A Sound and Complete Calculus for Reversible Circuit Equivalence

Abstract

Many recent advances in quantum computing, low-power design, nanotechnology, optical information processing, and bioinformatics are based on *reversible circuits*. With the aim of designing a semantically well-founded approach for modeling and reasoning about reversible circuits, we propose viewing such circuits as proof terms witnessing equivalences between finite types. Proving that these type equivalences satisfy the commutative semiring axioms, we proceed with the categorification of type equivalences as *symmetric rig weak groupoids*. The coherence conditions of these categories then produce, for free, a sound and complete calculus for reasoning about reversible circuit equivalence. The paper consists of the "unformalization" of an Agda package formalizing the connections between reversible circuits, equivalences between finite types, permutations between finite sets, and symmetric rig weak groupoids.

1. Introduction

Because physical laws obey various conservation principles (including conservation of information) and because computation is fundamentally a physical process, every computation is, at the physical level, an equivalence that preserves information. The idea that computation, at the logical and programming level, should also be based on "equivalences" (i.e., invertible processes) was originally motivated by such physical considerations (Landauer 1961; Bennett 1973; Toffoli 1980; Feynman 1982; Fredkin and Toffoli 1982; Peres 1985). More recently, the rising importance of energy conservation for both tiny mobile devices and supercomputers, the shrinking size of technology at which quantum effects become noticeable, and the potential for quantum computation and communication, are additional physical considerations adding momentum to such reversible computational models (Frank 1999; DeBenedictis 2005). From a more theoretical perspective, the recently proposed Univalent Foundations Program (2013), based on Homotopy Type Theory (HoTT), greatly emphasizes computation based on *equivalences* that are satisfied up to equivalences that are themselves satisfied up to equivalence, etc.

To summarize, we are witnessing a convergence of ideas from several distinct research communities, including physics, mathematics, and computer science, towards basing computations on equivalences (Baez and Stay 2011). A first step in that direction is the development of many *reversible programming languages* (e.g., (Kluge 2000; Mu et al. 2004; Abramsky 2005; Di Pierro et al. 2006; Yokoyama and Glück 2007; Mackie 2011).) Typically, programs in these languages correspond to some notion of equivalence. But reasoning *about* these programs abandons the notion of equivalence and uses conventional irreversible functions to specify evaluators and the derived notions of program equivalence. This unfortunately misses the beautiful combinatorial structure of programs and proofs that was first exposed in the historical paper by Hofmann and Streicher (1996) and that is currently the center of attention of HoTT and that requires keeping the focus on equivalences not only at the conventional level of programs but also at the higher levels of programs expressing equivalences, programs manipulating equivalences about other programs, etc.

This paper addresses — and completely solves — a well-defined part of the general problem of programming with equivalences up to equivalences. Our approach, we argue, might also be suitable for the more general problem. The particular problem we focus on is that of programming with the finite types built from the empty type, the unit type, and closed under sums and products, and reasoning about these programs between these finite types, i.e., the problem of equivalences between types and equivalences between such equivalences. Although limited in their expressive power, these types are rich enough to express all combinational (with no state or feedback) hardware circuits and, as we show, already exhibit substantial combinatorial structure at the "next level", i.e., at the level of equivalences about equivalences of types. What emerges from our study are the following results:

- a universal language for combinational reversible circuits that comes with a calculus for writing circuits and a calculus for manipulating that calculus;
- the language itself subsumes various representations for reversible circuits, e.g., truth tables, matrices, product of permutation cycles, etc. (Saeedi and Markov 2013);
- the first set of rules is sound and complete with respect to equivalences of types;
- the second set of rules is sound and complete with respect to equivalences of equivalences of types as specified by the first set of rules.

Outline. The next section reviews equivalences between finite types and relates them to various commutative semiring structures. The main message of that section is that, up to equivalence, the concept of equivalence of finite types is equivalent to permutations between finite sets. The latter is computationally well-behaved with existing reversible programming languages developed for programming with permutations and finite-type isomorphisms. This family of languages, called Π, is universal for describing combinational reversible circuits (see Sec. 3). The infrastructure of the HoTT-inspired type equivalences enriches these languages by viewing

their original design as 1-paths and systematically producing 2-paths (equivalences between equivalences) manifesting themselves as syntactic rules for reasoning about equivalences of programs representing reversible circuits. Sec. 4 starts the semantic investigation of the Π languages emphasizing the denotational approach that maps each Π program to a type equivalence or equivalently a permutation. The section also gives a small example showing how a few rules that are sound with respect to equivalence of permutations can be used to transform Π programs without reliance on any extensional reasoning. Sec. 5 then reveals that these rules are intimately related to the coherence conditions of the categorified analogues of the commutative semiring structures underlying type equivalences and permutations, namely, the so-called *symmetric rig weak groupoids*. Sec. 6 contains that "punchline": a sound and complete set of rules that can be used to reason about equivalences of Π programs. Before concluding, we devote Sec. 7 to a detailed analysis of the problem with extending our approach to accommodate higher-order functions, suggesting a possible path towards a solution. We note that because the issues involved are quite subtle, the paper is the "unformalization" of an executable `Agda 2.4.2.3` package with the global `without-K` option enabled.

2. Equivalences and Commutative Semirings

Our starting point is the notion of equivalence of types. We then connect this notion to several semiring structures on finite types, on permutations, and on equivalences, with the goal of reducing the notion of equivalence for finite types to a notion of reversible computation.

2.1 Finite Types

The elementary building blocks of type theory are the empty type ($\bot$), the unit type ($\top$), and the sum ($\uplus$) and product ($\times$) types. These constructors can encode any *finite type*. Traditional type theory also includes several facilities for building infinite types, most notably function types. We will however not address infinite types in this paper except for a discussion in Sec. 7. We will instead focus on thoroughly understanding the computational structures related to finite types.

An essential property of a finite type A is its size $|A|$ which is defined as follows:

$$\begin{array}{rcl} |\bot| & = & 0 \\ |\top| & = & 1 \\ |A \uplus B| & = & |A| + |B| \\ |A \times B| & = & |A| * |B| \end{array}$$

A result by Fiore (2004); Fiore et al. (2006) completely characterizes the isomorphisms between finite types using the axioms of commutative semirings.[1] Intuitively this result states that one can interpret each type by its size, and that this identification validates the familiar properties of the natural numbers, and is in fact isomorphic to the commutative semiring of the natural numbers.

Our work builds on this identification together with work by James and Sabry (2012a) which introduced the Π family of languages whose core computations are these isomorphisms between finite types. Taking into account the growing-in-importance idea that isomorphisms have interesting computational content and should not be silently or implicitly identified, we first recast Fiore et. al's result in the next section, making explicit that the commutative semiring structure can be defined up to the HoTT relation of *type equivalence* instead of strict equality $=$.

[1] Appendix A recalls the definition.

2.2 Commutative Semirings of Types

There are several equivalent definitions of the notion of equivalence of types. For concreteness, we use the following definition.

Definition 1 (Quasi-inverse)**.** *For a function $f : A \to B$, a* quasi-inverse *is a triple (g, α, β), consisting of a function $g : B \to A$ and homotopies $\alpha : f \circ g = \mathrm{id}_B$ and $\beta : g \circ f = \mathrm{id}_A$.*

Definition 2 (Equivalence of types)**.** *Two types A and B are equivalent $A \simeq B$ if there exists a function $f : A \to B$ together with a quasi-inverse for f.*

As the definition of equivalence is parameterized by a function f, we are concerned with, not just the fact that two types are equivalent, but with the precise way in which they are equivalent. For example, there are two equivalences between the type Bool and itself: one that uses the identity for f (and hence for g) and one that uses boolean negation for f (and hence for g). These two equivalences are themselves *not* equivalent: each of them can be used to "transport" properties of Bool in a different way.

It is straightforward to prove that the universe of types (Set in Agda terminology) is a commutative semiring up to equivalence of types $\simeq$.

Theorem 1. *The collection of all types (Set) forms a commutative semiring (up to $\simeq$).*

Proof. As expected, the additive unit is $\bot$, the multiplicative unit is $\top$, and the two binary operations are $\uplus$ and $\times$. □

For example, we have equivalences such as:

$$\begin{array}{rcl} \bot \uplus A & \simeq & A \\ \top \times A & \simeq & A \\ A \times (B \times C) & \simeq & (A \times B) \times C \\ A \times \bot & \simeq & \bot \\ A \times (B \uplus C) & \simeq & (A \times B) \uplus (A \times C) \end{array}$$

One of the advantages of using equivalence $\simeq$ instead of strict equality $=$ is that we can reason one level up about the type of all equivalences $\mathrm{EQ}_{A,B}$. For a given A and B, the elements of $\mathrm{EQ}_{A,B}$ are all the ways in which we can prove $A \simeq B$. For example, $\mathrm{EQ}_{\mathsf{Bool},\mathsf{Bool}}$ has two elements corresponding to the id-equivalence and to the negation-equivalence that were mentioned before. More generally, for finite types A and B, the type $\mathrm{EQ}_{A,B}$ is only inhabited if A and B have the same size in which case the type has $|A|\,!$ (factorial of the size of A) elements witnessing the various possible identifications of A and B. The type of all equivalences has some non-trivial structure: in particular, it is itself a commutative semiring.

Theorem 2. *The type of all equivalences $\mathrm{EQ}_{A,B}$ for finite types A and B forms a commutative semiring up to extensional equivalence of equivalences.*

Proof. The most important insight is the definition of equivalence of equivalences. Two equivalences $e_1, e_2 : \mathrm{EQ}_{A,B}$ with underlying functions f_1 and f_2 and underlying quasi-inverses g_1 and g_2 are themselves equivalent if we have that both $f_1 = f_2$ and $g_1 = g_2$ extensionally. Given this notion of equivalence of equivalences, the proof proceeds smoothly with the additive unit being the vacuous equivalence $\bot \simeq \bot$, the multiplicative unit being the trivial equivalence $\top \simeq \top$, and the two binary operations being essentially a mapping of $\uplus$ and $\times$ over equivalences. □

We reiterate that the commutative semiring axioms in this case are satisfied up to extensional equality of the functions underlying the equivalences. We could, in principle, consider a weaker notion

of equivalence of equivalences and attempt to iterate the construction but for the purposes of modeling circuits and optimizations, it is sufficient to consider just one additional level.

