Fractional Types

Jacques Carette[1], **Chao-Hong Chen**[2], **Vikraman Choudhury**[3], **and Amr Sabry**[4]

1 **Computing and Software Department, McMaster University, Hamilton, Ontario, Canada**
`carette@mcmaster.ca`

2 **Computer Science Department, Indiana University, Bloomington, Indiana, USA**
`chen464@indiana.edu`

3 **Computer Science Department, Indiana University, Bloomington, Indiana, USA**
`vikraman@indiana.edu`

4 **Computer Science Department, Indiana University, Bloomington, Indiana, USA**
`sabry@indiana.edu`

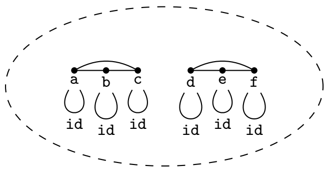
Abstract

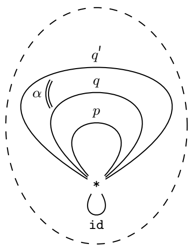
We exhibit types whose natural cardinality is fractional. More precisely, we show that the groupoid cardinality (as defined by Baez-Dolan) of the denotation of the type of a singleton reversible program p with exactly k distinct proofs of reversibility has cardinality $1/k$. We further show that this type is naturally a multiplicative inverse to the type of all iterates p^i of that reversible program. We situate this work as an extension of a larger reversible programming language (Π), and show that this extension is also reversible.

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

In modern treatments of type theory, types have the structure of *weak ω-groupoids*. As a first approximation, we can think of such structures as sets with points (objects) and paths (equivalences) between the points and higher paths between these paths and so on. Here are two simple but non-trivial examples:





Baez and Dolan [4] assign to each groupoid a *cardinality* that counts the objects up to equivalences. The groupoid on the left has six points `a`, `b`, `c`, `d`, `e`, and `f` with two groups of three points each clustered in an equivalence class and hence the groupoid has cardinality 2. The (2-)groupoid on the right has one point $*$ with four equivalences `id`, p, q, and q' on it. The equivalences q and q' are however identified by α leaving only three distinct isomorphism classes and hence making the cardinality $\frac{1}{3}$.

Both groupoids involve some notion of (semantic) “equivalence”, which we would like to capture as first-class entities in a programming language. In other words, we would like to have some syntactic notion of a type, a *fractional type*, whose denotation would be a groupoid with fractional cardinality. Our aim then is to create a language which possesses these fractional types, explore this notion of types, their equivalence, and their associated operational semantics.

The remainder of the paper is organized as follows. We start by reviewing necessary background material, consisting of the language Π for programming with isomorphisms or equivalences in a reversible information-preserving way. Sec. 4 explains the main novel semantic ideas of using Π programs to generate non-trivial groupoids with fractional cardinality, as well as translating the semantic ideas into an extension of Π with new type constructors denoting non-trivial groupoids and new programs that manipulate such types. The last section puts our work in perspective and concludes.

2 Programming with Equivalences

The main syntactic vehicle for the technical developments is the language Π whose only computations are isomorphisms between finite types and equivalences between these isomorphisms [10, 16]. We present the syntax and operational semantics of the parts of the language relevant to our work.

2.1 Syntax of Π

The Π family of languages is based on type isomorphisms. In the variant we consider, the set of types τ includes the empty type $\mathbb{0}$, the unit type $\mathbb{1}$, and sum $\oplus$ and product $\otimes$ types. The values classified by these types are the conventional ones: $()$ of type $\mathbb{1}$, $\mathsf{inl}(v)$ and $\mathsf{inr}(v)$ for injections into sum types, and (v_1, v_2) for product types. The language has two other syntactic categories of programs to be described in detail.

▶ **Definition 1** (Π). The syntax of Π is given by the following categories:

$$\begin{array}{llcl}
\text{(Types)} & \tau & ::= & \mathbb{0} \mid \mathbb{1} \mid \tau_1 \oplus \tau_2 \mid \tau_1 \otimes \tau_2 \\
\text{(Values)} & v & ::= & () \mid \mathsf{inl}(v) \mid \mathsf{inr}(v) \mid (v_1, v_2) \\
\text{(1-combinators)} & c, p & : & \tau_1 \leftrightarrow \tau_2 \; [\textit{see Fig. 1}] \\
\text{(2-combinators)} & \alpha & : & c_1 \Leftrightarrow c_2 \text{ where } c_1, c_2 : \tau_1 \leftrightarrow \tau_2 \; [\textit{see Fig. 2}]
\end{array}$$

Both classes of programs, 1-combinators c, and 2-combinators α, denote *equivalences* in the Homotopy Type Theory (HoTT) sense [29]. The elements c or p of 1-combinators denote type isomorphisms. The elements α of 2-combinators denote the set of sound and complete equivalences between these type isomorphisms. Using the 1-combinators, it is possible to write any reversible boolean function and hence encode arbitrary boolean functions by a technique that goes back to Toffoli [30]. The 2-combinators provide a layer of programs that computes semantics-preserving transformations of 1-combinators. As a small example, let us abbreviate $\mathbb{1} \oplus \mathbb{1}$ as the type $\mathbb{2}$ of booleans and examine two possible implementations of boolean negation. The first directly uses the primitive combinator $\mathsf{swap}_+ : \tau_1 \oplus \tau_2 \leftrightarrow \tau_2 \oplus \tau_1$ to exchange the two values of type $\mathbb{2}$; the second uses three consecutive swap_+s to achieve the same effect:

$$\begin{array}{lcl}
\mathsf{not}_1 & = & \mathsf{swap}_+ \\
\mathsf{not}_2 & = & (\mathsf{swap}_+ \odot \mathsf{swap}_+) \odot \mathsf{swap}_+
\end{array}$$

$$\begin{array}{rrcll}
\mathsf{unite_{+}l} : & \mathbb{0} \oplus \tau & \leftrightarrow & \tau & : \mathsf{uniti_{+}l} \\
\mathsf{unite_{+}r} : & \tau \oplus \mathbb{0} & \leftrightarrow & \tau & : \mathsf{uniti_{+}r} \\
\mathsf{swap_{+}} : & \tau_1 \oplus \tau_2 & \leftrightarrow & \tau_2 \oplus \tau_1 & : \mathsf{swap_{+}} \\
\mathsf{assocl_{+}} : & \tau_1 \oplus (\tau_2 \oplus \tau_3) & \leftrightarrow & (\tau_1 \oplus \tau_2) \oplus \tau_3 & : \mathsf{assocr_{+}} \\
\\
\mathsf{unite_{\star}l} : & \mathbb{1} \otimes \tau & \leftrightarrow & \tau & : \mathsf{uniti_{\star}l} \\
\mathsf{unite_{\star}r} : & \tau \otimes \mathbb{1} & \leftrightarrow & \tau & : \mathsf{uniti_{\star}r} \\
\mathsf{swap_{\star}} : & \tau_1 \otimes \tau_2 & \leftrightarrow & \tau_2 \otimes \tau_1 & : \mathsf{swap_{\star}} \\
\mathsf{assocl_{\star}} : & \tau_1 \otimes (\tau_2 \otimes \tau_3) & \leftrightarrow & (\tau_1 \otimes \tau_2) \otimes \tau_3 & : \mathsf{assocr_{\star}} \\
\\
\mathsf{absorbr} : & \mathbb{0} \otimes \tau & \leftrightarrow & \mathbb{0} & : \mathsf{factorzl} \\
\mathsf{absorbl} : & \tau \otimes \mathbb{0} & \leftrightarrow & \mathbb{0} & : \mathsf{factorzr} \\
\mathsf{dist} : & (\tau_1 \oplus \tau_2) \otimes \tau_3 & \leftrightarrow & (\tau_1 \otimes \tau_3) \oplus (\tau_2 \otimes \tau_3) & : \mathsf{factor} \\
\mathsf{distl} : & \tau_1 \otimes (\tau_2 \oplus \tau_3) & \leftrightarrow & (\tau_1 \otimes \tau_2) \oplus (\tau_1 \otimes \tau_3) & : \mathsf{factorl}
\end{array}$$

$$\frac{}{\mathsf{id} : \tau \leftrightarrow \tau} \qquad \frac{c_1 : \tau_1 \leftrightarrow \tau_2 \quad c_2 : \tau_2 \leftrightarrow \tau_3}{c_1 \odot c_2 : \tau_1 \leftrightarrow \tau_3}$$

$$\frac{c_1 : \tau_1 \leftrightarrow \tau_2 \quad c_2 : \tau_3 \leftrightarrow \tau_4}{c_1 \oplus c_2 : \tau_1 \oplus \tau_3 \leftrightarrow \tau_2 \oplus \tau_4} \qquad \frac{c_1 : \tau_1 \leftrightarrow \tau_2 \quad c_2 : \tau_3 \leftrightarrow \tau_4}{c_1 \otimes c_2 : \tau_1 \otimes \tau_3 \leftrightarrow \tau_2 \otimes \tau_4}$$

Each 1-combinator c has an inverse $!\ c$, e.g, $!\ \mathsf{unite_{+}l} = \mathsf{uniti_{+}l}$, $!(c_1 \odot c_2) = !c_2 \odot\ !c_1$, etc.

Figure 1 Π 1-combinators [16]

We can write a 2-combinator whose *type* is $\mathsf{not}_2 \Leftrightarrow \mathsf{not}_1$:

$$(\mathsf{linv_{\odot}l} \boxdot \mathsf{id}) \bullet \mathsf{idl_{\odot}l}$$

which not only shows the equivalence of the two implementations of negation but also shows *how* to transform one to the other. This rewriting focuses on the first two occurrences of $\mathsf{swap_{+}}$ and uses $\mathsf{linv_{\odot}l}$ to reduce them to id since they are inverses. It then uses $\mathsf{idl_{\odot}l}$ to simplify the composition of id with $\mathsf{swap_{+}}$ to just $\mathsf{swap_{+}}$.

