Fractional Types

Roshan P. James[1], Zachary Sparks[1], Jacques Carette[2], and Amr Sabry[1]

(1) Indiana University (2) McMaster University

Abstract. In previous work, we developed a *first-order*, information-preserving, and reversible programming language Π founded on type isomorphisms. Being restricted to first-order types limits the expressiveness of the language: it is not possible, for example, to abstract common program fragments into a higher-level combinator. In this paper, we introduce a higher-order extension of Π based on the novel concept of *fractional types* $1/b$. Intuitively, a value of a fractional type $1/v$ represents *negative* information. A function is modeled by a pair $(1/v_1, v_2)$ with $1/v_1$ representing the needed argument and v_2 representing the result. Fractional values are first-class: they can be freely propagated and transformed but must ultimately — in a complete program — be offset by the corresponding amount of positive information.

1 Introduction

We are witnessing a convergence of ideas from several distinct research communities (physics, mathematics, and computer science) towards replacing *equalities* by *isomorphisms*. The combined programme has sparked a significant amount of research that unveiled new and surprising connections between geometry, algebra, logic, and computation (see [2] for an overview of some of the connections).

In the physics community, Landauer [17, 18], Feynman [7], and others have interpreted the laws of physics as fundamentally related to computation. The great majority of these laws are formulated as equalities between different physical observables which is unsatisfying: *different* physical observables should not be related by an *equality*. It is more appropriate to relate them by an *isomorphism* that witnesses, explains, and models the process of transforming one observable to the other.

In the mathematics and logic community, Martin-Löf developed an extension of the simply typed λ-calculus originally intended to provide a rigorous framework for constructive mathematics [19]. This theory has been further extended with *identity types* representing the proposition that two terms are "equal." (See [23, 24] for a survey.) Briefly speaking, given two terms a and b of the same type A, one forms the type $\mathtt{Id}_A(a, b)$ representing the proposition that a and b are equal: in other words, a term of type $\mathtt{Id}_A(a, b)$ witnesses, explains, and models the process of transforming a to b and vice-versa.

In the computer science community, the theory and practice of type isomorphisms is well-established. Originally, such type isomorphisms were motivated by the pragmatic concern of searching large libraries of functions by providing one of the many possible isomorphic types for the desired function [20]. More recently, type isomorphisms have taken a more central role as *the* fundamental computational mechanism from which more conventional, i.e., irreversible computation, is derived. In our own previous work [13, 4, 14] we started with the notion of type isomorphism and developed from it a family of programming languages, Π with various superscripts, in which computation is an isomorphism preserving the information-theoretic entropy.

A major open problem remains, however: a higher-order extension of Π. This extension is of fundamental importance in all the originating research areas. In physics, it allows for quantum states to be viewed as processes and processes to be viewed as states, such as with the Choi-Jamiolkowski isomorphism [5, 15]. In mathematics and logic, it allows the equivalence between different proofs of type $\mathtt{Id}_A(a,b)$ to itself be expressed as an isomorphism (of a higher type) $\mathtt{Id}_{\mathtt{Id}_A(a,b)}(p,q)$. Finally, in computer science, higher-order types allow code to abstract over other code fragments as well as the manipulation of code as data and data as code.

Technically speaking, obtaining a higher-order extension requires the construction of a *closed category* from the underlying monoidal category for Π. Although the general idea of such a construction is well-understood, the details of adapting it to an actual programming language are subtle. Our main novel technical device to achieving the higher-order extension is a *fractional type* which represents *negative information* and which is so named because of its duality with conventional product types. The remainder of the paper reviews Π and then introduces the syntax and semantics of the extension with fractional types. We then study the properties of the extended language and establish its expressiveness via several constructions and examples that exploit its ability to model higher-order computations.

2 Background: Π

We review our language Π providing the necessary background and context for our higher-order extension.[1] The terms of Π are not classical values and functions; rather, the terms are isomorphism witnesses. In other words, the terms of Π 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 frequently 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 approach is different: we start from a type system, and then present a term language which naturally inhabits these types, along with an appropriate operational semantics.

[1] The presentation in this section focuses on the simplest version of Π. Other versions include the empty type, recursive types, and trace operators but these extensions are orthogonal to the higher-order extension emphasized in this paper.

Data. We view Π as having two levels: it has traditional values, given by:

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

and these are classified by ordinary types:

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

Types include the unit type 1, sum types $b_1 + b_2$, and product types $b_1 \times b_2$. Values include () 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.

