The Two Dualities of Computation: Negative and Fractional Types

Roshan P. James
Indiana University
rpjames@indiana.edu

Amr Sabry
Indiana University
sabry@indiana.edu

Abstract

Every functional programmer knows about sum and product types, $a+b$ and $a \times b$ respectively. Negative and fractional types, $a-b$ and a/b respectively, are much less known and their computational interpretation is unfamiliar and often complicated. We show that in a programming model in which information is preserved (such as the model introduced in our recent paper on *Information Effects*), these types have particularly natural computational interpretations. 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. The combination of these negative and fractional types enables greater flexibility in programming by breaking global invariants into local ones that can be autonomously satisfied by a subcomputation. Theoretically, these types give rise to *two* function spaces and to *two* notions of continuations, suggesting that the previously observed duality of computation conflated two orthogonal notions: an additive duality that corresponds to backtracking and a multiplicative duality that corresponds to constraint propagation.

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

In a recent paper [16], we argued that, because they include irreversible physical primitives, conventional abstract models of computation have inadvertently included some *implicit* computational effects which we called *information effects*. We then developed a pure reversible model of computation that is obtained from the type isomorphisms and categorical structures that underlie models of linear logic and quantum computing and that treats information as a linear resource that can neither be erased nor duplicated. In this paper, we show that our pure reversible model unveils deeper and more elegant symmetries of computation than have previously been reported. In particular, we expose two notions of duality of computation: an additive duality and a multiplicative duality that give rise to negative types and fractional types respectively. Although these types have previously appeared in the literature (see Sec. 6), they have typically appeared in the context of conventional languages with information effects, which limited their appeal and obscured their properties.

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

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

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. 4 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 $\mathit{left}\ 20$;
- We continue to line (2) with the same value $\mathit{left}\ 20$;
- 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 $\mathit{left}\ (\mathit{left}\ 20)$;
- 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 $\mathit{left}\ (\mathit{right}\ 20)$;
- Execution continues in reverse to line (2) with the value $\mathit{right}\ (\mathit{left}\ 20)$;
- At line (1) we find ourselves again facing an empty type so we reverse execution again; we go to line (2) with a value $\mathit{right}\ (\mathit{right}\ 20)$;
- We proceed to line (3) and (4) with the value $\mathit{right}\ 20$;
- 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.

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

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 α. The α 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 α 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. 6), 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. 3.3 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. 5).

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 Π our reversible programming language of type isomorphisms [6, 16] (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. 3)
- We independently extend Π 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. 3)
- 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. 4)
- We develop programming intuition and argue that negative and fractional types ought to be part of the vocabulary of every programmer. (Sec. 5)
- 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 [4] relating continuations to the fractional types of the Lambek-Grishin calculus. (Sec. 6)

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: Π

We review our reversible language Π: the presentation in this section differs from the one in our previous paper [16] in two 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 the connections with continuations explored in this paper.

2.1 Syntax and Types

The set of types includes the empty type 0, the unit type 1, sum types $b_1 + b_2$, and products types $b_1 \times b_2$. The set of values v includes () which is the only value of type 1, *left* v and *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:

$$\begin{array}{rcl} \textit{value types}, b & ::= & 0 \mid 1 \mid b+b \mid b \times b \\ \textit{values}, v & ::= & () \mid \textit{left } v \mid \textit{right } v \mid (v,v) \end{array}$$

The combinators of Π are witnesses to the following type isomorphisms:

$$\begin{array}{rcl} \textit{zeroe}: & 0+b \leftrightarrow b & :\textit{zeroi} \\ \textit{swap}^{+}: & b_1+b_2 \leftrightarrow b_2+b_1 & :\textit{swap}^{+} \\ \textit{assocl}^{+}: & b_1+(b_2+b_3) \leftrightarrow (b_1+b_2)+b_3 & :\textit{assocr}^{+} \\ \textit{unite}: & 1 \times b \leftrightarrow b & :\textit{uniti} \\ \textit{swap}^{\times}: & b_1 \times b_2 \leftrightarrow b_2 \times b_1 & :\textit{swap}^{\times} \\ \textit{assocl}^{\times}: & b_1 \times (b_2 \times b_3) \leftrightarrow (b_1 \times b_2) \times b_3 & :\textit{assocr}^{\times} \\ \textit{distrib}_0: & 0 \times b \leftrightarrow 0 & :\textit{factor}_0 \\ \textit{distrib}: & (b_1+b_2) \times b_3 \leftrightarrow (b_1 \times b_3)+(b_2 \times b_3) & :\textit{factor} \end{array}$$