Fig. 1 lists all the 1-combinators which consist of base combinators (top) and compositions (bottom). Each line of the base combinators introduces a pair of dual constants[1] that witness the type isomorphism in the middle. This set of isomorphisms is known to be sound and complete [13, 12]. As the full set of 2-combinators has 113 entries, Fig. 2 lists a few of the 2-combinators that we use in this paper. Each 2-combinator relates two 1-combinators of the same type and witnesses their equivalence. Both 1-combinators and 2-combinators are invertible and the 2-combinators behave as expected with respect to inverses of 1-combinators.

▶ Proposition 1. For any $c : \tau_1 \leftrightarrow \tau_2$, we have $c \Leftrightarrow\ !\ (!\ c)$.

▶ Proposition 2. For any $c_1, c_2 : \tau_1 \leftrightarrow \tau_2$, we have $c_1 \Leftrightarrow c_2$ implies $!\ c_1 \Leftrightarrow\ !\ c_2$.

[1] where $\mathsf{swap_{+}}$ and $\mathsf{swap_{\star}}$ are self-dual.

$$\frac{c : \tau_1 \leftrightarrow \tau_2}{\mathsf{id} : c \Leftrightarrow c} \qquad \frac{c_1, c_2, c_3 : \tau_1 \leftrightarrow \tau_2 \quad \alpha_1 : c_1 \Leftrightarrow c_2 \quad \alpha_2 : c_2 \Leftrightarrow c_3}{\alpha_1 \bullet \alpha_2 : c_1 \Leftrightarrow c_3}$$

$$\frac{c_1 : \tau_1 \leftrightarrow \tau_2 \quad c_2 : \tau_2 \leftrightarrow \tau_3 \quad c_3 : \tau_3 \leftrightarrow \tau_4}{\mathsf{assoc}_{\odot}\mathsf{l} : c_1 \odot (c_2 \odot c_3) \Leftrightarrow (c_1 \odot c_2) \odot c_3 : \mathsf{assoc}_{\odot}\mathsf{r}}$$

$$\frac{c : \tau_1 \leftrightarrow \tau_2}{\mathsf{idl}_{\odot}\mathsf{l} : \mathsf{id} \odot c \Leftrightarrow c : \mathsf{idl}_{\odot}\mathsf{r}} \qquad \frac{c : \tau_1 \leftrightarrow \tau_2}{\mathsf{idr}_{\odot}\mathsf{l} : c \odot \mathsf{id} \Leftrightarrow c : \mathsf{idr}_{\odot}\mathsf{r}}$$

$$\frac{c : \tau_1 \leftrightarrow \tau_2}{\mathsf{rinv}_{\odot}\mathsf{l} : !c \odot c \Leftrightarrow \mathsf{id} : \mathsf{rinv}_{\odot}\mathsf{r}} \qquad \frac{c : \tau_1 \leftrightarrow \tau_2}{\mathsf{linv}_{\odot}\mathsf{l} : c \odot !c \Leftrightarrow \mathsf{id} : \mathsf{linv}_{\odot}\mathsf{r}}$$

$$\frac{}{\mathsf{sumid} : \mathsf{id} \oplus \mathsf{id} \Leftrightarrow \mathsf{id} : \mathsf{splitid}}$$

$$\frac{c_1 : \tau_5 \leftrightarrow \tau_1 \quad c_2 : \tau_6 \leftrightarrow \tau_2 \quad c_3 : \tau_1 \leftrightarrow \tau_3 \quad c_4 : \tau_2 \leftrightarrow \tau_4}{\mathsf{hom}_{\oplus\odot} : (c_1 \odot c_3) \oplus (c_2 \odot c_4) \Leftrightarrow (c_1 \oplus c_2) \odot (c_3 \oplus c_4) : \mathsf{hom}_{\odot\oplus}}$$

$$\frac{c_1, c_3 : \tau_1 \leftrightarrow \tau_2 \quad c_2, c_4 : \tau_2 \leftrightarrow \tau_3 \quad \alpha_1 : c_1 \Leftrightarrow c_3 \quad \alpha_2 : c_2 \Leftrightarrow c_4}{\alpha_1 \boxdot \alpha_2 : c_1 \odot c_2 \Leftrightarrow c_3 \odot c_4}$$

$$\frac{c_1, c_3 : \tau_1 \leftrightarrow \tau_2 \quad c_2, c_4 : \tau_2 \leftrightarrow \tau_3 \quad \alpha_1 : c_1 \Leftrightarrow c_3 \quad \alpha_2 : c_2 \Leftrightarrow c_4}{\mathsf{resp}_{\oplus\Leftrightarrow}\ \alpha_1\ \alpha_2 : c_1 \oplus c_2 \Leftrightarrow c_3 \oplus c_4}$$

$$\frac{c_1, c_3 : \tau_1 \leftrightarrow \tau_2 \quad c_2, c_4 : \tau_2 \leftrightarrow \tau_3 \quad \alpha_1 : c_1 \Leftrightarrow c_3 \quad \alpha_2 : c_2 \Leftrightarrow c_4}{\mathsf{resp}_{\otimes\Leftrightarrow}\ \alpha_1\ \alpha_2 : c_1 \otimes c_2 \Leftrightarrow c_3 \otimes c_4}$$

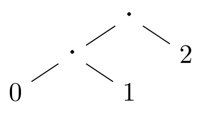
Each 2-combinator α has an inverse $2!\ \alpha$, e.g, $2!\ \mathsf{assoc}_{\odot}\mathsf{l} = \mathsf{assoc}_{\odot}\mathsf{r}$, $2!(\alpha_1 \bullet \alpha_2) = (2!\ \alpha_2) \bullet (2!\ \alpha_1)$, etc.

Figure 2 Π 2-combinators (excerpt) [10]

2.2 Semantics

We give an operational semantics for the 1-combinators of Π which represent the conventional layer of programs. Operationally, the semantics consists of a pair of evaluators that take a combinator and a value and propagate the value in the forward direction $\triangleright$ or in the backward direction $\triangleleft$. We show the complete forward evaluator in Fig. 3; the backward evaluator is easy to infer.

As an example, let $\mathbb{3}$ abbreviate the type $(\mathbb{1} \oplus \mathbb{1}) \oplus \mathbb{1}$. There are three values of type $\mathbb{3}$ which are $ll = \mathsf{inl}(\mathsf{inl}(()))$, $lr = \mathsf{inl}(\mathsf{inr}(()))$, and $r = \mathsf{inr}(())$. Pictorially, the type $\mathbb{3}$ with its three inhabitants can be represented as the left-leaning tree:



$$
\begin{array}{rlcl}
\mathsf{unite_{+}l} \triangleright & (\mathsf{inr}(v)) & = & v \\
\mathsf{uniti_{+}l} \triangleright & v & = & \mathsf{inr}(v) \\
\mathsf{unite_{+}r} \triangleright & (\mathsf{inl}(v)) & = & v \\
\mathsf{uniti_{+}r} \triangleright & v & = & \mathsf{inl}(v) \\
\mathsf{swap_{+}} \triangleright & (\mathsf{inl}(v)) & = & \mathsf{inr}(v) \\
\mathsf{swap_{+}} \triangleright & (\mathsf{inr}(v)) & = & \mathsf{inl}(v) \\
\mathsf{assocl_{+}} \triangleright & (\mathsf{inl}(v)) & = & \mathsf{inl}((\mathsf{inl}(v))) \\
\mathsf{assocl_{+}} \triangleright & (\mathsf{inr}((\mathsf{inl}(v)))) & = & \mathsf{inl}((\mathsf{inr}(v))) \\
\mathsf{assocl_{+}} \triangleright & (\mathsf{inr}((\mathsf{inr}(v)))) & = & \mathsf{inr}(v) \\
\mathsf{assocr_{+}} \triangleright & (\mathsf{inl}((\mathsf{inl}(v)))) & = & \mathsf{inl}(v) \\
\mathsf{assocr_{+}} \triangleright & (\mathsf{inl}((\mathsf{inr}(v)))) & = & \mathsf{inr}((\mathsf{inl}(v))) \\
\mathsf{assocr_{+}} \triangleright & (\mathsf{inr}(v)) & = & \mathsf{inr}((\mathsf{inr}(v)))
\end{array}
\qquad
\begin{array}{rlcl}
\mathsf{unite_{\star}l} \triangleright & ((), v) & = & v \\
\mathsf{uniti_{\star}l} \triangleright & v & = & ((), v) \\
\mathsf{unite_{\star}r} \triangleright & (v, ()) & = & v \\
\mathsf{uniti_{\star}r} \triangleright & v & = & (v, ()) \\
\mathsf{swap_{\star}} \triangleright & (v_1, v_2) & = & (v_2, v_1) \\
\mathsf{assocl_{\star}} \triangleright & (v_1, (v_2, v_3)) & = & ((v_1, v_2), v_3) \\
\mathsf{assocr_{\star}} \triangleright & ((v_1, v_2), v_3) & = & (v_1, (v_2, v_3))
\end{array}
$$

$$
\begin{array}{rlcl}
\mathsf{absorbr} \triangleright & (v, \_) & = & v \\
\mathsf{absorbl} \triangleright & (\_, v) & = & v \\
\mathsf{dist} \triangleright & (\mathsf{inl}(v_1), v_3) & = & \mathsf{inl}((v_1, v_3)) \\
\mathsf{dist} \triangleright & (\mathsf{inr}(v_2), v_3) & = & \mathsf{inr}((v_2, v_3)) \\
\mathsf{factor} \triangleright & \mathsf{inl}((v_1, v_3)) & = & (\mathsf{inl}(v_1), v_3) \\
\mathsf{factor} \triangleright & \mathsf{inr}((v_2, v_3)) & = & (\mathsf{inr}(v_2), v_3) \\
\mathsf{distl} \triangleright & (v_1, \mathsf{inl}(v_3)) & = & \mathsf{inl}((v_1, v_3)) \\
\mathsf{distl} \triangleright & (v_2, \mathsf{inr}(v_3)) & = & \mathsf{inr}((v_2, v_3)) \\
\mathsf{factorl} \triangleright & \mathsf{inl}((v_1, v_3)) & = & (v_1, \mathsf{inl}(v_3)) \\
\mathsf{factorl} \triangleright & \mathsf{inr}((v_2, v_3)) & = & (v_2, \mathsf{inr}(v_3))
\end{array}
\qquad
\begin{array}{rlcl}
\mathsf{id} \triangleright & v & = & v \\
(c_1 \odot c_2) \triangleright & v & = & c_2 \triangleright (c_1 \triangleright v) \\
(c_1 \oplus c_2) \triangleright & (\mathsf{inl}(v)) & = & \mathsf{inl}((c_1 \triangleright v)) \\
(c_1 \oplus c_2) \triangleright & (\mathsf{inr}(v)) & = & \mathsf{inr}((c_2 \triangleright v)) \\
(c_1 \otimes c_2) \triangleright & (v_1, v_2) & = & (c_1 \triangleright v_1, c_2 \triangleright v_2)
\end{array}
$$

