', "m_0"'] \arrow[dr, "f_0"] & & B \arrow[dl, hook', "m_1"'] \arrow[dr, "f_1"] & & {} \arrow[dl, hook'] \\
D(0) & & D(1) & & D(2) & \cdots
\end{tikzcd}$$

in $\overline{\mathscr{C}}$. Since $\overline{\mathscr{C}}$ is cocomplete, this diagram has a colimiting cocone $\alpha : D \Rightarrow \operatorname{colim} D$ such that the diagram below commutes. Further, since colimits in $\overline{\mathscr{C}}$ are constructed object-wise in **Set**, this colimit is stable under pullbacks, so the pullbacks in $\overline{\mathscr{C}}$ used for composition in $\mathrm{Par}(\overline{\mathscr{C}}, \widehat{\mathcal{M}_{\mathrm{gap}}})$ commutes with this colimit. Since embeddings are total, each m_i is an isomorphism, and since isomorphisms are stable under pullback, any m'_i arising from a pullback (corresponding to composition) is an isomorphism as well. As such, any $m_i \circ m'_{i+1}$ is an isomorphism, and so the isomorphism $m'^{(-1)}_{i+1}$ witnesses $(m_i \circ m'_{i+1}, \alpha_{D(i+2)} \circ f_{i+1} \circ f'_i) \sim (m_i, \alpha_{D(i+1)} \circ f_i)$; iterating this argument for arbitrary finite k, we see that everything commutes, as desired. Thus, the family of morphisms $\{(m_i, \alpha_{D(i+1)} \circ f_i)\}_{i \in \omega}$ is a colimiting cocone for D in $\mathrm{Par}(\overline{\mathscr{C}}, \widehat{\mathcal{M}_{\mathrm{gap}}})$.

$$\begin{tikzcd}
& & & C \arrow[dl, hook', "m'_{i+1}"'] \arrow[dr, "f'_i"] & & & \\
{} \arrow[dr] & & A \arrow[dl, hook', "m_i"'] \arrow[dr, "f_i"] & & B \arrow[dl, hook', "m_{i+1}"'] \arrow[dr, "f_{i+1}"] & & {} \arrow[dl, hook'] \\
\cdots \arrow[drrr, Rightarrow, bend right] & D(i) \arrow[drr, Rightarrow, "\alpha_{D(i)}"'] & & D(i+1) \arrow[d, Rightarrow, "\alpha_{D(i+1)}"] & & D(i+2) \arrow[dll, Rightarrow, "\alpha_{D(i+2)}"] & \cdots \arrow[dlll, Rightarrow, bend left] \\
& & & \operatorname{colim} D & & &
\end{tikzcd}$$ □

Finally, using this machinery, we can show how this theorem extends to inverse categories.

Corollary 15. *Every inverse category can be faithfully embedded in a join inverse category that is algebraically ω-compact over join restriction functors.*

Proof. Let $\mathscr{C}$ be an inverse category. Since $U(\mathscr{C})$ is the exact same category viewed as a restriction category, $U(\mathscr{C})$ embeds faithfully in $\mathrm{Par}(\overline{\mathscr{C}}, \widehat{\mathcal{M}_{\mathrm{gap}}})$, which is a join restriction category by Theorem 12, and algebraically ω-compact by Theorem 14. But then it follows by Corollary 10 that $\mathscr{C}$ embeds faithfully in $\mathrm{Inv}(\mathrm{Par}(\overline{\mathscr{C}}, \widehat{\mathcal{M}_{\mathrm{gap}}}))$, which is a join inverse category by Lemma 11, and is algebraically ω-compact for join restriction functors (which are specifically locally continuous) since fixed points of functors are (total) isomorphisms, so preserved in $\mathrm{Inv}(\mathrm{Par}(\overline{\mathscr{C}}, \widehat{\mathcal{M}_{\mathrm{gap}}}))$. □

4.3. Applications in models of reversible functional programming

This final corollary shows directly that join inverse categories are consistent with algebraic ω-compactness over join restriction functors, which can thus be used to model recursive data types in the style of Theseus as well as RFUN terms, given the existence of suitable join-preserving monoidal functors as follows: Suppose we are given an inverse product $(\otimes, 1)$ and a disjointness tensor $(\oplus, 0)$, both join-preserving (see [18] for details on both; the definition of latter can also found in Definition 17 in the following section) such that there are (total) natural isomorphisms

$$A \otimes (B \oplus C) \overset{\delta_{A,B,C}}{\cong} (A \otimes B) \oplus (A \otimes C) \qquad A \otimes 0 \overset{\nu_A}{\cong} 0$$

giving the category the structure of a *bimonoidal category* (or *rig category*; see also [36,37]). Using the compactness theorem from before, this enables us to model all (isorecursive) data types expressible in Theseus by modeling product types using the inverse product, sum types using the disjointness tensor, the unit type as 1, the empty type as 0, and recursive types using the canonical fixed point. For example, the two types from the Theseus example in Fig. 2, defined as

type Bool = False | True
type Nat = 0 | Succ Nat

can both be given very familiar denotations as $1 \oplus 1$ respectively $\mu X.1 \oplus X$ (*i.e.*, the least – in this case, unique – fixed point of the functor defined by $F(X) = 1 \oplus X$).

Though RFUN is untyped, with terms formed using Lisp-style symbols and constructors, this bimonoidal structure is also helpful in constructing a denotation of the universal type of RFUN terms. Formally, RFUN terms are constructed inductively as follows: Given a denumerable set $\mathbb{S}$ of symbols, a term t is either a symbol (*i.e.*, $t \in \mathbb{S}$) or of the form $c(t_1, t_2, \dots, t_k)$ for some finite k, where $c \in \mathbb{S}$ and each t_i is a term. Supposing that $\mathbb{S}$ is the Kleene star closure of the latin alphabet, examples of terms include a, $s(s(z))$, and $f(o, o(bar, baz))$.

Since the symbols $\mathbb{S}$ is denumerable, we can identify symbols one-to-one with natural numbers – and so with the object $\mu X.1 \oplus X$ using algebraic compactness. Further, in the familiar way, we can define the functor mapping an object A to lists of A by $A \mapsto \mu X.1 \oplus (A \otimes X)$. This, finally, means that we can define a functor mapping an object A to terms over A by $A \mapsto \mu X.A \otimes L(X)$.

Moreover, we saw how this view of join restriction categories as **DCPO**-categories also enabled the style of reversible recursion found in RFUN, exemplified in the Fibonacci-pair program in Fig. 1. More specifically, to give the denotation of a recursive function in RFUN, instead of trying to interpret it directly as an endomorphism on the term object $T \to T$, we instead interpret it as a morphism scheme $\mathrm{Hom}(T,T) \xrightarrow{[\![f']\!]} \mathrm{Hom}(T,T)$ in which the recursive call is replaced by a call to the morphism given as argument, and then take the least fixed point of this morphism scheme fix $[\![f']\!]$ as the denotation of the recursive function f. Further details on this construction, and the other constructions sketched in this section, will appear in a forthcoming paper.

5. As unique decomposition categories

Complementary to the view on inverse categories with countable joins as **DCPO**-categories, we will show that these can also be viewed as unique decomposition categories, a kind of category introduced by Haghverdi [20] equipped with a partial sum operation on hom-sets via enrichment in the category of Σ-monoids (shown to be symmetric monoidal by Hoshino [21]). Unique decomposition categories (including Hoshino's *strong* unique decomposition categories [21] which we will employ here) are specifically traced monoidal categories [38] if they satisfy certain conditions. This is desirable when modeling functional programming, as traces can be used to model notions of feedback [39] and recursion [40–42].

Here, we will show that inverse categories with countable joins and a join-preserving *disjointness tensor* (due to Giles [18]) are strong unique decomposition categories, and satisfy the conditions required to be equipped with a trace. We extend this result further to show that the trace is a †-trace [43], and thus has pleasant inversion properties (the trace in **PInj** is well studied, *cf.* [44,20,45]). This is particularly interesting given that the reversible programming language Theseus [15] and the combinator calculus Π^0 [16] both rely on a †-trace for reversible recursion.

We begin with the definition of a Σ-monoid [20] (see also Manes and Benson [46] where these first appeared as *positive partial monoids*):

Definition 14. A Σ-*monoid* (M, Σ) consists of a nonempty set M and a partial operator Σ defined on countable families in M (say that a family $\{x_i\}_{i\in I}$ is *summable* if $\sum_{i\in I} x_i$ is defined) such that

(i) if $\{x_i\}_{i\in I}$ is a countable family in M and $\{I_j\}_{j\in J}$ is a countable partitioning of I, then $\{x_i\}_{i\in I}$ is summable iff all $\{x_i\}_{i\in I_j}$ and $\sum_{i\in I_j} x_i$ are summable for all $j \in J$, and in this case

$$\sum_{j\in J}\sum_{i\in I_j} x_i = \sum_{i\in I} x_i\,,$$

(ii) any family $\{x_i\}_{i\in I}$ in M where I is singleton is summable with $\sum_{i\in I} x_i = x_j$ if $I = \{j\}$.

The class of Σ-monoids with homomorphisms preserving partial sums forms a category, Σ**Mon**. As such, a category $\mathscr{C}$ is enriched in Σ**Mon** if all hom-sets of $\mathscr{C}$ are Σ-monoids, and composition distributes over partial addition.

Lemma 16. *Every inverse category with countable joins is* Σ**Mon**-*enriched.*

Proof. Let $\mathscr{C}$ be an inverse category with countable joins. Let $\{s_i\}_{i\in I}$ be a countable family of morphisms taken from $\mathrm{Hom}_{\mathscr{C}}(A,B)$ for some objects A, B of $\mathscr{C}$ and countable index set I. We define

$$\sum_{i\in I} s_i = \bigvee_{s\in\{s_i \mid i\in I\}} s$$

so, by definition, summability coincides with join compatibility.

To see that axiom (i) of Definition 14 is satisfied, let $\{I_j\}_{j\in J}$ be a partitioning of I and suppose that $\{s_i\}_{i\in I}$ is summable, *i.e.*, inverse compatible. By definition of inverse compatibility, this means that all morphisms of $\{s_i\}_{i\in I}$ are pairwise inverse compatible, and since all partition families $\{s_i\}_{i \in I_j}$ for $j \in J$ consist only of morphisms taken from $\{s_i\}_{i \in I}$, they are summable by all $\{s_i \mid i \in I_j\}$ inverse compatible; that

$$\sum_{j \in J} \sum_{i \in I_j} x_i = \sum_{i \in I} x_i$$

follows by the least upper bound property of the join (Definition 12(i)).

Conversely, suppose that all $\{s_i\}_{i \in I_j}$ and all $\sum_{i \in I_j} s_i$ are summable for all $j \in J$. Let f and g be arbitrary morphisms of $\{s_i\}_{i \in I}$; then, f is an element of a partition $\{s_i\}_{i \in I_j}$ for $j \in J$, and g is an element of a partition $\{s_i\}_{i \in I_k}$ for $k \in J$. If $j = k$ then f and g are inverse compatible by $\{s_i\}_{i \in I_j}$ summable – if $j \neq k$, we have $f \leq \bigvee_{s \in \{s_i \mid i \in I_j\}} s = \sum_{i \in I_j} s_i$ so f and $\bigvee_{s \in \{s_i \mid i \in I_j\}} s$ are inverse compatible by Lemma 2, and g and $\bigvee_{s \in \{s_i \mid i \in I_k\}} s$ are inverse compatible by an analogous argument. But then f and g are inverse compatible by $\bigvee_{s \in \{s_i \mid i \in I_j\}} s = \sum_{i \in I_j} s_i$ and $\bigvee_{s \in \{s_i \mid i \in I_k\}} s = \sum_{i \in I_k} s_i$ summable (*i.e.*, inverse compatible) by assumption, and by transitivity of join compatibility. The summation identity follows, once again, using Definition 12(i).

For axiom (ii) of Definition 14, this follows by $f \leq f$ for any morphism $f : A \to B$, and so for a singleton $F = \{f\}$ (and such a singleton always exists, since every hom-set has a least element given by the empty join), $f \leq \bigvee_{s \in F} s$ and $\bigvee_{s \in F} s \leq f$ both follow by Definition 12(i), so $f = \bigvee_{s \in F} s$. That composition distributes over partial addition follows directly by Definition 12(iii) and (iv). $\square$

Haghverdi defines unique decomposition categories in the following way:

Definition 15 *(Haghverdi).* A unique decomposition category $\mathscr{C}$ is a symmetric monoidal category enriched in $\Sigma\mathbf{Mon}$ such that for all finite index sets I and all $j \in I$, there are *quasi-injections* $\iota_j : X_j \to \oplus_{i \in I} X_i$ and *quasi-projections* $\rho_j : \oplus_{i \in I} X_i \to X_j$ satisfying

(i) $\rho_k \circ \iota_j = 1_{X_k}$ if $j = k$, and $0_{X_j, X_k}$ otherwise, and
(ii) $\Sigma_{i \in I} \iota_i \circ \rho_i = 1_{\oplus_{i \in I} X_i}$.

By slight abuse of notation, we will use $0_{A,B} : A \to B$ to denote the morphism arising from summing the empty family of $\mathrm{Hom}_{\mathscr{C}}(A, B)$. That the empty family is always summable – and that its sum serves as unit – follows from axioms (i) *and* (ii) of Definition 14, as (i) and the assumption of nonemptiness guarantees the summability of at least one family, while (i) ensures that any partitioning of this family – including into empty partitions – is summable when the family is (so the empty family is summable), and that the sum of summed partitions coincides with the sum of the original family (giving us that the sum of the empty family is the unit; see also [46] or [20]). This allows us to state the strengthened definition by Hoshino:

Definition 16 *(Hoshino).* A strong unique decomposition category is a symmetric monoidal category enriched in $\Sigma\mathbf{Mon}$ satisfying that the identity on the monoidal unit I is $0_{I,I}$, and that

$$1_X \oplus 0_{Y,Y} + 0_{X,X} \oplus 1_Y = 1_{X \oplus Y}$$

for all X and Y.

An elementary result is that strong unique decomposition categories are unique decomposition categories, with their quasi injections and quasi projections given by $\iota_1 = (1_A \oplus 0_{0,B}) \circ u_r^{-1} : A \to A \oplus B$ and $\rho_1 = u_r \circ (1_A \oplus 0_{B,0}) : A \oplus B \to A$ (where $u_r : A \oplus 0 \to A$ is the right unitor of the monoidal functor $- \oplus -$), and analogously for ι_2 and ρ_2 (thus extending to any finite index set).

As such, (strong) unique decomposition categories rely on a sum-like monoidal tensor – in the context of inverse categories, such a one can be found in Giles' definition of a disjointness tensor [18, Definition 7.2.1].

Definition 17 *(Giles).* An inverse category $\mathscr{C}$ with a restriction zero object 0 is said to have a *disjointness tensor* if it is equipped with a symmetric monoidal restriction functor $- \oplus - : \mathscr{C} \times \mathscr{C} \to \mathscr{C}$ such that

(i) the restriction zero 0 is the tensor unit, and
(ii) the morphisms $\amalg_1 : A \to A \oplus B$ and $\amalg_2 : A \to B \oplus A$ given by $\amalg_1 = (1_A \oplus 0_{0,B}) \circ u_r^{-1}$ and $\amalg_2 = (0_{0,B} \oplus 1_A) \circ u_l^{-1}$ are jointly epic, and their partial inverses $\amalg_1^\dagger : A \oplus B \to A$ and $\amalg_2^\dagger : B \oplus A \to A$ are jointly monic,

where $u_l : 0 \oplus A \to A$ and $u_r : A \oplus 0 \to A$ denote the left respectively the right unitor of the symmetric monoidal tensor.

Though not required from this definition, since we are working exclusively with join inverse categories, we make the additional assumption that the disjointness tensor is a join restriction functor. Since Giles' definition already demands that the zero object be the monoidal unit, and even defines $\amalg_i$ precisely like Hoshino's definition of ι_i (one can similarly see that $\amalg_i^\dagger = \rho_i$), we can show the following:

Theorem 17. *Every inverse category with countable joins and a join-preserving disjointness tensor is a strong unique decomposition category.*

Proof. By Lemma 16, any inverse category with countable joins (and a join-preserving disjointness tensor) is enriched in Σ**Mon**, so it suffices to show that the (specifically symmetric monoidal) disjointness tensor satisfies Definition 16. That $1_{I,I} = 0_{I,I}$ follows by $1_{0,0} = 0_{0,0}$ for the (restriction) zero 0, and $1_X \oplus 0_{Y,Y} + 0_{X,X} \oplus 1_Y = 1_{X\oplus Y}$ by the definition of partial sums as joins and the additional requirement that the disjointness tensor preserves joins. $\square$

Due to the Σ**Mon**-enrichment on unique decomposition categories, the trace can be constructed as a denumerable sum of morphisms, provided that morphisms of a certain form are always summable, *cf.* [20, Prop. 4.0.11] and [21, Corr. 5.4]:

Theorem 18 *(Haghverdi, Hoshino). Let $\mathscr{C}$ be a (strong) unique decomposition category such that for every X, Y, and U and every $f : X \oplus U \to Y \oplus U$, the sum $f_{11} + \sum_{n=0}^{\infty} f_{21} \circ f_{22}^n \circ f_{12}$ exists, where $f_{ij} = \rho_j \circ f \circ \iota_i$. Then $\mathscr{C}$ has a uniform trace given by*

$$\mathrm{Tr}_{X,Y}^U(f) = f_{11} + \sum_{n=0}^{\infty} f_{21} \circ f_{22}^n \circ f_{12}\,.$$

In a restriction category, we say that parallel morphisms $f, g : A \to B$ are *disjoint* if $\overline{f} \circ \overline{g} = 0_{A,A}$.

Lemma 19. *In a restriction category, the following hold:*

(i) *All disjoint morphisms are restriction compatible,*
(ii) *$g \smile g'$ and $f \smile f'$ implies $g \circ f \smile g' \circ f'$ when* $\mathrm{dom}(g) = \mathrm{cod}(f)$, *and*
(iii) *$\overline{g \circ f} = \overline{g} \circ f \circ \overline{f}$ when* $\mathrm{dom}(g) = \mathrm{cod}(f)$.

Proof. For (i), suppose $f, g : A \to B$ are disjoint, *i.e.*, $\overline{f} \circ \overline{g} = 0_{A,A}$. Then

$$g \circ \overline{f} = g \circ \overline{g} \circ \overline{f} = g \circ \overline{f} \circ \overline{g} = g \circ 0_{A,A} = 0_{A,B} = f \circ 0_{A,A} = f \circ \overline{f} \circ \overline{g} = f \circ \overline{g}.$$

Part (ii) was shown by Guo [19, Lemma 3.1.3]. For (iii) we have $\overline{g \circ f} = \overline{g \circ f \circ \overline{f}} = \overline{g \circ f} \circ \overline{f}$, as desired. $\square$

This lemma allows us to show that all join inverse categories are traced monoidal categories with a uniform trace.

Theorem 20. *Every inverse category with countable joins and a disjointness tensor has a uniform trace.*

Proof. By Theorem 18, it suffices to show that all morphisms of the forms f_{11} or $f_{21} \circ f_{22}^n \circ f_{12}$ for any $n \in \mathbb{N}$ and some $f : X \oplus U \to Y \oplus U$ are pairwise inverse compatible. We notice that

$$(f_{ij})^\dagger = (\rho_j \circ f \circ \iota_i)^\dagger = (\amalg_j^\dagger \circ f \circ \amalg_i)^\dagger = \amalg_i^\dagger \circ f^\dagger \circ \amalg_j^{\dagger\dagger} = \amalg_i^\dagger \circ f^\dagger \circ \amalg_j = (f^\dagger)_{ji}$$

and so $(f_{11})^\dagger = (f^\dagger)_{11}$ and

$$(f_{21} \circ f_{22}^n \circ f_{12})^\dagger = (f_{12})^\dagger \circ (f_{22}^n)^\dagger \circ (f_{21})^\dagger = (f^\dagger)_{21} \circ (f_{22}^\dagger)^n \circ (f^\dagger)_{12}$$

so it suffices to show only restriction compatibility, since the restriction compatibility of the partial inverses will follow directly by this symmetry.

To see that $f_{11} \smile f_{21} \circ f_{22}^k \circ f_{12}$ for some $k \in \mathbb{N}$, notice that $f_{11} = \amalg_1^\dagger \circ f \circ \amalg_1$ and

$$f_{21} \circ f_{22}^k \circ f_{12} = f_{21} \circ f_{22}^k \circ \amalg_2^\dagger \circ f \circ \amalg_1$$

so it suffices by Lemma 19 to show that $\amalg_1^\dagger \smile f_{21} \circ f_{22}^n \circ \amalg_2^\dagger$. But then

$$\overline{f_{21} \circ f_{22}^n \circ \amalg_2^\dagger} \circ \overline{\amalg_1^\dagger} = \overline{f_{21} \circ f_{22}^n \circ \amalg_2^\dagger} \circ \overline{\amalg_2^\dagger} \circ \overline{\amalg_1^\dagger} = 0_{Y\oplus U, Y\oplus U}$$

since $\overline{\amalg_1^\dagger} = \amalg_1 \circ \amalg_1^\dagger = 1_Y \oplus 0_{U,U}$ and $\overline{\amalg_2^\dagger} = \amalg_2 \circ \amalg_2^\dagger = 0_{Y,Y} \oplus 1_U$, so these are restriction compatible by Lemma 19.

To see that $f_{21} \circ f_{22}^m \circ f_{12} \smile f_{21} \circ f_{22}^n \circ f_{12}$, assume without loss of generality that $m < n$ (the case where $m = n$ is trivial). Once again, by Lemma 19, it suffices to show $f_{21} \smile f_{21} \circ f_{22}^{n-m}$. But since

$$f_{21} = \amalg_1^\dagger \circ f \circ \amalg_2$$

and

$$f_{21} \circ f_{22}^{n-m} = f_{21} \circ (\amalg_2^\dagger \circ f \circ \amalg_2)^{(n-m)-1} \circ \amalg_2^\dagger \circ f \circ \amalg_2$$

restriction compatibility follows by analogous argument to the previous case. □

Recall that a †-category with a trace is said to have a *†-trace* (see, *e.g.*, [43]) if $\mathrm{Tr}^U_{X,Y}(f)^\dagger = \mathrm{Tr}^U_{Y,X}(f^\dagger)$ for every morphism $f : X \oplus U \to Y \oplus U$.

Theorem 21. *The canonical trace in an inverse category with countable joins and a disjointness tensor is a †-trace.*

Proof. To see that the trace induced by Theorems 18 and 20 is a †-trace, let $f : X \oplus U \to Y \oplus U$ be a morphism of $\mathscr{C}$.

In the proof of Theorem 20, we noticed that $(f_{ij})^\dagger = (f^\dagger)_{ji}$ and $(f_{21} \circ f_{22}^n \circ f_{12})^\dagger = (f^\dagger)_{21} \circ (f_{22}^\dagger)^n \circ (f^\dagger)_{12}$. Expanding this in the definition of the canonical trace given by Theorem 18, we get

$$\begin{aligned}
\mathrm{Tr}^U_{X,Y}(f)^\dagger &= \left(f_{11} + \sum_{n\in\omega} f_{21} \circ f_{22}^n \circ f_{12}\right)^\dagger = \left(f_{11} \vee \bigvee_{n\in\omega} f_{21} \circ f_{22}^n \circ f_{12}\right)^\dagger \\
&= (f_{11})^\dagger \vee \left(\bigvee_{n\in\omega} f_{21} \circ f_{22}^n \circ f_{12}\right)^\dagger = (f_{11})^\dagger \vee \bigvee_{n\in\omega} (f_{12})^\dagger \circ (f_{22}^n)^\dagger \circ (f_{21})^\dagger \\
&= (f^\dagger)_{11} \vee \bigvee_{n\in\omega} (f^\dagger)_{21} \circ (f^\dagger)_{22}^n \circ (f^\dagger)_{12} = \mathrm{Tr}^U_{Y,X}(f^\dagger)
\end{aligned}$$

by definition of the partial sum as join (Lemma 16), and by $(\bigvee_{f\in F} f)^\dagger = \bigvee_{f\in F} f^\dagger$ by Lemma 6. □

5.1. Applications in models of reversible functional programming

This final theorem is highly relevant to modeling Theseus in join inverse categories, as the *iteration label*-approach to reversible tail recursion (exemplified in the parity program in Fig. 2) is equivalent to the existence of a †-trace operator. This can be observed from the fact that we are not only able to provide a forward and backward semantics to functions with iteration labels via a †-trace [15] (see also the discussion in Section 2), but that it is also possible to express a †-trace operator as a parametrized function (which, in turn, can be naturally regarded categorically as a morphism scheme) in Theseus [15].

To give a concrete example, consider the recursive parity function in Theseus from Fig. 2. To give semantics to its recursive behavior using a †-trace, we systematically transform from a function of type $\underline{\mathrm{Nat}} \times \underline{\mathrm{Bool}} \leftrightarrow \underline{\mathrm{Nat}} \times \underline{\mathrm{Bool}}$ with an internal iteration label of type $\underline{\mathrm{Nat}} \times \underline{\mathrm{Nat}} \times \underline{\mathrm{Bool}} \leftrightarrow \underline{\mathrm{Nat}} \times \underline{\mathrm{Nat}} \times \underline{\mathrm{Bool}}$ into a function of type $(\underline{\mathrm{Nat}} \times \underline{\mathrm{Bool}}) + (\underline{\mathrm{Nat}} \times \underline{\mathrm{Nat}} \times \underline{\mathrm{Bool}}) \leftrightarrow (\underline{\mathrm{Nat}} \times \underline{\mathrm{Bool}}) + (\underline{\mathrm{Nat}} \times \underline{\mathrm{Nat}} \times \underline{\mathrm{Bool}})$ by prefacing patterns for the outer (parity) function by **Left**, replacing patterns for the inner (iteration label) function by **Right**-patterns, replacing calls to the inner function by **Right**-expressions, and prefacing return values by **Left**-expressions, as in the following:

$$\begin{array}{l}
\mathit{parity} \ :: \ \underline{\mathrm{Nat}} \times \underline{\mathrm{Bool}} \leftrightarrow \underline{\mathrm{Nat}} \times \underline{\mathrm{Bool}} \\
\begin{array}{lll}
\mid \ (n, b) & \leftrightarrow & \mathit{iter}\ (n, 0, b) \\
\mid \ \mathit{iter}\ (\mathrm{Succ}\ n, m, b) & \leftrightarrow & \mathit{iter}\ (n, \mathrm{Succ}\ m, \mathrm{not}\ b) \\
\mid \ \mathit{iter}\ (0, m, b) & \leftrightarrow & (m, b)
\end{array} \\
\quad \textbf{where}\ \mathit{iter} :: \underline{\mathrm{Nat}} \times \underline{\mathrm{Nat}} \times \underline{\mathrm{Bool}} \leftrightarrow \\
\qquad\qquad \underline{\mathrm{Nat}} \times \underline{\mathrm{Nat}} \times \underline{\mathrm{Bool}}
\end{array}
\quad \Rightarrow \quad
\begin{array}{l}
\mathit{parity}' \ :: \ (\underline{\mathrm{Nat}} \times \underline{\mathrm{Bool}}) + (\underline{\mathrm{Nat}} \times \underline{\mathrm{Nat}} \times \underline{\mathrm{Bool}}) \leftrightarrow \\
\qquad\qquad (\underline{\mathrm{Nat}} \times \underline{\mathrm{Bool}}) + (\underline{\mathrm{Nat}} \times \underline{\mathrm{Nat}} \times \underline{\mathrm{Bool}}) \\
\begin{array}{lll}
\mid \ \textbf{Left}\ (n, b) & \leftrightarrow & \textbf{Right}\ (n, 0, b) \\
\mid \ \textbf{Right}\ (\mathrm{Succ}\ n, m, b) & \leftrightarrow & \textbf{Right}\ (n, \mathrm{Succ}\ m, \mathrm{not}\ b) \\
\mid \ \textbf{Right}\ (0, m, b) & \leftrightarrow & \textbf{Left}\ (m, b)
\end{array}
\end{array}$$

Notice that this transformation preserves non-overlapping and exhaustive patterns, as are required of Theseus functions. In this way, we can obtain the denotation of the original *parity* function by taking the †-trace of the denotation of the transformed *parity*′ function.

6. Conclusion

We have shown that inverse categories with countable joins carry with them a few key properties that are highly useful for modeling partial reversible functional programming. Notably, we have shown that any inverse category with countable joins is **DCPO**-enriched – from this view, we gathered that morphism schemes have fixed points, and that the partial inverses of such fixed points can be computed as fixed points of adjoint morphism schemes. This gave us a model of recursion à la RFUN.

Further, we were able to show that any inverse category can be embedded in an inverse category with joins, in which all join restriction functors have canonical fixed points. Finally, we showed that the presence of a join-preserving disjointness tensor on an inverse category with countable joins gives us a strong unique decomposition category, and in turn, a uniform †-trace: a model of recursion à la Theseus and Π^0.

Restriction categories have recently been considered as enriched categories by Cockett and Garner [47], though their approach relied on enrichments based on *weak double categories* rather than monoidal categories, as it is otherwise usually done (including in this paper). Further, fixed points in categories with a notion of partiality have previously been considered, notably by Fiore [48] who also relied on order-enrichment, though his work was in categories of partial maps directly. Finally, Giles [18] has shown the construction of a trace in inverse categories recently, relying instead on the presence of countable *disjoint sums* rather than joins (whether or not this approach leads to a †-trace is unspecified). It should also be noted that the trace in the canonical inverse category **PInj** has been studied independently of unique decomposition and restriction categories, notably by Hines [44] and Abramsky, Haghverdi, and Scott [45].

As regards future work, since an inverse category with countable joins and a disjointness tensor is †-traced, it can be embedded in a †-compact closed category via the Int-construction [38,49]. It may be of interest to consider †-compact closed categories generated in this manner, as we suspect these will be inverse categories as well (notably, Int(**PInj**) is [44]) – and could provide, *e.g.*, an alternative treatment of projectors as restriction idempotents, and isometries as restriction monics (see also [50]).

Additionally, while our focus in this article has been on inverse categories, we conjecture that many of these results can be generalized to restriction categories.