Isomorphisms. The terms of Π witness type isomorphisms of the form $b \leftrightarrow b$. They consist of base isomorphisms, as defined below, and their composition.

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

Each line of the above table introduces a pair of dual constants[2] that witness the type isomorphism in the middle. 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. Second, 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.

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

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

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

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" via *application*:

$$\mathit{top\ level\ term}, l ::= c\ v$$

[2] where $\mathit{swap}^{\times}$ and swap^{+} are self-dual.

The language presented above, at the type level, models a commutative ringoid where the multiplicative structure forms a commutative monoid, but the additive structure is just a commutative semigroup. Note that the version of Π that includes the empty type with its usual laws exactly captures, at the type level, the notion of a *semiring* (occasionally called a *rig*) where we replace equality by isomorphism. Semantically, Π models a *bimonoidal category* whose simplest example is the category of finite sets and bijections. In that interpretation, each value type denotes a finite set of a size calculated by viewing the types as natural numbers and each combinator $c : b_1 \leftrightarrow b_2$ denotes a bijection between the sets denoted by b_1 and b_2. (We discuss the operational semantics in conjuction with our extension in the next section.)

3 The Language: $\Pi^{/}$

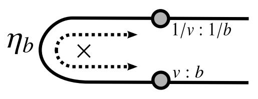
The language Π models isomorphisms of values rather well, and as we established before [13, 4, 14], it is logically reversible and each computation it expresses preserves the information-theoretic entropy. However, purposefully, it has no distinguished notion of *input* or *output*; more precisely, because of strict preservation of information, these two concepts coincide. This is the root of reversibility. But if we want to model functions of any flavor as values, we need to differentiate between these notions. Our idea, inspired by computational dualities [8, 6], polarity [9, 25], and the categorical notion of *compact closure* [21, 1], is to introduce a *formal dual* for our types. We thus consider a type in a negative position to possess negative information, as it is really a request for information. Since information is logarithmic, this means that a request for information should behave like a *fractional type.*

The extension of the sets of types and values from Π to $\Pi^{/}$ is simple:

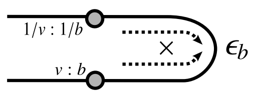
$$\begin{aligned} \textit{value types}, b &::= \ldots \mid 1/b \\ \textit{values}, v &::= \ldots \mid 1/v \end{aligned}$$

For a given type b, the values of type $1/b$ are of the form $1/v$ where $v : b$. Note that $1/v$ is purely formal, as is $1/b$.

Semantically, we are extending the symmetric monoidal category modeling Π to a *compact closed* one, i.e., a category in which morphisms are representable as objects. In such a setting, the new dual objects (i.e., the fractionals) must satisfy the following isomorphism witnessed by two new type indexed combinators η_b and ϵ_b:



$$\eta_b : 1 \leftrightarrow 1/b \times b : \epsilon_b$$



From a programming perspective, we think of a value $1/v$ as a *first-class constraint* that can only be satisfied if it is matched with an actual value v. In other words, $1/v$ is a *pattern* representing the absence of some information of a specific shape (i.e., *negative information*), that can only be reconciliated by an actual value v. The combinator $\eta_b : 1 \leftrightarrow 1/b \times b$ thus represents a fission point which creates — out of no information — an equal amount of negative and positive information. Symmetrically, the combinator $\epsilon_b : 1/b \times b \leftrightarrow 1$ matches up an equal amount of negative and positive information producing no residual information.

Historically, versions of compact closed categories were introduced by Abramsky and Coecke [1] and by Selinger [21] as the generalization of monoidal categories to model quantum computation. The first and most intuitive example of such categories is the category of finite sets and relations. In that case, the dual operation (i.e., the fractional type) collapses by defining $1/b$ to be b. In spite of this degenerate interpretation of fractionals, this is still an interesting category as it provides a model for a reversible higher-order language. Indeed, any relation is reversible via the standard *converse* operation on a relation. Furthermore, any relation between b_1 and b_2 can be represented as a subset of $b_1 \times b_2$, i.e., as a value.

In this section, we provide a simple semantics of $\Pi^/$ in the category of finite sets and relations, in the following way: We interpret each combinator $c : b_1 \leftrightarrow b_2$ as a *relation* between the sets denoted by b_1 and b_2 respectively. Sets b_1 and b_2 can be related only if the same information measure. As we illustrate, this semantics allows *arbitrary* relations (including the empty relation and relations that are not isomorphisms) to be represented as values in $\Pi^/$. In Sec. 5 we discuss more refined semantics that are more suitable for richer categories.