Figure 3 Π operational semantics

Note that the values of type $\mathbb{3}$ are the names of the paths from the root to each of the leaves. We use 0, 1 and 2 as ordinals, to give an order to each of the values.

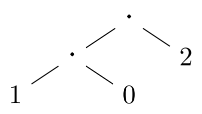
There are, up to equivalence, six combinators of type $\mathbb{3} \leftrightarrow \mathbb{3}$, each representing a different permutation of three elements that leave the *shape* of the three unchanged. The six permutations on $\mathbb{3}$ can be written as Π-terms:

$$
\begin{array}{rcl}
\mathit{perm}_{...} & = & \mathsf{id} \\
\mathit{perm}_{\times\times.} & = & \mathsf{swap}_{+} \oplus \mathsf{id} \\
\mathit{perm}_{.\times\times} & = & \mathsf{assocr}_{+} \odot (\mathsf{id} \oplus \mathsf{swap}_{+}) \odot \mathsf{assocl}_{+} \\
\mathit{perm}_{\rightarrow} & = & \mathit{perm}_{\times\times.} \odot \mathit{perm}_{.\times\times} \\
\mathit{perm}_{\leftarrow} & = & \mathit{perm}_{.\times\times} \odot \mathit{perm}_{\times\times.} \\
\mathit{perm}_{\times.\times} & = & \mathit{perm}_{\rightarrow} \odot \mathit{perm}_{\times\times.}
\end{array}
$$

Tracing the evaluation of $\mathit{perm}_{\times\times.}$ on each of the possible inputs yields:

$$
\begin{array}{rcl}
(\mathsf{swap}_{+} \oplus \mathsf{id}) \triangleright \mathsf{inl}(\mathsf{inl}(())) & = & \mathsf{inl}(\mathsf{swap}_{+} \triangleright \mathsf{inl}(())) \\
 & = & \mathsf{inl}(\mathsf{inr}(())) \\
 & & \\
(\mathsf{swap}_{+} \oplus \mathsf{id}) \triangleright \mathsf{inl}(\mathsf{inr}(())) & = & \mathsf{inl}(\mathsf{swap}_{+} \triangleright \mathsf{inr}(())) \\
 & = & \mathsf{inl}(\mathsf{inl}(())) \\
 & & \\
(\mathsf{swap}_{+} \oplus \mathsf{id}) \triangleright \mathsf{inr}(()) & = & \mathsf{inr}(\mathsf{id} \triangleright ()) \\
 & = & \mathsf{inr}(())
\end{array}
$$

Thus the effect of combinator $\mathit{perm}_{\times\times.}$ is to swap the values $\mathsf{inl}(\mathsf{inl}(()))$ and $\mathsf{inl}(\mathsf{inr}(()))$ leaving the value $\mathsf{inr}(())$ intact. In other words, the effect of $\mathit{perm}_{\times\times.}$ can be visualized as giving the tree:



These trees should also make it clear why mathematicians shorten their notation to $\begin{pmatrix} 0 & 1 & 2 \\ 1 & 0 & 2 \end{pmatrix}$ for the same permutation. We will not do so, as this notation is *untyped*, as it does not enforce that the shape of the tree is preserved.

Iterating $perm_{\times\times.}$ again is equivalent to the identity permutation, which can be verified using 2-combinators:

$$\begin{array}{rcl} perm_{\times\times.} \odot perm_{\times\times.} & = & (\mathsf{swap}_+ \oplus \mathsf{id}) \odot (\mathsf{swap}_+ \oplus \mathsf{id}) \\ & \Leftrightarrow & (\mathsf{swap}_+ \odot \mathsf{swap}_+) \oplus (\mathsf{id} \odot \mathsf{id}) \\ & \Leftrightarrow & \mathsf{id} \oplus \mathsf{id} \\ & = & \mathsf{id} \end{array}$$

More generally we can iterate 1-combinators to produce different reversible functions between finite sets, eventually wrapping around at some number which represents the *order* of the underlying permutation.

▶ **Definition 2** (Iterated Power of a 1-combinator)**.** The k^{th} iterated power of a 1-combinator $p : \tau \leftrightarrow \tau$, for $k \in \mathbb{Z}$ is

$$p^k = \begin{cases} \mathsf{id} & k = 0 \\ p \odot p^{k-1} & k > 0 \\ (!\ p) \odot p^{k+1} & k < 0 \end{cases}$$

▶ **Definition 3** (Order of a 1-combinator)**.** The order of a 1-combinator $p : \tau \leftrightarrow \tau$, $\mathsf{order}(p)$, is the least postitive natural number $k \in \mathbb{N}^+$ such that $p^k \Leftrightarrow \mathsf{id}$.

For our example combinators on the type $\mathbb{3}$, simple traces using the operational semantics show the combinator $perm_{...}$ is the identity permutation; the combinators $perm_{.\times\times}$ and $perm_{\times.\times}$ swap two of the three elements leaving the third intact; and the combinators $perm_{\rightarrow}$ and $perm_{\leftarrow}$ rotate the three elements. We therefore have:

$$\begin{array}{rcl} order(perm_{...}) & = & 1 \\ order(perm_{\times\times.}) = order(perm_{.\times\times}) = order(perm_{\times.\times}) & = & 2 \\ order(perm_{\rightarrow}) = order(perm_{\leftarrow}) & = & 3 \end{array}$$

We should note that the above definition is the only one in this paper which is not *effective*. While there is an obvious method to compute it using the action of a 1-combinator on the elements of the type it acts on, this is extremely inefficient. We do not have an effective algorithm for computing it that works on the syntax of combinators. The (only) difficulty is $\odot$, which can have an arbitrary effect on the order.

The 2-combinators, being complete equivalences between 1-combinators [10], also capture equivalences regarding power of combinators and their order.

▶ **Lemma 4.** *For $p : \tau \leftrightarrow \tau$, $m, n \in \mathbb{Z}$, we have a 2-combinator* $\mathsf{dist}\ p\ m\ n : (p^m \odot p^n) \Leftrightarrow p^{m+n}$.

▶ **Lemma 5.** *For $p : \tau \leftrightarrow \tau$, $n \in \mathbb{Z}$, $p^{k+n} \Leftrightarrow p^n$ where $k = \mathsf{order}(p)$.*

3 From Sets to Groupoids

From a denotational perspective, a Π type τ denotes a finite set, a Π 1-combinator denotes a permutation on finite sets, and the 2-combinators denote coherence conditions on these permutations [10]. Formally, the language Π is a *categorification* [3] of the natural numbers as a *symmetric rig groupoid* [26]. This structure is a *symmetric bimonoidal category* or a *commutative rig category* in which every morphism is invertible. The underlying category consists of two symmetric monoidal structures [24] separately induced by the properties of addition and multiplication of the natural numbers. The monoidal structures are then augmented with distributivity and absorption natural isomorphisms [22] to model the full commutative semiring (aka, commutative rig) of the natural numbers. Despite this rich structure, the individual objects in the category for Π are just plain sets with no interesting structure. In this section we introduce, in the denotation of Π, some non-trivial groupoids which we call *iteration groupoids* and *symmetry groupoids*. Products of these groupoids behave as expected which ensures that a sensible compositional programming language can be designed around them.

3.1 Π Types as Sets (Discrete Groupoids)

Each Π type τ denotes a (structured) finite set $[\![\tau]\!]$ as follows:

$$\begin{array}{rcl}
[\![\mathbb{0}]\!] & = & \bot \\
[\![\mathbb{1}]\!] & = & \top \\
[\![\tau_1 \oplus \tau_2]\!] & = & [\![\tau_1]\!] \uplus [\![\tau_2]\!] \\
[\![\tau_1 \otimes \tau_2]\!] & = & [\![\tau_1]\!] \times [\![\tau_2]\!]
\end{array}$$

where we use $\bot$ to denote the empty set, $\top$ to denote a set with one element, and $\uplus$ and $\times$ to denote the disjoint union of sets and the cartesian product of sets respectively. Each set can be viewed as a groupoid whose objects are the set elements and with only identity morphisms on each object. Nevertheless, the denotations of $\mathbb{1} \oplus (\mathbb{1} \oplus \mathbb{1})$ of $(\mathbb{1} \oplus \mathbb{1}) \oplus \mathbb{1}$ are not in fact equal, although they are trivially isomorphic.

By only being able to express types whose denotations are trivial groupoids, Π leaves untapped an enormous amount of combinatorial structure that is expressible in type theory. We show that with a small but deep technical insight, it is possible to extend Π with types whose denotations are not discrete.