Acknowledgements

The authors wish to thank the anonymous reviewers for their thoughtful and detailed comments, and the anonymous reviewers of FoSSaCS 2016 as well as the participants of NWPT 2015 for their useful comments on earlier versions of the paper. The research was partly funded by the Danish Council for Independent Research under the *Foundations of Reversible Computing* project. We also acknowledge the support given by COST *Action* IC1405 *Reversible Computation: Extending Horizons of Computing.*

References

[1] R. Kaarsgaard, Join inverse categories and reversible recursion, 2015, abstract presented at the 27th Nordic Workshop on Programming Theory.
[2] H.B. Axelsen, R. Kaarsgaard, Join inverse categories as models of reversible recursion, in: B. Jacobs, C. Löding (Eds.), Foundations of Software Science and Computation Structures, 19th International Conference, FoSSaCS 2016, Proceedings, in: Lecture Notes in Computer Science, vol. 9634, Springer, 2016, pp. 73–90.
[3] R. Landauer, Irreversibility and heat generation in the computing process, IBM J. Res. Dev. 5 (3) (1961) 183–191.
[4] E. Fredkin, T. Toffoli, Conservative logic, Int. J. Theor. Phys. 21 (3–4) (1982) 219–253.
[5] C.H. Bennett, Logical reversibility of computation, IBM J. Res. Dev. 17 (6) (1973) 525–532.
[6] H.B. Axelsen, R. Glück, What do reversible programs compute?, in: M. Hofmann (Ed.), Foundations of Software Science and Computational Structures, 14th International Conference, FOSSACS 2011, Proceedings, in: Lecture Notes in Computer Science, vol. 6604, Springer, 2011, pp. 42–56.
[7] M. Kutrib, M. Wendlandt, Reversible limited automata, in: J. Durand-Lose, B. Nagy (Eds.), Machines, Computations, and Universality, 7th International Conference, MCU 2015, Proceedings, in: Lecture Notes in Computer Science, vol. 9288, Springer, 2015, pp. 113–128.
[8] K. Morita, Two-way reversible multihead automata, Fundam. Inform. 110 (1–4) (2011) 241–254.
[9] M. Schordan, D. Jefferson, P. Barnes, T. Oppelstrup, D. Quinlan, Reverse code generation for parallel discrete event simulation, in: J. Krivine, J.-B. Stefani (Eds.), Reversible Computation, 7th International Conference, RC 2015, Proceedings, in: Lecture Notes in Computer Science, vol. 9138, Springer, 2015, pp. 95–110.
[10] I. Cristescu, J. Krivine, D. Varacca, A compositional semantics for the reversible π-calculus, in: LICS 2013, IEEE Computer Society, 2013, pp. 388–397.
[11] U.P. Schultz, M. Bordignon, K. Støy, Robust and reversible execution of self-reconfiguration sequences, Robotica 29 (1) (2011) 35–57.
[12] U.P. Schultz, J.S. Laursen, L. Ellekilde, H.B. Axelsen, Towards a domain-specific language for reversible assembly sequences, in: J. Krivine, J.-B. Stefani (Eds.), Reversible Computation, 7th International Conference, RC 2015, Proceedings, in: Lecture Notes in Computer Science, vol. 9138, Springer, 2015, pp. 111–126.
[13] T. Yokoyama, R. Glück, A reversible programming language and its invertible self-interpreter, in: Partial Evaluation and Program Manipulation. Proceedings, ACM, 2007, pp. 144–153.
[14] T. Yokoyama, H.B. Axelsen, R. Glück, Fundamentals of reversible flowchart languages, Theor. Comput. Sci. 611 (2016) 87–115.
[15] R.P. James, A. Sabry, Theseus: A high level language for reversible computing, work-in-progress report at RC, 2014, available at https://www.cs.indiana.edu/~sabry/papers/theseus.pdf, 2014.
[16] W.J. Bowman, R.P. James, A. Sabry, Dagger traced symmetric monoidal categories and reversible programming, in: A. De Vos, R. Wille (Eds.), Reversible Computation 2011, Proceedings, Ghent University, 2011, pp. 51–56.
[17] J.R.B. Cockett, S. Lack, Restriction categories I: categories of partial maps, Theor. Comput. Sci. 270 (1–2) (2002) 223–259.
[18] B.G. Giles, An investigation of some theoretical aspects of reversible computing, Ph.D. thesis, University of Calgary, 2014.
[19] X. Guo, Products, joins, meets, and ranges in restriction categories, Ph.D. thesis, University of Calgary, 2012.
[20] E. Haghverdi, A categorical approach to linear logic, geometry of proofs and full completeness, Ph.D. thesis, Carlton University and University of Ottawa, 2000.
[21] N. Hoshino, A representation theorem for unique decomposition categories, in: U. Berger, M. Mislove (Eds.), MFPS XXVIII, in: Electronic Notes in Theoretical Computer Science, vol. 286, Elsevier, 2012, pp. 213–227.
[22] T. Yokoyama, H.B. Axelsen, R. Glück, Towards a reversible functional language, in: A. De Vos, R. Wille (Eds.), Reversible Computation, Third International Workshop, RC 2011, Revised papers, in: Lecture Notes in Computer Science, vol. 7165, Springer, 2012, pp. 14–29.
[23] S.M. Abramov, R. Glück, Principles of inverse computation and the universal resolving algorithm, in: T.Æ. Mogensen, D.A. Schmidt, I.H. Sudborough (Eds.), The Essence of Computation: Complexity, Analysis, Transformation, in: Lecture Notes in Computer Science, vol. 2566, Springer, 2002, pp. 269–295.
[24] R. Glück, M. Kawabe, Derivation of deterministic inverse programs based on LR parsing, in: Y. Kameyama, P.J. Stuckey (Eds.), Functional and Logic Programming, 7th International Symposium, FLOPS 2004, Proceedings, in: Lecture Notes in Computer Science, vol. 2998, Springer, 2004, pp. 291–306.

[25] R. Glück, M. Kawabe, A program inverter for a functional language with equality and constructors, in: A. Ohori (Ed.), Programming Languages and Systems, First Asian Symposium, APLAS 2003, Proceedings, in: Lecture Notes in Computer Science, vol. 2895, Springer, 2003, pp. 246–264.
[26] R.P. James, A. Sabry, Information effects, in: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '12, ACM, 2012, pp. 73–84.
[27] J.R.B. Cockett, S. Lack, Restriction categories II: partial map classification, Theor. Comput. Sci. 294 (1–2) (2003) 61–102.
[28] J.R.B. Cockett, S. Lack, Restriction categories III: colimits, partial limits and extensivity, Math. Struct. Comput. Sci. 17 (4) (2007) 775–817.
[29] M.V. Lawson, Inverse Semigroups: The Theory of Partial Symmetries, World Scientific, 1998.
[30] J. Kastl, Inverse categories, in: H.-J. Hoehnke (Ed.), Algebraische Modelle, Kategorien und Gruppoide, in: Studien zur Algebra und Ihre Anwendungen, vol. 7, Akademie-Verlag, 1979, pp. 51–60.
[31] C. Heunen, On the functor ℓ^2, in: Computation, Logic, Games, and Quantum Foundations – The Many Facets of Samson Abramsky, in: Lecture Notes in Computer Science, vol. 7860, Springer, 2013, pp. 107–121.
[32] E. Robinson, G. Rosolini, Categories of partial maps, Inf. Comput. 79 (1988) 95–130.
[33] J. Adámek, Recursive data types in algebraically ω-complete categories, Inf. Comput. 118 (1995) 181–190.
[34] M. Barr, Algebraically compact functors, J. Pure Appl. Algebra 82 (3) (1992) 211–231.
[35] J.R.B. Cockett, X. Guo, Join restriction categories and the importance of being adhesive, presentation at Category Theory (2007), 2007.
[36] M.L. Laplaza, Coherence for distributivity, in: G.M. Kelly, M.L. Laplaza, G. Lewis, S. Mac Lane (Eds.), Coherence in Categories, in: Lecture Notes in Mathematics, vol. 281, Springer, 1972, pp. 29–65.
[37] J. Carette, A. Sabry, Computing with semirings and weak rig groupoids, in: P. Thiemann (Ed.), Programming Languages and Systems, 25th European Symposium on Programming, Proceedings, in: Lecture Notes in Computer Science, vol. 9632, Springer, 2016, pp. 123–148.
[38] A. Joyal, R. Street, D. Verity, Traced monoidal categories, Math. Proc. Camb. Philos. Soc. 119 (3) (1996) 447–468.
[39] S. Abramsky, Retracing some paths in process algebra, in: U. Montanari, V. Sassone (Eds.), CONCUR '96: Concurrency Theory, 7th International Conference, Proceedings, in: Lecture Notes in Computer Science, vol. 1119, Springer, 1996, pp. 1–17.
[40] M. Hasegawa, Recursion from cyclic sharing: traced monoidal categories and models of cyclic lambda calculi, in: P. de Groote, J.R. Hindley (Eds.), Typed Lambda Calculi and Applications, Third International Conference on Typed Lambda Calculi and Applications, TLCA '97, Proceedings, in: Lecture Notes in Computer Science, vol. 1210, Springer, 1997, pp. 196–213.
[41] M. Hasegawa, Models of sharing graphs, Ph.D. thesis, University of Edinburgh, 1997.
[42] M. Hyland, Abstract and concrete models for recursion, in: O. Grumberg, T. Nipkow, C. Pfaller (Eds.), Proceedings of the NATO Advanced Study Institute on Formal Logical Methods for System Security and Correctness, IOS Press, 2008, pp. 175–198.
[43] P. Selinger, A survey of graphical languages for monoidal categories, in: B. Coecke (Ed.), New Structures for Physics, in: Lecture Notes in Physics, vol. 813, Springer, 2011, pp. 289–355.
[44] P.M. Hines, The algebra of self-similarity and its applications, Ph.D. thesis, University of Wales, Bangor, 1998.
[45] S. Abramsky, E. Haghverdi, P. Scott, Geometry of Interaction and linear combinatory algebras, Math. Struct. Comput. Sci. 12 (05) (2002) 625–665.
[46] E.G. Manes, D.B. Benson, The inverse semigroup of a sum-ordered semiring, Semigroup Forum 31 (1) (1985) 129–152.
[47] R. Cockett, R. Garner, Restriction categories as enriched categories, Theor. Comput. Sci. 523 (2014) 37–55.
[48] M.P. Fiore, Axiomatic domain theory in categories of partial maps, Ph.D. thesis, University of Edinburgh, 1994.
[49] P. Selinger, Finite dimensional Hilbert spaces are complete for dagger compact closed categories, Log. Methods Comput. Sci. 8 (3) (2012) 1–12.
[50] P. Selinger, Idempotents in dagger categories, in: P. Selinger (Ed.), QPL 2006, in: Electronic Notes in Theoretical Computer Science, vol. 210, Elsevier, 2008, pp. 107–122.

Inversion, Fixed Points, and the Art of Dual Wielding

Robin Kaarsgaard[1,2]

DIKU, Department of Computer Science, University of Copenhagen

Abstract

In category theory, the symbol † ("dagger") is used to denote (at least) two very different operations on morphisms: Taking their *adjoint* (in the context of dagger categories) and finding their *least fixed point* (in the context of domain theory and categories enriched in domains). In the present paper, we wield both of these daggers at once and consider dagger categories enriched in domains. Exploiting the view of dagger categories as enriched in involutive monoidal categories, we develop a notion of a monotone dagger structure as a dagger structure that is well behaved with respect to the enrichment, and show that such a structure leads to pleasant inversion properties of the fixed points that arise as a result of this enrichment. Notably, such a structure guarantees the existence of *fixed point adjoints*, which we show are intimately related to the conjugates arising from the canonical involutive monoidal structure in the enrichment. Finally, we relate the results to applications in the design and semantics of reversible programming languages.

Keywords: reversible computing, dagger categories, domain theory, enriched category theory

1 Introduction

Dagger categories are categories that are canonically self-dual, assigning to each morphism an *adjoint* morphism in a contravariantly functorial way. In recent years, dagger categories have been used to capture central aspects of both reversible [28,29,31] and quantum [2,35,13] computing. Likewise, domain theory and categories enriched in domains (see, *e.g.*, [3,15,16,4,7,38]) have been successful since their inception in modelling both recursive functions and data types in programming.

In the present paper, we develop the art of dual wielding the two daggers that arise from respectively dagger category theory and domain theory (where the very same †-symbol is occasionally used to denote fixed points, *cf.* [15,16]). Concretely, we ask how these structures must interact in order to guarantee that fixed points are well-behaved with respect to the dagger, in the sense that each functional has a *fixed point adjoint* [31]. Previously, the author and others showed that certain

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

[2] The author would like to thank Martti Karvonen, Mathys Rennela, and Robert Glück for their useful comments, corrections, and suggestions; and to acknowledge the support given by *COST Action IC1405 Reversible computation: Extending horizons of computing.*

domain enriched dagger categories, join inverse categories, had such well-behaved fixed points [31]. Here, we identify a sufficient condition for fixed points to be well-behaved in the presence of a dagger, allowing us not only to generalize previous results, but also to show new ones about parametrized fixed points.

A slogan of domain theory could be that *well-behaved functions are continuous* – and as a corollary, that *well-behaved functors are locally continuous*. When augmented with a dagger, the proper addendum to this slogan turns out to be that *well-behaved inversion is monotone*, captured in the definition of a monotone dagger structure.

Given a domain enriched category $\mathscr{C}$ with a monotone dagger structure, we develop an induced involutive monoidal category of domains enriching $\mathscr{C}$, which we think of as the category of *continuous functionals* on $\mathscr{C}$. This canonically constructed involutive structure at the level of functionals proves fruitful in unifying seemingly disparate concepts from the literature under the banner of *conjugation of functionals*. Notably, we show that the conjugate functionals arising from this involutive structure coincide with *fixed point adjoints* [5,31], and that they occur naturally both in proving the ambidexterity of dagger adjunctions [23] and in natural transformations that preserve the dagger (which include dagger traces [36]).

While these results could be applied to model a reversible functional programming language with general recursion and parametrized functions (such as an extended version of Theseus [29]), they are general enough to account for even certain probabilistic and nondeterministic models of computation.

Overview: A brief introduction to the relevant background material on dagger categories, (**DCPO**-)enriched categories, iteration categories, and involutive monoidal categories is given in Section 2. In Section 3 the concept of a *monotone dagger structure* on a **DCPO**-category is introduced, and it is demonstrated that such a structure leads to the existence of fixed point adjoints for (ordinary and externally parametrized) fixed points, given by their conjugates. We also explore natural transformations in this setting, and develop a notion of *self-conjugate* natural transformations, of which †-trace operators are examples. Finally, we discuss potential applications and avenues for future research in Section 4, and end with a few concluding remarks in Section 5.

2 Background

Though familiarity with basic category theory, including monoidal categories, is assumed, we recall here some basic concepts relating to dagger categories, (**DCPO**)-enriched categories, iteration categories, and involutive monoidal categories [26,8]. The material is only covered here briefly, but can be found in much more detail in the numerous texts on dagger category theory (see, *e.g.*, [35,2,21]), enriched category theory (for which [33] is the standard text), and domain theory and iteration categories (see, *e.g.*, [3,16]).

2.1 Dagger categories

A dagger category (or †-category) is a category equipped with a suitable method for flipping the direction of morphisms, by assigning to each morphism an *adjoint* in a manner consistent with composition. They are formally defined as follows.

Definition 2.1 A dagger category is a category $\mathscr{C}$ equipped with an functor $(-)^\dagger : \mathscr{C}^{\mathrm{op}} \to \mathscr{C}$ satisfying that $\mathrm{id}_X^\dagger = \mathrm{id}_X$ and $f^{\dagger\dagger} = f$ for all identities $X \xrightarrow{\mathrm{id}_X} X$ and morphisms $X \xrightarrow{f} Y$.

Dagger categories, dagger functors (*i.e.*, functors F satisfying $F(f^\dagger) = F(f)^\dagger$), and natural transformations form a 2-category, **DagCat**.

A given category may have several different daggers which need not agree. An example of this is the groupoid of finite-dimensional Hilbert spaces and linear isomorphisms, which has (at least!) two daggers: One maps linear isomorphisms to their linear inverse, the other maps linear isomorphisms to their hermitian conjugate. The two only agree on the unitaries, *i.e.*, the linear isomorphisms which additionally preserve the inner product. For this reason, one would in principle need to specify *which* dagger one is talking about on a given category, though this is often left implicit (as will also be done here).

Let us recall the definition of the some interesting properties of morphisms in a dagger category: By theft of terminology from linear algebra, say that a morphism $X \xrightarrow{f} Y$ in a dagger category is *hermitian* or *self-adjoint* if $f = f^\dagger$, and *unitary* if it is an isomorphism and $f^{-1} = f^\dagger$. Whereas objects are usually considered equivalent if they are isomorphic, the "way of the dagger" [23] dictates that all structure in sight must cooperate with the dagger; as such, objects ought to be considered equivalent in dagger categories only if they are isomorphic via a unitary map.

We end with a few examples of dagger categories. As discussed above, **FHilb** is an example (*the* motivating one, even [35]) of dagger categories, with the dagger given by hermitian conjugation. The category **PInj** of sets and partial injective functions is a dagger category (indeed, it is an *inverse category* [32,12]) with $f^\dagger$ given by the partial inverse of f. Similarly, the category **Rel** of sets and relations has a dagger given by $R^\dagger = R^\circ$, *i.e.*, the relational converse of R. Noting that a dagger subcategory is given by the existence of a faithful dagger functor, it can be shown that **PInj** is a dagger subcategory of **Rel** with the given dagger structures.

2.2 **DCPO***-categories and other enriched categories*

Enriched categories (see, *e.g.*, [33]) capture the idea that homsets on certain categories can (indeed, ought to) be understood as something other than sets – or in other words, as objects of a another category than **Set**. A category $\mathscr{C}$ is *enriched* in a monoidal category $\mathscr{V}$ if all homsets $\mathscr{C}(X, Y)$ of $\mathscr{C}$ are objects of $\mathscr{V}$, and for all objects X, Y, Z of $\mathscr{C}$, $\mathscr{V}$ has families of morphisms $\mathscr{C}(Y, Z) \otimes \mathscr{C}(X, Y) \to \mathscr{C}(X, Z)$ and $I \to \mathscr{C}(X, X)$ corresponding to composition and identities in $\mathscr{C}$, subject to commutativity of diagrams corresponding to the usual requirements of associativity of composition, and of left and right identity. As is common, we will often use the shorthand "$\mathscr{C}$ is a $\mathscr{V}$-category" to mean that $\mathscr{C}$ is enriched in the category $\mathscr{V}$.

We focus here on categories enriched in the category of *domains* (see, *e.g.*, [3]), *i.e.*, the category **DCPO** of pointed directed complete partial orders and continuous maps. A partially ordered $(X, \sqsubseteq)$ is said to be directed complete if every directed set (*i.e.*, a *non-empty* $A \subseteq X$ satisfying that any pair of elements of A has a supremum in A) has a supremum in X. A function f between directed complete partial orders is monotone if $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$ for all x, y, and continuous if $f(\sup A) = \sup_{a \in A}\{f(a)\}$ for each directed set A (note that continuity implies monotony). A directed complete partial order is *pointed* if it has a least element $\perp$ (or, in other words, if also the empty set has a supremum), and a function f between such is called *strict* if $f(\perp) = \perp$ (*i.e.*, if also the supremum of the empty set is preserved [3]). Pointed directed complete partial orders and continuous maps form a category, **DCPO**.

As such, a category enriched in **DCPO** is a category $\mathscr{C}$ in which homsets $\mathscr{C}(X, Y)$ are directed complete partial orders, and composition is continuous. Additionally, we will require that composition is strict (meaning that $\perp \circ f = \perp$ and $g \circ \perp = \perp$ for all suitable morphisms f and g), so that the category is actually enriched in the category **DCPO**! of directed complete partial orders and strict continuous functions, though we will not otherwise require functions to be strict.

Enrichment in **DCPO** provides a method for constructing morphisms in the enriched category as least fixed points of continuous functions between homsets: This is commonly used to model recursion. Given a continuous function $\mathscr{C}(X, Y) \xrightarrow{\varphi} \mathscr{C}(X, Y)$, by Kleene's fixed point theorem there exists a least fixed point $X \xrightarrow{\operatorname{fix}\varphi} Y$ given by $\sup_{n \in \omega}\{\varphi^n(\perp)\}$, where φ^n is the n-fold composition of φ with itself.

2.3 Parametrized fixed points and iteration categories

Related to the fixed point operator is the *parametrized fixed point operator*, an operator pfix assigning morphisms of the form $X \times Y \xrightarrow{\psi} X$ to a morphism $Y \xrightarrow{\operatorname{pfix}\psi} X$ satisfying equations such as the *parametrized fixed point identity*

$$\operatorname{pfix}\psi = \psi \circ \langle \operatorname{pfix}\psi, \mathrm{id}_Y \rangle$$

and others (see, *e.g.*, [25,15]). Parametrized fixed points are used to solve domain equations of the form $x = \psi(x, p)$ for some given parameter $p \in Y$. Indeed, if for a continuous function $X \times Y \xrightarrow{\psi} X$ we define $\psi^0(x, p) = x$ and $\psi^{n+1}(x, p) = \psi(\psi^n(x, p), p)$, we can construct its parametrized fixed point in **DCPO** in a way reminiscent of the usual fixed point by

$$(\operatorname{pfix}\psi)(p) = \sup_{n \in \omega}\{\psi^n(\perp_X, p)\} \enspace .$$

In fact, a parametrized fixed point operator may be derived from an ordinary fixed point operator by $(\operatorname{pfix}\psi)(p) = \operatorname{fix}\psi(-, p)$. Similarly, we may derive an ordinary fixed point operator from a parametrized one by considering a morphism $X \xrightarrow{\varphi} X$ to be parametrized by the terminal object 1, so that the fixed point of $X \xrightarrow{\varphi} X$ is given by the parametrized fixed point of $X \times 1 \xrightarrow{\pi_1} X \xrightarrow{\varphi} X$.

The parametrized fixed point operation is sometimes also called a *dagger operation* [15], and denoted by $f^\dagger$ rather than $\operatorname{pfix} f$. Though this is indeed the other

[3] This is *not* the case in general, as continuous functions are only required to preserve least upper bounds of directed sets, which, by definition, does not include the empty set.

dagger that we are wielding, we will use the phrase "parametrized fixed point" and notation "pfix" to avoid unnecessary confusion.

An *iteration category* [16] is a cartesian category with a parametrized fixed point operator that behaves in a canonical way. The definition of an iteration category came out of the observation that the parametrized fixed point operator in a host of concrete categories (notably **DCPO**) satisfy the same identities. This lead to an elegant semantic characterization of iteration categories, due to [16].

Definition 2.2 An *iteration category* is a cartesian category with a parametrized fixed point operator satisfying all identities (of the parametrized fixed point operator) that hold in **DCPO**.

Note that the original definition defined iteration categories in relation to the category $\mathbf{CPO}_m$ of ω-complete partial orders and monotone functions, rather than to **DCPO**. However, the motivating theorem [16, Theorem 1] shows that the parametrized fixed point operator in $\mathbf{CPO}_m$ satisfies the same identities as the one found in **CPO** (*i.e.*, with continuous rather than monotone functions). Since the parametrized fixed point operator of **DCPO** is constructed *precisely* as it is in **CPO** (noting that ω-chains are directed sets), this definition is equivalent to the original.

2.4 Involutive monoidal categories

An involutive category [26] is a category in which every object X can be assigned a *conjugate* object $\overline{X}$ in a functorial way such that $\overline{\overline{X}} \cong X$. A novel idea by Egger [14] is to consider dagger categories as categories enriched in an *involutive monoidal category*. We will return to this idea in Section 3.1, and recall the relevant definitions in the meantime (due to [26], compare also with *bar categories* [8]).

Definition 2.3 A category $\mathscr{V}$ is *involutive* if it is equipped with a functor $\mathscr{V} \xrightarrow{\overline{(-)}} \mathscr{V}$ (the *involution*) and a natural isomorphism $\mathrm{id} \overset{\iota}{\Rightarrow} \overline{\overline{(-)}}$ satisfying $\iota_{\overline{X}} = \overline{\iota_X}$.

Borrowing terminology from linear algebra, we call $\overline{X}$ (respectively $\overline{f}$) the *conjugate* of an object X (respectively a morphism f), and say that an object X is *self-conjugate* if $X \cong \overline{X}$. Note that since conjugation is covariant, any category $\mathscr{C}$ can be made involutive by assigning $\overline{X} = X$, $\overline{f} = f$, and letting $\mathrm{id} \overset{\iota}{\Rightarrow} \overline{\overline{(-)}}$ be the identity in each component; as such, an involution is a structure rather than a property. Non-trivial examples of involutive categories include the category of complex vector spaces $\mathbf{Vect}_{\mathbb{C}}$, with the involution given by the usual conjugation of complex vector spaces; and the category **Poset** of partially ordered sets and monotone functions, with the involution given by order reversal.

When a category is both involutive and (symmetric) monoidal, we say that it is an *involutive (symmetric) monoidal category* when these two structures play well together, as in the following definition [26].

Definition 2.4 An *involutive (symmetric) monoidal category* is a (symmetric) monoidal category $\mathscr{V}$ which is also involutive, such that the involution is a monoidal functor, and $\mathrm{id} \Rightarrow \overline{\overline{(-)}}$ is a monoidal natural isomorphism.

This specifically gives us a natural family of isomorphisms $\overline{X \otimes Y} \cong \overline{X} \otimes \overline{Y}$, and when the monoidal product is symmetric, this extends to a natural isomorphism $\overline{X \otimes Y} \cong \overline{Y} \otimes \overline{X}$. This fact will turn out to be useful later on when we consider dagger categories as enriched in certain involutive symmetric monoidal categories.

3 Domain enriched dagger categories

Given a dagger category that also happens to be enriched in domains, we ask how these two structures ought to interact with one another. Since domain theory dictates that the well-behaved functions are precisely the continuous ones, a natural first answer would be to that the dagger should be locally continuous; however, it turns out that we can make do with less.

Definition 3.1 Say that a dagger structure on **DCPO**-category is *monotone* if the dagger is locally monotone, *i.e.*, if $f \sqsubseteq g$ implies $f^\dagger \sqsubseteq g^\dagger$ for all f and g.

In the following, we will use the terms "**DCPO**-category with a monotone dagger structure" and "**DCPO**-†-category" interchangably. That this is sufficient to get what we want – in particular to obtain local continuity of the dagger – is shown in the following lemma.

Lemma 3.2 *In any* **DCPO**-†*-category, the dagger is an order isomorphism on morphisms; in particular it is continuous and strict.*

Proof. For $\mathscr{C}$ a dagger category, $\mathscr{C} \cong \mathscr{C}^{\mathrm{op}}$ so $\mathscr{C}(X,Y) \cong \mathscr{C}^{\mathrm{op}}(X,Y) = \mathscr{C}(Y,X)$ for all objects X, Y; that this isomorphism of hom-objects is an order isomorphism follows directly by local monotony. □

Let us consider a few examples of **DCPO**-†-categories.

Example 3.3 The category **Rel** of sets and relations is a dagger category, with the dagger given by $R^\dagger = R^\circ$, the relational converse of R (*i.e.*, defined by $(y,x) \in R^\circ$ iff $(x,y) \in R$) for each such relation. It is also enriched in **DCPO** by the usual subset ordering: Since a relation $\mathcal{X} \to \mathcal{Y}$ is nothing more than a subset of $\mathcal{X} \times \mathcal{Y}$, equipped with the subset order $- \subseteq -$ we have that $\sup(\Delta) = \bigcup_{R \in \Delta} R$ for any directed set $\Delta \subseteq \mathbf{Rel}(\mathcal{X}, \mathcal{Y})$. It is also pointed, with the least element of each homset given by the empty relation.

To see that this is a monotone dagger structure, let $\mathcal{X} \xrightarrow{R,S} \mathcal{Y}$ be relations and suppose that $R \subseteq S$. Let $(y,x) \in R^\circ$.Since $(y,x) \in R^\circ$ we have $(x,y) \in R$ by definition of the relational converse, and by the assumption that $R \subseteq S$ we also have $(x,y) \in S$. But then $(y,x) \in S^\circ$ by definition of the relational converse, so $R^\dagger = R^\circ \subseteq S^\circ = S^\dagger$ follows by extensionality.

Example 3.4 We noted earlier that the category **PInj** of sets and partial injective functions is a dagger subcategory of **Rel**, with $f^\dagger$ given by the partial inverse (a special case of the relational converse) of a partial injection f. Further, it is also a **DCPO**-subcategory of **Rel**; in **PInj**, this becomes the relation that for $X \xrightarrow{f,g} Y$, $f \sqsubseteq g$ iff for all $x \in X$, if f is defined at x and $f(x) = y$, then g is also defined at x and $g(x) = y$. Like **Rel**, it is pointed with the nowhere defined partial function as the least element of each homset. That $\sup(\Delta)$ for some directed $\Delta \subseteq \mathbf{PInj}(X,Y)$ is a partial injection follows straightforwardly, and that this dagger structure is monotone follows by an argument analogous to the one for **Rel**.

Example 3.5 More generally, any *join inverse category* (see [17]), of which **PInj** is one, is a **DCPO**-†-category. Inverse categories are canonically dagger categories enriched in partial orders. That this extends to **DCPO**-enrichment in the presence of joins is shown in [5,31]; that the canonical dagger is monotonous with respect to the partial order is an elementary result (see, *e.g.*, [5, Lemma 2]).

Example 3.6 The category $\mathbf{DStoch}_{\leq 1}$ of finite sets and *doubly substochastic maps* is an example of a probabilistic **DCPO**-†-category. A doubly substochastic map $X \xrightarrow{f} Y$, where $|X| = |Y| = n$, is given by an $n \times n$ matrix $A = [a_{ij}]$ with non-negative real entries such that $\sum_{i=1}^{n} a_{ij} \leq 1$ and $\sum_{j=1}^{n} a_{ij} \leq 1$. Composition is given by the usual multiplication of matrices.

This is a dagger category with the dagger given by matrix transposition. It is also enriched in **DCPO** by ordering doubly substochastic maps entry-wise (*i.e.*, $A \leq B$ if $a_{ij} \leq b_{ij}$ for all i, j), with the everywhere-zero matrix as the least element in each homset, and with suprema of directed sets given by computing suprema entry-wise. That this dagger structure is monotone follows by the fact that if $A \leq B$, so $a_{ij} \leq b_{ij}$ for all i, j, then also $a_{ji} \leq b_{ji}$ for all j, i, which is precisely to say that $A^\dagger = A^T \leq B^T = B^\dagger$.

As such, in terms of computational content, these are examples of deterministic, nondeterministic, and probabilistic **DCPO**-†-categories. We will also discuss the related category $\mathbf{CP}^*(\mathbf{FHilb})$, used to model quantum phenomena, in Section 4.

3.1 The category of continuous functionals

We illustrate here the idea of dagger categories as categories enriched in an involutive monoidal category by an example that will be used throughout the remainder of this article: Enrichment in a suitable subcategory of **DCPO**. It is worth stressing, however, that the construction is *not* limited to dagger categories enriched in **DCPO**; any dagger category will do. As we will see later, however, this canonical involution turns out to be very useful when **DCPO**-†-categories are considered.

Let $\mathscr{C}$ be a **DCPO**-†-category. We define an induced (full monoidal) subcategory of **DCPO**, call it $\mathbf{DcpoOp}(\mathscr{C})$, which enriches $\mathscr{C}$ (by its definition) as follows:

Definition 3.7 For a **DCPO**-†-category $\mathscr{C}$, define $\mathbf{DcpoOp}(\mathscr{C})$ to have as objects all objects Θ, Λ of **DCPO** of the form $\mathscr{C}(X,Y)$, $\mathscr{C}^{\mathrm{op}}(X,Y)$ (for all objects X, Y of $\mathscr{C}$), 1, and $\Theta \times \Lambda$ (with 1 initial object of **DCPO**, and $- \times -$ the cartesian product), and as morphisms all continuous functions between these.

In other words, $\mathbf{DcpoOp}(\mathscr{C})$ is the (full) cartesian subcategory of **DCPO** generated by objects used in the enrichment of $\mathscr{C}$, with all continuous maps between these. That the dagger on $\mathscr{C}$ induces an involution on $\mathbf{DcpoOp}(\mathscr{C})$ is shown in the following theorem.

Theorem 3.8 $\mathbf{DcpoOp}(\mathscr{C})$ *is an involutive symmetric monoidal category.*

Proof. On objects, define an involution $\overline{(-)}$ with respect to the cartesian (specifically symmetric monoidal) product of **DCPO** as follows, for all objects Θ, Λ, Σ of **DcpoOp**($\mathscr{C}$): $\overline{\mathscr{C}(X,Y)} = \mathscr{C}^{\mathrm{op}}(X,Y)$, $\overline{\mathscr{C}^{\mathrm{op}}(X,Y)} = \mathscr{C}(X,Y)$, $\overline{1} = 1$, and $\overline{\Theta \times \Lambda} = \overline{\Theta} \times \overline{\Lambda}$. To see that this is well-defined, recall that $\mathscr{C} \cong \mathscr{C}^{\mathrm{op}}$ for any dagger category $\mathscr{C}$, so in particular there is an isomorphism witnessing $\mathscr{C}(X,Y) \cong \mathscr{C}^{\mathrm{op}}(X,Y)$ given by the mapping $f \mapsto f^\dagger$. But then $\mathscr{C}^{\mathrm{op}}(X,Y) = \{f^\dagger \mid f \in \mathscr{C}(X,Y)\}$, so if $\mathscr{C}(X,Y) = \mathscr{C}(X',Y')$ then $\overline{\mathscr{C}(X,Y)} = \mathscr{C}^{\mathrm{op}}(X,Y) = \{f^\dagger \mid f \in \mathscr{C}(X,Y)\} = \{f^\dagger \mid f \in \mathscr{C}(X',Y')\} = \mathscr{C}^{\mathrm{op}}(X',Y') = \overline{\mathscr{C}(X',Y')}$. That $\overline{\mathscr{C}^{\mathrm{op}}(X,Y)} = \mathscr{C}(X,Y)$ is well-defined follows by analogous argument.

On morphisms, we define a family ξ of isomorphisms by $\xi_I = \mathrm{id}_I$, $\xi_{\mathscr{C}(X,Y)} = (-)^\dagger$, $\xi_{\mathscr{C}^{\mathrm{op}}(X,Y)} = (-)^\dagger$, and $\xi_{\Theta\times\Lambda} = \xi_\Theta \times \xi_\Lambda$, and then define

$$\overline{\Theta \xrightarrow{\varphi} \Lambda} = \overline{\Theta} \xrightarrow{\xi_\Theta^{-1}} \Theta \xrightarrow{\varphi} \Lambda \xrightarrow{\xi_\Lambda} \overline{\Lambda}.$$

This is functorial as $\overline{\mathrm{id}_\Theta} = \xi_\Theta \circ \mathrm{id}_\Theta \circ \xi_\Theta^{-1} = \xi_\Theta \circ \xi_\Theta^{-1} = \mathrm{id}_{\overline{\Theta}}$, and for $\Theta \xrightarrow{\varphi} \Lambda \xrightarrow{\psi} \Sigma$,

$$\overline{\psi \circ \varphi} = \xi_\Sigma \circ \psi \circ \varphi \circ \xi_\Theta^{-1} = \xi_\Sigma \circ \psi \circ \xi_\Lambda^{-1} \circ \xi_\Lambda \circ \varphi \circ \xi_\Theta^{-1} = \overline{\psi} \circ \overline{\varphi}.$$

Finally, since the involution is straightforwardly a monoidal functor, and since the natural transformation $\mathrm{id} \Rightarrow \overline{\overline{(-)}}$ can be chosen to be the identity since all objects of **DcpoOp**($\mathscr{C}$) satisfy $\overline{\overline{\Theta}} = \Theta$ by definition, this is an involutive symmetric monoidal category. □

The resulting category **DcpoOp**($\mathscr{C}$) can very naturally be thought of as the induced *category of (continuous) functionals* (or second-order functions) of $\mathscr{C}$.