2.3 Commutative Semirings of Permutations

Type equivalences are fundamentally based on function extensionality. (Def. 1 explicitly compares functions for extensional equality.) It is folklore that, even when restricted to finite types, function extensionality needs to be assumed for effective reasoning about type equivalences. The situation gets worse when considering equivalences of equivalences. In the HoTT context, this is the open problem of finding a computational interpretation for *univalence*. In the case of finite types however, there is a computationally-friendly alternative characterization of type equivalences based on permutations of finite sets, which we prove to be formally equivalent.

The idea is that, *up to equivalence*, the only interesting property of a finite type is its size, so that type equivalences must be size-preserving maps and hence correspond to permutations. For example, given two equivalent types A and B of completely different structure, e.g., $A = (\top \uplus \top) \times (\top \uplus (\top \uplus \top))$ and $B = \top \uplus (\top \uplus (\top \uplus (\top \uplus (\top \uplus (\top \uplus \bot)))))$, we can find equivalences from either type to the finite set $\mathsf{Fin}\ 6$ and reduce all type equivalences between sets of size 6 to permutations.

We begin with the following theorem which precisely characterizes the relationship between finite types and finite sets.

Theorem 3. *If $A \simeq \mathsf{Fin}\ m$, $B \simeq \mathsf{Fin}\ n$ and $A \simeq B$ then $m = n$.*

Proof. We proceed by cases on the possible values for m and n. If they are different, we quickly get a contradiction. If they are both 0 we are done. The interesting situation is when $m = suc\ m'$ and $n = suc\ n'$. The result follows in this case by induction assuming we can establish that the equivalence between A and B, i.e., the equivalence between $\mathsf{Fin}\ (suc\ m')$ and $\mathsf{Fin}\ (suc\ n')$, implies an equivalence between $\mathsf{Fin}\ m'$ and $\mathsf{Fin}\ n'$. In a constructive setting, we actually need to construct a particular equivalence between the smaller sets given the equivalence of the larger sets with one additional element. This lemma is quite tedious as it requires us to isolate one element of $\mathsf{Fin}\ (suc\ m')$ and analyze every class of positions this element could be mapped to by the larger equivalence and in each case construct an equivalence that excludes it. □

Given the correspondence between finite types and finite sets, we now prove that equivalences on finite types are equivalent to permutations on finite sets. We proceed in steps: first by proving that finite sets for a commutative semiring up to $\simeq$ (Thm. 4); second by proving that, at the next level, the type of permutations between finite sets is also a commutative semiring up to strict equality of the representations of permutations (Thm. 5); third by proving that the type of type equivalences is equivalent to the type of permutations (Thm. 6); and finally by proving that the commutative semiring of type equivalences is isomorphic to the commutative semiring of permutations (Thm. 7). This series of theorems will therefore justify our focus in the next section of develop a term language for permutations as a way to compute with type equivalences.

Theorem 4. *The collection of all finite types ($\mathsf{Fin}\ m$ for natural number m) forms a commutative semiring (up to $\simeq$).*

Proof. The additive unit is $\mathsf{Fin}\ 0$ and the multiplicative unit is $\mathsf{Fin}\ 1$. For the two binary operations, the proof crucially relies on the following equivalences:

$$\mathsf{iso\text{-}plus} \quad : \{m\ n : \mathbb{N}\} \to (\mathsf{Fin}\ m \uplus \mathsf{Fin}\ n) \simeq \mathsf{Fin}\ (m + n)$$
$$\mathsf{iso\text{-}times} \quad : \{m\ n : \mathbb{N}\} \to (\mathsf{Fin}\ m \times \mathsf{Fin}\ n) \simeq \mathsf{Fin}\ (m * n)$$

□

Theorem 5. *The collection of all permutations $\mathrm{PERM}_{m,n}$ between finite sets $\mathsf{Fin}\ m$ and $\mathsf{Fin}\ n$ forms a commutative semiring up to strict equality of the representations of the permutations.*

Proof. The proof requires delicate attention to the representation of permutations as straightforward attempts turn out not to capture enough of the properties of permutations.[2] After several attempts, we settled on formalizing a permutation using the conventional one-line notation, e.g., giving a preferred enumeration 1 2 3 of a set with three elements, the one-line notion 2 3 1 denotes the permutation sending 1 to 2, 2 to 3, and 3 to 1. To make sure the sequence of numbers is of the right length and that each number is in the right range, we use Agda vectors $\mathsf{Vec}\ (\mathsf{Fin}\ m)\ n$ (abbreviated $\mathsf{FinVec}\ m\ n$). To ensure that the vector elements have no repetitions (i.e., represent 1-1 functions), we include in the representation of each permutation, an inverse vector $\mathsf{FinVec}\ n\ m$ as well as two proofs asserting that the compositions in both directions produce the identity permutation (which naturally forces m and n to be equal). Given this representation, we can prove that two permutations are equal if the one-line vector representations are strictly equal. The main proof then proceeds using the vacuous permutation $\mathsf{CPerm}\ 0\ 0$ for the additive unit and the trivial permutation $\mathsf{CPerm}\ 1\ 1$ for the multiplicative unit. The binary operations on permutations map $\mathsf{CPerm}\ m_1\ m_2$ and $\mathsf{CPerm}\ n_1\ n_2$ to $\mathsf{CPerm}\ (m_1 + n_1)\ (m_2 + n_2)$ and $\mathsf{CPerm}\ (m_1 * n_1)\ (m_2 * n_2)$ respectively. Their definition relies on the properties that the unions and products of the one-line vectors denoting permutations distribute over the sequential compositions of permutations. □

Theorem 6. *If $A \simeq \mathsf{Fin}\ m$ and $B \simeq \mathsf{Fin}\ n$, then the type of all equivalences $\mathrm{EQ}_{A,B}$ is equivalent to the type of all permutations $\mathrm{PERM}\ m\ n$.*

Proof. The main difficulty in this proof was to generalize from sets to setoids to make the equivalence relations explicit. The proof is straightforward but long and tedious. □

Theorem 7. *The equivalence of Theorem 6 is an* isomorphism *between the commutative semiring of equivalences of finite types and the commutative semiring of permutations.*

Proof. In the process of this proof, we show that every axiom of semirings of types is an equivalence and a permutation. Some of the axioms like the associativity of sums gets mapped to the trivial identity permutation. However, some axioms reveal interesting structure when expressed as permutations; the most notable is that the commutativity of products maps to a permutation solving the classical problem of in-place matrix transposition:

$$\mathsf{swap{\star}cauchy} : (m\ n : \mathbb{N}) \to \mathsf{FinVec}\ (n * m)\ (m * n)$$

□

Before concluding, we briefly mention that, with the proper Agda definitions, Thm. 6 can be rephrased in a more evocative way as follows.

Theorem 8.

$$(A \simeq B) \simeq \mathsf{Perm}|A||B|$$

This formulation shows that the univalence *postulate* can be proved and given a computational interpretation for finite types.

[2] All formalizations of permutations in Agda or Coq known to us do not support the full range of operations that we need including sequential compositions, disjoint unions, and products of permutations.

3. Programming with Permutations

In the previous section, we argued that, up to equivalence, the equivalence of types reduces to permutations on finite sets. We recall background work which proposed a term language for permutations and adapt it in later sections to be used to express, compute with, and reason about type equivalences between finite types.

3.1 The Π-Languages

Bowman et al. (2011); James and Sabry (2012a) introduced the Π family of languages whose only computations are permutations (isomorphisms) between finite types and which is complete for all reversible combinational circuits. We propose that this family of languages is exactly the right programmatic interface for manipulating and reasoning about type equivalences.

The syntax of the previously-developed Π language consists of types τ including the empty type 0, the unit type 1, and conventional sum and product types. The values classified by these types are the conventional ones: $()$ of type 1, $\mathsf{inl}\ v$ and $\mathsf{inr}\ v$ for injections into sum types, and (v_1, v_2) for product types:

(*Types*)	τ	::=	$0 \mid 1 \mid \tau_1 + \tau_2 \mid \tau_1 * \tau_2$
(*Values*)	v	::=	$() \mid \mathsf{inl}\ v \mid \mathsf{inr}\ v \mid (v_1, v_2)$
(*Combinator types*)			$\tau_1 \leftrightarrow \tau_2$
(*Combinators*)	c	::=	[*see Fig. 1*]

The interesting syntactic category of Π is that of *combinators* which are witnesses for type isomorphisms $\tau_1 \leftrightarrow \tau_2$. They consist of base combinators (on the left side of Fig. 1) and compositions (on the right side of the same figure). Each line of the figure on the left introduces a pair of dual constants[3] that witness the type isomorphism in the middle. Every combinator c has an inverse $!c$ according to the figure. The inverse is homomorphic on sums and products and flips the order of the combinators in sequential composition.

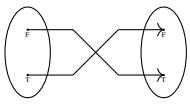
3.2 Example Circuits

The language Π is universal for reversible combinational circuits (James and Sabry 2012a).[4] We illustrate the expressiveness of the language with a few short examples.

The first example is simply boolean encoding and negation which can be defined as shown on the left and visualized as a permutation on finite sets on the right:

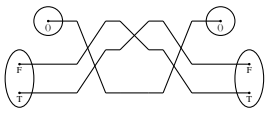
```
BOOL : U
BOOL = PLUS ONE ONE

NOT₁ : BOOL ↔ BOOL
NOT₁ = swap₊
```



Naturally there are many ways of encoding boolean negation. The following combinator implements a more convoluted circuit that computes the same function, which is also visualized as a permutation on finite sets:

```
NOT₂ : BOOL ↔ BOOL
NOT₂ =
  uniti⋆ ⊙
  swap⋆ ⊙
  (swap₊ ⊗ id↔) ⊙
  swap⋆ ⊙
  unite⋆
```



[3] where $swap_+$ and $swap_*$ are self-dual.

[4] With the addition of recursive types and trace operators (Hasegawa 1997), Π become a Turing complete reversible language (James and Sabry 2012a; Bowman et al. 2011).