Each line of the above table introduces one or two combinators that witness the 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. The isomorphisms are extended to form a congruence relation by adding the following constructors that witness equivalence and compatible closure:

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

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

To summarize, the syntax of Π is given as follows.

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

$$\begin{array}{rcl} \textit{value types}, b & ::= & 0 \mid 1 \mid b+b \mid b \times b \\ \textit{values}, v & ::= & () \mid \textit{left } v \mid \textit{right } v \mid (v,v) \\ \\ \textit{comb. types}, t & ::= & b \leftrightarrow b \\ \textit{iso} & ::= & \textit{zeroe} \mid \textit{zeroi} \\ & \mid & \textit{swap}^{+} \mid \textit{assocl}^{+} \mid \textit{assocr}^{+} \\ & \mid & \textit{unite} \mid \textit{uniti} \\ & \mid & \textit{swap}^{\times} \mid \textit{assocl}^{\times} \mid \textit{assocr}^{\times} \\ & \mid & \textit{distrib}_0 \mid \textit{factor}_0 \mid \textit{distrib} \mid \textit{factor} \\ \textit{comb.}, c & ::= & \textit{iso} \mid \textit{id} \mid \textit{sym } c \mid c \fatsemi c \mid c + c \mid c \times c \end{array}$$

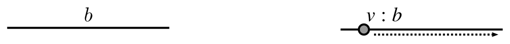
Adjoint. An important property of the language is that every combinator c has an adjoint $c^\dagger$ that reverses the action of c. This is evident by construction for the primitive isomorphisms. For the closure combinators, the adjoint is homomorphic except for the case of sequencing in which the order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

2.2 Graphical Language

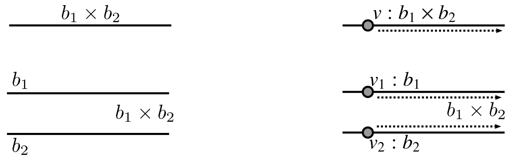
The syntactic notation above is often obscure and hard to read. Following the tradition established for monoidal categories [26], we present a graphical language that conveys the intuitive semantics of the language (which is formalized in the next section).

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.

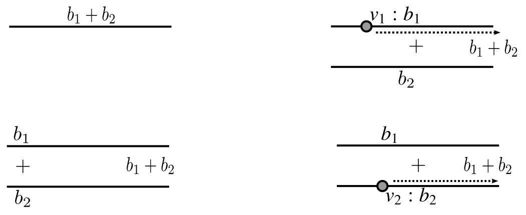
- The simplest sort of diagram is the $id : b \leftrightarrow b$ combinator which is simply represented as a wire labeled by its type b, as shown on the left. In more complex diagrams, if the type of a wire is obvious from the context, it may be omitted. When tracing a computation, one might imagine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



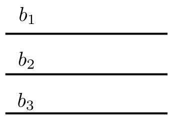
- The product type $b_1 \times b_2$ may be represented using either one wire labeled $b_1 \times b_2$ or two parallel wires labeled b_1 and b_2. In the case of products represented by a pair of wires, when tracing execution using particles, one should think of one particle on each wire or alternatively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



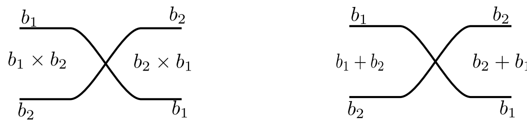
- Sum types may similarly be represented by one wire or using parallel wires with a $+$ operator between them. When tracing the execution of two additive wires, a value can reside on only one of the two wires.