3.2 Groupoids and Groupoid Cardinality

There are many definitions of groupoids that provide complementary perspectives and insights. Perhaps the simplest definition to state, and the one which is most immediately useful for our work, is that a groupoid is a category in which every morphism has an inverse. Intuitively, such a category consists of clusters of connected objects where each cluster is equivalent (in the category-theoretic sense) to a group, viewed as a 1-object category. Thus an alternative definition of a groupoid is as a generalization of a group that allows for individual elements to have "internal symmetries" [31]. Baez et al. [2] associate with each groupoid a cardinality that counts the elements up to these "internal symmetries".

▶ **Definition 6** (Groupoid cardinality [2]). The cardinality of a groupoid G is the (positive)

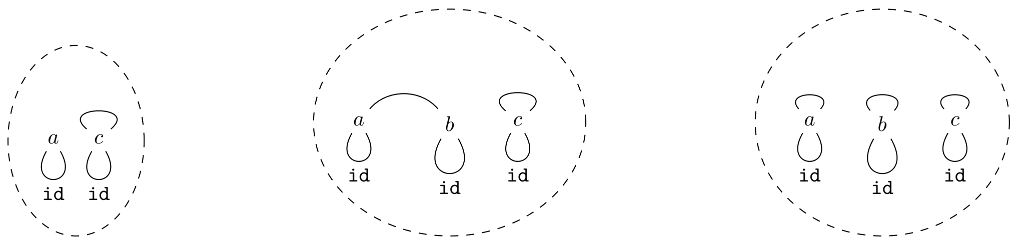


Figure 4 Example groupoids G_1, G_2, and G_3.

real number:

$$|G| = \sum_{[x]} \frac{1}{|\mathsf{Aut}(x)|}$$

provided the sum converges. The summation is over *isomorphism classes* $[x]$ of objects x and $|\mathsf{Aut}(x)|$ is the number of *distinct* automorphisms of x.

For plain sets, the definition just counts the elements as each element is its own equivalence class and has exactly one automorphism (the identity). Without quite formalizing them and relying on the informal diagrams until the next section, we argue that each of the groupoids G_1, G_2, and G_3 in Fig. 4 has cardinality $\frac{3}{2}$. Groupoid G_1 consists of two isomorphism classes: class a has one object with one automorphism (the identity) and class c has one object with two distinct automorphisms; the cardinality is $\frac{1}{1} + \frac{1}{2} = \frac{3}{2}$. For groupoid G_2, we also have two isomorphism classes with representatives a and c; the class containing a has two automorphisms starting from a: the identity and the loop going from a to b and back. By the groupoid axioms, this loop is equivalent to the identity which means that the class containing a has just one automorphism. The isomorphism class of c has two non-equivalent automorphisms on it and hence the cardinality of G_2 is also $\frac{1}{1} + \frac{1}{2} = \frac{3}{2}$. For G_3, we have three isomorphism classes, each with two non-equivalent automorphisms, and hence the cardinality of G_3 is $\frac{1}{2} + \frac{1}{2} + \frac{1}{2} = \frac{3}{2}$. It is important to note that G_1 and G_2 are categorically equivalent groupoids, but that G_3 is not categorically equivalent to either G_1 or G_2. Roughly speaking this is because the number of connected components is also a categorical invariant of a groupoid, and here G_1 and G_2 have 2 whilst G_3 has 3.

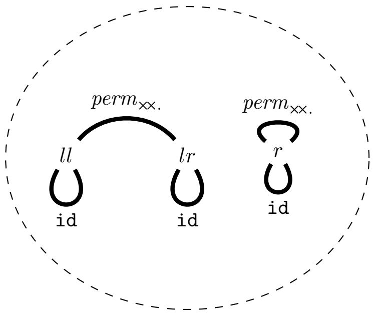
3.3 Π-Combinators as Automorphism Classes

To formalize the counting above, we need, in the context of Π, a precise definition of what it means for automorphisms to be "distinct". We start with an example. Recall the type $\mathbb{3}$ with its three elements $ll = \mathsf{inl}(\mathsf{inl}(()))$, $lr = \mathsf{inl}(\mathsf{inr}(()))$, and $r = \mathsf{inr}(())$. One of the combinators of type $\mathbb{3} \leftrightarrow \mathbb{3}$ is $perm_{\times\times.}$. Observing the results of applying the iterates $(perm_{\times\times.})^k$ for $k \in \mathbb{Z}$ on the three elements we find:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
(perm_{\times\times.})^{2k} \triangleright ll &=& ll & (perm_{\times\times.})^{2k+1} \triangleright ll &=& lr \\
(perm_{\times\times.})^{2k} \triangleright lr &=& lr & (perm_{\times\times.})^{2k+1} \triangleright lr &=& ll \\
(perm_{\times\times.})^{2k} \triangleright r &=& r & (perm_{\times\times.})^{2k+1} \triangleright r &=& r
\end{array}$$

Furthermore, Lem. 5 gives us the following families of 2-combinators $\alpha_{2k} : \mathsf{id} \Leftrightarrow (perm_{\times\times.})^{2k}$ and $\alpha_{2k+1} : perm_{\times\times.} \Leftrightarrow (perm_{\times\times.})^{2k+1}$. We can put these facts together to construct a groupoid whose objects are the elements of $\mathbb{3}$, whose 1-morphisms relate v_i and v_j if

$(perm_{\times\times.})^k \triangleright v_i = v_j$ for some $k \in \mathbb{Z}$, and whose 2-morphisms are the families α_{2k} and α_{2k+1} above. Such a construction produces the following groupoid where each family of 1-morphisms that are identified by a family of 2-morphisms is drawn using a thick line:



Clearly, the resulting groupoid is a reconstruction of G_2 in Fig. 4 using Π types and combinators. As analyzed earlier, this groupoid has cardinality $\frac{3}{2}$. From the perspective of Π, this cardinality corresponds to the number of elements in the underlying set which is 3 divided by the order of the combinator $perm_{\times\times.}$ which is 2. It is important to note that, as Def. 3 states, the calculation of the order of a 1-combinator is defined up to the equivalence induced by 2-combinators.

3.4 Iteration Groupoids $\#p$

The previous construction known in the literature as an *action groupoid* [31] is quite useful: it allows us to take a set of cardinality N and a permutation on that set of order P to construct a groupoid of cardinality $\frac{N}{P}$. Although this idea allows us to construct a groupoid of cardinality $\frac{3}{2}$ as shown above, it is not expressive enough to construct a groupoid of cardinality, say, $\frac{1}{3}$. Indeed, if the underlying set has only one element ($N = 1$) the only permutation is the identity and P must be 1.

The construction, however, already contains the main ingredient needed for the construction of more general groupoids with fractional cardinality. This key piece is the set of iterates of a combinator which we formally define as follows.

▶ **Definition 7** ($\textsc{Iter}(p)$)**.** For each 1-combinator $p : \tau \leftrightarrow \tau$, we form the set $\textsc{Iter}(p)$ whose elements are triples consisting of an integer k, a 1-combinator $q : \tau \leftrightarrow \tau$ and a 2-combinator $\alpha : q \Leftrightarrow p^k$.

Each triple encodes our knowledge that we have some (arbitrary) iterate q of p; we do not have any a priori knowledge of the actual syntactic structure of q, but we do know that it is equivalent to p^k. For example, we have:

$$\textsc{Iter}(perm_{\times\times.}) \quad = \quad \{\langle 0, \mathsf{id}, \mathsf{id}\rangle, \langle 1, perm_{\times\times.}, \mathsf{idr}_{\odot}\mathsf{r}\rangle, \langle -1, perm_{\times\times.}, \mathsf{id}\rangle, \ldots\}$$

The idea is that $\textsc{Iter}(perm_{\times\times.})$ is, up to equivalence, the set of all distinct iterates $(perm_{\times\times.})^k$ of $perm_{\times\times.}$. Because of the underlying group structure of automorphisms, there are, up to equivalence, only $\mathsf{order}(perm_{\times\times.})$ distinct iterates in $\textsc{Iter}(perm_{\times\times.})$. In a proof-irrelevant setting, $\textsc{Iter}(perm_{\times\times.})$ is simply $\{\mathsf{id}, perm_{\times\times.}\}$. In this section and the next, we will use the elements of $\textsc{Iter}(p)$ in two groupoid constructions as either objects (emphasizing their "data" aspect) or morphisms (emphasizing their "symmetry" aspect).

Given a 1-combinator $p : \tau \leftrightarrow \tau$, we define the groupoid $\#p$ as follows. The objects are the elements of $\textsc{Iter}(p)$, i.e., the triples $\langle k, q, \alpha\rangle$ indexed by integers k, 1-combinators

$q : \tau \leftrightarrow \tau$, and 2-combinators $\alpha : q \Leftrightarrow p^k$. We then add (reversible) morphisms between any iterates related by 2-combinators; categorically, this will make any such objects equivalent. If p has order o, Lem. 5 gives us a 2-combinator α which witnesses that $p^i \Leftrightarrow p^{i+o}$. Thus given two iterates $\langle i, q_i, \alpha_i \rangle$ and $\langle i+o, q_j, \alpha_j \rangle$, they must be equivalent since $\alpha_i \bullet \alpha \bullet !\alpha_j$ shows that $q_i \Leftrightarrow q_j$. In other words p^j will be equivalent to p^k exactly when j and k differ by o. This informal description formalizes straightforwardly.