Writing circuits using the raw syntax for combinators is clearly tedious. In other work, one can find a compiler from a conventional functional language to generate the circuits (James and Sabry 2012a), a systematic technique to translate abstract machines to Π (James and Sabry 2012b), and a Haskell-like surface language (James and Sabry 2014) which can be of help in writing circuits. These essential tools are however a distraction in the current setting and we content ourselves with some Agda syntactic sugar illustrated below and used again in the next section:

```
BOOL² : U
BOOL² = TIMES BOOL BOOL

CNOT : BOOL² ↔ BOOL²
CNOT = TIMES BOOL BOOL
       ↔⟨ id↔ ⟩
   TIMES (PLUS x y) BOOL
       ↔⟨ dist ⟩
   PLUS (TIMES x BOOL) (TIMES y BOOL)
       ↔⟨ id↔ ⊕ (id↔ ⊗ NOT₁) ⟩
   PLUS (TIMES x BOOL) (TIMES y BOOL)
       ↔⟨ factor ⟩
   TIMES (PLUS x y) BOOL
       ↔⟨ id↔ ⟩
   TIMES BOOL BOOL □
   where x = ONE; y = ONE

TOFFOLI : TIMES BOOL BOOL² ↔ TIMES BOOL BOOL²
TOFFOLI = TIMES BOOL BOOL²
       ↔⟨ id↔ ⟩
   TIMES (PLUS x y) BOOL²
       ↔⟨ dist ⟩
   PLUS (TIMES x BOOL²) (TIMES y BOOL²)
       ↔⟨ id↔ ⊕ (id↔ ⊗ CNOT) ⟩
   PLUS (TIMES x BOOL²) (TIMES y BOOL²)
       ↔⟨ factor ⟩
   TIMES (PLUS x y) BOOL²
       ↔⟨ id↔ ⟩
      TIMES BOOL BOOL² □
   where x = ONE; y = ONE
```

This style makes the intermediate steps explicit showing how the types are transformed in each step by the combinators. The example confirms that Π is universal for reversible circuits since the Toffoli gate is universal for such circuits (Toffoli 1980).

4. Semantics

In the previous sections, we established that type equivalences on finite types can be, up to equivalence, expressed as permutations and proposed a term language for expressing permutations on finite types that is complete for reversible combinational circuits. We are now ready for the main technical contribution of the paper: an effective computational framework for reasoning *about* type equivalences. From a programming perspective, this framework manifests itself as a collection of rewrite rules for optimizing circuit descriptions in Π. Naturally we are not concerned with just any collection of rewrite rules but with a sound and complete collection. The current section will set up the framework and illustrate its use on one example and the next sections will introduce the categorical framework in which soundness and completeness can be proved.

4.1 Operational and Denotational Semantics

In conventional programming language research, valid optimizations are specified with reference to the *observational equivalence*

$$
\begin{array}{rrcll}
unite_+ : & 0 + \tau & \leftrightarrow & \tau & : uniti_+ \\
swap_+ : & \tau_1 + \tau_2 & \leftrightarrow & \tau_2 + \tau_1 & : swap_+ \\
assocl_+ : & \tau_1 + (\tau_2 + \tau_3) & \leftrightarrow & (\tau_1 + \tau_2) + \tau_3 & : assocr_+ \\
\\
unite_* : & 1 * \tau & \leftrightarrow & \tau & : uniti_* \\
swap_* : & \tau_1 * \tau_2 & \leftrightarrow & \tau_2 * \tau_1 & : swap_* \\
assocl_* : & \tau_1 * (\tau_2 * \tau_3) & \leftrightarrow & (\tau_1 * \tau_2) * \tau_3 & : assocr_* \\
\\
dist_0 : & 0 * \tau & \leftrightarrow & 0 & : factorl_0 \\
dist : & (\tau_1 + \tau_2) * \tau_3 & \leftrightarrow & (\tau_1 * \tau_3) + (\tau_2 * \tau_3) & : factor
\end{array}
$$

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

$$
\frac{\vdash c_1 : \tau_1 \leftrightarrow \tau_2 \quad \vdash c_2 : \tau_3 \leftrightarrow \tau_4}{\vdash c_1 \oplus c_2 : \tau_1 + \tau_3 \leftrightarrow \tau_2 + \tau_4}
$$

$$
\frac{\vdash c_1 : \tau_1 \leftrightarrow \tau_2 \quad \vdash c_2 : \tau_3 \leftrightarrow \tau_4}{\vdash c_1 \otimes c_2 : \tau_1 * \tau_3 \leftrightarrow \tau_2 * \tau_4}
$$

Figure 1. Π-combinators (Bowman et al. 2011; James and Sabry 2012a).

relation which itself is defined with reference to an *evaluator*. As the language is reversible, a reasonable starting point would then be to define forward and backward evaluators with the following signatures:

```
eval  : {t1 t2 : U} → (t1 ↔ t2) → ⟦ t1 ⟧ → ⟦ t2 ⟧
evalB : {t1 t2 : U} → (t1 ↔ t2) → ⟦ t2 ⟧ → ⟦ t1 ⟧
```

In the definition, the function $[\![\cdot]\!]$ maps each type constructor to its Agda denotation, e.g., it maps the type 0 to $\bot$, the type 1 to $\top$, etc. The complete definitions for these evaluators can be found in the papers by Bowman et al. (2011); James and Sabry (2012b,a) and in the accompanying Agda code and will not be repeated here. The reason is that, although these evaluators adequately serve as semantic specifications, they drive the development towards extensional reasoning as evident from the signatures which map a permutation to a function. We will instead pursue a denotational approach mapping the combinators to type equivalences or equivalently to permutations:

```
c2equiv : {t1 t2 : U} → (c : t1 ↔ t2) → ⟦ t1 ⟧ ≃ ⟦ t2 ⟧
c2perm  : {t1 t2 : U} → (c : t1 ↔ t2) →
          CPerm (size t2) (size t1)
```

The advantage is that permutations have a concrete representation which can be effectively compared for equality as explained in the proof of Thm. 5.

4.2 Rewriting Approach

Having mapped each combinator to a permutation, we can reason about valid optimizations mapping a combinator to another by studying the equivalence of permutations on finite sets. Strict equality of permutations would distinguish the permutations corresponding to c and $\mathsf{id}\leftrightarrow \odot c$ for a combinator c and hence is inappropriate for reasoning about optimizations and equivalences of circuits. Of course, we could easily justify this particular equivalence, and many others, by calculating the actions of the two permutations on arbitrary incoming sets and checking that the results are identical; this extensional reasoning is however *not* our stated goal. We need instead, proof *terms* witnessing *all* possible equivalences on permutations.

Before we embark on the categorification program that will allow us to find this complete collection of rules in the next section, we show that, with some ingenuity, one can develop a reasonable set of rewrite rules that is rich enough to prove that the two negation circuits from the previous section are actually equivalent:

```
negEx : NOT₂ ⇔ NOT₁
negEx =
  uniti⋆ ⊙ (swap⋆ ⊙ ((swap₊ ⊗ id↔) ⊙ (swap⋆ ⊙ unite⋆)))
    ⇔⟨ id⇔ ⊡ assoc⊙l ⟩
  uniti⋆ ⊙ ((swap⋆ ⊙ (swap₊ ⊗ id↔)) ⊙ (swap⋆ ⊙ unite⋆))
    ⇔⟨ id⇔ ⊡ (swapl⋆⇔ ⊡ id⇔) ⟩
  uniti⋆ ⊙ (((id↔ ⊗ swap₊) ⊙ swap⋆) ⊙ (swap⋆ ⊙ unite⋆))
    ⇔⟨ id⇔ ⊡ assoc⊙r ⟩
  uniti⋆ ⊙ ((id↔ ⊗ swap₊) ⊙ (swap⋆ ⊙ (swap⋆ ⊙ unite⋆)))
    ⇔⟨ id⇔ ⊡ (id⇔ ⊡ assoc⊙l) ⟩
  uniti⋆ ⊙ ((id↔ ⊗ swap₊) ⊙ ((swap⋆ ⊙ swap⋆) ⊙ unite⋆))
    ⇔⟨ id⇔ ⊡ (id⇔ ⊡ (linv⊙l ⊡ id⇔)) ⟩
  uniti⋆ ⊙ ((id↔ ⊗ swap₊) ⊙ (id↔ ⊙ unite⋆))
    ⇔⟨ id⇔ ⊡ (id⇔ ⊡ idl⊙l) ⟩
  uniti⋆ ⊙ ((id↔ ⊗ swap₊) ⊙ unite⋆)
    ⇔⟨ assoc⊙l ⟩
  (uniti⋆ ⊙ (id↔ ⊗ swap₊)) ⊙ unite⋆
    ⇔⟨ unitil⋆⇔ ⊡ id⇔ ⟩
  (swap₊ ⊙ uniti⋆) ⊙ unite⋆
    ⇔⟨ assoc⊙r ⟩
  swap₊ ⊙ (uniti⋆ ⊙ unite⋆)
    ⇔⟨ id⇔ ⊡ linv⊙l ⟩
  swap₊ ⊙ id↔
    ⇔⟨ idr⊙l ⟩
  swap₊ □
```

The rules used in the derivation (and which are defined in the Agda code) capture the following properties about permutations: the sequential composition of permutations is associative, the identity permutation is a left and right unit of sequential composition, the composition of a permutation with its inverse produces the identity permutation, permutations on the set with one element are trivial and can be omitted, and swapping and parallel composition commute as follows:

$$\mathsf{swap}\star \odot (c_1 \otimes c_2) \Leftrightarrow (c_2 \otimes c_1) \odot \mathsf{swap}\star$$

The accompanying material includes a short slide deck animating the sequence of rewrites showing that they are all indeed intuitive transformations on the diagrams representing the circuits.

5. Categorification

The problem of finding a sound and complete set of rules for reasoning about equivalence of permutations is solved by appealing to various results about specialized monoidal categories (Selinger 2011). The main technical vehicle is that of *categorification* (Baez and Dolan 1998) which is a process, intimately related to homotopy theory, for finding category-theoretic analogs of set-theoretic concepts. From an intuitive perspective, the algebraic structure of a commutative semiring only captures a "static" relationship between types; it says nothing about how these relationships behave under *composition* which is after all the essence of computation (cf. Moggi (1989)'s original paper on monads). Thus, from a programmer's perspective, this categorification process is about understanding how type equivalences evolve under compositions, e.g., how two different paths of type equivalences sharing the same source and target relate two each other.