Notice that this is a special case of a more general construction on dagger categories: For a dagger category $\mathscr{C}$ enriched in some category $\mathscr{V}$ (which could simply be **Set** in the unenriched case), one can construct the category $\mathscr{V}$**Op**($\mathscr{C}$), given on objects by the image of the hom-functor $\mathscr{C}(-,-)$ closed under monoidal products, and on morphisms by all morphisms of $\mathscr{V}$ between objects of this form. Defining the involution as above, $\mathscr{V}$**Op**($\mathscr{C}$) can be shown to be involutive monoidal.

Example 3.9 One may question how natural (in a non-technical sense) the choice of involution on **DcpoOp**($\mathscr{C}$) is. One instance where it turns out to be useful is in the context of dagger adjunctions (see [23] for details), that is, adjunctions between dagger categories where both functors are dagger functors.

Dagger adjunctions have no specified left and right adjoint, as all such adjunctions can be shown to be ambidextrous in the following way: Given $F \dashv G$ between endofunctors on $\mathscr{C}$, there is a natural isomorphism $\mathscr{C}(FX,Y) \xrightarrow{\alpha_{X,Y}} \mathscr{C}(X,GY)$. Since $\mathscr{C}$ is a dagger category, we can define a natural isomorphism $\mathscr{C}(X,FY) \xrightarrow{\beta_{X,Y}} \mathscr{C}(GX,Y)$ by $f \mapsto \alpha_{Y,X}(f^\dagger)^\dagger$, *i.e.*, by the composition

$$\mathscr{C}(X,FY) \xrightarrow{\xi} \mathscr{C}(FY,X) \xrightarrow{\alpha_{Y,X}} \mathscr{C}(Y,GX) \xrightarrow{\xi} \mathscr{C}(GX,Y)$$

which then witnesses $G \dashv F$ (as it is a composition of natural isomorphisms). But then $\beta_{X,Y}$ is defined precisely to be $\overline{\alpha_{Y,X}}$ when F and G are endofunctors.

3.2 Daggers and fixed points

In this section we consider the morphisms of $\mathbf{DcpoOp}(\mathscr{C})$ in some detail, for a $\mathbf{DCPO}$-†-category $\mathscr{C}$. Since least fixed points of morphisms are such a prominent and useful feature of $\mathbf{DCPO}$-enriched categories, we ask how these behave with respect to the dagger. To answer this question, we transplant the notion of a *fixed point adjoint* from [5,31] to $\mathbf{DCPO}$-†-categories, where an answer to this question in relation to the more specific *join inverse categories* was given:

Definition 3.10 A functional $\mathscr{C}(Y,X) \xrightarrow{\varphi_\ddagger} \mathscr{C}(Y,X)$ is *fixed point adjoint* to a functional $\mathscr{C}(X,Y) \xrightarrow{\varphi} \mathscr{C}(X,Y)$ iff $(\operatorname{fix}\varphi)^\dagger = \operatorname{fix}\varphi_\ddagger$.

Note that this is symmetric: If $\varphi_\ddagger$ is fixed point adjoint to φ then $\operatorname{fix}(\varphi_\ddagger)^\dagger = (\operatorname{fix}\varphi)^{\dagger\dagger} = \operatorname{fix}\varphi$, so φ is also fixed point adjoint to $\varphi_\ddagger$. As shown in the following theorem, it turns out that the conjugate $\overline{\varphi}$ of a functional φ is precisely fixed point adjoint to it. This is a generalization of a theorem from [31], where a more ad-hoc formulation was shown for join inverse categories, which constitute a non-trivial subclass of $\mathbf{DCPO}$-†-categories.

Theorem 3.11 *Every functional is fixed point adjoint to its conjugate.*

Proof. The proof applies the exact same construction as in [31], since being a $\mathbf{DCPO}$-†-category suffices, and the constructed fixed point adjoint turns out to be the exact same. Let $\mathscr{C}(X,Y) \xrightarrow{\varphi} \mathscr{C}(X,Y)$ be a functional. Since $\overline{\varphi} = \xi_{\mathscr{C}(X,Y)} \circ \varphi \circ \xi^{-1}_{\mathscr{C}(X,Y)}$,

$$\bar{\varphi}^n = \left(\xi_{\mathscr{C}(X,Y)} \circ \varphi \circ \xi^{-1}_{\mathscr{C}(X,Y)}\right)^n = \xi_{\mathscr{C}(X,Y)} \circ \varphi^n \circ \xi^{-1}_{\mathscr{C}(X,Y)}$$

and so

$$\begin{aligned}\operatorname{fix}\overline{\varphi} &= \sup\{\bar{\varphi}^n(\perp_{Y,X})\}_{n\in\omega} = \sup\{\varphi^n(\perp^\dagger_{Y,X})^\dagger\} = \sup\{\varphi^n(\perp_{X,Y})^\dagger\} \\ &= \sup\{\varphi^n(\perp_{X,Y})\}^\dagger = (\operatorname{fix}\varphi)^\dagger\end{aligned}$$

as desired. □

This theorem is somewhat surprising, as the conjugate came out of the involutive monoidal structure on $\mathbf{DcpoOp}(\mathscr{C})$, which is not specifically related to the presence of fixed points. As previously noted, had $\mathscr{C}$ been enriched in another category $\mathscr{V}$, we would still be able to construct a category $\mathscr{V}\mathbf{Op}(\mathscr{C})$ of $\mathscr{V}$-functionals with the *exact same* involutive structure.

As regards recursion, this theorem underlines the slogan that *reversibility is a local phenomenon*: To construct the inverse to a recursively defined morphism $\operatorname{fix}\varphi$, it suffices to invert the local morphism φ at each step (which is essentially what is done by the conjugate $\overline{\varphi}$) in order to construct the global inverse $(\operatorname{fix}\varphi)^\dagger$.

Parametrized functionals and their external fixed points are also interesting to consider in this setting, as some examples of $\mathbf{DCPO}$-†-categories (*e.g.*, $\mathbf{PInj}$) fail to have an internal hom. For example, in a dagger category with objects $L(X)$ corresponding to "lists of X" (usually constructed as the fixed point of a suitable functor), one could very reasonably construe the usual map-function not as a higher-order function, but as a family of morphisms $LX \xrightarrow{\mathrm{map}\langle f\rangle} LY$ indexed by $X \xrightarrow{f} Y$ – or, in other words, as a functional $\mathscr{C}(X,Y) \xrightarrow{\mathrm{map}} \mathscr{C}(LX,LY)$. Indeed, this is how certain higher-order behaviours are mimicked in the reversible functional programming language Theseus (see also Section 4).

To achieve such parametrized fixed points of functionals, we naturally need a parametrized fixed point operator on $\mathbf{DcpoOp}(\mathscr{C})$ satisfying the appropriate equations – or, in other words, we need $\mathbf{DcpoOp}(\mathscr{C})$ to be an *iteration category*. That $\mathbf{DcpoOp}(\mathscr{C})$ is such an iteration category follows immediately by its definition (*i.e.*, since $\mathbf{DcpoOp}(\mathscr{C})$ is a full subcategory of $\mathbf{DCPO}$, we can define a parametrized fixed point operator in $\mathbf{DcpoOp}(\mathscr{C})$ to be precisely the one in $\mathbf{DCPO}$), noting that parametrized fixed points preserve continuity.

Lemma 3.12 $\mathbf{DcpoOp}(\mathscr{C})$ *is an iteration category.*

For functionals of the form $\mathscr{C}(X,Y) \times \mathscr{C}(P,Q) \xrightarrow{\psi} \mathscr{C}(X,Y)$, we can make a similar definition of a *parametrized fixed point adjoint*:

Definition 3.13 A functional $\mathscr{C}(X,Y) \times \mathscr{C}(P,Q) \xrightarrow{\psi_\ddagger} \mathscr{C}(X,Y)$ is *parametrized fixed point adjoint* to a functional $\mathscr{C}(X,Y) \times \mathscr{C}(P,Q) \xrightarrow{\psi} \mathscr{C}(X,Y)$ iff $(\mathrm{pfix}\,\psi)(p)^\dagger = (\mathrm{pfix}\,\psi_\ddagger)(p^\dagger)$.

We can now show a similar theorem for parametrized fixed points of functionals and their conjugates:

Theorem 3.14 *Every functional is parametrized fixed point adjoint to its conjugate.*

Proof. Let $\mathscr{C}(X,Y) \times \mathscr{C}(P,Q) \xrightarrow{\psi} \mathscr{C}(X,Y)$ be a functional. We start by showing that $\bar{\psi}^n(f,p) = \psi^n(f^\dagger,p^\dagger)^\dagger$ for all $Y \xrightarrow{f} X$, $Q \xrightarrow{p} P$, and $n \in \mathbb{N}$, by induction on n. For $n = 0$ we have

$$\bar{\psi}^0(f,p) = f = f^{\dagger\dagger} = (f^\dagger)^\dagger = \psi^0(f^\dagger,p^\dagger)^\dagger.$$

Assuming now the induction hypothesis for some n, we have

$$\begin{aligned}\bar{\psi}^{n+1}(f,p) &= \bar{\psi}(\bar{\psi}^n(f,p),p) = \bar{\psi}(\psi^n(f^\dagger,p^\dagger)^\dagger,p) = \psi(\psi^n(f^\dagger,p^\dagger)^{\dagger\dagger},p^\dagger)^\dagger \\ &= \psi(\psi^n(f^\dagger,p^\dagger),p^\dagger)^\dagger = \psi^{n+1}(f^\dagger,p^\dagger)^\dagger\end{aligned}$$

Using this fact, we now get

$$\begin{aligned}(\mathrm{pfix}\,\overline{\psi})(p^\dagger) &= \sup_{n\in\omega}\{\bar{\psi}^n(\bot_{Y,X},p^\dagger)\} = \sup_{n\in\omega}\{\psi^n(\bot_{Y,X}^\dagger,p^{\dagger\dagger})^\dagger\} \\ &= \sup_{n\in\omega}\{\psi^n(\bot_{X,Y},p)\}^\dagger = (\mathrm{pfix}\,\psi)(p)^\dagger\end{aligned}$$

which was what we wanted. □

Again, this theorem highlights the local nature of reversibility, here in the presence of additional parameters. We observe further the following highly useful property of parametrized fixed points in $\mathbf{DcpoOp}(\mathscr{C})$:

Lemma 3.15 *Parametrized fixed points in* $\mathbf{DcpoOp}(\mathscr{C})$ *preserve conjugation.*

Proof. Let $\mathscr{C}(X,Y) \times \mathscr{C}(P,Q) \xrightarrow{\psi} \mathscr{C}(X,Y)$ be continuous, and $P \xrightarrow{p} Q$. Then $\overline{\mathrm{pfix}\,\psi}(p) = (\xi \circ (\mathrm{pfix}\,\psi) \circ \xi^{-1})(p) = (\mathrm{pfix}\,\psi)(p^\dagger)^\dagger = (\mathrm{pfix}\,\overline{\psi})(p)^{\dagger\dagger} = (\mathrm{pfix}\,\overline{\psi})(p)$, so $\overline{\mathrm{pfix}\,\psi} = \mathrm{pfix}\,\overline{\psi}$. □

Note that a lemma of this form only makes sense for parametrized fixed points, as the usual fixed point of a functional $\mathscr{C}(X,Y) \xrightarrow{\varphi} \mathscr{C}(X,Y)$ results in a morphism $X \xrightarrow{\mathrm{fix}\,\varphi} Y$ in $\mathscr{C}$, not a functional in $\mathbf{DcpoOp}(\mathscr{C})$.

3.3 Naturality and self-conjugacy

We now consider the behaviour of functionals and their parametrized fixed points when they are natural. For example, given a natural family of functionals $\mathscr{C}(FX,FY) \xrightarrow{\alpha_{X,Y}} \mathscr{C}(GX,GY)$ natural in X and Y (for dagger endofunctors F and G on $\mathscr{C}$), what does it mean for such a family to be well-behaved with respect to the dagger on $\mathscr{C}$? We would certainly want that such a family preserves the dagger, in the sense that $\alpha_{X,Y}(f)^\dagger = \alpha_{Y,X}(f^\dagger)$ in each component X,Y. It turns out that this, too, can be expressed in terms of conjugation of functionals.

Lemma 3.16 *Let* $\mathscr{C}(FX,FY) \xrightarrow{\alpha_{X,Y}} \mathscr{C}(GX,GY)$ *be a family of functionals natural in* X *and* Y*. Then* $\alpha_{X,Y}(f)^\dagger = \alpha_{Y,X}(f^\dagger)$ *for all* $X \xrightarrow{f} Y$ *iff* $\alpha_{X,Y} = \overline{\alpha_{Y,X}}$*.*

Proof. Suppose $\alpha_{X,Y}(f)^\dagger = \alpha_{Y,X}(f^\dagger)$. Then $\alpha_{X,Y}(f) = \alpha_{X,Y}(f)^{\dagger\dagger} = \alpha_{Y,X}(f^\dagger)^\dagger = \overline{\alpha_{Y,X}}(f)$, so $\alpha_{X,Y} = \overline{\alpha_{Y,X}}$. Conversely, assuming $\alpha_{X,Y} = \overline{\alpha_{Y,X}}$ we then have for all $X \xrightarrow{f} Y$ that $\alpha_{X,Y}(f) = \alpha_{Y,X}(f^\dagger)^\dagger$, so $\alpha_{X,Y}(f)^\dagger = \alpha_{Y,X}(f^\dagger)^{\dagger\dagger} = \alpha_{Y,X}(f^\dagger)$. □

If a natural transformation α satisfies $\alpha_{X,Y} = \overline{\alpha_{Y,X}}$ in all components X,Y, we say that it is *self-conjugate*. An important example of a self-conjugate natural transformation is the *dagger trace operator*, as detailed in the following example.

Example 3.17 A trace operator [30] on a braided monoidal category $\mathscr{D}$ is family of functionals

$$\mathscr{D}(X \otimes U, Y \otimes U) \xrightarrow{\mathrm{Tr}^U_{X,Y}} \mathscr{D}(X,Y)$$

subject to equations such as naturality in X and Y, dinaturality in U, and others. Traces have been used to model features from traces in tensorial vector spaces [20] to tail recursion in programming languages [1,9,19], and occur naturally in compact closed (or, more generally, tortile monoidal) categories [30] and unique decomposition categories [18,24].

A *dagger trace operator* on a dagger category (see, *e.g.*, [36]) is precisely a trace operator on a dagger monoidal category (*i.e.*, a monoidal category where the monoidal functor is a dagger functor) that satisfies $\mathrm{Tr}^U_{X,Y}(f)^\dagger = \mathrm{Tr}^U_{Y,X}(f^\dagger)$ in all components X,Y. Such traces have been used to model reversible tail recursion in reversible programming languages [28,29,31], and also occur in the *dagger compact closed categories* (see, *e.g.*, [37]) used to model quantum computation. In light of Lemma 3.16, dagger traces are important examples of self-conjugate natural transformations on dagger categories.

Given the connections between (di)naturality and parametric polymorphism [39,6], one would wish that parametrized fixed points preserve naturality. Luckily, this does turn out to be the case, as shown in the proof of the following theorem.

Theorem 3.18 *If* $\mathscr{C}(FX, FY) \times \mathscr{C}(GX, GY) \xrightarrow{\alpha_{X,Y}} \mathscr{C}(FX, FY)$ *is natural in* X *and* Y*, so is its parametrized fixed point.*

Proof. Suppose that α is natural in X and Y, *i.e.*, the following diagram commutes for all X, Y.

$$\begin{tikzcd}
\mathscr{C}(FX, FY) \times \mathscr{C}(GX, GY) \arrow[r, "\alpha_{X,Y}"] \arrow[d, "Ff \times Gf \circ - \circ Fg \times Gg"'] & \mathscr{C}(FX, FY) \arrow[d, "Ff \circ - \circ Fg"] \\
\mathscr{C}(FX', FY') \times \mathscr{C}(GX', GY') \arrow[r, "\alpha_{X',Y'}"'] & \mathscr{C}(FX', FY')
\end{tikzcd}$$

Under this assumption, we start by showing naturality of α^n for all $n \in \mathbb{N}$, *i.e.*, for all $GX \xrightarrow{p} GY$

$$\alpha^n_{X',Y'}(\perp_{X',Y'}, Gf \circ p \circ Gg) = Ff \circ \alpha^n_{X,Y}(\perp_{X,Y}, p) \circ Fg$$

by induction on n. For $n = 0$ we have

$$\begin{aligned}
\alpha^0_{X',Y'}(\perp_{X,Y}, Gf \circ p \circ Gg) &= \perp_{X',Y'} \\
&= Ff \circ \perp_{X,Y} \circ Fg \\
&= Ff \circ \alpha^0_{X,Y}(\perp_{X,Y}, p) \circ Fg.
\end{aligned}$$

where $Ff \circ \perp_{X,Y} \circ Fg = \perp_{X',Y'}$ by strictness of composition. Assuming the induction hypothesis now for some n, we have

$$\begin{aligned}
\alpha^{n+1}_{X',Y'}(\perp_{X',Y'}, Gf \circ p \circ Gg) &= \alpha_{X',Y'}(\alpha^n_{X',Y'}(\perp_{X',Y'}, Gf \circ p \circ Gg), Gf \circ p \circ Gg) \\
&= \alpha_{X',Y'}(Ff \circ \alpha^n_{X,Y}(\perp_{X,Y}, p) \circ Fg, Gf \circ p \circ Gg) \\
&= Ff \circ \alpha_{X,Y}(\alpha^n_{X,Y}(\perp_{X,Y}, p), p) \circ Fg \\
&= Ff \circ \alpha^{n+1}_{X,Y}(\perp_{X,Y}, p) \circ Fg
\end{aligned}$$

so α^n is, indeed, natural for any choice of $n \in \mathbb{N}$. But then

$$\begin{aligned}
(\mathrm{pfix}\, \alpha_{X',Y'})(Gf \circ p \circ Gg) &= \sup_{n \in \omega} \left\{ \alpha^n_{X',Y'}(\perp_{X',Y'}, Gf \circ p \circ Gg) \right\} \\
&= \sup_{n \in \omega} \left\{ \alpha^n_{X',Y'}(Ff \circ \perp_{X,Y} \circ Fg, Gf \circ p \circ Gg) \right\} \\
&= \sup_{n \in \omega} \left\{ Ff \circ \alpha^n_{X,Y}(\perp_{X,Y}, p) \circ Fg \right\} \\
&= Ff \circ \sup_{n \in \omega} \left\{ \alpha^n_{X,Y}(\perp_{X,Y}, p) \right\} \circ Fg \\
&= Ff \circ (\mathrm{pfix}\, \alpha_{X,Y})(p) \circ Fg
\end{aligned}$$

so $\mathrm{pfix}\, \alpha_{X,Y}$ is natural as well. □

This theorem can be read as stating that, just like reversibility, a recursive polymorphic map can be obtained from one that is only locally polymorphic. Combining this result with Lemma 3.16 regarding self-conjugacy, we obtain the following corollary.

Corollary 3.19 *If* $\mathscr{C}(FX, FY) \times \mathscr{C}(GX, GY) \xrightarrow{\alpha_{X,Y}} \mathscr{C}(FX, FY)$ *is a self-conjugate natural transformation, so is* $\operatorname{pfix} \alpha_{X,Y}$.

Proof. If $\alpha_{X,Y} = \overline{\alpha_{Y,X}}$ for all X, Y then also $\operatorname{pfix} \alpha_{X,Y} = \operatorname{pfix} \overline{\alpha_{Y,X}}$, which is further natural in X and Y by Theorem 3.18. But then $\overline{\operatorname{pfix} \alpha_{X,Y}} = \operatorname{pfix} \overline{\alpha_{X,Y}} = \operatorname{pfix} \alpha_{Y,X}$, as parametrized fixed points preserve conjugation. □

4 Applications and future work

Reversible programming languages

Theseus [29] is a typed reversible functional programming language similar in syntax and spirit to Haskell. It has support for recursive data types, as well as reversible tail recursion using so-called *typed iteration labels* as syntactic sugar for a dagger trace operator. Theseus is based on the Π-family of reversible combinator calculi [28], which bases itself on dagger traced symmetric monoidal categories augmented with a certain class of algebraically ω-compact functors.

Theseus also supports *parametrized functions*, that is, families of reversible functions indexed by reversible functions of a given type, with the proviso that parameters must be passed to parametrized maps statically. For example, (if one extended Theseus with polymorphism) the reversible map function would have the signature $map :: (a \leftrightarrow b) \to ([a] \leftrightarrow [b])$, and so map is not in itself a reversible function, though $map\ \langle f \rangle$ is (for some suitable function f passed statically). This gives many of the benefits of higher-order programming, but without the headaches of higher-order reversible programming.

The presented results show very directly that we can extend Theseus with a fixed point operator for general recursion while maintaining desirable inversion properties, rather than making do with the simpler tail recursion. Additionally, the focus on the continuous functionals of $\mathscr{C}$ given by the category $\mathbf{DcpoOp}(\mathscr{C})$ also highlights the feature of parametrized functions in Theseus, and our results go further to show that even parametrized functions that use general recursion not only have desirable inversion properties, but also preserve naturality, the latter of which is useful for extending Theseus with parametric polymorphism.

Quantum programming languages

An interesting possibility as regards quantum programming languages is the category $\mathbf{CP}^*(\mathbf{FHilb})$ (see [13] for details on the $\mathbf{CP}^*$-construction), which is dagger compact closed and equivalent to the category of finite-dimensional C^*-algebras and completely positive maps [13]. Since finite-dimensional C^*-algebras are specifically von Neumann algebras, it follows (see [10,34]) that this category is enriched in the category of *bounded* directed complete partial orders; and since it inherits the dagger from **FHilb** (and is locally ordered by the pointwise extension of the Löwner order restricted to positive operators), the dagger structure is monotone, too. As such, the presented results ought to apply in this case as well – modulo concerns of boundedness – though this warrants more careful study.

Dagger traces in DCPO-†-categories

Given a suitable monoidal tensor (*e.g.*, one with the zero object as tensor unit) and a partial additive structure on morphisms, giving the category the structure of a *unique decomposition category* [18,24], a trace operator can be constructed by means of the so-called *trace formula*

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

where $f_{mn} = \rho_m \circ f \circ \iota_n$, with $X_n \xrightarrow{\iota_n} \bigoplus_{i\in I} X_i$ and $\bigoplus_{i\in I} X_i \xrightarrow{\rho_m} X_m$ are families of canonical *quasi-injections* respectively *quasi-projections* of the monoidal tensor. In previous work [5,31], the author (among others) demonstrated that a certain class of **DCPO**-†-categories, namely join inverse categories, had a dagger trace under suitably mild assumptions. It is conjectured that this theorem may be generalized to other **DCPO**-†-categories that are not necessarily inverse categories, again provided that certain assumptions are satisfied.

Involutive iteration categories

As it turned out that the category $\mathbf{DcpoOp}(\mathscr{C})$ of continuous functionals on $\mathscr{C}$ was both involutive and an iteration category, an immediate question to ask is how the involution functor ought to interact with parametrized fixed points in the general case. A remarkable fact of iteration categories is that they are defined to be cartesian categories that satisfy all equations of parametrized fixed points that hold in the category $\mathbf{CPO}_m$ of ω-complete partial orders and *monotone* functions, yet also have a complete (though infinite) equational axiomatization [16].

We have provided an example of an interaction between parametrized fixed points and the involution functor here, namely that $\mathbf{DcpoOp}(\mathscr{C})$ satisfies $\overline{\mathrm{pfix}\,\psi} = \mathrm{pfix}\,\overline{\psi}$. It could be interesting to search for examples of involutive iteration categories in the wild (as candidates for a semantic definition), and to see if Ésik's axiomatization could be extended to accomodate for the involution functor in the semantic category.

Algebraic compactness of dagger functors

Another useful feature of categories enriched in domains, as shown independently by Adámek [4] and Barr [7], is the algebraic compactness of locally continuous functors, provided that certain (co)completeness requirements are met. Since the way of the dagger dictates that fixed points of (dagger) functors ought to be unique up to *unitaries* (rather than up to any old isomorphism), the entire machinery of **DCPO**-categories developed for this purpose needs readjustment in order to accomodate for this requirement of uniqueness up to unitary maps. This is the topic of another paper by the author.

5 Conclusion and related work

We have developed a notion of **DCPO**-categories with a monotone dagger structure (of which **PInj**, **Rel**, and $\mathbf{DStoch}_{\leq 1}$ are examples, and $\mathbf{CP}^*(\mathbf{FHilb})$ is closely related), and shown that these categories can be taken to be enriched in an induced involutive monoidal category of continuous functionals. With this, we were able to account for (ordinary and parametrized) fixed point adjoints as arising from conjugation of the functional in the induced involutive monoidal category, to show that parametrized fixed points preserve conjugation and naturality, and that natural transformations that preserve the dagger are precisely those that are self-conjugate. We also described a number of potential applications in connection with reversible and quantum computing.

A great deal of work has been carried out in recent years on the domain theory of quantum computing, with noteworthy results in categories of von Neumann algebras (see, *e.g.*, [34,10,27,11]). Though the interaction between dagger structure and the domain structure on homsets was not the object of study, Heunen considers the similarities and differences of **FHilb** and **PInj**, also in relation to domain structure on homsets, in [22], though he also notes that **FHilb** fails to enrich in domains as composition is not even monotone (this is not to say that domain theory and quantum computing do not mix; only that **FHilb** is the wrong category to consider for this purpose). Finally, dagger traced symmetric monoidal categories, with the dagger trace serving as an operator for reversible tail recursion, have been studied in connection with reversible combinator calculi [28] and functional programming [29].

References

[1] Abramsky, S., *Retracing some paths in process algebra*, in: U. Montanari and V. Sassone, editors, *CONCUR '96*, Springer, 1996 pp. 1–17.

[2] Abramsky, S. and B. Coecke, *A categorical semantics of quantum protocol*, in: *Logic in Computer Science, 2004, Proceedings*, IEEE, 2004, pp. 415–425.

[3] Abramsky, S. and A. Jung, *Domain theory*, in: S. Abramsky, D. Gabbay and T. Maibaum, editors, *Handbook of Logic in Computer Science*, 3, Clarendon Press, 1994 pp. 1–168.

[4] Adámek, J., *Recursive data types in algebraically ω-complete categories*, Information and Computation **118** (1995), pp. 181–190.

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

[6] Bainbridge, E. S., P. J. Freyd, A. Scedrov and P. J. Scott, *Functorial polymorphism*, Theoretical Computer Science **70** (1990), pp. 35–64.

[7] Barr, M., *Algebraically compact functors*, Journal of Pure and Applied Algebra **82** (1992), pp. 211–231.

[8] Beggs, E. J. and S. Majid, *Bar categories and star operations*, Algebras and Representation Theory **12** (2009), pp. 103–152.

[9] Benton, N. and M. Hyland, *Traced premonoidal categories*, Theoretical Informatics and Applications **37** (2003), pp. 273–299.

[10] Cho, K., "Semantics for a Quantum Programming Language by Operator Algebras," Master's thesis, University of Tokyo (2014).

[11] Cho, K., B. Jacobs, B. Westerbaan and A. Westerbaan, *An Introduction to Effectus Theory* (2015), `arXiv:1512.05813 [cs.LO]`.

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

[13] Coecke, B., C. Heunen and A. Kissinger, *Categories of quantum and classical channels*, Quantum Information Processing **15** (2016), pp. 5179–5209.

[14] Egger, J., *Involutive monoidal categories and enriched dagger categories* (2008), seminar talk, University of Oxford, available at https://www.youtube.com/watch?v=75dTIkppk8Q (fetched Mar. 31., 2017).

[15] Ésik, Z., *Fixed point theory*, in: M. Droste, W. Kuich and H. Vogler, editors, *Handbook of Weighted Automata*, Springer, 2009 pp. 29–65.

[16] Ésik, Z., *Equational properties of fixed point operations in cartesian categories: An overview*, in: G. Italiano, G. Pighizzini and D. Sannella, editors, *MFCS 2015, Proceedings, Part I*, Springer, 2015 pp. 18–37.

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

[18] Haghverdi, E., *Unique decomposition categories, Geometry of Interaction and combinatory logic*, Mathematical Structures in Computer Science **10** (2000), pp. 205–230.

[19] Hasegawa, M., *Recursion from cyclic sharing: Traced monoidal categories and models of cyclic lambda calculi*, in: P. de Groote and J. R. Hindley, editors, *TLCA '97*, Lecture Notes in Computer Science **1210**, Springer, 1997 pp. 196–213.

[20] Hasegawa, M., M. Hofmann and G. Plotkin, *Finite dimensional vector spaces are complete for traced symmetric monoidal categories*, in: A. Avron, N. Dershowitz and A. Rabinovich, editors, *Pillars of Computer Science: Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday* (2008), pp. 367–385.

[21] Heunen, C., "Categorical quantum models and logics," Ph.D. thesis, Radboud University Nijmegen (2009).

[22] Heunen, C., *On the functor ℓ^2*, in: *Computation, Logic, Games, and Quantum Foundations – The Many Facets of Samson Abramsky*, Springer, 2013 pp. 107–121.

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

[24] Hoshino, N., *A representation theorem for unique decomposition categories*, Electronic Notes in Theoretical Computer Science **286** (2012), pp. 213–227.

[25] Hyland, M., *Abstract and concrete models for recursion*, in: O. Grumberg, T. Nipkow and C. Pfaller, editors, *Proceedings of the NATO Advanced Study Institute on Formal Logical Methods for System Security and Correctness* (2008), pp. 175–198.

[26] Jacobs, B., *Involutive categories and monoids, with a GNS-correspondence*, Foundations of Physics **42** (2012), pp. 874–895.

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

[28] James, R. P. and A. Sabry, *Information effects*, in: *POPL 2012, Proceedings* (2012), pp. 73–84.

[29] James, R. P. 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.

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

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

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

[33] Kelly, G. M., "Basic Concepts of Enriched Category Theory," London Mathematical Society Lecture Note Series **64**, Cambridge University Press, 1982.

[34] Rennela, M., *Towards a quantum domain theory: Order-enrichment and fixpoints in W*-algebras*, Electronic Notes in Theoretical Computer Science **308** (2014), pp. 289–307.

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

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

[37] Selinger, P., *Finite dimensional Hilbert spaces are complete for dagger compact closed categories*, Logical Methods in Computer Science **8** (2012), pp. 1–12.

[38] Smyth, M. B. and G. D. Plotkin, *The category-theoretic solution of recursive domain equations*, SIAM Journal on Computing **11** (1982), pp. 761–783.

[39] Wadler, P., *Theorems for free!*, in: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89 (1989), pp. 347–359.

When do I see a photograph, when a reflection?

Philip K. Dick, *A Scanner Darkly*

C

Semantics of reversible programming languages

This chapter contains three papers concerning the semantics of reversible programming languages.

(**C1**) R. Glück and R. Kaarsgaard. A categorical foundation for structured reversible flowchart languages: Soundness and adequacy. Unpublished manuscript, in review, 2017.

(**C2**) C. Heunen, R. Kaarsgaard, and M. Karvonen. Reversible effects as inverse arrows. Unpublished manuscript, in review, 2017.

(**C3**) R. Kaarsgaard and M. K. Thomsen. RFun Revisited. In Marina Waldén, editor, *Proceedings of the 29th Nordic Workshop on Programming Theory*, TUCS Lecture Notes No. 27, pages 65–67, Turku Centre for Computer Science, 2017.

Paper (**C1**) is an extended version of the following conference paper:

- R. Glück and R. Kaarsgaard. A categorical foundation for structured reversible flowchart languages. In *Proceedings of the 33rd Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIII)*, to appear, 2017.

Note that in (**C2**), the term "reversible" is used in a different way than in the introduction. Rather than following the Copenhagen interpretation of reversible computing (see page 2), it is instead used (consistent with, *e.g.*, [62, 64]) to mean that each morphism $f : X \to Y$

has an adjoint $f^\dagger : Y \to X$ such that this gives the category a dagger structure (see Definition 8 on page 4).

