A Computational Interpretation of Compact Closed Categories: Reversible Programming with Negative and Fractional Types

CHAO-HONG CHEN, Indiana University, USA
AMR SABRY, Indiana University, USA

Compact closed categories include objects representing higher-order functions and are well-established as models of linear logic, concurrency, and quantum computing. We show that it is possible to construct such compact closed categories for conventional sum and product types by defining a dual to sum types, a negative type, and a dual to product types, a fractional type. Inspired by the categorical semantics, we define a sound operational semantics for negative and fractional types in which a negative type represents a computational effect that "reverses execution flow" and a fractional type represents a computational effect that "garbage collects" particular values or throws exceptions.
Specifically, we extend a first-order reversible language of type isomorphisms with negative and fractional types, specify an operational semantics for each extension, and prove that each extension forms a compact closed category. We furthermore show that both operational semantics can be merged using the standard combination of backtracking and exceptions resulting in a smooth interoperability of negative and fractional types. We illustrate the expressiveness of this combination by writing a reversible SAT solver that uses backtracking search along freshly allocated and de-allocated locations. The operational semantics, most of its meta-theoretic properties, and all examples are formalized in a supplementary Agda package.

CCS Concepts: • **Theory of computation → Type theory**; **Abstract machines**; **Operational semantics**.

Additional Key Words and Phrases: Abstract Machines, Duality of Computation, Higher-Order Reversible Programming, Termination Proofs, Type Isomorphisms

ACM Reference Format:
Chao-Hong Chen and Amr Sabry. 2021. A Computational Interpretation of Compact Closed Categories: Reversible Programming with Negative and Fractional Types. *Proc. ACM Program. Lang.* 5, POPL, Article 9 (January 2021), 29 pages. https://doi.org/10.1145/3434290

1 INTRODUCTION

Compact closed categories [Kelly and Laplaza 1980; Kelly 1972] have found numerous applications in mathematics, physics, and computer science. Examples of compact closed categories include models of multiplicative linear logic [Abramsky and Jagadeesan 1994], models of concurrency [Abramsky et al. 1996], and models of quantum computing [Abramsky and Coecke 2004; Fiore 2015].
A compact closed category is a symmetric monoidal category [Bénabou 1963, 1964; MacLane 1963] in which every object has a dual. As a concrete example, the categorification of the monoid of natural numbers under addition $(\mathbb{N}, +, 0)$ is a symmetric monoidal category. One way to make

Authors' addresses: Chao-Hong Chen, Indiana University, Bloomington, Indiana, USA, chen464@indiana.edu; Amr Sabry, Indiana University, Bloomington, Indiana, USA, sabry@indiana.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2021 Copyright held by the owner/author(s).
2475-1421/2021/1-ART9
https://doi.org/10.1145/3434290

this category into a compact closed one is to add objects corresponding to negative numbers. In the resulting category, the following algebraic proof of $a = a$ would be represented in the standard notation of string diagrams as the "zigzag" or "snake" dotted line below. The coherence conditions of compact closed categories require this morphism to be equal to the identity morphism thus matching the categorical semantics with the underlying algebraic proof of $a = a$:

$$
\begin{array}{lllll}
 & a & & & \text{initial } a & (0) \\
= & a & + & 0 & \text{0 is identity for } + & (1) \\
= & a & + & (-a \quad + \quad a) & -a \text{ is the additive inverse of } a & (2) \\
= & (a & + & (-a)) \quad + \quad a & + \text{ is associative} & (3) \\
= & & 0 & \quad + \quad a & -a \text{ is the additive inverse of } a & (4) \\
= & & & a & \text{0 is identity for } + & (5)
\end{array}
$$

The main problem we solve is the computational interpretation of such morphisms. The shape of the diagram suggests that it would correspond to a program involving a form of backtracking but making this precise is quite subtle. To appreciate the difficulties, consider replacing the additive monoid $(\mathbb{N}, +, 0)$ by the multiplicative one $(\mathbb{N}, *, 1)$ and modifying the diagram *mutatis mutandis*. The same intuition suggests that some form of backtracking is also present in the multiplicative case but clearly it must be distinguished from the one present in the additive version.

Our main contribution is a formalization of these two notions of backtracking: we show that *negative types* can be interpreted as "reversing time" (i.e., reversing the control flow) and that *fractional types* can be interpreted as "reversing space" (i.e., reclaiming storage holding particular values or throwing exceptions). The formalization takes the form of an operational semantics satisfying the coherence conditions for compact closed categories. The operational semantics and most of its properties are formalized in a supplementary Agda package.

Our secondary, but quite significant, contribution is to show that these dualities have interesting applications in programming. Specifically, starting from a first-order reversible language in which all programs witness type isomorphisms between finite types, we show that the extension with negative and fractional types enriches it in several dimensions. First as an immediate consequence of forming compact closed categories, each duality defines a corresponding notion of higher-order functions. In the additive case, the type $-A + B$ internalizes functions as terms that convert a *demand* of a value of type A (flowing backwards) to the *production* of a value of type B (flowing forwards). In the multiplicative case, the type $\mathbb{1}/v \times B$ for some $v : A$ internalizes the action of a function on a particular argument v that trades the space used by v for the space used by the result of type B. Second, we suggest with several examples how negative and fractional types put the management and optimization of space and time resources under programmer control. Finally, we illustrate, via a full implementation of a reversible SAT solver, that negative and fractional types can be used to implement, in a reversible programming language, advanced control abstractions such as *backtracking search*.

Along the way, our third major contribution is a technique for proving the termination of a large class of reversible abstract machines. Remarkably, under mild conditions, no state ever repeats during the execution of this class of reversible abstract machines. The fully formalized Agda proof of this statement reduces the termination property of an abstract machine to the much simpler question of whether the set of reachable states from a start state is finite.

The paper is accompanied by approximately 5000 lines of Agda code implementing all the abstract machines, interpreters, and examples. [1] The code additionally includes proofs of all the theorems except for two combinatorial proofs in Sec. 5 and Sec. 7. The two proofs are tedious to formalize but are conceptually straightforward asserting that, for the abstract machines in question, the set

[1] https://doi.org/10.5281/zenodo.4070034

of states reachable from a start state is finite. This finiteness property is used in just two theorems (Thm. 19 and Thm.42) which are the only two theorems that are only partially formalized.

Related Work. Our work builds on extensive research in at least three areas:

- Compact closed categories and their applications. Excellent entry points to the most relevant material are Selinger's survey [2011] and Heunen and Vicary's book [2019]. Our work is however not category-theoretic in nature but is rather building bridges. First, we formalize connections between dual objects in category theory and the duality of values and computations in programming languages [Curien and Herbelin 2000; Filinski 1989; Selinger 2001; Wadler 2003, 2005]. Second, following a long tradition [Aman et al. 2020; Glück and Kaarsgaard 2018; James and Sabry 2012a,b, 2014; James et al. 2013], we design and implement reversible programming languages that match the categorical models.
- Type Theory and Logic of Continuations. The idea of "negative types" and their connection to continuations has appeared many times in the literature. Rauszer [1974a; 1974b; 1980] introduced a logic which contains a dual to implication. Her work has been distilled in the form of *subtractive logic* [Crolard 2001] which has been related to coroutines [Crolard 2004] and delimited continuations [Ariola et al. 2009]. Filinski [1992] uses the negative types of linear logic to model continuations. Reddy [1991] generalizes this idea by interpreting the negative types of linear logic as *acceptors*, which are like continuations in the sense that they take an input and return no output. Acceptors however are also similar in flavor to logic variables: they can be created and instantiated later once their context of use is determined. Our flavor of negative types is arguably simpler exactly matching the algebraic notion of negative numbers and the computational interpretations of linear logic [Abramsky et al. 2002; Abramsky and Jagadeesan 1994; Girard 1989; Mackie 1995, 2011].
- Type Isomorphisms. In the finitary setting, type isomorphisms provide a perfect, sound and complete, foundation for reversible programming languages [Fiore 2004; Fiore et al. 2006; James and Sabry 2012a]. This simple model can be extended by relaxing the isomorphisms to be partial — giving models for Turing-complete reversible languages [Bowman et al. 2011; James and Sabry 2012a; Kaarsgaard and Veltri 2019] and by incorporating reversible effects such as state [Heunen et al. 2018; Heunen and Karvonen 2015]. The simple model of type isomorphisms was also recently extended by Chen et al. [2020] with the same fractional types we use in this paper. That extension investigated several semantics of fractional types and their applications to the safe management of ancilla bits (which are bits used for temporary storage). Our semantics for fractional types matches one of theirs but instead of focusing on extracting safe programs, we focus on formalizing the meta-theoretic properties of the semantics including the fact that it forms a compact closed category. Our work also shares motivations, examples, and some constructions with widely-circulated unpublished papers on "negative and fractional types" [James and Sabry 2012b; James et al. 2013]. Regarding the technical details and contributions, however, our work differs from, and significantly improves upon them: (i) the unpublished semantics of fractional types assumed logical variables and unification while we have a different semantics that makes no such assumptions; (ii) the unpublished semantics of negative types uses the same intuition of reversing execution flow but was only presented using an abstract machine and without proving any properties in contrast to our semantics which is specified using both an abstract machine and an equivalent interpreter with exceptions and handlers; and (iii) our semantics of both negative and fractional types are formalized with termination proofs, reversibility proofs, and proofs that the constructions form compact closed categories.

$$
\begin{array}{rrcll}
\mathsf{id}\!\leftrightarrow : & A & \leftrightarrow & A & : \mathsf{id}\!\leftrightarrow \\
\mathsf{unite}_+\mathsf{l} : & \mathbb{0} + A & \leftrightarrow & A & : \mathsf{uniti}_+\mathsf{l} \\
\mathsf{swap}_+ : & A + B & \leftrightarrow & B + A & : \mathsf{swap}_+ \\
\mathsf{assocl}_+ : & A + (B + C) & \leftrightarrow & (A + B) + C & : \mathsf{assocr}_+ \\
\mathsf{unite}_*\mathsf{l} : & \mathbb{1} \times A & \leftrightarrow & A & : \mathsf{uniti}_*\mathsf{l} \\
\mathsf{swap}_* : & A \times B & \leftrightarrow & B \times A & : \mathsf{swap}_* \\
\mathsf{assocl}_* : & A \times (B \times C) & \leftrightarrow & (A \times B) \times C & : \mathsf{assocr}_* \\
\mathsf{absorbr} : & \mathbb{0} \times A & \leftrightarrow & \mathbb{0} & : \mathsf{factorzl} \\
\mathsf{dist} : & (A + B) \times C & \leftrightarrow & (A \times C) + (B \times C) & : \mathsf{factor}
\end{array}
$$

$$
\frac{\vdash c_1 : A \leftrightarrow B \quad \vdash c_2 : B \leftrightarrow C}{\vdash c_1 \fatsemi c_2 : A \leftrightarrow C}
\qquad
\frac{\vdash c_1 : A \leftrightarrow B \quad \vdash c_2 : C \leftrightarrow D}{\vdash c_1 \oplus c_2 : A + C \leftrightarrow B + D}
\qquad
\frac{\vdash c_1 : A \leftrightarrow B \quad \vdash c_2 : C \leftrightarrow D}{\vdash c_1 \otimes c_2 : A \times C \leftrightarrow B \times D}
$$

Fig. 1. Π-terms, combinators, and their types.

Outline. Sec. 2 introduces a first-order reversible language Π that is universal for combinational circuits. In Sec. 3 we state and prove the "no-repeating" lemma allowing us to conveniently prove the termination of all our abstract machines. In Sec. 4 we explain how the additive structure of the language models logical time, how the multiplicative structure of the language models space resources, and how the language allows arbitrary trade-offs between these two resources. In Sec. 5 and Sec. 6 we formalize the additive and multiplicative duals separately defining Π^m (for Π-types extended with minus) and Π^d (for Π-types extended with division) and combine them in Sec. 7 into a unified language $\Pi^{\mathbb{Q}}$ (for Π-types having the structure of the field of rational numbers). Sec. 8 collects a number of small examples and outlines the full implementation of our reversible SAT solver. In the last section, we conclude and outline future directions of research.

2 CORE REVERSIBLE LANGUAGE: Π

The syntax of the language Π consists of several sorts:

Value types	A, B, C, D	::=	$\mathbb{0} \mid \mathbb{1} \mid A + B \mid A \times B$
Values	v, w, x, y	::=	$\mathsf{tt} \mid \mathsf{inj}_1\ v \mid \mathsf{inj}_2\ v \mid (v, w)$
Program types			$A \leftrightarrow B$
Programs	c	::=	(See Fig. 1)

Focusing on finite types, the building blocks of type theory are: the empty type ($\mathbb{0}$), the unit type ($\mathbb{1}$) containing just one value tt, the sum type (+) containing values of the form $\mathsf{inj}_1\ v$ and $\mathsf{inj}_2\ v$, and the product (×) type containing pairs of values (v_1, v_2). One may view each type A as a collection of physical wires that can transmit $|A|$ distinct values where $|A|$ is a natural number that indicates the size of a type, computed as: $|\mathbb{0}| = 0$; $|\mathbb{1}| = 1$; $|A + B| = |A| + |B|$; and $|A \times B| = |A| * |B|$. Thus the type $\mathbb{B} = \mathbb{1} + \mathbb{1}$ corresponds to a wire that can transmit one of two values, i.e., bits, with the convention that $\mathsf{inj}_1\ \mathsf{tt}$ represents $\mathbb{F}$ and $\mathsf{inj}_2\ \mathsf{tt}$ represents $\mathbb{T}$. The type $\mathbb{B} \times \mathbb{B} \times \mathbb{B}$ corresponds to a collection of wires that can transmit three bits. From that perspective, a type isomorphism between types A and B (such that $|A| = |B| = n$) models a *reversible* combinational circuit that *permutes* the n different values. These type isomorphisms are collected in Fig. 1.
Each line in the top part of the figure introduces a pair of dual constants that witness the type isomorphism in the middle. These are the *base* (non-reducible) terms of Π. Note how the above has two readings: first as a set of typing relations for a set of constants. Second, if these axioms are seen as universally quantified, orientable statements, they also induce transformations of values.

$$
\begin{array}{rcl@{\qquad}rcl}
\delta(\mathsf{unite}_+\mathsf{l},\ \mathsf{inj}_2\ v) & = & v & \delta(\mathsf{uniti}_+\mathsf{l},\ v) & = & \mathsf{inj}_2\ v \\
\delta(\mathsf{swap}_+,\ \mathsf{inj}_1\ v) & = & \mathsf{inj}_2\ v & & & \\
\delta(\mathsf{swap}_+,\ \mathsf{inj}_2\ v) & = & \mathsf{inj}_1\ v & & & \\
\delta(\mathsf{assocl}_+,\ \mathsf{inj}_1\ v) & = & \mathsf{inj}_1\ (\mathsf{inj}_1\ v) & \delta(\mathsf{assocr}_+,\ \mathsf{inj}_1\ (\mathsf{inj}_1\ v)) & = & \mathsf{inj}_1\ v \\
\delta(\mathsf{assocl}_+,\ \mathsf{inj}_2\ (\mathsf{inj}_1\ v)) & = & \mathsf{inj}_1\ (\mathsf{inj}_2\ v) & \delta(\mathsf{assocr}_+,\ \mathsf{inj}_1\ (\mathsf{inj}_2\ v)) & = & \mathsf{inj}_2\ (\mathsf{inj}_1\ v) \\
\delta(\mathsf{assocl}_+,\ \mathsf{inj}_2\ (\mathsf{inj}_2\ v)) & = & \mathsf{inj}_2\ v & \delta(\mathsf{assocr}_+,\ \mathsf{inj}_2\ v) & = & \mathsf{inj}_2\ (\mathsf{inj}_2\ v) \\
\delta(\mathsf{unite}_*\mathsf{l},\ (\mathsf{tt}, v)) & = & v & \delta(\mathsf{uniti}_*\mathsf{l},\ v) & = & (\mathsf{tt}, v) \\
\delta(\mathsf{swap}_*,\ (x, y)) & = & (y, x) & & & \\
\delta(\mathsf{assocl}_*,\ (x, (y, z))) & = & ((x, y), z) & \delta(\mathsf{assocr}_*,\ ((x, y), z)) & = & (x, (y, z)) \\
\delta(\mathsf{dist},\ (\mathsf{inj}_1\ x, z)) & = & \mathsf{inj}_1\ (x, z) & \delta(\mathsf{factor},\ \mathsf{inj}_1\ (x, z)) & = & (\mathsf{inj}_1\ x, z) \\
\delta(\mathsf{dist},\ (\mathsf{inj}_2\ y, z)) & = & \mathsf{inj}_2\ (y, z) & \delta(\mathsf{factor},\ \mathsf{inj}_2\ (y, z)) & = & (\mathsf{inj}_2\ y, z)
\end{array}
$$

Fig. 2. Semantics of base combinators

$$
\frac{}{\square : \textsc{cont}_{A \leftrightarrow B}}
\qquad
\frac{c_2 : B \leftrightarrow C \quad \kappa : \textsc{cont}_{A \leftrightarrow C}}{(\square \fatsemi c_2) \bullet \kappa : \textsc{cont}_{A \leftrightarrow B}}
\qquad
\frac{c_1 : A \leftrightarrow B \quad \kappa : \textsc{cont}_{A \leftrightarrow C}}{(c_1 \fatsemi \square) \bullet \kappa : \textsc{cont}_{B \leftrightarrow C}}
$$

$$
\frac{c_2 : C \leftrightarrow D \quad \kappa : \textsc{cont}_{A+C \leftrightarrow B+D}}{(\square \oplus c_2) \bullet \kappa : \textsc{cont}_{A \leftrightarrow B}}
\qquad
\frac{c_1 : A \leftrightarrow B \quad \kappa : \textsc{cont}_{A+C \leftrightarrow B+D}}{(c_1 \oplus \square) \bullet \kappa : \textsc{cont}_{C \leftrightarrow D}}
$$

$$
\frac{c_2 : C \leftrightarrow D \quad y : C \quad \kappa : \textsc{cont}_{A \times C \leftrightarrow B \times D}}{(\square \otimes [c_2, y]) \bullet \kappa : \textsc{cont}_{A \leftrightarrow B}}
\qquad
\frac{c_1 : A \leftrightarrow B \quad x : B \quad \kappa : \textsc{cont}_{A \times C \leftrightarrow B \times D}}{([c_1, x] \otimes \square) \bullet \kappa : \textsc{cont}_{C \leftrightarrow D}}
$$

Fig. 3. Well-formed Continuation Stacks

The intuition here is that these axioms have computational content because they witness isomorphisms rather than merely stating an extensional equality. The isomorphisms are extended to form a congruence relation by adding the three *congruence* constructors at the bottom of the figure that witness equivalence and compatible closure.