▶ **Definition 8** ($\#p$)**.** For each 1-combinator $p : \tau \leftrightarrow \tau$, we form the groupoid $\#p$ as follows:

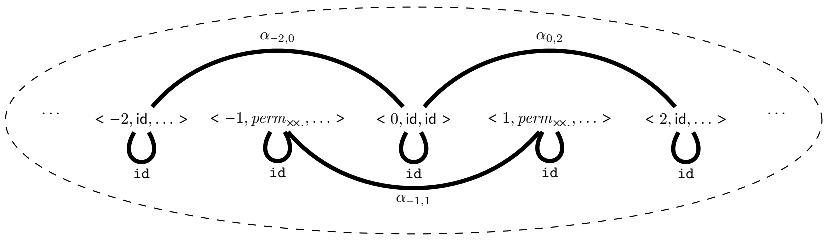
- objects are the elements $\langle k, q, \alpha \rangle$ of $\text{I}\textsc{ter}(p)$;
- there is a morphism between $\langle k_1, q_1, \alpha_1 \rangle$ and $\langle k_2, q_2, \alpha_2 \rangle$ for each $\alpha : q_1 \Leftrightarrow q_2$

Despite its involved internal structure, the groupoid $\#p$ is essentially a set of cardinality $\mathsf{order}(p)$.

▶ **Lemma 9.** $|\#p| = \mathsf{order}(p)$

Proof. Let $o = \mathsf{order}(p)$. There are o isomorphism classes of objects. Consider an object $x = \langle k, q, \alpha \rangle$, its isomorphism class $[x] = \langle k + io, q_i, \alpha_i \rangle$ where $i \in \mathbb{Z}$. The group $\mathsf{Aut}(x)$ is the group generated by id and has order 1. Hence $|\#p| = \sum_1^o \frac{1}{1} = o$. ◀

As an example, the groupoid $\#(perm_{\times\times.})$ can be represented as follows. Up to equivalence, this groupoid is indeed equivalent to a set with two elements id and $perm_{\times\times.}$.


$\alpha_{-2,0}$
$\alpha_{0,2}$
$\cdots$
$< -2, \mathsf{id}, \ldots >$
$< -1, perm_{\times\times.}, \ldots >$
$< 0, \mathsf{id}, \mathsf{id} >$
$< 1, perm_{\times\times.}, \ldots >$
$< 2, \mathsf{id}, \ldots >$
$\cdots$
id
id
id
id
id
$\alpha_{-1,1}$

3.5 Symmetry Groupoids $1/\#p$

The elements of $\text{I}\textsc{ter}(p)$ form a group under the following operation:

$$\langle k_1, p_1, \alpha_1 \rangle \circ \langle k_2, p_2, \alpha_2 \rangle = \langle k_1 + k_2, p_1 \odot p_2, (\alpha_1 \boxdot \alpha_2) \bullet (\mathsf{dist}\ p\ k_1\ k_2) \rangle$$

where $\mathsf{dist}\ p\ k_1\ k_2$ is defined in Lem. 4. The common categorical representation of a group is a category with one trivial object and the group elements as morphisms on that trivial object. Our construction of the groupoid $1/\#p$ is essentially the same.

▶ **Definition 10** ($1/\#p$)**.** For each 1-combinator $p : \tau \leftrightarrow \tau$, we form the 2-groupoid $1/\#p$ as follows:

- the objects are the iterates of the identity combinator on τ;
- the morphisms between every pair of objects are the elements of $\text{I}\textsc{ter}(p)$;
- there is a 2-morphism between 1-morphisms $\langle k_1, q_1, \alpha_1 \rangle$ and $\langle k_2, q_2, \alpha_2 \rangle$ for every $\alpha : q_1 \Leftrightarrow q_2$.

Note that for each power p^i of p, there is a morphism $\langle k, q, \alpha \rangle$ in $\text{I}\textsc{ter}(p)$ such that q annihilates p^i to the identity.

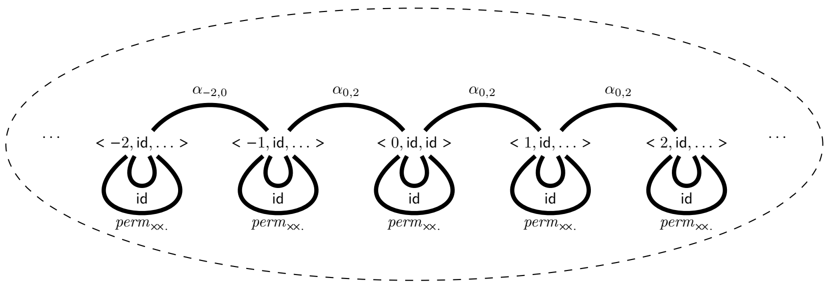
Remark.

Note also that everything is well-defined even if we choose $p : \mathbb{0} \leftrightarrow \mathbb{0}$. In that case, the cardinality is 1.

▶ **Lemma 11.** $|1/\#p| = \frac{1}{\mathsf{order}(p)}$

Proof. Let $o = \mathsf{order}(p)$. The objects form one isomorphism class $[p]$. There are, up to equivalence, exactly $\mathsf{order}(p)$ distinct morphisms on this equivalence class. Hence, the group $\mathsf{Aut}([p])$ is the group generated by $p^0, p^1 \dots p^{o-1}$, and the cardinality $|1/\#p|$ is $\frac{1}{o}$. ◀

As an example, the groupoid $1/\#(perm_{\mathsf{XX}.})$ can be represented as follows.



3.6 Looping and Delooping

Our constructions of $\#p$ and $1/\#p$ are instances of a general pattern that we have instantiated to the setting of Π and adapted to explicitly refer to weak (i.e., up to $\Leftrightarrow$ in our case) equivalences. In the general construction [23], a group G is known to induce two groupoids, a looping groupoid $\mathcal{L}G$ and a delooping groupoid $\mathcal{B}G$. These are defined as follows.

The looping groupoid $\mathcal{L}G$ has the elements g of the group G as objects and morphisms between elements in the same conjugacy class; that is, there is a morphism between elements g and h iff there exists an $x \in G$ such that $h = x^{-1} \cdot g \cdot x$. Fixing a type τ and a particular 1-combinator $p : \tau \leftrightarrow \tau$, we get a group whose elements are the iterates p^k. The corresponding looping groupoid has all the iterates as objects, and morphisms between p^{k_1} and p^{k_2} whenever there exists an integer j such that p^{k_2} is equivalent to $p^{-j} \odot p^{k_1} \odot p^j$. By Lem. 4, the latter term is $\Leftrightarrow$-equivalent to p^{-j+k_1+j} which is $\Leftrightarrow$-equivalent to p^{k_1}. In other words, we have a morphism between p^{k_1} and p^{k_2} whenever they are $\Leftrightarrow$-equivalent which is consistent with our definition of iteration groupoids.

The delooping groupoid $\mathcal{B}G$ has one trivial element $*$, and for each group element g a morphism (loop) from $*$ to $*$. Modulo $\Leftrightarrow$-equivalences, this is consistent with our definition of symmetry groupoids.

3.7 Products of Iteration and Symmetry Groupoids

If we are to build a compositional programming language around iteration and symmetry groupoids, we need to ensure that they compose sensibly with the existing type formers. Groupoids, viewed as categories, come associated with natural notions of sums and products, and one might expect or hope that identities which hold for rational numbers lift to identities in our situation. The integration of these groupoids with the categorical sum is complicated and left for future work. For products, the most prominent identity, from which many other identities follow, is:

$$\#p \circledast 1/\#p \simeq \#\mathsf{id}$$

The left hand side is the categorical product of the iteration groupoid and symmetry groupoid for some arbitrary 1-combinator p. The cardinality of this groupoid is 1. The right hand side is the iteration groupoid for the identity 1-combinator which also has cardinality 1. The groupoids on either side are however not isomorphic, there do not exist full and faithful functors between them, and hence $\simeq$ cannot be categorical equivalence.

There are however weaker notions of groupoid equivalence, such as *Morita equivalence* [23][17, C5.3], that identify the two groupoids above. Although the notion of equivalence we use in the next section appears novel, it is related "in spirit" to Morita equivalence and we find it useful to give a small example. We will verify that the product of the groupoid $\#perm_{\leftharpoonup}$ of cardinality 3 and the groupoid $1/\#perm_{\leftharpoonup}$ of cardinality $\frac{1}{3}$ is Morita equivalent to the trivial groupoid with only one distinct object and one trivial automorphism. To avoid clutter, we assume the groupoids are strict, e.g., that the groupoid $\#perm_{\leftharpoonup}$ has three elements (instead of 3 isomorphism classes of elements).

Getting an equivalence in this case reduces to finding a *Morita morphism* from $\#\mathsf{id}$ to $\#perm_{\leftharpoonup} \circledast 1/\#perm_{\leftharpoonup}$. This is a functor ψ from the unit groupoid to the product groupoid, say G, with an additional condition that the commutative diagram below is a *pullback*, where s, t are the *source* and *target* maps, and $\mathbb{1}$, $\mathbb{3}$ are the 1- and 3-point discrete groupoids defined previously.

$$\begin{tikzcd}
Q \arrow[rrd, bend left, "\xi_1"] \arrow[rd, dotted] \arrow[rdd, bend right, "(s,t)"'] & & \\
& \#\mathsf{id} \arrow[r, "\psi_1"] \arrow[d, "(s,t)"] & G \arrow[d, "(s,t)"] \\
& \mathbb{1} \times \mathbb{1} \arrow[r, "\psi_0 \times \psi_0"] & \mathbb{3} \times \mathbb{3}
\end{tikzcd}$$

The intuition is that the codomain consists of three objects with associated morphisms, each of them counting as a "third." The functor from $\#\mathsf{id}$ to $\#perm_{\leftharpoonup} \circledast 1/\#perm_{\leftharpoonup}$ must include an arbitrary choice of "which third" to target and the universality condition of the pullback ensures that the choice is ultimately insignificant as it leads to the same result as every other possible functor.

