# Computing in the Field of Rationals

Roshan P. James
Indiana University
rpjames@indiana.edu

Zachary Sparks
Indiana University
zasparks@indiana.edu

Jacques Carette
McMaster University
carette@mcmaster.ca

Amr Sabry
Indiana University
sabry@indiana.edu

## Abstract

Previous work on information effects introduced the computational model $\Pi$ which lacked a natural notion of first-class functions. This paper connects the line of work starting with Filinski[12] on the duality of computation [11, 30] with work on information preserving computation [7, 15].

We present a computational model whose types are *the field of rational numbers*. This computational model is derived systematically from the $\Pi$ by considering a symmetric notion of duality. Unlike the $\lambda$-calculus which shows a classical De Morgan duality, here we have two axis of dualization and hence two dualities – an additive duality, namely negative types and a multiplicative duality, namely fractional types. Intuitively, values of negative types are values that flow "backwards" to satisfy demands and values of fractional types are values that impose constraints on their context.

***Categories and Subject Descriptors*** D.3.1 [*Formal Definitions and Theory*]; F.3.2 [*Semantics of Programming Languages*]; F.3.3 [*Studies of Program Constructs*]: Type structure

***General Terms*** Languages, Theory

***Keywords*** continuations, information flow, linear logic, logic programming, quantum computing, reversible logic, symmetric monoidal categories, compact closed categories.

## 1. Introduction

Andrzej Filinski's Masters thesis [12] suggested a remarkable symmetry underlying computation in the $\lambda$-calculus. Filinski inroduced a symmetric extension of the $\lambda$-calculus in which he showed that values are dual to continuations, functions are dual to delimited continuations, pairs of values as dual to sums (choice) of continuations ($(a \times b)^{\perp} \leftrightarrow a^{\perp} + b^{\perp}$), sums of values are dual to pairs of continuations ($(a + b)^{\perp} \leftrightarrow a^{\perp} \times b^{\perp}$) and that call-by-value is dual to call-by-name.

The symmetries exposed by Filinski suggested that values/continuations and functions/delimited continuations are different aspects of the same computational phenomenon and led to much work in this area. Symmetry between the value fragment and the continuation fragment was refined by Curien and Herbelin [11] who introduced the sequent-style $\mu\tilde{\mu}$-calculus, that exhibits symmetries between values and continuations and between call-by-value and call-by-name. The duality between call-by-name and call-by-value was further investigated by Selinger using control categories [25] and by Wadler [29, 30].

Previous work on *information effects* established the logically reversible computational model $\Pi$ (and its extension $\Pi^o$) wherein computation is information/entropy preserving. The computational model is derived from the isomorphisms of finite-types, is complete for combinational circuits and every computation admits an adjoint which is the "inverse of the computation". The term model of $\Pi$ corresponds to the categorification of a commutative semiring with two commutative monoidal structures $(0, +)$ and $(\times, 1)$ with $\times$ distributing over $+$ and is sometimes called a commutative bimonoidal category.

The language $\Pi$ however lacks a natural representation for first-class functions due to the first-order nature of its values. The first approach to try is to re-use Filinski's duality (or one of its later variants, such duality in the linear logic setting). The De Morgan style duality in Filinski's symmetric $\lambda$-calculus (and later works) is computationally interpreted as follows: if one has a pair of values $(v_1, v_2)$, then the context that consumes the value has a choice of comsuming either $v_1$ or $v_2$. Hence the dual of $(v_1, v_2)$ maybe thought of as the continuation that does $fst$ or $snd$. Such a De Morgan style computational interpretation is inherently based on the deletion of values and completely collapses the logical reversibility and information preservation properties of $\Pi$. Hence classical De Morgan duality and linear logic style De Morgan duality (which appears in later works on polarised focalized $\mu\tilde{\mu}$) do not apply directly in the context of $\Pi$.

The other obvious approach to try and build functions is to construct a "closed" category. It is well know that the Int-construction of Joyal et al [17] and the $\mathcal{G}$ construction of Abramsky allow the construction of compact closed category (sometimes also called monoidal closed) from a traced symmetric monoidal category. The traced extension of $\Pi$, called $\Pi^o$ has been studied before and indeed the Int-construction gives us a compact closed structure in this setting. However, the Int-construction does not preserve the bimonoidal structure of $\Pi$, i.e. if we use an additive trace, we lose the $(1, \times)$ monoid and distributivity.

The technical goal of this paper is to present the compuational model $\Pi^{\eta\epsilon}$, a Filinski-style symmetric dual extension of $\Pi$. The extended language has two distinct dualizing actions – an additive duality characterized by negative types $-(\_)$ and a multiplicative du-

ality characterized by fractional types $1/(\_)$. Each dualizing operation preserves the underlying connective $-(a+b) \leftrightarrow (-a)+(-b)$ and $1/(a \times b) \leftrightarrow 1/a \times 1/b$. The types of $\Pi^{\eta\epsilon}$ correspond to the *field of rational numbers*. Consequently algebraic manipulations of rationals can be given an operational interpretation and *have computational content*.

The term model of $\Pi^{\eta\epsilon}$ corresponds to the *categorification of a field*. The category is compact closed over the additive monoid. In the multiplicative monoid (since there is no division by zero) respects compact closed structure on non-zero objects. Compact closed categories preserve the monoidal tensor on dualizing. In other words, negative and fractional dualities must not confused with $()^{\perp}$ duality in the additive and multiplicative fragment of linear logic. Models of linear logic rely on *-autonomous categories which change the monoidal tensor when dualizing.

We present $\Pi$ in the big-step style presented in the previous work, develop the categorical semantics of its term model and present a small-step semantics useful for modeling fractionals and negatives. We proceed to present the fractional fragment of the language $\Pi^{\eta\epsilon}_{\times}$, work out its properties and categorical semantics. We further use the multiplicative *trace* to express a SAT solver. We then present the negative fragment of the language $\Pi^{\eta\epsilon}_{+}$ and finally present the unified language $\Pi^{\eta\epsilon}$.

***Negative Types.*** Consider the following algebraic manipulation relating a natural number $a$ to itself (ignoring the dotted line for a moment):

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

Although seemingly pointless, this algebraic proof corresponds, in our model, to an isomorphism of type $a \leftrightarrow a$ with a non-trivial and interesting computational interpretation. The witness for this isomorphism is a computation that takes a value of type $a$, say \$20.00, and eventually produces another \$20.00 value as its output. As the semantics of Sec. **??** formalizes, this computation flows along the dotted line with the following intermediate steps:

- We start at line (0) with \$20.00;
- We proceed to line (1) with the same \$20.00 but tagged as being in the left summand of the sum type $a+0$; we indicate this value as *left* 20;
- We continue to line (2) with the same value *left* 20;
- At line (3), as a result of re-association the tag on the \$20.00 changes to indicate that it is in the left-left summand, i.e., the value is now *left* (*left* 20);
- At line (4), we find ourselves needing to produce a value of type 0 which is impossible; this signals the beginning of a reverse execution which sends us back to line (3) with a value *left* (*right* 20);
- Execution continues in reverse to line (2) with the value *right* (*left* 20);
- At line (1) we find ourselves again facing an empty type so we reverse execution again; we go to line (2) with a value *right* (*right* 20);
- We proceed to line (3) and (4) with the value *right* 20;
- We finally reach line 5 with the value 20.

The example illustrates that the empty type and negative types have a computational interpretation related to continuations: negative types denote values that backtrack to satisfy dependencies, or in other words act as debts that are satisfied by the backward flow of information.

***Fractional Types.*** Consider a similar algebraic manipulation involving fractional types.

$$
\begin{array}{lcccccl}
 & a & & & & & \text{initial } a \quad (0)\\
= & a & * & & 1 & & 1 \text{ is identity for } * \quad (1)\\
= & a & * & ((1/a) & * & a) & 1/a \text{ is the multiplicative inverse of } a \quad (2)\\
= & (a & * & (1/a)) & * & a & * \text{ is associative} \quad (3)\\
= & & 1 & & * & a & 1/a \text{ is the multiplicative inverse of } a \quad (4)\\
= & & & & & a & 1 \text{ is identity for } * \quad (5)
\end{array}
$$

In the case of negatives, the dotted line indicated the flow of control whereas for fractionals it indicates the flow of constraints. At the heart of logic programming is the idea of variables that capture constraints. Hence it is useful to trace the computation corresponding to the algebraic proof above, with the analogy to logic variables in mind.

As before, the execution begins at line (0) with the value 20. At line (1) two values, 20 and (), flow forward. One can think of the value () (of type 1) as "having a credit card." The credit card isn't money, nor is it debt, but is the option to generate a credit-debt constraint. At line (2) we exercise this option and hence have three values: the initial value 20 flowing from line (1) and two entangled values, $1/\alpha$ and $\alpha$. The $\alpha$ and $1/\alpha$ are unspecified values, i.e., we don't yet know how much money we need to borrow, but we do know that what is borrowed must be what is returned. Hence $\alpha$ denotes the presence of an unknown quantity and dually $1/\alpha$ should be thought of as the absence of an unknown quantity. At line (3), the missing unknown $1/\alpha$ is brought together with a value 20 and at line (4) we use the 20 to satisfy the constraint $1/\alpha$. In other words, this branch of the computation succeeded in borrowing 20 which immediately communicates the 20 to the rightmost branch.

Unlike with negative types, wherein only one value existed at a time and the computation backtracked, here we have three values that *exist at the same time*. In other words, the computation with fractions is realized with a schedule in which every value independently and concurrently proceeds through its subcomputation. The example illustrates that fractional types also have a computational interpretation that have some flavor of continuations: the fractional types denote values $(1/\alpha)$ that represent missing information that must be supplied in much the same sense that continuations denote evaluation contexts with holes that must be filled.

There are at least four fundamental points about the examples above that must be emphasized:

- As the examples illustrate, both negative types and fractional types corresponds to "debts" but in different ways: negatives are satisfied by backtracking and fractionals are satisfied by constraint propagation.
- It would clearly be disastrous if debts could be deleted or duplicated. This simple observation explains why these types are much simpler and much more appealing in a framework where information is guaranteed to be preserved. In previous work that used negative types (see Sec. 8), complicated mechanisms are typically needed to constrain the propagation and use of negative values because the surrounding computational framework is, generally speaking, careless in its treatment of information.
- Each of the values $-a + b$ and $(1/a) \times b$ can be viewed as a function that asks for an $a$ and then produces a $b$. When viewed as functions, we write these types as $a \multimap^{+} b$ and

$a \multimap^{\times} b$ respectively. Alternatively we can view these values as first producing a value of type $b$ and then demanding an $a$ and in that perspective they correspond to delimited continuations. Evidently, as the discussion above suggests, these two notions of functions are not the same at all and should not be conflated. Sec. 7 discusses this point in detail.

- The main reason credit card transactions are convenient is because they disentangle the propagation of the resources (money) from the propagation of the services. Not every transaction needs both the resources and services to be brought together: it is sufficient to have a promise that the demand for resources will be somehow satisfied, as long as the infrastructure can be trusted with such promises. This idea that dependencies can be freely decoupled and propagated can be a powerful programming tool and we leverage this in the construction of a novel SAT-solver (see Sec. 4).

***Contributions and Outline.*** To summarize, in a computational framework that guarantees that information is preserved, negative and fractional types provide fascinating mechanisms in which computations can be sliced and diced, decomposed and recomposed, run forwards and backwards, in arbitrary ways. The remainder of the paper formalizes these informal observations. Specifically our main contributions are:

- We extend $\Pi$ our reversible programming language of type isomorphisms [7, 15] (reviewed in Sec. 2) with a notion of negative types, that satisfies the isomorphism $a + (-a) \leftrightarrow 0$. The semantics of this extension is expressed by having a *dual* evaluator that reverses the flow of execution for negative values. (Sec. **??**)
- We independently extend $\Pi$ with a notion of fractional types, that satisfies the isomorphism $a \times (1/a) \leftrightarrow 1$. The semantics of this extension is expressed by introducing logic variables and a unification mechanism to model and resolve the constraints introduced by the fractional types. (Sec. **??**)
- We combine the above two extensions into a language, which we call $\Pi^{\eta\epsilon}$, whose type system allows any rational number to be used as a type. Moreover the types satisfy the same familiar and intuitive isomorphisms that are satisfied in the mathematical field of rational numbers. (Sec. **??**)
- We develop programming intuition and argue that negative and fractional types ought to be part of the vocabulary of every programmer. (Sec. **??**)
- We relate our notions of negative and fractional types to previous work on continuations. Briefly, we argue that conventional continuations conflate negative and fractional components. This observation allows us to relate two apparently unrelated lines of work: the first pioneered by Filinski [12] relating continuations to negative types and the second [5] relating continuations to the fractional types of the Lambek-Grishin calculus. (Sec. 8)

**Note:** All the constructions, semantics, and examples in this paper have been implemented and tested in Haskell. We will make the URL available once the code is organized for better presentation.