Although austere, this combinator-based language has the advantage of being amenable to formal analysis for at least two reasons: (i) it is conceptually simple and small, and (ii) it has direct and evident connections to type theory and category theory and hence possesses a rich algebraic structure which we exploit in the remainder of the paper. Specifically, the type isomorphisms of Π are sound and complete for all permutations on finite types [Fiore 2004; Fiore et al. 2006] and hence they are *complete* for expressing reversible combinational circuits [Fredkin and Toffoli 1982; James and Sabry 2012a; Toffoli 1980]. Algebraically, these types and combinators form a *commutative semiring* (up to type isomorphism). Logically, they form a superstructural logic capturing space-time trade-offs [Sparks and Sabry 2014]. Categorically, they form a *distributive bimonoidal category* [Laplaza 1972].

2.1 Abstract Machine Semantics

To formalize the semantics of Π, we first specify the semantics of the base combinators in Fig. 2 using a function δ. It is clear that this function has an inverse $\delta^\dagger$ such that $\delta^\dagger(c, \delta(c, v)) = v$. To specify the semantics of full programs, we use an abstract machine consisting of three registers: a code register containing a combinator c, a value register containing a value v, and a continuation register containing a continuation κ. Continuations are lists of frames $F_1 \bullet F_2 \bullet \cdots \bullet F_j \bullet \square$ where each frame has a "hole" representing a missing combinator and where holes are filled according to the order $F_j[\cdots[F_2[F_1]]\cdots]$. In other words, the missing combinator is used to fill the hole in F_1 and the result is used to fill the hole in F_2 and so on. Fig. 3 gives the definition of well-formed continuations where $\kappa : \textsc{cont}_{A \leftrightarrow B}$ represents an evaluation context with a hole of type $A \leftrightarrow B$.

$$
\begin{array}{rcl}
\langle c \mid v \mid \kappa \rangle & \mapsto_1 & [c \mid \delta(c, v) \mid \kappa] \quad \text{for base combinators } c \\
\langle \mathsf{id}\!\leftrightarrow \mid v \mid \kappa \rangle & \mapsto_2 & [\mathsf{id}\!\leftrightarrow \mid v \mid \kappa] \\
\langle c_1 \fatsemi c_2 \mid v \mid \kappa \rangle & \mapsto_3 & \langle c_1 \mid v \mid (\Box \fatsemi c_2) \bullet \kappa \rangle \\
\langle c_1 \oplus c_2 \mid \mathsf{inj}_1\ x \mid \kappa \rangle & \mapsto_4 & \langle c_1 \mid x \mid (\Box \oplus c_2) \bullet \kappa \rangle \\
\langle c_1 \oplus c_2 \mid \mathsf{inj}_2\ y \mid \kappa \rangle & \mapsto_5 & \langle c_2 \mid y \mid (c_1 \oplus \Box) \bullet \kappa \rangle \\
\langle c_1 \otimes c_2 \mid (x, y) \mid \kappa \rangle & \mapsto_6 & \langle c_1 \mid x \mid (\Box \otimes [c_2, y]) \bullet \kappa \rangle \\
\\
[c_1 \mid v \mid (\Box \fatsemi c_2) \bullet \kappa] & \mapsto_7 & \langle c_2 \mid v \mid (c_1 \fatsemi \Box) \bullet \kappa \rangle \\
[c_1 \mid x \mid (\Box \otimes [c_2, y]) \bullet \kappa] & \mapsto_8 & \langle c_2 \mid y \mid ([c_1, x] \otimes \Box) \bullet \kappa \rangle \\
[c_2 \mid y \mid ([c_1, x] \otimes \Box) \bullet \kappa] & \mapsto_9 & [c_1 \otimes c_2 \mid (x, y) \mid \kappa] \\
[c_2 \mid v \mid (c_1 \fatsemi \Box) \bullet \kappa] & \mapsto_{10} & [c_1 \fatsemi c_2 \mid v \mid \kappa] \\
[c_1 \mid x \mid (\Box \oplus c_2) \bullet \kappa] & \mapsto_{11} & [c_1 \oplus c_2 \mid \mathsf{inj}_1\ x \mid \kappa] \\
[c_2 \mid y \mid (c_1 \oplus \Box) \bullet \kappa] & \mapsto_{12} & [c_1 \oplus c_2 \mid \mathsf{inj}_2\ y \mid \kappa]
\end{array}
$$

Fig. 4. Π-abstract machine

There are two kinds of machine states $\langle c \mid v \mid \kappa \rangle$ and $[c \mid v \mid \kappa]$. The first is an "enter" state where the focus is on the combinator c and the second is a "return" state where the focus is on the continuation κ. In more detail, in a state $\langle c \mid v \mid \kappa \rangle$ evaluation is about to apply c to v; in a state $[c \mid v \mid \kappa]$ evaluation has just finished applying c resulting in v which is ready to be consumed by the continuation. These distinctions are made precise in the following definition.

Definition 1 (Π-machine states). *A Π-machine state σ is either:*

- *An enter state: $\langle c \mid v \mid \kappa \rangle$ where $c : A \leftrightarrow B$, $v : A$, and $\kappa : \textsc{cont}_{A \leftrightarrow B}$.*
- *A return state: $[c \mid v \mid \kappa]$ where $c : A \leftrightarrow B$, $v : B$, and $\kappa : \textsc{cont}_{A \leftrightarrow B}$.*

Fig. 4 gives the transition rules of the abstract machine. The first group defines transition steps starting from enter states: these evaluation steps either immediately return if the computation is trivial (cases $\mapsto_1$ and $\mapsto_2$) or push a frame onto the continuation stack to focus on the sub-combinator in evaluation position. The second group defines the transition steps from return states. In the first rule $\mapsto_7$, evaluation of c_1 has just finished; as the first frame on the continuation stack is $(\Box \fatsemi c_2)$, evaluation proceeds by entering c_2. Rule $\mapsto_{10}$ shows what happens when the evaluation of c_2 terminates. In that case, the evaluation of the sequence $c_1 \fatsemi c_2$ is complete and v is ready to be consumed by κ.
The machine transition relation is both forward and backward deterministic.

Lemma 2 (Π-forward deterministic). *If $\sigma \mapsto \sigma_1$ and $\sigma \mapsto \sigma_2$ then $\sigma_1 = \sigma_2$*

Lemma 3 (Π-backward deterministic). *If $\sigma_1 \mapsto \sigma$ and $\sigma_2 \mapsto \sigma$ then $\sigma_1 = \sigma_2$*

Definition 4 (Π-forward evaluation). *We say $\mathrm{eval}(c, v_1) = v_2$ if $\langle c \mid v_1 \mid \Box \rangle \mapsto^* [c \mid v_2 \mid \Box]$ where $\mapsto^*$ is the reflexive transitive closure of the machine transition relation.*

Definition 5 (Π-backward evaluation). *We define the relation $\mapsto^\dagger$ such that $\sigma_1 \mapsto^\dagger \sigma_2$ if $\sigma_2 \mapsto \sigma_1$. All the rules can be directly read from right to left except rule $\mapsto_1^\dagger$ which uses $\delta^\dagger$ to invert the result of δ. We say $\mathrm{eval}^\dagger(c, v_1) = v_2$ if $[c \mid v_1 \mid \Box] \mapsto^{\dagger *} \langle c \mid v_2 \mid \Box \rangle$*

The evaluation functions are inverses to each other.

Theorem 6 (Π-reversible). *For all c, v_1, and v_2, we have $\mathrm{eval}(c, v_1) = v_2$ iff $\mathrm{eval}^\dagger(c, v_2) = v_1$.*

2.2 Interpreter

Although termination of the Π-abstract machine is straightforward to prove directly, we will only state it in Thm. 11 as a corollary of a general proof technique developed in the next section. The

$$\frac{c \text{ base combinator}}{\text{interp}(c, v) \Downarrow \delta(c, v)} \qquad \frac{\text{interp}(c_1, v) \Downarrow w}{\text{interp}(c_1 \oplus c_2, \mathsf{inj}_1\ v) \Downarrow \mathsf{inj}_1\ w} \qquad \frac{\text{interp}(c_2, v) \Downarrow w}{\text{interp}(c_1 \oplus c_2, \mathsf{inj}_2\ v) \Downarrow \mathsf{inj}_2\ w}$$

$$\frac{}{\text{interp}(\mathsf{id}\leftrightarrow, v) \Downarrow v} \qquad \frac{\text{interp}(c_1, v) \Downarrow v_1 \quad \text{interp}(c_2, v_1) \Downarrow v_2}{\text{interp}(c_1 \fatsemi c_2, v) \Downarrow v_2} \qquad \frac{\text{interp}(c_1, v_1) \Downarrow w_1 \quad \text{interp}(c_2, v_2) \Downarrow w_2}{\text{interp}(c_1 \otimes c_2, (v_1, v_2)) \Downarrow (w_1, w_2)}$$

Fig. 5. Π-interpreter

fact that Π-evaluation terminates and is deterministic means that it is a total function. A more efficient and more readable specification of this total function is in Fig. 5.

Theorem 7 (Π-interpreter). *For all c, v_1, and v_2,* $\text{eval}(c, v_1) = v_2$ *iff* $\text{interp}(c, v_1) \Downarrow v_2$.

3 TERMINATION OF REVERSIBLE ABSTRACT MACHINES

We dedicate this section to establish general properties that imply the termination of a large class of reversible abstract machines. These properties will allow us to prove the termination of the extensions of Π with negative and fractional types even though these extensions involve backtracking, iterators, allocation, and other features that could potentially cause non-termination. Our technique is similar to the technique used to prove the termination of reversible flowcharts [Yokoyama et al. 2016] but our setup and theorem statements are slightly more general and might apply to other reversible languages [Abramsky 2005; Yokoyama and Glück 2007].
We begin by formulating a general definition of reversible abstract machines.

Definition 8 (Reversible Abstract Machine). *A reversible abstract machine is a pair $(S, \rightsquigarrow)$, where S is the set of machine states and $\rightsquigarrow$ is a transition relation satisfying the following two properties:*

- *$\rightsquigarrow$ is forward deterministic: $\forall s, s_1, s_2 \in S$ if $s \rightsquigarrow s_1$ and $s \rightsquigarrow s_2$ then $s_1 = s_2$.*
- *$\rightsquigarrow$ is backward deterministic: $\forall s, s_1, s_2 \in S$ if $s_1 \rightsquigarrow s$ and $s_2 \rightsquigarrow s$ then $s_1 = s_2$.*

A remarkable property of all such reversible abstract machines is that no state ever repeats during an evaluation trace.

Lemma 9 (non-repeating). *For any reversible abstract machine $(S, \rightsquigarrow)$, if $s_0 \in S$ is an initial state (i.e., a state s_0 such that $\nexists s.s \rightsquigarrow s_0$) then for any $n < m$ and $s_n, s_m \in S$ such that $s_0 \rightsquigarrow^n s_n$ and $s_0 \rightsquigarrow^m s_m$ we have that $s_n \neq s_m$.*

Proof. If $s_n = s_m$, since $s_0 \rightsquigarrow^n s_n$, $s_0 \rightsquigarrow^m s_m$ and $n < m$ so there exists s_{m-n} such that $s_0 \rightsquigarrow^{m-n} s_{m-n} \rightsquigarrow^n s_m = s_n$. Since $\rightsquigarrow$ is backward deterministic, so $s_0 = s_{m-n}$. However, since $n < m$ there must exist s_{m-n-1} such that $s_{m-n-1} \rightsquigarrow s_{m-n} = s_0$ which contradicts to the assumption that s_0 is an initial state. Hence, $s_n \neq s_m$. □

The above lemma does not, by itself, imply termination. In fact, the lemma applies to extensions of Π with recursive types [Bowman et al. 2011; James and Sabry 2012a] where there exist infinite evaluation traces. In those cases, the lemma just implies that these infinite evaluation traces do not include repeated states. What the lemma does provide is a reduction of termination to the finiteness of the set of states reachable from a start state. Since the latter property is straightforward for the case of Π, we can immediately conclude the termination of the Π-abstract machine.

Lemma 10 (Π-stuck states). *For all states σ, if σ is stuck then $\sigma = [c \mid v \mid \square]$.*

Theorem 11 (Π-termination). *For all Π-combinators $c : A \leftrightarrow B$ and $v_1 : A$ there exists $v_2 : B$ such that* $\text{eval}(c, v_1) = v_2$

Proof. The Π-abstract machine is both forward and backward deterministic. Furthermore, the set of reachable states starting from $\langle c \mid v \mid \square \rangle$ is finite. Hence, because no state repeats, evaluation must eventually reach a stuck state σ such that $\nexists \sigma'.\sigma \mapsto \sigma'$. By Lem. 10, the stuck state must be a final state of the form $[c \mid v_2 \mid \square]$. □

In Secs. 6 and 7, we consider more general reversible machines which compute *partial* reversible functions. The following is a more general lemma for such reversible machines.

Definition 12 (Partial Reversible Abstract Machine). *A partial reversible machine is a triple $(S, \mathcal{E}, \rightsquigarrow)$, where S is the set of machine states, $\mathcal{E} \subseteq S$ is a set of error states, and $\rightsquigarrow$ is a transition relation satisfying the following three properties:*

- *$\rightsquigarrow$ is forward deterministic: $\forall s, s_1, s_2 \in S$ if $s \rightsquigarrow s_1$ and $s \rightsquigarrow s_2$ then $s_1 = s_2$.*
- *$\rightsquigarrow$ is backward deterministic on proper states: $\forall s, s_1, s_2 \in S$ if $s \notin \mathcal{E}$, $s_1 \rightsquigarrow s$, and $s_2 \rightsquigarrow s$ then $s_1 = s_2$.*
- *If $s \in \mathcal{E}$ then s is stuck ($\nexists s'.s \rightsquigarrow s'$).*

Lemma 13 (non-repeating with failure). *For any partial reversible machine $(S, \mathcal{E}, \rightsquigarrow)$, if $s_0 \in S$ is an initial state then for any $n < m$ and $s_n, s_m \in S$ such that $s_0 \rightsquigarrow^n s_n$ and $s_0 \rightsquigarrow^m s_m$ we have that $s_n \neq s_m$.*

Proof. Assume $s_n = s_m$, if $s_n \in \mathcal{E}$ then it is stuck and there does not exist s_m such that $s_0 \mapsto^n s_n \mapsto^{m-n} s_m$. Hence $s_m = s_n \notin \mathcal{E}$. The rest of the proof is similar to Lem. 9. □

4 SPACE AND TIME RESOURCES AND TRADE-OFFS

In the next two sections, we explain how to extend Π in two independent directions each yielding a distinct compact closed category: one of these categories will have a dual $-$ for the type constructor $+$ and the other a dual $/$ for the type constructor $\times$. This section develops the intuition that the type constructor $+$ is related to the time needed for runtime execution and that the type constructor $\times$ is related to the space needed for runtime execution thus setting the stage for an interpretation of $-$ as "going backwards in time" (backtracking of control flow) and of $/$ as "going backwards in space" (reclaiming of allocated storage).

Given the small-step abstract machine introduced in the previous section, it is relatively straightforward to compute the time and space resources needed by a computation: "time" is modeled by the number of machine transition steps and an upper bound on "space" resources is modeled by the maximum size of intermediate machine states visited during execution. A simple abstract measure of the size of a machine state $\#\sigma$ is the number of values stored in that state (whether in the v register or in the continuation register κ). This measure essentially counts the number of live "pebbles" in the "pebble game" models of reversible computation [Bennett 1989; Chan 2013] and is defined as follows:

$$\begin{array}{lll}
\#\sigma = 1 + \#\sigma.\kappa & \#\square = 0 & \\
 & \#(\square \fatsemi c_2) \bullet \kappa = \#\kappa & \#(c_1 \fatsemi \square) \bullet \kappa = \#\kappa \\
 & \#(\square \oplus c_2) \bullet \kappa = \#\kappa & \#(c_1 \oplus \square) \bullet \kappa = \#\kappa \\
 & \#(\square \otimes [c_2, v]) \bullet \kappa = 1 + \#\kappa & \#([c_1, v] \otimes \square) \bullet \kappa = 1 + \#\kappa
\end{array}$$

Since every state has a value register, its size is 1 plus the size of the continuation register. This measure ignores the (fixed and constant) space needed to store the program itself (i.e., the combinators occurring in either the c component or the κ component) and focuses on the maximum dynamic space requirements needed for each state during execution.

