A Sound and Complete Calculus for Reversible Circuit Equivalence

Jacques Carette
McMaster University
carette@mcmaster.ca

Amr Sabry
Indiana University
sabry@indiana.edu

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 groupoids*. The coherence conditions of these categories then produces, 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 groupoids.

1. Introduction

Reversible circuits are NOT a restriction; they are a generalization; conventional irreversible circuits are a special case.

reversible circuits for supercomputers of the future (DeBenedictis 2005)

- BACKGROUND: realizing HoTT requires we be able to program with type equivalences and equivalences of type equivalences and so on; univalence is a postulate; caveat Coquand et al.
- RESULT: limit ourselves to finite types: what emerges is an interesting universal language for combinational reversible circuits that comes with a calculus for writing circuits and a calculus for manipulating that calculus; in other words; rules for writing circuits and rules for rewriting (optimizing) circuits

Define and motivate that we are interested in defining HoTT equivalences of types, characterizing them, computing with them, etc.

Homotopy type theory (HoTT) (The Univalent Foundations Program 2013) has a convoluted treatment of functions. It starts with a class of arbitrary functions, singles out a smaller class of "equivalences" via extensional methods, and then asserts via the *univalence* **axiom** that the class of functions just singled out is equivalent to paths. Why not start with functions that are, by construction, equivalences?

The idea that computation should be based on "equivalences" is an old one and is motivated by physical considerations. Because physics requires various conservation principles (including conservation of information) and because computation is fundamentally a physical process, every computation is fundamentally an equivalence that preserves information. This idea fits well with the HoTT philosophy that emphasizes equalities, isomorphisms, equivalences, and their computational content.

In more detail, a computational world in which the laws of physics are embraced and resources are carefully maintained (e.g., quantum computing (Nielsen and Chuang 2000; Abramsky and Coecke 2004)), programs must be reversible. Although this is apparently a limiting idea, it turns out that conventional computation can be viewed as a special case of such resource-preserving reversible programs. This thesis has been explored for many years from different perspectives (Fredkin and Toffoli 1982; Toffoli 1980; Bennett 2010, 2003, 1973; Landauer 1961, 1996) and more recently in the context of type isomorphisms (James and Sabry 2012a).

This paper explores the basic ingredients of HoTT from the perspective that computation is all about type isomorphisms. Because the issues involved are quite subtle, the paper is an executable `Agda 2.4.0` file with the global `without-K` option enabled. The main body of the paper reconstructs the main features of HoTT for the limited universe of finite types consisting of the empty type, the unit type, and sums and products of types. Sec. 7 outlines directions for extending the result to richer types.

Quantum Computing. Quantum physics differs from classical physics in many ways:

- Superpositions
- Entanglement
- Unitary evolution
- Composition uses tensor products
- Non-unitary measurement

Quantum Computing & Programming Languages.

- It is possible to adapt all at once classical programming languages to quantum programming languages.
- Some excellent examples discussed in this workshop
- This assumes that classical programming languages (and implicitly classical physics) can be smoothly adapted to the quantum world.
- There are however what appear to be fundamental differences between the classical and quantum world that make them incompatible

- Let us *re-think* classical programming foundations before jumping to the quantum world.

  Resource-Aware Classical Computing.

- The biggest questionable assumption of classical programming is that it is possible to freely copy and discard information
- A classical programming language which respects no-cloning and no-discarding is the right foundation for an eventual quantum extension
- We want these properties to be inherent in the language; not an afterthought filtered by a type system
- We want to program with isomorphisms or equivalences
- The simplest instance is permutations between finite types which happens to correspond to reversible circuits.

Representing Reversible Circuits: truth table, matrix, reed muller expansion, product of cycles, decision diagram, etc.

any easy way to reproduce Figure 4 on p.7 of Saeedi and Markov? important remark: these are all *Boolean* circuits! Most important part: reversible circuits are equivalent to permutations.

A (Foundational) Syntactic Theory. Ideally, want a notation that

1. is easy to write by programmers
2. is easy to mechanically manipulate
3. can be reasoned about
4. can be optimized.