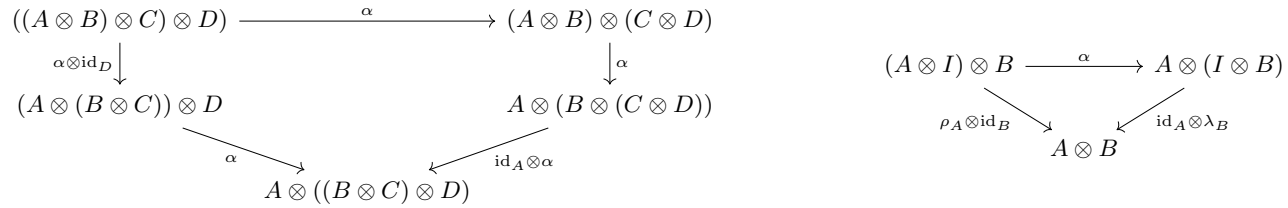


Figure 2. Pengaton and triangle diagrams.

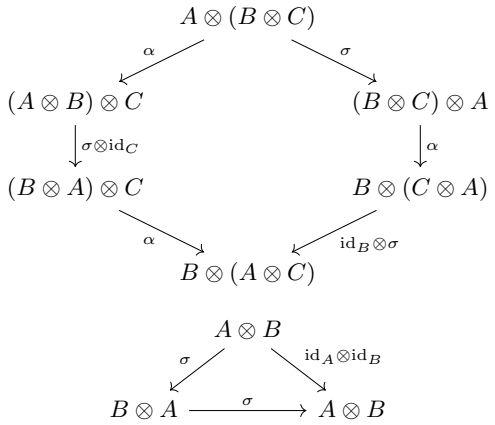
5.1 Monoidal Categories

We begin with the conventional definitions for monoidal and symmetric monoidal categories.

Definition 3 (Monoidal Category)**.** *A* monoidal category *(Mac Lane 1971) is a category with the following additional structure:*

- *a bifunctor* $\otimes$ *called the monoidal or tensor product,*
- *an object* I *called the unit object, and*
- *natural isomorphisms* $\alpha_{A,B,C} : (A \otimes B) \otimes C \xrightarrow{\sim} A \otimes (B \otimes C)$, $\lambda_A : I \otimes A \xrightarrow{\sim} A$, *and* $\rho_A : A \otimes I \xrightarrow{\sim} A$, *such that the two diagrams (known as the* associativity pentagon *and the* triangle for unit*) in Fig. 2 commute.*

Definition 4 (Symmetric Monoidal Category)**.** *A monoidal category is* symmetric *if it has an isomorphism* $\sigma_{A,B} : A \otimes B \xrightarrow{\sim} B \otimes A$ *where* σ *is a natural transformation which satisfies the following two coherence conditions (called* bilinearity *and* symmetry*):*



According to Mac Lane's coherence theorem, the coherence laws for monoidal categories are justified by the desire to equate any two isomorphisms built using σ, λ, and ρ and having the same source and target. Similar considerations justify the coherence laws for symmetric monoidal categories. It is important to note that the coherence conditions do *not* imply that every pair of parallel morphisms with the same source and target are equal. Indeed, as Dosen and Petric explain:

> In Mac Lane's second coherence result of [...], which has to do with symmetric monoidal categories, it is not intended that all equations between arrows of the same type should hold. What Mac Lane does can be described in logical terms in the following manner. On the one hand, he has an axiomatization, and, on the other hand, he has a model category where arrows are permutations; then he shows that his axiomatization is complete with respect to this model. It is no wonder that his coherence problem reduces to the completeness problem for the usual axiomatization of symmetric groups (Dosen and Petric 2004).

The informal idea was already silently used in the examples in Sec. 3.2 in which the types were named x and y during the derivation to distinguish operations that would otherwise be ambiguous if all the types were instantiated with ONE.

From a different perspective, Baez and Dolan (1998) explain the source of these coherence laws as arising from homotopy theory. In this theory, laws are only imposed up to homotopy with these homotopies satisfying certain laws, up again only up to homotopy, with these higher homotopies satisfying their own higher coherence laws, and so on. Remarkably, they report, among other results, that the pentagon identity of Fig. 2 arises when studying the algebraic structure possessed by a space that is homotopy equivalent to a loop space and that the hexagon identity arises in the context of spaces homotopy equivalent to double loop spaces.

In our context, we will build monoidal categories where the objects are finite types and the morphisms are reversible circuits represented as Π-combinators. Clearly not all reversible circuits mapping Bool to Bool are equal. There are at least two distinct such circuits: the identity and boolean negation. The coherence laws should not equate these two morphisms and they do not. We might also hope that the two versions of boolean negation in Sec. 3.2 and Sec. 4.2 could be identified using the coherence conditions of monoidal categories. This is not the case but will be once we capture the full structure of commutative semirings categorically.

5.2 Symmetric Rig Weak Groupoids

Symmetric monoidal categories discussed in the previous section are the categorifications of commutative monoids. The categorification of a commutative semiring is called a *symmetric rig category*. It is build from a *symmetric bimonoidal category* to which distributivity natural isomorphisms are added, and accompanying coherence laws added. Since we can easily set things up so that every morphism is a isomorphism, the category will also be a groupoid. Since the laws of the category only hold up to a higher equivalence, the entire setting is that of weak categories.

There are several equivalent definitions of rig categories. We use the following definition from nLab.

Definition 5 (Rig Category)**.** *A* rig category C *is a category with a symmetric monoidal structure* $(C, \oplus, 0)$ *for addition and a monoidal structure* $(C, \otimes, 1)$ *for multiplication together with left and right distributivity natural isomorphisms:*

$$\begin{aligned} d_\ell : x \otimes (y \oplus z) &\xrightarrow{\sim} (x \otimes y) \oplus (x \otimes z) \\ d_r : (x \oplus y) \otimes z &\xrightarrow{\sim} (x \otimes z) \oplus (y \otimes z) \end{aligned}$$

and absorption/annihilation isomorphisms $a_\ell : x \otimes 0 \xrightarrow{\sim} 0$ *and* $a_r : 0 \otimes x \xrightarrow{\sim} 0$ *satisfying coherence conditions worked out by Laplaza (1972) and discussed below.*

Definition 6 (Symmetric Rig Category)**.** *A* symmetric rig category *is a rig category in which the multiplicative structure is symmetric.*

Definition 7 (Symmetric Rig Groupoid)**.** *A* symmetric rig groupoid *is a symmetric rig category in which every morphism is invertible.*

$$
\begin{array}{rrcll}
unites_+ : & \tau + 0 & \leftrightarrow & \tau & : unitis_+ \\
unites_* : & \tau * 1 & \leftrightarrow & \tau & : unitis_* \\
absorbr_0 : & 0 * \tau & \leftrightarrow & 0 & : factorl_0 \\
absorbl_0 : & \tau * 0 & \leftrightarrow & 0 & : factorr_0 \\
distl : & \tau_1 * (\tau_2 + \tau_3) & \leftrightarrow & (\tau_1 * \tau_2) + (\tau_1 * \tau_3) & : factorl
\end{array}
$$

Figure 3. Additional Π-combinators.

The coherence conditions for rig categories were worked out by Laplaza (1972). Pages 31-35 of his paper report 24 coherence conditions that vary from simple diagrams to one that includes 9 nodes showing that two distinct ways of simplifying $(A \oplus B) \otimes (C \oplus D)$ to $(((A \otimes C) \oplus (B \otimes C)) \oplus (A \otimes D)) \oplus (B \otimes D)$ commute. The 24 coherence conditions are not independent which somewhat simplifies the situation and allows us to prove that our structures satisfy the coherence conditions in a more economical way.

5.3 Instances of Symmetric Rig Categories

Most of the structures we have discussed so far are instances of symmetric rig weak groupoids.

Theorem 9. *The collection of all types and type equivalences is a symmetric rig groupoid.*

Proof. The objects of the category are Agda types and the morphisms are type equivalences. These morphisms directly satisfy the axioms stated in the definitions of the various categories. The bulk of the work is in ensuring that the coherence conditions are satisfied up to extensional equality. We only show the proof of one coherence condition, the first one in Laplaza's paper shown below:

$$
\begin{array}{ccc}
A \otimes (B \oplus C) & \xrightarrow{distl} & (A \otimes B) \oplus (A \otimes C) \\
{\scriptstyle \mathrm{id}_A \otimes swap_+}\downarrow & & \downarrow{\scriptstyle swap_+} \\
A \otimes (C \oplus B) & \xrightarrow[distl]{} & (A \otimes C) \oplus (A \otimes B)
\end{array}
$$

We first have a lemma that shows that the two paths starting from the top left node are equivalent:

```
distl-swap₊-lemma : {A B C : Set} → (x : (A × (B ⊎ C))) →
  TE.distl (map× F.id TE.swap₊ x) P.≡
  (TE.swap₊ (distl x))
distl-swap₊-lemma (x , inj₁ y) = P.refl
distl-swap₊-lemma (x , inj₂ y) = P.refl
```

The lemma asserts the extensional equivalence of the functions representing the two paths for all arguments x. This lemma is sufficient to prove we have a rig *category*. To prove we also have a groupoid, we need a converse lemma starting from the bottom right node and following the paths backwards towards the top left node:

```
factorl-swap₊-lemma : {A B C : Set} →
  (x : (A × C) ⊎ (A × B)) →
  map× F.id TE.swap₊ (TE.factorl x) P.≡
  TE.factorl (TE.swap₊ x)
factorl-swap₊-lemma (inj₁ x) = P.refl
factorl-swap₊-lemma (inj₂ y) = P.refl
```

Finally we show that the forward equivalence and the backward equivalence are themselves equivalent:

```
laplazaI = eq distl-swap₊-lemma factorl-swap₊-lemma
```

where eq is the constructor for the equivalence of equivalences used in the proof of Thm. 2. □

More directly relevant to our purposes, is the next theorem which applies to reversible circuits (represented as Π-combinators).