Definition 1 (Denotation of Value Types). *Each type denotes a finite set of values as follows:*

$$\begin{aligned}
[\![1]\!] &= \{()\} \\
[\![b_1 + b_2]\!] &= \{\mathit{left}\ v \mid v \leftarrow [\![b_1]\!]\} \cup \{\mathit{right}\ v \mid v \leftarrow [\![b_2]\!]\} \\
[\![b_1 \times b_2]\!] &= \{(v_1, v_2) \mid v_1 \leftarrow [\![b_1]\!], v_2 \leftarrow [\![b_2]\!]\} \\
[\![1/b]\!] &= \{1/v \mid v \leftarrow [\![b]\!]\}
\end{aligned}$$

We specify relations using a deductive system whose judgments are of the form $v_1 \;\; c \;\; v_2$ indicating that the pair (v_1, v_2) is in the relation denoted by c.

Definition 2 (Relational semantics). *Each combinator $c : b_1 \leftrightarrow b_2$ in $\Pi^/$ denotes a relation as specified below.* [3]

$$\frac{}{(\mathit{left}\ v)\;\; \mathit{swap}^+ \;\; (\mathit{right}\ v)} \qquad \frac{}{(\mathit{right}\ v)\;\; \mathit{swap}^+ \;\; (\mathit{left}\ v)}$$

$$\frac{}{(\mathit{left}\ v)\;\; \mathit{assocl}^+ \;\; (\mathit{left}\ (\mathit{left}\ v))} \qquad \frac{}{(\mathit{right}\ (\mathit{left}\ v))\;\; \mathit{assocl}^+ \;\; (\mathit{left}\ (\mathit{right}\ v))}$$

$$\frac{}{(\mathit{right}\ (\mathit{right}\ v))\;\; \mathit{assocl}^+ \;\; (\mathit{right}\ v)} \qquad \frac{}{(\mathit{left}\ (\mathit{left}\ v))\;\; \mathit{assocr}^+ \;\; (\mathit{left}\ v)}$$

[3] In the interest of brevity, we have treated the type annotations in the base isomorphisms as implicit; for example, the rule for $\mathit{assocr}^\times$ should really read:

$$\frac{v_1 \in b_1 \quad v_2 \in b_2 \quad v_3 \in b_3}{((v_1, v_2), v_3)\;\; \mathit{assocr}^\times_{b_1,b_2,b_3} \;\; (v_1, (v_2, v_3))}$$

$$\frac{}{(\mathit{left}\ (\mathit{right}\ v))\ \ \mathit{assocr}^{+}\ \ (\mathit{right}\ (\mathit{left}\ v))} \quad \frac{}{(\mathit{right}\ v)\ \ \mathit{assocr}^{+}\ \ (\mathit{right}\ (\mathit{right}\ v))}$$

$$\frac{}{((),v)\ \ \mathit{unite}\ \ v} \quad \frac{}{v\ \ \mathit{uniti}\ \ ((),v)} \quad \frac{}{(v_1,v_2)\ \ \mathit{swap}^{\times}\ \ (v_2,v_1)}$$

$$\frac{}{(v_1,(v_2,v_3))\ \ \mathit{assocl}^{\times}\ \ ((v_1,v_2),v_3)} \quad \frac{}{((v_1,v_2),v_3)\ \ \mathit{assocr}^{\times}\ \ (v_1,(v_2,v_3))}$$

$$\frac{}{(\mathit{left}\ v_1,v_3)\ \ \mathit{distrib}\ \ (\mathit{left}\ (v_1,v_3))} \quad \frac{}{(\mathit{right}\ v_2,v_3)\ \ \mathit{distrib}\ \ (\mathit{right}\ (v_2,v_3))}$$

$$\frac{}{(\mathit{left}\ (v_1,v_3))\ \ \mathit{factor}\ \ (\mathit{left}\ v_1,v_3)} \quad \frac{}{(\mathit{right}\ (v_2,v_3))\ \ \mathit{factor}\ \ (\mathit{right}\ v_2,v_3)}$$