To make the case that the number of machine transition steps (i.e., time) is related to the $+$ type constructor and that the space used by intermediate machine states is related to the $\times$ type constructor, we provide a small compelling example. A type containing 16 values can be represented

as a sum of 8 booleans $\mathbb{B} + (\mathbb{B} + (\mathbb{B} + (\mathbb{B} + (\mathbb{B} + (\mathbb{B} + (\mathbb{B} + \mathbb{B}))))))$ or the product of 4 booleans $\mathbb{B} \times (\mathbb{B} \times (\mathbb{B} \times \mathbb{B}))$. In both cases, we can distinguish one boolean b and consider it indexed by the remaining 8 values as shown below:

$(\mathsf{inj}_1\ \mathsf{b})$	$(\mathbb{F}, (\mathbb{F}, (\mathbb{F}, \mathsf{b})))$
$(\mathsf{inj}_2\ (\mathsf{inj}_1\ \mathsf{b}))$	$(\mathbb{F}, (\mathbb{F}, (\mathbb{T}, \mathsf{b})))$
$(\mathsf{inj}_2\ (\mathsf{inj}_2\ (\mathsf{inj}_1\ \mathsf{b}))$	$(\mathbb{F}, (\mathbb{T}, (\mathbb{F}, \mathsf{b})))$
$(\mathsf{inj}_2\ (\mathsf{inj}_2\ (\mathsf{inj}_2\ (\mathsf{inj}_1\ \mathsf{b}))))$	$(\mathbb{F}, (\mathbb{T}, (\mathbb{T}, \mathsf{b})))$
$(\mathsf{inj}_2\ (\mathsf{inj}_2\ (\mathsf{inj}_2\ (\mathsf{inj}_2\ (\mathsf{inj}_1\ \mathsf{b}))))))$	$(\mathbb{T}, (\mathbb{F}, (\mathbb{F}, \mathsf{b})))$
$(\mathsf{inj}_2\ (\mathsf{inj}_2\ (\mathsf{inj}_2\ (\mathsf{inj}_2\ (\mathsf{inj}_2\ (\mathsf{inj}_1\ \mathsf{b}))))))$	$(\mathbb{T}, (\mathbb{F}, (\mathbb{T}, \mathsf{b})))$
$(\mathsf{inj}_2\ (\mathsf{inj}_2\ (\mathsf{inj}_2\ (\mathsf{inj}_2\ (\mathsf{inj}_2\ (\mathsf{inj}_2\ (\mathsf{inj}_1\ \mathsf{b}))))))))$	$(\mathbb{T}, (\mathbb{T}, (\mathbb{F}, \mathsf{b})))$
$(\mathsf{inj}_2\ (\mathsf{inj}_2\ (\mathsf{inj}_2\ (\mathsf{inj}_2\ (\mathsf{inj}_2\ (\mathsf{inj}_2\ (\mathsf{inj}_2\ \mathsf{b}))))))))$	$(\mathbb{T}, (\mathbb{T}, (\mathbb{T}, \mathsf{b})))$

On the left, the boolean b is indexed by its position in the wide sum type; on the right, the boolean b is indexed by the values of the three other booleans. Clearly the two representations are in 1-1 correspondence and hence it is possible to write a Π-combinator that mediates between the two representations (see the accompanying code for the definition).
Now consider two programs that negate the distinguished b and apply them to the inputs in the last line. The definitions, correctness, and equivalence of these two programs are in the accompanying code. The big difference between them is in the amount of resources they use. Accessing b in the additive representation on the left requires "time" to decode the sum type; accessing b in the multiplicative representation on the right is immediate but requires additional space to store the three booleans. Indeed, when indexed by integers from 0 to 511, the additive program takes 1024 steps using 1 unit of space, whereas the multiplicative program takes 128 steps using 10 units of space. Generally, the additive approach takes $O(n)$ time using just one unit of space whereas the multiplicative approach takes $O(\log n)$ time at the expense of using $O(\log n)$ units of space.

5 NEGATIVE TYPES: Π^m

Aiming to construct a compact closed category with an additive dual, we extend Π with two combinators η_+ and ε_+ witnessing the isomorphism $\mathbb{0} \leftrightarrow A + (-A)$. As explained in the previous section, values of a sum type $A + B$ represent choices that are exercised at different times: at one time we may have a value of type A and at another time we may have a value of type B. In the case of A and $-A$ we would like the two options to cancel each other in some kind of destructive interference. This suggests that values should be equipped with a wave-like (but degenerate) notion of amplitude or phase. We simply equip values v with either a positive sign (which is omitted by convention) or a negative sign $-v$. Semantically the negative sign is interpreted as flipping the flow of evaluation: forward evaluation becomes backward evaluation and vice-versa. We do expect (and we will confirm) that flipping the flow of evaluation twice has no effect. Operationally, since the Π-abstract machine is already reversible, all that is needed is to arrange for negative values to use the backward evaluation relation $\mapsto^\dagger$.
Before proceeding with the semantic definitions, we note that our interpretation of negative types as values flowing in a different direction suggests that the size of $-A$ is equal to the size of A, i.e., that $|-A| = |A|$. Hence if A is inhabited, the isomorphism $\mathbb{0} \leftrightarrow A + (-A)$ can no longer be interpreted as a permutation as the two sides have a different number of elements. Our operational semantics instead suggests that $-A$ should be considered as a "set with negative cardinality" [Propp 2002; Schanuel 1991] so that the *net* flow of values associated with a type $A + (-A)$ is indeed 0.

5.1 Abstract Machine Semantics

The syntax of Π^m extends the syntax of Π as follows:

$$
\begin{array}{llcl}
\textit{Value types} & A, B, C, D & ::= & \cdots \mid -A \\
\textit{Values} & v, w, x, y & ::= & \cdots \mid -v \\
\textit{Programs} & c & ::= & \cdots \mid \eta_+ : \mathbb{0} \leftrightarrow A - A : \epsilon_+
\end{array}
$$

where we abbreviate $A + (-B)$ as $A - B$. Graphically, the combinators η_+ and ϵ_+ look like "U-turns":

It is impossible to evaluate η_+ in the forward direction as this would require supplying a value of the empty type $\mathbb{0}$. In the backward direction, η_+ either expects a value $\mathsf{inj}_1\ v$ or a value $\mathsf{inj}_2\ (-v)$. In either case, it flips the flow of evaluation with the appropriate value.

$$
\frac{\langle c \mid v \mid \kappa\rangle \mapsto_i \sigma}{\langle c \mid v \mid \kappa\rangle_\triangleright \overset{m}{\longmapsto}_i \sigma_\triangleright}
\quad
\frac{[c \mid v \mid \kappa] \mapsto_i \sigma}{[c \mid v \mid \kappa]_\triangleright \overset{m}{\longmapsto}_i \sigma_\triangleright}
\quad
\frac{\langle c \mid v \mid \kappa\rangle \mapsto_i^\dagger \sigma}{\langle c \mid v \mid \kappa\rangle_\triangleleft \overset{m}{\longmapsto}_{i^\dagger} \sigma_\triangleleft}
\quad
\frac{[c \mid v \mid \kappa] \mapsto_i^\dagger \sigma}{[c \mid v \mid \kappa]_\triangleleft \overset{m}{\longmapsto}_{i^\dagger} \sigma_\triangleleft}
$$

$$
\begin{array}{rcl@{\qquad}rcl}
\langle \epsilon_+ \mid \mathsf{inj}_1\ v \mid \kappa\rangle_\triangleright & \overset{m}{\longmapsto}_{13} & \langle \epsilon_+ \mid \mathsf{inj}_2\ (-v) \mid \kappa\rangle_\triangleleft & [\eta_+ \mid \mathsf{inj}_2\ (-v) \mid \kappa]_\triangleleft & \overset{m}{\longmapsto}_{15} & [\eta_+ \mid \mathsf{inj}_1\ v \mid \kappa]_\triangleright \\
\langle \epsilon_+ \mid \mathsf{inj}_2\ (-v) \mid \kappa\rangle_\triangleright & \overset{m}{\longmapsto}_{14} & \langle \epsilon_+ \mid \mathsf{inj}_1\ v \mid \kappa\rangle_\triangleleft & [\eta_+ \mid \mathsf{inj}_1\ v \mid \kappa]_\triangleleft & \overset{m}{\longmapsto}_{16} & [\eta_+ \mid \mathsf{inj}_2\ (-v) \mid \kappa]_\triangleright
\end{array}
$$

Fig. 6. Π^m-abstract machine

In order to manage the additional expressiveness of negative types, we extend machine states to internally maintain a direction $\triangleright$ or $\triangleleft$. The definition of extended machine states follows.

Definition 14 (Π^m-machine states). *A Π^m-machine state σ is either:*

- *An enter state: $\langle c \mid v \mid \kappa\rangle_d$ where $c : A \leftrightarrow B$, $v : A$, $\kappa : \textsc{cont}_{A\leftrightarrow B}$, and $d \in \{\triangleright, \triangleleft\}$.*
- *A return state: $[c \mid v \mid \kappa]_d$ where $c : A \leftrightarrow B$, $v : B$, $\kappa : \textsc{cont}_{A\leftrightarrow B}$, and $d \in \{\triangleright, \triangleleft\}$.*

Fig. 6 gives the transition rules of the abstract machine. The first four judgments lift all the rules of the Π-abstract machine to the extended machine: the first two rules lift the regular $\mapsto$ rules and the next two rules lift the reverse rules $\mapsto^\dagger$ with the convention that the new names are tagged with $i^\dagger$ for each transition i. The remaining four transitions deal with the new combinators η_+ and ε_+. As there are no values of type $\mathbb{0}$, it is impossible for an enter state to make progress on η_+ and it is impossible for a return state to make progress on ε_+. For ε_+ it is only possible to make progress when entering the state in the forward direction resulting in a state that executes in the backward direction, swapping the direction of the value and the injection tag in the sum type. Symmetrically, a return state with η_+ in the combinator position does not return anything to the continuation but instead flips the direction of execution again. Final states include the usual final states from the Π-abstract machine but also "backward states" of the form $\langle c \mid v \mid \square\rangle_\triangleleft$. Such final states can be proper results for programs with negative types. For example, evaluating $\langle \varepsilon_+ \mid \mathsf{inj}_1\ \mathsf{tt} \mid \square\rangle_\triangleright$ proceeds using $\overset{m}{\longmapsto}_{13}$ to $\langle \varepsilon_+ \mid \mathsf{inj}_2\ (-\mathsf{tt}) \mid \square\rangle_\triangleleft$ from which no further transitions are possible.

5.2 Properties

As explained in Sec. 3, the finiteness of the set of reachable states along with Lem. 9 are sufficient to prove termination of the Π^m-abstract machine. The latter lemma just requires that we prove the transition relation to be forward and backward deterministic.

Lemma 15 (Π^m-forward deterministic). *If $\sigma \overset{m}{\longmapsto} \sigma_1$ and $\sigma \overset{m}{\longmapsto} \sigma_2$ then $\sigma_1 = \sigma_2$*

Lemma 16 (Π^m-backward deterministic). *If $\sigma_1 \overset{m}{\longmapsto} \sigma$ and $\sigma_2 \overset{m}{\longmapsto} \sigma$ then $\sigma_1 = \sigma_2$*

Lemma 17 (Stuck). *If σ is stuck ($\nexists \sigma'.\sigma \overset{m}{\longmapsto} \sigma'$), then either $\sigma = \langle c \mid v \mid \square\rangle_\triangleleft$ or $\sigma = [c \mid v \mid \square]_\triangleright$.*

Definition 18 (Π^m-forward evaluation (limited)). *We say* $\mathrm{eval}^m(c, v_1) = v_2$ *if* $\langle c \mid v_1 \mid \square\rangle_\triangleright \overset{m\,*}{\longmapsto} [c \mid v_2 \mid \square]_\triangleright$ *or* $\langle c \mid v_1 \mid \square\rangle_\triangleright \overset{m\,*}{\longmapsto} \langle c \mid v_2 \mid \square\rangle_\triangleleft$. *Note that in the first case* $v_2 : B$ *but in the second case* $v_2 : A$. *We address this point below in Def. 20.*

Theorem 19 (Π^m-termination). *For all* Π^m*-combinators* $c : A \leftrightarrow B$ *and* $v_1 : A$ *there exists* v_2 *such that* $\mathrm{eval}^m(c, v_1) = v_2$

Proof. The result follows from the following facts: (i) starting from an initial state, the set of reachable states is finite, and (ii) the Π^m-abstract is both forward and backward deterministic and hence by Lem. 9, no state ever repeats during an evaluation trace. □

The theorem does not imply that there does not exist machine states from which evaluation will diverge. For example, evaluation starting from $\langle \varepsilon_+ \mid \mathsf{inj}_1\ \mathsf{tt} \mid (\eta_+ \fatsemi \square) \bullet \square\rangle_\triangleright$ will loop forever. This state is, however, not reachable from an initial state because there are no inhabitants of $\mathbb{0}$.
The language Π^m is also reversible but expressing this is more complicated than the case for Π. In fact, even Def. 18 of forward evaluation is not general enough as applying a combinator c to a value of type B in the backward direction turns out to be identical to applying it to a value of type $-B$ in the forward direction. To formalize this idea we introduce the composite set $(\!|A^{\rightarrow}, B^{\leftarrow}|\!)$ whose elements can be either a value $v : A^{\rightarrow}$ flowing in the forward direction or a value $w : B^{\leftarrow}$ flowing in the backward direction. Given this definition, we can now express the correct general definitions of both forward and backward evaluations.

Definition 20 (Π^m-forward evaluation). *We define:*

$$\mathrm{eval}^m : (A \leftrightarrow B) \to (\!|A^{\rightarrow}, B^{\leftarrow}|\!) \to (\!|B^{\rightarrow}, A^{\leftarrow}|\!)$$

as follows:

$$\begin{array}{rcl}
\mathrm{eval}^m(c, v : A^{\rightarrow}) & = & \begin{cases} w : B^{\rightarrow} & \text{if } \langle c \mid v \mid \square\rangle_\triangleright \overset{m\,*}{\longmapsto} [c \mid w \mid \square]_\triangleright \\ w : A^{\leftarrow} & \text{if } \langle c \mid v \mid \square\rangle_\triangleright \overset{m\,*}{\longmapsto} \langle c \mid w \mid \square\rangle_\triangleleft \end{cases} \\
\mathrm{eval}^m(c, v : B^{\leftarrow}) & = & \begin{cases} w : B^{\rightarrow} & \text{if } [c \mid v \mid \square]_\triangleleft \overset{m\,*}{\longmapsto} [c \mid w \mid \square]_\triangleright \\ w : A^{\leftarrow} & \text{if } [c \mid v \mid \square]_\triangleleft \overset{m\,*}{\longmapsto} \langle c \mid w \mid \square\rangle_\triangleleft \end{cases}
\end{array}$$

Evaluation can start in either direction and end in either direction.

Definition 21 (Π^m-backward evaluation). *We define:*

$$\mathrm{eval}^{m\dagger} : (A \leftrightarrow B) \to (\!|B^{\rightarrow}, A^{\leftarrow}|\!) \to (\!|A^{\rightarrow}, B^{\leftarrow}|\!)$$

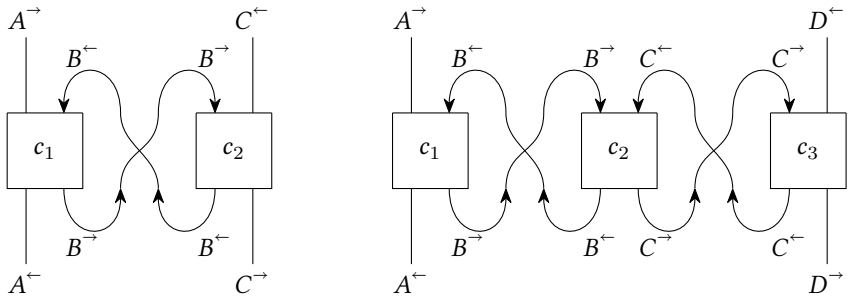
as follows:

$$\begin{array}{rcl}
\mathrm{eval}^{m\dagger}(c, v : B^{\rightarrow}) & = & \begin{cases} w : A^{\rightarrow} & \text{if } [c \mid v \mid \square]_\triangleleft \overset{m\,*}{\longmapsto} \langle c \mid w \mid \square\rangle_\triangleleft \\ w : B^{\leftarrow} & \text{if } [c \mid v \mid \square]_\triangleleft \overset{m\,*}{\longmapsto} [c \mid w \mid \square]_\triangleright \end{cases} \\
\mathrm{eval}^{m\dagger}(c, v : A^{\leftarrow}) & = & \begin{cases} w : A^{\rightarrow} & \text{if } \langle c \mid v \mid \square\rangle_\triangleright \overset{m\,*}{\longmapsto} \langle c \mid w \mid \square\rangle_\triangleleft \\ w : B^{\leftarrow} & \text{if } \langle c \mid v \mid \square\rangle_\triangleright \overset{m\,*}{\longmapsto} [c \mid w \mid \square]_\triangleright \end{cases}
\end{array}$$

Evaluation can start in either direction and end in either direction.

The punch line is that these two evaluations are inverses.

Theorem 22. *For all* $c : A \leftrightarrow B$, $V : (\!|A^{\rightarrow}, B^{\leftarrow}|\!)$, *and* $W : (\!|B^{\rightarrow}, A^{\leftarrow}|\!)$, *we have* $\mathrm{eval}^m(c, V) = W$ *iff* $\mathrm{eval}^{m\dagger}(c, W) = V$.