## 2. The Core Reversible Language: $\Pi$

```
(parts of rewrite that are done are removed from here)

In plain Pi:
------------

Semantics of combinators:
 - iso-types b1 <-> b2 are interpreted as bijections
   between the sets denoted by b1 and b2
 - a combinator c : b1 <-> b2 is interpeted as a
   bijection from set b1 to set b2.
 - there is a natural adjoint to c : b1 <-> b2
   corresponding to the bijection c^* : b2 <-> b1

Evaluation is function application, i.e.,
  eval c v applies the function (bijection) denoted by c to
  the value denoted by v.

[If we want to use categorical language,
we can express the above semantics using the category
of finite sets and bijections.]
```

We review our reversible language $\Pi$: the presentation in this section differs from the one in [15] in two technical aspects. First, we add the empty type 0 which is necessary to express the additive duality. Second, instead of explaining evaluation using a natural semantics, we give a small-step operational semantics that is more appropriate for later parts of this paper.

There is also a notable shift in emphasis. The terms of $\Pi$ are not classical values and functions; rather, the terms are isomorphism witnesses. In other words, the terms of $\Pi$ are proofs that certain "shapes of values" are isomorphic. And, in classical Curry-Howard fashion, our operational semantics shows how these proofs can be directly interpreted as actions on ordinary values which effect this shape transformation.

Of course, "shapes of values" are very familiar already: they are usually called *types*. But usually one designs a type system as a method of classifying terms, with the eventual purpose to show that certain properties of well-typed terms hold, such as safety. Our eventual goal is different: we start from a type system, and are striving to discover a term language which naturally inhabits these types, along with an appropriate operational semantics.

### 2.1 Syntax and Types

***Data.*** We view $\Pi$ as having two levels: it has traditional values, given by

$$\mathit{values}, v \quad ::= \quad () \mid \mathit{left}\ v \mid \mathit{right}\ v \mid (v, v)$$

and these are classified by ordinary types

$$\mathit{value\ types}, b \quad ::= \quad 0 \mid 1 \mid b + b \mid b \times b$$

Types include the empty type 0, the unit type 1, sum types $b_1 + b_2$, and products types $b_1 \times b_2$. Values includes $()$ which is the only value of type 1, $\mathit{left}\ v$ and $\mathit{right}\ v$ which inject $v$ into a sum type, and $(v_1, v_2)$ which builds a value of product type. There are no values of type 0.

But these should be regarded as largely ancilliary. In particular we do not treat the above values as first-class citizens. They only occur when we want to observe the effect of an isomorphism.

Nevertheless, these have a precise denotation: a type $b$ is interpreted as a set, and $v : b$ denotes that $v$ is a member of the set denoted by $b$. As our (traditional) values are first-order, and non-polymorphic, such a set interpretation is sound.

***Isomorphisms.*** The terms of $\Pi$ are witnesses to type isomorphisms. They have (iso) types $v \leftrightarrow v$. Specifically, they are wit-

nesses to the following type isomorphisms:

$$\begin{array}{rcl}
zeroe: & 0 + b \leftrightarrow b & : zeroi \\
swap^+: & b_1 + b_2 \leftrightarrow b_2 + b_1 & : swap^+ \\
assocl^+: & b_1 + (b_2 + b_3) \leftrightarrow (b_1 + b_2) + b_3 & : assocr^+ \\
unite: & 1 \times b \leftrightarrow b & : uniti \\
swap^\times: & b_1 \times b_2 \leftrightarrow b_2 \times b_1 & : swap^\times \\
assocl^\times: & b_1 \times (b_2 \times b_3) \leftrightarrow (b_1 \times b_2) \times b_3 & : assocr^\times \\
distrib_0: & 0 \times b \leftrightarrow 0 & : factor_0 \\
distrib: & (b_1 + b_2) \times b_3 \leftrightarrow (b_1 \times b_3) + (b_2 \times b_3) & : factor
\end{array}$$

Each line of the above table introduces a pair of dual constants[1] (and its typing) that witness the type isomorphism in the middle. Collectively the isomorphisms state that the structure $(b, +, 0, \times, 1)$ is a *commutative semiring*, i.e., that each of $(b, +, 0)$ and $(b, \times, 1)$ is a commutative monoid and that multiplication distributes over addition. These are the base (non-reducible) terms of the second, principal level of Π.

Note how the above has two readings: first as a set of typing relations for a set of constants, as well as giving names to the *axioms* of commutative semirings. However, if these axioms are seen as universally quantified, orientable statements, they also induce transformations of the (traditional) values. The (categorical) intuition here is that these axioms have computational content because they witness isomorphisms rather than merely stating an extensional equality.

For example $swap^\times$ induces a function (akin to)

```
swapProd (a,b) = (b,a)
```

The isomorphisms are extended to form a congruence relation by adding the following constructors that witness equivalence and compatible closure:

$$\frac{}{id : b \leftrightarrow b} \quad \frac{c : b_1 \leftrightarrow b_2}{sym\ c : b_2 \leftrightarrow b_1} \quad \frac{c_1 : b_1 \leftrightarrow b_2 \quad c_2 : b_2 \leftrightarrow b_3}{c_1 \fatsemi c_2 : b_1 \leftrightarrow b_3}$$

$$\frac{c_1 : b_1 \leftrightarrow b_3 \quad c_2 : b_2 \leftrightarrow b_4}{c_1 + c_2 : b_1 + b_2 \leftrightarrow b_3 + b_4} \quad \frac{c_1 : b_1 \leftrightarrow b_3 \quad c_2 : b_2 \leftrightarrow b_4}{c_1 \times c_2 : b_1 \times b_2 \leftrightarrow b_3 \times b_4}$$

The syntax is overloaded: we use the same symbol at the value-type level and at the isomorphism-type level for denoting sums and products. Hopefully this will not cause undue confusion.

It is important to note that "values" and "isomorphisms" are completely separate syntactic categories which do not intermix. The semantics of the language come when these are made to interact at the "top level", a new syntactic category:

$$top - level\ term, l \quad ::= \quad c\ v$$

We refer to this as *application*.
To summarize, the syntax of Π is given as follows.

DEFINITION 2.1. *(Syntax of* Π*) We collect our types, values, and isomorphisms, to get the full language definition.*

$$\begin{array}{rcl}
value\ types, b & ::= & 0 \mid 1 \mid b + b \mid b \times b \\
values, v & ::= & () \mid left\ v \mid right\ v \mid (v, v) \\
\\
iso.\ types, t & ::= & b \leftrightarrow b \\
base\ iso & ::= & zeroe \mid zeroi \\
& \mid & swap^+ \mid assocl^+ \mid assocr^+ \\
& \mid & unite \mid uniti \\
& \mid & swap^\times \mid assocl^\times \mid assocr^\times \\
& \mid & distrib_0 \mid factor_0 \mid distrib \mid factor \\
iso\ comb., c & ::= & iso \mid id \mid sym\ c \mid c \fatsemi c \mid c + c \mid c \times c \\
top - level\ term, l & ::= & c\ v
\end{array}$$

[1] where $swap^\times$ and $swap^+$ are self-dual

***Adjoint.*** An important property of the language is that every term $c$ has an adjoint $c^\dagger$ that reverses the isomorphisms $c$. This is evident by construction for the primitive isomorphisms. For the closure combinators, the adjoint is homomorphic except for the case of sequencing in which the order is reversed, i.e., $(c_1 \fatsemi c_2)^\dagger = (c_2{}^\dagger) \fatsemi (c_1{}^\dagger)$.

DEFINITION 2.2 (Size of a type). *The size of a type b, denoted by* $\lceil b \rceil$*, is a numeric value and is defined to be:*

$$\begin{array}{rclcrcl}
\lceil b_1 + b_2 \rceil & = & \lceil b_1 \rceil + \lceil b_2 \rceil & \qquad & \lceil 0 \rceil & = & 0 \\
\lceil b_1 \times b_2 \rceil & = & \lceil b_1 \rceil \times \lceil b_2 \rceil & & \lceil 1 \rceil & = & 1
\end{array}$$

*where $+$ is numeric addition and $\times$ is numeric multiplication.*

In the setting of Π the *size*, $\lceil b \rceil$, may be simply thought of as the number of inhabitants (an arity) of the type. This intuition will however become tenuous in the presence of negative and fractional types.

## 2.2 Semantics

The operational semantics of top-level terms of Π is summarized below (also see [15]). The semantics of applying the primitive combinators to a value is given by the following single-step reductions below. Since there are no values of type 0, there are no rules for the impossible cases:

$$\begin{array}{rlcl}
zeroi & v & \mapsto & right\ v \\
zeroe & (right\ v) & \mapsto & v \\
swap^+ & (left\ v) & \mapsto & right\ v \\
swap^+ & (right\ v) & \mapsto & left\ v \\
assocl^+ & (left\ v_1) & \mapsto & left\ (left\ v_1) \\
assocl^+ & (right\ (left\ v_2)) & \mapsto & left\ (right\ v_2) \\
assocl^+ & (right\ (right\ v_3)) & \mapsto & right\ v_3 \\
assocr^+ & (left\ (left\ v_1)) & \mapsto & left\ v_1 \\
assocr^+ & (left\ (right\ v_2)) & \mapsto & right\ (left\ v_2) \\
assocr^+ & (right\ v_3) & \mapsto & right\ (right\ v_3) \\
unite & ((), v) & \mapsto & v \\
uniti & v & \mapsto & ((), v) \\
swap^\times & (v_1, v_2) & \mapsto & (v_2, v_1) \\
assocl^\times & (v_1, (v_2, v_3)) & \mapsto & ((v_1, v_2), v_3) \\
assocr^\times & ((v_1, v_2), v_3) & \mapsto & (v_1, (v_2, v_3)) \\
distrib & (left\ v_1, v_3) & \mapsto & left\ (v_1, v_3) \\
distrib & (right\ v_2, v_3) & \mapsto & right\ (v_2, v_3) \\
factor & (left\ (v_1, v_3)) & \mapsto & (left\ v_1, v_3) \\
factor & (right\ (v_2, v_3)) & \mapsto & (right\ v_2, v_3)
\end{array}$$

The operational semantics of the closure conditions are presented in the usual big-step style.

$$\frac{}{id\ v \mapsto v} \quad \frac{c^\dagger v_1 \mapsto v_2}{(sym\ c)\ v_1 \mapsto v_2} \quad \frac{c_1\ v_1 \mapsto v \quad c_2\ v \mapsto v_2}{(c_1 \fatsemi c_2)\ v_1 \mapsto v_2}$$

$$\frac{c_1\ v_1 \mapsto v_2}{(c_1 + c_2)\ (left\ v_1) \mapsto left\ v_2} \quad \frac{c_2\ v_1 \mapsto v_2}{(c_1 + c_2)\ (right\ v_1) \mapsto right\ v_2}$$

$$\frac{c_1\ v_1 \mapsto v_3 \quad c_2\ v_2 \mapsto v_4}{(c_1 \times c_2)\ (v_1, v_2) \mapsto (v_3, v_4)}$$

The type safety of Π follows directly from the correspondence between inductive definitions of the types and the big-step semantics. Previous work also established the following properties:

PROPOSITION 2.3 (Strongly Normalizing). Π *computations always terminate.* $\forall c : b_1 \leftrightarrow b_2, v : b_1, \exists\ v' : b_2.\ c\ v \mapsto v'$

PROPOSITION 2.4 (Logical Reversibility). $c\ v \mapsto v'$ *iff* $c^\dagger v' \mapsto v$

DEFINITION 2.5 ( $c_1 = c_2$ ). *We say* $c_1 = c_2$ *if* $c_1 : b_1 \leftrightarrow b_2$ *and* $c_2 : b_1 \leftrightarrow b_2$ *and for all* $v_1 : b$*, we have* $c_1\ v_1 \mapsto v_2$ *iff* $c_2\ v_1 \mapsto v_2$ *and* $v_2 : b_2$.