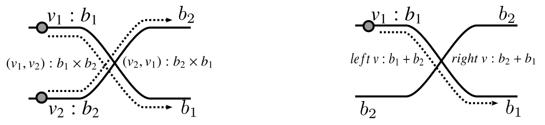
- Associativity is implicit in the graphical language. Three parallel wires represent $b_1 \times (b_2 \times b_3)$ or $(b_1 \times b_2) \times b_3$, based on the context.



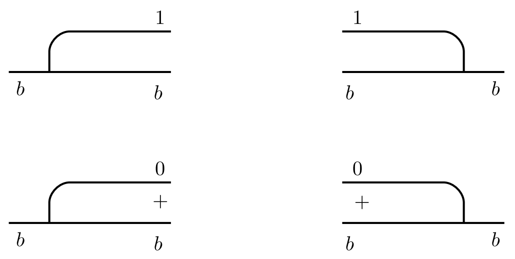
- Commutativity is represented by crisscrossing wires.



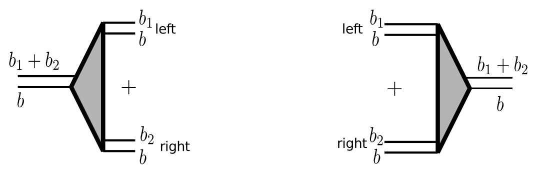
By visually tracking the flow of particles on the wires, one can verify that the expected types for commutativity are satisfied.



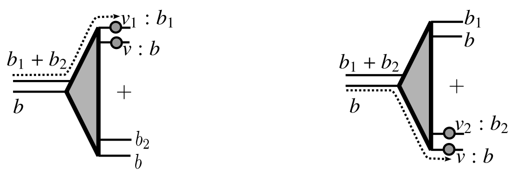
- The morphisms that witness that 0 and 1 are the additive and multiplicative units are represented as shown below. Note that since there is no value of type 0, there can be no particle on a wire of type 0. Also since the monoidal units can be freely introduced and eliminated, in many diagrams they are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



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



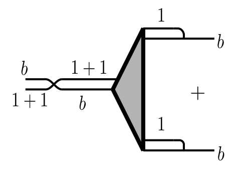
Distributivity and factoring are interesting because they represent interactions between sum and pair types. Distributivity should essentially be thought of as a multiplexer that redirects the flow of $v : b$ depending on what value inhabits the type $b_1 + b_2$, as shown below. Factoring is the corresponding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



Example. We use the type *bool* as a shorthand to denote the type $1 + 1$ and use *left* () to be *true* and *right* () to be *false*. The following combinator is represented by the given diagram:

$c : b \times bool \leftrightarrow b + b$

$c = swap^{\times} \fatsemi distrib \fatsemi (unite + unite)$



2.3 Semantics

The semantics of the primitive combinators is given by the following single-step reductions. Since there are no values of type 0, the rules omit the impossible cases:

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

The reductions for the primitive isomorphisms above are exactly the same as have been presented before [16]. 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 $\hookleftarrow$. 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 2.2 (Logical Reversibility). $\langle c, v, \Box\rangle \mapsto [c, v', \Box]$ *iff* $\langle c^\dagger, v', \Box\rangle \mapsto [c^\dagger, v, \Box]$

3. Negative and Fractional Types

This section introduces the syntax and graphical languages for negatives and fractionals. The general outline is as follows: [1]

- As established in our earlier work [6, 16], the underlying categorical semantics of our core reversible language Π is based on *two* distinct symmetric monoidal structures, one with $+$ as the monoidal tensor, and one with $\times$ as the monoidal tensor.
- We extend each underlying symmetric monoidal structure to a *compact closed* structure by adding a dual for each object and two special morphisms traditionally called η and ϵ.

The extended language is referred to as $\Pi^{\eta\epsilon}$. After presenting the extension at the syntactic level, we discuss the categorical semantics informally via the graphical language. The operational semantics is presented in Sec. 4.