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

ROBERT GLÜCK AND ROBIN KAARSGAARD

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

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

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

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

1. Introduction

Reversible computing is an emerging paradigm that adopts a physical principle of reality into a *computation model without information erasure*. Reversible computing extends the standard forward-only mode of computation with the ability to execute in reverse as easily as forward. Reversible computing is a necessity in the context of quantum computing and some bio-inspired computation models. Regardless of the physical motivation, bidirectional determinism is interesting in its own right. The potential benefits include the design of innovative reversible architectures (*e.g.*, [28, 27, 30]), new programming models and techniques (*e.g.*, [32, 15, 23]), and the enhancement of software with reversibility (*e.g.*, [6]).

1998 ACM Subject Classification: D.3.1, F.3.2.

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

This is the extended version of an article presented at MFPS XXXIII [14], extended with proofs that previously appeared in the appendix, as well as new sections on soundness, adequacy, and full abstraction.

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

LOGICAL METHODS IN COMPUTER SCIENCE   DOI:10.2168/LMCS-???

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

This paper aims at providing a new categorical foundation specifically for formalizing reversible programming languages, in particular the semantics of reversible structured flowchart languages [29], which are the reversible counterpart of the structured programming languages used today. This formalization is based on join inverse categories with a developed notion of *extensivity* for inverse categories, which gives rise to natural representations of predicates and assertions, and consequently to models of reversible structured flowcharts. The goal is to provide a framework for modelling these languages, such that the reversible semantic properties of the object language are naturally ensured by the meta language.

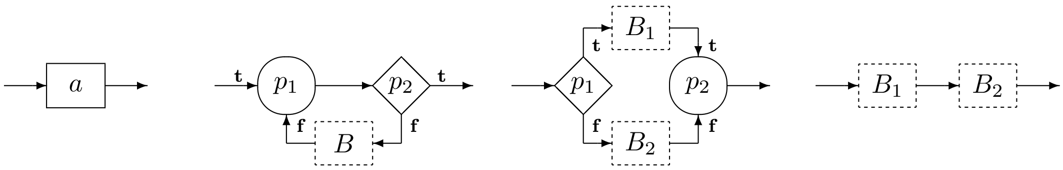
The semantic framework we are going to present in this paper covers the reversible structured languages regardless of their concrete formation, such as atomic operations, elementary predicates, and value domains. Reversible programming languages that are instances of this computation model include the imperative language Janus [32] without recursion, and the while languages `R-WHILE` and `R-CORE` with dynamic data structures [16, 17]. Further, unstructured reversible flowchart languages, such as reversible assembly languages with jumps [12, 3], can be transformed into structured ones thanks to the structured reversible program theorem [29].

Overview: In Section 2, we give an introduction to structured reversible flowchart languages, while Section 3 describes the restriction and inverse category theory used as backdrop in later sections. In Section 4, we warm up by developing a notion of extensivity for inverse categories, based on extensive restriction categories and its associated concept of *decisions*. Then, in Section 5, we put it all to use by showing how decisions may be used to model predicates and ultimately also reversible flowcharts, and we show that these are sound and adequate with respect to the operational semantics in Section 6. In Section 7, we extend the previous theorems by giving a sufficient condition for full abstraction. In Section 8, we show how to extract a program inverter from the categorical semantics, develop a small language to exemplify our framework, and discuss other applications in reversible programming. Section 9 offers some concluding remarks.

2. Reversible structured flowcharts

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

A structured reversible flowchart F is built from four blocks (Figure 1): An *atomic step* that performs an elementary operation on a domain X specified by a partial injective function $a : X \rightharpoonup X$; a *while loop* over a block B with entry assertion $p_1 : X \to \mathit{Bool}$ and exit test $p_2 : X \to \mathit{Bool}$; a *selection* of block B_1 or B_2 with entry test $p_1 : X \to \mathit{Bool}$ and exit



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

Figure 1: Structured reversible flowcharts.

assertion $p_2 : X \to Bool$; and a *sequence* of blocks B_1 and B_2. The operational semantics of these are shown in Figure 2.

A structured reversible flowchart F consists of one main block. Blocks have unique entry and exit points, and can be nested any number of times to form more complex flowcharts. The interpretation of F consists of a given domain X (typically, a store) and a finite set of partial injective functions a and predicates $p : X \to Bool$. Computation starts at the entry point of F in an initial x_0 (the input), proceeds sequentially through the edges of F, and ends at the exit point of F in a final x_n (the output), if F is defined on the given input. Though the specific set of predicates depend on the flowchart language, they are often (as we will do here) assumed to be closed under Boolean operators, in particular conjunction and negation. The operational semantics for these are the same as in the irreversible case; see Figure 3.

The assertion p_1 in a reversible while loop (marked by the circle [32]) is a new flowchart operator: the predicate p_1 must be *true* when the control flow reaches the assertion along the **t**-edge, and *false* when it reaches the assertion along the **f**-edge; otherwise, the loop is undefined. The test p_2 (marked by a diamond) has the usual semantics. This means that B in a loop is repeated as long as p_1 and p_2 are *false*.

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

Despite their simplicity, reversible structured flowcharts are *reversibly universal* [2], which means that they are computationally as powerful as any reversible programming language can be. Given a suitable domain X for finite sets of atomic operations and predicates, there exists, for every injective computable function $f : X \to Y$, a reversible flowchart F that computes f.

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

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

It is often assumed, as we will do here, that the set of atomic steps contains a step **skip** that acts as the identity. Reversible structured flowcharts defined above corresponds to the reversible language `R-WHILE` [16], but their value domain, atomic functions and predicates are unspecified. As a minimum, a reversible flowchart needs blocks (a,b,d) because selection (c) can be simulated by combining while loops that conditionally skip the body block or execute it once. `R-CORE` [17] is an example of such a minimal language.

$$\boxed{\sigma \vdash p \downarrow \sigma'}$$

$$\frac{}{\sigma \vdash \textbf{skip} \downarrow \sigma} \qquad \frac{\sigma \vdash c_1 \downarrow \sigma' \qquad \sigma' \vdash c_2 \downarrow \sigma''}{\sigma \vdash c_1 \,;\, c_2 \downarrow \sigma''}$$

$$\frac{\sigma \vdash p \downarrow tt \qquad \sigma \vdash c_1 \downarrow \sigma' \qquad \sigma' \vdash q \downarrow tt}{\sigma \vdash \textbf{if}\ p\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \textbf{fi}\ q \downarrow \sigma'} \qquad \frac{\sigma \vdash p \downarrow f\!f \qquad \sigma \vdash c_2 \downarrow \sigma' \qquad \sigma' \vdash q \downarrow f\!f}{\sigma \vdash \textbf{if}\ p\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \textbf{fi}\ q \downarrow \sigma'}$$

$$\frac{\sigma \vdash p \downarrow tt \qquad \sigma \vdash q \downarrow tt}{\sigma \vdash \textbf{from}\ p\ \textbf{loop}\ c\ \textbf{until}\ q \downarrow \sigma} \qquad \frac{\sigma \vdash p \downarrow f\!f \qquad \sigma \vdash q \downarrow tt}{\sigma \vdash \underline{loop}[p, c, q] \downarrow \sigma}$$

$$\frac{\sigma \vdash p \downarrow f\!f \qquad \sigma \vdash q \downarrow f\!f \qquad \sigma \vdash c \downarrow \sigma' \qquad \sigma' \vdash \underline{loop}[p, c, q] \downarrow \sigma''}{\sigma \vdash \underline{loop}[p, c, q] \downarrow \sigma''}$$

$$\frac{\sigma \vdash p \downarrow tt \qquad \sigma \vdash q \downarrow f\!f \qquad \sigma \vdash c \downarrow \sigma' \qquad \sigma' \vdash \underline{loop}[p, c, q] \downarrow \sigma''}{\sigma \vdash \textbf{from}\ p\ \textbf{loop}\ c\ \textbf{until}\ q \downarrow \sigma''}$$

Figure 2: Operational semantics for the reversible flowchart structures.

$$\boxed{\sigma \vdash p \downarrow b}$$

$$\frac{}{\sigma \vdash tt \downarrow tt} \qquad \frac{}{\sigma \vdash f\!f \downarrow f\!f}$$

$$\frac{\sigma \vdash p \downarrow tt}{\sigma \vdash \textbf{not}\ p \downarrow f\!f} \qquad \frac{\sigma \vdash p \downarrow f\!f}{\sigma \vdash \textbf{not}\ p \downarrow tt}$$

$$\frac{\sigma \vdash p \downarrow tt \qquad \sigma \vdash q \downarrow tt}{\sigma \vdash p\ \textbf{and}\ q \downarrow tt} \qquad \frac{\sigma \vdash p \downarrow tt \qquad \sigma \vdash q \downarrow f\!f}{\sigma \vdash p\ \textbf{and}\ q \downarrow f\!f}$$

$$\frac{\sigma \vdash p \downarrow f\!f \qquad \sigma \vdash q \downarrow tt}{\sigma \vdash p\ \textbf{and}\ q \downarrow f\!f} \qquad \frac{\sigma \vdash p \downarrow f\!f \qquad \sigma \vdash q \downarrow f\!f}{\sigma \vdash p\ \textbf{and}\ q \downarrow f\!f}$$

Figure 3: Operational semantics for Boolean predicates.

3. Restriction and inverse categories

The following section contains the background on restriction and inverse category theory necessary for our later developments. Unless otherwise specified, the definitions and results presented in this section can be found in introductory texts on the subject (*e.g.*, [13, 18, 8, 9, 10]).

Restriction categories [8, 9, 10] axiomatize categories of partial maps. This is done by assigning to each morphism f a *restriction idempotent* $\overline{f}$, which we think of as a partial identity defined precisely where f is. Formally, restriction categories are defined as follows.

Definition 1. A *restriction category* is a category $\mathscr{C}$ equipped with a combinator mapping each morphism $A \xrightarrow{f} B$ to a morphism $A \xrightarrow{\overline{f}} A$ satisfying

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

for all suitable g.

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

In restriction categories, we say that a morphism $A \xrightarrow{f} B$ is *total* if $\overline{f} = \mathrm{id}_A$, and a *partial isomorphism* if there exists a (necessarily unique) *partial inverse* $B \xrightarrow{f^\dagger} A$ such that $f^\dagger f = \overline{f}$ and $ff^\dagger = \overline{f^\dagger}$. Isomorphisms are then simply the total partial isomorphisms with total partial inverses. An inverse category can then be defined as a special kind of restriction category[1].

Definition 2. An *inverse category* is a restriction category where each morphism is a partial isomorphism.

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

Since each morphism in an inverse category has a unique partial inverse, as also suggested by our notation this makes inverse categories canonically *dagger categories* [25], in the sense that they come equipped with a contravariant endofunctor $(-)^\dagger$ satisfying $f = f^{\dagger\dagger}$ and $\mathrm{id}_A^\dagger = \mathrm{id}_A$ for each morphism f and object A.

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

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

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

[1]This is a rather modern definition due to [8]. Originally, inverse categories were defined as the categorical extensions of inverse semigroups; see [22].

Definition 3. In a restriction category, say that parallel maps f and g are *disjoint* iff $f\overline{g} = 0$; and *compatible* iff $f\overline{g} = g\overline{f}$.

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

Definition 4. A restriction category $\mathscr{C}$ has compatible (disjoint) joins if it has a restriction zero, and satisfies that for each compatible (disjoint) subset S of any hom set $\mathscr{C}(A, B)$, there exists a morphism $\bigvee_{s\in S} s$ such that

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

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

Definition 5. In an inverse category, say that parallel maps f and g are *disjoint* iff $f\overline{g} = 0$ and $f^\dagger\overline{g^\dagger} = 0$; and *compatible* iff $f\overline{g} = g\overline{f}$ and $f^\dagger\overline{g^\dagger} = g^\dagger\overline{f^\dagger}$.

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

Definition 6. An inverse category $\mathscr{C}$ has compatible (disjoint) joins if it has a restriction zero and satisfies that for all compatible (disjoint) subsets S of all hom sets $\mathscr{C}(A, B)$, there exists a morphism $\vee_{s\in S} s$ satisfying (i) – (iv) of Definition 4.

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

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

In [10], it is shown that the existence of certain maps, called *decisions*, in a restriction category $\mathscr{C}$ with restriction coproducts leads to the subcategory Total($\mathscr{C}$) of total maps being extensive (in the sense of, *e.g.*, [5]). This leads to the definition of an *extensive restriction category*[2].

Definition 7. A restriction category is said to be *extensive* (as a restriction category) if it has restriction coproducts and a restriction zero, and for each map $A \xrightarrow{f} B + C$ there is a unique *decision* $A \xrightarrow{\langle f\rangle} A + A$ satisfying

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

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

In the above, ∇ denotes the codiagonal $[\mathrm{id},\mathrm{id}]$. A consequence of these axioms is that each decision is a partial isomorphism; one can show that $\langle f\rangle$ must be partial inverse to $[\overline{\kappa_1^\dagger f}, \overline{\kappa_2^\dagger f}]$ (see [10]). Further, when a restriction category with restriction coproducts has finite joins, it is also extensive with $\langle f\rangle = \kappa_1\overline{\kappa_1^\dagger f} \vee \kappa_2\overline{\kappa_2^\dagger f}$. As an example, **Pfn** is extensive with $A \xrightarrow{\langle f\rangle} A+A$ for $A \xrightarrow{f} B+C$ given by

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

While inverse categories only have coproducts (much less restriction coproducts) in very degenerate cases (see [13]), they may very well be equipped with a more general sum-like symmetric monoidal tensor, a disjointness tensor.

Definition 8. A *disjointness tensor* on a restriction category is a symmetric monoidal restriction functor $-\oplus-$ satisfying that its unit is the restriction zero, and that the canonical maps

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

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

It can be straightforwardly shown that any restriction coproduct gives rise to a disjointness tensor. A useful interaction between compatible joins and a join-preserving disjointness tensor in inverse categories was shown in [4, 21], namely that it leads to a †-trace (in the sense of [20, 26]):

Proposition 1. *Let $\mathscr{C}$ be an inverse category with (at least countable) compatible joins and a join-preserving disjointness tensor. Then $\mathscr{C}$ has a trace operator given by*

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

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

4. Extensivity of inverse categories

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

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

Recall that decisions satisfy the following two axioms:

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

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

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

Note that this axiom also subtly states that disjoint joins of the given form always exist.

Say that a restriction category is *pre-extensive* if it has restriction coproducts, a restriction zero, and a combinator mapping each map $A \xrightarrow{f} B+C$ to a *pre-decision* $A \xrightarrow{\langle f\rangle} A+A$ (with no additional requirements). We can then show the following:

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

(i) $\mathscr{C}$ is an extensive restriction category.
*(ii) Every pre-decision of $\mathscr{C}$ satisfies **(D.1)** and **(D.2)**.*
*(iii) Every pre-decision of $\mathscr{C}$ satisfies **(D'.1)** and **(D.2)**.*

To show this theorem, we will need the following lemma:

Lemma 4.2. *In an extensive restriction category, joins of the form $(f+0)\vee(0+g)$ exist for all maps $A \xrightarrow{f} B$ and $C \xrightarrow{g} D$ and are equal to $f+g$.*

Proof. By [7], for any map $A \xrightarrow{h} B+C$ in an extensive restriction category, $h = (\overline{\kappa_1^\dagger h}) \vee (\overline{\kappa_2^\dagger h})$. But then $f+g = (\overline{\kappa_1^\dagger f+g}) \vee (\overline{\kappa_2^\dagger f+g}) = (\mathrm{id}+0f+g)\vee(0+\mathrm{id}f+g) = (f+0)\vee(0+g)$. □

We can now continue with the proof.

Proof. The equivalence between (i) and (ii) was given in [10]. That (ii) and (iii) are equivalent follows by

$$\begin{aligned}(\kappa_1^\dagger\langle f\rangle)\vee(\kappa_2^\dagger\langle f\rangle) &= ([\mathrm{id},0]\langle f\rangle)\vee([0,\mathrm{id}]\langle f\rangle)\\ &= (\nabla\mathrm{id}+0\langle f\rangle)\vee(\nabla 0+\mathrm{id}\langle f\rangle)\\ &= \nabla(\mathrm{id}+0\vee 0+\mathrm{id})\langle f\rangle\\ &= \nabla(\mathrm{id}+\mathrm{id})\langle f\rangle = \nabla\langle f\rangle\end{aligned}$$

where we note that the join $\mathrm{id}+0\vee 0+\mathrm{id}$ exists and equals $\mathrm{id}+\mathrm{id}$ when every pre-decision satisfies **(D.1)** and **(D.2)** by Lemma 4.2. That the join also exists when every pre-decision satisfies **(D'.1)** and **(D.2)** follows as well, since the universal mapping property for coproducts guarantees that the only map g satisfying $(\kappa_1+\kappa_2)+(\kappa_1+\kappa_2)g = (\kappa_1+\kappa_2)(\kappa_1+\kappa_2)$ is $\kappa_1+\kappa_2$ itself, so we must have $\langle\kappa_1+\kappa_2\rangle = \kappa_1+\kappa_2$, and

$$\begin{aligned}(\mathrm{id}+0)\vee(0+\mathrm{id}) &= \overline{\kappa_1^\dagger}\vee\overline{\kappa_2^\dagger} = (\kappa_1\kappa_1^\dagger)\vee(\kappa_2\kappa_2^\dagger)\\ &= (\kappa_1^\dagger(\kappa_1+\kappa_2))\vee\kappa_2^\dagger(\kappa_1+\kappa_2))\\ &= \overline{\kappa_1+\kappa_2} = \mathrm{id}+\mathrm{id}\end{aligned}$$

which was what we wanted. □

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

Our next snag in transporting this definition to inverse categories has to do with the restriction coproducts themselves, as it is observed in [13] that any inverse category with restriction coproducts is a preorder. Intuitively, the problem is not that unicity of coproduct maps cannot be guaranteed in non-preorder inverse categories, but rather that the coproduct map $A + B \xrightarrow{[f,g]} C$ in a restriction category is not guaranteed to be a partial isomorphism when f and g are.

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

Definition 9. An inverse category with a disjointness tensor is said to be *extensive* when each map $A \xrightarrow{f} B \oplus C$ has a unique decision $A \xrightarrow{\langle f\rangle} A \oplus A$ satisfying

(D'.1): $(\amalg_1^\dagger\langle f\rangle) \vee (\amalg_2^\dagger\langle f\rangle) = \overline{f}$
(D'.2): $(f \oplus f)\langle f\rangle = (\amalg_1 \oplus \amalg_2)f$.

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

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

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

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

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

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

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

5. Modelling structured reversible flowcharts

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

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

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

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

$$\begin{aligned} \llbracket tt \rrbracket &= \amalg_1 \\ \llbracket ff \rrbracket &= \amalg_2 . \end{aligned}$$

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

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

$$\llbracket \textbf{not } p \rrbracket = \gamma \llbracket p \rrbracket .$$

With this, it is straightforward to verify that, *e.g.*, $\llbracket \textbf{not } tt \rrbracket = \llbracket ff \rrbracket$, as

$$\llbracket \textbf{not } tt \rrbracket = \gamma \amalg_1 = \gamma \mathrm{id} \oplus 0 \rho^{-1} = 0 \oplus \mathrm{id} \gamma \rho^{-1} = 0 \oplus \mathrm{id} \lambda^{-1} = \amalg_2 = \llbracket ff \rrbracket .$$

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

$$\llbracket p \textbf{ and } q \rrbracket = (\amalg_1 \overline{\amalg_1^\dagger \llbracket p \rrbracket}\; \overline{\amalg_1^\dagger \llbracket q \rrbracket}) \vee (\amalg_2 (\overline{\amalg_2^\dagger \llbracket p \rrbracket} \vee \overline{\amalg_2^\dagger \llbracket q \rrbracket})) \overline{\llbracket p \rrbracket}\; \overline{\llbracket q \rrbracket} .$$

The intuition behind this definition is that the witnesses of a conjunction of predicates is given by the meet of the witnesses of the each predicate, while the counterexamples of a conjunction of predicates is the join of the counterexamples of each predicate. Note that this is then precomposed with $\overline{\llbracket p \rrbracket}\; \overline{\llbracket q \rrbracket}$ to ensure that the result is only defined where both p and q are; this gives

Noting that the meet of two restriction idempotents is given by their composition, this is precisely what this definition states. Similarly we define the disjunction of $\llbracket p \rrbracket$ and $\llbracket q \rrbracket$ by

$$\llbracket p \textbf{ or } q \rrbracket = (\amalg_1(\overline{\amalg_1^\dagger \llbracket p \rrbracket} \vee \overline{\amalg_1^\dagger \llbracket q \rrbracket})) \vee (\amalg_2(\overline{\amalg_2^\dagger \llbracket p \rrbracket}\ \overline{\amalg_2^\dagger \llbracket q \rrbracket}))\overline{\llbracket p \rrbracket}\ \overline{\llbracket q \rrbracket},$$

as $\llbracket p \textbf{ or } q \rrbracket$ then has as witnesses the join of the witnesses of $\llbracket p \rrbracket$ and $\llbracket q \rrbracket$, and as counterexamples the meet of the counterexamples of $\llbracket p \rrbracket$ and $\llbracket q \rrbracket$. With these definitions, it can be shown that, *e.g.*, the De Morgan laws are satisfied. However, since we can thus construct this from conjunctions and negations, we will leave disjunctions as syntactic sugar.

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

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

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

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

Likewise, elementary predicates (*e.g.*, comparison of values in a store) may vary from language to language, so we assume that such elementary predicates have denotations as well as decisions $\Sigma \to \Sigma \oplus \Sigma$. If necessary (as is the case for Janus [32]), we may then close these elementary predicates under boolean combinations as discussed in the previous section.

To start, we note how sequencing of flowcharts may be modelled trivially by means of composition, *i.e.*,

$$\llbracket c_1 \ ; c_2 \rrbracket = \llbracket c_2 \rrbracket \llbracket c_1 \rrbracket$$

or, using the diagrammatic notation of flowcharts and the string diagrams for monoidal categories in the style of [26] (read left-to-right and bottom-to-top),

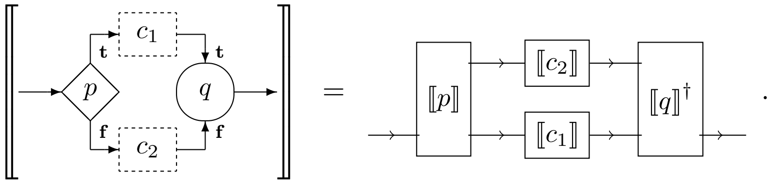
$$\llbracket \to \boxed{c_1} \to \boxed{c_2} \to \rrbracket \quad = \quad \to \boxed{\llbracket c_1 \rrbracket} \to \boxed{\llbracket c_2 \rrbracket} \to \quad .$$

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

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

$$\llbracket \textbf{if } p \textbf{ then } c_1 \textbf{ else } c_2 \textbf{ fi } q \rrbracket = \llbracket q \rrbracket^\dagger \llbracket c_1 \rrbracket \oplus \llbracket c_2 \rrbracket \llbracket p \rrbracket$$

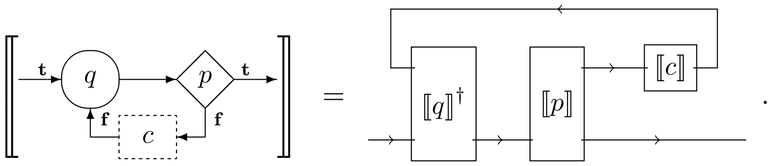
or, as diagrams



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

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

or diagrammatically



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

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

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

In order to be able to provide a correspondence between categorical and operational semantics, we also need an interpretation of the meta-command $\underline{loop}$. While it may not be so clear at the present, it turns out that the appropriate one is

$$[\![\underline{loop}[p,c,q]]\!] = \bigvee_{n\in\omega} \beta[p,c,q]_{21} \beta[p,c,q]_{22}^{n}$$

where $\beta[p,c,q] = \mathrm{id}_\Sigma \oplus [\![c]\!]\, [\![q]\!]\, [\![p]\!]^\dagger$, *i.e.*, the inner part of the interpretation of the **from**-loop.

While it may seem like a small point, the mere existence of a categorical semantics in inverse categories for a reversible programming language has some immediate benefits. In particular, that a programming language is reversible can be rather complicated to show by means of operational semantics (see, *e.g.*, [32, Sec. 2.3]), yet it follows directly in our categorical semantics, as all morphisms in inverse categories have a unique partial inverse.

6. Soundness and adequacy

Soundness and adequacy (see, *e.g.*, [11]) are the two fundamental properties of operational semantics with respect to their denotational counterparts, as soundness and completeness are for proof systems with respect to their semantics. In brief, soundness and adequacy state that the respective notions of *convergence* of the operational and denotational semantics are in agreement.

In the operational semantics, the notion of convergence seems straightforward: a program p converges in a state σ if there exists another state σ' such that $\sigma \vdash p \downarrow \sigma'$. On the denotational side, it seems less obvious what a proper notion of convergence is.

An idea (used by, *e.g.*, Fiore [11]) is to let values (in this case, states) be interpreted as *total* morphisms from some sufficiently simple object I into an appropriate object V (here, we will use our object Σ of states). In this context, the notion of convergence for a program p in a state σ is then that the resulting value (state) $[\![p]\!]\,[\![\sigma]\!]$ is again, a state – *i.e.*, it is total. Naturally, this approach requires machinery to separate total maps from partial ones. As luck would have it, inverse categories fit the bill perfectly, as they can be regarded as special instances of restriction categories.

To make this idea more clear in the current context, and to allow us to use the established formulations of soundness and adequacy, we define a model of a structured reversible flowchart language to be the following:

Definition 10. A model of a structured reversible flowchart language $\mathcal{L}$ consists of a join inverse category $\mathscr{C}$ with a disjointness tensor, further equipped with distinguished objects I and Σ satisfying

(i) the identity and zero maps on I are distinct, *i.e.*, $\mathrm{id}_I \neq 0_{I,I}$,
(ii) if $I \xrightarrow{e} I$ is a restriction idempotent then $e = \mathrm{id}_I$ or $e = 0_{I,I}$, and
(iii) each $\mathcal{L}$-state σ is interpreted as a *total* morphism $I \xrightarrow{[\![\sigma]\!]} \Sigma$.

Here, we think of I as the *indexing object*, and Σ as the *object of states*. In irreversible programming languages, the first two conditions in the definition above are often left out, as the indexing object is typically chosen to be the terminal object 1. However, terminal objects are degenerate in inverse categories, as they always coincide with the initial object when they exist – that is, they are zero objects. For this reason, we require instead the existence of a sufficiently simple indexing object, as described by these two properties. For example, in **PInj**, any one-element set will satisfy these conditions.

Even further, the third condition is typically proven rather than assumed. We include it here as an assumption since structured reversible flowchart languages may take many different forms, and we have no way of knowing how the concrete states are formed. As such, rather than limiting ourselves to languages where states take a certain form in order to show totality of interpretation, we instead assume it to be able to show properties about more programming languages.

This also leads us to another important point: We are only able to show soundness and adequacy for the operational semantics as they are stated, *i.e.*, we are not able to take into account the specific atomic steps (besides **skip**) or elementary predicates of the language.

As such, soundness and adequacy (and what may follow from that) should be understood *conditionally*: If a structured reversible flowchart language has a model of the form above *and* it is sound and adequate with respect to its atomic steps and elementary predicates, then the entire interpretation is sound and adequate as well.

We begin by recalling the definition of the denotation of predicates and commands in a model of a structured reversible flowchart language from Section 5.

Definition 11. Recall the interpretation of predicates in $\mathcal{L}$ as decisions in $\mathscr{C}$:

(i) $[\![tt]\!] = \amalg_1$,
(ii) $[\![ff]\!] = \amalg_2$,
(iii) $[\![\textbf{not } p]\!] = \gamma\,[\![p]\!]$,

(iv) $\llbracket p \textbf{ and } q \rrbracket = \left(\amalg_1 \overline{\amalg_1^\dagger \llbracket p \rrbracket}\ \overline{\amalg_1^\dagger \llbracket q \rrbracket}\right) \vee \left(\amalg_2 \left(\overline{\amalg_2^\dagger \llbracket p \rrbracket} \vee \overline{\amalg_2^\dagger \llbracket q \rrbracket}\right)\right) \overline{\llbracket p \rrbracket}\ \overline{\llbracket q \rrbracket}$.

Definition 12. Recall the interpretation of commands in $\mathcal{L}$ (and the meta-command $\underline{loop}$) as morphisms $\Sigma \to \Sigma$ in $\mathscr{C}$:

(i) $\llbracket \textbf{skip} \rrbracket = \mathrm{id}_\Sigma$,
(ii) $\llbracket c_1 \ ; \ c_2 \rrbracket = \llbracket c_2 \rrbracket \llbracket c_1 \rrbracket$,
(iii) $\llbracket \textbf{if } p \textbf{ then } c_1 \textbf{ else } c_2 \textbf{ fi } q \rrbracket = \llbracket q \rrbracket^\dagger (\llbracket c_1 \rrbracket \oplus \llbracket c_2 \rrbracket) \llbracket p \rrbracket$,
(iv) $\llbracket \textbf{from } p \textbf{ loop } c \textbf{ until } q \rrbracket = \mathrm{Tr}^\Sigma_{\Sigma,\Sigma}(\beta[p,c,q])$, and
(v) $\llbracket \underline{loop}[p,c,q] \rrbracket = \bigvee_{n\in\omega} \beta[p,c,q]_{21} \beta[p,c,q]_{22}^n$

where $\beta[p,c,q] = (\mathrm{id}_\Sigma \oplus \llbracket c \rrbracket) \llbracket q \rrbracket \llbracket p \rrbracket^\dagger : \Sigma \oplus \Sigma \to \Sigma \oplus \Sigma$. Note also that $f_{ij} = \amalg_j^\dagger f \amalg_i$.

The overall strategy we will use to show soundness and adequacy for programs is to start by showing it for predicates. To begin to tackle this, we first need a lemma regarding the totality of predicates.

Lemma 6.1. *Let p and q be $\mathcal{L}$-predicates. It is the case that*

(i) $I \xrightarrow{\llbracket \sigma \rrbracket} \Sigma \xrightarrow{\llbracket tt \rrbracket} \Sigma \oplus \Sigma$ *and* $I \xrightarrow{\llbracket \sigma \rrbracket} \Sigma \xrightarrow{\llbracket f\!f \rrbracket} \Sigma \oplus \Sigma$ *are total,*
(ii) $I \xrightarrow{\llbracket \sigma \rrbracket} \Sigma \xrightarrow{\llbracket \textbf{not } p \rrbracket} \Sigma \oplus \Sigma$ *is total iff* $I \xrightarrow{\llbracket \sigma \rrbracket} \Sigma \xrightarrow{\llbracket p \rrbracket} \Sigma \oplus \Sigma$ *is, and*
(iii) $I \xrightarrow{\llbracket \sigma \rrbracket} \Sigma \xrightarrow{\llbracket p \textbf{ and } q \rrbracket} \Sigma \oplus \Sigma$ *is total iff* $I \xrightarrow{\llbracket \sigma \rrbracket} \Sigma \xrightarrow{\llbracket p \rrbracket} \Sigma \oplus \Sigma$ *and* $I \xrightarrow{\llbracket \sigma \rrbracket} \Sigma \xrightarrow{\llbracket q \rrbracket} \Sigma \oplus \Sigma$ *both are.*

Proof. For (i), it follows that

$$\overline{\llbracket tt \rrbracket \llbracket \sigma \rrbracket} = \overline{\amalg_1 \llbracket \sigma \rrbracket} = \overline{\overline{\amalg_1} \llbracket \sigma \rrbracket} = \overline{\mathrm{id}_\Sigma \llbracket \sigma \rrbracket} = \overline{\llbracket \sigma \rrbracket} = \mathrm{id}_I,$$

where the final equality follows by the definition of a model. The case for $f\!f$ is entirely analogous.

For (ii), we have that

$$\overline{\llbracket \textbf{not } p \rrbracket \llbracket \sigma \rrbracket} = \overline{\gamma \llbracket p \rrbracket \llbracket \sigma \rrbracket} = \overline{\overline{\gamma} \llbracket p \rrbracket \llbracket \sigma \rrbracket} = \overline{\mathrm{id}_{\Sigma\oplus\Sigma} \llbracket p \rrbracket \llbracket \sigma \rrbracket} = \overline{\llbracket p \rrbracket \llbracket \sigma \rrbracket}$$

which implies directly that $I \xrightarrow{\llbracket \sigma \rrbracket} \Sigma \xrightarrow{\llbracket \textbf{not } p \rrbracket} \Sigma \oplus \Sigma$ is total iff $I \xrightarrow{\llbracket \sigma \rrbracket} \Sigma \xrightarrow{\llbracket p \rrbracket} \Sigma \oplus \Sigma$ is.