## 2.3 Constructions

The constructions in this section provide a brief overview to writing programs in Π. More details maybe found in previous work (Sec. 3.3 [15] and the constructions in [16]).

***Booleans, Conditionals and Cloning.*** We use the type $1+1$ to denote *bool*, with *left* () as *true* and *right* () as *false*. The combinator $not : bool \leftrightarrow bool$ can be expressed by $swap^+$ at the type *bool*. Further, given any two combinators $c_1 : b_1 \leftrightarrow b_2$ and $c_2 : b_1 \leftrightarrow b_2$, we can write the conditional combinator $if_{c_1,c_2} : bool \times b_1 \leftrightarrow bool \times b_2$ as $distrib \fatsemi ((id \times c_1) + (id \times c_2)) \fatsemi factor$. If $c_1\ v \mapsto v'$ and $c_2\ v \mapsto v''$, we can verify that the conditional maps $if_{c_1,c_2}(true, v) \mapsto (true, v')$ and $if_{c_1,c_2}(false, v) \mapsto (false, v'')$. Several useful combinators follow from this basic construction:

1. The combinator $cnot : bool \times bool \leftrightarrow bool \times bool$ that negates the second input if the first input (called the control wire) is *true* may be defined as $if_{not,id}$.
2. The universal reversible combinator, the Toffoli gate, may be defined as $if_{cnot,id} : bool \times (bool \times bool) \leftrightarrow bool \times (bool \times bool)$. This can be extended to test $n$ control wires and we call this combinator $cnot^n : bool^{n+1} \leftrightarrow bool^{n+1}$.
3. The combinator $if_{id,not} : bool \times bool \leftrightarrow bool \times bool$ has the property that it maps $(x, true) \mapsto (x, x)$ i.e. if the second bit is fixed to *true* it clones the first bit. We can generalize this to clone $n$ bits and we write $clone^n : bool^n \times bool^n \leftrightarrow bool^n \times bool^n$. The second argument $bool^n$ must be supplied *true* values to correctly simulate cloning.

### 2.4 Small Step Semantics

The reductions for the primitive isomorphisms above are exactly the same as have been presented before [15]. The reductions for the closure combinators are however presented in a small-step operational style using the following definitions of evaluation contexts and machine states:

$$\begin{array}{rcl} Combinator\ Contexts, C & = & \Box \mid Fst\ C\ c \mid Snd\ c\ C \\ & \mid & L^{\times}\ C\ c\ v \mid R^{\times}\ c\ v\ C \\ & \mid & L^{+}\ C\ c \mid R^{+}\ c\ C \\ Machine\ states & = & \langle c, v, C\rangle \mid [c, v, C] \\ Start\ state & = & \langle c, v, \Box\rangle \\ Stop\ State & = & [c, v, \Box] \end{array}$$

The machine transitions below track the flow of particles through a circuit. The start machine state, $\langle c, v, \Box\rangle$, denotes the particle $v$ about to be evaluated by the circuit $c$. The end machine state, $[c, v, \Box]$, denotes the situation where the particle $v$ has exited the circuit $c$.

$$\begin{array}{rcll} \langle iso, v, C\rangle & \mapsto & [iso, v', C] & (1) \\ & & where\ iso\ v \mapsto v' & \\ \langle c_1 \fatsemi c_2, v, C\rangle & \mapsto & \langle c_1, v, Fst\ C\ c_2\rangle & (2) \\ [c_1, v, Fst\ C\ c_2] & \mapsto & \langle c_2, v, Snd\ c_1\ C\rangle & (3) \\ [c_2, v, Snd\ c_1\ C] & \mapsto & [c_1 \fatsemi c_2, v, C] & (4) \\ \langle c_1 + c_2, left\ v, C\rangle & \mapsto & \langle c_1, v, L^{+}\ C\ c_2\rangle & (5) \\ [c_1, v, L^{+}\ C\ c_2] & \mapsto & [c_1 + c_2, left\ v, C] & (6) \\ \langle c_1 + c_2, right\ v, C\rangle & \mapsto & \langle c_2, v, R^{+}\ c_1\ C\rangle & (7) \\ [c_2, v, R^{+}\ c_1\ C] & \mapsto & [c_1 + c_2, right\ v, C] & (8) \\ \langle c_1 \times c_2, (v_1, v_2), C\rangle & \mapsto & \langle c_1, v_1, L^{\times}\ C\ c_2\ v_2\rangle & (9) \\ [c_1, v_1, L^{\times}\ C\ c_2\ v_2] & \mapsto & \langle c_2, v_2, R^{\times}\ c_1\ v_1\ C\rangle & (10) \\ [c_2, v_2, R^{\times}\ c_1\ v_1\ C] & \mapsto & [c_1 \times c_2, (v_1, v_2), C] & (11) \end{array}$$

Rule (1) describes evaluation by a primitive isomorphism. Rules (2), (3) and (4) deal with sequential evaluation. Rule (2) says that for the value $v$ to flow through the sequence $c_1 \fatsemi c_2$, it should first flow through $c_1$ with $c_2$ pending in the context $(Fst\ C\ c_2)$. Rule (3) says the value $v$ that exits from $c_1$ should proceed to flow through $c_2$. Rule (4) says that when the value $v$ exits $c_2$, it also exits the sequential composition $c_1 \fatsemi c_2$. Rules (5) to (8) deal with $c_1 + c_2$ in the same way. In the case of sums, the shape of the value, i.e., whether it is tagged with *left* or *right*, determines whether path $c_1$ or path $c_2$ is taken. Rules (9), (10) and (11) deal with $c_1 \times c_2$ similarly. In the case of products the value should have the form $(v_1, v_2)$ where $v_1$ flows through $c_1$ and $v_2$ flows through $c_2$. Both these paths are entirely independent of each other and we could evaluate either first, or evaluate both in parallel. In this presentation we have chosen to follow $c_1$ first, but this choice is entirely arbitrary.

The interesting thing about the semantics is that it represents a reversible abstract machine. In other words, we can compute the start state from the stop state by changing the reductions $\mapsto$ to run backwards $\leftarrow$. When running backwards, we use the isomorphism represented by a combinator $c$ in the reverse direction, i.e., we use the adjoint $c^{\dagger}$.

PROPOSITION 2.6 (Correspondence). *Evaluation in the small-step evaluator corresponds to evaluation in the natural semantics.*
$c\ v \mapsto v'$ *iff* $\langle c, v, \Box\rangle \mapsto^* [c, v', \Box]$

**Proof.** To prove the above, we first show that a more general lemma holds, namely that: $c\ v \mapsto v'$ iff $\langle c, v, C\rangle \mapsto^* [c, v', C]$.

To show the left-to-right direction we proceed by induction on the derivation of $c\ v \mapsto v'$. In the case of primitive isomorphisms the condition holds trivially. In the case of composition, we work out the case of $c_1 + c_2$ as an example. Given $c_1 + c_2\ v \mapsto v'$ we have to show that there is a small step derivation sequence that matches it. Here $v : b_1 + b_2$ can be of the form $left\ v_1$ or $right\ v_2$. Assuming $left\ v_1$, we have:

$$\frac{c_1\ v_1 \mapsto v_1' \Longrightarrow \langle c_1, v_1, C'\rangle \mapsto^* [c_1, v_1', C']}{c_1 + c_2\ (left\ v_1) \mapsto left\ v_1' \Longrightarrow ?}$$

Choosing $C' = L^{+}\ c_2\ C$, we have the required derivation sequence: $\langle c_1 + c_2, left\ v_1, C\rangle \mapsto \langle c_1, v_1, L^{+}\ c_2\ C\rangle \mapsto^* [c_1, v_1', L^{+}\ c_2\ C] \mapsto [c_1 + c_2, left\ v_1', C]$. The proof follows similarly for the $right\ v_2$ case.

To show the right-to-left direction we proceed by induction of the sequence of $\mapsto^*$ derivations. Again the case for primitive isomorphisms follows trivially. Taking the case of $c_1 + c_2$ we are required to show that given a sequence $\langle c_1 + c_2, v, C\rangle \mapsto^* [c_1 + c_2, v', C]$ there is a derivation tree for $c_1 + c_2\ v \mapsto v'$. In the case that $v : v_1 + b_2$ has the form $left\ v_1$, this follows by observing that the given sequence must have the form $\langle c_1 + c_2, left\ v_1, C\rangle \mapsto \langle c_1, v_1, L^{+}\ c_2\ C\rangle \mapsto^* [c_1, v_1', L^{+}\ c_2\ C] \mapsto [c_1 + c_2, left\ v_1', C]$. For the strictly smaller inner subsequence by induction we have $c_1\ v_1 \mapsto v_1'$ which lets us complete the derivation of $c_1 + c_2\ (left\ v_1) \mapsto left\ v_1'$. The $right\ v_2$ follows similarly.

Consequently the small step semantics is logically reversible and strong normalizing.

PROPOSITION 2.7 (Type Safety).

### 2.5 Graphical Language

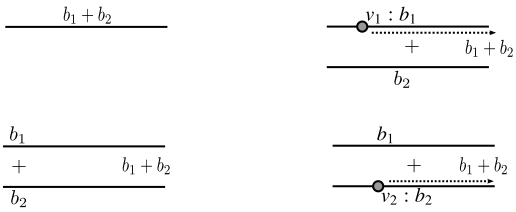
The syntactic notation above is often obscure and hard to read. Following the tradition established for monoidal categories [27], we present a graphical language that conveys the intuitive semantics of the language.

The general idea of the graphical notation is that combinators are modeled by "wiring diagrams" or "circuits" and that values are modeled as "particles" or "waves" that may appear on the wires. Evaluation therefore is modeled by the flow of waves and particles along the wires.

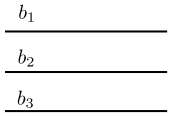
- The simplest sort of diagram is the $id : b \leftrightarrow b$ combinator which is simply represented as a wire labeled by its type $b$, as shown on the left. In more complex diagrams, if the type of a wire is obvious from the context, it may be omitted. When tracing a computation, one might imagine a value $v$ of type $b$ on the wire, as shown on the right.


- The product type $b_1 \times b_2$ may be represented using either one wire labeled $b_1 \times b_2$ or two parallel wires labeled $b_1$ and $b_2$. In the case of products represented by a pair of wires, when tracing execution using particles, one should think of one particle on each wire or alternatively as in folklore in the literature on monoidal categories as a "wave."


- Sum types may similarly be represented by one wire or using parallel wires with a $+$ operator between them. When tracing the execution of two additive wires, a value can reside on only one of the two wires.

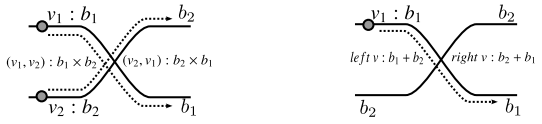



- Associativity is implicit in the graphical language. Three parallel wires represent $b_1 \times (b_2 \times b_3)$ or $(b_1 \times b_2) \times b_3$, based on the context.

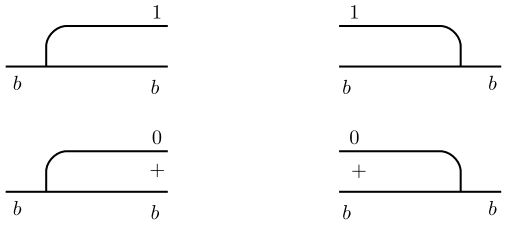



- Commutativity is represented by crisscrossing wires.

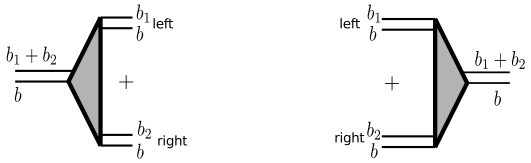

By visually tracking the flow of particles on the wires, one can verify that the expected types for commutativity are satisfied.