As will be detailed in the remainder of this section, the compact closed structure provides several properties of interest: (i) morphisms or wires are allowed to run from right to left, (ii) the structure admits a trace that, depending on the situation, can be used to implement various notions of loops and recursion [14, 15, 17], (iii) the structure includes an isomorphism showing that the dual operator is an involution, and (iv) the structure is equipped with objects representing higher-order functions and a morphism *eval* that applies these functional objects. Interestingly each monoidal structure provides the same ingredients, resulting in two dualities, two traces, two involutions, and two notions of higher-order functions.

3.1 Syntax and Types

Describing the syntax and types of an extension to Π with additive and multiplicative duality is fairly straightforward. Basically, for the additive case, we extend the language with negative types denoted $-b$, negative values denoted $-v$, and two isomorphisms η^+ and ϵ^+. Similarly, for the multiplicative case, we extend the language with fractional types denoted $1/b$, fractional values denoted $1/v$, and two isomorphisms $\eta^\times$ and $\epsilon^\times$.

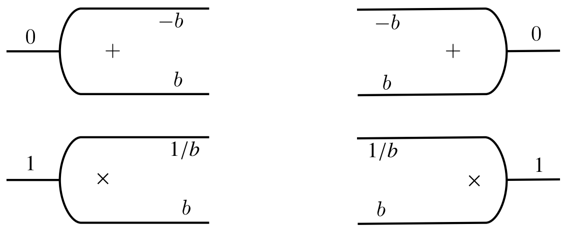
$$\begin{array}{rcl} \mathit{Value\ Types}, b &=& \ldots \mid -b \mid 1/b \\ \mathit{Values}, v &=& \ldots \mid -v \mid 1/v \\ \\ \mathit{Isomorphisms}, \mathit{iso} &=& \ldots \mid \eta^+ \mid \epsilon^+ \mid \eta^\times \mid \epsilon^\times \end{array}$$

For convenience, we sometimes use the notations $b_1 - b_2$ and b_1/b_2 to indicate the types $b_1 + (-b_2)$ and $b_1 \times (1/b_2)$ respectively. The types of the new constructs are:

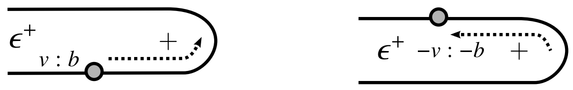
$$\begin{array}{lcl} \eta^+ & : 0 \leftrightarrow (-b) + b : & \epsilon^+ \\ \eta^\times & : 1 \leftrightarrow (1/b) \times b : & \epsilon^\times \end{array} \qquad \frac{\vdash v : b}{\vdash -v : -b} \quad \frac{\vdash v : b}{\vdash 1/v : 1/b}$$

For the graphical language, we visually represent η^+, ϵ^+, $\eta^\times$, and $\epsilon^\times$ as U-shaped connectors. On the left below are η^+ and $\eta^\times$ showing the maps from 0 to $-b + b$ and 1 to $1/b \times b$. On the right are ϵ^+ and $\epsilon^\times$ showing the maps from $-b + b$ to 0 and $1/b \times b$ to 1. Even though the diagrams below show the 0 and 1 wires for completeness, later diagrams will always drop them in contexts where they can be implicitly introduced and eliminated.

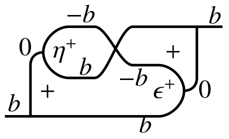
[1] We refer the reader to Selinger's excellent survey article of monoidal categories [26] for the precise definitions.



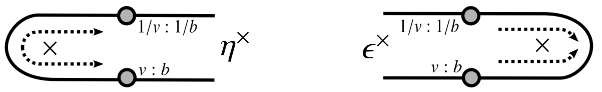
The usual interpretation of the type $b_1 + b_2$ is that we either have an appropriately tagged value of type b_1 or an appropriately tagged value of type b_2. This interpretation is maintained in the presence of η^+ and ϵ^+ in the following sense: a value of type *right* v : $-b + b$ flowing into an ϵ^+ on the lower wire switches to a value *left* $(-v)$: $-b + b$ that flows backwards on the upper wire. The inversion of direction is captured by the negative sign on the value.