Theorem 10. *The collection of finite types and* Π*-combinators is a symmetric rig groupoid.*

Proof. The objects of the category are finite types and the morphisms are the Π-combinators. Short proofs establish that these morphisms satisfy the axioms stated in the definitions of the various categories. The bulk of the work is in ensuring that the coherence conditions are satisfied. This required us to add a few Π combinators (see Fig. 3) and then to add a whole new layer of 2-combinators witnessing enough equivalences of Π combinators to prove the coherence laws (see Fig. 4). The new Π combinators, also discussed in more detail in the next section, are redundant (from an operational perspective) exactly because of the coherence conditions; they are however important as they have rather non-trivial relations to each other that are captured in the more involved coherence laws. □

Putting the result above together with Laplaza's coherence result about rig categories, we conclude with our main result, which will be detailed in the next section by giving the full details of the second level of combinators.

Theorem 11. *We have two levels of* Π*-combinators such that:*

- *The first level of* Π*-combinators is complete for representing reversible combinational circuits.*
- *The second level of* Π*-combinators is sound and complete for the equivalence of circuits represented by the first level.*

6. Revised Π and its Optimizer

Collecting the previous results we arrive at a universal language for expressing reversible combinational circuits *together with* a sound and complete metalanguage for reasoning about equivalences of programs written in the lower level language.

6.1 Example

Given two Π-combinators:

```
c₁ : {B C : U} →  B ↔ C
c₂ : {A D : U} →  A ↔ D
```

we can build two larger combinators p_1 and p_2,

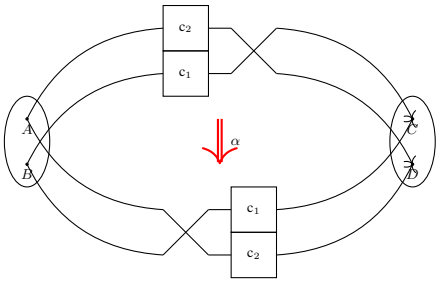
```
p₁ p₂ : {A B C D : U} → PLUS A B ↔ PLUS C D
p₁ = _↔_.swap₊ ⊙ (c₁ ⊕ c₂)
p₂ = (c₂ ⊕ c₁) ⊙ _↔_.swap₊
```

As reversible circuits, p_1 and p_2 evaluate as follows. If p_1 is given the value inl a, it first transforms it to inr a, and then passes it to c_2. If p_2 is given the value inl a, it first passes it to c_2 and then flips the tag of the result. Since c_2 is functorial, it must act polymorphically on its input and hence, it must be the case that the two evaluations produce the same result. The situation for the other possible input value is symmetric. This extensional reasoning is embedded once and for all in the proofs of coherence and distilled in a 2-level combinator:

[it would be nice to remove the leading ↔ qualifier as it is an artifact how all the pieces were put together rather than something essential. —JC]

```
swapl₊⇔ : {t₁ t₂ t₃ t₄ : U} {c₁ : t₁ ↔ t₂} {c₂ : t₃ ↔
  (_↔_.swap₊ ⊙ (c₁ ⊕ c₂)) ⇔' ((c₂ ⊕ c₁) ⊙ _↔_.swap₊)
```

Pictorially, this 2-level combinator is a 2-path showing how the two paths can be transformed to one another. The proof of equivalence can be visualized by simply imagining the connections as wires whose endpoints are fixed: holding the wires on the right side of the top path and flipping them produces the connection in the bottom path:



Categorically speaking, this combinator expresses exactly that the braiding $\sigma_{A,B}$ is a natural transformation, in other words that $\sigma_{A,B}$ must commute with $\oplus$.

6.2 Revised Syntax

The inspiration of symmetric rig groupoids suggested a refactoring of Π with additional level-1 combinators. The added combinators 3 are redundant (from an operational perspective) exactly because of the coherence conditions. In addition to being critical to the proofs, they are useful when representing circuits, leading to smaller programs with fewer redexes.

The big addition of course is the level-2 combinators which are collected in Fig. 4. To avoid clutter we omit the names of the combinators and only show the signatures.

Even though we know that these combinators come from the coherence conditions inherence in the definition of a symmetric rig weak groupoid, it would still be nice to get a better understanding of what these really say. About a third of the combinators come from the definition of the various natural isomorphisms ($\alpha_{A,B,C}, \lambda_A, \rho_A, \sigma_{A,B}, d_l, d_r, a_l$ and a_r). The first 4 natural isomorphisms actually occur twice, once for each of the symmetric monoidal structures at play. Each natural isomorphism is composed of 2 natural transformations (one in each direction) that must compose to the identity. This in turn induces 4 coherences laws: two *naturality laws* which indicate that the combinator commutes with structure construction, and two which express that the resulting combinators are left and right inverses of each other. But note also that there mere desire that $\oplus$ be a bifunctor induces 3 coherence laws. And then of course each "structure" (monoidal, braided, symmetric) comes with more, as outlined in the previous section, culminating with 13 additional coherence laws for rig.

The coherence laws for a symmetric rig category, whether presented in their full diagrammatic glory or through their signatures, may at first appear rather obscure. But these can be "unformalized" to relatively understandable statements:

I given $A \otimes (B \oplus C)$, swapping B and C then distributing (on the left) is the same as first distributing, then swapping the two summands,

II given $(A \oplus B) \otimes C$, first switch the order of the products then distributing (on the left) is the same as distributing (on the right) and then switching the order of both products.

IV given $(A \oplus (B \oplus C)) \otimes D$, we can either distribute then associate, or associate then distribute.

VI given $A \otimes (B \otimes (C \oplus D))$, we can either associate then distribute, or first do the inner distribution, then the outer, and map associativity on each term.

IX given $(A \oplus B) \otimes (C \oplus D)$, we can either first distribute on the left, map right-distribution and finally associate, or we can go "the long way around" by right-distributing first, then mapping left-distribution, and then a long chain of administrative shuffles to get to the same point.

X given $0 \otimes 0$, left or right absorption both give 0 in equivalent ways

XI given $0 \otimes (A \oplus B)$, left absorption or distribution, then mapping left absorption, followed by (additive) left unit are equivalent.

XIII given $0 * 1$, left absorption or (multiplicative) right unit are equivalent.

XV given $A \otimes 0$, we can either absorb 0 on the left, or commute and absorb 0 on the right.

XVI given $0 \otimes (A \otimes B)$, we can either absorb 0 on the left, or associate, and then absorb twice.

XVII given $A \otimes (0 \otimes B)$, the two obvious paths to 0 commute.

XIX given $A \otimes (0 \oplus B)$, we can either eliminate the (additive) identity in the right term, or distribute, right absorb 0 in the left term, then eliminate the resulting (additive) identity to get to $A \otimes B$.

XXIII Given $1 \otimes (A \oplus B)$, we can either eliminate the (multiplicative) identity on the left or distribute the map left-elimination.

Going through the details of the proof of the coherence theorem in Laplaza (1972) with a "modern" eye, one cannot help but think of Knuth-Bendix completion. Although it is known that coherence laws for some categorical structures can be obtained in this way (Beke 2011), it is also known that in the presence of certain structures (such as symmetry), Knuth-Bendix completion will not terminate. It would be interesting to know if there is indeed a systematic way to obtain these laws; the wider mathematical community does not seem to know (Carette 2015).

It is worth noting that most (but not all) of the properties of $\uplus$ were already in Agda's standard library (in Data.Sum.Properties to be precise), whereas all properties of $\times$ were immediately provable due to h expansion. None of the mixed properties involved with distributivity and absorption were present, although the proof of all of them was very straightforward.

As Fig. 4 illustrates, we have rules to manipulate code fragments rewriting them in a small-step fashion. The rules apply only when both sides are well-typed. The small-step nature of the rules should allow us to make efficient optimizers following the experience in functional languages (Peyton Jones and Santos 1998). In contrast the coherence conditions are much smaller in number and many then express invariants about much bigger "chunks." From our small experiments, an effective way to use the rules is to fix a canonical representation of circuits that has the "right" properties and use the rules in a directed fashion to produce that canonical representation. For example, Saeedi and Markov (2013) survey several possible canonical representations that trade-off various desired properties. Of course, finding a rewriting procedure that makes progress towards the canonical representation is far from trivial.

It should be noted that a few of the "raw" signatures in Fig. 4 are slightly misleading, as we omit the signature of the underlying combinators. For example, $unite_* \fatsemi c_2 \Leftrightarrow (c_1 \otimes c_2) \fatsemi unite_*$ hides the fact that c_1 here is restricted to have signature c_1 : ZERO $\leftrightarrow$ ZERO. The reader should consult the code for full details.

Amr says: One of the interesting conclusions of the coherence laws (see the comments in the file above) is that it forces all (putative) elements of bot to be equal. This comes from the coherence law for the two ways of proving that 0 * 0 = 0.

Amr says: swapfl* and swapfr* were never used, so I removed them (commented them out of PiLevel1).I'd lean towards leaving it and saying that the axioms are not independent, just like Laplaza's conditions.