(a) Π^m composition (b) Associativity of Π^m composition

Fig. 7. Sequential composition in Π^m.

5.3 Interpreter

The accompanying Agda code includes a more intuitive, and provably equivalent, specification of the Π^m-evaluation relation as a recursive interpreter. The specification includes many cases, most of which are simple, so we content ourselves with presenting the most instructive cases in this section. The interpreter has the same signature as the evaluation relation in Def. 20:

$$\mathrm{interp}^m : (A \leftrightarrow B) \rightarrow (\!|A^{\rightarrow}, B^{\leftarrow}|\!) \rightarrow (\!|B^{\rightarrow}, A^{\leftarrow}|\!)$$

It may be called with a forward or backward value and may return a forward or backward value making the specification somewhat long. The general structure is however simple:

- For base combinators, evaluation in each direction is identical to evaluation in the Π-interpreter: $\mathrm{interp}^m(c, v^{\rightarrow}) \Downarrow \delta(c, v)^{\rightarrow}$ and $\mathrm{interp}^m(c, v^{\leftarrow}) \Downarrow \delta^{\dagger}(c, v)^{\leftarrow}$.
- For $\oplus$ there are many cases that are again identical to the corresponding cases in the Π-interpreter for each combination of directions. We show two of the cases; the others are similar:

$$\frac{\mathrm{interp}^m(c_1, v^{\rightarrow}) \Downarrow w^{\rightarrow}}{\mathrm{interp}^m(c_1 \oplus c_2, (\mathsf{inj}_1\ v)^{\rightarrow}) \Downarrow (\mathsf{inj}_1\ w)^{\rightarrow}} \qquad \frac{\mathrm{interp}^m(c_1, v^{\leftarrow}) \Downarrow w^{\leftarrow}}{\mathrm{interp}^m(c_1 \oplus c_2, (\mathsf{inj}_1\ v)^{\leftarrow}) \Downarrow (\mathsf{inj}_1\ w)^{\leftarrow}}$$

- There are new cases for η_+ that only accept values flowing backwards flipping the evaluation direction:

$$\mathrm{interp}^m(\eta_+, \mathsf{inj}_1\ v^{\leftarrow}) \Downarrow \mathsf{inj}_2\ (-v)^{\rightarrow} \qquad \mathrm{interp}^m(\eta_+, \mathsf{inj}_2\ (-v)^{\leftarrow}) \Downarrow \mathsf{inj}_1\ v^{\rightarrow}$$

- There are new cases for ε_+ that only accept values flowing forwards flipping the evaluation direction:

$$\mathrm{interp}^m(\varepsilon_+, \mathsf{inj}_1\ v^{\rightarrow}) \Downarrow \mathsf{inj}_2\ (-v)^{\leftarrow} \qquad \mathrm{interp}^m(\varepsilon_+, \mathsf{inj}_2\ (-v)^{\rightarrow}) \Downarrow \mathsf{inj}_1\ v^{\leftarrow}$$

- For $\otimes$, we start evaluating the first component of the pair. The value of this first component may either be in the same direction as the incoming pair in which case we continue evaluation with the second component as usual. If however the value of the first component is tagged with the opposite direction, we treat this an *exception*, i.e., we interrupt the evaluation in the given direction and immediately return in the reverse direction. The following two cases capture the main idea:

$$\frac{\mathrm{interp}^m(c_1, v_1^{\rightarrow}) \Downarrow w_1^{\rightarrow} \quad \mathrm{interp}^m(c_2, v_2^{\rightarrow}) \Downarrow w_2^{\rightarrow}}{\mathrm{interp}^m(c_1 \otimes c_2, (v_1, v_2)^{\rightarrow}) \Downarrow (w_1, w_2)^{\rightarrow}} \qquad \frac{\mathrm{interp}^m(c_1, v_1^{\rightarrow}) \Downarrow w^{\leftarrow}}{\mathrm{interp}^m(c_1 \otimes c_2, (v_1, v_2)^{\rightarrow}) \Downarrow (w, v_2)^{\leftarrow}}$$

- The case for sequential composition $c_1 \fatsemi c_2$ is the most interesting one. Assume the incoming value is $v^{\rightarrow}$: we begin by applying c_1 to $v^{\rightarrow}$; if the result is tagged in the backward direction, we throw an exception like above. Otherwise, we would like to continue by applying c_2 to the result. To prepare for the possibility that c_2 might throw an exception, we apply c_2 in the context of a *handler* whose signature is: $(A \leftrightarrow B) \rightarrow B \rightarrow (B \leftrightarrow C) \rightarrow (\!|C^{\rightarrow}, A^{\leftarrow}|\!)$. The handler describes a state in which the intermediate value between c_1 and c_2 is known and evaluation is about to start applying c_2 in the forward direction. If the result of c_2 is also in the forward direction, the evaluation of the entire sequential composition terminates. If however the result of evaluating c_2 produces a value flowing in the backward direction, we would like to continue by applying c_1 in the backward direction. To prepare for the possibility that c_1 might throw an exception we apply c_1 in the context of *yet another handler* with the same signature: $(A \leftrightarrow B) \rightarrow B \rightarrow (B \leftrightarrow C) \rightarrow (\!|C^{\rightarrow}, A^{\leftarrow}|\!)$. This handler however describes a state in which the intermediate value between c_1 and c_2 is known but evaluation is about to start with c_1 in the backward direction. A priori, there is no guarantee that evaluation will not bounce back and forth in an infinite loop but the proof of equivalence of the interpreter to the abstract machine guarantees termination. Below we show two cases of the evaluation for sequential composition:

$$\frac{\mathrm{interp}^m(c_1, v^{\rightarrow}) \Downarrow w^{\leftarrow}}{\mathrm{interp}^m(c_1 \fatsemi c_2, v^{\rightarrow}) \Downarrow w^{\leftarrow}} \qquad \frac{\mathrm{interp}^m(c_1, v^{\rightarrow}) \Downarrow w^{\rightarrow} \quad \mathrm{handle}\triangleright(c_1, w^{\rightarrow}, c_2) \Downarrow V}{\mathrm{interp}^m(c_1 \fatsemi c_2, v^{\rightarrow}) \Downarrow V}$$

The forward handler is defined using two cases. The backward handler is similar:

$$\frac{\mathrm{interp}^m(c_2, v^{\rightarrow}) \Downarrow w^{\rightarrow}}{\mathrm{handle}\triangleright(c_1, v^{\rightarrow}, c_2) \Downarrow w^{\rightarrow}} \qquad \frac{\mathrm{interp}^m(c_2, v^{\rightarrow}) \Downarrow w^{\leftarrow} \quad \mathrm{handle}\triangleleft(c_1, w^{\leftarrow}, c_2) \Downarrow V}{\mathrm{handle}\triangleright(c_1, v^{\rightarrow}, c_2) \Downarrow V}$$

The flow of values in the case of sequential composition is intuitively described by the H-shape diagram in Fig. 7a. Note that this is exactly the same flow of values that happen in both the geometry of interaction [Abramsky et al. 2002] and the Int-construction [Joyal et al. 1996].

Theorem 23 (Π^m-interpreter). *For all c, V, and W, $\mathrm{eval}^m(c, V) = W$ iff $\mathrm{interp}^m(c, V) \Downarrow W$.*

5.4 Compact Closed Category

We conclude this section by confirming that the operational semantics above forms a compact closed category. The construction is formalized in Agda using the `agda-categories` library[2].

Theorem 24. *Let $\sim$ be the following equivalence relation on combinators $c_1 \sim c_2$ iff $\mathrm{eval}^m(c_1, _) = \mathrm{eval}^m(c_2, _)$ identifying combinators with the same behavior under evaluation. The notation $[c]_\sim$ refers to a representative combinator from a $\sim$-equivalence class.[3] Using this relation, we define the category $(\mathcal{C}, +, \mathbb{0})$ as follows:*

- $\mathrm{Obj}(\mathcal{C})$ *is the set of* Π^m *types,*
- $\mathrm{Hom}(A, B) = [c : A \leftrightarrow B]_\sim$, *and*
- *dual objects* $-A$ *for every* A

The category $(\mathcal{C}, +, \mathbb{0})$ is a compact closed category.

Proof. Composition in $\mathcal{C}$ is the concatenation of combinators $_\fatsemi_$; the identity morphisms are the equivalence classes of $\mathsf{id}\leftrightarrow$ at each type. The basic properties of identity morphisms $\mathrm{eval}^m(\mathsf{id}\leftrightarrow \fatsemi c) = \mathrm{eval}^m(c) = \mathrm{eval}^m(c \fatsemi \mathsf{id}\leftrightarrow)$, for all c, are straightforward by appealing to the interpreter

[2] https://github.com/agda/agda-categories [Hu and Carette 2020]

[3] This approach is sufficient to prove the semantics forms a 1-category but ignores the rich structure at the next level [Carette and Sabry 2016].

instead of the abstract machine. The most interesting proof is the one for the associativity of sequential composition. The situation is diagrammatically depicted in Fig. 7b where it is intuitively clear that composition is associative. We outline a proof that establishes a bisimulation between the different execution traces of the abstract machine. There are two possible inputs $a^{\rightarrow}$ and $b^{\leftarrow}$. We consider the $a^{\rightarrow}$ case below; the other case is similar. Let $\sigma_0 = \langle c_1 \fatsemi (c_2 \fatsemi c_3) \mid a \mid \square \rangle_{\triangleright}$ and for each n, let σ_n be the state reachable after n steps $\sigma_0 \overset{m}{\longmapsto}^{n} \sigma_n$. We aim to relate these states to the states reachable from $(c_1 \fatsemi c_2) \fatsemi c_3$. To that end, we define a function T such that $T(\sigma_n) \overset{m}{\longmapsto}^{*} T(\sigma_{n+1})$ whenever $\sigma_n \overset{m}{\longmapsto} \sigma_{n+1}$:

$$
\begin{array}{rcl}
T(\langle c_1 \fatsemi (c_2 \fatsemi c_3) \mid v \mid \square \rangle_d) & = & \langle (c_1 \fatsemi c_2) \fatsemi c_3 \mid v \mid \square \rangle_d \\
T(\langle c \mid v \mid \ldots (\square \fatsemi (c_2 \fatsemi c_3)) \bullet \square \rangle_d) & = & \langle c \mid v \mid \ldots (\square \fatsemi c_2) \bullet (\square \fatsemi c_3) \bullet \square \rangle_d \\
T(\langle c_2 \fatsemi c_3 \mid v \mid (c_1 \fatsemi \square) \bullet \square \rangle_d) & = & \langle c_2 \mid v \mid (c_1 \fatsemi \square) \bullet (\square \fatsemi c_3) \bullet \square \rangle_d \\
T(\langle c \mid v \mid \ldots (\square \fatsemi c_3) \bullet (c_1 \fatsemi \square) \bullet \square \rangle_d) & = & \langle c \mid v \mid \ldots (c_1 \fatsemi \square) \bullet (\square \fatsemi c_3) \bullet \square \rangle_d \\
T(\langle c \mid v \mid \ldots (c_2 \fatsemi \square) \bullet (c_1 \fatsemi \square) \bullet \square \rangle_d) & = & \langle c \mid v \mid \ldots ((c_1 \fatsemi c_2) \fatsemi \square) \bullet \square \rangle_d \\
\\
T([c_1 \fatsemi (c_2 \fatsemi c_3) \mid v \mid \square]_d) & = & [(c_1 \fatsemi c_2) \fatsemi c_3 \mid v \mid \square]_d \\
T([c \mid v \mid \ldots (\square \fatsemi (c_2 \fatsemi c_3)) \bullet \square]_d) & = & [c \mid v \mid \ldots (\square \fatsemi c_2) \bullet (\square \fatsemi c_3) \bullet \square]_d \\
T([c_2 \fatsemi c_3 \mid v \mid (c_1 \fatsemi \square) \bullet \square]_d) & = & [c_2 \mid v \mid (c_1 \fatsemi \square) \bullet (\square \fatsemi c_3) \bullet \square]_d \\
T([c \mid v \mid \ldots (\square \fatsemi c_3) \bullet (c_1 \fatsemi \square) \bullet \square]_d) & = & [c \mid v \mid \ldots (c_1 \fatsemi \square) \bullet (\square \fatsemi c_3) \bullet \square]_d \\
T([c \mid v \mid \ldots (c_2 \fatsemi \square) \bullet (c_1 \fatsemi \square) \bullet \square]_d) & = & [c \mid v \mid \ldots ((c_1 \fatsemi c_2) \fatsemi \square) \bullet \square]_d
\end{array}
$$

By Lem. 17, the transition steps starting from σ_0 end in one of two possible final states. If:

$$\langle c_1 \fatsemi (c_2 \fatsemi c_3) \mid a \mid \square \rangle_{\triangleright} \overset{m}{\longmapsto}^{*} [c_1 \fatsemi (c_2 \fatsemi c_3) \mid v \mid \square]_{\triangleright}$$

we can obtain the corresponding evaluation on the re-associated combinator:

$$T(\sigma_0) = \langle (c_1 \fatsemi c_2) \fatsemi c_3 \mid a \mid \square \rangle_{\triangleright} \overset{m}{\longmapsto}^{*} [(c_1 \fatsemi c_2) \fatsemi c_3 \mid v \mid \square]_{\triangleright}$$

Otherwise:

$$\langle c_1 \fatsemi (c_2 \fatsemi c_3) \mid a \mid \square \rangle_{\triangleright} \overset{m}{\longmapsto}^{*} \langle c_1 \fatsemi (c_2 \fatsemi c_3) \mid v \mid \square \rangle_{\triangleleft}$$

and we can obtain:

$$T(\sigma_0) = \langle (c_1 \fatsemi c_2) \fatsemi c_3 \mid a \mid \square \rangle_{\triangleright} \overset{m}{\longmapsto}^{*} \langle (c_1 \fatsemi c_2) \fatsemi c_3 \mid v \mid \square \rangle_{\triangleleft}$$

After checking that $\mathcal{C}$ is indeed a category, it remains to verify that $(\mathcal{C}, _+_)$ is a rigid symmetric monoidal category. It is straightforward to confirm that the right unitors are definable using the left unitors and braiding:

$$
\begin{array}{rll}
\mathsf{unite_+l} & : \mathbb{0} + A \leftrightarrow A : & \mathsf{uniti_+l} \\
\mathsf{unite_+r} = \mathsf{swap_+} \fatsemi \mathsf{unite_+l} & : A + \mathbb{0} \leftrightarrow A : & \mathsf{uniti_+l} \fatsemi \mathsf{swap_+} = \mathsf{uniti_+r}
\end{array}
$$

All the remaining coherence conditions are straightforward to check (some require bisimulation proofs similar to the above) and are omitted from this proof outline. □

Given that Π^m is a reversible language, one might wonder if the category above forms a groupoid where every morphism f has an inverse f^{-1} such that $f \fatsemi f^{-1} \sim f^{-1} \fatsemi f \sim \mathsf{id}\leftrightarrow$. Unfortunately, this is not the case as there is no combinator c such that $\varepsilon_+ \fatsemi c \sim \mathsf{id}\leftrightarrow$. Instead, following a suggestion by one of the anonymous reviewers, we show that the semantics of Π^m forms an *inverse category* [Kastl 1979], i.e. each morphism f has a unique morphism f^{-1} such that $f \fatsemi f^{-1} \fatsemi f \sim f$ and $f^{-1} \fatsemi f \fatsemi f^{-1} \sim f^{-1}$.

Theorem 25. *The category $\mathcal{C}$ defined in Thm. 24 is an inverse category.*

Proof. We first define the inverse of each combinator:

$$
\begin{array}{rcl}
!c & = & c' \quad \text{if } c \text{ is base combinator, where } c' \text{ is } c\text{'s dual} \\
!\mathsf{id}\leftrightarrow & = & \mathsf{id}\leftrightarrow \\
!(c_1 \oplus c_2) & = & !c_1 \oplus !c_2 \\
!(c_1 \otimes c_2) & = & \mathsf{swap}_* \fatsemi (!c_2 \otimes !c_1) \fatsemi \mathsf{swap}_* \\
!(c_1 \fatsemi c_2) & = & !c_2 \fatsemi !c_1 \\
!\eta_+ & = & \varepsilon_+ \\
!\varepsilon_+ & = & \eta_+
\end{array}
$$

By induction we can show that ! is involutive, i.e., for all c, we have $c \sim !(!c)$ and that it inverts evaluation in the sense that if $\mathrm{eval}^m(c, v_1) = v_2$ then $\mathrm{eval}^m(!c, v_2) = v_1$. Next, using the fact that ! inverts evaluation, we can show that $c \fatsemi !c \fatsemi c \sim c$ by case analysis on the possible outputs of c. And using the fact that ! is involutive we obtain: $!c \fatsemi c \fatsemi !c \sim !c \fatsemi !(!c) \fatsemi !c \sim !c$. □

An immediate consequence of the construction of a compact closed category is that we can define internal hom objects. Specifically, we get a bijection between $A \leftrightarrow B$ and $\mathbb{0} \leftrightarrow (-A + B)$ allowing any combinator $A \leftrightarrow B$ to be used as an object converting demands for values of type A in the backward direction to productions of values of type B in the forward direction. The internal hom is not the familiar exponential object because the relevant tensor is $+$ not $\times$. In this context, the evaluation map is a combinator of type $(-A+B)+A \leftrightarrow B$ and "currying" converts the combinator on the left to the one on the right in the diagrams below:

Operationally, the significance of this "currying" is as follows. On the left, f expects either a value of type A or a value of type B. The caller decides. On the right f speculatively opts to receive only values of type A. If the caller provides a value of type A, then f produces a value of type C as before. A value of type B can still be supplied but after backtracking and entering f in the backward direction. The idea is the basis of our backtracking search abstraction detailed in Sec. 8.