Start with a *foundational* syntactic theory on our way there:

1. easy to explain
2. clear operational rules
3. fully justified by the semantics
4. sound and complete reasoning
5. sound and complete methods of optimization

A Syntactic Theory. Ideally want a notation that is easy to write by programmers and that is easy to mechanically manipulate for reasoning and optimizing of circuits.

Syntactic calculi good. Popular semantics: Despite the increasing importance of formal methods to the computing industry, there has been little advance to the notion of a "popular semantics" that can be explained to *and used* effectively (for example to optimize or simplify programs) by non-specialists including programmers and first-year students. Although the issue is by no means settled, syntactic theories are one of the candidates for such a popular semantics for they require no additional background beyond knowledge of the programming language itself, and they provide a direct support for the equational reasoning underlying many program transformations.

The primary abstraction in HoTT is 'type equivalences.' If we care about resource preservation, then we are concerned with 'type equivalences'.

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, permutations and 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. 8. 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:

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

Our starting point is a result by Fiore et. al (Fiore 2004; Fiore et al. 2006) that completely characterizes the isomorphisms between finite types using the axioms of commutative semirings. (See Appendix A for the complete definition of commutative semirings.) 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.

In previous work (James and Sabry 2012a), we introduced the Π family of languages whose core computations are these isomorphisms between finite types. Building on that work and on 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 $=$.

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 as it appears to be the most intuitive in our setting.

Definition 1 (Quasi-inverse)**.** *For a function $f : A \to B$, a* quasi-inverse *of f 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 the quasi-inverse) and one that uses boolean negation for f (and hence for the quasi-inverse). 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 $\times$. □

[Do we want to have a bunch of appendices, or perhaps a web link, to all the Agda code which formalizes all of this? —JC]

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:

- for all $a \in A$, $f_1(a) = f_2(a)$, and
- for all $b \in B$, $g_1(b) = g_2(b)$.

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

[actually, it is equivalences-of-equivalences which are fundamentally based on fun-ext; type equivalences themselves are mostly computationally effective. Otherwise they could not be equivalent to permutations... But they are somehow less tangible, while permutations are quite concrete. —JC]

Type equivalences are fundamentally based on function extensionality and hence are generally not computationally effective. 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 (and as we prove equivalent) characterization of type equivalences based on permutations of finite sets.

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 our 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) position this element could be mapped to by the larger equivalence and in each case construct an equivalence that excludes this element. □

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:

$$\begin{array}{ll} \mathsf{iso\text{-}plus} & : \{m\ n : \mathbb{N}\} \to (\mathsf{Fin}\ m \uplus \mathsf{Fin}\ n) \simeq \mathsf{Fin}\ (m + n) \\ \mathsf{iso\text{-}times} & : \{m\ n : \mathbb{N}\} \to (\mathsf{Fin}\ m \times \mathsf{Fin}\ n) \simeq \mathsf{Fin}\ (m * n) \end{array}$$

□

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. A permutation of one set to another is represented using two sizes: n for the size of the input finite set and m for the size of the resulting finite set. Naturally in any well-formed permutations, these two sizes are equal but the presence of both types allows us to conveniently define a permutation $\mathsf{CPerm}\ m\ n$ using four components. The first two components are:

- a vector of size n containing elements drawn from the finite set $\mathsf{Fin}\ m$;
- a dual vector of size m containing elements drawn from the finite set $\mathsf{Fin}\ n$;

Each of the above vectors can be interpreted as a map f that acts on the incoming finite set sending the element at index i to position $f\,!!\,i$ in the resulting finite set. To guarantee that these maps define an actual permutation, the last two components are proofs that the sequential composition of the maps in both direction produce the

identity. Given this representation, we can prove that two permutations are equal if the underlying vectors are strictly equal. The proof proceeds using the vacuous permutation CPerm 0 0 for the additive unit and the trivial permutation CPerm 1 1 for the multiplicative unit. [we should detail sum and product as well, as they are non-trivial. —JC] □

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.*

[the other thing worth pointing out is that every axiom of semirings (of types) is an equivalence, and thus corresponds to a permutation. Some are trivial: associativity of + in particular gets mapped to the identity permutation. However, some are more interesting. In particular, commutativity of * is intimately related to matrix transpose. —JC]

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.