As an example, the diagram representing the first isomorphism in the introduction is:



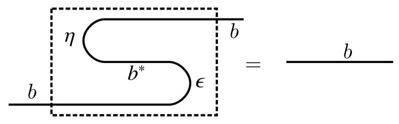
Similarly, in the usual interpretation of $b_1 \times b_2$ we have both a value of type b_1 and a value of type b_2. This interpretation is maintained in the presence of fractionals. Hence an $\eta^\times$: $1 \leftrightarrow (1/b) \times b$ is to be viewed as a fission point for a value of type b and its multiplicative inverse $1/b$. Operationally, this corresponds to the creation of two values α and $1/\alpha$ where α is a fresh logic variable. The operator $\epsilon^\times$ then becomes a unification site for these logic variables:



3.2 Categorical Constructions

All the constructions below are standard: they are collected from Selinger's survey paper on monoidal categories [26] and presented in the context of our language.

For a monoidal category to be compact closed the maps η and ϵ must satisfy a coherence condition that is usually visualized as follows:

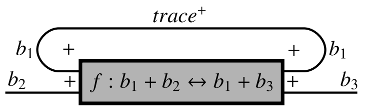


where b^* represents the dual of b. In the case of negatives, the condition amounts to checking that reversing direction twice is a no-op. In the case of fractionals, the condition amounts to checking that creating values α and $1/\alpha$ and immediately unifying them is also a no-op. Both checks are straightforward and are essentially the constructions in the introduction.

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:

$$trace^+ f = zeroi \fatsemi (id + \eta^+) \fatsemi (f + id) \fatsemi (id + \epsilon^+) \fatsemi zeroe$$



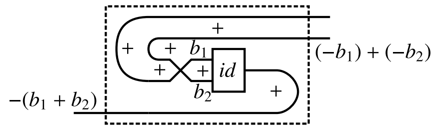
We have omitted some of the commutativity and associativity shuffling to communicate the main idea. We are given a value of type b_2 which we embed into $0 + b_2$ and then $(-b_1 + b_1) + b_2)$. This can be re-associated into $-b_1 + (b_1 + b_2)$. The component $b_1 + b_2$, which until now is just an 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; ϵ^+ is used to make it flow backwards to be fed to the η^+ 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:

$$(id + \eta^+) \fatsemi (swap^+ + id) \fatsemi (id + \epsilon^+)$$

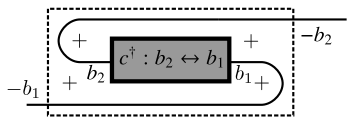
where we have omitted the 0 introduction and elimination. The idea is as follows: we start with a value of type b, embed it into $b + 0$ and use η to create something of type $b + (-(-b) + (-b))$. This is possible because η has the polymorphic type $-a + a$ which can be instantiated to $-b$. We then reshuffle the type to produce $-(-b) + (-b + b)$ and cancel the right hand side using ϵ^+. 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:



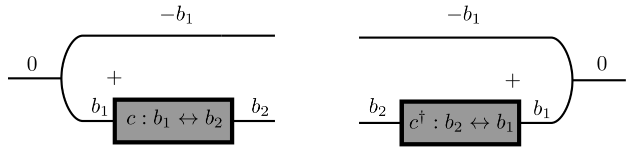
$$\frac{c : b_1 \leftrightarrow b_2}{neg\ c : (-b_1) \leftrightarrow (-b_2)}$$

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

3.3 Functions and Delimited Continuations

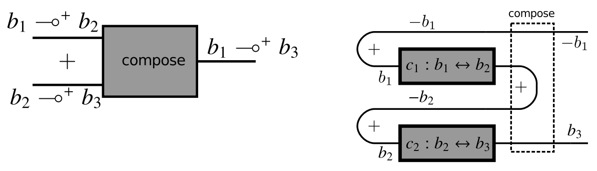
Although these constructions are also standard, they are less known and they are particularly important in our context: we devote a little more time to explain them. Our discussion is mostly based on Abramsky and Coecke's article on categorical quantum mechanics [1].

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:

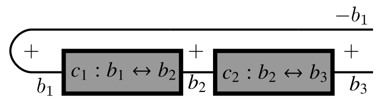


Intuitively the name consists of viewing c as a function and the coname consists of viewing c as a delimited continuation.