For (iii), it suffices to show that $\overline{\llbracket p \textbf{ and } q \rrbracket \llbracket \sigma \rrbracket} = \overline{\llbracket p \rrbracket \llbracket \sigma \rrbracket}\,\overline{\llbracket q \rrbracket \llbracket \sigma \rrbracket}$, since $\overline{\llbracket p \rrbracket \llbracket \sigma \rrbracket} = \overline{\llbracket q \rrbracket \llbracket \sigma \rrbracket} = \mathrm{id}_I$ then yields $\overline{\llbracket p \textbf{ and } q \rrbracket \llbracket \sigma \rrbracket} = \mathrm{id}_I \mathrm{id}_I = \mathrm{id}_I$ directly; the other direction follows by the fact that if $\overline{g}\,\overline{f} = \mathrm{id}$ then $\mathrm{id} = \overline{g}\,\overline{f} = \overline{g}\,\overline{f f} = \mathrm{id}\overline{f} = \overline{f}$ (and analogously for g).

We start by observing that

$$\begin{aligned}
\overline{\llbracket p \rrbracket \llbracket \sigma \rrbracket} &= \overline{((\amalg_1 \overline{\amalg_1^\dagger \llbracket p \rrbracket}) \vee (\amalg_2 \overline{\amalg_2^\dagger \llbracket p \rrbracket})) \llbracket \sigma \rrbracket} \\
&= \overline{((\amalg_1 \overline{\amalg_1^\dagger \llbracket p \rrbracket} \llbracket \sigma \rrbracket) \vee (\amalg_2 \overline{\amalg_2^\dagger \llbracket p \rrbracket} \llbracket \sigma \rrbracket))} \\
&= \overline{((\amalg_1 \llbracket \sigma \rrbracket\, \overline{\amalg_1^\dagger \llbracket p \rrbracket \llbracket \sigma \rrbracket}) \vee (\amalg_2 \llbracket \sigma \rrbracket\, \overline{\amalg_2^\dagger \llbracket p \rrbracket \llbracket \sigma \rrbracket}))} \\
&= (\overline{(\amalg_1 \llbracket \sigma \rrbracket\, \overline{\amalg_1^\dagger \llbracket p \rrbracket \llbracket \sigma \rrbracket})} \vee \overline{(\amalg_2 \llbracket \sigma \rrbracket\, \overline{\amalg_2^\dagger \llbracket p \rrbracket \llbracket \sigma \rrbracket}))} \\
&= (\overline{(\overline{\amalg_1 \llbracket \sigma \rrbracket}\, \overline{\amalg_1^\dagger \llbracket p \rrbracket \llbracket \sigma \rrbracket})} \vee \overline{(\overline{\amalg_2 \llbracket \sigma \rrbracket}\, \overline{\amalg_2^\dagger \llbracket p \rrbracket \llbracket \sigma \rrbracket}))} \\
&= \overline{\overline{\amalg_1^\dagger \llbracket p \rrbracket \llbracket \sigma \rrbracket}} \vee \overline{\overline{\amalg_2^\dagger \llbracket p \rrbracket \llbracket \sigma \rrbracket}} \\
&= \overline{\amalg_1^\dagger \llbracket p \rrbracket \llbracket \sigma \rrbracket} \vee \overline{\amalg_2^\dagger \llbracket p \rrbracket \llbracket \sigma \rrbracket}
\end{aligned}$$

where $\overline{\amalg_1 [\![\sigma]\!]} = \overline{\amalg_2 [\![\sigma]\!]} = \mathrm{id}_I$ follows by (i), and $\overline{\amalg_1^\dagger [\![p]\!]} = \amalg_1^\dagger [\![p]\!]$ and $\overline{\amalg_2^\dagger [\![p]\!]} = \amalg_2^\dagger [\![p]\!]$ follow by $[\![p]\!]$ a decision. We may establish by analogous argument that

$$\overline{[\![q]\!]\,[\![\sigma]\!]} = \overline{\amalg_1^\dagger [\![q]\!]\,[\![\sigma]\!]} \vee \overline{\amalg_2^\dagger [\![q]\!]\,[\![\sigma]\!]}$$

as well. In the following, let $\sigma_p = [\![p]\!]\,[\![\sigma]\!]$ and $\sigma_q = [\![q]\!]\,[\![\sigma]\!]$. We have

$$\begin{aligned}
\overline{[\![p \textbf{ and } q]\!]\,[\![\sigma]\!]} &= \left(\left(\overline{\amalg_1^\dagger [\![p]\!]\,[\![\sigma]\!]}\ \overline{\amalg_1^\dagger [\![q]\!]\,[\![\sigma]\!]}\right) \vee \left(\overline{\amalg_2^\dagger [\![p]\!]\,[\![\sigma]\!]} \vee \overline{\amalg_2^\dagger [\![q]\!]\,[\![\sigma]\!]}\right)\right) \overline{[\![p]\!]\,[\![\sigma]\!]}\ \overline{[\![q]\!]\,[\![\sigma]\!]} \\
&= \left(\left(\overline{\amalg_1^\dagger \sigma_p}\ \overline{\amalg_1^\dagger \sigma_q}\right) \vee \left(\overline{\amalg_2^\dagger \sigma_p} \vee \overline{\amalg_2^\dagger \sigma_q}\right)\right) \overline{\sigma_p}\ \overline{\sigma_q} \\
&= \left(\left(\overline{\amalg_1^\dagger \sigma_p}\ \overline{\amalg_1^\dagger \sigma_q}\right) \vee \overline{\amalg_2^\dagger \sigma_p} \vee \overline{\amalg_2^\dagger \sigma_q}\right) \overline{\sigma_p}\ \overline{\sigma_q} \\
&= \left(\overline{\amalg_1^\dagger \sigma_p}\ \overline{\amalg_1^\dagger \sigma_q}\ \overline{\sigma_p}\ \overline{\sigma_q}\right) \vee \left(\overline{\amalg_2^\dagger \sigma_p}\ \overline{\sigma_p}\ \overline{\sigma_q}\right) \vee \left(\overline{\amalg_2^\dagger \sigma_q}\ \overline{\sigma_p}\ \overline{\sigma_q}\right) \\
&= \left(\overline{\amalg_1^\dagger \sigma_p}\ \overline{\sigma_p}\ \overline{\amalg_1^\dagger \sigma_q}\ \overline{\sigma_q}\right) \vee \left(\overline{\amalg_2^\dagger \sigma_p}\ \overline{\sigma_p}\ \overline{\sigma_q}\right) \vee \left(\overline{\amalg_2^\dagger \sigma_q}\ \overline{\sigma_q}\ \overline{\sigma_p}\right) \\
&= \left(\overline{\amalg_1^\dagger \sigma_p}\ \overline{\amalg_1^\dagger \sigma_q}\right) \vee \left(\overline{\amalg_2^\dagger \sigma_p}\ \overline{\sigma_q}\right) \vee \left(\overline{\amalg_2^\dagger \sigma_q}\ \overline{\sigma_p}\right) \\
&= \left(\overline{\amalg_1^\dagger \sigma_p}\ \overline{\amalg_1^\dagger \sigma_q}\right) \vee \left(\overline{\amalg_2^\dagger \sigma_p}\left(\overline{\amalg_1^\dagger \sigma_q} \vee \overline{\amalg_2^\dagger \sigma_q}\right)\right) \vee \left(\overline{\amalg_2^\dagger \sigma_q}\left(\overline{\amalg_1^\dagger \sigma_p} \vee \overline{\amalg_2^\dagger \sigma_p}\right)\right) \\
&= \left(\overline{\amalg_1^\dagger \sigma_p}\ \overline{\amalg_1^\dagger \sigma_q}\right) \vee \left(\overline{\amalg_2^\dagger \sigma_p}\ \overline{\amalg_1^\dagger \sigma_q}\right) \vee \left(\overline{\amalg_2^\dagger \sigma_p}\ \overline{\amalg_2^\dagger \sigma_q}\right) \vee \left(\overline{\amalg_2^\dagger \sigma_q}\ \overline{\amalg_1^\dagger \sigma_p}\right) \vee \left(\overline{\amalg_2^\dagger \sigma_q}\ \overline{\amalg_2^\dagger \sigma_p}\right) \\
&= \left(\overline{\amalg_1^\dagger \sigma_p}\ \overline{\amalg_1^\dagger \sigma_q}\right) \vee \left(\overline{\amalg_1^\dagger \sigma_p}\ \overline{\amalg_2^\dagger \sigma_q}\right) \vee \left(\overline{\amalg_2^\dagger \sigma_p}\ \overline{\amalg_1^\dagger \sigma_q}\right) \vee \left(\overline{\amalg_2^\dagger \sigma_p}\ \overline{\amalg_2^\dagger \sigma_q}\right) \\
&= \left(\overline{\amalg_1^\dagger \sigma_p} \vee \overline{\amalg_2^\dagger \sigma_p}\right)\left(\overline{\amalg_1^\dagger \sigma_q} \vee \overline{\amalg_2^\dagger \sigma_q}\right) = \overline{\sigma_p}\ \overline{\sigma_q}
\end{aligned}$$

which was what we wanted. □

A common way to show soundness (see, *e.g.*, [11]) is to show a kind of preservation property; that interpretations are, in a sense, preserved across evaluation in the operational semantics. This is shown in the following lemma:

Lemma 6.2. *If* $\sigma \vdash p \downarrow b$ *then* $[\![p]\!]\,[\![\sigma]\!] = [\![b]\!]\,[\![\sigma]\!]$.

Proof. By induction on the structure of the derivation $\mathcal{D}$ of $\sigma \vdash p \downarrow b$.

- Case $\mathcal{D} = \dfrac{}{\sigma \vdash tt \downarrow tt}$. We trivially have $[\![tt]\!]\,[\![\sigma]\!] = [\![tt]\!]\,[\![\sigma]\!]$.
- Case $\mathcal{D} = \dfrac{}{\sigma \vdash f\!f \downarrow f\!f}$. Again, we trivially have $[\![f\!f]\!]\,[\![\sigma]\!] = [\![f\!f]\!]\,[\![\sigma]\!]$.
- Case $\mathcal{D} = \dfrac{\sigma \vdash p \downarrow tt}{\sigma \vdash \textbf{not } p \downarrow f\!f}$.

  By induction we have that $[\![p]\!]\,[\![\sigma]\!] = [\![tt]\!]\,[\![\sigma]\!] = \amalg_1 [\![\sigma]\!]$. But then $[\![\textbf{not } p]\!]\,[\![\sigma]\!] = \gamma\, [\![p]\!]\,[\![\sigma]\!] = \gamma\, \amalg_1 [\![\sigma]\!] = \amalg_2 [\![\sigma]\!] = [\![f\!f]\!]\,[\![\sigma]\!]$.
- Case $\mathcal{D} = \dfrac{\sigma \vdash p \downarrow f\!f}{\sigma \vdash \textbf{not } p \downarrow tt}$.

  By induction we have that $[\![p]\!]\,[\![\sigma]\!] = [\![f\!f]\!]\,[\![\sigma]\!] = \amalg_2 [\![\sigma]\!]$. Thus $[\![\textbf{not } p]\!]\,[\![\sigma]\!] = \gamma\, [\![p]\!]\,[\![\sigma]\!] = \gamma\, \amalg_2 [\![\sigma]\!] = \amalg_1 [\![\sigma]\!] = [\![tt]\!]\,[\![\sigma]\!]$.
- Case $\mathcal{D} = \dfrac{\sigma \vdash p \downarrow tt \qquad \sigma \vdash q \downarrow tt}{\sigma \vdash p \textbf{ and } q \downarrow tt}$.

By induction $\llbracket p \rrbracket \llbracket \sigma \rrbracket = \llbracket tt \rrbracket \llbracket \sigma \rrbracket = \amalg_1 \llbracket \sigma \rrbracket$ and $\llbracket q \rrbracket \llbracket \sigma \rrbracket = \llbracket tt \rrbracket \llbracket \sigma \rrbracket = \amalg_1 \llbracket \sigma \rrbracket$. We compute

$$\begin{aligned}
\llbracket p \textbf{ and } q \rrbracket \llbracket \sigma \rrbracket &= \left(\amalg_1 \overline{\amalg_1^\dagger \llbracket p \rrbracket}\; \overline{\amalg_1^\dagger \llbracket q \rrbracket}\right) \vee \left(\amalg_2 \left(\overline{\amalg_2^\dagger \llbracket p \rrbracket} \vee \overline{\amalg_2^\dagger \llbracket q \rrbracket}\right)\right) \overline{\llbracket p \rrbracket}\; \overline{\llbracket q \rrbracket} \llbracket \sigma \rrbracket \\
&= \left(\amalg_1 \llbracket \sigma \rrbracket \overline{\amalg_1^\dagger \llbracket p \rrbracket \llbracket \sigma \rrbracket}\; \overline{\amalg_1^\dagger \llbracket q \rrbracket \llbracket \sigma \rrbracket}\right) \vee \left(\amalg_2 \llbracket \sigma \rrbracket \left(\overline{\amalg_2^\dagger \llbracket p \rrbracket \llbracket \sigma \rrbracket} \vee \overline{\amalg_2^\dagger \llbracket q \rrbracket \llbracket \sigma \rrbracket}\right)\right) \\
&\qquad \overline{\llbracket p \rrbracket \llbracket \sigma \rrbracket}\; \overline{\llbracket q \rrbracket \llbracket \sigma \rrbracket} \\
&= \left(\amalg_1 \llbracket \sigma \rrbracket \overline{\amalg_1^\dagger \amalg_1 \llbracket \sigma \rrbracket}\; \overline{\amalg_1^\dagger \amalg_1 \llbracket \sigma \rrbracket}\right) \vee \left(\amalg_2 \llbracket \sigma \rrbracket \left(\overline{\amalg_2^\dagger \amalg_1 \llbracket \sigma \rrbracket} \vee \overline{\amalg_2^\dagger \amalg_1 \llbracket \sigma \rrbracket}\right)\right) \\
&\qquad \overline{\amalg_1 \llbracket \sigma \rrbracket}\; \overline{\amalg_1 \llbracket \sigma \rrbracket} \\
&= \left(\amalg_1 \llbracket \sigma \rrbracket \overline{\llbracket \sigma \rrbracket}\; \overline{\llbracket \sigma \rrbracket}\right) \vee \left(\amalg_2 \llbracket \sigma \rrbracket \left(\overline{0_{\Sigma,\Sigma} \llbracket \sigma \rrbracket} \vee \overline{0_{\Sigma,\Sigma} \llbracket \sigma \rrbracket}\right)\right) \overline{\llbracket \sigma \rrbracket}\; \overline{\llbracket \sigma \rrbracket} \\
&= (\amalg_1 \llbracket \sigma \rrbracket) \vee (\amalg_2 \llbracket \sigma \rrbracket (0_{I,I} \vee 0_{I,I})) \overline{\llbracket \sigma \rrbracket} \\
&= (\amalg_1 \llbracket \sigma \rrbracket) \vee (\amalg_2 \llbracket \sigma \rrbracket 0_{I,I}) \overline{\llbracket \sigma \rrbracket} \\
&= ((\amalg_1 \llbracket \sigma \rrbracket) \vee 0_{I,\Sigma\oplus\Sigma}) \overline{\llbracket \sigma \rrbracket} \\
&= \amalg_1 \llbracket \sigma \rrbracket \overline{\llbracket \sigma \rrbracket} = \amalg_1 \llbracket \sigma \rrbracket = \llbracket tt \rrbracket \llbracket \sigma \rrbracket .
\end{aligned}$$

- Case $\mathcal{D} = \dfrac{\sigma \vdash p \downarrow ff \qquad \sigma \vdash q \downarrow tt}{\sigma \vdash p \textbf{ and } q \downarrow ff}$.

  By induction $\llbracket p \rrbracket \llbracket \sigma \rrbracket = \llbracket ff \rrbracket \llbracket \sigma \rrbracket = \amalg_2 \llbracket \sigma \rrbracket$ and $\llbracket q \rrbracket \llbracket \sigma \rrbracket = \llbracket tt \rrbracket \llbracket \sigma \rrbracket = \amalg_1 \llbracket \sigma \rrbracket$.

$$\begin{aligned}
\llbracket p \textbf{ and } q \rrbracket \llbracket \sigma \rrbracket &= \left(\amalg_1 \overline{\amalg_1^\dagger \llbracket p \rrbracket}\; \overline{\amalg_1^\dagger \llbracket q \rrbracket}\right) \vee \left(\amalg_2 \left(\overline{\amalg_2^\dagger \llbracket p \rrbracket} \vee \overline{\amalg_2^\dagger \llbracket q \rrbracket}\right)\right) \overline{\llbracket p \rrbracket}\; \overline{\llbracket q \rrbracket} \llbracket \sigma \rrbracket \\
&= \left(\amalg_1 \llbracket \sigma \rrbracket \overline{\amalg_1^\dagger \llbracket p \rrbracket \llbracket \sigma \rrbracket}\; \overline{\amalg_1^\dagger \llbracket q \rrbracket \llbracket \sigma \rrbracket}\right) \vee \left(\amalg_2 \llbracket \sigma \rrbracket \left(\overline{\amalg_2^\dagger \llbracket p \rrbracket \llbracket \sigma \rrbracket} \vee \overline{\amalg_2^\dagger \llbracket q \rrbracket \llbracket \sigma \rrbracket}\right)\right) \\
&\qquad \overline{\llbracket p \rrbracket \llbracket \sigma \rrbracket}\; \overline{\llbracket q \rrbracket \llbracket \sigma \rrbracket} \\
&= \left(\amalg_1 \llbracket \sigma \rrbracket \overline{\amalg_1^\dagger \amalg_2 \llbracket \sigma \rrbracket}\; \overline{\amalg_1^\dagger \amalg_1 \llbracket \sigma \rrbracket}\right) \vee \left(\amalg_2 \llbracket \sigma \rrbracket \left(\overline{\amalg_2^\dagger \amalg_2 \llbracket \sigma \rrbracket} \vee \overline{\amalg_2^\dagger \amalg_1 \llbracket \sigma \rrbracket}\right)\right) \\
&\qquad \overline{\amalg_2 \llbracket \sigma \rrbracket}\; \overline{\amalg_1 \llbracket \sigma \rrbracket} \\
&= \left(\amalg_1 \llbracket \sigma \rrbracket \overline{0_{\Sigma,\Sigma} \llbracket \sigma \rrbracket}\; \overline{\llbracket \sigma \rrbracket}\right) \vee \left(\amalg_2 \llbracket \sigma \rrbracket \left(\overline{\llbracket \sigma \rrbracket} \vee \overline{0_{\Sigma,\Sigma} \llbracket \sigma \rrbracket}\right)\right) \overline{\llbracket \sigma \rrbracket}\; \overline{\llbracket \sigma \rrbracket} \\
&= \left(\amalg_1 \llbracket \sigma \rrbracket 0_{I,I}\; \overline{\llbracket \sigma \rrbracket}\right) \vee \left(\amalg_2 \llbracket \sigma \rrbracket \left(\overline{\llbracket \sigma \rrbracket} \vee 0_{I,I}\right)\right) \overline{\llbracket \sigma \rrbracket} \\
&= \left(0_{I,\Sigma\oplus\Sigma} \vee \left(\amalg_2 \llbracket \sigma \rrbracket \overline{\llbracket \sigma \rrbracket}\right)\right) \overline{\llbracket \sigma \rrbracket} \\
&= \amalg_2 \llbracket \sigma \rrbracket \overline{\llbracket \sigma \rrbracket}\; \overline{\llbracket \sigma \rrbracket} = \amalg_2 \llbracket \sigma \rrbracket = \llbracket ff \rrbracket \llbracket \sigma \rrbracket
\end{aligned}$$

- Case $\mathcal{D} = \dfrac{\sigma \vdash p \downarrow tt \qquad \sigma \vdash q \downarrow ff}{\sigma \vdash p \textbf{ and } q \downarrow ff}$, similar to the previous case.
- Case $\mathcal{D} = \dfrac{\sigma \vdash p \downarrow ff \qquad \sigma \vdash q \downarrow ff}{\sigma \vdash p \textbf{ and } q \downarrow ff}$.

By induction $[\![p]\!]\,[\![\sigma]\!] = [\![\mathit{ff}]\!]\,[\![\sigma]\!] = \amalg_2\,[\![\sigma]\!]$ and $[\![q]\!]\,[\![\sigma]\!] = [\![\mathit{ff}]\!]\,[\![\sigma]\!] = \amalg_2\,[\![\sigma]\!]$.

$$
\begin{aligned}
[\![p \textbf{ and } q]\!]\,[\![\sigma]\!] &= \left(\amalg_1 \overline{\amalg_1^\dagger\,[\![p]\!]}\;\overline{\amalg_1^\dagger\,[\![q]\!]}\right) \vee \left(\amalg_2 \left(\overline{\amalg_2^\dagger\,[\![p]\!]} \vee \overline{\amalg_2^\dagger\,[\![q]\!]}\right)\right) \overline{[\![p]\!]}\;\overline{[\![q]\!]}\,[\![\sigma]\!] \\
&= \left(\amalg_1\,[\![\sigma]\!]\,\overline{\amalg_1^\dagger\,[\![p]\!]\,[\![\sigma]\!]}\;\overline{\amalg_1^\dagger\,[\![q]\!]\,[\![\sigma]\!]}\right) \vee \left(\amalg_2\,[\![\sigma]\!] \left(\overline{\amalg_2^\dagger\,[\![p]\!]\,[\![\sigma]\!]} \vee \overline{\amalg_2^\dagger\,[\![q]\!]\,[\![\sigma]\!]}\right)\right) \\
&\qquad \overline{[\![p]\!]\,[\![\sigma]\!]}\;\overline{[\![q]\!]\,[\![\sigma]\!]} \\
&= \left(\amalg_1\,[\![\sigma]\!]\,\overline{\amalg_1^\dagger\,\amalg_2\,[\![\sigma]\!]}\;\overline{\amalg_1^\dagger\,\amalg_2\,[\![\sigma]\!]}\right) \vee \left(\amalg_2\,[\![\sigma]\!] \left(\overline{\amalg_2^\dagger\,\amalg_2\,[\![\sigma]\!]} \vee \overline{\amalg_2^\dagger\,\amalg_2\,[\![\sigma]\!]}\right)\right) \\
&\qquad \overline{\amalg_2\,[\![\sigma]\!]}\;\overline{\amalg_2\,[\![\sigma]\!]} \\
&= \left(\amalg_1\,[\![\sigma]\!]\,\overline{0_{\Sigma,\Sigma}\,[\![\sigma]\!]}\;\overline{0_{\Sigma,\Sigma}\,[\![\sigma]\!]}\right) \vee \left(\amalg_2\,[\![\sigma]\!] \left(\overline{[\![\sigma]\!]} \vee \overline{[\![\sigma]\!]}\right)\right) \overline{[\![\sigma]\!]}\;\overline{[\![\sigma]\!]} \\
&= (\amalg_1\,[\![\sigma]\!]\,0_{I,I}) \vee \left(\amalg_2\,[\![\sigma]\!]\,\overline{[\![\sigma]\!]}\right) \overline{[\![\sigma]\!]} \\
&= \left(0_{I,\Sigma\oplus\Sigma} \vee \left(\amalg_2\,[\![\sigma]\!]\,\overline{[\![\sigma]\!]}\right)\right) \overline{[\![\sigma]\!]} \\
&= \amalg_2\,[\![\sigma]\!]\,\overline{[\![\sigma]\!]}\;\overline{[\![\sigma]\!]} = \amalg_2\,[\![\sigma]\!] = [\![\mathit{ff}]\!]\,[\![\sigma]\!]\,.
\end{aligned}
$$

□

With this done, the soundness lemma for predicates follows readily.

Lemma 6.3. *If there exists b such that $\sigma \vdash p \downarrow b$ then $[\![p]\!]\,[\![\sigma]\!]$ is total.*

Proof. Suppose there exists b such that $\sigma \vdash p \downarrow b$ by some derivation. It follows by the operational semantics that b must be either tt or ff, and in either case it follows by Lemma 6.1 (i) that $[\![b]\!]\,[\![\sigma]\!]$ is total, *i.e.*, $\overline{[\![b]\!]\,[\![\sigma]\!]} = \mathrm{id}_I$. Applying the derivation of $\sigma \vdash p \downarrow b$ to Lemma 6.2 yields that $[\![p]\!]\,[\![\sigma]\!] = [\![b]\!]\,[\![\sigma]\!]$, so specifically $\overline{[\![p]\!]\,[\![\sigma]\!]} = \overline{[\![b]\!]\,[\![\sigma]\!]} = \mathrm{id}_I$, as desired. □

Adequacy for predicates can then be shown by induction on the structure of the predicate, and by letting Lemma 6.1 (regarding the totality of predicates) do much of the heavy lifting.

Lemma 6.4. *If $[\![p]\!]\,[\![\sigma]\!]$ is total then there exists b such that $\sigma \vdash p \downarrow b$.*

Proof. By induction on the structure of p.

- Case $p = \mathit{tt}$. Then $\sigma \vdash \mathit{tt} \downarrow \mathit{tt}$ by $\dfrac{}{\sigma \vdash \mathit{tt} \downarrow \mathit{tt}}$.
- Case $p = \mathit{ff}$. Then $\sigma \vdash \mathit{ff} \downarrow \mathit{ff}$ by $\dfrac{}{\sigma \vdash \mathit{ff} \downarrow \mathit{ff}}$.
- Case $p = \textbf{not } p'$. Since $[\![\textbf{not } p']\!]\,[\![\sigma]\!]$ is total, it follows by Lemma 6.1 that $[\![p']\!]\,[\![\sigma]\!]$ is total as well, so by induction, there exists b such that $\sigma \vdash p' \downarrow b$ by some derivation $\mathcal{D}$. We have two cases to consider: If $b = \mathit{tt}$, $\mathcal{D}$ is a derivation of $\sigma \vdash p' \downarrow \mathit{tt}$, and so we may derive $\sigma \vdash \textbf{not } p' \downarrow \mathit{ff}$ by
$$\dfrac{\overset{\mathcal{D}}{\sigma \vdash p' \downarrow \mathit{tt}}}{\sigma \vdash \textbf{not } p' \downarrow \mathit{ff}}$$
If on the other hand $b = \mathit{ff}$, $\mathcal{D}$ is a derivation of $\sigma \vdash p' \downarrow \mathit{ff}$, and we may use the other **not**-rule with $\mathcal{D}$ to derive
$$\dfrac{\overset{\mathcal{D}}{\sigma \vdash p' \downarrow \mathit{ff}}}{\sigma \vdash \textbf{not } p' \downarrow \mathit{tt}}\ .$$

- Case $p = q$ **and** r. Since we have that $[\![q \textbf{ and } r]\!]\,[\![\sigma]\!]$ is total, by Lemma 6.1, so are $[\![q]\!]\,[\![\sigma]\!]$ and $[\![r]\!]\,[\![\sigma]\!]$. Thus, it follows by induction that there exist b_1 and b_2 such that $\sigma \vdash q \downarrow b_1$ respectively $\sigma \vdash r \downarrow b_2$ by derivations $\mathcal{D}_1$ respectively $\mathcal{D}_2$. This gives us four cases depending on what b_1 and b_2 are. Luckily, these four cases match precisely the four different rules we have for **and**: For example, if $b_1 = tt$ and $b_2 = ff$, we may derive $\sigma \vdash q$ **and** $r \downarrow ff$ by

$$\frac{\overset{\mathcal{D}_1}{\sigma \vdash q \downarrow tt} \qquad \overset{\mathcal{D}_2}{\sigma \vdash r \downarrow ff}}{\sigma \vdash q \textbf{ and } r \downarrow ff},$$

and so on.

□

With soundness and adequacy done for the predicates, we turn our attention to commands. Before we can show soundness, we will need a technical lemma regarding the denotational behaviour of loop bodies in states σ when the relevant predicates are either true or false (see Definition 12 for the definition of the loop body $\beta[p, c, q]$).

Lemma 6.5. *Let σ be a state, and p and q be predicates. Then*
(1) If $\sigma \vdash p \downarrow tt$ and $\sigma \vdash q \downarrow tt$ then $\beta[p, c, q]_{11}\,[\![\sigma]\!] = [\![\sigma]\!]$,
(2) If $\sigma \vdash p \downarrow ff$ and $\sigma \vdash q \downarrow tt$ then $\beta[p, c, q]_{21}\,[\![\sigma]\!] = [\![\sigma]\!]$,
(3) If $\sigma \vdash p \downarrow tt$ and $\sigma \vdash q \downarrow ff$ then $\beta[p, c, q]_{12}\,[\![\sigma]\!] = [\![c]\!]\,[\![\sigma]\!]$, and
(4) If $\sigma \vdash p \downarrow ff$ and $\sigma \vdash q \downarrow ff$ then $\beta[p, c, q]_{22}\,[\![\sigma]\!] = [\![c]\!]\,[\![\sigma]\!]$.
Further, in each case, for all other choices of i and j, $\beta[p, c, q]_{ij}\,[\![\sigma]\!] = 0_{I,\Sigma}$.

Proof. For (1), suppose $\sigma \vdash p \downarrow tt$ and $\sigma \vdash q \downarrow tt$, so by Lemma 6.2, $[\![p]\!]\,[\![\sigma]\!] = [\![tt]\!]\,[\![\sigma]\!] = \amalg_1\,[\![\sigma]\!]$ and $[\![q]\!]\,[\![\sigma]\!] = [\![tt]\!]\,[\![\sigma]\!] = \amalg_1\,[\![\sigma]\!]$. We have

$$\begin{aligned}
\beta[p, c, q]_{11}\,[\![\sigma]\!] &= \amalg_1^\dagger(\mathrm{id}_\Sigma \oplus [\![c]\!])\,[\![q]\!]\,[\![p]\!]^\dagger\,\amalg_1\,[\![\sigma]\!] = \amalg_1^\dagger(\mathrm{id}_\Sigma \oplus [\![c]\!])\,[\![q]\!]\,[\![p]\!]^\dagger\,[\![p]\!]\,[\![\sigma]\!] \\
&= \amalg_1^\dagger(\mathrm{id}_\Sigma \oplus [\![c]\!])\,[\![q]\!]\,\overline{[\![p]\!]}\,[\![\sigma]\!] = \amalg_1^\dagger(\mathrm{id}_\Sigma \oplus [\![c]\!])\,[\![q]\!]\,[\![\sigma]\!]\,\overline{[\![p]\!]\,[\![\sigma]\!]} \\
&= \amalg_1^\dagger(\mathrm{id}_\Sigma \oplus [\![c]\!])\,\amalg_1\,[\![\sigma]\!]\,\overline{\amalg_1\,[\![\sigma]\!]} = \amalg_1^\dagger(\mathrm{id}_\Sigma \oplus [\![c]\!])\,\amalg_1\,[\![\sigma]\!] \\
&= \amalg_1^\dagger\,\amalg_1\,\mathrm{id}_\Sigma\,[\![\sigma]\!] = \overline{\amalg_1}\,[\![\sigma]\!] = \mathrm{id}_\Sigma\,[\![\sigma]\!] = [\![\sigma]\!]\,.
\end{aligned}$$

The proof of (2) is analogous to that of (1).

For (3), suppose $\sigma \vdash p \downarrow tt$ and $\sigma \vdash q \downarrow ff$, so by Lemma 6.2, $[\![p]\!]\,[\![\sigma]\!] = [\![tt]\!]\,[\![\sigma]\!] = \amalg_1\,[\![\sigma]\!]$ and $[\![q]\!]\,[\![\sigma]\!] = [\![ff]\!]\,[\![\sigma]\!] = \amalg_2\,[\![\sigma]\!]$. We compute

$$\begin{aligned}
\beta[p, c, q]_{12}\,[\![\sigma]\!] &= \amalg_2^\dagger(\mathrm{id}_\Sigma \oplus [\![c]\!])\,[\![q]\!]\,[\![p]\!]^\dagger\,\amalg_1\,[\![\sigma]\!] = \amalg_2^\dagger(\mathrm{id}_\Sigma \oplus [\![c]\!])\,[\![q]\!]\,[\![p]\!]^\dagger\,[\![p]\!]\,[\![\sigma]\!] \\
&= \amalg_2^\dagger(\mathrm{id}_\Sigma \oplus [\![c]\!])\,[\![q]\!]\,\overline{[\![p]\!]}\,[\![\sigma]\!] = \amalg_2^\dagger(\mathrm{id}_\Sigma \oplus [\![c]\!])\,[\![q]\!]\,[\![\sigma]\!]\,\overline{[\![p]\!]\,[\![\sigma]\!]} \\
&= \amalg_2^\dagger(\mathrm{id}_\Sigma \oplus [\![c]\!])\,\amalg_2\,[\![\sigma]\!]\,\overline{\amalg_1\,[\![\sigma]\!]} = \amalg_2^\dagger\,\amalg_2\,[\![c]\!]\,[\![\sigma]\!]\,\overline{\overline{\amalg_1}\,[\![\sigma]\!]} \\
&= \overline{\amalg_2}\,[\![c]\!]\,[\![\sigma]\!]\,\overline{\mathrm{id}_\Sigma\,[\![\sigma]\!]} = \mathrm{id}_\Sigma\,[\![c]\!]\,[\![\sigma]\!]\,\overline{[\![\sigma]\!]} = [\![c]\!]\,[\![\sigma]\!]
\end{aligned}$$

The proof of (4) is analogous.