6 FRACTIONAL TYPES: Π^d

We now define an independent extension of Π with multiplicative duals. The analogy to the additive case suggests the introduction of two combinators η_* and ε_* witnessing the isomorphism $\mathbb{1} \leftrightarrow A \times (\mathbb{1}/A)$. The situation however is more subtle than before for at least two reasons. First the empty type $\mathbb{0}$ must be excluded from this extension as otherwise we would have:

$$
\begin{array}{rcll}
\mathbb{1} & \leftrightarrow & \mathbb{0} \times \mathbb{1}/\mathbb{0} & \text{by } \eta_* \\
 & \leftrightarrow & \mathbb{0} & \text{by absorbr}
\end{array}
$$

The second subtlety can be appreciated as soon as one attempts to define the operational semantics for η_*. The application of η_* to tt at type $\mathbb{B}$ must return a pair whose first component is a boolean, but which one? Either we introduce some form of non-determinism or we introduce a mechanism to deterministically select a particular boolean value. Keeping with the deterministic approach in this paper, there are two natural choices: (i) hardwire in the semantics a choice of a default value for each type, or (ii) introduce families of η_* and ε_* parameterized by values. We discovered that hardwiring a default value fails to produce a compact closed category as one coherence condition (snake) would only be satisfied for that particular default value. Fortunately, the second approach that introduces families of η_* and ε_* parameterized by values both eliminates the problem with the empty type and induces a compact closed category as we prove below.

$$
\begin{array}{rcll}
\langle \eta_*{}^{v:A} \mid \mathsf{tt} \mid \kappa \rangle & \overset{d}{\longmapsto}_{13} & \left[\eta_*{}^{v:A} \mid (v, \circlearrowright) \mid \kappa\right] & \\
\langle \varepsilon_*{}^{v_1:A} \mid (v_2, \circlearrowright) \mid \kappa \rangle & \overset{d}{\longmapsto}_{14} & \left[\varepsilon_*{}^{v_1:A} \mid \mathsf{tt} \mid \kappa\right] & \text{if } v_1 = v_2 \\
\langle \varepsilon_*{}^{v_1:A} \mid (v_2, \circlearrowright) \mid \kappa \rangle & \overset{d}{\longmapsto}_{15} & \boxtimes & \text{if } v_1 \neq v_2
\end{array}
$$

Fig. 8. Additional reduction rules for the Π^d-abstract machine.

6.1 Abstract Machine Semantics

The syntax of Π^d extends the syntax of Π as follows:

$$
\begin{array}{llcl}
\textit{Value types} & A, B, C, D & ::= & \cdots \mid \mathbb{1}/v \\
\textit{Values} & v, w, x, y & ::= & \cdots \mid \circlearrowright \\
\textit{Program types} & & & t \leftrightarrow t \\
\textit{Programs} & c & ::= & \cdots \mid \eta_*{}^{v:A} : \mathbb{1} \leftrightarrow A \times \mathbb{1}/v : \varepsilon_*{}^{v:A}
\end{array}
$$

As discussed above, instead of η_* and ε_*, we introduce $\eta_*{}^{v:A}$ and $\varepsilon_*{}^{v:A}$ with the understanding that applying $\eta_*{}^{v:A}$ to tt produces a pair whose first component is v. The second component of this pair should be a value that can interact with v in a way that makes both values disappear. Given the association of $\times$ with space explained in Sec. 4, we posit that this new value is a garbage-collection (GC) process specialized to collect values equal to v. The fractional type $\mathbb{1}/v$ denotes such GC processes. Since all the information about fractional types is represented in the type, we just need a unique value $\circlearrowright$ for their runtime representation. When $\varepsilon_*{}^{v:A}$ is applied to a pair $(v', \circlearrowright)$, it checks if $v = v'$, and in that case applies the GC process annihilating both. If, however, $v \neq v'$, a runtime error will be raised. As evaluation might fail, the set of machine states includes a distinguished failure state.

Definition 26 (Π^d-machine states). *A Π^d-machine state σ is either:*

- *An enter state: $\langle c \mid v \mid \kappa \rangle$ where $c : A \leftrightarrow B$, $v : A$, and $\kappa : \textsc{cont}_{A \leftrightarrow B}$.*
- *A return state: $[c \mid v \mid \kappa]$ where $c : A \leftrightarrow B$, $v : B$, and $\kappa : \textsc{cont}_{A \leftrightarrow B}$.*
- *A fail state $\boxtimes$ which can make no progress.*

The transition rules for the Π^d-abstract machine are the following:

- All the twelve transition rules for the Π machine of Fig. 4.
- The three additional rules in Fig. 8. In rule $\overset{d}{\longmapsto}_{13}$ a new pair is created with the given value and a GC process. In rules $\overset{d}{\longmapsto}_{14}$ and $\overset{d}{\longmapsto}_{15}$ if the value reaching ε_* is the expected value then it is collected; otherwise the machine enters the failure state.

6.2 Properties

Because the Π^d semantics involves error states, termination for the abstract machine uses the more general Lem. 13 together with the usual requirement of the finiteness of the set of states reachable from an initial state. To use Lem. 13 we need to establish three properties: forward determinism, backward determinism on proper states, and verify that no progress is possible from failure states. The latter property holds by construction and the next two lemmas establish the required determinism properties.

Lemma 27 (Π^d-forward deterministic). *If $\sigma \overset{d}{\longmapsto} \sigma_1$ and $\sigma \overset{d}{\longmapsto} \sigma_2$ then $\sigma_1 = \sigma_2$*

Lemma 28 (Π^d-backward deterministic on proper states). *If $\sigma_1 \overset{m}{\longmapsto} \sigma$, $\sigma_2 \overset{m}{\longmapsto} \sigma$, and $\sigma \neq \boxtimes$ then $\sigma_1 = \sigma_2$*

Lemma 29 (Stuck). *If σ is stuck ($\nexists \sigma'.\sigma \overset{d}{\longmapsto} \sigma'$), then either $\sigma = [c \mid v \mid \square]$ or $\sigma = \boxtimes$.*

Definition 30 (Π^d-forward evaluation). *The function $\mathrm{eval}^d(c, _)$ is defined as follows:*

$$\mathrm{eval}^d(c, v_1) = \begin{cases} v_2 & \text{if } \langle c \mid v_1 \mid \square\rangle \overset{d\,*}{\longmapsto} [c \mid v_2 \mid \square] \\ \mathrm{error} & \text{if } \langle c \mid v_1 \mid \square\rangle \overset{d\,*}{\longmapsto} \boxtimes \end{cases}$$

Theorem 31 (Π^d-termination). *For all Π^d-combinators $c : A \leftrightarrow B$ and $v_1 : A$, either there exists v_2 such that $\mathrm{eval}^d(c, v_1) = v_2$ or $\mathrm{eval}^d(c, v_1) = \mathrm{error}$.*

According to the previous lemmas, the Π^d abstract machine is a partial reversible machine with every combinator inducing partial reversible function. As usual, we define $\overset{d\,\dagger}{\longmapsto}$ such that $\sigma_1 \overset{d\,\dagger}{\longmapsto} \sigma_2$ if $\sigma_2 \overset{d}{\longmapsto} \sigma_1$ and define backward evaluation.

Definition 32 (Π^d-backward evaluation). *We define $\mathrm{eval}^{d\dagger}$ as follows:*

$$\mathrm{eval}^{d\dagger}(c, v_1) = \begin{cases} v_2 & \text{if } [c \mid v_1 \mid \square] \overset{d\,\dagger *}{\longmapsto} \langle c \mid v_2 \mid \square\rangle \\ \mathrm{error} & \text{if } [c \mid v_1 \mid \square] \overset{d\,\dagger *}{\longmapsto} \left[\eta_*{}^{v_2:A} \mid (v_3, \circlearrowright) \mid \kappa\right] \quad \text{where } v_2 \neq v_3 \end{cases}$$

Theorem 33. *For all Π^d-combinators $c : A \leftrightarrow B$ and $v_1 : B$, either there exists v_2 such that $\mathrm{eval}^{d\dagger}(c, v_1) = v_2$ or $\mathrm{eval}^{d\dagger}(c, v_1) = \mathrm{error}$.*

Theorem 34. *For all $c : A \leftrightarrow B$, $v_1 : A$, and $v_2 : B$, we have $\mathrm{eval}^d(c, v_1) = v_2$ iff $\mathrm{eval}^{d\dagger}(c, v_2) = v_1$.*

6.3 Interpreter

The semantics of Π^d is particularly simple to define using a high-level interpreter of type:

$$\mathrm{interp}^d : (A \leftrightarrow B) \to A \to \mathrm{Maybe}\ B$$

The interpreter is a monadic version of the Π-interpreter in Fig. 5 extended with the following clauses:

$$\frac{}{\mathrm{interp}^d(\eta_*{}^{v:A}, \mathsf{tt}) \Downarrow \mathsf{just}\ (v, \circlearrowright)} \qquad \frac{v_1 = v_2}{\mathrm{interp}^d(\varepsilon_*{}^{v_1:A}, (v_2, \circlearrowright)) \Downarrow \mathsf{just}\ \mathsf{tt}} \qquad \frac{v_1 \neq v_2}{\mathrm{interp}^d(\varepsilon_*{}^{v_1:A}, (v_2, \circlearrowright)) \Downarrow \mathsf{nothing}}$$

Theorem 35 (Π^d-interpreter). *For all c, v_1, and v_2, $\mathrm{eval}^d(c, v_1) = v_2$ iff $\mathrm{interp}^d(c, v_1) \Downarrow$ just v_2 and $\mathrm{eval}^d(c, v_1) = \mathrm{error}$ iff $\mathrm{interp}^d(c, v_1) \Downarrow$ nothing.*

6.4 Compact Closed Category

Since η_* and ε_* are indexed by values, the operational semantics for fractionals does not form a compact closed category directly. Inspired by the relational models of geometry of interactions [Honsell and Lenisa 2004] we will build the appropriate category (groupoid actually) using pointed types.

Theorem 36. *We define the category of pointed types $(\mathcal{C}^\bullet, \times, \mathbb{1})$ as follows:*

- $\mathrm{Obj}(\mathcal{C}^\bullet)$ *are of the form (A, a) where A is a Π^d-type and $a : A$,*
- $\mathrm{Hom}((A, a), (B, b)) = c : A \leftrightarrow B$ *where $\mathrm{eval}^d(c, a) = b$, and*
- *dual objects $(\mathbb{1}/a, \circlearrowright)$ for every (A, a)*

The category $(\mathcal{C}^\bullet, \times, \mathbb{1})$ is a compact closed groupoid.

An immediate consequence of forming a compact closed category on multiplicative tensor is that morphisms become representable as values. Since the category we built is pointed, the morphisms are also *pointed*, mapping a particular $a : A$ to a particular $b : B$. Such a morphism can be represented as a value of type $\mathbb{1}/(a : A) \times B$. The same currying construction presented in the previous section also applies, converting the combinator on the left to the one on the right in the diagram below:

On the left, f expects a pair of some value of type A and $v : B$, it consumes both returning a value of type C. On the right, the consumption of $v : B$ is delayed by issuing a GC process that can collect it when it becomes available. From a programming perspective, this allows programs to be staged, consuming inputs as they become available.

7 COMBINING NEGATIVE AND FRACTIONAL TYPES: $\Pi^{\mathbb{Q}}$

A possible way to combine the languages Π^m and Π^d ensuring that the result is also a compact closed category is to simply take their product. The types of the new language would be pairs of types and the programs in the new language (morphisms in the new category) would be pairs of programs/morphisms acting pointwise [Clark et al. 2008]. This however would not enable interactions between backtracking and allocation/de-allocation. Our design, motivated by the operational semantics perspective is to combine the languages in a way that allows interactions of effects in the spirit of monad composition. The design is apparent in the types below:

$$\begin{array}{lcl} \mathrm{interp}^m & : & (A \leftrightarrow B) \to (\!|A^{\rightarrow}, B^{\leftarrow}|\!) \to (\!|B^{\rightarrow}, A^{\leftarrow}|\!) \\ \mathrm{interp}^d & : & (A \leftrightarrow B) \to A \to \mathrm{Maybe}\ B \\ \mathrm{interp}^{\mathbb{Q}} & : & (A \leftrightarrow B) \to (\!|A^{\rightarrow}, B^{\leftarrow}|\!) \to \mathrm{Maybe}\ ((\!|B^{\rightarrow}, A^{\leftarrow}|\!)) \end{array}$$

The type of $\mathrm{interp}^{\mathbb{Q}}$ suggests the following properties: (i) in the absence of memory effects, we recover the type of interp^m, (ii) in the absence of backtracking effects, we recover the type of interp^d, (iii) the combination of the effects is controlled by the Maybe monad, i.e., no backtracking is possible from an error state.

7.1 Abstract Machine Semantics

The syntax of $\Pi^{\mathbb{Q}}$ is given as follows:

$$\begin{array}{llcl} \textit{Value types} & A, B, C, D & ::= & \mathbb{0} \mid \mathbb{1} \mid A + B \mid A \times B \mid -A \mid \mathbb{1}/v \\ \textit{Values} & v, w, x, y & ::= & \mathsf{tt} \mid \mathsf{inj}_1\ v \mid \mathsf{inj}_2\ v \mid (v, v) \mid -v \mid \circlearrowright \\ \textit{Program types} & & & A \leftrightarrow B \\ \textit{Programs} & c & ::= & (\text{See Fig. 1}) \mid \eta_+ : \mathbb{0} \leftrightarrow A + (-A) : \varepsilon_+ \mid {\eta_*}^{v:A} : \mathbb{1} \leftrightarrow A \times \mathbb{1}/v : {\varepsilon_*}^{v:A} \end{array}$$

The set of machine states includes both directional and fail states.

Definition 37 ($\Pi^{\mathbb{Q}}$-machine states). *A $\Pi^{\mathbb{Q}}$-machine state σ is either:*

- *An enter state: $\langle c \mid v \mid \kappa \rangle_d$ where $c : A \leftrightarrow B$, $v : A$, $\kappa : \textsc{cont}_{A \leftrightarrow B}$, and $d \in \{\triangleright, \triangleleft\}$.*
- *A return state: $[c \mid v \mid \kappa]_d$ where $c : A \leftrightarrow B$, $v : B$, $\kappa : \textsc{cont}_{A \leftrightarrow B}$, and $d \in \{\triangleright, \triangleleft\}$.*
- *A fail state $\boxtimes$ which can make no progress.*

The transition rules for the $\Pi^{\mathbb{Q}}$-abstract machine are collected in Fig. 9. The rules include all the Π^m-rules without change. For the Π^d-rules, we include versions of them in both directions.

Include all the Π^m-rules.

$$\frac{\langle c \mid v \mid \kappa\rangle \mapsto_i \sigma}{\langle c \mid v \mid \kappa\rangle_\triangleright \overset{\mathbb{Q}}{\longmapsto}_i \sigma_\triangleright} \qquad \frac{[c \mid v \mid \kappa] \mapsto_i \sigma}{[c \mid v \mid \kappa]_\triangleright \overset{\mathbb{Q}}{\longmapsto}_i \sigma_\triangleright} \qquad \frac{\langle c \mid v \mid \kappa\rangle \mapsto_i^\dagger \sigma}{\langle c \mid v \mid \kappa\rangle_\triangleleft \overset{\mathbb{Q}}{\longmapsto}_{i^\dagger} \sigma_\triangleleft} \qquad \frac{[c \mid v \mid \kappa] \mapsto_i^\dagger \sigma}{[c \mid v \mid \kappa]_\triangleleft \overset{\mathbb{Q}}{\longmapsto}_{i^\dagger} \sigma_\triangleleft}$$

$$\begin{array}{rcl} \langle \epsilon_+ \mid \mathsf{inj}_1\ v \mid \kappa\rangle_\triangleright & \overset{\mathbb{Q}}{\longmapsto}_{12} & \langle \epsilon_+ \mid \mathsf{inj}_2\ (-v) \mid \kappa\rangle_\triangleleft \\ \langle \epsilon_+ \mid \mathsf{inj}_2\ (-v) \mid \kappa\rangle_\triangleright & \overset{\mathbb{Q}}{\longmapsto}_{13} & \langle \epsilon_+ \mid \mathsf{inj}_1\ v \mid \kappa\rangle_\triangleleft \end{array} \qquad \begin{array}{rcl} [\eta_+ \mid \mathsf{inj}_2\ (-v) \mid \kappa]_\triangleleft & \overset{\mathbb{Q}}{\longmapsto}_{14} & [\eta_+ \mid \mathsf{inj}_1\ v \mid \kappa]_\triangleright \\ [\eta_+ \mid \mathsf{inj}_1\ v \mid \kappa]_\triangleleft & \overset{\mathbb{Q}}{\longmapsto}_{15} & [\eta_+ \mid \mathsf{inj}_2\ (-v) \mid \kappa]_\triangleright \end{array}$$

Include all the Π^d-rules in the forward direction.

$$\begin{array}{rcll} \langle {\eta_*}^{v:A} \mid \mathsf{tt} \mid \kappa\rangle_\triangleright & \overset{\mathbb{Q}}{\longmapsto}_{16} & \left[{\eta_*}^{v:A} \mid (v, \circlearrowright) \mid \kappa\right]_\triangleright & \\ \langle {\varepsilon_*}^{v_1:A} \mid (v_2, \circlearrowright) \mid \kappa\rangle_\triangleright & \overset{\mathbb{Q}}{\longmapsto}_{17} & \left[{\varepsilon_*}^{v_1:A} \mid \mathsf{tt} \mid \kappa\right]_\triangleright & \text{if } v_1 = v_2 \\ \langle {\varepsilon_*}^{v_1:A} \mid (v_2, \circlearrowright) \mid \kappa\rangle_\triangleright & \overset{\mathbb{Q}}{\longmapsto}_{18} & \boxtimes & \text{if } v_1 \neq v_2 \end{array}$$

Include all the Π^d-rules in the backward direction.

$$\begin{array}{rcll} \left[{\eta_*}^{v_1:A} \mid (v_2, \circlearrowright) \mid \kappa\right]_\triangleleft & \overset{\mathbb{Q}}{\longmapsto}_{19} & \langle {\eta_*}^{v_1:A} \mid \mathsf{tt} \mid \kappa\rangle_\triangleleft & \text{if } v_1 = v_2 \\ \left[{\eta_*}^{v_1:A} \mid (v_2, \circlearrowright) \mid \kappa\right]_\triangleleft & \overset{\mathbb{Q}}{\longmapsto}_{20} & \boxtimes & \text{if } v_1 \neq v_2 \\ \left[{\varepsilon_*}^{v:A} \mid \mathsf{tt} \mid \kappa\right]_\triangleleft & \overset{\mathbb{Q}}{\longmapsto}_{21} & \langle {\varepsilon_*}^{v:A} \mid (v, \circlearrowright) \mid \kappa\rangle_\triangleleft & \end{array}$$

Fig. 9. $\Pi^{\mathbb{Q}}$-abstract machine

7.2 Properties

We establish the same properties as for the other interpreters: evaluation is deterministic, terminating, and partially reversible.