To summarize, the equivalence between the two groupoids requires a *local* map that chooses a target and a *global* condition ensuring that the choice ultimately annihilates. The challenge we aim to solve in the next section is to turn this idea into an operational semantics that somehow combines the local and global aspects of the equivalence. We will be inspired by the folklore result in the categorical semantics of dependent type theory [5], first explained by Paul Taylor [28], that "substitution into a dependent type is a pullback".

4 $\Pi^{/}$: Syntax and Semantics

We are now ready to turn the groupoid constructions from the previous section into an extension of Π with new type constructors and combinators for creating and manipulating syntactic counterparts to $\#p$ and $1/\#p$. Just adding the "obvious" types and combinators does not work; however, as the failures are quite informative, we nevertheless go through this first attempt. This will help illustrate why we need extra machinery in Sec. 4.2.

$$
\begin{array}{rrcll}
\mathsf{unite}_\star\mathsf{l}/ : & \#\mathsf{id} \circledast \mathbb{T} & \multimap\!\!\circ & \mathbb{T} & : \mathsf{uniti}_\star\mathsf{l}/ \\
\mathsf{unite}_\star\mathsf{r}/ : & \mathbb{T} \circledast \#\mathsf{id} & \multimap\!\!\circ & \mathbb{T} & : \mathsf{uniti}_\star\mathsf{r}/ \\
\mathsf{swap}_\star/ : & \mathbb{T}_1 \circledast \mathbb{T}_2 & \multimap\!\!\circ & \mathbb{T}_2 \circledast \mathbb{T}_1 & : \mathsf{swap}_\star/ \\
\mathsf{assocl}_\star/ : & \mathbb{T}_1 \circledast (\mathbb{T}_2 \circledast \mathbb{T}_3) & \multimap\!\!\circ & (\mathbb{T}_1 \circledast \mathbb{T}_2) \circledast \mathbb{T}_3 & : \mathsf{assocr}_\star/ \\
\\
\eta- : & \#\mathsf{id} & \multimap\!\!\circ & 1/\#c \circledast \#c & : \epsilon- \\
\eta+ : & \#\mathsf{id} & \multimap\!\!\circ & \#c \circledast 1/\#c & : \epsilon+
\end{array}
$$

$$
\frac{}{\mathsf{id}/ : \mathbb{T} \multimap\!\!\circ \mathbb{T}}
\qquad
\frac{\rho_1 : \mathbb{T}_1 \multimap\!\!\circ \mathbb{T}_2 \quad \rho_2 : \mathbb{T}_2 \multimap\!\!\circ \mathbb{T}_3}{\rho_1 \circledcirc \rho_2 : \mathbb{T}_1 \multimap\!\!\circ \mathbb{T}_3}
$$

$$
\frac{\alpha : c_1 \Leftrightarrow c_2}{\#\alpha : \#c_1 \multimap\!\!\circ \#c_2}
\qquad
\frac{\rho_1 : \mathbb{T}_1 \multimap\!\!\circ \mathbb{T}_2 \quad \rho_2 : \mathbb{T}_3 \multimap\!\!\circ \mathbb{T}_4}{\rho_1 \circledast \rho_2 : \mathbb{T}_1 \circledast \mathbb{T}_3 \multimap\!\!\circ \mathbb{T}_2 \circledast \mathbb{T}_4}
$$

Each combinator ρ has an inverse.

Figure 5 $\Pi^/$ fractional combinators (non-dependent version)

4.1 First attempt: non-dependent version

As a first approximation to the language we seek, we consider adding three new syntactic categories to the definition of Π in Sec. 2.1:

▶ **Definition 12** (Non-dependent $\Pi^/$). The syntax of (non-dependent) $\Pi^/$ is given by the following categories:

$$
\begin{array}{llcl}
\text{(Types)} & \tau & ::= & [\textit{see Def. 1}] \\
\text{(Values)} & v & ::= & [\textit{see Def. 1}] \\
\text{(1-combinators)} & c & : & \tau_1 \leftrightarrow \tau_2 \ ::= [\textit{see Def. 1}] \\
\text{(2-combinators)} & \alpha & : & c_1 \Leftrightarrow c_2 \text{ where } c_1, c_2 : \tau_1 \leftrightarrow \tau_2 \ ::= [\textit{see Def. 1}] \\
\\
\text{(Fractional Types)} & \mathbb{T} & ::= & \#c \mid 1/\#c \mid \mathbb{T}_1 \circledast \mathbb{T}_2 \\
\text{(Fractional Values)} & \mathbb{V} & ::= & c^k \mid 1/c^k \mid \langle \mathbb{V}, \mathbb{V} \rangle \\
\text{(/-combinators)} & \rho & : & \mathbb{T}_1 \multimap\!\!\circ \mathbb{T}_2 \ ::= [\textit{see Fig. 5}]
\end{array}
$$

The syntactic category $\mathbb{T}$ of fractional types introduces type expressions for iteration groupoids, symmetry groupoids, and their products. The combinators that relate these fractional types are in Fig. 5. The bottom group includes identity and sequential composition $\circledcirc$ combinators and ensures that the combinators can be applied anywhere inside a product $\circledast$. There is also a combinator that relates two types $\#c_1$ and $\#c_2$ via $\#\alpha$ whenever $\alpha : c_1 \Leftrightarrow c_2$. Indeed if c_1 and c_2 are considered equivalent via a 2-combinator then we also consider their iteration groupoids to be equivalent. The top group ensures that $\circledast$ carries a symmetric monoidal structure with combinators witnessing the unit, commutativity, and associativity properties. The last two pairs of combinators $\eta-$ /$\epsilon-$ and $\eta+$ /$\epsilon+$ are inspired from the definition of compact closed categories [19] which are symmetric monoidal categories in which every object has a dual. They are the ones that witness the "fractional" nature of symmetry groupoids as motivated in Sec. 3.7.

The last new syntactic category in $\Pi^/$ is that of values which deserves some additional discussion. When types denote sets, values of a type are clear: they are just the elements of

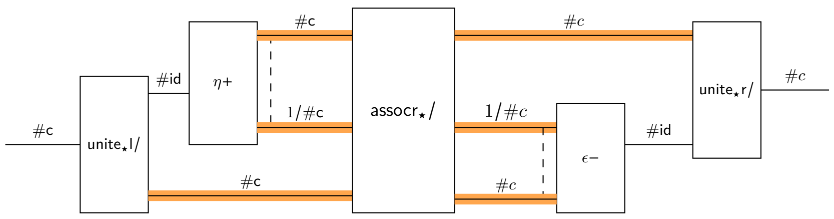


Figure 6 Zigzag

the set. In the case when types are groupoids, this is less clear, especially when the groupoid cardinality is a proper fraction. What we get instead are equivalence classes of values. The idea is not uncommon: in the conventional λ-calculus, we list $\lambda x.x$ and $\lambda y.y$ as separate values of type $\tau \to \tau$ and then provide a separate equivalence relation (α-equivalence) to express the fact that these two values are indistinguishable. The treatment in our setting is similar: for iteration groupoids $\#c$, every distinct iterate c^k is listed as a separate value with the understanding that some iterates are $\Leftrightarrow$-equivalent. Similarly for symmetry groupoids $1/\#c$, every morphism is listed as a value $1/c^k$ with the same understanding that some of these morphisms are $\Leftrightarrow$-equivalent.

To understand the challenge in designing an operational semantics for η and ϵ in the non-dependent setting, we consider the "zigzag" circuit in Fig. 6. Algebraically this circuit corresponds to the following manipulation on types:

$$x \mapsto (1 \times x) \mapsto ((x \times 1/x) \times x) \mapsto (x \times (1/x \times x)) \mapsto (x \times 1) \mapsto x$$

and by the coherence conditions of compact closed categories, we expect this circuit, when run in either direction, to be equivalent to the identity. Consider now the case of a combinator c whose order is greater than or equal to 2, i.e., the type $\#c$ has at least two distinct values id and c and the type $1/\#c$ has at least two distinct values $1/\mathsf{id}$ and $1/c$. We require this circuit to produce id when given id from either end, and similarly to produce c when given c from either end. The problem becomes apparent when one observes that the uses of $\eta+$ in the left-to-right execution and the use of $\epsilon-$ in the right-to-left execution must behave differently depending on the incoming value. But the "local" information available to either of these combinators does not include the "global" information about the incoming value and there is no way for them to make consistent guesses locally. It is possible to design an operational semantics that uses computational effects to enable spatially separated parts of the program to communicate in a way that is reminiscent of entanglement in quantum mechanics. Possibilities to realize this communication are global reference cells, backtracking, and logical variables with unification. We choose however to present in the next section a more abstract approach that encodes the required dependency in dataflow constraints using dependent types.

4.2 Dependent Pointed Types

We modify the syntax of $\Pi^{/}$ as follows.