$$
\begin{array}{rcl}
id \fatsemi c & \Leftrightarrow & c \\
c \fatsemi id & \Leftrightarrow & c \\
c \fatsemi (!\, c) & \Leftrightarrow & id \\
(!\, c) \fatsemi c & \Leftrightarrow & id \\
id \oplus id & \Leftrightarrow & id \\
id \otimes id & \Leftrightarrow & id \\
c_1 \fatsemi (c_2 \fatsemi c_3) & \Leftrightarrow & (c_1 \fatsemi c_2) \fatsemi c_3 \\
(c_1 \fatsemi c_3) \oplus (c_2 \fatsemi c_4) & \Leftrightarrow & (c_1 \oplus c_2) \fatsemi (c_3 \oplus c_4) \\
(c_1 \fatsemi c_3) \otimes (c_2 \fatsemi c_4) & \Leftrightarrow & (c_1 \otimes c_2) \fatsemi (c_3 \otimes c_4) \\
\\
swap_+ \fatsemi (c_1 \oplus c_2) & \Leftrightarrow & (c_2 \oplus c_1) \fatsemi swap_+ \\
swap_* \fatsemi (c_1 \otimes c_2) & \Leftrightarrow & (c_2 \otimes c_1) \fatsemi swap_* \\
swap_+ \fatsemi factor & \Leftrightarrow & factor \fatsemi (swap_+ \otimes id) \\
\\
unite_+ \fatsemi c_2 & \Leftrightarrow & (c_1 \oplus c_2) \fatsemi unite_+ \\
uniti_+ \fatsemi (c_1 \oplus c_2) & \Leftrightarrow & c_2 \fatsemi uniti_+ \\
unites_+ \fatsemi c_2 & \Leftrightarrow & (c_2 \oplus c_1) \fatsemi unites_+ \\
unitis_+ \fatsemi (c_2 \oplus c_1) & \Leftrightarrow & c_2 \fatsemi unitis_+ \\
unites_+ \oplus id & \Leftrightarrow & assocr_+ \fatsemi (id \oplus unite_+) \\
\\
(c_1 \oplus (c_2 \oplus c_3)) \fatsemi assocl_+ & \Leftrightarrow & assocl_+ \fatsemi ((c_1 \oplus c_2) \oplus c_3) \\
((c_1 \oplus c_2) \oplus c_3) \fatsemi assocr_+ & \Leftrightarrow & assocr_+ \fatsemi (c_1 \oplus (c_2 \oplus c_3)) \\
(c_1 \otimes (c_2 \otimes c_3)) \fatsemi assocl_* & \Leftrightarrow & assocl_* \fatsemi ((c_1 \otimes c_2) \otimes c_3) \\
((c_1 \otimes c_2) \otimes c_3) \fatsemi assocr_* & \Leftrightarrow & assocr_* \fatsemi (c_1 \otimes (c_2 \otimes c_3)) \\
((a \oplus b) \otimes c) \fatsemi dist & \Leftrightarrow & dist \fatsemi ((a \otimes c) \oplus (b \otimes c)) \\
((a \otimes c) \oplus (b \otimes c)) \fatsemi factor & \Leftrightarrow & factor \fatsemi ((a \oplus b) \otimes c) \\
(a \otimes (b \oplus c)) \fatsemi distl & \Leftrightarrow & distl \fatsemi ((a \otimes b) \oplus (a \otimes c)) \\
((a \otimes b) \oplus (a \otimes c)) \fatsemi factorl & \Leftrightarrow & factorl \fatsemi (a \otimes (b \oplus c))
\end{array}
$$

$$
\begin{array}{rcl}
unite_* \fatsemi c_2 & \Leftrightarrow & (c_1 \otimes c_2) \fatsemi unite_* \\
uniti_* \fatsemi (c_1 \otimes c_2) & \Leftrightarrow & c_2 \fatsemi uniti_* \\
unites_* \fatsemi c_2 & \Leftrightarrow & (c_2 \otimes c_1) \fatsemi unites_* \\
unitis_* \fatsemi (c_2 \otimes c_1) & \Leftrightarrow & c_2 \fatsemi unitis_* \\
unites_* \otimes id & \Leftrightarrow & assocr_* \fatsemi (id \oplus unite_*) \\
\\
absorbr_0 & \Leftrightarrow & unite_* \\
absorbr_0 & \Leftrightarrow & absorbl_0 \\
absorbr_0 & \Leftrightarrow & (assocl_* \fatsemi (absorbr_0 \otimes id)) \\
 & & \fatsemi absorbr_0 \\
absorbr_0 & \Leftrightarrow & (distl \\
 & & \quad \fatsemi (absorbr_0 \oplus absorbr_0)) \\
 & & \fatsemi unite_+ \\
absorbl_0 & \Leftrightarrow & swap_* \fatsemi absorbr_0 \\
(c \otimes id) \fatsemi absorbl_0 & \Leftrightarrow & absorbl_0 \fatsemi id \\
(id \otimes c) \fatsemi absorbr_0 & \Leftrightarrow & absorbr_0 \fatsemi id \\
(id \otimes absorbr_0) \fatsemi absorbl_0 & \Leftrightarrow & (assocl_* \fatsemi (absorbl_0 \otimes id) \\
 & & \fatsemi absorbr_0 \\
(id \otimes unite_+) & \Leftrightarrow & (distl \fatsemi (absorbl_0 \oplus id)) \\
 & & \fatsemi unite_+ \\
\\
unite_+ & \Leftrightarrow & distl \fatsemi (unite_+ \oplus unite_+) \\
(id \otimes swap_+) \fatsemi distl & \Leftrightarrow & distl \fatsemi swap_+ \\
dist \fatsemi (swap_* \oplus swap_*) & \Leftrightarrow & swap_* \fatsemi distl \\
id \fatsemi factorl_0 & \Leftrightarrow & factorl_0 \fatsemi (id \otimes c) \\
id \fatsemi factorr_0 & \Leftrightarrow & factorr_0 \fatsemi (c \otimes id)
\end{array}
$$

$$
\begin{array}{rcl}
((assocl_+ \otimes id) \fatsemi dist) \fatsemi (dist \oplus id) & \Leftrightarrow & (dist \fatsemi (id \oplus dist)) \fatsemi assocl_+ \\
(distl \fatsemi (dist \oplus dist)) \fatsemi assocl_+ & \Leftrightarrow & ((((dist \fatsemi (distl \oplus distl)) \fatsemi assocl_+) \fatsemi (assocr_+ \oplus id)) \\
 & & \fatsemi (id \oplus swap_+) \oplus id) \fatsemi (assocl_+ \oplus id) \\
assocl_* \fatsemi distl & \Leftrightarrow & ((id \otimes distl) \fatsemi distl) \fatsemi (assocl_* \oplus assocl_*) \\
assocr_+ \fatsemi assocr_+ & \Leftrightarrow & ((assocr_+ \oplus id) \fatsemi assocr_+) \fatsemi (id \oplus assocr_+) \\
assocr_* \fatsemi assocr_* & \Leftrightarrow & ((assocr_* \otimes id) \fatsemi assocr_*) \fatsemi (id \otimes assocr_*) \\
(assocr_+ \fatsemi swap_+) \fatsemi assocr_+ & \Leftrightarrow & ((swap_+ \oplus id) \fatsemi assocr_+) \fatsemi (id \oplus swap_+) \\
(assocl_+ \fatsemi swap_+) \fatsemi assocl_+ & \Leftrightarrow & ((id \oplus swap_+) \fatsemi assocl_+) \fatsemi (swap_+ \oplus id) \\
(assocr_* \fatsemi swap_*) \fatsemi assocr_* & \Leftrightarrow & ((swap_* \oplus id) \fatsemi assocr_*) \fatsemi (id \otimes swap_*) \\
(assocl_* \fatsemi swap_*) \fatsemi assocl_* & \Leftrightarrow & ((id \otimes swap_*) \fatsemi assocl_*) \fatsemi (swap_* \otimes id)
\end{array}
$$

$$
\frac{}{\vdash c \Leftrightarrow c} \qquad
\frac{\vdash c_1 \Leftrightarrow c_2 \quad \vdash c_2 \Leftrightarrow c_3}{\vdash c_1 \Leftrightarrow c_3} \qquad
\frac{\vdash c_1 \Leftrightarrow c_3 \quad \vdash c_2 \Leftrightarrow c_4}{\vdash (c_1 \fatsemi c_2) \Leftrightarrow (c_3 \fatsemi c_4)} \qquad
\frac{\vdash c_1 \Leftrightarrow c_3 \quad \vdash c_2 \Leftrightarrow c_4}{\vdash (c_1 \oplus c_2) \Leftrightarrow (c_3 \oplus c_4)} \qquad
\frac{\vdash c_1 \Leftrightarrow c_3 \quad \vdash c_2 \Leftrightarrow c_4}{\vdash (c_1 \otimes c_2) \Leftrightarrow (c_3 \otimes c_4)}
$$

Figure 4. Signatures of level-2 Π-combinators.

7. The Problem with Higher-Order Functions

In the context of monoidal categories, it is known that a notion of higher-order functions emerges from having an additional degree of *symmetry*. In particular, both the **Int** construction of Joyal et al. (1996) and the closely related $\mathcal{G}$ construction of linear logic (Abramsky 1996) construct higher-order *linear* functions by considering a new category built on top of a given base traced monoidal category (Hasegawa 2009). The objects of the new category are of the form $\boxed{\tau_1}\boxed{\tau_2}$ where τ_1 and τ_2 are objects in the base category. Intuitively, this object represents the *difference* $\tau_1 - \tau_2$ with the component τ_1 viewed as conventional type whose elements represent values flowing, as usual, from producers to consumers, and the component τ_2 viewed as a *negative type* whose elements represent demands for values or equivalently values flowing backwards. Under this interpretation, a function is nothing but an object that converts a demand for an argument into the production of a result. We will explain in this section that the naïve generalization of the construction from monoidal to bimonoidal (aka rig) categories fails but that a recent result by Baas et al. (2012) might provide a path towards a solution.

7.1 The Int Construction

For this construction, we assume that we have extended Π with a trace operator to implement recursion or feedback, as done in some of the work on Π (Bowman et al. 2011; James and Sabry 2012a). The trace operator has the following type rule:

$$\frac{\vdash c : a + b \leftrightarrow a + c}{\vdash trace\ c : b \leftrightarrow c}$$

Under "normal" operation, a b-input is expected which is injected into the sum and fed to the traced computation. The evaluation continues until a c-value is produced, possibly after feeding several intermediate a-results back to the input.

We then extend Π with a new universe of types $\mathbb{T}$ that consists of composite types $\boxed{\tau_1}\boxed{\tau_2}$:

$$(\textit{1d types}) \quad \mathbb{T} \quad ::= \quad \boxed{\tau_1}\boxed{\tau_2}$$

We will refer to the original types τ as 0-dimensional (0d) types and to the new types $\mathbb{T}$ as 1-dimensional (1d) types. The 1d level is a "lifted" instance of Π with its own notions of empty, unit, sum, and product types, and its corresponding notions of isomorphisms and composition on these 1d types.