Lemma 38 ($\Pi^{\mathbb{Q}}$-forward deterministic). *If* $\sigma \overset{\mathbb{Q}}{\longmapsto} \sigma_1$ *and* $\sigma \overset{\mathbb{Q}}{\longmapsto} \sigma_2$ *then* $\sigma_1 = \sigma_2$

Lemma 39 ($\Pi^{\mathbb{Q}}$-backward deterministic on proper states). *If* $\sigma_1 \overset{\mathbb{Q}}{\longmapsto} \sigma$, $\sigma_2 \overset{\mathbb{Q}}{\longmapsto} \sigma$, *and* $\sigma \neq \boxtimes$ *then* $\sigma_1 = \sigma_2$

Lemma 40 (Stuck). *If* σ *is stuck* ($\nexists \sigma'.\sigma \overset{\mathbb{Q}}{\longmapsto} \sigma'$), *then either* $\sigma = \langle c \mid v \mid \square\rangle_\triangleleft$, $\sigma = [c \mid v \mid \square]_\triangleright$, *or* $\sigma = \boxtimes$.

Definition 41 ($\Pi^{\mathbb{Q}}$-forward evaluation). *We define:*

$$\mathrm{eval}^{\mathbb{Q}} : (A \leftrightarrow B) \to (\!|A^{\rightarrow}, B^{\leftarrow}|\!) \to (\!|B^{\rightarrow}, A^{\leftarrow}|\!) \uplus \{\mathrm{error}\}$$

as follows:

$$\mathrm{eval}^{\mathbb{Q}}(c, v : A^{\rightarrow}) = \begin{cases} w : B^{\rightarrow} & \text{if } \langle c \mid v \mid \square\rangle_\triangleright \overset{\mathbb{Q}\,*}{\longmapsto} [c \mid w \mid \square]_\triangleright \\ w : A^{\leftarrow} & \text{if } \langle c \mid v \mid \square\rangle_\triangleright \overset{\mathbb{Q}\,*}{\longmapsto} \langle c \mid w \mid \square\rangle_\triangleleft \\ \mathrm{error} & \text{if } \langle c \mid v \mid \square\rangle_\triangleright \overset{\mathbb{Q}\,*}{\longmapsto} \boxtimes \end{cases}$$

$$\mathrm{eval}^{\mathbb{Q}}(c, v : B^{\leftarrow}) = \begin{cases} w : B^{\rightarrow} & \text{if } [c \mid v \mid \square]_\triangleleft \overset{\mathbb{Q}\,*}{\longmapsto} [c \mid w \mid \square]_\triangleright \\ w : A^{\leftarrow} & \text{if } [c \mid v \mid \square]_\triangleleft \overset{\mathbb{Q}\,*}{\longmapsto} \langle c \mid w \mid \square\rangle_\triangleleft \\ \mathrm{error} & \text{if } [c \mid v \mid \square]_\triangleleft \overset{\mathbb{Q}\,*}{\longmapsto} \boxtimes \end{cases}$$

Evaluation can start in either direction and end in either direction.

Theorem 42 ($\Pi^{\mathbb{Q}}$-termination). *For all* $\Pi^{\mathbb{Q}}$*-combinators* $c : A \leftrightarrow B$ *and* $v_1 : (\!|A^{\rightarrow}, B^{\leftarrow}|\!)$, *either there exists* $v_2 : (\!|B^{\rightarrow}, A^{\leftarrow}|\!)$ *such that* $\mathrm{eval}^{\mathbb{Q}}(c, v_1) = v_2$ *or* $\mathrm{eval}^{\mathbb{Q}}(c, v_1) = \mathrm{error}$.

The language $\Pi^{\mathbb{Q}}$ is also partially reversible.

Definition 43 ($\Pi^{\mathbb{Q}}$-backward evaluation). *We define:*

$$\mathrm{eval}^{\mathbb{Q}\dagger} : (A \leftrightarrow B) \to (\!| B^{\rightarrow}, A^{\leftarrow} |\!) \to (\!| A^{\rightarrow}, B^{\leftarrow} |\!) \uplus \{\mathrm{error}\}$$

as follows:

$$\mathrm{eval}^{\mathbb{Q}\dagger}(c, v : B^{\rightarrow}) = \begin{cases} w : A^{\rightarrow} & \text{if } [c \mid v \mid \square]_{\triangleleft} \overset{\mathbb{Q}\,*}{\longmapsto} \langle c \mid w \mid \square \rangle_{\triangleleft} \\ w : B^{\leftarrow} & \text{if } [c \mid v \mid \square]_{\triangleleft} \overset{\mathbb{Q}\,*}{\longmapsto} [c \mid w \mid \square]_{\triangleright} \\ \mathrm{error} & \text{if } [c \mid v \mid \square]_{\triangleleft} \overset{\mathbb{Q}\,*}{\longmapsto} \boxtimes \end{cases}$$

$$\mathrm{eval}^{\mathbb{Q}\dagger}(c, v : A^{\leftarrow}) = \begin{cases} w : A^{\rightarrow} & \text{if } \langle c \mid v \mid \square \rangle_{\triangleright} \overset{\mathbb{Q}\,*}{\longmapsto} \langle c \mid w \mid \square \rangle_{\triangleleft} \\ w : B^{\leftarrow} & \text{if } \langle c \mid v \mid \square \rangle_{\triangleright} \overset{\mathbb{Q}\,*}{\longmapsto} [c \mid w \mid \square]_{\triangleright} \\ \mathrm{error} & \text{if } \langle c \mid v \mid \square \rangle_{\triangleright} \overset{\mathbb{Q}\,*}{\longmapsto} \boxtimes \end{cases}$$

Evaluation can start in either direction and end in either direction.

Theorem 44. *For all $c : A \leftrightarrow B$, $V : (\!| A^{\rightarrow}, B^{\leftarrow} |\!)$, and $W : (\!| B^{\rightarrow}, A^{\leftarrow} |\!)$, we have $\mathrm{eval}^{\mathbb{Q}}(c, V) = W$ iff $\mathrm{eval}^{\mathbb{Q}\dagger}(c, W) = V$.*

7.3 Interpreter

The complete interpreter for $\Pi^{\mathbb{Q}}$ is in the accompanying Agda code. It is essentially the Π^{m}-interpreter modified as follows. First, all operations are lifted to the exception monad. Second, now that the multiplicative fragment is also subject to backtracking, the interpreter is extended with clauses for η_* and ε_* in both directions.

8 PROGRAMMING WITH NEGATIVE AND FRACTIONAL TYPES

We present a number of small examples illustrating the expressiveness of $\Pi^{\mathbb{Q}}$. Most of the examples also lead to the implementation of the SAT solver. Because the examples get progressively larger and more complex, the presentation in this section uses executable Agda code, which is kept quite similar to the mathematical presentation used so far. In this context, Agda is used as a "macro processor" that generates $\Pi^{\mathbb{Q}}$ combinators from given parameters and then as an interpreter to execute the resulting combinators. The code omits some of the boilerplate definitions which can be found in the supplement.

Patterns and Data Definitions. The universe of $\Pi^{\mathbb{Q}}$-types is called $\mathbb{U}$ in the Agda code. For convenience, we define abbreviations of booleans and products of booleans:

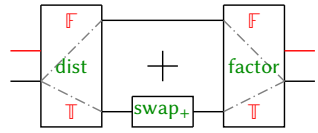
```
pattern 𝔹 = 𝟙 +u 𝟙
pattern 𝔽 = inj1 tt
pattern 𝕋 = inj2 tt

𝔹^ : ℕ → 𝕌
𝔹^ 0 = 𝟙
𝔹^ 1 = 𝔹
𝔹^ (suc (suc n)) = 𝔹 ×u 𝔹^ (suc n)
```

Reversible Conditionals and Reversible Copy. Some of the basic abstractions developed in the early proposals of reversible computing [Toffoli 1980] are the ability to perform conditional execution of combinators and to make copies of data. We first define CNOT, an important gate in the context of reversible and quantum computation. As shown in the diagram below, the combinator CNOT takes a pair booleans with the first one acting as a "control bit" and the second

one acting as a "target bit." If the control bit is true (i.e., if we are in the right injection after distributing), then we negate the target bit (by swapping the left and right injections). Otherwise, no action is taken and the gate acts as the identity. If we execute CNOT on input $(\mathbb{T}, \mathbb{F})$ which we recall is an abbreviation for $(\mathsf{inj}_2\ \mathsf{tt}, \mathsf{inj}_1\ \mathsf{tt})$, the evaluation terminates with $(\mathbb{T}, \mathbb{T})$.

```
CNOT : 𝔹 ×u 𝔹 ↔ 𝔹 ×u 𝔹
CNOT = dist ⨾ (id↔ ⊕ (id↔ ⊗ swap+)) ⨾ factor
```



The combinator CIF generalizes CNOT to define a reversible if-expression.[4]

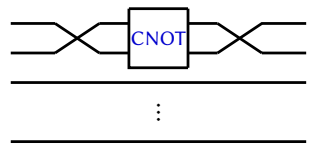
```
CIF : {A : 𝕌} → (c1 c2 : A ↔ A) → 𝔹 ×u A ↔ 𝔹 ×u A
CIF c1 c2 = dist ⨾ ((id↔ ⊗ c1) ⊕ (id↔ ⊗ c2)) ⨾ factor
```

The combinator TOFFOLI' is an iterated controlled operation that flips the last bit if all the previous bits are false. (TOFFOLI'$(b_1, \ldots, b_n, b) = (b_1, \ldots, b_n, b\ xor\ (\neg b_1 \wedge \ldots \wedge \neg b_n)))$.

```
TOFFOLI' : {n : ℕ} → 𝔹^ n ↔ 𝔹^ n
TOFFOLI' {0} = id↔
TOFFOLI' {1} = swap+
TOFFOLI' {suc (suc n)} = CIF TOFFOLI' id↔
```

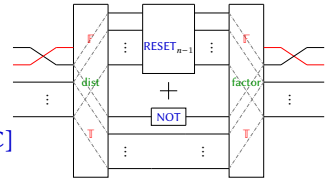
An interesting observation going back to Toffoli [1980] is that, when the target bit of CNOT is set to $\mathbb{F}$, it performs a reversible copy. Indeed, the input $(\mathbb{F}, \mathbb{F})$ is mapped to $(\mathbb{F}, \mathbb{F})$ and the input $(\mathbb{T}, \mathbb{F})$ is mapped to $(\mathbb{T}, \mathbb{T})$. The combinator COPY (shown below) generalizes this idea to perform the following operation COPY$(\mathbb{F}, b_1, \ldots, b_n) = (b_1, b_1, \ldots, b_n)$ which copies the first bit in a tuple.

```
COPY : ∀ {n} → 𝔹 ×u 𝔹^ n ↔ 𝔹 ×u 𝔹^ n
COPY {0} = id↔
COPY {1} = swap* ⨾ CNOT ⨾ swap*
COPY {suc (suc n)} = assocl* ⨾ (COPY ⊗ id↔) ⨾ assocr*
```



The combinator RESET is also a controlled operation modulo some re-arranging of values. Its semantics can be summarized as follows: RESET$(b, b_1, \ldots, b_n) = (b \veebar (\bigvee_{i=1}^{n} b_i), b_1, \ldots, b_n)$ where $\vee$ is logical-or and $\veebar$ is exclusive-or. As shown in the last line of the definition, as long as we are in the first branch of the conditional (i.e., the current bit is false), we keep calling RESET until we reach the last bit.

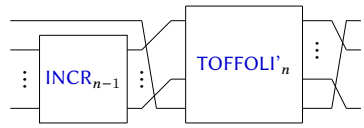
```
RESET : ∀ {n} → 𝔹 ×u 𝔹^ n ↔ 𝔹 ×u 𝔹^ n
RESET {0} = id↔
RESET {1} = swap* ⨾ CNOT ⨾ swap*
RESET {suc (suc n)} =
  Ax[BxC]=Bx[AxC] ⨾ CIF RESET (swap+ ⊗ id↔) ⨾ Ax[BxC]=Bx[AxC]
```



Arithmetic. Given the ability of perform conditional operations on bits, it is relatively straightforward to implement reversible arithmetic operations. The program below increments an n-bit binary number (without carry). The definition of INCR is by cases: incrementing 1 bit without carry just flips the bit using swap_+. For a binary number with $n > 1$ bits, we first run INCR on the lower $n - 1$ bits, and if the result is a collection of bits that are all $\mathbb{F}$, we flip the highest bit.

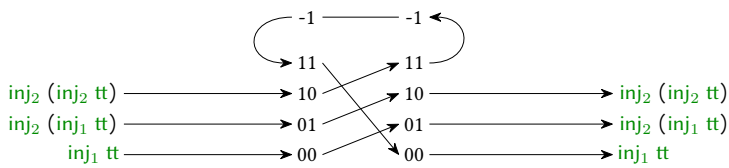
[4] Arguments within braces $\{\cdot\}$ are implicit arguments that can often be automatically inferred by Agda.

```
INCR : ∀ {n} → 𝔹^ n ↔ 𝔹^ n
INCR {0} = id↔
INCR {1} = swap₊
INCR {suc (suc n)} =
  (id↔ ⊗ INCR) ⨾ FST2LAST ⨾ TOFFOLI' ⨾ FST2LAST⁻¹
```



Data Structures. Negative types can be used to build expressive and more efficient data structures. Consider a situation in which we have an enumeration consisting of exactly 2047 elements. Two extreme representations would be to use an inefficient "wide" sum type or a product type of 11 booleans with some informal convention that one of the elements is unused. With negative types, we can use a type "$2^{11} - 1$" that uses the efficient product type while also formally enforcing that one of the elements is unused. The advantage of the "$2^{11} - 1$" representation over the wide sum type becomes apparent when writing functions that manipulate either type. For example, as shown in the accompanying code, incrementing a value represented in the additive representation takes 32721 machine transition steps whereas the same operation over the "$2^{11} - 1$" representation only takes 3453 steps.

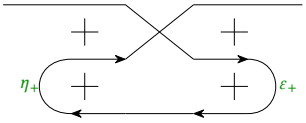
To more clearly see the role that -1 plays in this setting, let's consider a smaller example in which we represent a type with three elements as a pair of booleans (representing values 00, 01, 10, and 11) minus one element (11) leaving only three values (00, 01, and 10) in the type. The goal is to write a function that increments the three values mapping $00 \mapsto 01$, $01 \mapsto 10$, and $10 \mapsto 00$. This can be achieved using INCR : $\mathbb{B}^2 \leftrightarrow \mathbb{B}^2$ as shown in the diagram below:



When given the inputs 00 and 01, INCR performs the desired action. But when given input 10, INCR produces the excluded element 11: this triggers a backtracking action feeding the value 11 back into INCR producing 00 as the final result.[5]

Control Flow. The example from the introduction can now be executed in both the additive and multiplicative fragments. We show the additive execution which relies on backtracking.

```
zigzag : 𝔹 ↔ 𝔹
zigzag =
  uniti₊l ⨾ (η₊ ⊕ id↔) ⨾ [A+B]+C=[C+B]+A ⨾ (ε₊ ⊕ id↔) ⨾ unite₊l
```



Execution starts in the forward direction. When $\eta_+ \oplus$ id↔ is first encountered, evaluation proceeds to the right following id↔ until it reaches $\varepsilon_+ \oplus$ id↔. At that point evaluation reverses with a negative value executing the machine transitions in reverse. Eventually, evaluation reaches $\eta_+ \oplus$ id↔ again but this time in the reverse direction. Evaluation then flips direction again and terminates with the same input we started with. Indeed this circuit implements one of the coherence conditions for compact closed categories which essentially states that η_+ following by ε_+ is the identity. This trace confirms the intuition that the first occurrence of η_+ is transparent in the forward direction but is used to mark a backtracking point.

[5]In general, using negative types, we can build an n-element wrap-around counter using an $n + k$ wrap-around counter for arbitrary k.