$$\frac{}{v\ \ \mathit{id}\ \ v} \quad \frac{v'\ \ c\ \ v}{v\ \ (\mathit{sym}\ c)\ \ v'} \quad \frac{v_1\ \ c_1\ \ v_2 \qquad v_2\ \ c_2\ \ v_3}{v_1\ \ (c_1 \fatsemi c_2)\ \ v_3}$$

$$\frac{v\ \ c_1\ \ v'}{(\mathit{left}\ v)\ \ (c_1+c_2)\ \ (\mathit{left}\ v')} \quad \frac{v\ \ c_2\ \ v'}{(\mathit{right}\ v)\ \ (c_1+c_2)\ \ (\mathit{right}\ v')}$$

$$\frac{v_1\ \ c_1\ \ v_1' \qquad v_2\ \ c_2\ \ v_2'}{(v_1,v_2)\ \ (c_1\times c_2)\ \ (v_1',v_2')} \quad \frac{}{()\ \ \eta_b\ \ (1/v,v)} \quad \frac{}{(1/v,v)\ \ \epsilon_b\ \ ()}$$

The semantics above defines a relation rather than a function. It is still reversible in the sense that it "mode-checks". In other words, for any combinator c we can treat either its left or its right side as an input; this allows us to define *sym* as just relational converse, which amounts to an argument swap. We can then give an operational interpretation to the semantic relation by defining two interpreters: a forward evaluator and a backwards evaluator that exercise the relation in opposite directions, but which now can return a *set* of results. This operational view exploits the usual conversion of a relation from a subset of $A \times B$ to a function from A to the powerset of B whose composition is given by the Kleisli composition in the powerset monad. It is straightforward to check that if the forward evaluation of $c\ v_1$ produces v_2 as a possible answer then the backwards evaluation of $c\ v_2$ produces v_1 as a possible answer. One can check that all relations for Π are total functional relations whose converse are also total functional relations – aka isomorphisms. However η_b relates $()$ to multiple values (one for each value of type b), and is thus not functional; ϵ_b is exactly its relational converse, as expected.

4 Expressiveness and Examples

Having introduced the syntax of our extended language, its semantics in the category of sets and relations, and its operational semantics using forward and backwards interpreters, we now illustrate its expressiveness as a higher-order programming language. In the following presentation consisting of numerous programming examples, we use *bool* as an abbreviation of $1 + 1$ with *true* as an abbreviation for *left* $()$ and *false* as an abbreviation for *right* $()$. In addition, instead of presenting the code using the syntax of Π, we use circuit diagrams that are hopefully more intuitive. Each diagrams represents a combinator whose evaluation consists of propagating values along the wires. For the sake of readability, we omit obvious re-shuffling circuitry and elide trivial unit wires which carry no information.[4]

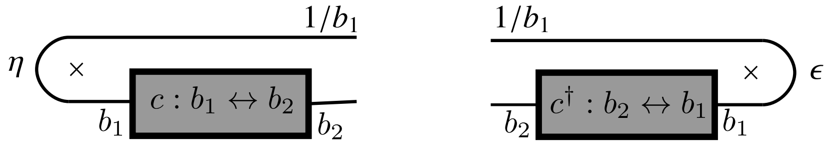
4.1 First-Class Relations

The most basic additional expressiveness of $\Pi^{/}$ over Π is the ability to express relations as values. Indeed a value of type $1/b_1 \times b_2$ is a pair of a *constraint* that can only be satisfied by some $v_1 : b_1$ and a value $v_2 : b_2$. In other words, it corresponds to a function or relation which when "given" a value $v_1 : b_1$ "releases" the value $v_2 : b_2$. To emphasize this view, we introduce the abbreviation:

$$b_1 \multimap b_2 ::= 1/b_1 \times b_2$$

which suggests a function-like behavior for the pair of a fractional value and a regular value. What is remarkable is that we can almost trivially turn any combinator $c : b_1 \leftrightarrow b_2$ into a (constant) value of type $b_1 \multimap b_2$ as shown below on the left:

$$\begin{aligned} &\mathit{name} \ : \ (b_1 \leftrightarrow b_2) \rightarrow (1 \leftrightarrow (b_1 \multimap b_2)) \\ &\mathit{name}\ c = \eta_{b_1} \fatsemi (\mathit{id} \times c) \end{aligned}$$


$1/b_1$
η
$\times$
$c : b_1 \leftrightarrow b_2$
b_1
b_2
$1/b_1$
$\times$
ϵ
$c^\dagger : b_2 \leftrightarrow b_1$
b_2
b_1

Dually, as illustrated above on the right, we also have a *coname* combinator which when given an answer $v_2 : b_2$, and a request for the corresponding input $1/b_1$ for $c : b_1 \leftrightarrow b_2$, eliminates this as a un-needed computation.