Our next step is to define lifted versions of the 0d types:

$$\begin{array}{rcl}
\mathbb{0} & \triangleq & \boxed{0}\boxed{0} \\
\mathbb{1} & \triangleq & \boxed{1}\boxed{0} \\
\boxed{\tau_1}\boxed{\tau_2} \boxplus \boxed{\tau_3}\boxed{\tau_4} & \triangleq & \boxed{\tau_1 + \tau_3}\boxed{\tau_2 + \tau_4} \\
\boxed{\tau_1}\boxed{\tau_2} \boxtimes \boxed{\tau_3}\boxed{\tau_4} & \triangleq & \boxed{(\tau_1 * \tau_3) + (\tau_2 * \tau_4)}\boxed{(\tau_1 * \tau_4) + (\tau_2 * \tau_3)}
\end{array}$$

Building on the idea that Π is a categorification of the natural numbers and following a long tradition that relates type isomorphisms and arithmetic identities (Di Cosmo 2005), one is tempted to think that the **Int** construction (as its name suggests) produces a categorification of the integers. Based on this hypothesis, the definitions above can be intuitively understood as arithmetic identities. The same arithmetic intuition which says $a-b=c-d$ iff $a+d=b+c$ explains the lifting of isomorphisms to 1d types:

$$\boxed{\tau_1}\boxed{\tau_2} \Leftrightarrow \boxed{\tau_3}\boxed{\tau_4} \quad \triangleq \quad (\tau_1 + \tau_4) \leftrightarrow (\tau_2 + \tau_3)$$

In other words, an isomorphism between 1d types is really an isomorphism between "re-arranged" 0d types where the negative input τ_2 is viewed as an output and the negative output τ_4 is viewed as an input. Using these ideas, it is now a fairly standard exercise to define the lifted versions of most of the combinators in Fig. 1.[5] We discuss a few cases in detail.

Trivial cases. Many of the 0d combinators lift easily to the 1d level. For example:

$$\begin{array}{rcl}
id & : & \mathbb{T} \Leftrightarrow \mathbb{T} \\
 & : & \boxed{\tau_1}\boxed{\tau_2} \Leftrightarrow \boxed{\tau_1}\boxed{\tau_2} \\
 & \triangleq & (\tau_1 + \tau_2) \leftrightarrow (\tau_2 + \tau_1) \\
id & = & swap_+
\end{array}$$

$$\begin{array}{rcl}
unite_+ & : & \mathbb{0} \boxplus \mathbb{T} \Leftrightarrow \mathbb{T} \\
 & \triangleq & \boxed{0 + \tau_1}\boxed{0 + \tau_2} \Leftrightarrow \boxed{\tau_1}\boxed{\tau_2} \\
 & \triangleq & ((0 + \tau_1) + \tau_2) \leftrightarrow ((0 + \tau_2) + \tau_1) \\
 & = & assocr_+ \fatsemi (id \oplus swap_+) \fatsemi assocl_+
\end{array}$$

Composition using $trace$***.*** Let $assoc_1$, $assoc_2$, and $assoc_3$ be the very simple Π combinators with the following signatures:

$$\begin{array}{rl}
assoc_1: & \tau_1 + (\tau_2 + \tau_3) \leftrightarrow (\tau_2 + \tau_1) + \tau_3 \\
assoc_2: & (\tau_1 + \tau_2) + \tau_3 \leftrightarrow (\tau_2 + \tau_3) + \tau_1 \\
assoc_3: & (\tau_1 + \tau_2) + \tau_3 \leftrightarrow \tau_1 + (\tau_3 + \tau_2)
\end{array}$$

Composition is then defined as follows:

$$\begin{array}{rcl}
(\fatsemi) & : & (\mathbb{T}_1 \Leftrightarrow \mathbb{T}_2) \to (\mathbb{T}_2 \Leftrightarrow \mathbb{T}_3) \to (\mathbb{T}_1 \Leftrightarrow \mathbb{T}_3) \\
(\fatsemi) & : & (\boxed{\tau_1}\boxed{\tau_2} \Leftrightarrow \boxed{\tau_3}\boxed{\tau_4}) \to (\boxed{\tau_3}\boxed{\tau_4} \Leftrightarrow \boxed{\tau_5}\boxed{\tau_6}) \\
 & & \quad \to (\boxed{\tau_1}\boxed{\tau_2} \Leftrightarrow \boxed{\tau_5}\boxed{\tau_6}) \\
 & \triangleq & ((\tau_1 + \tau_4) \leftrightarrow (\tau_2 + \tau_3)) \to ((\tau_3 + \tau_6) \leftrightarrow (\tau_4 + \tau_5)) \\
 & & \quad \to ((\tau_1 + \tau_6) \leftrightarrow (\tau_2 + \tau_5)) \\
f \fatsemi g = & & trace\ (assoc_1 \fatsemi (f \oplus id) \fatsemi assoc_2 \fatsemi (g \oplus id) \fatsemi assoc_3)
\end{array}$$

At the level of 1d-types, the first computation produces $\boxed{\tau_3}\boxed{\tau_4}$ which is consumed by the second computation. Expanding these types, we realize that the τ_4 produced from the first computation is actually a demand for a value of that type and that the τ_4 consumed by the second computation is actually produced in the backwards direction to satisfy demands by earlier computations. This explains the need for a feedback mechanism to send future values back to earlier computations.

Higher-order functions. Given values that flow in both directions, it is fairly straightforward to encode functions. At the type level, we have:

$$\begin{array}{rcl}
\boxminus(\boxed{\tau_1}\boxed{\tau_2}) & \triangleq & \boxed{\tau_2}\boxed{\tau_1} \\
\boxed{\tau_1}\boxed{\tau_2} \multimap \boxed{\tau_3}\boxed{\tau_4} & \triangleq & \boxminus(\boxed{\tau_1}\boxed{\tau_2}) \boxplus \boxed{\tau_3}\boxed{\tau_4} \\
 & \triangleq & \boxed{\tau_2 + \tau_3}\boxed{\tau_1 + \tau_4}
\end{array}$$

The $\boxminus$ flips producers and consumers and the $\multimap$ states that a function is just a transformer demanding values of the input type and producing values of the output type. As shown below, it now becomes possible to manipulate functions as values by currying and uncurrying:

$$\begin{array}{rcl}
flip & : & (\mathbb{T}_1 \Leftrightarrow \mathbb{T}_2) \to (\boxminus\mathbb{T}_2 \Leftrightarrow \boxminus\mathbb{T}_1) \\
flip\ f & = & swap_+ \fatsemi f \fatsemi swap_+ \\
 & & \\
curry & : & ((\mathbb{T}_1 \boxplus \mathbb{T}_2) \Leftrightarrow \mathbb{T}_3) \to (\mathbb{T}_1 \Leftrightarrow (\mathbb{T}_2 \multimap \mathbb{T}_3)) \\
curry\ f & = & assocl_+ \fatsemi f \fatsemi assocr_+ \\
 & & \\
uncurry & : & (\mathbb{T}_1 \Leftrightarrow (\mathbb{T}_2 \multimap \mathbb{T}_3)) \to ((\mathbb{T}_1 \boxplus \mathbb{T}_2) \Leftrightarrow \mathbb{T}_3) \\
uncurry\ f & = & assocr_+ \fatsemi f \fatsemi assocl_+
\end{array}$$

7.2 Cartesian Types

The **Int** construction works perfectly well as a technique to define higher-order functions if we just have *one* monoidal structure: the additive one as we have assumed so far. We will now try to extend the construction to encompass the other (multiplicative) monoidal structure. Recall that natural definition for the product of 1d types used above was:

$$\boxed{\tau_1}\boxed{\tau_2} \boxtimes \boxed{\tau_3}\boxed{\tau_4} \triangleq \boxed{(\tau_1 * \tau_3) + (\tau_2 * \tau_4)}\boxed{(\tau_1 * \tau_4) + (\tau_2 * \tau_3)}$$

That definition is "obvious" in some sense as it matches the usual understanding of types as modeling arithmetic identities. Using this definition, it is possible to lift all the 0d combinators involving products *except* the functor:

$$(\otimes) : (\mathbb{T}_1 \Leftrightarrow \mathbb{T}_2) \to (\mathbb{T}_3 \Leftrightarrow \mathbb{T}_4) \to ((\mathbb{T}_1 \boxtimes \mathbb{T}_3) \Leftrightarrow (\mathbb{T}_2 \boxtimes \mathbb{T}_4))$$

After a few failed attempts, we suspected that this definition of multiplication is not functorial. Further investigation reveals that our suspicion is well-justified and that there is indeed a fundamental technical problem. Furthermore, this problem is well known in algebraic topology and homotopy theory and was identified more than thirty years ago and advertised as the **phony multiplication** problem (Thomason 1980).

> This means that the **Int** construction only provides a limited notion of higher-order functions at the cost of losing the multiplicative structure at higher-levels. This observation, *that the **Int** construction on the additive monoidal structure does* not *allow one to lift multiplication in a straightforward manner* is less well-known than it should be.

This long-standing problem was recently solved by Baas et al. (2012) using a technique whose fundamental ingredients are to add more dimensions and then take homotopy colimits. The process of adding more dimensions is relatively straightforward: if the original intuition was that 1d types like $\boxed{\tau_1}\boxed{\tau_2}$ represent $\tau_1 - \tau_2$, i.e., two

[5] See Krishnaswami (2012)'s excellent blog post implementing this construction in OCaml.

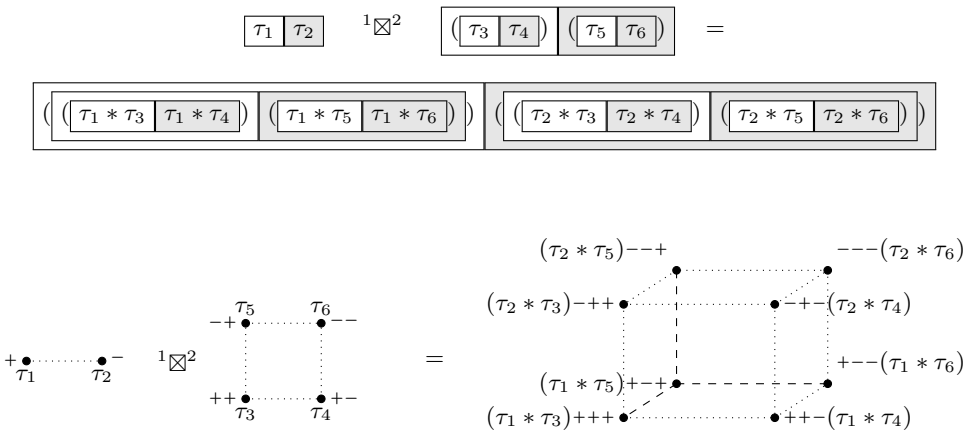


Figure 5. Example of multiplication of two cartesian types.

types one in the + direction and one in the − direction, then multiplication of two such 1d types ought to produce components in the ++, +−, −+, and −− directions and so on generalizing to more and more dimensions. We call the resulting n-dimensional types *cartesian types* and we illustrate the general idea using a simple example in Fig. 5. It remains to investigate whether this idea could lead to a generalization of our results to incorporate higher-order functions while retaining the multiplicative structure. Another intriguing point to consider is the connection between this idea and the recently proposed cubical models of type theory that also aim at producing computational interpretations of univalence (Bezem et al. 2014) sharing some of the same intuition of cartesian types.