▶ **Definition 13** (Dependent $\Pi^{/}$). The syntax of dependent $\Pi^{/}$ differs from Def. 12 as follows:

(Fractional Types)	$\mathbb{T}_\bullet$	::=	$[\#c, c^k] \mid [1/\#c, 1/c^k] \mid [\mathbb{T}_1 \circledast \mathbb{T}_2, \langle v_1, v_2 \rangle] \mid \forall k.\mathbb{T}_\bullet \mid \exists k.\mathbb{T}_\bullet$
(/-combinators)	$\rho_\bullet$	:	$\mathbb{T}_{1\bullet} \leftrightarrow \mathbb{T}_{2\bullet}$ ::= [*see Fig. 7*]

$$
\begin{array}{rrcll}
\mathsf{unite}_{\star}\mathsf{l}/ : & \boxtimes\overline{k}.[\#\mathsf{id} \circledast \mathbb{T}, \langle \mathsf{id}, v\rangle] & \multimapdotboth & \boxtimes\overline{k}.[\mathbb{T}, v] & : \mathsf{uniti}_{\star}\mathsf{l}/ \\
\mathsf{unite}_{\star}\mathsf{r}/ : & \boxtimes\overline{k}.[\mathbb{T} \circledast \#\mathsf{id}, \langle v, \mathsf{id}] & \multimapdotboth & \boxtimes\overline{k}.[\mathbb{T}, v] & : \mathsf{uniti}_{\star}\mathsf{r}/ \\
\mathsf{swap}_{\star}/ : & \boxtimes\overline{k}.[\mathbb{T}_1 \circledast \mathbb{T}_2, \langle v_1, v_2\rangle] & \multimapdotboth & \boxtimes\overline{k}.[\mathbb{T}_2 \circledast \mathbb{T}_1, \langle v_2, v_1\rangle] & : \mathsf{swap}_{\star}/ \\
\mathsf{assocl}_{\star}/ : & \boxtimes\overline{k}.[\mathbb{T}_1 \circledast (\mathbb{T}_2 \circledast \mathbb{T}_3), \langle v_1, \langle v_2, v_3\rangle] & \multimapdotboth & \boxtimes\overline{k}.[(\mathbb{T}_1 \circledast \mathbb{T}_2) \circledast \mathbb{T}_3, \langle\langle v_1, v_2\rangle, v_3\rangle] & : \mathsf{assocr}_{\star}/ \\
\\
\eta- : & [\#\mathsf{id}, \mathsf{id}] & \multimapdotboth & \boxtimes k.[1/\#c \circledast \#c, \langle 1/c^k, c^k\rangle] & : \epsilon- \\
\eta+ : & [\#\mathsf{id}, \mathsf{id}] & \multimapdotboth & \boxtimes k.[\#c \circledast 1/\#c, \langle c^k, 1/c^k\rangle] & : \epsilon+ \\
\mathsf{synchl} : & \boxtimes k.[(\#c \circledast 1/\#c) \circledast \#c, \langle\langle c^k, 1/c^k\rangle, c^i] & \multimapdotboth & [(\#c \circledast 1/\#c) \circledast \#c, \langle\langle c^i, 1/c^i\rangle, c^i] & : \mathsf{packl} \\
\mathsf{synchr} : & \boxtimes k.[\#c \circledast (1/\#c \circledast \#c), \langle c^i, \langle 1/c^k, c^k\rangle\rangle] & \multimapdotboth & [\#c \circledast (1/\#c \circledast \#c), \langle c^i, \langle 1/c^i, c^i\rangle\rangle] & : \mathsf{packr}
\end{array}
$$

$$
\frac{}{\mathsf{id}/ : \boxtimes\overline{k}.[\mathbb{T}, v] \multimapdotboth \boxtimes\overline{k}.[\mathbb{T}, v]}
\qquad
\frac{\alpha : c_1 \Leftrightarrow c_2}{(\#\alpha)_k : [\#c_1, c_1^k] \multimapdotboth [\#c_2, c_2^k]}
$$

$$
\frac{\rho_1 : \boxtimes\overline{k_1}.[\mathbb{T}_1, v_1] \multimapdotboth \boxtimes\overline{k_2}.[\mathbb{T}_2, v_2] \quad \rho_2 : \boxtimes\overline{k_2}.[\mathbb{T}_2, v_2] \multimapdotboth \boxtimes\overline{k_3}.[\mathbb{T}_3, v_3]}{\rho_1 \circledcirc \rho_2 : \boxtimes\overline{k_1}.[\mathbb{T}_1, v_1] \multimapdotboth \boxtimes\overline{k_3}.[\mathbb{T}_3, v_3]}
$$

$$
\frac{\rho_1 : \boxtimes\overline{k_1}.[\mathbb{T}_1, v_1] \multimapdotboth \boxtimes\overline{k_2}.[\mathbb{T}_2, v_2] \qquad \rho_2 : \boxtimes\overline{k_3}.[\mathbb{T}_3, v_3] \multimapdotboth \boxtimes\overline{k_4}.[\mathbb{T}_4, v_4]}{\rho_1 \circledast \rho_2 : \boxtimes\overline{k_1 k_3}.[\mathbb{T}_1 \circledast \mathbb{T}_3, \langle v_1, v_3\rangle] \multimapdotboth \boxtimes\overline{k_2 k_4}.[\mathbb{T}_2 \circledast \mathbb{T}_4, \langle v_2, v_4\rangle]}
$$

Each combinator $\rho_\bullet$ has an inverse $\rho_\bullet^{-1}$. A rule $\mathbb{T}_{\mathbb{1}\bullet} \multimapdotboth \boxtimes k.\mathbb{T}_{2\bullet}$ introduces a $\forall$ in the left-to-right direction and eliminates an $\exists$ in the right-to-left direction. A rule $\boxtimes k.\mathbb{T}_{\mathbb{1}\bullet} \multimapdotboth \mathbb{T}_{2\bullet}$ eliminates a $\forall$ in the left-to-right direction and introduces an $\exists$ in the right-to-left direction. A rule $\boxtimes k.\mathbb{T}_{\mathbb{1}\bullet} \multimapdotboth \boxtimes k.\mathbb{T}_{2\bullet}$ maintains (think of elimination then introduction) the universal quantifier in the left-to-right direction and the existential quantifier in the right-to-left direction. We omit the quantifiers when the sequence of bound variables is empty.

Figure 7 $\Pi^{/}$ fractional combinators (dependent version)

In the new definition, we enrich the types to pointed types $\mathbb{T}_\bullet$ where each type $\mathbb{T}$ refers to a particular value v in $\mathbb{T}$. As all fractional types are inhabited (see Remark in Sec. 3.5), there is no loss in generality in going to pointed types. We then allow universal quantification over indices k associated to values, over a collection of types. In particular, the type $\forall k.[\#c \circledast 1/\#c, \langle c^k, 1/c^k\rangle]$ imposes a constraint on the two values associated with the types $\#c$ and $1/\#c$; they must be “synchronized” by sharing the same k but are otherwise arbitrary. Once we introduce a universal quantifier in one direction of execution, we are led to also include an existential quantifier for the reverse execution. Intuitively, a forward evaluation step that is at liberty to choose a k will, when reversed encounter a particular, i.e., existentially quantified, k. A type $\mathbb{T}_\bullet$ is therefore generally of the form $\boxtimes\overline{k}.[\mathbb{T}, v]$ where all the quantifiers are lifted to the top followed by a non-dependent type and a value in that type.

The adaptation of the combinators to the pointed dependent setting in Fig . 7 is relatively straightforward. First, the types of the combinators must keep track of how the value currently in focus changes during evaluation, and hence directly encode the operational semantics of the language. As suggested above, the quantifiers are interpreted differently depending on which direction the rule is used. We use a $\boxtimes$ notation to express the presence of *one* quantifier that is interpreted differently in each direction. The notation has the advantage of being compact, avoiding duplication, and making the symmetry of the rules explicit. We also include two new pairs of combinators synchl/packl and synchr/packr that serve as explicit quantifier introduction and elimination rules: they serve as explicit “synchronization” points as illustrated in the example below.

The example revisits the zigzag circuit in Fig. 6 using the dependent types and the new explicit synchronization primitives. The evaluation proceeds as follows where we disambiguate the uses of ∀ and ∃ in each direction for clarity:

Forward execution of zigzag circuit with synchronization.

$$\begin{array}{lcl}
[\#c, c^i] & \overset{\mathsf{unite}_\star\mathsf{l}/}{\leftrightarrow} & [\#id \circledast \#c, \langle \mathsf{id}, c^i \rangle] \\
 & \overset{\eta{+}\circledast\mathsf{id}/}{\leftrightarrow} & \forall k.[(\#c \circledast 1/\#c) \circledast \#c, \langle\langle c^k, 1/c^k \rangle, c^i \rangle] \\
 & \overset{\mathsf{synchl}}{\leftrightarrow} & [\langle\langle c^i, 1/c^i \rangle, c^i \rangle] \\
 & \overset{\mathsf{assocr}_\star/}{\leftrightarrow} & [\#c \circledast (1/\#c \circledast \#c), \langle c^i, \langle 1/c^i, c^i \rangle\rangle] \\
 & \overset{\mathsf{packr}}{\leftrightarrow} & \exists k.[\#c \circledast (1/\#c \circledast \#c), \langle c^i, \langle 1/c^k, c^k \rangle\rangle] \\
 & \overset{\mathsf{id}/\circledast\epsilon-}{\leftrightarrow} & [\#c \circledast \#\mathsf{id}, \langle c^i, \mathsf{id} \rangle] \\
 & \overset{\mathsf{unite}_\star\mathsf{r}/}{\leftrightarrow} & [\#c, c^i]
\end{array}$$

Reverse execution of zigzag circuit with synchronization.

$$\begin{array}{lcl}
[\#c, c^i] & \overset{\mathsf{unite}_\star\mathsf{r}/}{\leftrightarrow} & [\#c \circledast \#\mathsf{id}, \langle c^i, \mathsf{id} \rangle] \\
 & \overset{\mathsf{id}/\circledast\epsilon-}{\leftrightarrow} & \forall k.[\#c \circledast (1/\#c \circledast \#c), \langle c^i, \langle 1/c^k, c^k \rangle\rangle] \\
 & \overset{\mathsf{synchr}}{\leftrightarrow} & [(\#c \circledast 1/\#c) \circledast \#c, \langle\langle c^i, 1/c^i \rangle, c^i \rangle] \\
 & \overset{\mathsf{assocr}_\star/}{\leftrightarrow} & [(\#c \circledast 1/\#c) \circledast \#c, \langle\langle c^i, 1/c^i \rangle, c^i \rangle] \\
 & \overset{\mathsf{packl}}{\leftrightarrow} & \exists k.[(\#c \circledast 1/\#c) \circledast \#c, \langle\langle c^k, 1/c^k \rangle, c^i \rangle] \\
 & \overset{\eta{+}\circledast\mathsf{id}/}{\leftrightarrow} & [\#\mathsf{id} \circledast \#c, \langle \mathsf{id}, c^i \rangle] \\
 & \overset{\mathsf{unite}_\star\mathsf{l}/}{\leftrightarrow} & [\#c, c^i]
\end{array}$$

4.3 Additional Examples

Compact closed categories support various interesting constructions [1], such as duals, name and coname, and trace [18, 15], which eventually lead to a certain notion of recursive higher-order functions. These constructions do not immediately transfer to the context of $\Pi^/$ as we have pointed types with explicit quantifiers and synchronization primitives that are not present in the conventional setting. Adapting the conventional constructions requires resolving some subtleties and will provide the first operational interpretation known to us of multiplicative higher-order recursive functions.