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 our previous work which proposed a term language for permutations and adapt it to be used to express, compute with, and reason about type equivalences between finite types.

3.1 The Π-Languages

In previous work (James and Sabry 2012a), we introduced the Π family of languages whose only computations are isomorphisms between finite types. 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[1] that witness the type isomorphism in the middle.

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

3.2 Example Circuits

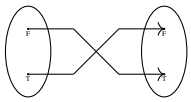
The language Π is universal for reversible combinational circuits (James and Sabry 2012a).[2] We illustrate the expressiveness of the language with a few short examples.

The first example is simply boolean negation which is easily achieved:

```
BOOL : U
BOOL = PLUS ONE ONE

NOT₁ : BOOL ↔ BOOL
NOT₁ = swap₊
```

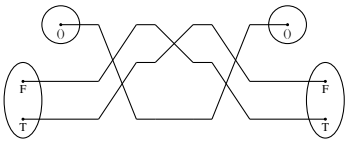
Viewing the combinator as a permutation on finite sets, we might visualize it as follows:



Naturally there are many ways of encoding boolean negation. The following example is a more convoluted circuit that computes the same function:

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

Viewing this combinator as a permutation on finite sets, we might visualize it as follows:



Writing circuits using the raw syntax for combinators is clearly tedious. In other work, we have investigated 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:

```
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
```

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

$$
\begin{array}{rrcll}
identl_+ : & 0+\tau & \leftrightarrow & \tau & : identr_+ \\
swap_+ : & \tau_1+\tau_2 & \leftrightarrow & \tau_2+\tau_1 & : swap_+ \\
assocl_+ : & \tau_1+(\tau_2+\tau_3) & \leftrightarrow & (\tau_1+\tau_2)+\tau_3 & : assocr_+ \\
identl_* : & 1*\tau & \leftrightarrow & \tau & : identr_* \\
swap_* : & \tau_1*\tau_2 & \leftrightarrow & \tau_2*\tau_1 & : swap_* \\
assocl_* : & \tau_1*(\tau_2*\tau_3) & \leftrightarrow & (\tau_1*\tau_2)*\tau_3 & : assocr_*
\end{array}
$$

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

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

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

Figure 1. Π-combinators (James and Sabry 2012a)

```
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 incidentally 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* 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:

$$
\begin{array}{ll}
\mathsf{eval} & : \{t_1\ t_2 : \mathsf{U}\} \to (t_1 \leftrightarrow t_2) \to [\![\ t_1\ ]\!] \to [\![\ t_2\ ]\!] \\
\mathsf{evalB} & : \{t_1\ t_2 : \mathsf{U}\} \to (t_1 \leftrightarrow t_2) \to [\![\ t_2\ ]\!] \to [\![\ t_1\ ]\!]
\end{array}
$$

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 previous publications (Bowman et al. 2011; James and Sabry 2012b,a) 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:

$$
\begin{array}{ll}
\mathsf{c2equiv} & : \{t_1\ t_2 : \mathsf{U}\} \to (c : t_1 \leftrightarrow t_2) \to [\![\ t_1\ ]\!] \simeq [\![\ t_2\ ]\!] \\
\mathsf{c2perm} & : \{t_1\ t_2 : \mathsf{U}\} \to (c : t_1 \leftrightarrow t_2) \to \\
 & \quad \mathsf{CPerm}\ (\mathsf{size}\ t_2)\ (\mathsf{size}\ t_1)
\end{array}
$$

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. The traditional definition of equivalence might equate two permutations if their actions on every input produce the same output but we again resist that extensional reasoning. Instead we are interested in a calculus, a set of rules, that can be used to rewrite combinators preserving their meaning. It is trivial to come up with a few rules such as:

$$
\begin{array}{lll}
\mathsf{id}\Leftrightarrow & : & \{t_1\ t_2 : \mathsf{U}\}\ \{c : t_1 \leftrightarrow t_2\} \to c \Leftrightarrow' c \\
\\
\mathsf{trans}\Leftrightarrow & : & \{t_1\ t_2 : \mathsf{U}\}\ \{c_1\ c_2\ c_3 : t_1 \leftrightarrow t_2\} \to \\
 & & (c_1 \Leftrightarrow' c_2) \to (c_2 \Leftrightarrow' c_3) \to (c_1 \Leftrightarrow' c_3) \\
\\
\mathsf{assoc}\Leftrightarrow & : & \{t_1\ t_2\ t_3\ t_4 : \mathsf{U}\} \\
 & & \{c_1 : t_1 \leftrightarrow t_2\}\ \{c_2 : t_2 \leftrightarrow t_3\}\ \{c_3 : t_3 \leftrightarrow t_4\} \to \\
 & & (c_1 \odot (c_2 \odot c_3)) \Leftrightarrow' ((c_1 \odot c_2) \odot c_3) \\
\\
\mathsf{id}\odot\Leftrightarrow & : & \{t_1\ t_2 : \mathsf{U}\}\ \{c : t_1 \leftrightarrow t_2\} \to (\mathsf{id}\leftrightarrow \odot c) \Leftrightarrow' c \\
\\
\mathsf{swap}_+\Leftrightarrow & : & \{t_1\ t_2\ t_3\ t_4 : \mathsf{U}\}\ \{c_1 : t_1 \leftrightarrow t_2\}\ \{c_2 : t_3 \leftrightarrow t_4\} \to \\
 & & (\mathsf{swap}_+ \odot (c_1 \oplus c_2)) \Leftrightarrow' ((c_2 \oplus c_1) \odot \mathsf{swap}_+)
\end{array}
$$

which are evidently sound. The challenge of course is to come up with a sound and complete set of such rules.

Before we embark on the categorification program in the next section, we show that, with some ingenuity, one can develop a reasonable set of rewrite rules that would allow us to prove that the two negation circuits from the previous section are actual 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+ □
```