8. Conclusion

We have developed a tight integration between *reversible circuits* with *symmetric rig weak groupoids* based on the following elements:

- reversible circuits are represented as terms witnessing morphisms between finite types in a symmetric rig groupoid;
- the term language for reversible circuits is universal; it could be used as a standalone point-free programming language or as a target for a higher-level language with a more conventional syntax;
- the symmetric rig groupoid structure ensures that programs can be combined using sums and products satisfying the familiar laws of these operations;
- the *weak* versions of the categories give us a second level of morphisms that relate programs to equivalent programs and is exactly captured in the coherence conditions of the categories; this level of morphisms also comes equipped with sums and products with the familiar laws and the coherence conditions capture how these operations interact with sequential composition;
- a sound and complete optimizer for reversible circuits can be represented as terms that rewrite programs in small steps witnessing this second level of morphisms.

Our calculus provides a semantically well-founded approach to the representation, manipulation, and optimization of reversible circuits. In principle, subsets of the optimization rules can be selected to rewrite programs to several possible canonical forms as desired. We aim to investigate such frameworks in the future.

From a much more general perspective, our result can be viewed as part of a larger programme aiming at a better integration of several disciplines most notably computation, topology, and physics. Computer science has traditionally been founded on models such as the λ-calculus which are at odds with the increasingly relevant physical principle of conservation of information as well as the recent foundational proposal of HoTT that identifies equivalences (i.e., reversible, information-preserving, functions) as a primary notion of interest.[6] Currently, these reversible functions are a secondary notion defined with reference to the full λ-calculus in what appears to be a detour. In more detail, current constructions start with the class of all functions $A \to B$, then introduce constraints to filter those functions which correspond to type equivalences $A \simeq B$, and then attempt to look for a convenient computational framework for effective programming with type equivalences. As we have shown, in the case of finite types, this is just convoluted since the collection of functions corresponding to type equivalences is the collection of isomorphisms between finite types and these isomorphisms can be inductively defined, giving rise to a well-behaved programming language and its optimizer.

More generally, reversible computational models — in which all functions have inverses — are known to be universal computational models (Bennett 1973) and more importantly they can be defined without any reference to irreversible functions, which ironically become the derived notion (Green and Altenkirch 2008). It is therefore, at least plausible, that a variant of HoTT based exclusively on reversible functions that directly correspond to equivalences would have better computational properties. Our current result is a step, albeit preliminary in that direction as it only applies to finite types. However, it is plausible that this categorification approach can be generalized to accommodate higher-order functions. The intuitive idea is that our current development based on the categorification of the commutative semiring of the natural numbers might be generalizable to the categorification of the ring of integers or even to the categorification of the field of rational numbers. The generalization to rings would introduce *negative types* and the generalization to fields would further introduce *fractional types*. As Sec. 7 suggests, there is good evidence that these generalizations would introduce some notion of higher-order functions. It is even possible to conceive of more exotic types such as types with square roots and

[6] The λ-calculus is not even suitable for keeping track of computational resources; linear logic (Girard 1987) is a much better framework for that purpose but it does not go far enough as it only tracks "multiplicative resources" (Sparks and Sabry 2014).

imaginary numbers by further generalizing the work to the field of *algebraic numbers*. These types have been shown to make sense in computations involving recursive datatypes such as trees that can be viewed as solutions to polynomials over type variables (Blass 1995; Fiore 2004; Fiore and Leinster 2004).

A. Commutative Semirings

Given that the structure of commutative semirings is central to this paper, we recall the formal algebraic definition. Commutative semirings are sometimes called *commutative rigs* as they are commutative rings without negative elements.

Definition 8. *A* commutative semiring *consists of a set R, two distinguished elements of R named 0 and 1, and two binary operations $+$ and $\cdot$, satisfying the following relations for any $a, b, c \in R$:*

$$\begin{array}{rcl} 0 + a & = & a \\ a + b & = & b + a \\ a + (b + c) & = & (a + b) + c \end{array}$$

$$\begin{array}{rcl} 1 \cdot a & = & a \\ a \cdot b & = & b \cdot a \\ a \cdot (b \cdot c) & = & (a \cdot b) \cdot c \end{array}$$

$$\begin{array}{rcl} 0 \cdot a & = & 0 \\ (a + b) \cdot c & = & (a \cdot c) + (b \cdot c) \end{array}$$

References

S. Abramsky. Retracing some paths in process algebra. In *CONCUR '96: Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin Heidelberg, 1996.

S. Abramsky. A structural approach to reversible computation. *Theor. Comput. Sci.*, 347:441–464, December 2005.

N. Baas, B. Dundas, B. Richter, and J. Rognes. Ring completion of rig categories. *Journal für die reine und angewandte Mathematik (Crelles Journal)*, 2013(674):43–80, Mar. 2012.

J. Baez and M. Stay. Physics, topology, logic and computation: a Rosetta stone. *New Structures for Physics*, pages 95–172, 2011.

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

T. Beke. Categorification, term rewriting and the knuth–bendix procedure. *Journal of Pure and Applied Algebra*, 215(5):728 – 740, 2011. ISSN 0022-4049. . URL http://www.sciencedirect.com/science/article/pii/S0022404910001325.

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

M. Bezem, T. Coquand, and S. Huber. A model of type theory in cubical sets. In *LIPIcs. Leibniz Int. Proc. Inform.*, volume 26, pages 107–128, Dagstuhl, Germany, 2014. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

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

W. J. Bowman, R. P. James, and A. Sabry. Dagger traced symmetric monoidal categories and reversible programming. In *RC*, 2011.

J. Carette. Is there a reasoned derivation of the coherence conditions for symmetric rig categories? http://mathoverflow.net/questions/207485/is-there-a-reasoned-derivation-of-the-coherence-conditions-for-symmetric-rig-cat, June 2015.

E. P. DeBenedictis. Reversible logic for supercomputing. In *Proceedings of the 2Nd Conference on Computing Frontiers*, CF '05, pages 391–402, New York, NY, USA, 2005. ACM. ISBN 1-59593-019-1. . URL http://doi.acm.org/10.1145/1062261.1062325.

R. Di Cosmo. A short survey of isomorphisms of types. *Mathematical. Structures in Comp. Sci.*, 15(5):825–838, Oct. 2005.

A. Di Pierro, C. Hankin, and H. Wiklicky. Reversible combinatory logic. *MSCS*, 16:621–637, August 2006.

K. Dosen and Z. Petric. *Proof-Theoretical Coherence*. KCL Publications (College Publications), London, 2004. (revised version available at: http://www.mi.sanu.ac.yu/~kosta/coh.pdf).

R. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21:467–488, 1982.

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

M. Fiore and T. Leinster. An objective representation of the Gaussian integers. *Journal of Symbolic Computation*, 37(6):707 – 716, 2004. ISSN 0747-7171. . URL http://www.sciencedirect.com/science/article/pii/S0747717104000094.

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.

M. P. Frank. *Reversibility for efficient computing*. PhD thesis, Massachusetts Institute of Technology, 1999.

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

J.-Y. Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

A. S. Green and T. Altenkirch. From reversible to irreversible computations. *Electron. Notes Theor. Comput. Sci.*, 210:65–74, July 2008.

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

M. Hasegawa. On traced monoidal closed categories. *MSCS*, 19:217–244, April 2009. ISSN 0960-1295. .

M. Hofmann and T. Streicher. The groupoid interpretation of type theory. In *Venice Festschrift*, pages 83–111, 1996.

R. James and A. Sabry. Theseus: A high-level language for reversible computation. In *Reversible Computation*, 2014. Booklet of work-in-progress and short reports.

R. P. James and A. Sabry. Information effects. In *POPL*, pages 73–84. ACM, 2012a.

R. P. James and A. Sabry. Isomorphic interpreters from logically reversible abstract machines. In *RC*, 2012b.

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

W. E. Kluge. A reversible SE(M)CD machine. In *International Workshop on Implementation of Functional Languages*, pages 95–113. Springer-Verlag, 2000.

N. Krishnaswami. The geometry of interaction, as an OCaml program. http://semantic-domain.blogspot.com/2012/11/in-this-post-ill-show-how-to-turn.html, 2012.

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

M. Laplaza. Coherence for distributivity. In *Lecture Notes in Mathematics*, number 281, pages 29–72. Springer Verlag, Berlin, 1972.

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

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

E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, Piscataway, NJ, USA, 1989. IEEE Press. ISBN 0-8186-1954-6. URL http://dl.acm.org/citation.cfm?id=77350.77353.

S.-C. Mu, Z. Hu, and M. Takeichi. An injective language for reversible computation. In *MPC*, pages 289–313, 2004.

nLab. rig category. http://ncatlab.org/nlab/show/rig+category.

A. Peres. Reversible logic and quantum computers. *Phys. Rev. A*, 32(6), Dec 1985.

S. L. Peyton Jones and A. L. M. Santos. A transformation-based optimiser for Haskell. *Sci. Comput. Program.*, 32(1-3):3–47, Sept. 1998. ISSN 0167-6423. . URL http://dx.doi.org/10.1016/S0167-6423(97)00029-4.

M. Saeedi and I. L. Markov. Synthesis and optimization of reversible circuits — a survey. *ACM Comput. Surv.*, 45(2):21:1–21:34, Mar. 2013. ISSN 0360-0300. . URL http://doi.acm.org/10.1145/2431211.2431220.

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

Z. Sparks and A. Sabry. Superstructural reversible logic. In *3rd International Workshop on Linearity*, 2014.

R. Thomason. Beware the phony multiplication on Quillen's $\mathcal{A}^{-1}\mathcal{A}$. *Proc. Amer. Math. Soc.*, 80(4):569–573, 1980.

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

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

T. Yokoyama and R. Glück. A reversible programming language and its invertible self-interpreter. In *PEPM*, pages 144–153. ACM, 2007.