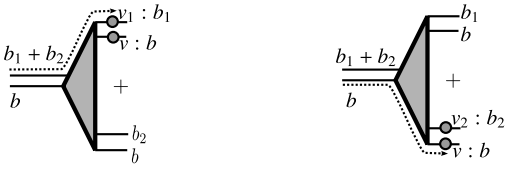

- The morphisms that witness that 0 and 1 are the additive and multiplicative units are represented as shown below. Note that since there is no value of type 0, there can be no particle on a wire of type 0. Also since the monoidal units can be freely introduced and eliminated, in many diagrams they are omitted and dealt with explicitly only when they are of special interest.



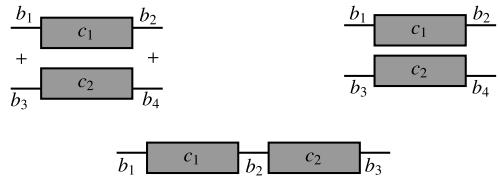

- Finally, distributivity and factoring are represented using the dual boxes shown below:




Distributivity and factoring are interesting because they represent interactions between sum and pair types. Distributivity should essentially be thought of as a multiplexer that redirects the flow of $v : b$ depending on what value inhabits the type $b_1 + b_2$, as shown below. Factoring is the corresponding adjoint operation.




- Combinators can be composed is series ($c_1 \fatsemi c_2$) or parallel. There are two forms of parallel composition – combinators can be combined additively $c_1 + c_2$ (shown on the left) or multiplicatively $c_1 \times c_2$ (shown on the right).




### 2.6 Categorical Structure

We present the categorical structure with minimum commentary. More details may be found in excellent references such as Barr and Wells [CITE] and Selinger [27]. For brevity, we don't restate standard categorical definitions and subsequently present their $\Pi$ equivalents. Instead, we directly state categorical definitions in terms of $\Pi$ types and combinators, and provide appropriate references to the former. For example, associativity is denoted by the Greek letter $\alpha$ (and $\alpha^{-1}$) in standard definitions, whereas we use $assocl^{\times}$ (and $assocr^{\times}$).

LEMMA 2.8 ($\Pi$ is a category). *The category $\Pi$ has the types $b$ as objects and equivalence classes of well-typed combinators $b_1 \leftrightarrow b_2$ as morphisms. One can check:*

1. *Every object $b$ has an identity morphism $id : b \leftrightarrow b$.*
2. *Composition $g \circ f$ of morphisms $f : b_1 \leftrightarrow b_2$ and $g : b_2 \leftrightarrow b_3$ is given by sequencing $f \fatsemi g$.*

3. *Associativity of composition follows from operational equivalence of $f \fatsemi (g \fatsemi h)$ and $(f \fatsemi g) \fatsemi h$ (where $h : b_3 \leftrightarrow b_4$). Assuming $v_1 : b_1$, $v_2 : b_2$, $v_3 : b_3$, $v_4 : b_4$, $f\ v_1 \mapsto v_2$, $g\ v_2 \mapsto v_3$ and $h\ v_3 \mapsto v_4$, one can check:*

$$\dfrac{f\ v_1 \mapsto v_2 \qquad \dfrac{g\ v_2 \mapsto v_3 \quad h\ v_3 \mapsto v_4}{g \fatsemi h\ v_2 \mapsto v_4}}{f \fatsemi (g \fatsemi h)\ v_1 \mapsto v_4} \qquad \dfrac{\dfrac{f\ v_1 \mapsto v_2 \quad g\ v_2 \mapsto v_3}{f \fatsemi g\ v_1 \mapsto v_3} \qquad h\ v_3 \mapsto v_4}{(f \fatsemi g) \fatsemi h\ v_1 \mapsto v_4}$$

4. *Composition respects identity: $id \fatsemi f = f$ and $g \fatsemi id = g$.*

LEMMA 2.9 (Dagger). $\Pi$ *is a dagger category, where every morphism $f : b_1 \leftrightarrow b_2$ has the adjoint $f^\dagger : b_2 \leftrightarrow b_1$. The following properties hold (where $g : b_2 \leftrightarrow b_3$):*

1. $id^\dagger = id : b \leftrightarrow b$.
2. $(f \fatsemi g)^\dagger = g^\dagger \fatsemi f^\dagger : b_3 \leftrightarrow b_1$.
3. $f^{\dagger\,\dagger} = f : b_1 \leftrightarrow b_2$.

LEMMA 2.10 (Symmetric Monoidal (+, 0)). $\Pi$ *is a symmetric monoidal category with tensor $+$ and monoidal unit $0$. The monoidal operation on morphisms is the additive composition of combinators $c_1 + c_2$.*

***Proof.*** *To establish that category is monoidal one must show isomorphisms*

1. $b_1 + (b_2 + b_3) \leftrightarrow (b_1 + b_2) + b_3$ *given by* $assocl^+$.
2. $0 + b \leftrightarrow b$ *given by* $zeroe$.
3. $b + 0 \leftrightarrow b$ *given by* $swap^+ \fatsemi zeroe$.
4. *We need to check that $+$ is a bifunctor.*
   (a) $id_{b_1 \leftrightarrow b_1} + id_{b_2 \leftrightarrow b_2} = id_{b_1+b_2 \leftrightarrow b_1+b_2}$.
   (b) $(f + g) \fatsemi (j + k) = (f \fatsemi j) + (g \fatsemi k)$.
5. *We need to check the naturality of $assocl^+$, $zeroe$ and $swap^+ \fatsemi zeroe$.*
   (a) $assocl^+ \fatsemi ((f + g) + h) = (f + (g + h)) \fatsemi assocl^+$.
   (b) $zeroe \fatsemi f = (id + f) \fatsemi zeroe$.
   (c) $(swap^+ \fatsemi zeroe) \fatsemi f = (f + id) \fatsemi (swap^+ \fatsemi zeroe)$.
6. *Satisfy certain coherence conditions which are usually called the "pentagon" and "triangle" axioms (see Sec 3.1 [27])*

*The last three points require checking equality of combinators by writing out their derivation trees as we did in the case of associativity of sequential composition. To show symmetry, we need a braiding operation $b_1 + b_2 \leftrightarrow b_2 + b_1$ which is given by $swap^+$ (see Sec. 3.3 and 3.5 [27]).*

1. *The braiding must satisfy two "hexagon" axioms.*
2. *The braiding is self inverse, $swap^+_{b_1+b_2} = (swap^+_{b_2+b_1})^\dagger$.*

LEMMA 2.11 (Symmetric Monoidal $(\times, 1)$). $\Pi$ *is a symmetric monoidal category over the tensor $\times$ and unit $1$. The details mirror those of the $(0, +)$ monoid.*

Some technical and pedantic comments are due at this point.

- We have established the categorical structure of $\Pi$ as a dagger symmetric monoidal category with two monoidal structures, $(0, +)$ and $(\times, 1)$. In Sec. 3 we will see that a *trace* operator can be admitted in this category without any change of expressiveness.
- To be pedantic, what we have shown is that the "term model" of $\Pi$ that follows from the extensional operational equality of combinators has the requisite categorical structure. A consequence is that the "wiring diagrams" of $\Pi$ correspond closely with "string diagrams" developed for categories.

  To establish the later rigorously, we will need to show when it is valid to slide one wire over the other and that equivalent diagrams for syntactically different combinators such as $(f + g) \fatsemi (j + k)$ and $(f \fatsemi j) + (g \fatsemi k)$ do respect operational equivalence. We don't formalize the graphical notation in this work. Joyal et. al's work on "planar isotopy" and Selinger's survey [27] show how this has been addressed before in the categorical setting. In the absence of any prior knowledge of category theory however, our wiring diagrams may be read as the "flow of types" in a combinator-circuit.
- By definition, in $\Pi$ every morphism is an isomorphism – this makes $\Pi$ a groupoid.
- The category $\Pi$ has no initial and terminal objects. The objects $0$ and $1$ would be initial and terminal if we admitted all functions (as in the category **Set**).
- The category $\Pi$ has neither categorical products, nor categorical co-products, i.e. $\times$ and $+$ are merely monoidal tensors. This follows from the fact that injection and projection maps are not isomorphisms.

While we don't do so in this work, if we extend $\Pi$ with products and co-products (say through the addition of information effects) it is conceivable that part of its structure collapses. This sort of collapse, while catastrophic for algebraic structures (effectively trivializing them), still retains some interest in computing, because in computing we are interested in the specific operational nature (the computational content, so to speak) of the morphisms. Anecdotal evidence follows from the fact several real-world programming language have inconsistent type systems. blah blah blah...

## 3. Fractional Types : $\Pi^{\eta\epsilon}_{\times}$

In arithmetic the main identity satisfied by fractional numbers is that for non-zero $b$, we have $b \times 1/b = 1$. Accommodating this identity in the setting of $\Pi$ requires us to find an isomorphism between the types $1$ and $b \times 1/b$ (where $b$ is not equivalent to $0$). Formally, we move from the setting of symmetric monoidal categories to that of compact closed categories.

$$\begin{array}{rcl} \mathit{Value\ Types}, b & = & 0 \mid 1 \mid b + b \mid b \times b \mid 1/b \\ \mathit{Values}, v & = & () \mid \mathit{left}\ v \mid \mathit{right}\ v \mid (v, v) \mid 1/v \\ \\ \mathit{Isomorphisms}, iso & = & \ldots \mid \eta^\times \mid \epsilon^\times \end{array}$$

***Traces and the Int-Construction.*** For a traced symmetric monoidal category, the Int-construction of Joyal, Street and Verity [17] gives us a way of constructing a compact closed category. This is the obvious first thing to try for a duality in this setting.

$$\dfrac{c : b_1 + b_2 \leftrightarrow b_1 + b_3}{\mathit{trace}\ c : b_2 \leftrightarrow b_3}$$

Show that the int construction does not preserve the $\times$ tensor. We need a duality that preserves both tensors.

***No division by zero.*** The important side-condition on the types not captured by the syntax above is that there are no types of the form $1/b$, where $\lceil b \rceil = 0$ (see Def. 2.2 and 3.1 below). This eliminates terms of the form $1/0$, $1/(0 + 0)$, $1/(0 \times b)$ etc, from being treated as valid $b$ types. Intuitively, this translates into the familiar restriction that there is no division by 0 in arithmetic, nor in algebraic fields.

$$\eta^{\times} \quad : 1 \leftrightarrow (1/b) \times b :: \quad \epsilon^{\times} \qquad \frac{\vdash v : b}{\vdash 1/v : 1/b}$$

For the graphical language, we visually represent $\eta^{\times}$, and $\epsilon^{\times}$ as U-shaped connectors. On the left below is $\eta^{\times}$ showing the map and 1 to $1/b \times b$. On the right is $\epsilon^{\times}$ showing the map from $1/b \times b$ to 1. Even though the diagrams below show the 1 wires for completeness, later diagrams will always drop them in contexts where they can be implicitly introduced and eliminated.


The usual interpretation of $b_1 \times b_2$ we have both a value of type $b_1$ and a value of type $b_2$. This interpretation is maintained in the presence of fractionals. Hence an $\eta^{\times} : 1 \leftrightarrow (1/b) \times b$ is to be viewed as a fission point for a value of type $b$ and its multiplicative inverse $1/b$.


Negative Information and Measurements.

DEFINITION 3.1 (Size of a type.). *The definition of size, $\lceil b \rceil$, is extended to include fractionals as $\lceil 1/b \rceil = 1/\lceil b \rceil$.*

### 3.1 Semantics

The operational semantics for $\Pi_{\times}^{\eta\epsilon}$ involve the addition of the following two rules to the small-step operational semantics of $\Pi$.

$$\begin{array}{rcll} \langle \eta^{\times}, (), C \rangle & \mapsto & \forall\, v : b.[\eta^{\times}, (1/v, v), C] & (\eta^{\times}) \\ \langle \epsilon^{\times}, (1/v, v), C \rangle & \mapsto & [\epsilon^{\times}, (), C] & (\epsilon^{\times}) \end{array}$$

The rule $(\eta^{\times})$ is different from other rules encountered before due to the $\forall\ v\ :\ b$ quantification. It is to be understood as follows: when the machine encounters an $\eta^{\times}$ reduction it nondeterministically replicates into several machine states (or worlds). Each world is seeded by a unique value $v : b$, where $b$ comes from $\eta^{\times} : 1 \leftrightarrow 1/b \times b$. For example, at $\eta^{\times} : 1 \leftrightarrow 1/bool \times bool$, we would have two possible machine states:

$$\begin{array}{rcl} \langle \eta^{\times}, (), C \rangle & \mapsto & [\eta^{\times}, (1/true, true), C] \\ and & \mapsto & [\eta^{\times}, (1/false, false), C] \end{array}$$

The rule $(\epsilon^{\times})$ is straightforward : when given a tuple $(1/v, v) : 1/b \times b$ where $\epsilon^{\times} : 1/b \times b \leftrightarrow 1$ the result is $()$. The machine is undefined for cases where where the input is $(1/v', v) : 1/b \times b$ and $v' \neq v$.