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. The following quote sets up the context for our development.

> 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).

5.1 Monoidal Categories

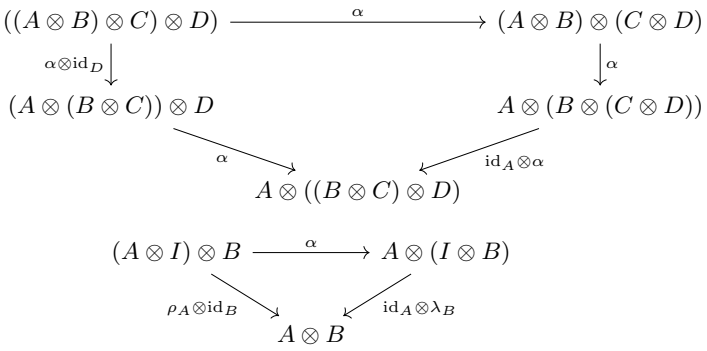
Definition 3 (Category)**.** *A* category $\mathbf{C}$ *consists of:*

- *a class* $|\mathbf{C}|$ *of* objects*, denoted* $A, B, C, \ldots$*;*
- *for each pair of objects* A, B*, a set* $\hom_{\mathbf{C}}(A, B)$ *of* morphisms*, which are denoted* $f : A \to B$*;*
- identity morphisms $\mathrm{id}_A : A \to A$ *and the operation of* composition*: if* $f : A \to B$ *and* $g : B \to C$*, then* $g \circ f : A \to C$ *subject to three equations stating that* id *is the left and right unit for composition and that composition is associative:*
  - *Unit:* $f \circ \mathrm{id} = f = \mathrm{id} \circ f$
  - *Associativity:* $h \circ (g \circ f) = (h \circ g) \circ f$