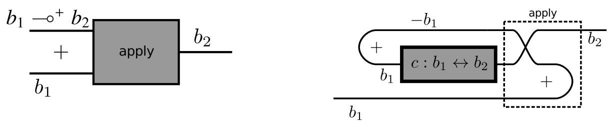
In addition to being able to represent morphisms, it is possible to express function composition. For the additive case, the composition is depicted below:



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 1×1;
- 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 α_1 and α_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 α_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.

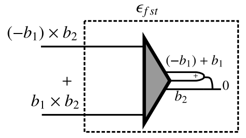
3.4 Additional Constructions

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.

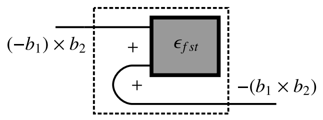
Lifting negation out of $\times$. The isomorphisms below state that the direction is *relative*. If b_1 and b_2 are flowing opposite to each other then it doesn't matter which direction is forwards and which is backwards. More interestingly as b_1 is moving backwards, it can "see the past" of b_2 which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

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

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

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

In this section we develop an operational semantics for Π extended with negative and fractional types, $\Pi^{\eta\epsilon}$. The operational semantics takes the abstract categorical diagrams which were previously known but gives them a computational interpretation based on reverse execution for negative types and unification for fractional types. Even though this computational interpretation appears straightforward in retrospect, it constitutes a breakthrough because it breaks the preconceived idea that there is only one notion of duality and hence that both negative and fractional types should be implemented using the same underlying mechanism. In some sense, the folklore literature on monoidal categories describing the evaluations for additive and multiplicative fragments are "particle-style" and "wave-style" can be seen to suggest either a unified or separate implementations. But, since de Morgan, we have been predisposed to think of only one notion of duality which persisted to even linear logic. (See Sec. 6 for further discussion of duality.)

For the purposes of this section, we start with the semantics for Π defined in Sec 2, and we will systematically rewrite it to achieve the desired $\Pi^{\eta\epsilon}$ semantics. This rewriting consists of two main steps:

1. We rewrite the semantics in Sec 2 using unification of the values instead of direct pattern matching. This gives us the necessary infrastructure for fractional types. We assume the reader is familiar with the idea of unification as realized using logic variables, substitutions, and reification as presented in any standard text on logic programming.
2. We write a reverse interpreter by reversing the direction of the $\mapsto$ reductions. This gives us the necessary infrastructure for negative types.

4.1 Unification

Previously the reductions of the primitive isomorphisms were specified in the following form:

$iso\ v_{input\ pattern} \mapsto v_{output\ pattern}$

Instead of relying on pattern-matching, we rewrite the rules to accept an incoming substitution to which the required pattern-matching rules are added as constraints. The general case is:

$iso\ s\ v' \mapsto (v_{output\ pattern}, s[v' \approx v_{input\ pattern}])$

Using the same idea, the entire semantics can be extended to thread the substitution as shown below:

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

Most of the rules look obviously correct but there is a subtle point. Consider the case for $c_1 + c_2$. Previously the shape of the value determined which of the combinators c_1 or c_2 to use. Now the incoming value could be a fresh logical variable, and indeed we have two rules with the same left hand side, and only depending on the success or failure of some further unification, can we decide which of them applies. The situation is common in logic programming in the sense that it is considered a non-deterministic process that keeps searching for the right substitution if any. Typical implementations of logic programming languages further provide top-level mechanisms to manipulate this non-deterministic search, for example by returning one answer and giving the user the option to ask for more answers. We abstract from this top-level semantics and view the rules above as being applied non-deterministically.

4.2 Reverse Execution

Given that our language is reversible (Prop. 2.2), 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.

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

4.3 Semantics of $\Pi^{\eta\epsilon}$

Combining the two constructions above, we get the semantics of the full $\Pi^{\eta\epsilon}$ language by adding the rules for the two variants of η and ϵ.

1. To add multiplicative duality, we add the following rules to the set of primitive isomorphisms:

   (a) $\eta^\times s\ () \mapsto ((1/v, v), s)$ where v is a fresh logic variable.