***Annihilation.*** A consequence of the semantics is that they admit computations that are undefined. For instance consider, where $not : bool \leftrightarrow bool$.

$\eta^{\times} \fatsemi (id \times not) \fatsemi \epsilon^{\times} : 1 \leftrightarrow 1$

Tracing the execution of this circuit gives us:

$$\begin{array}{rcl} & & \langle \eta^{\times} \fatsemi (id \times not) \fatsemi \epsilon^{\times}, (), \Box \rangle \\ & \mapsto^* & \langle \eta^{\times}, (), Fst\ \Box\ (id \times not) \fatsemi \epsilon^{\times} \rangle \\ (1) & \mapsto & [\eta^{\times}, (1/true, true), Fst\ \Box\ (id \times not) \fatsemi \epsilon^{\times}] \\ & \mapsto^* & \langle not, true, R^{\times}\ id\ 1/true\ (Fst\ (Snd\ \eta^{\times} \Box)\ \epsilon^{\times}) \rangle \\ & \mapsto^* & [not, false, R^{\times}\ id\ 1/true\ (Fst\ (Snd\ \eta^{\times} \Box)\ \epsilon^{\times})] \\ & \mapsto^* & [\epsilon^{\times}, (1/true, false), Snd\ (id \times not)\ (Snd\ \eta^{\times} \Box)] \\ & \mapsto & undefined \\ (2) & \mapsto & [\eta^{\times}, (1/false, false), Fst\ \Box\ (id \times not) \fatsemi \epsilon^{\times}] \\ & \mapsto^* & \langle not, false, R^{\times}\ id\ 1/false\ (Fst\ (Snd\ \eta^{\times} \Box)\ \epsilon^{\times}) \rangle \\ & \mapsto^* & [not, true, R^{\times}\ id\ 1/false\ (Fst\ (Snd\ \eta^{\times} \Box)\ \epsilon^{\times})] \\ & \mapsto^* & [\epsilon^{\times}, (1/false, true), Snd\ (id \times not)\ (Snd\ \eta^{\times} \Box)] \\ & \mapsto & undefined \end{array}$$

PROPOSITION 3.2 (Type Safety).

PROPOSITION 3.3 (Strong Normalizing).

We use $c\ v\ \mapsto^*\ v'$ as a shorthand for the reduction sequence $\langle c, v, \Box \rangle \mapsto^* [c, v', \Box]$.

DEFINITION 3.4 (eval(c,v)). *Given a combinator $c : b_1 \leftrightarrow b_2$ and a value $v : b_1$ we define $eval(c, v)$ to be the set of values $v' : b$ such that there is small-step reduction sequence $c\ v \mapsto^* v'$.*

$eval(c, v) = \{v' \mid c\ v \mapsto^* v'\}$

The above definition implies that combinators $c : b_1 \leftrightarrow b_2$ denote relations between the sets $b_1$ and $b_2$, i.e. $c : b_1 \leftrightarrow b_2 = \{(v, v') | v : b_2, v' : b_2\ and\ c\ v \mapsto^* v'\}$. The definition of combinator equality reflects the fact that combinators denote relations.

DEFINITION 3.5 ($c_1 = c_2$). *We say that $c_1 = c_2$, for combinators $c_1 : b_1 \leftrightarrow b_2$ and $c_2 : b_1 \leftrightarrow b_2$ iff they denote the same relation. Equivalently, $\forall\ v : b_1 . eval(c_1, v) = eval(c_2, v)$.*

PROPOSITION 3.6 (Logical Reversibility).

$c\ v \mapsto^* v'$ *iff* $c^{\dagger} v' \mapsto^* v$

In other words, for any execution path from $v$ to $v'$ in $c$, $c^{\dagger}$ has an execution path from $v'$ to $v$. Intuitively this maybe understood as follows: For every particular execution sequence $\eta^{\times}$ transforms $()$ into particular $(1/v, v)$. On reverse execution the adjoint $\epsilon^{\times}$ will transform $(1/v, v)$ back to $()$. Logical reversibility implies that $c^{\dagger}$ denotes the inverse relation that $c$ denotes, i.e. $c^{\dagger} = \{(v', v) \mid (v, v') \in c\}$.

***Union.*** The union of the two relations $c_1 : b_1 \leftrightarrow b_2$ and $c_2 : b_1 \leftrightarrow b_2$ is expressed by $trace^{\times}\, if_{c_1,c_2} : b_1 \leftrightarrow b_2$ (see Sec. 2.3 for $if_{c_1,c_2}$ and Sec. **??** for $trace^{\times}$). Intersection can be expressed by "controlled-annihilation" detailed in Sec. 4.

### 3.2 Categorical Structure

Like $\Pi$, $\Pi_{\times}^{\eta\epsilon}$ is a commutative bimonoidal category. Symmetric monoidal categories where every object had a dual object and where dualizing preserves the monoidal tensor are referred to as compact closed categories. Since division is defined only for types of non-zero size, not all objects have duals. We show that $\Pi_{\times}^{\eta\epsilon}$ respects compact closed structure for non-zero objects. We introduce the term "locally compact closed" to refer to this notion.

It is important to note that in a compact closed category, the dualizing action, usually denoted $(-)^*$, is a contravariant functor. However $\Pi_{\times}^{\eta\epsilon}$ is not compact closed and $1/(-)$ is not a functor specifically because division is not defined on 0 sized types.

**Roshan:** ***This is an important question – is $\Pi_{\times}^{\eta\epsilon}$ still bimonoidal? i.e. do all the coherence conditions on isomorphisms naturally extend to relations. I suspect yes.***

PROPOSITION 3.7 (Locally Compact Closed). *For every non-zero object $b$ of $\Pi_\times^{\eta\epsilon}$, there exists an inverse object $1/b$ in the $(\times, 1)$ symmetric monoid with the morphisms:*

1. $1 \leftrightarrow 1/b \times b$ *given by* $\eta^\times$*, called the 'unit map'.*
2. $b \times 1/b \leftrightarrow 1$ *given by* $swap^\times \fatsemi \epsilon^\times$*, called the 'counit map'.*

*such that the following coherence conditions hold (see 2.1 in [26]):*

1. $uniti \fatsemi swap^\times \fatsemi (id \times \eta^\times) \fatsemi assocl^\times \fatsemi ((swap^\times \fatsemi \epsilon^\times) \times id) \fatsemi unite = id : b \leftrightarrow b$
2. $uniti \fatsemi (\eta^\times \times id) \fatsemi assocr^\times \fatsemi (id \times (swap^\times \fatsemi \epsilon^\times)) \fatsemi swap^\times \fatsemi unite = id : 1/b \leftrightarrow 1/b$

**Proof.** For convenience, we draw the combinator $swap^\times \fatsemi \epsilon^\times$ (shown on the left) as shown on the right.


The coherence conditions are intuitive when presented graphically. They require that the combinators below be equal to $id$, at the appropriate type (see Sec. 4 [27]).


We briefly trace the execution of the first combinator (let us call it $c$). To show that $c = id$, we must show for $v : b$, the evaluation of $\langle c, v, \square \rangle$ always produces $v : b$ as output. Verification for the second combinator is similar.

$$\begin{array}{rcll} & \langle c, v : b, \square \rangle & \\ & \langle \eta^\times, (), C_1 \rangle & \\ & \langle \eta^\times, (1/v', v'), C_2 \rangle & \textit{some } v' : b \\ (1) & \langle \epsilon^\times, (1/v', v), C_3 \rangle & \textit{when } v' = v \\ & [\epsilon^\times, (), C_3] & \\ & [c, v, \square] & \\ (2) & \langle \epsilon^\times, (1/v', v), C_3 \rangle & \textit{when } v' \neq v \\ & \textit{undefined} & \end{array}$$

### 3.3 Zero Totalized Fields

Without the important side condition on division by zero, we would have $1 \leftrightarrow 1/0 \times 0$ which implies $1 \leftrightarrow 0$ consequently collapsing the entire structure.

While fields require the "no division by zero" constraint, alternate algebraic structures such as meadows [4] have been studied which admit division and negative without this constraint. Such structures are called "zero totalized fields" and define multiplicative inverse of 0. The axiomatic presentation of meadows involve removing the combinators corresponding the axiom $0 \times b \leftrightarrow 0$ (i.e. $distrib_0$ and $factor_0$) and $1 \leftrightarrow 1/b \times b$ (i.e. $\eta^\times$ and $\epsilon^\times$). Instead meadows include axioms *reflexivity*, *refl* and *refr*, and *restricted inverse*, *ril* and *rir*, which are:

$$\begin{array}{rcl} refl: & 1/1/b \leftrightarrow b: & refr \\ ril: & b \times (b \times 1/b) \leftrightarrow b: & rir \end{array}$$

Combinators *refl* and *refr* are primitive and can be defined as:

$$\begin{array}{rcl} refl\ 1/(1/v) & \mapsto & v \\ refr\ v & \mapsto & 1/(1/v) \end{array}$$

Combinators *ril* and *rir* are defined as machine transitions similar to $\eta^\times/\epsilon^\times$:

$$\begin{array}{rcl} \langle ril, (v, (v', 1/v')), C \rangle & \mapsto & [ril, v, C] \\ \langle rir, v, C \rangle & \mapsto & \forall\, v' : b. [rir, (v, (v', 1/v')), C] \end{array}$$

While not as commonly used as fields, zero totalization is useful not just because we are spared from continuously checking the division constraint, but also because many other structures such as the category of vectors spaces do admit a dual to the 0 vector space. However their categorization is not well developed and we however lose the nice correspondence that $\eta^\times/\epsilon^\times$ have with compact closed structures.

## 4. SAT Solver

As we will see if Sec. 5.3, every compact closed category admits a *trace* operator. In the case of $\Pi_\times^{\eta\epsilon}$, the operator $trace^\times$ is defined only on the non-zero fragment. Given $f : a \times c \leftrightarrow b \times c$, we have $trace^\times f : a \leftrightarrow b$:

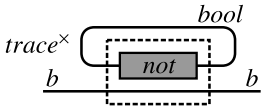
$$trace^\times f = uniti \fatsemi (id \times \eta^\times) \fatsemi (f \times id) \fatsemi (id \times \epsilon^\times) \fatsemi unite$$

This circuit uses $\eta^\times$ to generate all possible $c$-values together with an associated $(1/c)$-constraint. It then applies $f$ to the pair $(a, c)$. The function $f$ must produce an output $(b, c')$ for each such input. If the input $c$ and the output $c'$ are the same they can be annihilated by $\epsilon^\times$; otherwise the execution gets stuck and this particular choice of $c$ is pruned.

A large class of constraint satisfaction problems can be expressed using $trace^\times$. We illustrate the main ideas with the implementation of a SAT-solver.

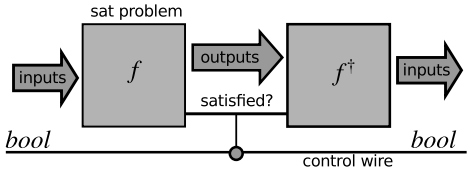
### 4.1 Construction of the Solver

The key insight underlying the construction comes from the fact that we can build *annihilation circuits* such as the one below:




The circuit constructs a boolean $b$ and its dual $1/b$, negates one of them and attempts to satisfy the constraint that they are equal which evidently fails.