Iteration. We show how to implement a **for**-loop using negative types. We start with the trace operator definable in compact closed categories:

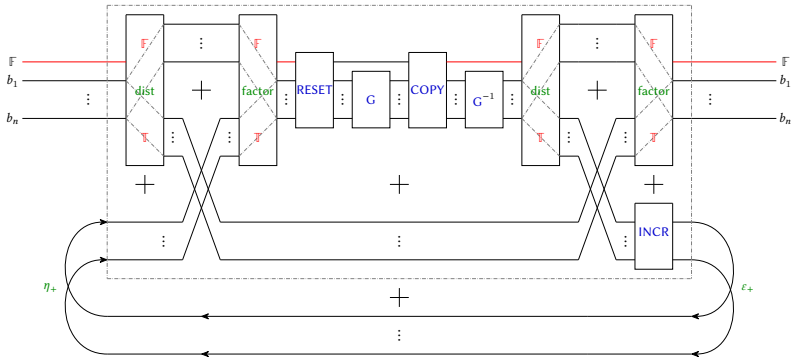
$\mathsf{trace}_+ : \forall \{A\ B\ C\} \longrightarrow (A +_u C \leftrightarrow B +_u C) \longrightarrow A \leftrightarrow B$

$\mathsf{trace}_+\ f = \mathsf{uniti}_+\mathsf{r} \fatsemi (\mathsf{id}\!\leftrightarrow \oplus\ \eta_+) \fatsemi \mathsf{assocl}_+ \fatsemi (f \oplus \mathsf{id}\!\leftrightarrow) \fatsemi \mathsf{assocr}_+ \fatsemi (\mathsf{id}\!\leftrightarrow \oplus\ \varepsilon_+) \fatsemi \mathsf{unite}_+\mathsf{r}$

Given an input a, the execution of $(\mathsf{trace}_+ f)$ starts by applying f to $\mathsf{inj}_1\ a$. The result of f can either be of the form $\mathsf{inj}_1\ b$ or $\mathsf{inj}_2\ c$. In the first case, evaluation terminates with result b. In the second case, evaluation proceeds backwards re-entering f with $\mathsf{inj}_2\ c$ and repeating the process. Since f is injective and has a finite graph, it must eventually return a value of the form $\mathsf{inj}_1\ b$ terminating the iteration. Since execution is guaranteed to terminate, the number of copies of f is bounded and the trace operator realizes the functionality of a bounded **for**-loop. We make this connection concrete by implementing a combinator that behaves like the following loop written in pseudo-code:

$$\textbf{for}\ (\overline{b} = \overline{\mathbb{F}};\ G(\overline{b}) = (\mathbb{T}, _);\ \overline{b}{+}{+});$$

The loop maintains a tuple of booleans representing a binary number that starts at 0 and is incremented at each step using the circuit INCR. In each iteration the current binary number is passed to a function G and the loop terminates when the first bit of G's return value is $\mathbb{T}$. The corresponding $\Pi^{\mathbb{Q}}$ program will take the form $\mathsf{LOOP}(G)(\mathbb{F}, \mathbb{F}, \ldots, \mathbb{F})$ where LOOP takes a n-bit reversible function to construct a $n+1$-bit reversible function. To maintain reversibility, we use a common design pattern of reversible programs: compute-copy-uncompute [Bennett 1973; Toffoli 1980]. In more detail, in each iteration, we compute $G(\overline{b})$, copy the result to an auxiliary wire using COPY, and then run G backwards to uncompute its result restoring the original $\overline{b}$. After the copy operation and uncomputing the action of G, the auxiliary wire still holds the copied value and can be used to decide whether to terminate or continue the loop. If the loop continues, we use RESET to reset the auxiliary wire to $\mathbb{F}$ in preparation for the next copy operation.



$\mathsf{LOOP} : \forall \{n\} \longrightarrow \mathbb{B}\hat{}\ n \leftrightarrow \mathbb{B}\hat{}\ n \longrightarrow \mathbb{B} \times_u \mathbb{B}\hat{}\ n \leftrightarrow \mathbb{B} \times_u \mathbb{B}\hat{}\ n$

$\mathsf{LOOP}\ \{0\}\ G = \mathsf{id}\!\leftrightarrow$

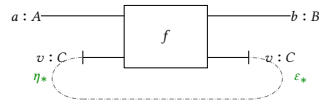
$\mathsf{LOOP}\ \{1\}\ G = \mathsf{id}\!\leftrightarrow \otimes\ G$

$\mathsf{LOOP}\ \{\mathsf{suc}\ (\mathsf{suc}\ n)\}\ G = \mathsf{trace}_+\ (\ (\mathsf{dist} \oplus \mathsf{id}\!\leftrightarrow) \fatsemi [\mathsf{A{+}B}]{+}\mathsf{C}{=}[\mathsf{A{+}C}]{+}\mathsf{B} \fatsemi (\mathsf{factor} \oplus \mathsf{id}\!\leftrightarrow) \fatsemi$
$((\mathsf{RESET} \fatsemi (\mathsf{id}\!\leftrightarrow \otimes\ G) \fatsemi \mathsf{COPY} \fatsemi (\mathsf{id}\!\leftrightarrow \otimes\ \mathsf{G}^{-1})) \oplus \mathsf{id}\!\leftrightarrow) \fatsemi$
$(\mathsf{dist} \oplus \mathsf{id}\!\leftrightarrow) \fatsemi [\mathsf{A{+}B}]{+}\mathsf{C}{=}[\mathsf{A{+}C}]{+}\mathsf{B} \fatsemi (\mathsf{factor} \oplus (\mathsf{id}\!\leftrightarrow \otimes\ \mathsf{INCR}))\ )$

We note that it is possible to build an equivalent circuit in plain Π but at the cost of replicating 2^n copies of the circuit controlled by the trace operator.

Allocation and De-Allocation. The multiplicative version of the trace operator has a similar definition to the additive one but has fundamentally different behavior:

$\mathsf{trace}_* : \forall \{A\ B\ C\} \to [\![\ C\ ]\!] \to (A \times_u C \leftrightarrow B \times_u C) \to A \leftrightarrow B$
$\mathsf{trace}_*\ v\ f =$
$\quad \mathsf{uniti}_*\mathsf{r} \fatsemi (\mathsf{id}\!\leftrightarrow \otimes\ \eta_\times\ v) \fatsemi \mathsf{assocl}_* \fatsemi (f \otimes \mathsf{id}\!\leftrightarrow) \fatsemi$
$\quad \mathsf{assocr}_* \fatsemi (\mathsf{id}\!\leftrightarrow \otimes\ \varepsilon_\times\ v) \fatsemi \mathsf{unite}_*\mathsf{r}$



Since η_* and ε_* take a value argument, the entire trace operator takes a v which should be a fixed point of f. A small example can illustrate this point. Let f be swap_* and consider applying $\mathsf{trace}_*\ \mathbb{F}\ \mathsf{swap}_*$ to some value a. During this evaluation swap_* is applied to $(a, \mathbb{F})$ and returns $(\mathbb{F}, a)$. If $a \neq \mathbb{F}$, execution will raise an error at ε_*.
The multiplicative trace operator can be used to allocate a temporary (ancilla) value for the duration of the execution f and safely de-allocate on exit from f under the condition that it has been restored to its initial value [Chen et al. 2020; Green et al. 2013; Thomsen et al. 2015]. This space management approach should be contrasted with the standard approach in programming languages in which allocation would have type: $\mathrm{alloc} : \mathbb{1} \to A$. Attempting to directly use this type for allocation, and adding an inverse of type $A \to \mathbb{1}$ for de-allocation completely destroys the foundation of the language based on type isomorphisms and would invalidate much of the basic reasoning principles in the language. For example, the type system would no longer be able to match allocations with de-allocations allowing terms like $\mathsf{uniti}_*\mathsf{l} \fatsemi (\mathsf{id}\!\leftrightarrow \otimes\ \mathrm{alloc}) : \mathbb{1} \leftrightarrow \mathbb{B} \times \mathbb{B}$ which leak memory.

Higher-Order Combinators. As explained during the construction of compact closed categories, each extension supports a notion of higher-order functions. A combinator can be converted to a higher-order value and then these values can be composed, curried, and applied. We only show the signatures of the combinators:

$_\multimap_+_ : (A\ B : \mathbb{U}) \to \mathbb{U}$
$A \multimap_+ B = \text{-}\ A +_u B$

$_\multimap_*_ : \{A : \mathbb{U}\} \to (v : [\![\ A\ ]\!]) \to (B : \mathbb{U}) \to \mathbb{U}$
$v \multimap_* B = \mathbb{1}/\ v \times_u B$

$\mathsf{hof\text{-}} : \{A\ B : \mathbb{U}\} \to$
$\quad (A \leftrightarrow B) \to (\mathbb{0} \leftrightarrow A \multimap_+ B)$

$\mathsf{hof/} : \{A\ B : \mathbb{U}\} \to$
$\quad (A \leftrightarrow B) \to (v : [\![\ A\ ]\!]) \to (\mathbb{1} \leftrightarrow v \multimap_* B)$

$\mathsf{comp\text{-}} : \{A\ B\ C : \mathbb{U}\} \to$
$\quad (A \multimap_+ B) +_u (B \multimap_+ C) \leftrightarrow (A \multimap_+ C)$

$\mathsf{comp/} : \{A\ B\ C : \mathbb{U}\}\ \{v : [\![\ A\ ]\!]\} \to (w : [\![\ B\ ]\!]) \to$
$\quad (v \multimap_* B) \times_u (w \multimap_* C) \leftrightarrow (v \multimap_* C)$

$\mathsf{curry\text{-}} : \{A\ B\ C : \mathbb{U}\} \to$
$\quad (A +_u B \leftrightarrow C) \to (A \leftrightarrow B \multimap_+ C)$

$\mathsf{curry/} : \{A\ B\ C : \mathbb{U}\} \to$
$\quad (A \times_u B \leftrightarrow C) \to (v : [\![\ B\ ]\!]) \to (A \leftrightarrow v \multimap_* C)$

$\mathsf{app\text{-}} : \{A\ B : \mathbb{U}\} \to (A \multimap_+ B) +_u A \leftrightarrow B$

$\mathsf{app/} : \{A\ B : \mathbb{U}\} \to (v : [\![\ A\ ]\!]) \to (v \multimap_* B) \times_u A \leftrightarrow B$

As an example, $(\mathsf{curry\text{-}}\ \mathsf{swap}_+)$ is a combinator of type $\mathbb{B} \leftrightarrow -\mathbb{B}+(\mathbb{B}+\mathbb{B})$. One may think of swap_+ as a combinator ready to receive either a left value or a right value and think of $(\mathsf{curry\text{-}}\ \mathsf{swap}_+)$ as a combinator only receiving the original left value as a regular input; the curried combinator can also receive the original right value by demanding it from the output side. In contrast, $(\mathsf{curry/}\ \mathsf{CNOT}\ \mathbb{T})$ is a combinator of type $\mathbb{B} \leftrightarrow \mathbb{1}/\mathbb{T} \times (\mathbb{B} \times \mathbb{B})$. The fractional part of the type reveals that CNOT has already been partially applied to a $\mathbb{T}$ target bit. When given a control bit b, the combinator behaves like $\mathsf{CNOT}\ (b, \mathbb{T})$.

Algebraic Identities from the Field of Rational Numbers. Any equation that is valid in the field of rational numbers has a computational interpretation in $\Pi^{\mathbb{Q}}$. For example, since $\frac{3}{2} - \frac{1}{2} = 1$, we can write four different programs with types that realize this identity (depending on whether we use $\mathbb{F}$ or $\mathbb{T}$ to represent the fraction $\frac{1}{2}$). The supplementary package includes definitions for a number of combinators whose types are identities in the field of rational numbers. We only highlight a few signatures from the code:

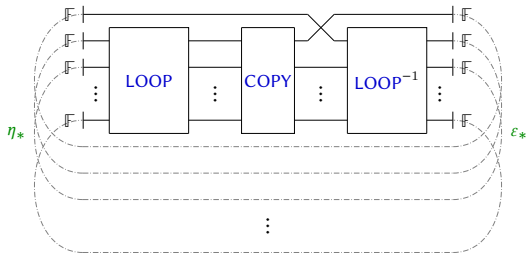


Fig. 10. Structure of SAT solver

- Reversing execution flow twice is a no-op:
  $\mathsf{inv\text{-}} : \{A : \mathbb{U}\} \to A \leftrightarrow \text{-}\,(\text{-}\,A)$
- Garbage collecting the garbage collector for v rematerializes v:
  $\mathsf{inv/} : \{A : \mathbb{U}\}\,\{v : [\![\, A \,]\!]\} \to A \leftrightarrow (\mathbb{1}/_\ \{\mathbb{1}/\ v\}\ \circlearrowleft)$
- A garbage collector for collecting a pair is a pair of garbage collectors:
  $\mathsf{dist/} : \{A\ B : \mathbb{U}\}\,\{a : [\![\, A \,]\!]\}\,\{b : [\![\, B \,]\!]\} \to \mathbb{1}/\ (a\ ,\ b) \leftrightarrow \mathbb{1}/\ a \times_u \mathbb{1}/\ b$
- If one component of a pair reverses direction, the entire pair reverses:
  $\mathsf{dist\times\text{-}} : \{A\ B : \mathbb{U}\} \to (\text{-}\,A) \times_u B \leftrightarrow \text{-}\,(A \times_u B)$
- A choice between two multiplicative functions is a multiplicative function taking a pair. Expanding the definition of $\multimap_*$, this is just the formula for summing two fractions: $\frac{A}{v} + \frac{B}{w} = \frac{Aw+Bv}{vw}$:
  $\mathsf{addFrac} : \forall\,\{A\ B\ C\ D\} \to (v : [\![\, C \,]\!]) \to (w : [\![\, D \,]\!]) \to$
  $(v \multimap_* A) +_u (w \multimap_* B) \leftrightarrow (v\ ,\ w) \multimap_* ((A \times_u D) +_u (C \times_u B))$

Reversible Backtracking Search: A SAT Solver. The previous abstractions just need to be assembled to implement a SAT solver in $\Pi^{\mathbb{Q}}$. Our SAT solver checks the satisfiability for a given function $G : \mathbb{B}^n \to \mathbb{B}$, i.e., it succeeds if $\exists \bar{b}.G(\bar{b}) = \mathbb{T}$. The first step is to use the standard Toffoli construction [1980] to construct $G^r : \mathbb{B}^{1+n} \to \mathbb{B}^{1+n}$ which is a reversible version of G satisfying $G^r(\mathbb{F}, \bar{b}) = (G(\bar{b}), _)$. In our implementation it turns out to be simpler to negate G^r and checks the unsatisfiability. Below we use $\sim G^r = G^r \fatsemi (\mathsf{NOT} \otimes \mathsf{id}\leftrightarrow)$ as a reversible version of $\mathsf{NOT} \circ G$ satisfying $\exists \bar{b}.\ \sim G^r(\bar{b}) = (\mathbb{F}, \ldots)$ iff $\exists \bar{b}.G(\bar{b}) = \mathbb{T}$. The executable definition takes an instance of a SAT problem and returns a combinator of type $\mathbb{1} \leftrightarrow \mathbb{1}$. When applied to tt this combinator either returns tt indicating success or throws an exception:[6]

```
SAT : ∀ {n} → 𝔹^ (1 + n) ↔ 𝔹^ (1 + n) → 𝟙 ↔ 𝟙
SAT {n} Gʳ = trace* (𝔽^ (3 + n)) (id↔ ⊗ ( (id↔ ⊗ (LOOP (~ Gʳ) ⨾ (id↔ ⊗ SPLIT 1 n))) ⨾
                                          (id↔ ⊗ (assocl* ⨾ (COPY ⊗ id↔) ⨾ assocr*)) ⨾
                                          assocl* ⨾ (swap* ⊗ id↔) ⨾ assocr* ⨾
                                          (id↔ ⊗ ((id↔ ⊗ MERGE 1 n) ⨾ LOOP⁻¹ (~ Gʳ)))) )
  where
  MERGE : (n m : ℕ) → 𝔹^ n ×u 𝔹^ m ↔ 𝔹^ (n + m)
  SPLIT : (n m : ℕ) → 𝔹^ (n + m) ↔ 𝔹^ n ×u 𝔹^ m
```

Other than the previously-constructed combinators for the multiplicative trace, COPY, and LOOP, the implementation only uses primitive combinators and two easy combinators to split and merge

[6] It would be possible to reformulate the SAT solver to return a boolean result instead of an exception. One possibility is to leave the ancilla bit as part of the result instead of GC'ing it. This would however also keep the associated GC token as part of the result.