To see that in each case, for all other choices of i, j, $\beta[p, c, q]_{ij}\,[\![\sigma]\!] = 0_{I,\Sigma}$, we show a few of the cases where $\sigma \vdash p \downarrow tt$ and $\sigma \vdash q \downarrow tt$. The rest follow by the same line of reasoning. Recall that when $\sigma \vdash p \downarrow tt$ and $\sigma \vdash q \downarrow tt$ we have $[\![p]\!]\,[\![\sigma]\!] = [\![tt]\!]\,[\![\sigma]\!] = \amalg_1\,[\![\sigma]\!]$

and $[\![q]\!]\,[\![\sigma]\!] = [\![tt]\!]\,[\![\sigma]\!] = \amalg_1\,[\![\sigma]\!]$.

$$\begin{aligned}
\beta[p,c,q]_{12}\,[\![\sigma]\!] &= \amalg_2^\dagger(\mathrm{id}_\Sigma \oplus [\![c]\!])\,[\![q]\!]\,[\![p]\!]^\dagger\,\amalg_1\,[\![\sigma]\!] = \amalg_2^\dagger(\mathrm{id}_\Sigma \oplus [\![c]\!])\,[\![q]\!]\,[\![p]\!]^\dagger\,[\![p]\!]\,[\![\sigma]\!] \\
&= \amalg_2^\dagger(\mathrm{id}_\Sigma \oplus [\![c]\!])\,[\![q]\!]\,\overline{[\![p]\!]}\,[\![\sigma]\!] = \amalg_2^\dagger(\mathrm{id}_\Sigma \oplus [\![c]\!])\,[\![q]\!]\,[\![\sigma]\!]\,\overline{[\![p]\!]\,[\![\sigma]\!]} \\
&= \amalg_2^\dagger(\mathrm{id}_\Sigma \oplus [\![c]\!])\,\amalg_1\,[\![\sigma]\!]\,\overline{[\![p]\!]\,[\![\sigma]\!]} = \amalg_2^\dagger\,\amalg_1\,\mathrm{id}_\Sigma\,[\![\sigma]\!]\,\overline{[\![p]\!]\,[\![\sigma]\!]} \\
&= 0_{\Sigma,\Sigma}\mathrm{id}_\Sigma\,[\![\sigma]\!]\,\overline{[\![p]\!]\,[\![\sigma]\!]} = 0_{I,\Sigma}.
\end{aligned}$$

For $\beta[p,c,q]_{21}$, we have

$$\begin{aligned}
\beta[p,c,q]_{21}\,[\![\sigma]\!] &= \amalg_1^\dagger(\mathrm{id}_\Sigma \oplus [\![c]\!])\,[\![q]\!]\,[\![p]\!]^\dagger\,\amalg_2\,[\![\sigma]\!] = \amalg_1^\dagger(\mathrm{id}_\Sigma \oplus [\![c]\!])\,[\![q]\!]\,[\![p]\!]^\dagger\,\gamma\,\amalg_1\,[\![\sigma]\!] \\
&= \amalg_1^\dagger(\mathrm{id}_\Sigma \oplus [\![c]\!])\,[\![q]\!]\,[\![p]\!]^\dagger\,\gamma\,[\![p]\!]\,[\![\sigma]\!] = \amalg_1^\dagger(\mathrm{id}_\Sigma \oplus [\![c]\!])\,[\![q]\!]\,[\![p]\!]^\dagger\,[\![\mathbf{not}\ p]\!]\,[\![\sigma]\!] \\
&= \amalg_1^\dagger(\mathrm{id}_\Sigma \oplus [\![c]\!])\,[\![q]\!]\,0_{\Sigma,\Sigma}\,[\![\sigma]\!] = 0_{I,\Sigma},
\end{aligned}$$

and so on. □

With this lemma done, we turn our attention to the preservation lemma for commands in order to show soundness.

Lemma 6.6. *If $\sigma \vdash c \downarrow \sigma'$ then $[\![c]\!]\,[\![\sigma]\!] = [\![\sigma']\!]$.*

Proof. By induction on the structure of the derivation $\mathcal{D}$ of $\sigma \vdash c \downarrow \sigma'$.

- Case $\mathcal{D} = \dfrac{}{\sigma \vdash \mathbf{skip} \downarrow \sigma}$. We have $[\![\mathbf{skip}]\!]\,[\![\sigma]\!] = \mathrm{id}_\Sigma\,[\![\sigma]\!] = [\![\sigma]\!]$.
- Case $\mathcal{D} = \dfrac{\sigma \vdash c_1 \downarrow \sigma' \qquad \sigma' \vdash c_2 \downarrow \sigma''}{\sigma \vdash c_1\ ;\ c_2 \downarrow \sigma''}$.

  By induction, $[\![c_1]\!]\,[\![\sigma]\!] = [\![\sigma']\!]$ and $[\![c_2]\!]\,[\![\sigma']\!] = [\![\sigma'']\!]$. But then
  $$[\![c_1\ ;\ c_2]\!]\,[\![\sigma]\!] = [\![c_2]\!]\,[\![c_1]\!]\,[\![\sigma]\!] = [\![c_2]\!]\,[\![\sigma']\!] = [\![\sigma'']\!]$$
  as desired.
- Case $\mathcal{D} = \dfrac{\sigma \vdash p \downarrow tt \qquad \sigma \vdash c_1 \downarrow \sigma' \qquad \sigma' \vdash q \downarrow tt}{\sigma \vdash \mathbf{if}\ p\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi}\ q \downarrow \sigma'}$.

  By induction, $[\![c_1]\!]\,[\![\sigma]\!] = [\![\sigma']\!]$, and by Lemma 6.2, $[\![p]\!]\,[\![\sigma]\!] = [\![tt]\!]\,[\![\sigma]\!] = \amalg_1\,[\![\sigma]\!]$ and $[\![q]\!]\,[\![\sigma']\!] = [\![tt]\!]\,[\![\sigma']\!] = \amalg_1\,[\![\sigma']\!]$. We compute:

$$\begin{aligned}
[\![\mathbf{if}\ p\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi}\ q]\!]\,[\![\sigma]\!] &= [\![q]\!]^\dagger\,([\![c_1]\!] \oplus [\![c_2]\!])\,[\![p]\!]\,[\![\sigma]\!] = [\![q]\!]^\dagger\,([\![c_1]\!] \oplus [\![c_2]\!])\,\amalg_1\,[\![\sigma]\!] \\
&= [\![q]\!]^\dagger\,\amalg_1\,[\![c_1]\!]\,[\![\sigma]\!] = [\![q]\!]^\dagger\,\amalg_1\,[\![\sigma']\!] = [\![q]\!]^\dagger\,[\![q]\!]\,[\![\sigma']\!] \\
&= \overline{[\![q]\!]}\,[\![\sigma']\!] = [\![\sigma']\!]\,\overline{[\![q]\!]\,[\![\sigma']\!]} = [\![\sigma']\!]\,\overline{\amalg_1\,[\![\sigma']\!]} \\
&= [\![\sigma']\!]\,\overline{\overline{\amalg_1}\,[\![\sigma']\!]} = [\![\sigma']\!]\,\overline{\mathrm{id}_\Sigma\,[\![\sigma']\!]} = [\![\sigma']\!]\,\overline{[\![\sigma']\!]} = [\![\sigma']\!]\,.
\end{aligned}$$

- Case $\mathcal{D} = \dfrac{\sigma \vdash p \downarrow f\!f \qquad \sigma \vdash c_2 \downarrow \sigma' \qquad \sigma' \vdash q \downarrow f\!f}{\sigma \vdash \mathbf{if}\ p\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi}\ q \downarrow \sigma'}$.

  By induction, $[\![c_2]\!]\,[\![\sigma]\!] = [\![\sigma']\!]$, and by Lemma 6.2, $[\![p]\!]\,[\![\sigma]\!] = [\![f\!f]\!]\,[\![\sigma]\!] = \amalg_2\,[\![\sigma]\!]$ and

$[\![q]\!]\,[\![\sigma']\!] = [\![ff]\!]\,[\![\sigma']\!] = \amalg_2\,[\![\sigma']\!]$. We have

$$\begin{aligned}
[\![\textbf{if } p \textbf{ then } c_1 \textbf{ else } c_2 \textbf{ fi } q]\!]\,[\![\sigma]\!] &= [\![q]\!]^\dagger\,([\![c_1]\!] \oplus [\![c_2]\!])\,[\![p]\!]\,[\![\sigma]\!] = [\![q]\!]^\dagger\,([\![c_1]\!] \oplus [\![c_2]\!])\,\amalg_2\,[\![\sigma]\!] \\
&= [\![q]\!]^\dagger\,\amalg_2\,[\![c_2]\!]\,[\![\sigma]\!] = [\![q]\!]^\dagger\,\amalg_2\,[\![\sigma']\!] = [\![q]\!]^\dagger\,[\![q]\!]\,[\![\sigma']\!] \\
&= \overline{[\![q]\!]}\,[\![\sigma']\!] = [\![\sigma']\!]\,\overline{[\![q]\!]\,[\![\sigma']\!]} = [\![\sigma']\!]\,\overline{\amalg_2\,[\![\sigma']\!]} \\
&= [\![\sigma']\!]\,\overline{\overline{\amalg_2}\,[\![\sigma']\!]} = [\![\sigma']\!]\,\overline{\mathrm{id}_\Sigma\,[\![\sigma']\!]} = [\![\sigma']\!]\,\overline{[\![\sigma']\!]} = [\![\sigma']\!]\,.
\end{aligned}$$

- Case $\mathcal{D} = \dfrac{\sigma \vdash p \downarrow tt \qquad \sigma \vdash q \downarrow tt}{\sigma \vdash \textbf{from } p \textbf{ loop } c \textbf{ until } q \downarrow \sigma}$.

  Since $\sigma \vdash p \downarrow tt$ and $\sigma \vdash q \downarrow tt$, by Lemma 6.5 we get $\beta[p,c,q]_{11}\,[\![\sigma]\!] = [\![\sigma]\!]$ and $\beta[p,c,q]_{12}\,[\![\sigma]\!] = 0_{I,\Sigma}$, and so

$$\begin{aligned}
&[\![\textbf{from } p \textbf{ loop } c \textbf{ until } q]\!]\,[\![\sigma]\!] = \mathrm{Tr}^{\Sigma}_{\Sigma,\Sigma}(\beta[p,c,q]) \\
&\quad = \left(\beta[p,c,q]_{11} \vee \bigvee_{n\in\omega} \beta[p,c,q]_{21}\ \beta[p,c,q]_{22}^n\ \beta[p,c,q]_{12}\right)[\![\sigma]\!] \\
&\quad = \left((\beta[p,c,q]_{11}\,[\![\sigma]\!]) \vee \bigvee_{n\in\omega} \beta[p,c,q]_{21}\ \beta[p,c,q]_{22}^n\ \beta[p,c,q]_{12}\,[\![\sigma]\!]\right) \\
&\quad = [\![\sigma]\!] \vee \bigvee_{n\in\omega} \beta[p,c,q]_{21}\ \beta[p,c,q]_{22}^n\ 0_{I,\Sigma} = [\![\sigma]\!] \vee 0_{I,\Sigma} = [\![\sigma]\!]\,.
\end{aligned}$$

- Case $\mathcal{D} = \dfrac{\sigma \vdash p \downarrow tt \qquad \sigma \vdash q \downarrow ff \qquad \sigma \vdash c \downarrow \sigma' \qquad \sigma' \vdash \underline{loop}[p,c,q] \downarrow \sigma''}{\sigma \vdash \textbf{from } p \textbf{ loop } c \textbf{ until } q \downarrow \sigma''}$.

  By induction, $[\![c]\!]\,[\![\sigma]\!] = [\![\sigma']\!]$ and $[\![\underline{loop}[p,c,q]]\!]\,[\![\sigma']\!] = [\![\sigma'']\!]$, and since $\sigma \vdash p \downarrow tt$ and $\sigma \vdash q \downarrow ff$, by Lemma 6.5 we get $\beta[p,c,q]_{11}\,[\![\sigma]\!] = 0_{I,\Sigma}$ and $\beta[p,c,q]_{12}\,[\![\sigma]\!] = [\![c]\!]\,[\![\sigma]\!]$. Thus

$$\begin{aligned}
&[\![\textbf{from } p \textbf{ loop } c \textbf{ until } q]\!]\,[\![\sigma]\!] = \mathrm{Tr}^{\Sigma}_{\Sigma,\Sigma}(\beta[p,c,q]) \\
&\quad = \left(\beta[p,c,q]_{11} \vee \bigvee_{n\in\omega} \beta[p,c,q]_{21}\ \beta[p,c,q]_{22}^n\ \beta[p,c,q]_{12}\right)[\![\sigma]\!] \\
&\quad = \left((\beta[p,c,q]_{11}\,[\![\sigma]\!]) \vee \bigvee_{n\in\omega} \beta[p,c,q]_{21}\ \beta[p,c,q]_{22}^n\ \beta[p,c,q]_{12}\,[\![\sigma]\!]\right) \\
&\quad = \left(0_{I,\Sigma} \vee \bigvee_{n\in\omega} \beta[p,c,q]_{21}\ \beta[p,c,q]_{22}^n\ \ [\![c]\!]\,[\![\sigma]\!]\right) \\
&\quad = \bigvee_{n\in\omega} \beta[p,c,q]_{21}\ \beta[p,c,q]_{22}^n\ \ [\![\sigma']\!] \\
&\quad = \left(\bigvee_{n\in\omega} \beta[p,c,q]_{21}\ \beta[p,c,q]_{22}^n\right)[\![\sigma']\!] \\
&\quad = [\![\underline{loop}[p,c,q]]\!]\,[\![\sigma']\!] = [\![\sigma'']\!]\,.
\end{aligned}$$

- Case $\mathcal{D} = \dfrac{\sigma \vdash p \downarrow ff \qquad \sigma \vdash q \downarrow tt}{\sigma \vdash \underline{loop}[p,c,q] \downarrow \sigma}$.

Since $\sigma \vdash p \downarrow \mathit{ff}$ and $\sigma \vdash q \downarrow tt$, by Lemma 6.5 we get $\beta[p,c,q]_{22} \llbracket\sigma\rrbracket = 0_{I,\Sigma}$ and $\beta[p,c,q]_{21} \llbracket\sigma\rrbracket = \llbracket\sigma\rrbracket$. This gives us

$$\begin{aligned}
\llbracket \underline{loop}[p,c,q] \rrbracket \llbracket\sigma\rrbracket &= \left(\bigvee_{n\in\omega} \beta[p,c,q]_{21}\beta[p,c,q]_{22}^{n} \right) \llbracket\sigma\rrbracket \\
&= \left(\beta[p,c,q]_{21} \vee \bigvee_{n\in\omega} \beta[p,c,q]_{21}\beta[p,c,q]_{22}^{n+1} \right) \llbracket\sigma\rrbracket \\
&= \left((\beta[p,c,q]_{21} \llbracket\sigma\rrbracket) \vee \bigvee_{n\in\omega} \beta[p,c,q]_{21}\beta[p,c,q]_{22}^{n+1} \llbracket\sigma\rrbracket \right) \\
&= \left(\llbracket\sigma\rrbracket \vee \bigvee_{n\in\omega} \beta[p,c,q]_{21}\beta[p,c,q]_{22}^{n}\beta[p,c,q]_{22} \llbracket\sigma\rrbracket \right) \\
&= \left(\llbracket\sigma\rrbracket \vee \bigvee_{n\in\omega} \beta[p,c,q]_{21}\beta[p,c,q]_{22}^{n} 0_{I,\Sigma} \right) \\
&= \llbracket\sigma\rrbracket \vee 0_{I,\Sigma} = \llbracket\sigma\rrbracket
\end{aligned}$$

- Case $\mathcal{D} = \dfrac{\sigma \vdash p \downarrow \mathit{ff} \qquad \sigma \vdash q \downarrow \mathit{ff} \qquad \sigma \vdash c \downarrow \sigma' \qquad \sigma' \vdash \underline{loop}[p,c,q] \downarrow \sigma''}{\sigma \vdash \underline{loop}[p,c,q] \downarrow \sigma''}$.

By induction, $\llbracket c\rrbracket \llbracket\sigma\rrbracket = \llbracket\sigma'\rrbracket$ and $\llbracket \underline{loop}[p,c,q] \rrbracket \llbracket\sigma'\rrbracket = \llbracket\sigma''\rrbracket$, and since $\sigma \vdash p \downarrow \mathit{ff}$ and $\sigma \vdash q \downarrow \mathit{ff}$, it follows by Lemma 6.5 that $\beta[p,c,q]_{22} \llbracket\sigma\rrbracket = \llbracket c\rrbracket \llbracket\sigma\rrbracket$ and $\beta[p,c,q]_{21} \llbracket\sigma\rrbracket = 0_{I,\Sigma}$. We then have

$$\begin{aligned}
\llbracket \underline{loop}[p,c,q] \rrbracket \llbracket\sigma\rrbracket &= \left(\bigvee_{n\in\omega} \beta[p,c,q]_{21}\beta[p,c,q]_{22}^{n} \right) \llbracket\sigma\rrbracket \\
&= \left(\beta[p,c,q]_{21} \vee \bigvee_{n\in\omega} \beta[p,c,q]_{21}\beta[p,c,q]_{22}^{n+1} \right) \llbracket\sigma\rrbracket \\
&= \left((\beta[p,c,q]_{21} \llbracket\sigma\rrbracket) \vee \bigvee_{n\in\omega} \beta[p,c,q]_{21}\beta[p,c,q]_{22}^{n+1} \llbracket\sigma\rrbracket \right) \\
&= 0_{I,\Sigma} \vee \bigvee_{n\in\omega} \beta[p,c,q]_{21}\beta[p,c,q]_{22}^{n+1} \llbracket\sigma\rrbracket \\
&= \bigvee_{n\in\omega} \beta[p,c,q]_{21}\beta[p,c,q]_{22}^{n}\beta[p,c,q]_{22} \llbracket\sigma\rrbracket \\
&= \bigvee_{n\in\omega} \beta[p,c,q]_{21}\beta[p,c,q]_{22}^{n} \llbracket c\rrbracket \llbracket\sigma\rrbracket \\
&= \bigvee_{n\in\omega} \beta[p,c,q]_{21}\beta[p,c,q]_{22}^{n} \llbracket\sigma'\rrbracket \\
&= \left(\bigvee_{n\in\omega} \beta[p,c,q]_{21}\beta[p,c,q]_{22}^{n} \right) \llbracket\sigma'\rrbracket \\
&= \llbracket \underline{loop}[p,c,q] \rrbracket \llbracket\sigma'\rrbracket = \llbracket\sigma''\rrbracket
\end{aligned}$$

which was what we wanted.

□

This finally allows us to show the soundness theorem for commands – and so, for programs – in a straightforward manner.

Theorem 6.7 (Soundness). *If there exists σ' such that $\sigma \vdash c \downarrow \sigma'$ then $[\![c]\!]\,[\![\sigma]\!]$ is total.*

Proof. Suppose there exists σ' such that $\sigma \vdash c \downarrow \sigma'$. By Lemma 6.6, $[\![c]\!]\,[\![\sigma]\!] = [\![\sigma']\!]$, and since the interpretation of *any* state is assumed to be total, it follows that $\overline{[\![c]\!]\,[\![\sigma]\!]} = \overline{[\![\sigma']\!]} = \mathrm{id}_I$, which was what we wanted. □

With soundness done, we only have adequacy left to prove. Adequacy is much simpler than usual in our case, as we have no higher order data to deal with, and as such, it can be shown by plain structural induction rather than by the assistance of logical relations. Nevertheless, we require two technical lemmas in order to succeed.

Lemma 6.8. *If gf is total, so is f.*

Proof. Let $f : A \to B$ and $g : B \to C$ be morphisms such that $\overline{gf} = \mathrm{id}_A$. Then $\overline{f} = \mathrm{id}_A \overline{f} = \overline{gf}\;\overline{f} = \overline{gf\overline{f}} = \overline{gf} = \mathrm{id}_A$, so f is total as well. □

Lemma 6.9. *Let $f : A \oplus U \to B \oplus U$ and $s : I \to A$ be morphisms. If $\mathrm{Tr}^U_{A,B}(f)s$ is total, either $\mathrm{Tr}^U_{A,B}(f)s = f_{11}s$, or there exists $n \in \omega$ such that $\mathrm{Tr}^U_{A,B}(f)s = f_{21}f_{22}^n f_{12}s$.*

Proof. Since the trace is canonically constructed, it takes the form

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

Further, in the proof of Theorem 20 in [21], it is shown that this join not only exists but is a *disjoint* join, *i.e.*, for any choice of $n \in \omega$,

$$f_{11}\overline{f_{21}f_{22}^n f_{12}} = (f_{21}f_{22}^n f_{12})\,\overline{f_{11}} = 0_{A,B}$$

and for all $n, m \in \omega$ with $n \neq m$,

$$\overline{f_{21}f_{22}^n f_{12}}f_{21}f_{22}^m f_{12} = \overline{f_{21}f_{22}^m f_{12}}f_{21}f_{22}^n f_{12} = 0_{A,B}.$$

But then,

$$\begin{aligned}\overline{\mathrm{Tr}^U_{A,B}(f)s} &= \overline{\left(f_{11} \vee \bigvee_{n\in\omega} f_{21}f_{22}^n f_{12}\right)s} \\ &= \overline{(f_{11}s) \vee \bigvee_{n\in\omega} (f_{21}f_{22}^n f_{12}s)} \\ &= \overline{f_{11}s} \vee \bigvee_{n\in\omega} \overline{f_{21}f_{22}^n f_{12}s}.\end{aligned}$$

Since all of the morphisms $\overline{f_{11}s}$ and $\overline{f_{21}f_{22}^n f_{12}s}$ for any $n \in \omega$ are restriction idempotents $I \to I$, it follows for each of them that they are either equal to id_I or to $0_{I,I}$. Suppose that none of these are equal to the identity id_I. Then they must all be $0_{I,I}$, and so $\overline{\mathrm{Tr}^U_{A,B}(f)s} = 0_{I,I} \neq \mathrm{id}_I$, contradicting totality. On the other hand, suppose that there exists an identity among these. Then, it follows by the disjointness property above that the rest must be $0_{I,I}$. □

With these done, we are finally ready to tackle the adequacy theorem.

Theorem 6.10 (Adequacy)**.** *If* $[\![c]\!]\,[\![\sigma]\!]$ *is total then there exists* σ' *such that* $\sigma \vdash c \downarrow \sigma'$.

Proof. By induction on the structure of c.

- Case $c = \textbf{skip}$. Then $\sigma \vdash \textbf{skip} \downarrow \sigma$ by $\dfrac{}{\sigma \vdash \textbf{skip} \downarrow \sigma}$.
- Case $c = c_1 \ ; \ c_2$.

  In this case, $[\![c]\!]\,[\![\sigma]\!] = [\![c_1 \ ; \ c_2]\!]\,[\![\sigma]\!] = [\![c_2]\!]\,[\![c_1]\!]\,[\![\sigma]\!]$. Since this is total, so is $[\![c_1]\!]\,[\![\sigma]\!]$ by Lemma 6.8. But then, by induction, there exists σ' such that $\sigma \vdash c_1 \downarrow \sigma'$ by some derivation $\mathcal{D}_1$, and by Lemma 6.6, $[\![c_1]\!]\,[\![\sigma]\!] = [\![\sigma']\!]$. But then $[\![c_2]\!]\,[\![c_1]\!]\,[\![\sigma]\!] = [\![c_2]\!]\,[\![\sigma']\!]$, so by induction there exists σ'' such that $\sigma' \vdash c_2 \downarrow \sigma''$ by some derivation $\mathcal{D}_2$. But then $\sigma \vdash c_1 \ ; \ c_2 \downarrow \sigma''$ by

$$\dfrac{\overset{\mathcal{D}_1}{\sigma \vdash c_1 \downarrow \sigma'} \qquad \overset{\mathcal{D}_2}{\sigma' \vdash c_2 \downarrow \sigma''}}{\sigma \vdash c_1 \ ; \ c_2 \downarrow \sigma''} \ .$$

- Case $c = \textbf{if}\ p\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \textbf{fi}\ q$.

  Thus, $[\![c]\!]\,[\![\sigma]\!] = [\![\textbf{if}\ p\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \textbf{fi}\ q]\!]\,[\![\sigma]\!] = [\![q]\!]^\dagger\,([\![c_1]\!] \oplus [\![c_2]\!])\,[\![p]\!]\,[\![\sigma]\!]$, and since this is total, $[\![p]\!]\,[\![\sigma]\!]$ is total as well by analogous argument to the previous case. It then follows by Lemma 6.4 that there exists b such that $\sigma \vdash p \downarrow b$ by some derivation $\mathcal{D}_1$, and by Lemma 6.2, $[\![p]\!]\,[\![\sigma]\!] = [\![b]\!]\,[\![\sigma]\!]$. We have two cases depending on what b is.

  When $b = tt$ we have

$$\begin{aligned}[\![c]\!]\,[\![\sigma]\!] &= [\![q]\!]^\dagger\,([\![c_1]\!] \oplus [\![c_2]\!])\,[\![p]\!]\,[\![\sigma]\!] = [\![q]\!]^\dagger\,([\![c_1]\!] \oplus [\![c_2]\!])\,[\![tt]\!]\,[\![\sigma]\!] \\ &= [\![q]\!]^\dagger\,([\![c_1]\!] \oplus [\![c_2]\!])\,\amalg_1\,[\![\sigma]\!] = [\![q]\!]^\dagger\,\amalg_1\,[\![c_1]\!]\,[\![\sigma]\!]\end{aligned}$$

  so since this is total, $[\![c_1]\!]\,[\![\sigma]\!]$ must be total as well by Lemma 6.8. But then, by induction, there exists σ' such that $\sigma \vdash c_1 \downarrow \sigma'$ by some derivation $\mathcal{D}_2$, and by Lemma 6.6, $[\![c_1]\!]\,[\![\sigma]\!] = [\![\sigma']\!]$. Continuing the computation, we get

$$\begin{aligned}[\![c]\!]\,[\![\sigma]\!] &= [\![q]\!]^\dagger\,\amalg_1\,[\![c_1]\!]\,[\![\sigma]\!] = [\![q]\!]^\dagger\,\amalg_1\,[\![\sigma']\!] = (\amalg_1^\dagger\,[\![q]\!])^\dagger\,[\![\sigma']\!] = \overline{\amalg_1^\dagger\,[\![q]\!]}^\dagger\,[\![\sigma']\!] \\ &= \overline{\amalg_1^\dagger\,[\![q]\!]}\,[\![\sigma']\!] = [\![\sigma']\!]\,\overline{\amalg_1^\dagger\,[\![q]\!]\,[\![\sigma']\!]} = [\![\sigma']\!]\,\overline{\amalg_1^\dagger\,[\![q]\!]\,[\![\sigma']\!]}\;\overline{[\![q]\!]\,[\![\sigma']\!]}\end{aligned}$$

  so $\overline{[\![q]\!]\,[\![\sigma']\!]}$ must be total, in turn meaning that $[\![q]\!]\,[\![\sigma']\!]$ must be total. But then by Lemma 6.4, there must exist b' such that $\sigma' \vdash q \downarrow b'$ by some derivation $\mathcal{D}_3$, with $[\![q]\!]\,[\![\sigma']\!] = [\![b']\!]\,[\![\sigma']\!]$ by Lemma 6.2. Again, we have two cases depending on b'. If $b' = tt$, we derive $\sigma \vdash \textbf{if}\ p\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \textbf{fi}\ q \downarrow \sigma'$ by

$$\dfrac{\overset{\mathcal{D}_1}{\sigma \vdash p \downarrow tt} \qquad \overset{\mathcal{D}_2}{\sigma \vdash c_1 \downarrow \sigma'} \qquad \overset{\mathcal{D}_3}{\sigma' \vdash q \downarrow tt}}{\sigma \vdash \textbf{if}\ p\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \textbf{fi}\ q \downarrow \sigma'}$$

  On the other hand, when $b' = f\!f$, we have

$$\begin{aligned}[\![c]\!]\,[\![\sigma]\!] &= [\![\sigma']\!]\,\overline{\amalg_1^\dagger\,[\![q]\!]\,[\![\sigma']\!]} = [\![\sigma']\!]\,\overline{\amalg_1^\dagger\,[\![f\!f]\!]\,[\![\sigma']\!]} = [\![\sigma']\!]\,\overline{\amalg_1^\dagger\,\amalg_2\,[\![\sigma']\!]} \\ &= [\![\sigma']\!]\,\overline{0_{\Sigma,\Sigma}\,[\![\sigma']\!]} = [\![\sigma']\!]\,\overline{0_{I,\Sigma}} = [\![\sigma']\!]\,0_{I,I} = 0_{I,\Sigma}\end{aligned}$$

  so $\overline{[\![c]\!]\,[\![\sigma]\!]} = 0_{I,I}$, contradicting $\overline{[\![c]\!]\,[\![\sigma]\!]} = \mathrm{id}_I$ since $0_{I,I} \neq \mathrm{id}_I$ by definition of a model. Thus there exists σ' such that $\sigma \vdash \textbf{if}\ p\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \textbf{fi}\ q \downarrow \sigma'$.

To show the case when $b = \mathit{ff}$, we proceed as before. We then have

$$\begin{aligned}\llbracket c \rrbracket \llbracket \sigma \rrbracket &= \llbracket q \rrbracket^\dagger (\llbracket c_1 \rrbracket \oplus \llbracket c_2 \rrbracket) \llbracket p \rrbracket \llbracket \sigma \rrbracket = \llbracket q \rrbracket^\dagger (\llbracket c_1 \rrbracket \oplus \llbracket c_2 \rrbracket) \llbracket \mathit{ff} \rrbracket \llbracket \sigma \rrbracket \\ &= \llbracket q \rrbracket^\dagger (\llbracket c_1 \rrbracket \oplus \llbracket c_2 \rrbracket) \amalg_2 \llbracket \sigma \rrbracket = \llbracket q \rrbracket^\dagger \amalg_2 \llbracket c_2 \rrbracket \llbracket \sigma \rrbracket\end{aligned}$$

So $\llbracket c_2 \rrbracket \llbracket \sigma \rrbracket$ must be total by Lemma 6.8, which means that by induction there exists σ' such that $\sigma \vdash c_2 \downarrow \sigma'$ by a derivation $\mathcal{D}_2$, and by Lemma 6.6, $\llbracket c_2 \rrbracket \llbracket \sigma \rrbracket = \llbracket \sigma' \rrbracket$. Continuing as before, we obtain now that

$$\begin{aligned}\llbracket c \rrbracket \llbracket \sigma \rrbracket &= \llbracket q \rrbracket^\dagger \amalg_2 \llbracket c_2 \rrbracket \llbracket \sigma \rrbracket = \llbracket q \rrbracket^\dagger \amalg_2 \llbracket \sigma' \rrbracket = (\amalg_2^\dagger \llbracket q \rrbracket)^\dagger \llbracket \sigma' \rrbracket = \overline{\amalg_2^\dagger \llbracket q \rrbracket}^\dagger \llbracket \sigma' \rrbracket \\ &= \overline{\amalg_2^\dagger \llbracket q \rrbracket} \llbracket \sigma' \rrbracket = \llbracket \sigma' \rrbracket \overline{\amalg_2^\dagger \llbracket q \rrbracket \llbracket \sigma' \rrbracket} = \llbracket \sigma' \rrbracket \overline{\amalg_2^\dagger \llbracket q \rrbracket \llbracket \sigma' \rrbracket} \; \overline{\llbracket q \rrbracket \llbracket \sigma' \rrbracket}\end{aligned}$$

and so $\llbracket q \rrbracket \llbracket \sigma' \rrbracket$ must be total in this case as well, so by Lemma 6.4 there must exist b' such that $\sigma' \vdash q \downarrow b'$ by some derivation $\mathcal{D}_3$, and $\llbracket q \rrbracket \llbracket \sigma' \rrbracket = \llbracket b' \rrbracket \llbracket \sigma' \rrbracket$ by Lemma 6.2. Again, we do a case analysis depending on the value of b'.