With a little work, we can modify this circuit to only annihilate values that fail to satisfy the constraints represented by a SAT-instance $f$. In more detail, an instance of SAT is a function/circuit $f$ that given some boolean inputs returns *true* or *false* which we interpret as whether the inputs satisfy the constraints imposed by the structure of $f$. Because we are in a reversible world, our instance of SAT must be expressed as an isomorphism: this is easily achieved as shown in Sec. 4.2 below. Assuming that $f$ is expressed as an isomorphism, we have enough information to reconstruct the input from the output. This can be done by using the adjoint of $f$. At this point we have, the top half of the construction below:

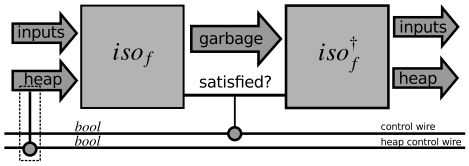



To summarize, the top half of the circuit is the identity function except that we have also managed to produce a boolean wire labeled satisfied? that tells us if the inputs satisfy the desired constraints. We can take this boolean value and use it to decide whether to negate the control wire or not. Thus, the circuit achieves the following goal: if the inputs do not satisfy $f$, the control wire is

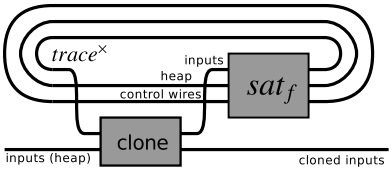
negated. We can now use $trace^{\times}$ to annihilate all these bad values because the control wire acts like the closed-loop $not$ in the previous construction.

### 4.2 Final Details

Any boolean expression $f : bool^n \to bool$ can be compiled into the isomorphism $iso_f : bool^h \times bool^n \leftrightarrow bool^g \times bool$ where the extra bits $bool^h$ and $bool^g$ are considered as heap and garbage. Constructing such an isomorphism has been detailed before [15, 28]. The important relation to note is that applying $iso_f$ to some special heap values and an input $bs$ produces some bits that can be ignored and the same output that $f$ would have produced on $bs$. We can ensure that the heap has the appropriate initial values by checking the heap and negating a second control wire, if the values do not match using, i.e., the dotted part in the diagram below. This is achieved by first applying $not$ to the control wire and then testing that all the heap values are true using $cnot^n$, undoing the $not$ only if they are.




Let us call the above construction which maps inputs, heap, and control wires to inputs, heap, and control wires as $sat_f$. The SAT-solver is is completed by tracing the $sat_f$ and cloning the inputs using $clone^n_{bool}$.




When the solver is fed inputs initialized to $true$, it clones only those inputs to $sat_f$ that satisfy $f$ and the heap constraints. In the case of unique-SAT the solver will produce exactly 0 or 1 solutions. In the case of general SAT, the solver will produce all satisfying solution.

## 5. Negative Types : $\Pi_+^{\eta\epsilon}$

Charles Pinter in his book on Abstract Algebra talks about how negatives were used by *logisticians* of a 1000 years ago used negative numbers as a "useful fiction". The concept of negative numbers were "discovered"much later than fractionals. Like Joan Baez says "half an apple is easier to understand than a negative apple".

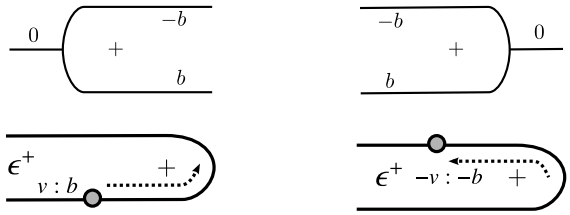
Negative types should correspond to sets with negative arity. But what does that mean? Various approaches have been proposed – Loeb uses the notion of multiplicity functions to express negative sized sets [CITE]. These notions have been extended to Euler characteristic and Homotopy cardinality of categories by Leinster, Joyal and Schanuel – negative values are explained sometimes as equalities relating otherwise independent objects – Baez tells a story about "island" and positive and "bridges connecting islands" as negatives (since the effectively reduce the number of disconnected elements.)

We use the use the syntax $-b$ for negative types and $-v$ for negative values and the axiom $0 \leftrightarrow (-b) + b$ as the basis for negatives. In $\Pi/\Pi_{\times}^{\eta\epsilon}$, values flowed from left-to-right in circuits. Negative types are negative in the sense that they flow in the opposite direction.

$$
\begin{array}{rcl}
\mathit{Value\ Types}, b & = & 0 \mid 1 \mid b + b \mid b \times b \mid -b \\
\mathit{Values}, v & = & () \mid \mathit{left}\ v \mid \mathit{right}\ v \mid (v, v) \mid -v \\
\\
\mathit{Isomorphisms}, \mathit{iso} & = & \ldots \mid \eta^+ \mid \epsilon^+
\end{array}
$$

For convenience, we sometimes use the notations $b_1 - b_2$ to indicate the types $b_1 + (-b_2)$. The types of the new constructs are:

$$\eta^+ \quad : 0 \leftrightarrow (-b) + b : \quad \epsilon^+ \qquad\qquad \frac{\vdash v : b}{\vdash -v : -b}$$




DEFINITION 5.1 (Size of a type). *The definition of size is extended to include negative types.* $\lceil -b \rceil = -\lceil b \rceil$

### 5.1 Semantics

Given that our language is reversible (Prop. 2.4), a backward evaluator is relatively straightforward to implement: using the backward evaluator to calculate $c\ v$ is equivalent $c^{\dagger}\ v$ in the forward evaluator.

$$
\begin{array}{rcl}
[\mathit{iso}, v, C]^{\dagger} & \mapsto & \langle \mathit{iso}, v', C\rangle^{\dagger} \\
 & & \mathit{where}\ \mathit{iso}^{\dagger}\ v \mapsto v' \\
\langle c_1, v, \mathit{Fst}\ C\ c_2\rangle^{\dagger} & \mapsto & \langle c_1 \fatsemi c_2, v, C\rangle^{\dagger} \\
\langle c_2, v, \mathit{Snd}\ c_1\ C\rangle^{\dagger} & \mapsto & [c_1, v, \mathit{Fst}\ C\ c_2]^{\dagger} \\
[c_1 \fatsemi c_2, v, C]^{\dagger} & \mapsto & [c_2, v, \mathit{Snd}\ c_1\ C]^{\dagger} \\
\langle c_1, v, L^+\ C\ c_2\rangle^{\dagger} & \mapsto & \langle c_1 + c_2, \mathit{left}\ v, C\rangle^{\dagger} \\
[c_1 + c_2, \mathit{left}\ v, C]^{\dagger} & \mapsto & [c_1, v, L^+\ C\ c_2]^{\dagger} \\
\langle c_2, v, R^+\ c_1\ C, s\rangle^{\dagger} & \mapsto & \langle c_1 + c_2, \mathit{right}\ v, C\rangle^{\dagger} \\
[c_1 + c_2, \mathit{right}\ v, C]^{\dagger} & \mapsto & [c_2, v, R^+\ c_1\ C]^{\dagger} \\
\langle c_1, v_1, L^{\times}\ C\ c_2\ v_2\rangle^{\dagger} & \mapsto & \langle c_1 \times c_2, (v_1, v_2), C\rangle^{\dagger} \\
\langle c_2, v_2, R^{\times}\ c_1\ v_1\ C\rangle^{\dagger} & \mapsto & [c_1, v_1, L^{\times}\ C\ c_2\ v_2]^{\dagger} \\
[c_1 \times c_2, (v_1, v_2), C]^{\dagger} & \mapsto & [c_2, v_2, R^{\times}\ c_1\ v_1\ C]^{\dagger}
\end{array}
$$

To add negative types we add the following rules to the reductions above. The additions formalize our previous discussions and should not be surprising at this point.

1. The rules for $\epsilon^+$ essentially transfer control from the forward evaluator (whose states are tagged by $\triangleright$) to the backward evaluator (whose states are tagged by $\triangleleft$). In other words, after an $\epsilon^+$ the direction of the world is reversed. The pattern matching done by the unification ensures that a value on the *right* wire is tagged to be negative and transferred to the *left* wire, and vice versa.

$$
\begin{array}{rcl}
\langle \epsilon^+, \mathit{right}\ v, C, s\rangle & \mapsto & \langle \epsilon^+, \mathit{left}\ (-v), C]\rangle^{\dagger} \\
\langle \epsilon^+, \mathit{left}\ (-v), C, s\rangle & \mapsto & \langle \epsilon^+, \mathit{right}\ v, C\rangle^{\dagger}
\end{array}
$$

Note that there is no evaluation rule for $\eta^+$ in the forward evaluator. This corresponds to the fact that there is no value of type 0 and hence the forward evaluator can never execute an $\eta^+$.

2. The rules for $\eta^+$ are added to the backward evaluator. A program executing backwards starts executing forwards after the execution of the $\eta^+$. Dual to the previous case, there is no rule for $\epsilon^+$ in the backward evaluator since the output type of $\epsilon^+$ is 0.

$$\begin{array}{rcl}[\eta^+, \mathit{right}\ v, C, s]^\dagger & \mapsto & [\eta^+, \mathit{left}\ (-v), C] \\ [\eta^+, \mathit{left}\ (-v), C, s]^\dagger & \mapsto & [\eta^+, \mathit{right}\ v, C]\end{array}$$

Consider the combinator $c : (-1+1)+1 \leftrightarrow 1$ defined below. On providing the value $\mathit{right}\ () : (-1+1)+1$ we get the output $() : 1$. But on supply input $\mathit{left}\ (\mathit{right}\ ())$ evaluation stop on the reverse interpreter with value $\mathit{left}\ (\mathit{left}\ ()) : (-1+1)+1$.

By definition, since negative values flow in the opposite direction of positive values, this is not surprising. Combinators can terminate in the forward interpreter (with result of type $b_2$) or in the reverse interpreter (with result of type $b_1$). To define an *eval* in a consistent manner without worrying about state of the evaluator, we transform every combinator of the form $c : b_1 \leftrightarrow b_2$ to one of the form $c : -b_2 + b_1 \leftrightarrow 0$ by the construction $action(c) = id + c \fatsemi \epsilon^+$.

DEFINITION 5.2 ($eval(c, v)$). *Given $c : b_1 \leftrightarrow b_2$ and $v : b_1$ we define $eval(c, v) = v'$ such that*

1. *$\mathit{right}\ v : -b_2 + b_1$, $v' : -b_2 + b_1$ and*
2. *$\langle action(c), \mathit{right}\ v, \Box\rangle \mapsto^* \langle action(c), v', \Box\rangle^\dagger$*

DEFINITION 5.3 ($c_1 = c_2$). *For $c_1 : b_1 \leftrightarrow b_2$ and $c_2 : b_1 \leftrightarrow b_2$, we define $c_1 = c_2$ if $\forall\ v : b_1 . eval(c_1, v) = eval(c_2, v)$.*

PROPOSITION 5.4 (Logical Reversibility).

PROPOSITION 5.5 (Strong Normalization).

PROPOSITION 5.6 (Type Safety).

### 5.2 Categorical Structure

As before, we can show that $\Pi_+^{\eta\epsilon}$ is a bimonoidal category. Additionally it has a compact closed structure in the $(0, +)$ monoid.

PROPOSITION 5.7 (Compact Closed $(0, +, -)$). *$\Pi_+^{\eta\epsilon}$ is a compact closed category with over the symmetric monoid $(0, +)$ with the dual of objects $b$ give by $-b$ with morphisms:*

1. *$0 \leftrightarrow -b + b$ given by $\eta^+$, called the 'unit map'.*
2. *$b + -b \leftrightarrow 0$ given by $swap^+ \fatsemi \epsilon^+$, called the 'counit map'.*

*such that the following coherence conditions hold (see 2.1 in [26]):*

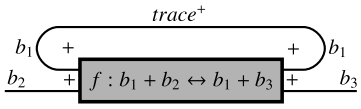
1. $zeroi \fatsemi swap^+ \fatsemi (id + \eta^+) \fatsemi assocl^+ \fatsemi ((swap^+ \fatsemi \epsilon^+) + id) \fatsemi zeroe = id : b \leftrightarrow b$
2. $zeroi \fatsemi (\eta^+ + id) \fatsemi assocr^+ \fatsemi (id + (swap^+ \fatsemi \epsilon^+)) \fatsemi swap^+ \fatsemi zeroe = id : -b \leftrightarrow -b$

### 5.3 Categorical Constructions

All the constructions below are standard: they are collected from Selinger's survey paper on monoidal categories [27] and presented in the context of our language.

We now review several interesting constructions related to looping, involution, and higher order functions.