5 Conclusion

We have presented a natural notion of *fractional types* that enriches a class of reversible programming languages in several dimensions. While further research might show how to use fractional types in a conventional (i.e., irreversible) programming language, their full potential is only achieved when the ambient language guarantees that no information is created or erased. The key semantic insight is that iterating a reversible program p on a finite type must eventually reach the identity in $\mathsf{order}(p)$ steps. By being careful not to collapse proofs, each such reversible program has $\mathsf{order}(p)$ distinct proofs of reversibility and hence gives rise to a groupoid with cardinality $\frac{1}{\mathsf{order}(p)}$. Going from this observation to a full programming language required several difficult and subtle design choices which we have explored to produce $\Pi^{/}$. As with most functional languages, it has a natural denotational semantics where types denote groupoids. Its operational semantics requires a mechanism to express a computational effect which enables spatially separated parts of the program to communicate in a way that is reminiscent of entanglement in quantum mechanics. It appears possible to realize such an operational semantics using global reference cells, backtracking, or other conventional technique. Rather than going via effects, we chose dependent types to track the required constraints via dataflow. Of our various attempts at an operational semantics, this one turned out simplest.

Our fractional types have natural denotations which are non-trivial groupoids, but they cannot be composed with sums and they only scratch the surface of the tower of weak ω-groupoids that is expressible in HoTT [29]. A long term goal of our research is to find natural type constructors inspired by the rich combinatorial structure of weak ω-groupoids that provide novel programming abstractions.

We conclude by outlining several interesting connections and potential avenues for future work.

Quotient Types.

Certain groupoids naturally correspond to conventional *quotient types.* Traditionally [25], a quotient type $T /\!\!/ E$ combines a type T with an equivalence relation E that serves as the equality relation on the elements of T. Our notion of fractional types in $\Pi^{/}$ appears to subsume conventional quotient types and their applications [11] (e.g., defining fractions, multivariate polynomials, field extensions, algebraic numbers, etc.) and it would interesting to explicitly formalize this connection.

Conservation of Information and Negative Entropy.

According to the conventional theory of communication [27], a type with N values is viewed as an abstract system that has N distinguishable states where the amount of information contained in each state is $(\log N)$. This entropy is a measure of information which materializes itself in memory or bandwidth requirements when storing or transmitting elements of this type. Thus a type with 8 elements needs 3 bits of memory for storage or 3 bits of bandwidth for communication. The logarithmic map implies that information contained in a composite state is the sum of the information contained in its constituents. For example, the type $\mathbb{2} \otimes \mathbb{3}$ can be thought of a composite system consisting of two independent unrelated subsystems. Each state of the composite system therefore contains $\log(2 * 3) = \log 2 + \log 3$ bits which is the sum of the information contained in each subsystem. If quantum field theory is correct (as it so far seems to be) then information, during any physical process, is neither created nor destroyed. Landauer [20, 9, 21], Bennet [6, 7, 8], Fredkin [14] and others made compelling arguments that this physical principle induces a corresponding computational principle of "conservation of information." In the context of finite types, generated from the empty type $\mathbb{0}$, the unit type $\mathbb{1}$, and sums and products $\oplus$ and $\otimes$, this principle states that the foundational (i.e., physical) notion of computation is computation via type isomorphisms [16] or type equivalences [10]; these are both sound and complete with respect to cardinality-preserving maps. The introduction, in $\Pi^{/}$, of types (groupoids) with fractional cardinalities introduces types with *negative entropy.* For example, a type with cardinality $\frac{1}{8}$ has entropy $\log \frac{1}{8} = -3$. It is natural to interpret this negative entropy just like we interpret "negative money," as a debt to be repaid by some other part of the system. In fact, the zigzag example can be viewed as modeling a credit card transaction where the money is produced at the output site by generating a corresponding debt that is reconciled at the input site.

Correspondence with Commutative Semifields.

Computations over finite types naturally emerge from viewing types as syntax for semiring elements, semiring identities as type isomorphisms, and justifications for semiring identities as program transformations and optimizations [10]. This correspondence provides a rich proof-relevant version of the Curry-Howard correspondence between algebra and reversible programming languages. The addition of full fractional types to the mix would enrich the correspondence to commutative semifields, providing a categorification [3] of the non-negative rational numbers in a computational setting. This correspondence in $\Pi^{/}$ is still lacking, however. Given any non-negative rational number, we can form a type whose cardinality is that number. And yet, our types do not capture the full structure of the non-negative rational numbers, as these form a commutative semifield. There are a number of operations and properties which need to be added to complete this. Most seem quite straightforward. We can however single one out which may not be: what we are missing is a multiplicative inverse for every type, and not just $\#p$. In particular, we would like to form the type $1/\#(1/\#p)$ and have it be equivalent to $\#p$. As explained in Sec. 3, this would require a further generalization of looping and delooping that can be iterated. We leave a possible extension with a general looping/delooping process to future work.

References

1 Samson Abramsky and Bob Coecke. Categorical quantum mechanics, 2008. `arXiv:0808.1023` [quant-ph].

2 J. C. Baez, A. E. Hoffnung, and C. D. Walker. Higher-Dimensional Algebra VII: Groupoidification. *ArXiv e-prints*, August 2009. `arXiv:0908.4305`.

3 John C. Baez and James Dolan. Categorification. In Higher Category Theory, Contemp. Math. 230, 1998, pp. 1-36., 1998.

4 John C. Baez and James Dolan. From finite sets to Feynman diagrams. *Mathematics Unlimited - 2001 and beyond, Springer*, 1:29–50, 2001.

5 Andrej Bauer. Substitution is pullback. `http://math.andrej.com/2012/09/28/substitution-is-pullback/`, 2012.

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

7 C.H. Bennett. Notes on Landauer's principle, reversible computation, and Maxwell's Demon. *Studies In History and Philosophy of Science Part B: Studies In History and Philosophy of Modern Physics*, 34(3):501–510, 2003.

8 C.H. Bennett. Notes on the history of reversible computation. *IBM Journal of Research and Development*, 32(1):16–23, 2010.

9 C.H. Bennett and R. Landauer. The fundamental physical limits of computation. *Scientific American*, 253(1):48–56, 1985.

10 Jacques Carette and Amr Sabry. *ESOP 2016*, chapter Computing with Semirings and Weak Rig Groupoids, pages 123–148. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

11 Cyril Cohen. *Interactive Theorem Proving: 4th International Conference, ITP 2013*, chapter Pragmatic Quotient Types in Coq, pages 213–228. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

12 M. P. Fiore, R. Di Cosmo, and V. Balat. Remarks on isomorphisms in typed calculi with empty and sum types. *Annals of Pure and Applied Logic*, 141(1-2):35–50, 2006.

13 Marcelo Fiore. Isomorphisms of generic recursive polynomial types. In *POPL*, pages 77–88. ACM, 2004.

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

15 Masahito Hasegawa. Recursion from cyclic sharing: Traced monoidal categories and models of cyclic lambda calculi. In *TLCA*, pages 196–213, 1997.

16 Roshan P. James and Amr Sabry. Information effects. In *POPL*, pages 73–84. ACM, 2012.

17 Peter T Johnstone. *Sketches of an elephant: A topos theory compendium*, volume 2. Oxford University Press, 2002.

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

19 G. M. Kelly and M. L. Laplaza. Coherence for compact closed categories. *Journal of Pure and Applied Algebra*, pages 193–213, 1980.

20 R. Landauer. Irreversibility and heat generation in the computing process. *IBM J. Res. Dev.*, 5:183–191, July 1961.

21 Rolf Landauer. The physical nature of information. *Physics Letters A*, 1996.

22 Miguel L. Laplaza. Coherence for distributivity. In G.M. Kelly, M. Laplaza, G. Lewis, and Saunders Mac Lane, editors, *Coherence in Categories*, volume 281 of *Lecture Notes in Mathematics*, pages 29–65. Springer Verlag, Berlin, 1972. `doi:10.1007/BFb0059555`.

23 Ernesto Lupercio and Bernado Uribe. Loop groupoids, gerbes, and twisted sectors on orbifolds. `arXiv:math/0110207` [math.AT], 2002.

24 Saunders Mac Lane. *Categories for the working mathematician*. Springer-Verlag, 1971.

25 N. Mendler. Quotient types via coequalizers in martin-löf type theory. In *Proceedings of the Logical Frameworks Workshop*, page 349–361, 1990.

26 nLab. rig category. `http://ncatlab.org/nlab/show/rig+category`, 2015.

27 Claude Elwood Shannon. A mathematical theory of communication. *Bell Systems Technical Journal*, 27:379–423,623–656, 1948.

28 Paul Taylor. *Practical foundations of mathematics*. Cambridge studies in advanced mathematics. Cambridge University Press, Cambridge, New York (N. Y.), Melbourne, 1999. URL: `http://opac.inria.fr/record=b1095522`.

29 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `http://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

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

31 Michael B. Williams. Introduction to groupoids. Available from `http://www.math.ucla.edu/~mwilliams/pdf/groupoids.pdf`, 2016.