More generally any combinator manipulating values of type b can be turned into a combinator manipulating values of type $1/b$ and vice-versa. This is due to the fact that fractionals types satisfy a self-dual involution relating b and $1/(1/b)$:

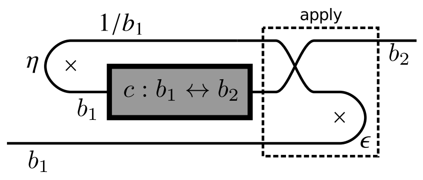
$$\begin{aligned} &\mathit{doubleDiv} \ : \ b \leftrightarrow 1/(1/b) \\ &\mathit{doubleDiv} = \mathit{uniti} \fatsemi (\eta_{1/b} \times \mathit{id}) \fatsemi \mathit{assocr}^{\times} \fatsemi (\mathit{id} \times \epsilon_b) \fatsemi \mathit{swap}^{\times} \fatsemi \mathit{unite} \end{aligned}$$

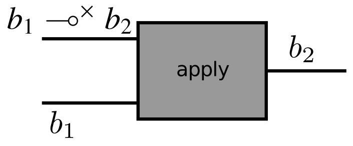
4.2 Higher-Order Relations

We are now a small step from implementing various higher-order combinators that manipulate functions or relations. In particular, we can *apply*, *compose*, *curry*, and *uncurry* values representing relations. We first show the realization of *apply*:

$$\begin{aligned} &\mathit{apply} \ : \ (b_1 \multimap b_2) \times b_1 \leftrightarrow b_2 \\ &\mathit{apply} = \mathit{swap}^{\times} \fatsemi \mathit{assocl}^{\times} \fatsemi (\mathit{swap}^{\times} \times \mathit{id}) \fatsemi (\epsilon_{b_1} \times \mathit{id}) \fatsemi \mathit{unite} \end{aligned}$$

[4] The full code is available upon request. We plan on releasing it as soon as possible.



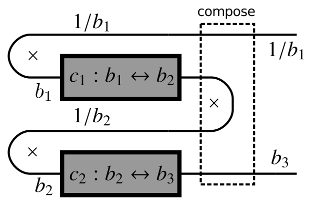


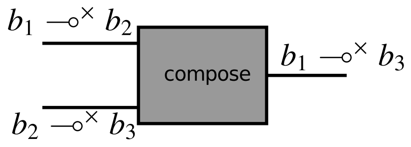
Intuitively, we simply match the incoming argument of type b_1 with the constraint encoded by the function. If they match, they cancel each other and the value of type b_2 is exposed with no constraints. Otherwise, the result is undefined. A flipped variant is also useful:

$$\begin{aligned} &apply' \ : \ b_1 \times (b_1 \multimap b_2) \leftrightarrow b_2 \\ &apply' = assocl^{\times} \fatsemi (swap^{\times} \times id) \fatsemi (\epsilon_{b_1} \times id) \fatsemi unite \end{aligned}$$

Function or relation composition is now straightforward:

$$\begin{aligned} &compose \ : \ (b_1 \multimap b_2) \times (b_2 \multimap b_3) \rightarrow (b_1 \multimap b_3) \\ &compose = assocr^{\times} \fatsemi (id \times apply') \end{aligned}$$




$b_1 \multimap^{\times} b_2$
compose
$b_1 \multimap^{\times} b_3$
$b_2 \multimap^{\times} b_3$

We can also derive currying (and dually, uncurrying) combinators. Observe that the type of *curry* needs to be $(b_1 \times b_2 \multimap b_3) \leftrightarrow (b_1 \multimap (b_2 \multimap b_3))$, which can be written in mathematical notation as $1/(b_1 \times b_2) \times b_3 = (1/b_1) \times ((1/b_1) \times b_3)$. This means *curry* can be written using *recip*, the implementation of the mathematical identity $1/(b_1 \times b_2) = 1/b_1 \times 1/b_2$:

$$\begin{aligned} &recip \ : \ 1/(b_1 \times b_2) \leftrightarrow 1/b_1 \times 1/b_2 \\ &recip = (uniti) \fatsemi (uniti) \fatsemi (assocl^{\times}) \fatsemi ((\eta_{b_1} \times \eta_{b_2}) \times id) \fatsemi \\ &\qquad\quad (reorder \times id) \fatsemi (assocr^{\times}) \fatsemi (id \times swap^{\times}) \ (id \times \epsilon) \fatsemi swap^{\times} \fatsemi (unite) \end{aligned}$$

where *reorder* is the obvious combinator of type $b_1 \times (b_2 \times b_3) \times b_4 \leftrightarrow b_1 \times (b_3 \times b_2) \times b_4$. With *recip* out of the way, we can easily write *curry*:

$$\begin{aligned} &curry \ : \ b_1 \times b_2 \multimap b_3 \leftrightarrow b_1 \multimap (b_2 \multimap b_3) \\ &curry = (id \times recip) \fatsemi (id \times swap^{\times}) \fatsemi assocl^{\times} \end{aligned}$$

That *recip* is the heart of currying seems quite remarkable.

4.3 Feedback, Iteration, and Trace Operators

Mathematically speaking, recursion and iteration can be expressed using categorical trace operators [16, 10]. In a language like *Π* there are two natural families of trace operators that can be defined, an additive family (explored in detail in our previous work [4]) and a multiplicative family, which is expressible using fractionals.

The idea of the multiplicative trace operator is as follows. We are given a computation $c : b_1 \times b_2 \leftrightarrow b_1 \times b_3$, and we build a "looping" version which feeds the output value of type b_1 back as an input. Effectively, this construction cancels the common type b_1, to produce a new combinator $\mathit{trace}^{\times}\ c : b_2 \leftrightarrow b_3$.

With fractionals, $\mathit{trace}^{\times}$ becomes directly expressible:

$$\begin{aligned} \mathit{trace}^{\times}_{b} &: ((b \times b_1) \leftrightarrow (b \times b_2)) \rightarrow (b_1 \leftrightarrow b_2) \\ \mathit{trace}^{\times}_{b}\ c &= \mathit{uniti} \fatsemi (\eta_b \times \mathit{id}) \fatsemi \mathit{assocr}^{\times} \fatsemi \\ &\quad (\mathit{id} \times c) \fatsemi \mathit{assocl}^{\times} \fatsemi (\epsilon_b \times \mathit{id}) \fatsemi \mathit{unite} \end{aligned}$$

As an example, we can use the operational semantics (outlined in the previous section) to calculate the result of applying $\mathit{trace}^{\times}_{\mathit{bool}}(\mathit{swap}^{+} \times \mathit{id})$ to false.

$$\begin{array}{ll} \{((), \mathit{false})\} & (\mathit{uniti}) \\ \{((1/\mathit{false}, \mathit{false}), \mathit{false}), ((1/\mathit{true}, \mathit{true}), \mathit{false})\} & (\eta_{\mathit{bool}} \times \mathit{id}) \\ \{(1/\mathit{false}, (\mathit{false}, \mathit{false})), (1/\mathit{true}, (\mathit{true}, \mathit{false}))\} & (\mathit{assocr}^{\times}) \\ \{(1/\mathit{false}, (\mathit{true}, \mathit{false})), (1/\mathit{true}, (\mathit{false}, \mathit{false}))\} & (\mathit{id} \times \mathit{not}) \\ \{((1/\mathit{false}, \mathit{true}), \mathit{false}), ((1/\mathit{true}, \mathit{false}), \mathit{false})\} & (\mathit{assocl}^{\times}) \\ \emptyset & (\epsilon_{\mathit{bool}} \times \mathit{id}) \end{array}$$

The computation with true gives the same result. This confirms that although our semantics is formally reversible, it does not result in isomorphisms.

More abstractly, the evaluation of $\mathit{trace}^{\times}\ c : b_2 \leftrightarrow b_3$ must "guess" a value of type b_1 to be provided to to the inner combinator $c : b_1 \times b_2 \leftrightarrow b_1 \times b_3$. This value $v_1 : b_1$ cannot be arbitrary: it must be such that it is the value produced as the first component of the result. In general, there may be several such fixed-point values, or none. Indeed with a little ingenuity (see the next section), one can express any desired relation by devising a circuit which keeps the desired pairs as valid fixed-points.

4.4 (Finite) Relational Programming

Relational programming leverages set-theoretic relations and their composition to express computational tasks in a declarative way. It turns out that with the addition of the multiplicative trace, and the move to relations motivated in the previous section, we can express relational programming.