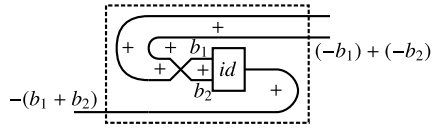
***Trace.*** Every compact closed category admits a trace. For the additive case, we get the following definition. Given $f : b_1 + b_2 \leftrightarrow b_1 + b_3$, define $trace^+ f : b_2 \leftrightarrow b_3$ as:
$trace^+ f = zeroi \fatsemi (id + \eta^+) \fatsemi (f + id) \fatsemi (id + \epsilon^+) \fatsemi zeroe$




We have omitted some of the commutativity and associativity shuffling to communicate the main idea. We are given a value of type $b_2$ which we embed into $0 + b_2$ and then $(-b_1 + b_1) + b_2)$. This can be re-associated into $-b_1 + (b_1 + b_2)$. The component $b_1 + b_2$, which until now is just an appropriately tagged value of type $b_2$, is transformed to a value of type $b_1 + b_3$ by $f$. If the result is in the $b_3$-summand, it is produced as the answer; otherwise the result is in the $b_1$-summand; $\epsilon^+$ is used to make it flow backwards to be fed to the $\eta^+$ located at the beginning of the sequence. Iteration continues until a $b_3$ is produced.

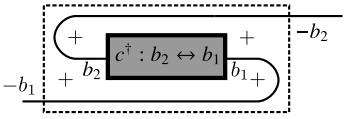
***Involution (Principium Contradictiones)*** In a symmetric compact closed category, we can build isomorphisms that the dual operation is an involution. Specifically, we get the isomorphisms $b \leftrightarrow -(-b)$ and $b \leftrightarrow (1/(1/b))$. For the additive case, the isomorphism is defined as follows:
$(id + \eta^+) \fatsemi (swap^+ + id) \fatsemi (id + \epsilon^+)$
where we have omitted the 0 introduction and elimination. The idea is as follows: we start with a value of type $b$, embed it into $b + 0$ and use $\eta$ to create something of type $b + (-(-b) + (-b))$. This is possible because $\eta$ has the polymorphic type $-a + a$ which can be instantiated to $-b$. We then reshuffle the type to produce $-(-b) + (-b + b)$ and cancel the right hand side using $\epsilon^+$. The construction for the multiplicative case is identical and omitted.

***Duality preserves the monoidal tensor.*** As with compact closed categories, the dual on the objects distributes over the tensor. In terms of $\Pi^{\eta\epsilon}$ we have that $-(b_1 + b_2)$ can be mapped to $(-b_1) + (-b_2)$ and that $1/(b_1 \times b_2)$ can be mapped to $(1/b_1) \times (1/b_2)$. The isomorphism $-(b_1 + b_2) \leftrightarrow (-b_1) + (-b_2)$ can be realized as follows:




The multiplicative construction is similar.

***Duality is a functor.*** Duality in $\Pi^{\eta\epsilon}$ can map objects to their duals and morphisms to act on dual objects. In other $c : b_1 \leftrightarrow b_2$ to $neg\ c : -b_1 \leftrightarrow -b_2$ in the additive monoid and to $inv\ c : 1/b_1 \leftrightarrow 1/b_2$ in the multiplicative monoid. The idea is simply to reverse the flow of values and use the adjoint of the operation:




$$\frac{c : b_1 \leftrightarrow b_2}{neg\ c : (-b_1) \leftrightarrow (-b_2)}$$

This construction relies on the fact that every $\Pi^{\eta\epsilon}$ morphism has an adjoint. The $inv$ construction is similar.

## 6. Computing in the Field of Rationals : $\Pi^{\eta\epsilon}$

Not logically reversible and we can express infinite loops.

***Observability.*** The reductions above allow us to apply a program $c : b_1 \leftrightarrow b_2$ to an input $v_1 : b_1$ to produce a result $v_2 : b_2$ on termination. Execution is well defined only if $b_1$ and $b_2$ are entirely positive types. If either $b_1$ or $b_2$ is a negative or fractional type, the system has "dangling" unsatisfied demands or constraints. For this reason, we constrain entire programs to have positive non-fractional types. This is similar to the constraint that Zeilberger imposes to explain intuitionistic polarity and delimited control [31].

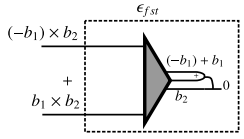
### 6.1 Constructions with Negatives and Fractionals

The additional constructions below (presented with minimal commentary) confirm that conventional algebraic manipulations in the mathematical field of rationals do indeed correspond to realizable type isomorphisms in our setting. The constructions involving both negative and fractional types are novel.

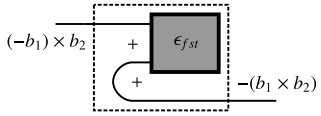
***Lifting negation out of*** $\times$**.** The isomorphisms below state that the direction is *relative*. If $b_1$ and $b_2$ are flowing opposite to each other then it doesn't matter which direction is forwards and which is backwards. More interestingly as $b_1$ is moving backwards, it can "see the past" of $b_2$ which is equivalent to both particles moving backwards.

$(-b_1) \times b_2 \leftrightarrow -(b_1 \times b_2) \leftrightarrow b_1 \times (-b_2)$

To build these isomorphisms, we first build an intermediate construction which we call $\epsilon_{fst} : (-b_1) \times b_2 + b_1 \times b_2 \leftrightarrow 0$.




The isomorphism $(-b_1) \times b_2 \leftrightarrow -(b_1 \times b_2)$ can be constructed in terms of $\epsilon_{fst}$ as shown below.




The second isomorphism can be built in the same way by merely swapping the arguments.

***Multiplying Negatives.*** $b_1 \times b_2 \leftrightarrow (-b_1) \times (-b_2)$

This isomorphism is a consequence of the fact that $-$ is an involution: it corresponds to the algebraic manipulation:

$b_1 \times b_2 = -(-(b_1 \times b_2)) = -((-b_1) \times b_2) = (-b_1) \times (-b_2)$

***Multiplying and Adding Fractions.*** An isomorphism witnessing:

$b_1/b_2 \times b_3/b_4 \leftrightarrow (b_1 \times b_3)/(b_2 \times b_4)$

is straightforward. More surprisingly, it is also possible to construct isomorphisms witnessing:

$b_1/b + b_2/b \leftrightarrow (b_1 + b_2)/b$

$b_1/b_2 + b_3/b_4 \leftrightarrow (b_1 \times b_4 + b_3 \times b_2)/(b_2 \times b_4)$

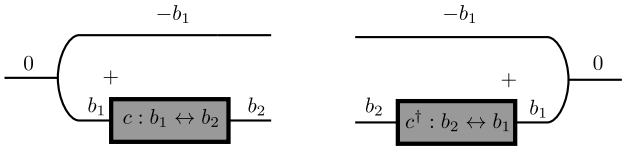
## 7. Two Dualities of Computation

Information Effects, Filinski, De Morgan

### 7.1 Function Spaces

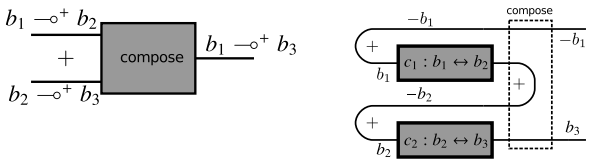
Although these constructions are also standard, they are less known and they are particularly important in our context: we devote a little more time to explain them. Our discussion is mostly based on Abramsky and Coecke's article on categorical quantum mechanics [1].

In a compact closed category, each morphism $f : b_1 \leftrightarrow b_2$ can be given a *name* and a *coname*. For the additive fragment, the name $\ulcorner f \urcorner$ has type $0 \leftrightarrow (-b_1 + b_2)$ and the coname $\llcorner f \lrcorner$ has type $b_1 + (-b_2) \leftrightarrow 0$. For the multiplicative fragment, the name $\ulcorner f \urcorner$ has type $1 \leftrightarrow ((1/b_1) \times b_2))$ and the coname $\llcorner f \lrcorner$ has type $(b_1 \times (1/b_2)) \leftrightarrow 1$. Intuitively, this means that for each morphism, it is possible to construct, from "nothing," an object in the category that denotes this morphism, and dually it is possible to eliminate this object. The construction of the name and coname of $c : b_1 \leftrightarrow b_2$ in the additive case can be visualized as follows:

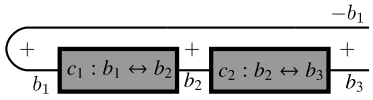



Intuitively the name consists of viewing $c$ as a function and the coname consists of viewing $c$ as a delimited continuation.

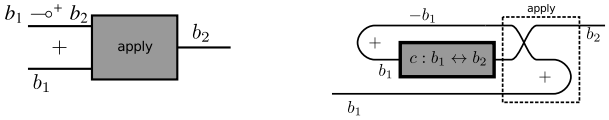
In addition to being able to represent morphisms, it is possible to express function composition. For the additive case, the composition is depicted below:




which is essentially equivalent to sequencing both the computation blocks as shown below:




Applying a function to an argument consists of making the argument flow backwards to satisfy the demand of the function:




Having reviewed the representation of functions, we now discuss the similarities and differences between the two notions of functions and their relation to conventional (linear) functions which mix additive and multiplicative components. For that purpose, we use a small example. Consider a datatype $color = R|G|B$, and let us consider the following manipulations:

- Using the fact that 1 is the multiplicative unit, generate from the input $()$ the value $((), ())$ of type $1 \times 1$;
- Apply the isomorphism $1 \leftrightarrow (1/b) \times b$ in parallel to each of the components of the above tuple. The resulting value is $((1/\alpha_1, \alpha_1), (1/\alpha_2, \alpha_2))$ where $\alpha_1$ and $\alpha_2$ are fresh logic variables;
- Using the fact that $\times$ is associative and commutative, we can rearrange the above tuple to produce the value:

  $((1/\alpha_1, \alpha_2), (1/\alpha_2, \alpha_1))$.

At this point we have constructed a strange mix of two $b \multimap^\times b$ functions; inputs of one function manifest themselves as outputs of the other. If $(1/\alpha_1, \alpha_2)$ is held by one subcomputation and $(1/\alpha_2, \alpha_1)$ is held by another subcomputation, these remixed functions form a communication channel between the two concurrent subcomputations. Unifying $1/\alpha_1$ with *color* $R$ in one subcomputation, fixes $\alpha_1$ to be $R$ in the other. The type $b$ thus takes the role of the type of the communication channel, indicating how much information can be communicated between the two subcomputations. Depending on the choice of the type $b$, an arbitrary number of bits may be communicated.

Dually, the additive reading of the above manipulations correspond to functions of the form $b \multimap^+ b$, witnessing isomorphisms

of the form $0 \leftrightarrow (-b) + b$). The remixed additive functions express control flow transfer between two subcomputations, *only one of which exists* at any point, i.e., they capture the essence of coroutines.

It should be evident that in a universe in which information is not guaranteed to be preserved by the computational infrastructure, the above slicing and dicing of functions would make no sense. But linearity is not sufficient: one must also recognize that the additive and multiplicative spaces are different.

## 8. Related Work

The idea of "negative types" has appeared many times in the literature and has often been related to some form of continuations. Fractional types are less common but have also appeared in relation to parsing natural languages. Although each of these previous occurrences of negative and fractional types is somewhat related to our work, our results are substantially different. To clarify this point, we start by reviewing the salient point of the major pieces of related work and conclude this section with a summary contrasting our approach to previous work.

***Declarative Continuations.*** In his Masters thesis [12], Filinski proposes that continuations are a *declarative* concept. He, furthermore, introduces a symmetric extension of the $\lambda$-calculus in which call-by-value is dual to call-by-name and values are dual to continuations. In more detail, the symmetric calculus contains a "value" fragment and a "continuation" fragment which are mirror images. Pairs and sums are treated as duals in the sense that the "value" fragment includes pairs whose mirror image in the "continuation" fragment are sums. In contrast, our language includes pairs and sums in the value fragment and two symmetries: one that maps the pairs to fractions and another that maps the sums to subtractions.

***The Duality of Computation.*** The duality between call-by-name and call-by-value was further investigated by Selinger using control categories [25]. Curien and Herbelin [11] also introduce a calculus that exhibits symmetries between values and continuations and between call-by-value and call-by-name. The calculus includes the type $A - B$ which is the dual of implication, i.e., a value of type $A - B$ is a context expecting a function of type $A \to B$. Alternatively a value of type $A - B$ is also explained as a *pair* consisting of a value of type $A$ and a continuation of type $B$. This is to be contrasted with our interpretation of a value of that type as *either* a value of $A$ or a demand for a value of type $B$. This calculus was further analyzed and extended by Wadler [29, 30]. The extension gives no interpretation to the subtraction connective and like the original symmetric calculus of Filinski, introduces a duality that relates sums to products and vice-versa.