tuples of booleans. As shown in Fig. 10, the general structure of the solver is to use LOOP to iterate over boolean assignments to find smallest $\overline{b}$ such that $\sim G^r(\overline{b}) = (\mathbb{F}, _)$, using COPY to copy the first bit of $\overline{b}$ to an auxiliary wire, and running LOOP^{-1} to uncompute $\overline{b}$ back to $\overline{\mathbb{F}}$. If the first bit of $\overline{b}$ is $\mathbb{F}$, the GC process succeeds, otherwise it throws an exception. This construction is correct because if $\overline{b} = (\mathbb{F}, \overline{x})$ then $\sim G^r(\overline{x}) = \mathbb{F}$ which implies $G(\overline{x}) = \mathbb{T}$. Otherwise since LOOP starts from $\mathbb{F}^{1+n}$ so $\forall \overline{x}. \sim G^r(\overline{x}) = \mathbb{T}$ which implies $\forall \overline{x}. G(\overline{x}) = \mathbb{F}$.
As a small example, consider $G(a, b) = (a \wedge b) \veebar (a \vee b)$ (where $\wedge$ is logical-and, $\vee$ is logical-or, and $\veebar$ is exclusive-or). The SAT solver accepts the reversible version of G satisfying $G^r(\mathbb{F}, b_1, b_2) = (G(b_1, b_2), _, _)$ and executes as follows: (i) use the multiplicative trace to initialize 2+3 ancilla $\mathbb{F}$ bits (1 for result, 1 for temporary storage in LOOP and n for inputs); (ii) use LOOP that iterates over all possible values of (b_0, b_1, b_2) starting from $(\mathbb{F}, \mathbb{F}, \mathbb{F})$, which results in (b_0', b_1', b_2') such that $G^r(b_0, b_1, b_2) = (\mathbb{F}, _, _)$ ($(\mathbb{F}, \mathbb{F}, \mathbb{T})$ in this case); (iii) copy b_0' to the unused auxiliary wire; (iv) uncompute LOOP which takes (b_0', b_1', b_2') back to $(\mathbb{F}, \mathbb{F}, \mathbb{F})$; (v) the GC succeeds since the b_0' we copied is $\mathbb{F}$. The following are some additional small examples:

```
-- Ex1(𝔽,a,b) = ((a∧b) xor (a∧b),_,_)
SAT-Ex1 = eval' (SAT Ex1) tt -- nothing

-- Ex2(𝔽,a,b) = ((a∧b) xor (a∨b),_,_)
SAT-Ex2 = eval' (SAT Ex2) tt -- just tt

-- Ex3(𝔽,a,b) = (((a∧b) ∧ (a xor b)),_,_)
SAT-Ex3 = eval' (SAT Ex3) tt -- nothing

-- Ex4(𝔽,a,b) = (((a∨b) ∧ (a xor b)),_,_)
SAT-Ex4 = eval' (SAT Ex4) tt -- just tt
```

9 CONCLUSION AND FUTURE WORK

We have constructed computational models of compact closed categories by proposing that negative and fractional types express backtracking in "time" and "space." Significantly we have demonstrated that negative and fractional types have applications in programming, enriching reversible programming languages with facilities for allocation, de-allocation, exceptions, and backtracking. The salient aspects of our approach are:

- Negative and fractional types have an elementary and familiar interpretation borrowed from the algebra of rational numbers. One can write any algebraic identity that is valid for the rational numbers and interpret it as an isomorphism with a clear computational interpretation: negative values flow backwards and fractional values represent garbage-collection processes.
- Because we are *not* in the context of the full λ-calculus, which allows arbitrary duplication and erasure of information, values of negative and fractional types are first-class values that can flow anywhere. The information-preserving computational infrastructure guarantees that, in a complete program ($c : A \leftrightarrow B$ where A and B does not contain duals), every negative demand will be satisfied, and every allocation matched by a fractional value will also be collected.

We envision two broad directions of future research.

Quantum Computing. At a foundational level, the marriage of quantum mechanics and computer science provides a computational interpretation of physics, and in particular directly addresses the question of interpretation of quantum mechanics. Aspects of fractional and negative types appear related to the creation and annihilation of entangled particle/antiparticle pairs [Panangaden and Paquette 2011] and negative information and entropy [Loeb 1992; Oppenheim et al. 2005; Penrose 1971; Rio et al. 2011; Schanuel 1991]. At a more pragmatic level, reversible languages such as Π can be expressed in a more intuitive syntax for programmers by using symmetric pattern matching [James and Sabry 2014] and can become quantum programming languages when extended with superpositions of patterns [Sabry et al. 2018]. There does not appear to be any obstacles in combining the quantum extension with the negative and fractional types thus enriching quantum programming languages with high-level abstractions such as backtracking.

General Purpose Programming Languages. The extension of Π to $\Pi^{\mathbb{Q}}$ corresponds to a move from a ring-like structure to a full algebraic field. As we have argued, computation in the field of rational numbers is quite expressive and interesting and yet, it has two fundamental limitations. First it cannot express recursive types, and second it cannot express user-defined datatype definitions. We believe these two extensions to be orthogonal to each other. Recursive types were considered by James and Sabry [2012a]; there should be no difficulty in extending $\Pi^{\mathbb{Q}}$ with recursive types following the same technique to produce a Turing-complete language with the additional abstractions provided by negative and fractional types. Arbitrary datatype definitions are more interesting as each datatype definition can be viewed as a solution to a polynomial over types. For example, the datatype of binary trees $\mu x.(1 + x * x)$ can be re-arranged as $x = 1 + x * x$ or $x^2 - x + 1 = 0$ whose solution would be the type "$(1/2) + i(\sqrt{3}/2)$." These types have been studied extensively following a paper by Blass [1995] which used the above datatype of trees to infer an isomorphism between seven binary trees and one! Enriching $\Pi^{\mathbb{Q}}$ beyond the field of rational numbers to the field of algebraic numbers would be an interesting problem to investigate.

ACKNOWLEDGMENTS

We sincerely thank the anonymous for their careful reviews, comments, and feedback that improved the paper. We also acknowledge Jacques Carette and Vikraman Choudhury for many discussions and collaborations that shaped the current work. Finally, we gratefully acknowledge the insights of Roshan James who conceived of the idea of negative and fractional types as early as 2012 in his unpublished manuscript. This material is based upon work supported by the National Science Foundation under Grant No. 1936353. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Samson Abramsky. 2005. A structural approach to reversible computation. *Theor. Comput. Sci.* 347 (December 2005), 441–464. Issue 3.

S. Abramsky and B. Coecke. 2004. A categorical semantics of quantum protocols. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004.* 415–425.

Samson Abramsky, Simon Gay, and Rajagopal Nagarajan. 1996. Interaction Categories and the Foundations of Typed Concurrent Programming. In *Proceedings of the NATO Advanced Study Institute on Deductive Program Design* (Marktoberdorf, Germany). Springer-Verlag, Berlin, Heidelberg, 35–113.

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

Samson Abramsky and Radha Jagadeesan. 1994. New foundations for the geometry of interaction. *Inf. Comput.* 111 (May 1994), 53–119. Issue 1.

Bogdan Aman, Gabriel Ciobanu, Robert Glück, Robin Kaarsgaard, Jarkko Kari, Martin Kutrib, Ivan Lanese, Claudio Antares Mezzina, Łukasz Mikulski, Rajagopal Nagarajan, Iain Phillips, G. Michele Pinna, Luca Prigioniero, Irek Ulidowski, and Germán Vidal. 2020. *Foundations of Reversible Computation.* Springer International Publishing, Cham, 1–40. https://doi.org/10.1007/978-3-030-47361-7_1

Zena M. Ariola, Hugo Herbelin, and Amr Sabry. 2009. A type-theoretic foundation of delimited continuations. *Higher Order Symbol. Comput.* 22 (September 2009), 233–273. Issue 3.

Jean Bénabou. 1963. Catégories avec multiplication. *C. R. de l'Académie des Sciences de Paris* 256, 9 (1963), 1887–1890.

Jean Bénabou. 1964. Algèbre élémentaire dans les catégories avec multiplication. *C. R. Acad. Sci. Paris* 258, 9 (1964), 771–774.

C. H. Bennett. 1973. Logical reversibility of computation. *IBM J. Res. Dev.* 17 (November 1973), 525–532. Issue 6.

Charles H. Bennett. 1989. Time/Space Trade-Offs for Reversible Computation. *SIAM J. Comput.* 18, 4 (1989), 766–776. https://doi.org/10.1137/0218053

A. Blass. 1995. Seven trees in one. *Journal of Pure and Applied Algebra* 103, 1-21 (1995).

William J. Bowman, Roshan P. James, and Amr Sabry. 2011. Dagger Traced Symmetric Monoidal Categories and Reversible Programming. In *RC 2011.*

Jacques Carette and Amr Sabry. 2016. Computing with Semirings and Weak Rig Groupoids. In *ESOP (Lecture Notes in Computer Science, Vol. 9632)*. Springer, 123–148.

Siu Man Chan. 2013. *Pebble Games and Complexity*. Ph.D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-145.html

Chao-Hong Chen, Vikraman Choudhury, Jacques Carette, and Amr Sabry. 2020. Fractional Types. In *Reversible Computation*, Ivan Lanese and Mariusz Rawski (Eds.). Springer International Publishing, Cham, 169–186.

Stephen Clark, Bob Coecke, and Mehrnoosh Sadrzadeh. 2008. A Compositional Distributional Model of Meaning. *Proceedings of the Second Symposium on Quantum Interaction (QI-2008)* (2008), 133–140.

Tristan Crolard. 2001. Subtractive logic. *Theoretical Computer Science* 254, 1-2 (2001), 151–185. https://doi.org/10.1016/S0304-3975(99)00124-3

Tristan Crolard. 2004. A Formulae-as-Types Interpretation of Subtractive Logic. *Journal of Logic and Computation* 14, 4 (2004), 529–570.

Pierre-Louis Curien and Hugo Herbelin. 2000. The Duality of Computation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 233–243. https://doi.org/10.1145/351240.351262

Andrzej Filinski. 1989. Declarative Continuations: an Investigation of Duality in Programming Language Semantics. In *Category Theory and Computer Science*.

Andrzej Filinski. 1992. Linear Continuations. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Mexico, USA) *(POPL '92)*. Association for Computing Machinery, New York, NY, USA, 27–38. https://doi.org/10.1145/143165.143174

Marcelo Fiore. 2004. Isomorphisms of Generic Recursive Polynomial Types. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy) *(POPL '04)*. Association for Computing Machinery, New York, NY, USA, 77–88. https://doi.org/10.1145/964001.964008

Marcelo Fiore. 2015. An Axiomatics and a Combinatorial Model of Creation/Annihilation Operators. (2015). arXiv:1506.06402[math.CT].

M. P. Fiore, R. Di Cosmo, and V. Balat. 2006. Remarks on Isomorphisms in Typed Calculi with Empty and Sum Types. *Annals of Pure and Applied Logic* 141, 1-2 (2006), 35–50.

E. Fredkin and T. Toffoli. 1982. Conservative logic. *International Journal of Theoretical Physics* 21, 3 (1982), 219–253.

J.Y. Girard. 1989. Geometry of interaction 1: Interpretation of System F. *Studies in Logic and the Foundations of Mathematics* 127 (1989), 221–260.

Robert Glück and Robin Kaarsgaard. 2018. A Categorical Foundation for Structured Reversible Flowchart Languages. *Electronic Notes in Theoretical Computer Science* 336 (2018), 155 – 171. https://doi.org/10.1016/j.entcs.2018.03.021 The Thirty-third Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIII).

Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: A Scalable Quantum Programming Language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. ACM, New York, NY, USA, 333–342.

Chris Heunen, Robin Kaarsgaard, and Martti Karvonen. 2018. Reversible Effects as Inverse Arrows. *Electronic Notes in Theoretical Computer Science* 341 (2018), 179 – 199. https://doi.org/10.1016/j.entcs.2018.11.009 Proceedings of the Thirty-Fourth Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIV).

Chris Heunen and Martti Karvonen. 2015. Reversible Monadic Computing. *Electronic Notes in Theoretical Computer Science* 319 (2015), 217 – 237. https://doi.org/10.1016/j.entcs.2015.12.014 The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI).

Christiaan Heunen and Jamie Vicary. 2019. *Categories for quantum theory: an introduction*. Oxford University Press, United Kingdom.

Furio Honsell and Marina Lenisa. 2004. "Wave-Style" Geometry of Interaction Models in Rel Are Graph-Like Lambda-Models. In *Types for Proofs and Programs*, Stefano Berardi, Mario Coppo, and Ferruccio Damiani (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 242–258.

Jason Z.S. Hu and Jacques Carette. 2020. Proof-relevant Category Theory in Agda. (2020). https://arxiv.org/abs/2005.07059.

Roshan P. James and Amr Sabry. 2012a. Information Effects. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 73–84. https://doi.org/10.1145/2103656.2103667

Roshan P. James and Amr Sabry. 2012b. *The Two Dualities of Computation : Negative and Fractional Types*. Technical Report. Indiana University.

Roshan P James and Amr Sabry. 2014. Theseus: A high level language for reversible computing. In *Work-in-progress report in the Conference on Reversible Computation*.

Roshan P James, Zachary Sparks, Jacques Carette, and Amr Sabry. 2013. Fractional types. (2013).

A. Joyal, R. Street, and D. Verity. 1996. Traced monoidal categories. In *Mathematical Proceedings of the Cambridge Philosophical Society*. Cambridge Univ Press.

Robin Kaarsgaard and Niccolò Veltri. 2019. En Garde! Unguarded Iteration for Reversible Computation in the Delay Monad. *Lecture Notes in Computer Science* (2019), 366–384. https://doi.org/10.1007/978-3-030-33636-3_13

J. Kastl. 1979. Inverse Categories. *Studien zur Algebra und ihre Anwendungen* 7 (1979), 51–60.

G.M. Kelly and M.L. Laplaza. 1980. Coherence for compact closed categories. *Journal of Pure and Applied Algebra* 19 (1980), 193 – 213. https://doi.org/10.1016/0022-4049(80)90101-2

G Maxwell Kelly. 1972. Many-variable functorial calculus. I. In *Coherence in categories*. Springer, 66–105.

Miguel L. Laplaza. 1972. Coherence for distributivity. In *Coherence in Categories*, G.M. Kelly, M. Laplaza, G. Lewis, and Saunders Mac Lane (Eds.). Lecture Notes in Mathematics, Vol. 281. Springer Verlag, Berlin, 29–65.

D. Loeb. 1992. Sets with a negative number of elements. *Advances in Mathematics* 91, 1 (1992), 64–74.

Ian Mackie. 1995. The Geometry of Interaction Machine. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. Association for Computing Machinery, New York, NY, USA, 198–208. https://doi.org/10.1145/199448.199483

Ian Mackie. 2011. Reversible Higher-order Computations. In *Workshop on Reversible Computation*.

Saunders MacLane. 1963. Natural associativity and commutativity. *Rice Institute Pamphlet-Rice University Studies* 49, 4 (1963).

Jonathan Oppenheim, Robert W Spekkens, and Andreas Winter. 2005. A classical analogue of negative information. *arXiv preprint quant-ph/0511247* (2005).

P. Panangaden and É.O. Paquette. 2011. *A Categorical Presentation of Quantum Computation with Anyons*. Springer Berlin Heidelberg, Berlin, Heidelberg, 983–1025.

R Penrose. 1971. Applications of negative dimensional tensors. In *Combinatorial Mathematics and its Applications*. Academic Press.

James Propp. 2002. Euler measure as generalized cardinality. (2002). arXiv:math/0203289 [math.CO].

Cecylia Rauszer. 1974a. A formalization of the propositional calculus of H-B logic. *Studia Logica* 33 (1974), 23–34. Issue 1.

C. Rauszer. 1974b. Semi-boolean algebras and their applications to intuitionistic logic with dual operators. *Fundamenta Mathematicae* 83 (1974), 219–249.

C. Rauszer. 1980. An algebraic and Kripke-style approach to a certain extension of intuitionistic logic. In *Dissertationes Mathmaticae*. Vol. 167. Institut Mathématique de l'Académie Polonaise des Sciences.

Udday S. Reddy. 1991. Acceptors as values: Functional programming in classical linear logic. (Dec. 1991). Manuscript.

Lídia del Rio, Johan Åberg, Renato Renner, Oscar Dahlsten, and Vlatko Vedral. 2011. The thermodynamic meaning of negative entropy. *Nature* 474, 7349 (2011), 61–63. https://doi.org/10.1038/nature10123

Amr Sabry, Benoît Valiron, and Juliana Kaizer Vizzotto. 2018. From Symmetric Pattern-Matching to Quantum Control. In *Foundations of Software Science and Computation Structures*, Christel Baier and Ugo Dal Lago (Eds.). Springer International Publishing, Cham, 348–364.

Stephen H. Schanuel. 1991. Negative sets have Euler characteristic and dimension. In *Category Theory*, Aurelio Carboni, Maria Cristina Pedicchio, and Guiseppe Rosolini (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 379–385.

Peter Selinger. 2001. Control categories and duality: on the categorical semantics of the lambda-mu calculus. *Mathematical. Structures in Comp. Sci.* 11 (April 2001), 207–260. Issue 2.

Peter Selinger. 2011. A Survey of Graphical Languages for Monoidal Categories. In *New Structures for Physics*, Bob Coecke (Ed.). Lecture Notes in Physics, Vol. 813. Springer Berlin / Heidelberg, 289–355.

Zachary Sparks and Amr Sabry. 2014. Superstructural Reversible Logic. In *3rd International Workshop on Linearity*.

Michael Kirkedal Thomsen, Robin Kaarsgaard, and Mathias Soeken. 2015. Ricercar: A Language for Describing and Rewriting Reversible Circuits with Ancillae and Its Permutation Semantics. In *Reversible Computation*, Jean Krivine and Jean-Bernard Stefani (Eds.). Springer International Publishing, Cham, 200–215.

Tommaso Toffoli. 1980. Reversible Computing. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*. Springer-Verlag, 632–644.

Philip Wadler. 2003. Call-by-value is dual to call-by-name. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming* (Uppsala, Sweden). ACM, New York, NY, USA, 189–201. https://doi.org/10.1145/944705.944723

Philip Wadler. 2005. Call-by-Value Is Dual to Call-by-Name – Reloaded. In *Term Rewriting and Applications*, Jürgen Giesl (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 185–203.

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

Tetsuo Yokoyama and Robert Glück. 2007. A reversible programming language and its invertible self-interpreter. In *PEPM* (Nice, France). ACM, 144–153.