If $b' = tt$, we have

$$\begin{aligned}\llbracket c \rrbracket \llbracket \sigma \rrbracket &= \llbracket \sigma' \rrbracket \overline{\amalg_2^\dagger \llbracket q \rrbracket \llbracket \sigma' \rrbracket} = \llbracket \sigma' \rrbracket \overline{\amalg_2^\dagger \llbracket tt \rrbracket \llbracket \sigma' \rrbracket} = \llbracket \sigma' \rrbracket \overline{\amalg_2^\dagger \amalg_1 \llbracket \sigma' \rrbracket} \\ &= \llbracket \sigma' \rrbracket \overline{0_{\Sigma,\Sigma} \llbracket \sigma' \rrbracket} = \llbracket \sigma' \rrbracket \overline{0_{I,\Sigma}} = \llbracket \sigma' \rrbracket 0_{I,I} = 0_{I,\Sigma}\end{aligned}$$

which contradicts the totality of $\llbracket c \rrbracket \llbracket \sigma \rrbracket$ by $\overline{\llbracket c \rrbracket \llbracket \sigma \rrbracket} = \overline{0_{I,\Sigma}} = 0_{I,I} \neq \mathrm{id}_I$, and we get by contradiction that there exists σ' such that $\sigma \vdash \textbf{if } p \textbf{ then } c_1 \textbf{ else } c_2 \textbf{ fi } q \downarrow \sigma'$.

If $b' = \mathit{ff}$, we derive $\sigma \vdash \textbf{if } p \textbf{ then } c_1 \textbf{ else } c_2 \textbf{ fi } q \downarrow \sigma'$ by

$$\frac{\overset{\mathcal{D}_1}{\sigma \vdash p \downarrow \mathit{ff}} \qquad \overset{\mathcal{D}_2}{\sigma \vdash c_2 \downarrow \sigma'} \qquad \overset{\mathcal{D}_3}{\sigma' \vdash q \downarrow \mathit{ff}}}{\sigma \vdash \textbf{if } p \textbf{ then } c_1 \textbf{ else } c_2 \textbf{ fi } q \downarrow \sigma'} .$$

- Case $c = \textbf{from } p \textbf{ loop } c_1 \textbf{ until } q$.

In this case, $\llbracket c \rrbracket \llbracket \sigma \rrbracket = \llbracket \textbf{from } p \textbf{ loop } c_1 \textbf{ until } q \rrbracket \llbracket \sigma \rrbracket = \mathrm{Tr}^\Sigma_{\Sigma,\Sigma}(\beta[p, c_1, q]) \llbracket \sigma \rrbracket$. Since this is total and $\llbracket \sigma \rrbracket : I \to \Sigma$, it follows by Lemma 6.9 that either

$$\llbracket \textbf{from } p \textbf{ loop } c_1 \textbf{ until } q \rrbracket \llbracket \sigma \rrbracket = \beta[p, c_1, q]_{11} \llbracket \sigma \rrbracket$$

or there exists $n \in \omega$ such that

$$\llbracket \textbf{from } p \textbf{ loop } c_1 \textbf{ until } q \rrbracket \llbracket \sigma \rrbracket = \beta[p, c_1, q]_{21} \beta[p, c_1, q]_{22}^n \beta[p, c_1, q]_{12} \llbracket \sigma \rrbracket .$$

If $\llbracket \textbf{from } p \textbf{ loop } c_1 \textbf{ until } q \rrbracket \llbracket \sigma \rrbracket = \beta[p, c_1, q]_{11} \llbracket \sigma \rrbracket$, we have

$$\begin{aligned}\llbracket \textbf{from } p \textbf{ loop } c_1 \textbf{ until } q \rrbracket \llbracket \sigma \rrbracket &= \beta[p, c_1, q]_{11} \llbracket \sigma \rrbracket = \amalg_1^\dagger (\mathrm{id}_\Sigma \oplus \llbracket c_1 \rrbracket) \llbracket q \rrbracket \llbracket p \rrbracket^\dagger \amalg_1 \llbracket \sigma \rrbracket \\ &= \amalg_1^\dagger (\mathrm{id}_\Sigma \oplus \llbracket c_1 \rrbracket) \llbracket q \rrbracket \overline{\amalg_1^\dagger \llbracket p \rrbracket} \llbracket \sigma \rrbracket \\ &= \amalg_1^\dagger (\mathrm{id}_\Sigma \oplus \llbracket c_1 \rrbracket) \llbracket q \rrbracket \llbracket \sigma \rrbracket \overline{\amalg_1^\dagger \llbracket p \rrbracket \llbracket \sigma \rrbracket} \\ &= \amalg_1^\dagger (\mathrm{id}_\Sigma \oplus \llbracket c_1 \rrbracket) \llbracket q \rrbracket \llbracket \sigma \rrbracket \overline{\amalg_1^\dagger \llbracket p \rrbracket \llbracket \sigma \rrbracket} \; \overline{\llbracket q \rrbracket \llbracket \sigma \rrbracket} \; \overline{\llbracket p \rrbracket \llbracket \sigma \rrbracket}\end{aligned}$$

so it follows by totality of $\llbracket c \rrbracket \llbracket \sigma \rrbracket$ that $\overline{\llbracket p \rrbracket \llbracket \sigma \rrbracket}$ and $\overline{\llbracket q \rrbracket \llbracket \sigma \rrbracket}$ must be total, so $\llbracket p \rrbracket \llbracket \sigma \rrbracket$ and $\llbracket q \rrbracket \llbracket \sigma \rrbracket$ must be total as well. It then follows by Lemma 6.4 that there exist b_1 and b_2 such that $\sigma \vdash p \downarrow b_1$ and $\sigma \vdash q \downarrow b_2$ by derivations $\mathcal{D}_1$ respectively $\mathcal{D}_2$. But then it follows by Lemma 6.5 that $b_1 = b_2 = tt$, as we would otherwise have $[\![c]\!]\,[\![\sigma]\!] = \beta[p,c_1,q]_{11}\,[\![\sigma]\!] = 0_{I,\Sigma}$, contradicting totality. Thus we may derive $\sigma \vdash \textbf{from}\ p\ \textbf{loop}\ c_1\ \textbf{until}\ q \downarrow \sigma$ by

$$\frac{\overset{\mathcal{D}_1}{\sigma \vdash p \downarrow tt} \qquad \overset{\mathcal{D}_2}{\sigma \vdash q \downarrow tt}}{\sigma \vdash \textbf{from}\ p\ \textbf{loop}\ c_1\ \textbf{until}\ q \downarrow \sigma}.$$

On the other hand, suppose that there exists $n \in \omega$ such that

$$[\![c]\!]\,[\![\sigma]\!] = [\![\textbf{from}\ p\ \textbf{loop}\ c_1\ \textbf{until}\ q]\!]\,[\![\sigma]\!] = \beta[p,c_1,q]_{21}\beta[p,c_1,q]_{22}^n\beta[p,c_1,q]_{12}\,[\![\sigma]\!]\,.$$

Since this is total, by Lemma 6.8 $\beta[p,c_1,q]_{12}\,[\![\sigma]\!]$ is total as well, and

$$\begin{aligned}\beta[p,c_1,q]_{12}\,[\![\sigma]\!] &= \amalg_2^\dagger(\mathrm{id}_\Sigma \oplus [\![c_1]\!])\,[\![q]\!]\,[\![p]\!]^\dagger\,\amalg_1\,[\![\sigma]\!] = \amalg_2^\dagger(\mathrm{id}_\Sigma \oplus [\![c_1]\!])\,[\![q]\!]\,\overline{\amalg_1^\dagger\,[\![p]\!]}\,[\![\sigma]\!] \\ &= \amalg_2^\dagger(\mathrm{id}_\Sigma \oplus [\![c_1]\!])\,[\![q]\!]\,[\![\sigma]\!]\,\overline{\amalg_1^\dagger\,[\![p]\!]\,[\![\sigma]\!]} \\ &= \amalg_2^\dagger(\mathrm{id}_\Sigma \oplus [\![c_1]\!])\,[\![q]\!]\,[\![\sigma]\!]\,\overline{\amalg_1^\dagger\,[\![p]\!]\,[\![\sigma]\!]}\,\overline{[\![p]\!]\,[\![\sigma]\!]}\ \overline{[\![q]\!]\,[\![\sigma]\!]}\end{aligned}$$

But then $[\![p]\!]\,[\![\sigma]\!]$ and $[\![q]\!]\,[\![\sigma]\!]$ must be total as well, so by Lemma 6.4 there exist b_1 and b_2 such that $\sigma \vdash p \downarrow b_1$ and $\sigma \vdash q \downarrow b_2$ by derivations $\mathcal{D}_1$ respectively $\mathcal{D}_2$. Since $\beta[p,c_1,q]_{12}\,[\![\sigma]\!]$ is total, it further follows by Lemma 6.5 that $b_1 = tt$ and $b_2 = ff$, as we would otherwise have $\beta[p,c_1,q]_{12}\,[\![\sigma]\!] = 0_{I,\Sigma}$. Further, $\beta[p,c_1,q]_{12}\,[\![\sigma]\!] = [\![c_1]\!]\,[\![\sigma]\!]$, and since this is total, by induction there exists σ' such that $\sigma \vdash c_1 \downarrow \sigma'$ by some derivation $\mathcal{D}_3$, with $[\![c_1]\!]\,[\![\sigma]\!] = [\![\sigma']\!]$ by Lemma 6.6.

To summarize, we have now that

$$\begin{aligned}[\![\textbf{from}\ p\ \textbf{loop}\ c_1\ \textbf{until}\ q]\!]\,[\![\sigma]\!] &= \beta[p,c_1,q]_{21}\beta[p,c_1,q]_{22}^n\beta[p,c_1,q]_{12}\,[\![\sigma]\!] \\ &= \beta[p,c_1,q]_{21}\beta[p,c_1,q]_{22}^n\,[\![c_1]\!]\,[\![\sigma]\!] \\ &= \beta[p,c_1,q]_{21}\beta[p,c_1,q]_{22}^n\,[\![\sigma']\!]\end{aligned}$$

is total, and we have derivations $\mathcal{D}_1$ of $\sigma \vdash p \downarrow tt$, $\mathcal{D}_2$ of $\sigma \vdash q \downarrow ff$, and $\mathcal{D}_3$ of $\sigma \vdash c_1 \downarrow \sigma'$. To finish the proof, we show by induction on n that if $\beta[p,c_1,q]_{21}\beta[p,c_1,q]_{22}^n\,[\![\sigma']\!]$ is total for any state σ', there exists a state σ'' such that $\sigma' \vdash \underline{loop}[p,c_1,q] \downarrow \sigma''$ by some derivation $\mathcal{D}_4$.

- In the base case $n = 0$, so $\beta[p,c_1,q]_{21}\beta[p,c_1,q]_{22}^n\,[\![\sigma']\!] = \beta[p,c_1,q]_{21}\,[\![\sigma']\!]$. By proof analogous to previous cases, we have that $[\![p]\!]\,[\![\sigma']\!]$ and $[\![q]\!]\,[\![\sigma']\!]$ must be total, so by Lemma 6.4 there exist b_1' and b_2' such that $\sigma' \vdash p \downarrow b_1'$ and $\sigma' \vdash q \downarrow b_1'$ by derivations $\mathcal{D}_1'$ respectively $\mathcal{D}_2'$. Further, by totality, it follows by Lemma 6.5 that we must have $b_1' = ff$, $b_2' = tt$. Thus we may produce our derivation $\mathcal{D}_4$ of $\sigma' \vdash \underline{loop}[p,c_1,q] \downarrow \sigma'$ by

$$\frac{\overset{\mathcal{D}_1'}{\sigma' \vdash p \downarrow ff} \qquad \overset{\mathcal{D}_2'}{\sigma' \vdash q \downarrow tt}}{\sigma' \vdash \underline{loop}[p,c,q] \downarrow \sigma'}$$

- In the inductive case, we have that

$$\beta[p,c_1,q]_{21}\beta[p,c_1,q]_{22}^{n+1}\,[\![\sigma']\!] = \beta[p,c_1,q]_{21}\beta[p,c_1,q]_{22}^n\beta[p,c_1,q]_{22}\,[\![\sigma']\!]$$

and since this is assumed to be total, it follows by Lemma 6.8 that $\beta[p,c_1,q]_{22}\,[\![\sigma']\!]$ is total as well. Again, by argument analogous to previous cases, this implies that $[\![p]\!]\,[\![\sigma']\!]$ and $[\![q]\!]\,[\![\sigma']\!]$ are both total, so by Lemma 6.4 there exist b_1' and b_2' such that $\sigma' \vdash p \downarrow b_1'$ and $\sigma' \vdash q \downarrow b_2'$ by derivations $\mathcal{D}_1'$ respectively $\mathcal{D}_2'$. Likewise, it then follows by Lemma 6.5 that since this is total, we must have $b_1' = b_2' = \mathit{ff}$, and so $\beta[p, c_1, q]_{22} \llbracket \sigma' \rrbracket = \llbracket c_1 \rrbracket \llbracket \sigma' \rrbracket$, again by Lemma 6.5. Since $\llbracket c_1 \rrbracket \llbracket \sigma' \rrbracket$ is total, by the outer induction hypothesis there exists a derivation $\mathcal{D}_3'$ of $\sigma' \vdash c_1 \downarrow \sigma''$ for some σ'', and so $\llbracket c_1 \rrbracket \llbracket \sigma' \rrbracket = \llbracket \sigma'' \rrbracket$ by Lemma 6.6. But then

$$\begin{aligned}
\beta[p, c_1, q]_{21} \beta[p, c_1, q]_{22}^{n+1} \llbracket \sigma' \rrbracket &= \beta[p, c_1, q]_{21} \beta[p, c_1, q]_{22}^{n} \beta[p, c_1, q]_{22} \llbracket \sigma' \rrbracket \\
&= \beta[p, c_1, q]_{21} \beta[p, c_1, q]_{22}^{n} \llbracket c_1 \rrbracket \llbracket \sigma' \rrbracket \\
&= \beta[p, c_1, q]_{21} \beta[p, c_1, q]_{22}^{n} \llbracket \sigma'' \rrbracket
\end{aligned}$$

since this is total, by (inner) induction there exists a derivation $\mathcal{D}_4'$ of $\sigma'' \vdash \underline{loop}[p, c_1, q] \downarrow \sigma'''$. Thus, we may produce our derivation $\mathcal{D}_4$ as

$$\frac{\overset{\mathcal{D}_1'}{\sigma' \vdash p \downarrow \mathit{ff}} \qquad \overset{\mathcal{D}_2'}{\sigma' \vdash q \downarrow \mathit{ff}} \qquad \overset{\mathcal{D}_3'}{\sigma' \vdash c_1 \downarrow \sigma''} \qquad \overset{\mathcal{D}_4'}{\sigma'' \vdash \underline{loop}[p, c, q] \downarrow \sigma'''}}{\sigma' \vdash \underline{loop}[p, c_1, q] \downarrow \sigma'''}$$

concluding the internal lemma.

Since this is the case, we may finally show $\sigma \vdash \textbf{from } p \textbf{ loop } q \textbf{ until } c \downarrow \sigma'$ by

$$\frac{\overset{\mathcal{D}_1}{\sigma \vdash p \downarrow \mathit{tt}} \qquad \overset{\mathcal{D}_2}{\sigma \vdash q \downarrow \mathit{ff}} \qquad \overset{\mathcal{D}_3}{\sigma \vdash c \downarrow \sigma'} \qquad \overset{\mathcal{D}_4}{\sigma' \vdash \underline{loop}[p, c, q] \downarrow \sigma''}}{\sigma \vdash \textbf{from } p \textbf{ loop } c \textbf{ until } q \downarrow \sigma''},$$

which concludes the proof.

□

7. Full abstraction

Where soundness and adequacy state an agreement in the notions of convergence between the operational and denotational semantics, full abstraction deals with their respective notions of equivalence (see, *e.g.*, [24]). Unlike the case for soundness and adequacy, where defining a proper notion of convergence required more work on the categorical side, the tables have turned when it comes to program equivalence. In the categorical semantics, program equivalence is clear – equality of interpretations. Operationally, however, there is nothing immediately corresponding to equality of behaviour at runtime.

To produce this, we consider how two programs may behave when executed from the same start state. If they always produce the same result, we say that they are observationally equivalent. Formally, we define this as follows:

Definition 13. Say that programs p_1 and p_2 are *observationally equivalent*, denoted $p_1 \approx p_2$, if for all states σ, $\sigma \vdash p_1 \downarrow \sigma'$ if and only if $\sigma \vdash p_2 \downarrow \sigma'$.

A model is said to be fully abstract if these two notions of program equivalence are in agreement. In the present section, we will show a sufficient condition for full abstraction of models of structured reversible flowchart languages. This condition will be that the given model additionally has the properties of being *I-well pointed* and *bijective on states.*

Definition 14. Say that a model of a structured reversible flowchart language is *I-well pointed* if, for all parallel morphisms $f, g : A \to B$, $f = g$ precisely when $fp = gp$ for all $p : I \to A$.

Definition 15. Say that a model $\mathscr{C}$ of a structured reversible flowchart language $\mathcal{L}$ is *bijective on states* if there is a bijective correspondence between states of $\mathcal{L}$ and *total* morphisms $I \to \Sigma$ of $\mathscr{C}$.

If a sound and adequate model of a structured reversible flowchart language is bijective on states, we can show a stronger version of Lemma 6.6.

Lemma 7.1. *If $\mathscr{C}$ be a sound and adequate model of a structured reversible flowchart language which is bijective on states. Then $\sigma \vdash c \downarrow \sigma'$ iff $[\![c]\!]\,[\![\sigma]\!] = [\![\sigma']\!]$ and this is total.*

Proof. By Lemma 6.6 and Theorem 6.7, we only need to show that $[\![c]\!]\,[\![\sigma]\!] = [\![\sigma']\!]$ implies $\sigma \vdash c \downarrow \sigma'$. Assume that $[\![c]\!]\,[\![\sigma]\!] = [\![\sigma']\!]$ and that this is total. By Theorem 6.10, there exists σ'' such that $\sigma \vdash c \downarrow \sigma''$, so by Lemma 6.6, $[\![c]\!]\,[\![\sigma]\!] = [\![\sigma'']\!]$. Thus $[\![\sigma']\!] = [\![c]\!]\,[\![\sigma]\!] = [\![\sigma'']\!]$, so $\sigma' = \sigma''$ by bijectivity on states. □

With this, we can show full abstraction.

Theorem 7.2 (Full abstraction)**.** *Let $\mathscr{C}$ be a sound and adequate model of a structured reversible flowchart language that is furthermore I-well pointed and bijective on states. Then $\mathscr{C}$ is fully abstract,* i.e., *$p_1 \approx p_2$ if and only if $[\![p_1]\!] = [\![p_2]\!]$.*

Proof. Suppose $p_1 \approx p_2$, *i.e.*, for all σ, $\sigma \vdash p_1 \downarrow \sigma'$ if and only if $\sigma \vdash p_2 \downarrow \sigma'$. Let $s : I \to \Sigma$ be a morphism. Since $\mathscr{C}$ is a model of a structured reversible flowchart language, it follows that either $\overline{s} = \mathrm{id}_I$ or $\overline{s} = 0_{I,I}$ where $\mathrm{id}_I \neq 0_{I,I}$.

If $\overline{s} = 0_{I,I}$, we have $[\![p_1]\!]\,s = [\![p_2]\!]\,s = 0_{I,\Sigma}$ by unicity of the zero map.

On the other hand, if $\overline{s} = \mathrm{id}_I$, by bijectivity on states there exists σ_0 such that $s = [\![\sigma_0]\!]$. Consider now $[\![p_1]\!]\,s = [\![p_1]\!]\,[\![\sigma_0]\!]$. If this is total, by Theorem 6.10 there exists σ_0' such that $\sigma_0 \vdash p_1 \downarrow \sigma_0'$, and by $p_1 \approx p_2$, $\sigma_0 \vdash p_2 \downarrow \sigma_0'$ as well. But then, applying Lemma 6.6 on both yields that

$$[\![p_1]\!]\,s = [\![p_1]\!]\,[\![\sigma_0]\!] = [\![\sigma_0']\!] = [\![p_2]\!]\,[\![\sigma_0]\!] = [\![p_2]\!]\,s\ .$$

If, on the other hand, $[\![p_1]\!]\,s$ is not total, by the contrapositive to Theorem 6.7, there exists *no* σ_0' such that $\sigma_0 \vdash p_1 \downarrow \sigma_0'$, so by $p_1 \approx p_2$ there exists no σ_0'' such that $\sigma_0 \vdash p_1 \downarrow \sigma_0''$. But then, by the contrapositive of Theorem 6.10 and the fact that restriction idempotents on I are either id_I or $0_{I,I}$, it follows that

$$[\![p_1]\!]\,s = [\![p_1]\!]\,[\![\sigma_0]\!] = 0_{I,\Sigma} = [\![p_2]\!]\,[\![\sigma_0]\!] = [\![p_1]\!]\,s.$$

Since s was chosen arbitrarily and $[\![p_1]\!]\,s = [\![p_2]\!]\,s$ in all cases, it follows by I-well pointedness that $[\![p_1]\!] = [\![p_2]\!]$.

In the other direction, suppose $[\![p_1]\!] = [\![p_2]\!]$, let σ_0 be a state, and suppose that there exists σ_0' such that $\sigma_0 \vdash p_1 \downarrow \sigma_0'$. By Lemma 6.6, $[\![p_1]\!]\,[\![\sigma_0]\!] = [\![\sigma_0']\!]$, and by Theorem 6.7 this is total. But then, by $[\![p_1]\!] = [\![p_2]\!]$, $[\![p_2]\!]\,[\![\sigma_0]\!] = [\![p_1]\!]\,[\![\sigma_0]\!] = [\![\sigma_0']\!]$, so by Lemma 7.1, $\sigma_0 \vdash p_2 \downarrow \sigma_0'$. The other direction follows similarly. □

8. Applications

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

8.1. **Extracting a program inverter.** A desirable syntactic property for reversible programming languages is to be closed under program inversion, in the sense that for each program p, there is another program $\mathcal{I}[p]$ such that $[\![\mathcal{I}[p]]\!] = [\![p]\!]^\dagger$. Janus, `R-WHILE`, and `R-CORE` [32, 16, 17] are all examples of reversible programming languages with this property. This is typically witnessed by a *program inverter* [1], that is, a procedure mapping the program text of a program to the program text of its inverse program[3].

Suppose that we are given a language where elementary operations are closed under program inversion (*i.e.*, where each elementary operation b has an inverse $\mathcal{I}[b]$ such that $[\![\mathcal{I}[b]]\!] = [\![b]\!]^\dagger$). We can extend this to a program inverter for **skip**, sequencing, reversible conditionals and loops as follows, by structural induction with the hypothesis that $[\![\mathcal{I}[c]]\!] = [\![c]\!]^\dagger$. For **skip**, we have

$$[\![\mathbf{skip}]\!]^\dagger = \mathrm{id}_\Sigma^\dagger = \mathrm{id}_\Sigma = [\![\mathbf{skip}]\!]$$

giving us the usual inversion rule of $\mathcal{I}[\mathbf{skip}] = \mathbf{skip}$. Likewise for sequences,

$$[\![c_1 \ ; \ c_2]\!]^\dagger = ([\![c_2]\!] \, [\![c_1]\!])^\dagger = [\![c_1]\!]^\dagger \, [\![c_2]\!]^\dagger = [\![\mathcal{I}[c_1]]\!] \, [\![\mathcal{I}[c_2]]\!] = [\![\mathcal{I}[c_2] \ ; \ \mathcal{I}[c_1]]\!]$$

giving us the inversion rule

$$\mathcal{I}[c_1 \ ; \ c_2] = \mathcal{I}[c_2] \ ; \ \mathcal{I}[c_1] \ .$$

Our approach becomes more interesting when we come to conditionals. Given some conditional statement **if** p **then** c_1 **else** c_2 **fi** q, we notice that

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

which yields the inversion rule

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

Fortunately, this is precisely the usual inversion rule for reversible conditionals (see, *e.g.*, [16, 17]). For reversible loops, we have

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

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

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

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

which matches the usual inversion rule for reversible loops [17]. We summarize this in the following theorem:

Theorem 8.1. *If a reversible structured flowchart language is syntactically closed under inversion of elementary operations, it is also closed under inversion of reversible conditionals and loops.*

8.2. **Example: A reversible flowchart language.** Consider the following family of (neither particularly useful nor particularly useless) reversible flowchart languages for reversible computing with integer data, $\mathtt{RINT}_k$. $\mathtt{RINT}_k$ has precisely k variables available for storage, denoted $\mathtt{x_1}$ through $\mathtt{x_k}$ (of which $\mathtt{x_1}$ is designated by convention as the input/output variable), and its only atomic operations are addition and subtraction of variables, as well as addition with a constant. Variables are used as elementary predicates, with zero designating truth and non-zero values all designating falsehood. For control structures we have reversible conditionals and loops, and sequencing as usual. This gives the syntax:

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

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

We may now give semantics to this language in our framework. For a concrete model, we select the category **PInj** of sets and partial injections, which is a join inverse category with a join-preserving disjointness tensor (given on objects by the disjoint union of sets), so it is extensive in the sense of Definition 9 by Theorem 4.4. By our developments previously in this section, to give a full semantics to $\mathtt{RINT}_k$ in **PInj**, it suffices to provide an object (*i.e.*, a set) of stores Σ, denotations of our three classes of elementary operations (addition by a variable, addition by a constant, and subtraction by a variable) as morphisms (*i.e.*, partial injective functions) $\Sigma \to \Sigma$, and denotations of our class of elementary predicates (here, testing whether a variable is zero or not) as decisions $\Sigma \to \Sigma \oplus \Sigma$. These are all shown in Figure 4. It is uncomplicated to show that all of these are partial injective functions, and that the denotation of each predicate $[\![\mathtt{x_i}]\!]$ is a decision, so that this is, in fact, a model of $\mathtt{RINT}_k$ in **PInj**.

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

Theorem 8.2 (Reversibility)**.** *Every* $\mathtt{RINT}_k$ *program* p *is semantically reversible in the sense that* $[\![p]\!]$ *is a partial isomorphism.*

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

Figure 4: The object of stores and semantics of elementary operations and predicates of $\mathtt{RINT}_k$ in **PInj**.

Further, since we can straightforwardly show that $[\![\mathtt{x_i} \mathrel{+}= \mathtt{x_j}]\!]^\dagger = [\![\mathtt{x_i} \mathrel{-}= \mathtt{x_j}]\!]$ and $[\![\mathtt{x_i} \mathrel{+}= \overline{n}]\!]^\dagger = [\![\mathtt{x_i} \mathrel{+}= \overline{-n}\,]\!]$, we can use the technique from Sec. 8.1 to obtain a sound and complete program inverter.

Theorem 8.3 (Program inversion)**.** $\mathtt{RINT}_k$ *has a (sound and complete) program inverter. In particular, for every* $\mathtt{RINT}_k$ *program p there exists a program $\mathcal{I}[p]$ such that* $[\![\mathcal{I}[p]]\!] = [\![p]\!]^\dagger$.

8.3. **Decisions as a programming technique.** Decisions offer a solution to the awkwardness in representing predicates reversibly. On the programming side, the reversible *duplication/equality operator* [15] (see also [31]) can be seen as a distant ancestor to predicates-as-decisions, in that it provides an ad-hoc solution to the problem of checking whether two values are equal in a reversible manner.

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

data *PBool* α = *True* α | *False* α

storing not only the *result*, but also *what* was tested to begin with. With this definition, negation on these polymorphic Booleans (*pnot*) may be defined straightforwardly as shown in Figure 5. In turn, this allows for more complex predicates to be expressed in a largely familiar way. For example, the decision for testing whether a natural number is even (*peven*) is also shown in Figure 5, with *fmap* given in the straightforward way on polymorphic Booleans. For comparison, the corresponding irreversible predicate is typically defined as follows, with *not* the usual negation of Booleans

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

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

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

Figure 5: The definition of the *even*-predicate as a decision on natural numbers.

9. Concluding remarks

In the present paper, we have built on the work on extensive restriction categories to derive a related concept of extensivity for inverse categories. We have used this concept to give a novel reversible representation of predicates and their corresponding assertions in (specifically extensive) join inverse categories with a disjointness tensor, and in turn used these to model the fundamental control structures of reversible loops and conditionals in structured reversible flowchart languages. We have shown that these categorical semantics are sound and adequate with respect to the operational semantics, and given a sufficient condition for full abstraction.

Further, this approach also allowed us to derive a program inversion theorem for structured reversible flowchart languages, and we illustrated our approach by developing a family of structured reversible flowchart languages and using our framework to give it denotational semantics, with theorems regarding reversibility and program inversion for free.

The idea to represent predicates by decisions was partially inspired by the *instruments* associated with predicates in Effectus theory [19]. Given that *side effect free* instruments ι satisfy a similar rule, $\nabla\iota = \mathrm{id}$, and that Boolean effecti are extensive, it could be interesting to explore the connections between extensive restriction categories and Boolean effecti, especially as regards their internal logic.

Finally, on the programming language side, it could be interesting to further explore how decisions can be used in reversible programming, *e.g.*, to do the heavy lifting involved in pattern matching and branch joining. As our focus has been on the representation of predicates, our approach may be easily adapted to other reversible flowchart structures, *e.g.*, Janus-style loops [32].

References

[1] Abramov, S. M. and R. Glück, *The universal resolving algorithm: inverse computation in a functional language*, in: R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction. Proceedings*, LNCS 1837 (2000), pp. 187–212.

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

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

[4] Axelsen, H. B. and R. Kaarsgaard, *Join inverse categories as models of reversible recursion*, in: B. Jacobs and C. Löding, editors, *FOSSACS 2016, Proceedings*, number 9634 in LNCS (2016), pp. 73–90.

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

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

[7] Cockett, J. R. B., *Itegories & PCAs* (2007), slides from talk at FMCS 2007.

[8] Cockett, J. R. B. and S. Lack, *Restriction categories I: Categories of partial maps*, Theoretical Computer Science **270** (2002), pp. 223–259.
[9] Cockett, J. R. B. and S. Lack, *Restriction categories II: Partial map classification*, Theoretical Computer Science **294** (2003), pp. 61–102.
[10] Cockett, R. and S. Lack, *Restriction categories III: Colimits, partial limits and extensivity*, Mathematical Structures in Computer Science **17** (2007), pp. 775–817.
[11] Fiore, M. P., "Axiomatic Domain Theory in Categories of Partial Maps," Ph.D. thesis, University of Edinburgh (1994).
[12] Frank, M. P., "Reversibility for efficient computing," Ph.D. thesis, EECS Dept., Massachusetts Institute of Technology (1999).
[13] Giles, B. G., "An Investigation of some Theoretical Aspects of Reversible Computing," Ph.D. thesis, University of Calgary (2014).
[14] Glück, R. and R. Kaarsgaard, *A categorical foundation for structured reversible flowchart languages*, in: *Proceedings of the 33rd Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIII)*, 2017, to appear.
[15] Glück, R. and M. Kawabe, *A program inverter for a functional language with equality and constructors*, in: A. Ohori, editor, *Programming Languages and Systems. Proceedings*, Lecture Notes in Computer Science **2895** (2003), pp. 246–264.
[16] Glück, R. and T. Yokoyama, *A linear-time self-interpreter of a reversible imperative language*, Computer Software **33** (2016), pp. 108–128.
[17] Glück, R. and T. Yokoyama, *A minimalist's reversible while language*, IEICE Transactions on Information and Systems **E100-D** (2017), pp. 1026–1034.
[18] Guo, X., "Products, Joins, Meets, and Ranges in Restriction Categories," Ph.D. thesis, University of Calgary (2012).
[19] Jacobs, B., *New directions in categorical logic, for classical, probabilistic and quantum logic*, Logical Methods in Computer Science **11** (2015), pp. 1–76.
[20] Joyal, A., R. Street and D. Verity, *Traced monoidal categories*, Mathematical Proceedings of the Cambridge Philosophical Society **119** (1996), pp. 447–468.
[21] Kaarsgaard, R., H. B. Axelsen and R. Glück, *Join inverse categories and reversible recursion*, Journal of Logical and Algebraic Methods in Programming **87** (2017), pp. 33–50.
[22] Kastl, J., *Inverse categories*, in: H.-J. Hoehnke, editor, *Algebraische Modelle, Kategorien und Gruppoide*, Studien zur Algebra und ihre Anwendungen **7**, Akademie-Verlag, 1979 pp. 51–60.
[23] Mogensen, T. Æ., *Partial evaluation of the reversible language Janus*, in: *Partial Evaluation and Program Manipulation. Proceedings* (2011), pp. 23–32.
[24] Plotkin, G., *Full abstraction, totality and PCF*, Mathematical Structures in Computer Science **9** (1999), pp. 1–20.
[25] Selinger, P., *Dagger compact closed categories and completely positive maps*, Electronic Notes in Theoretical Computer Science **170** (2007), pp. 139–163.
[26] Selinger, P., *A survey of graphical languages for monoidal categories*, in: B. Coecke, editor, *New Structures for Physics*, number 813 in Lecture Notes in Physics (2011), pp. 289–355.
[27] Thomsen, M. K., H. B. Axelsen and R. Glück, *A reversible processor architecture and its reversible logic design*, in: A. De Vos and R. Wille, editors, *Reversible Computation. Proceedings*, Lecture Notes in Computer Science **7165** (2012), pp. 30–42.
[28] Thomsen, M. K., R. Glück and H. B. Axelsen, *Reversible arithmetic logic unit for quantum arithmetic*, Journal of Physics A: Mathematical and Theoretical **43** (2010), p. 382002.
[29] Yokoyama, T., H. B. Axelsen and R. Glück, *Reversible flowchart languages and the structured reversible program theorem*, in: L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir and I. Walukiewicz, editors, *International Colloquium on Automata, Languages and Programming. Proceedings*, Lecture Notes in Computer Science **5126** (2008), pp. 258–270.
[30] Yokoyama, T., H. B. Axelsen and R. Glück, *Optimizing clean reversible simulation of injective functions*, Journal of Multiple-Valued Logic and Soft Computing **18** (2012), pp. 5–24.
[31] Yokoyama, T., H. B. Axelsen and R. Glück, *Towards a reversible functional language*, in: A. De Vos and R. Wille, editors, *RC 2011*, LNCS **7165** (2012), pp. 14–29.
[32] Yokoyama, T. and R. Glück, *A reversible programming language and its invertible self-interpreter*, in: *Partial Evaluation and Semantics-Based Program Manipulation. Proceedings* (2007), pp. 144–153.