***Subtractive Logic.*** Rauszer [20–22] introduced a logic which contains a dual to implication. Her work has been distilled in the form of *subtractive logic* [9] which has recently been related to coroutines [10] and delimited continuations [3]. In more detail, Crolard explains the type $A - B$ as the type of *coroutines* with a local environment of type $A$ and a continuation of type $B$. The description is complicated by what is essentially the desire to enforce linearity constraints so that coroutines cannot access the local environment of other coroutines.

***Negation in Classical Linear Logic*** Filinski [13] uses the negative types of linear logic to model continuations. Reddy [23] 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. Although a formal connection is lacking, it is clear that, at an intuitive level, acceptors are entities that combine elements of our negative and fractional types.

***The Lambek-Grishin Calculus.*** The "parsing-as-deduction" style of linguistic analysis uses the Lambek-Grishin calculus with the following types: product, left division, right division, sum, right difference, and left difference [5]. The division and difference types are similar to our types but because the calculus lacks commutativity and associativity and only has limited notions of distributivity, each connective needs a left and right version. The Lambek-Grishin exhibits two notions of symmetry but they are unrelated to our notions. In particular, the first notion of symmetry expresses commutativity and the second relates products to sums and divisions to subtractions. In contrast, our two symmetries relate sums to subtractions and products to divisions.

***Our Approach.*** The salient aspects of our approach are the following:

- 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 constraints on the context. None of the systems above has such a natural interpretation of negative and fractional types.
- Because we are *not* in the context of the full $\lambda$-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, every negative demand will be satisfied exactly once, and every constraint imposed by a fractional value will also be satisfied exactly once. This property is shared with systems that are based on linear logic; other systems must impose ad hoc constraints to ensure negative and fractional values are used exactly once.
- In contrast to all the work that takes continuations as primitive entities of negative types, we view continuations as a derived notion that combines a demand for a value with constraints on how this value will be used to proceed with the evaluation (to the closest delimiter or to the end of the program). In other words, we view a continuation as a non-elementary notion that combines the negative types to demand a value and the fractional types to explain how this value will be used to continue the evaluation. As a consequence, the previously observed duality between values and continuations can be teased into two dualities: a duality between values flowing in one direction or the other and a duality between aggregate values composing and decomposing into smaller values. Arguably each of the dualities is more natural than a duality that maps regular values to a conflated notion of negative and fractional types, and hence requires notions like "additive pairs" and "multiplicative sums."

## 9. Conclusion and Future Work

We have extended the language $\Pi$ that expressed computation in the commutative semiring of whole numbers to $\Pi^{\eta\epsilon}$ that expresses computation in the field of rationals. Every algebraic identity that holds for the rational numbers corresponds to a type isomorphism with a computational interpretation in our model. We have examined the two function spaces that arise in this model and developed non-trivial constructions such as a SAT-solver that relies on a multiplicative trace.

In another sense however, this paper is about the nature of duality in computation. The concept of duality is deep and significant:

we have opened the door for us to consider, not one but two notions of duality. Surprisingly this makes things substantially simpler. In particular, instead of conflating pairs as dual to sums, the tradition in mathematics has long been to consider fields with two notions of duality: one for sums and one for pairs. This double notion of duality has a crisp semantics, clear computational interpretation, and an information theoretic basis.

Our work has barely scratched the surface of an area of computing which has been explored in depth before but without the combined reversible information-preserving framework and the two notions of duality. The new insights point to further new areas of investigation, of which we mention the three most significant ones (in our opinion).

***Geometry of Interaction (GoI).*** Geometry of Interaction was developed by Girard [14] as part of the development of linear logic. It was given a computational interpretation by Abramsky and Jagadeesan [2], and was developed into a reversible model of computing by Mackie [18, 19]. Preliminary investigations suggest that many of the GoI machine constructions can be simulated in $\Pi^{\eta\epsilon}$ by treating Mackie's bi-directional wires as pairs of wires in $\Pi^{\eta\epsilon}$ and replacing the machine's global state with a typed value on the wire that captures the appropriate state. This connection is exciting because when viewed through a Curry-Howard lens it suggests that the logical interpretation of $\Pi^{\eta\epsilon}$ would be a linear-like logic with a notion of resource preservation and with a natural computational interpretation.

***Computing in the Field of Algebraic Numbers.*** Algebraically, the move from $\Pi$ to $\Pi^{\eta\epsilon}$ corresponds to a move from a ring-like structure to a full field. Our language $\Pi^{\eta\epsilon}$ captures the structure of one particular field: that of the rational numbers. As we have seen, computation in this field is quite expressive and interesting and yet, it has two fundamental limitations. First it cannot express any recursive type, and second it cannot express any datatype definitions. We believe these to be two orthogonal extensions: recursive types were considered in our previous paper [15]; arbitrary datatypes are however even more exciting that plain rationals as each datatype definition can be viewed as a polynomial (see below) which essentially means that we start computing in the field of algebraic numbers, which includes square roots and imaginary numbers. As crazy as it might seem, the type $\sqrt{2}$ and even the type $(1/2) + i(\sqrt{3}/2)$ "make sense." In fact the latter type is the solution to the polynomial $x^2 - x + 1 = 0$ which if re-arranged looks like $x = 1 + x \times x$ and perhaps more familiarly as the datatype of binary trees $\mu x.(1 + x \times x)$. These types happen to have been studied extensively following a paper by Blass [6] which used the above datatype of trees to infer an isomorphism between seven binary trees and one!

We have confirmed that we can extended $\Pi^{\eta\epsilon}$ with the datatype declaration for binary trees and build a witness for this isomorphism that works as expected. However not every isomorphism constructed from algebraic manipulation is computationally meaningful. To understand the issue in more detail, consider the following algebraically valid proof of the isomorphism in question:

$$\begin{array}{lllllllll} x^3 & = & x^2x & = & (x-1)x & = & x^2-x & = & -1 \\ x^6 & = & 1 & & & & & & \\ x^7 & = & x^6x & = & x & & & & \end{array}$$

The question is why such an algebraic manipulation makes sense type theoretically, even though the intermediate step asking for an isomorphism between $x^6$ and 1 has no computational context. In the setting of $\Pi^{\eta\epsilon}$, this isomorphism can be constructed but it diverges on all inputs (in both ways). This suggests that, in the field of algebraic numbers, some algebraic manipulations are somehow "more constructive" than others.

A related issue is that not all meaningful recursive types are meaningful polynomials. For instance $nat = \mu x.(1 + x)$ implies the polynomial $x = 1 + x$ which has no algebraic solutions without appeal to more complex structures with limits etc.

***Quantum Computing.*** One understanding of quantum computing is that it exploits the laws of physics to build faster machines (perhaps). Another more foundational understanding is that it provides a computational interpretation of physics, and in particular directly addresses the question of interpretation of quantum mechanics. In a little known document, Rozas [24] uses continuations to implement the transactional interpretation of quantum mechanics [8] which includes as its main ingredient a fixpoint calculation between waves or particles traveling forwards and backwards in time. Our work sheds no light on whether this interpretation is the "right one" but it is interesting that we can directly realize it using the primitives of $\Pi^{\eta\epsilon}$.

The multiplicative structure of $\Pi^{\eta\epsilon}$ also has a direct connection to entangled quantum particles, or perhaps entangled particles and anti-particles. The idea of entanglement, that an action on one particle is "instantaneously" communicated to the other, is analogous to how unifying one value affects its dual pair which is possibly in another part of the computation. Again our model sheds no light on whether this is related to how nature computes but it is again interesting that we can directly realize the idea using the primitives of $\Pi^{\eta\epsilon}$.

## Acknowledgments

We thank Jacques Carette for stimulating discussion, and Michael Adams, Will Byrd, Lindsey Kuper, and Yin Wang for helpful comments and questions. This project was partially funded by Indiana University's Office of the Vice President for Research and the Office of the Vice Provost for Research through its Faculty Research Support Program. We also acknowledge support from Indiana University's Institute for Advanced Study.

## References

[1] S. Abramsky and B. Coecke. Categorical quantum mechanics, 2008.

[2] S. Abramsky and R. Jagadeesan. New foundations for the geometry of interaction. *Inf. Comput.*, 111:53–119, May 1994.

[3] Z. M. Ariola, H. Herbelin, and A. Sabry. A type-theoretic foundation of delimited continuations. *Higher Order Symbol. Comput.*, 22:233–273, September 2009.

[4] J. A. Bergstra, Y. Hirshfeld, and J. V. Tucker. Meadows and the equational specification of division. *Theor. Comput. Sci.*, 410(12-13), 2009.

[5] R. Bernardi and M. Moortgat. Continuation semantics for the Lambek–Grishin calculus. *Inf. Comput.*, 208:397–416, May 2010.

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

[7] W. J. Bowman, R. P. James, and A. Sabry. Dagger Traced Symmetric Monoidal Categories and Reversible Programming. In *Workshop on Reversible Computation*, 2011.

[8] J. Cramer. The transactional interpretation of quantum mechanics. *Reviews of Modern Physics*, 58:647–688, 1986.

[9] T. Crolard. Subtractive logic. *Theoretical Computer Science*, 254(1-2):151–185, 2001.

[10] T. Crolard. A formulae-as-types interpretation of subtractive logic. *Journal of Logic and Computation*, 14(4):529–570, 2004.

[11] P.-L. Curien and H. Herbelin. The duality of computation. In *ICFP*, pages 233–243, New York, NY, USA, 2000. ACM.

[12] A. Filinski. Declarative continuations: an investigation of duality in programming language semantics. In *Category Theory and Computer Science*, pages 224–249, London, UK, 1989. Springer-Verlag.

[13] A. Filinski. Linear continuations. In *POPL*, pages 27–38. ACM Press, Jan. 1992.

[14] J. Girard. Geometry of interaction 1: Interpretation of system f. *Studies in Logic and the Foundations of Mathematics*, 127:221–260, 1989.

[15] R. P. James and A. Sabry. Information effects. In *POPL*, 2012.

[16] R. P. James and A. Sabry. Isomorphic Interpreters from Logically Reversible Abstract Machines. In *Workshop on Reversible Computation*, 2012.

[17] A. Joyal, R. Street, and D. Verity. Traced monoidal categories. *Math. Proc. Camb. Philos. Soc.*, 119(3):447–468, 1996.

[18] I. Mackie. The geometry of interaction machine. In *POPL*, pages 198–208, 1995.

[19] I. Mackie. Reversible higher-order computations. In *Workshop on Reversible Computation*, 2011.

[20] C. Rauszer. Semi-boolean algebras and their applications to intuitionistic logic with dual operators. *Fundamenta Mathematicae*, 83:219–249, 1974.

[21] C. Rauszer. A formalization of the propositional calculus of H-B logic. *Studia Logica*, 33:23–34, 1974.

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

[23] U. S. Reddy. Acceptors as values: Functional programming in classical linear logic. Manuscript, Dec. 1991.

[24] G. J. Rozas. A computational model for observation in quantum mechanics. Technical report, MIT, Cambridge, MA, USA, 1987.

[25] P. Selinger. Control categories and duality: on the categorical semantics of the lambda-mu calculus. *Mathematical. Structures in Comp. Sci.*, 11:207–260, April 2001.

[26] P. Selinger. Dagger compact closed categories and completely positive maps. *Electronic Notes in Theoretical Computer Science*, 170:139–163, 2007.

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

[28] T. Toffoli. Reversible computing. In *Proceedings of the Colloquium on Automata, Languages and Programming*, pages 632–644. Springer-Verlag, 1980.

[29] P. Wadler. Call-by-value is dual to call-by-name. In *ICFP*, pages 189–201, New York, NY, USA, 2003. ACM.

[30] P. Wadler. Call-by-value is dual to call-by-name - reloaded. In J. Giesl, editor, *RTA*, volume 3467 of *Lecture Notes in Computer Science*, pages 185–203. Springer, 2005.

[31] N. Zeilberger. Polarity and the logic of delimited continuations. In *LICS*, pages 219–227, Los Alamitos, CA, USA, 2010. IEEE Computer Society.