   As explained previously $\eta^\times$ creates two values $1/v$ and $1/v$ from a single logic variable.

   (b) $\epsilon^\times s\ v \mapsto (1, s[v \approx (1/v', v')])$ where v' is a fresh logic variable.

   Similarly, $\epsilon^\times$ unifies the values on incoming wires. The incoming value v represents the values of both wires and hence unifying them is accomplished in terms of an intermediate logic variable v'.

2. 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.

   (a) The rules for ϵ^+ 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 ϵ^+ 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.

   $$\begin{array}{l}
\left\langle \epsilon^+, v, C, s\right\rangle_\triangleright \mapsto \left\langle \epsilon^+, \mathit{left}\ (-v'), C, s[v \approx \mathit{right}\ v']\right\rangle_\triangleleft \\
\left\langle \epsilon^+, v, C, s\right\rangle_\triangleright \mapsto \left\langle \epsilon^+, \mathit{right}\ v', C, s[v \approx \mathit{left}\ (-v')]\right\rangle_\triangleleft
\end{array}$$

   Note that there is no evaluation rule for η^+ 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 η^+.

   (b) The rules for η^+ are added to the backward evaluator. A program executing backwards starts executing forwards after the execution of the η^+. Dual to the previous case, there is no rule for ϵ^+ in the backward evaluator since the output type of ϵ^+ is 0.

   $$\begin{array}{l}
\left\langle \eta^+, v, C, s\right\rangle_\triangleleft \mapsto \left\langle \eta^+, \mathit{left}\ (-v'), C, s[v \approx \mathit{right}\ v']\right\rangle_\triangleright \\
\left\langle \eta^+, v, C, s\right\rangle_\triangleleft \mapsto \left\langle \eta^+, \mathit{right}\ v', C, s[v \approx \mathit{left}\ (-v')]\right\rangle_\triangleright
\end{array}$$

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 [30].

5. Advanced Example: SAT Solver in $\Pi^{\eta\epsilon}$

We illustrate the expressive power of first-class constraints represented by fractional types. To understand the intuition, recall the definition of *trace* for the multiplicative fragment of $\Pi^{\eta\epsilon}$. Given $f : a \times c \leftrightarrow b \times c$, we have $\mathit{trace}^\times f : a \leftrightarrow b$:

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

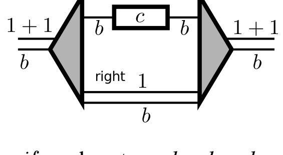
This circuit uses $\eta^\times$ to generate all possible c-values together with an associated $(1/c)$-constraint. It then applies f to the pair (a, c). The function f must produce an output (b, c') for each such input. If the input c and the output c' are the same they can be annihilated by $\epsilon^\times$;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

A large class of constraint satisfaction problems can be expressed using $\mathit{trace}^\times$. We illustrate the main ideas with the implementation of a SAT-solver. We proceed in small steps, reviewing some of the necessary constructions presented in our earlier paper [16].

5.1 Booleans and Conditionals

Given any combinator $c : b \leftrightarrow b$ we can construct a combinator called $\mathit{if}_c : \mathit{bool} \times b \leftrightarrow \mathit{bool} \times b$ in terms of c, where if_c behaves like a one-armed *if*-expression. If the supplied boolean is *true* then the combinator c is used to transform the value of type b. If the boolean is *false*, then the value of type b remains unchanged. We can write down the combinator for if_c in terms of c as $\mathit{distrib} \fatsemi ((\mathit{id} \times c) + \mathit{id}) \fatsemi \mathit{factor}$.

The diagram below shows the input value of type $(1 + 1) \times b$ processed by the distribute operator *distrib*, which converts it into a value of type $(1 \times b) + (1 \times b)$. In the *left* branch, which corresponds to the case when the boolean is *true* (i.e. the value was *left* ()), the combinator c is applied to the value of type b. The right branch which corresponds to the boolean being *false* passes along the value of type b unchanged.