Reversible effects as inverse arrows

Chris Heunen[1], Robin Kaarsgaard[2], and Martti Karvonen[3]

[1] University of Edinburgh, chris.heunen@ed.ac.uk
[2] University of Copenhagen, robin@di.ku.dk
[3] University of Edinburgh, martti.karvonen@ed.ac.uk

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 concurrency 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 [29], process calculi [8], probabilistic computing [32], quantum computing [31], and robotics [30].

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 [33]. There are also various other models of reversible automata [26, 24] and combinator calculi [1, 20].

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 about, which is a boon if we want to understand the logic behind reversible programming. Second, they are not stateful by definition, which makes it easier to reverse programs. It is fair to say that existing reversible functional programming languages [21, 34] 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 [25] or more generally arrows [18].

A word on terminology. We call computation $a\colon X \to Y$ is *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 effect its *dagger*. In other words, reversible computing for us concerns dagger arrows on dagger categories, and is modeled using involutions [16]. Unfortunately, categories of partially invertible maps are called inverse categories [6], and categories of invertible maps are called groupoids [11]. 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, concurrency, dagger Frobenius monads [15, 16], recursion [22], 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 [19], 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 background, and does not adopt syntax from one specific language.

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 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*

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

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. *A* (monoidal) inverse category *is a (monoidal) dagger category of partial isometries in which positive morphisms 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 morphisms $f \colon X \to Y$ 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 3. *A dagger category is said to have* inverse products *[12] 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:*

$$\begin{tikzcd}
X \arrow[r, "\Delta_X"] \arrow[dr, "\Delta_X"'] & X \otimes X \arrow[d, "\sigma_{X,X}"] \\
& X \otimes X
\end{tikzcd}
\qquad
\begin{tikzcd}
X \arrow[rr, "\Delta_X"] \arrow[d, "\Delta_X"] & & X \otimes X \arrow[d, "\Delta_X \otimes \mathrm{id}"'] \\
X \otimes X \arrow[r, "\mathrm{id} \otimes \Delta_X"'] & X \otimes (X \otimes X) \arrow[r, "\alpha"'] & (X \otimes X) \otimes X
\end{tikzcd}$$

$$\begin{tikzcd}
X \arrow[r, "\Delta_X"] \arrow[dr, "\mathrm{id}"'] & X \otimes X \arrow[d, "\Delta_X^\dagger"] \\
& X
\end{tikzcd}
\qquad
\begin{tikzcd}
X \otimes X \arrow[rr, "\mathrm{id} \otimes \Delta_X"] \arrow[dr, "\Delta_X^\dagger"] \arrow[dd, "\Delta \otimes \mathrm{id}"] & & X \otimes (X \otimes X) \arrow[dd, "(\Delta_X^\dagger \otimes \mathrm{id}) \circ \alpha"'] \\
& X \arrow[dr, "\Delta_X"'] & \\
(X \otimes X) \otimes X \arrow[rr, "(\mathrm{id} \otimes \Delta_X^\dagger) \circ \alpha^\dagger"'] & & X \otimes X
\end{tikzcd}$$

These diagrams express cocommutativity, coassociativity, speciality and the Frobenius law. That is, each Δ_X is a special dagger Frobenius structure.

For example, **PInj** has inverse products $\Delta_X : X \to X \otimes X$ with $x \mapsto (x, x)$.

Another way to define inverse categories is as certain restriction categories. Informally, a restriction category models partially defined morphisms, by assigning to each $f : A \to B$ a morphism $\bar{f} : A \to A$ that is the identity on the domain of definition of f and undefined otherwise. For more details, see [6].

Definition 4. *A* restriction category *is a category equipped with an operation that assigns to each* $f : A \to B$ *a morphism* $\bar{f} : A \to A$ *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 satisfying* $F(\bar{f}) = \overline{F(f)}$*. A* monoidal restriction category *is a restriction category with a monoidal structure for which* $\otimes : \mathbf{C} \times \mathbf{C} \to \mathbf{C}$ *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* $\mathrm{Inv}(\mathbf{C})$ *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 $\{(x, x) \mid \exists y \in Y : (x, y) \in R\}$.

Remark 1. 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.

From that, in turn, it follows 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

$$(f \otimes g)^\dagger = ((f \otimes \mathrm{id}_Y) \circ (\mathrm{id}_X \otimes g))^\dagger = (\mathrm{id}_X \otimes g)^\dagger \circ (f \otimes \mathrm{id}_Y)^\dagger = f^\dagger \otimes g^\dagger .$$

3 Arrows as an interface for reversible effects

Arrows are a standard way to encapsulate computational side-effects in a functional (irreversible) programming language [17, 18]. 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 5. *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* $A(X,Y)$*.*

There is an operation second(a)*, given by* $\mathrm{arr}(\sigma) \ggg \mathrm{first}(a) \ggg \mathrm{arr}(\sigma)$*, that uses the symmetry and satisfies analogs of* (4)*–*(8)*. Arrows makes sense for (nonsymmetric) monoidal categories if we add this operation and these laws.*

Definition 6. *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 all the above data except for* first*, satisfying all the laws, except those mentioning* first*.*

Remark 2. 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 that (9)–(14) hold for their implementation. Once that is done, the language guarantees arrow inversion.

Remark 3. Now follows a long list of examples of inverse arrows, described in a typed first-order reversible functional pseudocode (akin to Theseus [21, 20]). 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 5 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 1 (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 [23]) on pure functions; not a trivial matter at all.

Example 2 (Information effects). James and Sabry's *information effects* [20] 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 [20] 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:

$$\begin{aligned} \texttt{erase} &= [\pi_H \colon X \times H \to H]_\simeq \in A(X, 1), \\ \texttt{create}_X &= [\pi_H^\dagger \colon H \to X \times H]_\simeq \in A(1, X). \end{aligned}$$

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

Example 3 (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 in the state monad $\texttt{State}\ S\ X = S \Rightarrow (X \otimes S)$, where $- \Rightarrow -$ is the right adjoint to $- \otimes -$, and can be thought of as the object of morphisms from X to Y. Morphisms in its corresponding Kleisli category are morphisms of the form $X \to S \Rightarrow (Y \otimes S)$ in the ambient monoidal closed category. In this formulation, fetching the current state is performed by $\texttt{get} :: \texttt{State}\ S\ S$ defined as $\texttt{get}\ s = (s, s)$, while (destructive) state updating is performed by $\texttt{put} :: S \to \texttt{State}\ 1\ S$ defined as $\texttt{put}\ x\ s = ((), x)$.

Such arrows are tricky for inverse categories, however, as canonical examples (such as **PInj**) fail to be monoidal closed. On the other hand, it follows from monoidal closure that $\hom(X, S \Rightarrow (Y \otimes S)) \simeq \hom(X \otimes S, Y \otimes S)$, 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:

$$
\begin{aligned}
\texttt{RState}\ S\ X\ Y &= X \otimes S \leftrightarrow Y \otimes S \\
\operatorname{arr} f &= f \otimes \mathrm{id}_S \\
a \ggg b &= b \circ a \\
\operatorname{first} a &= (\mathrm{id}_Y \otimes \sigma_{S,Z}) \circ (a \otimes \mathrm{id}_Z) \circ (\mathrm{id}_X \otimes \sigma_{Z,S}) \\
\operatorname{inv} a &= a^\dagger
\end{aligned}
$$

suppressing associators α for readability. This satisfies the inverse arrow laws.

If the monoidal product is an inverse product, we can use the natural diagonal $\Delta_X : X \to X \otimes X$ to access the state by means of the inverse arrow

$$
\begin{aligned}
&\texttt{get} :: \texttt{RState}\ S\ X\ (X \otimes S) \\
&\texttt{get} = \mathrm{id}_X \otimes \Delta_S
\end{aligned}
$$

The inverse to this arrow is $\texttt{assert} :: \texttt{RState}\ S\ (X \otimes S)\ X$, 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:

$$
\begin{aligned}
&\texttt{update} :: (S \leftrightarrow S) \to \texttt{RState}\ S\ X\ X \\
&\texttt{update}\ f = \mathrm{id}_X \otimes f
\end{aligned}
$$

This is analogous to how variable assignment works in the reversible programming language Janus [35]: 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 4 (Concurrency). Another frequently used effect is input and output, allowing programs means of communication with the surrounding environment. The concurrency arrow can be seen as a special case of the reversible state arrow as follows. We construct a formal object E of *environments* and formal partial isomorphisms $E \to E$ corresponding to reversible transformations of this environment (*e.g.*, exchange of data between running processes). We then define

$$\texttt{Con}\ X\ Y = X \otimes E \leftrightarrow Y \otimes E,$$

with arr, first, $\ggg$, and inv as in `RState`, satisfying the inverse arrow laws.

This definition is accurate, but also somewhat unsatisfying operationally. To reify this idea, consider the following reversible input/output protocol. Let E be an object of *process tables*, containing all necessary information about currently running processes (such as internal state). Suppose a (suitably reversible) interface for buffers exists. With this, we can imagine a family of morphisms

$$\texttt{exchange} :: \texttt{Process} \to (\texttt{Con}\ X\ Y)$$

where `Process` is a type of *process handles*, and X and Y are instances of the reversible buffer interface. Operationally, `exchange` exchanges control of the buffer of the currently running process for the one of the process it communicates with, and vice versa. For example, if p_1 calls `exchange` p_2 "`cat`" and p_2 calls `exchange` p_1 "`dog`", after synchronization the buffer received by p_1 will contain "`dog`" while the one received by p_2 will contain "`cat`".

While this simple protocol is reversible, it sidesteps the asymmetry of process communication; one process sends a message, another process waits to receive it. To accommodate this, one could agree that a process expecting to *read* from another process should exchange the empty buffer, while a process that *writes* to another process should expect the empty buffer in return. Thus, asymmetric reading and writing reduce to symmetric buffer exchange. See also the literature on reversible CCS [9] and reversible variations of the π-calculus [8].

Example 5 (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 $T(f^\dagger) = T(f)^\dagger$ 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 [16, 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 [15], giving a good example of capturing an irreversible effect in a reversible setting. For more examples see [16].

Example 6 (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 7 (Control flow). While only trivial inverse categories have coproducts [12], less structure suffices for reversible control structures. Consider a symmetric monoidal inverse category with a zero object as unit. This gives canonical natural *quasi-injections* $X \to X \oplus Y$ and $Y \to X \oplus Y$; we speak of a *disjointness tensor* [12] when these maps are jointly epic.

When the domain and codomain of an inverse arrow have disjointness tensors, it can often be used to implement *ArrowChoice*. For a simple example, the pure arrow on an inverse category with disjointness tensors implements `left` :: $A\ X\ Y \to A\ (X \oplus Z)\ (Y \oplus Z)$ as `left` $f = f \oplus Z$; the laws of *ArrowChoice* [17] 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), 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 [35, 34, 13].

Example 8 (Rewriter). In irreversible computing, another example of an effect is the *writer monad* (useful for *e.g.* logging), which in its arrow form is given by the Kleisli category $\mathcal{K}\ell(- \otimes M)$ for a monoid object M. In the reversible case, we need not just to be able to "write" entries into our log, but also to "unwrite" them again. That is, we need a group object G instead of a monoid M. But that is not enough, as group multiplication $G \otimes G \to G$ is generally not reversible.

Inverse arrows sidestep this issue: given group G in a monoidal restriction category $\mathbf{C}$, the functor $- \otimes G$ is a (strong) restriction monad on $\mathbf{C}$. Now $(X, Y) \mapsto (\mathrm{Inv}(\mathcal{K}\ell(- \otimes G)))(X, Y)$ gives an inverse arrow on $\mathrm{Inv}(\mathbf{C})$.

Morphisms of $\mathrm{Inv}(\mathcal{K}\ell(- \otimes G))$ are morphisms $f\colon X \to Y \otimes G$ of $\mathbf{C}$ for which there exists a unique $g\colon Y \to X \otimes G$ making the following diagram in $\mathbf{C}$ (together with a similar diagram with the roles of f and g interchanged) commute:

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[d, "\bar{f}"'] & Y \otimes G \arrow[r, "g \otimes \mathrm{id}_G"] & X \otimes G \otimes G \arrow[d, "\mathrm{id}_X \otimes \mu"] \\
X \arrow[r, "\rho_X"] & X \otimes I \arrow[r, "\mathrm{id}_X \otimes \eta"] & X \otimes G
\end{tikzcd}$$

Here η and μ are the unit respectively the multiplication of the group object G. For example, when $\mathbf{C}$ is $\mathbf{Pfn}$, and G is an ordinary group written multiplicatively, if f is such a partial isomorphism, then $f(x) = (y, h)$ for some particular $x \in X$ iff its unique inverse g satisfies $g(y) = (x, h^{-1})$.

As with the irreversible writer monad, given a particular element of G, we can write this element to the log by means of the family

$$\begin{aligned} &\texttt{rewrite} :: G \to \texttt{Rewriter}\ X\ X \\ &\texttt{rewrite}\ g\ x = (x, g) \ . \end{aligned}$$

A message g can then be "unwritten" by $\texttt{rewrite}\ g^{-1}$, the partial inverse of $\texttt{rewrite}\ g$. For a toy example, take $G = (\mathbb{Z}, +)$; we may then think of 'writing' 1 to the log as typing a dot, and 'unwriting' 1, or equivalently 'writing' -1, as typing backspace, thus modelling a simple progress bar.

It might be difficult to construct inverses of morphisms in $\mathrm{Inv}(\mathcal{K}\ell(- \otimes G))$ in general: given such an f, we are not aware of a formula that expresses the inverse g in terms of f and operations in $\mathrm{Inv}(\mathbf{C})$. Thus, even though the partial inverse is guaranteed to exist, computing it might be hard. However, when $\mathbf{C}$ is $\mathbf{Pfn}$, this works out by observing that any partial isomorphism f of $\mathcal{K}\ell(- \otimes G)$ factors as a pure arrow followed by an arrow of the form $\langle \mathrm{id}, h \rangle$ for some $h \colon X \to G$ in $\mathbf{Pfn}$, but it is not clear if this is the case for an arbitrary restriction category $\mathbf{C}$.

Example 9 (Recursion). Inverse categories can be outfitted with *joins* on hom sets, giving rise to **DCPO**-enrichment, and in particular to a *fixed point operator*

$$\mathrm{fix}_{X,Y} \colon (\hom(X, Y) \to \hom(X, Y)) \to \hom(X, Y)$$

on continuous functionals [22]. Such joins can be formally adjoined to any inverse category $\mathbf{C}$, yielding an inverse category $J(\mathbf{C})$ with joins and a faithful inclusion functor $I \colon \mathbf{C} \to J(\mathbf{C})$ [14, Sec. 3.1.3].

With this we may obtain an inverse arrow $\hom(I(-), I(+))$ for recursion as an effect. Such an arrow could be useful in a reversible programming language that seeks to guarantee properties like termination and totality for pure functions, as these properties can no longer be guaranteed when general recursion is thrown into the mix. Given such an arrow $\texttt{RFix}\ X\ Y$, one would get a fixed point operator of the form $\texttt{fix} :: (\texttt{RFix}\ X\ Y \to \texttt{RFix}\ X\ Y) \to \texttt{RFix}\ X\ Y$, provided that all expressible functions of type $\texttt{RFix}\ X\ Y \to \texttt{RFix}\ X\ Y$ can be shown to be continuous (e.g., by showing that all such functions must be composed of only continuous things). This operator could then be used to define recursive functions, while maintaining a type-level separation of terminating and potentially non-terminating functions.

The concept of recursion requires no modification to work reversibly, and may even be implemented as usual using a call stack [34]. We illustrate the concept of reversible recursion by two examples: Consider the reversible addition function, mapping a pair of natural numbers (x, y) to the pair $(x, x + y)$. This can be implemented as a recursive reversible function [34]. Since this function returns both the sum and the first component of the input pair (addition on its own is irreversible), it stores in the output the number of times the inverse function must "unrecurse" to get back to the original pair.

Another example is the reversible Fibonacci function, mapping a natural number n to the pair (x_n, x_{n+1}) where each x_i is the i'th number in the Fibonacci series. This may also be implemented as a reversible recursive function. Here, however, the number of times that the inverse function must "unrecurse" is given only implicitly in the output: The inverse iteratively computes $(x_i, x_{i+1}) \mapsto (x_{i+1} - x_i, x_i)$ until the result becomes $(0, 1)$ – the first Fibonacci pair – and then returns the number of iterations it had to perform. If the inverse is given a pair of natural numbers that is not a Fibonacci pair, the result is undefined (i.e., the inverse function may never terminate, or may produce a garbage output).

Example 10 (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 [31, 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 $\mathrm{inv}\, a = a^\dagger$.

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 $[\mathbf{C}^{\mathrm{op}} \times \mathbf{C}, \mathbf{Set}]$ [19]. 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 [19, 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 7. *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]*:*

$$
\begin{tikzcd}
\overline{X}\otimes(\overline{Y}\otimes\overline{Z}) \arrow[r, "\alpha"] \arrow[d, "\mathrm{id}\otimes\chi"'] & (\overline{X}\otimes\overline{Y})\otimes\overline{Z} \arrow[d, "\chi\otimes\mathrm{id}"] \\
\overline{X}\otimes\overline{Z\otimes Y} \arrow[d, "\alpha"'] & \overline{Y\otimes X}\otimes\overline{Z} \arrow[d, "\chi"] \\
\overline{(Z\otimes Y)\otimes X} & \overline{Z\otimes(Y\otimes X)} \arrow[l, "\overline{\alpha}"]
\end{tikzcd}
\qquad
\begin{tikzcd}
\overline{\overline{X}}\otimes\overline{\overline{Y}} \arrow[r, "\chi"] \arrow[d, "\mathrm{id}"'] & \overline{\overline{Y}\otimes\overline{X}} \arrow[d, "\overline{\chi}"] \\
X\otimes Y \arrow[r, "\mathrm{id}"'] & \overline{\overline{X\otimes Y}}
\end{tikzcd}
$$

Just like monoidal categories are the natural setting for monoids, involutive categories are the natural setting for involutive monoids. Any involutive monoidal category has a canonical isomorphism $\phi\colon I \to \overline{I}$ [10, 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 8. *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:*

$$
\begin{tikzcd}
\overline{M} \arrow[r, "\overline{f}"] \arrow[d, "i_M"'] & \overline{N} \arrow[d, "i_N"] \\
M \arrow[r, "f"'] & N
\end{tikzcd}
$$

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 9. *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 1. *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* $\overline{\tau}_{X,Y} = \tau_{Y,X}$*.*

[4] There is a more general definition allowing a natural isomorphism with components $\overline{\overline{X}} \to X$ (see [10] for details), but we only need the strict case.

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:

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

Checking that χ make the relevant diagrams commute is routine.

Theorem 1. *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. 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, relegating some of the less enlightening calculations to the appendix. 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 a 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 10. *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 5. 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 10.

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, as shown in the appendix.

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:

$$\begin{tikzcd} L^+M \times L^+(\hom) \arrow[rr] \arrow[d] & & LM \times L^+(\hom) \arrow[d] \\ L^+(\hom) \times L^+M \arrow[r] & L^+(\hom) \times LM \arrow[r] & LM \end{tikzcd} \tag{15}$$

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, as shown in the appendix. This multiplication is commutative iff the following diagram commutes:

$$\begin{tikzcd} L^+M \times L^+M \arrow[r, "\sigma"] \arrow[dr] & L^+M \times L^+M \arrow[d] \\ & LM \end{tikzcd} \tag{16}$$

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 a calculation included in the appendix shows that the map $D_M \to M$ defined by $(a, a^\dagger, a) \mapsto a \circ a^\dagger \circ a$ is natural.

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

$$\begin{tikzcd} M \arrow[r] \arrow[dr, "\mathrm{id}"'] & D_M \arrow[d] \\ & M \end{tikzcd} \tag{17}$$

Hence we have established the following theorem.

Theorem 2. *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 [21]) 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 [17], 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 [27, 28] 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 subtle but pleasant consequence of the semantics of inverse arrows is that inverse arrows are safe: So long as the inverse arrow laws are satisfied by the implemented arrows, 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. V. Danos and J. Krivine. Reversible communicating systems. In *International Conference on Concurrency Theory*, volume 3170 of *LNCS*, pages 292–307, 2004.
10. J. M. Egger. On involutive monoidal categories. *Theory and Applications of Categories*, 25(14):368–393, 2011.
11. M. J. Gabbay and P. H. Kropholler. Imaginary groups: lazy monoids and reversible computation. *Mathematical Structures in Computer Science*, 23(5):1002–10031, 2013.
12. B. G. Giles. *An investigation of some theoretical aspects of reversible computing.* PhD thesis, University of Calgary, 2014.
13. R. Glück and R. Kaarsgaard. A categorical foundations for structured reversible flowchart languages. In *Mathematical Foundations of Program Semantics XXXIII, Proceedings*, 2017. to appear.
14. X. Guo. *Products, Joins, Meets, and Ranges in Restriction Categories*. PhD thesis, University of Calgary, 2012.
15. 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.
16. C. Heunen and M. Karvonen. Monads on dagger categories. *Theory and Applications of Categories*, 31(35):1016–1043, 2016.
17. J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, 2000.
18. J. Hughes. *Advanced Functional Programming*, volume 3622 of *Lecture Notes in Computer Science*, chapter Programming with Arrows, pages 73–129. Springer, 2005.
19. B. Jacobs, C. Heunen, and I. Hasuo. Categorical semantics for arrows. *Journal of Functional Programming*, 19(3-4):403–438, 2009.
20. R. P. James and A. Sabry. Information effects. In *Principles of Programming Languages*, pages 73–84. ACM, 2012.
21. 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.

22. R. Kaarsgaard, H. B. Axelsen, and R. Glück. Join inverse categories and reversible recursion. *Journal of Logical and Algebraic Methods in Programming*, 87, 2017.
23. 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.
24. 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.
25. E. Moggi. Computational lambda-calculus and monads. *Logic in Computer Science*, 1989.
26. K. Morita. Two-way reversible multihead automata. *Fundamenta Informaticae*, 110(1–4):241–254, 2011.
27. R. Paterson. A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240, 2001.
28. R. Paterson. Arrows and computation. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, pages 201–222. Palgrave, 2003.
29. 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.
30. U. P. Schultz, M. Bordignon, and K. Støy. Robust and reversible execution of self-reconfiguration sequences. *Robotica*, 29(1):35–37, 2011.
31. 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.
32. 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.
33. T. Toffoli. Reversible computing. In *International Colloquium on Automata, Languages and Programming*, pages 632–644, 1980.
34. 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.
35. 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.

Appendix

This appendix contains calculations omitted from the proof of Theorem 2.

For an involutive monoid M, the map $LM \times L^+(\mathrm{hom}_{\mathbf{C}}) \to LM$ defined by $(a, g^\dagger \circ g) \mapsto a \circ g^\dagger \circ g$ is natural:

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

If an involutive monoid M satisfies (15), then 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:

$$
\begin{aligned}
& L^+M \times L^+M(f, \mathrm{id})(a^\dagger \circ a, b^\dagger \circ b) && \\
& = (f^\dagger \circ a^\dagger \circ a \circ f, f^\dagger \circ b^\dagger \circ b \circ f) && \\
& \mapsto f^\dagger \circ a^\dagger \circ a \circ f \circ f^\dagger \circ b^\dagger \circ b \circ f && \\
& = f \circ a^\dagger \circ a \circ b^\dagger \circ b \circ f \circ f^\dagger \circ f && \text{by (15)} \\
& = f \circ a^\dagger \circ a \circ b^\dagger \circ b \circ f && \text{because } \mathbf{C} \text{ is an inverse category} \\
& = L^+M(f, \mathrm{id})(a^\dagger \circ a \circ b^\dagger \circ b) &&
\end{aligned}
$$

If an involutive monoid 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:

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

RFun Revisited

Robin Kaarsgaard and Michael Kirkedal Thomsen

DIKU, Department of Computer Science, University of Copenhagen
{robin, m.kirkedal}@di.ku.dk

We describe here the steps taken in the further development of the reversible functional programming language RFun. Originally, RFun was designed as a first-order untyped language that could only manipulate constructor terms; later it was also extended with restricted support for function pointers [6, 5]. We outline some of the significant updates ot the language, including a static type system based on relevant typing, with special support for ancilla (read-only) variables added through an unrestricted fragment. This has further resulted in a complete makeover of the syntax, moving towards a more modern, Haskell-like language.

Background In the study of reversible computation, one investigates computational models in which individual computation steps can be uniquely and unambiguously inverted. For programming languages, this means languages in which programs can be run *backward* and get a unique result (the exact input).

Though the field is often motivated by a desire for energy and entropy preservation though the work of Landauer [3], we are more interested in the possibility to use reversibility as a property that can aid in the execution of a system; an approach which can be credited to Huffman [1]. In this paper we specifically consider RFun. Another notable example of a reversible functional language is Theseus [2], which has also served as a source of inspiration for some of the developments described here.

Ancillae Ancillae (considered ancillary variables in this context) is a term adopted from physics to describe a state in which entropy is unchanged. Here we specifically use it for variables for which we can guarantee that their values are unchanged over a function call. We cannot put too little emphasis on the guarantee, because we have taken a conservative approach and will only use it when we statically can ensure that it is upheld.

1 RFun version 2

In this section, we will describe the most interesting new additions to RFun and how they differ from the original work. Rather than showing the full formalisation, we will instead argue for their benefits to a reversible (functional) language.

Figure 1 shows an implementation of the Fibonacci function in RFun, which we will use as a running example. Since the Fibonacci function is not injective (the first and second Fibonacci numbers are both 1), we instead compute *Fibonacci pairs*, which *are* unique. Hence, the first Fibonacci pair is $(0, 1)$, the second to $(1, 1)$, third $(2, 1)$, and so forth.

The implementation in RFun can be found in Figure 1 and consists of a type definition `Nat` and two functions `plus` and `fib`. Here, `Nat` defines the natural numbers as Peano numbers, `plus` implements addition over the defined natural numbers, while `fib` is the implementation of the Fibonacci pairfunction. Further, Figure 2 shows an implementation of the map function.

```
data Nat = Z | S Nat

plus :: Nat → Nat ↔ Nat
plus Z     x = x
plus (S y) x =
  let x' = plus y x
   in (S x')

fib :: Nat ↔ (Nat, Nat)
fib Z     = ((S Z), Z)
fib (S m) =
  let (x, y) = fib m
       y'    = plus x y
  in  (y', x)
```

Figure 1: RFun program computing Fibonacci pairs.

```
map :: (a ↔ b) → [a] ↔ [b]
map fun [ ]  = [ ]
map fun (l:ls) =
  let l'  = fun l
      ls' = map fun ls
  in  (l': ls')
```

Figure 2: Map function in RFun.

1.1 Type system

With Milner's motto that "well-typed programs cannot go wrong," type systems have proven immensely successful in guaranteeing fundamental well-behavedness properties of programs. In reversible functional programming, linear type systems (see, *e.g.*, [2]) have played an important role in ensuring reversibility.

Fundamentally, a reversible computation can be considered as an injective transformation of a state into an updated state. In this view, it seems obvious to let the type system guarantee linearity, *i.e.*, that each available resource (in this case, variable) is used exactly once. Though linearity is not enough to guarantee reversibility, it enables the type system to statically reject certain irreversible operations (*e.g.*, projections). However, linearity is also more restrictive than needed: if we accept that functions may be partial (a necessity for r-Turing completeness), first-order data *can* be duplicated reversibly. For this reason, we may relax the linearity constraint to *relevance*, *i.e.*, that all available variables must be used *at least* once. This guarantees that values are never lost, while also enabling implicit duplication of values.

A useful concept in reversible programming is access to ancillae, *i.e.*, values that remain unchanged across function calls. Such values are often used as a means to guarantee reversibility in a straightforward manner. For example, in Figure 1, the first input variable of the `plus` function is ancillary; it's value is tacitly returned automatically as part of the output.

To support such ancillary variables at the type level, a type system inspired by Polakow's combined reasoning system of ordered, linear, and unrestricted intuitionistic logic [4] is used. The type system splits the typing contexts into two parts: a static one (containing ancillary variables and other static parts of the environment), and a dynamic one (containing variables not considered ancillary). This gives a typing judgment of $\Sigma; \Gamma \vdash e : \tau$, where Σ is the static context, and Γ the dynamic one.

Whereas we must ensure that variables in the dynamic context Γ are used in a relevant manner to guarantee reversibility, there are no restrictions on the use of variables in the static context – these can used as many or as few times (including not at all) as desired. To distinguish between ancillary and dynamic variables at the type level, two different arrow types are used: $t_1 \rightarrow t_2$ denotes that the input variable is ancillary, whereas $t_1 \leftrightarrow t_2$ denotes that it is dynamic. As such, the type of `plus` in Figure 1 signifies that the first input variable is ancillary, and the second is dynamic.

A neat use of ancillae is to provide limited support for behaviour similar to higher-order functions. For example, the usual `map` function is not reversible, as not every function of type `[a] ↔ [b]` arises as a functorial application of some function of type `a ↔ b`. However, if we consider the input function `f :: a ↔ b` to be ancillary, one can straightforwardly define a reversible `map` function (see Figure 2) as one of type `(a ↔ b) → ([a] ↔ [b])`. In this way, ancillae can be considered as a slight generalization of the *parametrized maps* found in Theseus [2].

1.2 Duplication/Equality

In the first version of RFun, duplication and equality was included as a special operator, which could perform deep copying or uncopying reversibly, depending on the usage. However, we have found that understanding the semantics this operator often poses a problem for programmers.

To remedy this, we propose to use type classes, and implement equality instead using a type class similar to the `EQ` type class found in Haskell. As in Haskell, the functions needed to be member of this class can often be automatically derived.

1.3 First-match policy

The first-match policy (FMP) is essential to ensuring injectivity of individual functions. It states that a returned value of a function *must not* match any previous leaf of the function, and can be compared to checking the validity of an assertion on exit.

I the first version of RFun, the check to ensure that the first-match policy was upheld was always performed at run-time, and, thus, posed a limitation to the performance. However, with the type system, it will now often be possible to perform this check statically, as the types of the leaves or even the ancillae inputs can be orthogonal. *E.g.* in the `plus` function (in Figure 1), the use of ancillae input ensures that the FMP is always upheld, while `map` (in Figure 2) is reversible by orthogonality of the leaves. Unfortunately, this cannot always guaranteed statically, and the `fib` function (in Figure 1) is an example where a runtime check is still required.

1.4 Conclusion

In this paper we have outlined the future development of the reversible function language RFun. A central element of this is the development of the type system. The work shows both an interesting new application of relevant type systems, and gives RFun a more modern design that will make it easier for programmers to understand.

Acknowledgements This work was partly supported by the European COST Action IC 1405: Reversible Computation - Extending Horizons of Computing.

References

[1] D. A. Huffman. Canonical forms for information-lossless finite-state logical machines. *IRE Transactions on Information Theory*, 5(5):41–59, 1959.

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

[3] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5(3):183–191, 1961.

[4] J. Polakow. *Ordered Linear Logic and Applications.* PhD thesis, Carnegie Mellon University, 2001.

[5] M. K. Thomsen and H. B. Axelsen. Interpretation and programming of the reversible functional language. In *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages*, IFL '15, pages 8:1–8:13. ACM, 2016.

[6] T. Yokoyama, H. B. Axelsen, and R. Glück. Towards a reversible functional language. In A. De Vos and R. Wille, editors, *Reversible Computation, RC '11*, volume 7165 of *LNCS*, pages 14–29. Springer-Verlag, 2012.