Consider the relation R on booleans given by:

$\{(\mathit{false}, \mathit{false}), (\mathit{false}, \mathit{true}), (\mathit{true}, \mathit{false})\}$.

We can define a combinator c_R whose denotation is R, by defining a combinator $\mathit{cInner} : (a \times \mathit{bool}) \leftrightarrow (a \times \mathit{bool})$ for some type a such that $c_R = \mathit{trace}^{\times}\ \mathit{cInner}$. The basic requirement of cInner is that for each desired pair (v_1, v_2) in R, it maps (a_0, v_1) to (a_0, v_2) for some value a_0 and that for each pair (v_1, v_2) that is *not* in R, it maps (a_1, v_1) to (a_2, v_2) for *different* values a_1 and a_2.

For small examples, finding such combinators by trial and error is a relatively straightforward but tedious task. It is, however, possible to automate this task by expressing what is essentially a reversible SAT solver as follows. In the usual setting, an instance of SAT is a function f which, when given some boolean inputs, returns *true* or *false*. The function returns *true* when the inputs satisfy the constraints imposed by the structure of f and a solution to the SAT problem is the set of all inputs on which f produces *true*. The basic idea of our construction is to use $\mathit{trace}^{\times}$ to annihilate values that fail to satisfy the constraints represented by the SAT instance f. (The accompanying code details the construction of such a solver.)

5 Conclusion

We have introduced the idea of *fractional types* in the context of a reversible language founded on type isomorphisms and preservation of information. Values of fractional types represent *negative information*, a concept which is difficult to introduce in a conventional language that allows arbitrary creation and deletion of information but which is much simpler to deal with when the surrounding language infrastructure guarantees preservation of information. Fractional types and values can be used to express a simple and elegant notion of higher-order functions: a function $b_1 \multimap b_2$ is a first-class value consisting of negative information drawn from b_1 and positive information drawn from b_2.

The interpretation of our language $\Pi^{/}$ in the category of sets and relations is adequate in the sense that it produces a language in which every program is a reversible relation and in which relations are first-class values. This relational model is however, unsatisfactory, for several reasons:

- the type $1/b$ is interpreted in the same way as b, which gives no insight into the "true meaning" of fractionals;
- the interpretation is inconsistent with the view of types as algebraic structures; for example, Π with the empty type, is the categorification of a semiring but although fractional types in $\Pi^{/}$ syntactically "look like" rational numbers, there is no such formal connection to the rational numbers;
- finally, we have lost some delicate structure moving from Π to $\Pi^{/}$ as we can express arbitrary relations between types and not just *isomorphisms*.

For these reasons, it is interesting to consider other possible semantic interpretations of fractionals. A natural alternative is to use the "canonical" compact closed category, that is finite dimensional vector spaces and linear maps [22, 11] over fields of characteristic 0 (or even the category of finite dimensional Hilbert spaces). Let us fix an arbitrary field k of characteristic 0. Then each type b in $\Pi^{/}$ is interpreted as a finite dimensional vector space $\mathbf{V}_b$ over the field k. In particular:

- every vector space contains a zero vector which means that the type 0 (if included in $\Pi^{/}$) would not be the "empty" type. Furthermore all combinators would be strict as they would have to map the zero vector to itself.

– the type 1 is interpreted as a 1-dimensional vector space and hence is isomorphic to the underlying field.
– the fractional type $1/b$ is interpreted as the dual vector space to the vector space $\mathbf{V}_b$ representing b consisting of all the *linear functionals* on $\mathbf{V}_b$.
– one can then validate certain desirable properties: $1/(1/b)$ is isomorphic to b; and ϵ_b corresponds to a bilinear form which maps a dual vector and a vector to a field element.

In such categories, the fractional type is given a non-trivial interpretation. Indeed, while the space of column vectors is isomorphic to the space of row vectors, they are nevertheless quite different, being $(1,0)$ and $(0,1)$ tensors (respectively). In other words, the category provides a more refined model in which isomorphic negative-information values and positive-information are not identified.

Although this semantics appears to have "better" properties than the relational one, we argue that it is not yet *the* "perfect" semantics. By including the zero vector, the language has morphisms that do not correspond to isomorphisms (in particular it allows partial morphisms by treating the 0 element of the zero-dimensional vector space as a canonical "undefined" value). It is also difficult to reconcile the interpretation with the view that the types correspond to the (positive) rational numbers, something we are actively seeking. What would really be a "perfect" interpretation is one in which we can only express Π-isomorphisms as first class values and in which the types are interpreted in a way that is consistent with the rational numbers.