The combinator $\mathit{if}_{\mathit{not}}$ has type $\mathit{bool} \times \mathit{bool} \leftrightarrow \mathit{bool} \times \mathit{bool}$ and negates its second argument if the first argument is *true*. This gate $\mathit{if}_{\mathit{not}}$ is often referred to as the *cnot* gate. An equivalent construction that is useful is $\mathit{else}_{\mathit{not}}$ where we negate the second argument only if the first is *false*.

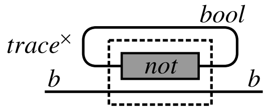
Similarly, we can iterate the construction of if_c to check several bits. The gate $\mathit{if}_{\mathit{cnot}}$, which we may also write as $\mathit{if}^2_{\mathit{not}}$, checks two booleans and negates the result wire only if they are both $true$. The gate if^2_{not} is well known as the Toffoli gate and is a universal reversible gate. We can generalize this construction to if^n_{not} which checks n bits and negates the result wire only if they are all $true$.

5.2 Cloning

Although cloning is generally not allowed in reversible languages, it is possible at the cost of having additional constant inputs. For example, consider the gate $else_{not}$. Generally, the gate maps $(false, a)$ to $(false, not\ a)$ and $(true, a)$ to $(true, a)$. Focusing on the cases in which the second input is $true$, we get that the gate maps $(false, true)$ to $(false, false)$ and $(true, true)$ to $(true, true)$, i.e., the gate clones the first input. A circuit of n parallel $else_{not}$ gates can hence clone n bits. They also consume n $true$ inputs in the process. Let us call this construction $clone^n_{bool}$.

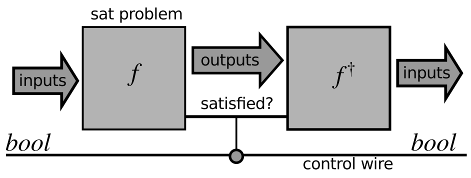
5.3 Construction of the Solver

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.

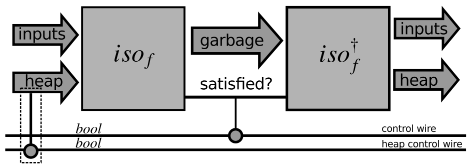
With a little work, we can modify this circuit to only annihilate values that fail to satisfy the constraints represented by a SAT-instance f. In more detail, an instance of SAT is a function/circuit f that given some boolean inputs returns $true$ or $false$ which we interpret as whether the inputs satisfy 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. 5.4 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:



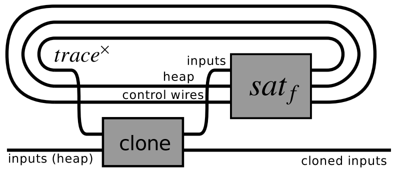
To summarize, the top half of the circuit is the identity function except that we have also managed to produce a boolean wire labeled satisfied? that tells us if the inputs satisfy the desired constraints. We can take this boolean value and use it to decide whether to negate the control wire or not.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

5.4 Final Details

Any boolean expression $f : bool^n \rightarrow bool$ 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 [16, 27]. 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, i.e., the dotted part in the diagram below.



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 solutions as determined by the semantics of the top-level interaction (see discussion in Sec. 4).

6. 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 λ-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 [10] 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 $A \rightarrow B$. 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 [28, 29]. 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* [8] which has recently been related to coroutines [9] 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 [11] 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 [4]. 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 λ-calculus, which allows arbitrary duplication and erasure of information, values of negative and fractional types are first-class values that can flow anywhere. The information-preserving computational infrastructure guarantees that, in a complete program, 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."

7. Conclusion and Future Work

We have extended the language Π 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 [13] 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 Π 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 [16]; 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 [5] 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:

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

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

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

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

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

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

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

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

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

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

[14] M. Hasegawa. Recursion from cyclic sharing: Traced monoidal categories and models of cyclic lambda calculi. In *TLCA*, pages 196–213. Springer-Verlag, 1997.

[15] M. Hasegawa. On traced monoidal closed categories. *MSCS*, 19:217–244, April 2009.

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

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

[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. 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.

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

[28] 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.

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

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