Fortunately, there is promising significant work on the groupoid interpretation of type theory [12] and on the categorification of the rational numbers [3] that may well give us the model we desire. The fundamental idea in both cases is that groupoids (viewed as sets with explicit isomorphisms as morphisms) naturally have a *fractional cardinality*. Types would be interpreted as groupoids, and terms would be (invertible) groupoid actions. The remaining challenge is to identify those groupoid actions which are proper generalizations of isomorphisms *and* can be represented as groupoids, so as to obtain a proper interpretation for a higher-order language. As the category of groupoids is cartesian closed, this appears eminently feasible.

Another promising approach is the use of dependent types for ϵ_b and η_b; more precisely, ϵ_b would have type $\Sigma\ (v : b)\ (1/v, v) \leftrightarrow 1$ where $(1/v, v)$ here denotes a singleton type. This extra precision appears to restrict combinators of $\Pi^{/}$ back to denoting only isomorphisms.

References

1. Abramsky, S., Coecke, B.: A categorical semantics of quantum protocols. In: LICS (2004)
2. Baez, J., Stay, M.: Physics, topology, logic and computation: a rosetta stone. New Structures for Physics pp. 95–172 (2011)
3. Baez, J.C., Dolan, J.: Categorification. In Higher Category Theory, Contemp. Math. 230, 1998, pp. 1-36. (1998)

4. Bowman, W.J., James, R.P., Sabry, A.: Dagger Traced Symmetric Monoidal Categories and Reversible Programming. In: RC (2011)
5. Choi, M.D.: Completely positive linear maps on complex matrices. Linear algebra and its applications (1975)
6. Curien, P.L., Herbelin, H.: The duality of computation. In: ICFP (2000)
7. Feynman, R.: Simulating physics with computers. International Journal of Theoretical Physics 21, 467–488 (1982)
8. Filinski, A.: Declarative continuations: an investigation of duality in programming language semantics. In: Category Theory and Computer Science (1989)
9. Girard, J.Y.: Linear logic. Theor. Comput. Sci. 50, 1–102 (1987)
10. Hasegawa, M.: Recursion from cyclic sharing: Traced monoidal categories and models of cyclic lambda calculi. In: TLCA. pp. 196–213 (1997)
11. Hasegawa, M., Hofmann, M., Plotkin, G.: Finite dimensional vector spaces are complete for traced symmetric monoidal categories. Pillars of computer science (2008)
12. Hofmann, M., Streicher, T.: The groupoid interpretation of type theory. In: Venice Festschrift. pp. 83–111 (1996)
13. James, R.P., Sabry, A.: Information effects. In: POPL. pp. 73–84. ACM (2012)
14. James, R.P., Sabry, A.: Isomorphic interpreters from logically reversible abstract machines. In: RC (2012)
15. Jamiołkowski, A.: Linear transformations which preserve trace and positive semidefiniteness of operators. Reports on Mathematical Physics (1972)
16. Joyal, A., Street, R., Verity, D.: Traced monoidal categories. In: Mathematical Proceedings of the Cambridge Philosophical Society. Cambridge Univ Press (1996)
17. Landauer, R.: Irreversibility and heat generation in the computing process. IBM J. Res. Dev. 5, 183–191 (July 1961)
18. Landauer, R.: The physical nature of information. Physics Letters A (1996)
19. Martin-Löf, P.: An Intuitionistic Theory of Types: Predicative Part. The Journal of Symbolic Logic 49(1), 311+ (Mar 1984)
20. Rittri, M.: Using types as search keys in function libraries. In: FPCA (1989)
21. Selinger, P.: Dagger compact closed categories and completely positive maps. ENTCS 170, 139–163 (Mar 2007)
22. Selinger, P.: Finite dimensional hilbert spaces are complete for dagger compact closed categories (extended abstract). Electron. Notes Theor. Comput. Sci. 270(1) (2011)
23. Streicher, T.: Investigations into intensional type theory (1993), habilitationsschrift, Universität München
24. Warren, M.: Homotopy theoretic aspects of constructive type theory. Ph.D. thesis, Carnegie-Mellon University (2008)
25. Zeilberger, N.: Polarity and the logic of delimited continuations. LICS (2010)