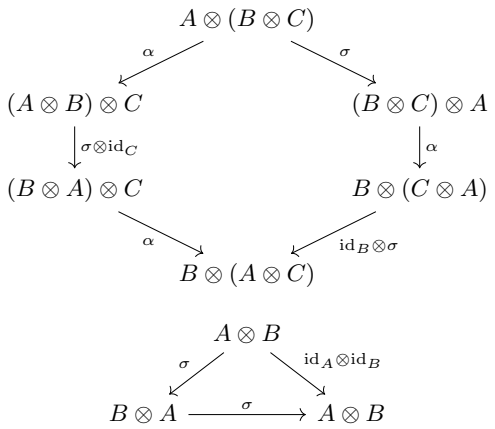
An important subtlety of this definition is in the equality of arrows. Indeed the programs f and $\mathrm{id} \circ f$ are different syntactic programs and their compilation might be different in any concrete implementation of the language. Hence the equality of arrows should be based upon the semantic equivalence of these two programs, for example, using *observational equivalence*. The definition of categories allows one to specify the arrows and the objects, but is not parametrized by an equivalence relationship over the arrows. The $=$ in the definition of categories is the mathematical equivalence of the mathematical entities that are the arrows. Hence the 'operations' that are the arrows must already include the relevant equivalence relation on programs. This is sometimes referred to as a *term model* and essentially states that operations of the language are the equivalence classes of the programs one writes in the language. Thus f and $\mathrm{id} \circ f$ become *the same arrow*. See Section 2.2 of Barr and Wells's book (Barr and Wells 1995) for a detailed discussion and Def. 2.4 of Moggi's paper (Moggi 1991) for an example of such usage.

Definition 4 (Monoidal Category)**.** *A* monoidal category *(Mac Lane 1971) is a category with the following additional structure:*

- *a functor* $\otimes$ *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 following two diagrams (known as the* associativity pentagon *and the* triangle for unit*) commute:*



Definition 5 (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*):*



Define monoidal categories; explain categorification of monoid explain that arrows can be viewed as permutations; not all arrows are 'equal'; leads to coherence conditions

5.2 Coherence Conditions

for monoidal categories are reasonably easy; explain in detail

5.3 Commutative Rig Groupoids

these things are not very well-known; these are the right beasts that are the categorification of commutative semigroups.

categorification (Baez and Dolan 1998) of the natural numbers. A simple (slightly degenerate) example of such

This has been done generically: coherence conditions for commutative rig groupoids. These generalize type equivalences and permutations;

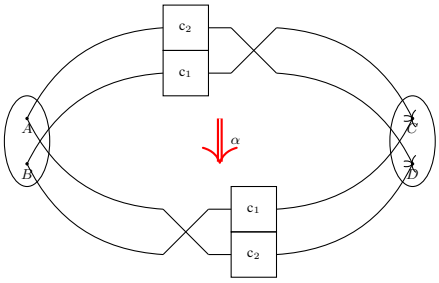
We haven't said anything about the categorical structure: it is not just a commutative semiring but a commutative rig; this is crucial because the former doesn't take composition into account. Perhaps that is the next section in which we talk about computational interpretation as one of the fundamental things we want from a notion of computation is composition (cf. Moggi's original paper on monads).

Type equivalences (such as between $A \times B$ and $B \times A$) are Functors.
Equivalences between Functors are Natural Isomorphisms. At the value-level, they induce 2-morphisms:

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

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

2-morphism of circuits



Categorification II. The categorification of a semiring is called a Rig Category. As with a semiring, there are two monoidal structures, which interact through some distributivity laws.

Theorem 9. *The following are Symmetric Bimonoidal Groupoids:*

- *The class of all types (Set)*
- *The set of all finite types*
- *The set of permutations*
- *The set of equivalences between finite types*
- *Our syntactic combinators*

The coherence rules for Symmetric Bimonoidal groupoids give us 58 rules.
Categorification III.

Conjecture 1. *The following are Symmetric Rig Groupoids:*

- *The class of all types (Set)*
- *The set of all finite types, of permutations, of equivalences between finite types*
- *Our syntactic combinators*

and of course the punchline:

Theorem 10 (Laplaza 1972)**.** *There is a sound and complete set of coherence rules for Symmetric Rig Categories.*

Conjecture 2. *The set of coherence rules for Symmetric Rig Groupoids are a sound and complete set for circuit equivalence.*

6. Revised Π and its Optimizer

6.1 Revised Syntax

The refactoring of Pi from the inspiration of symmetric rig groupoids. The added combinators are redundant (from an operational perspective) exactly because of the coherences. But some of these higher combinators have rather non-trivial relations to each other [ex: pentagon, hexagon, and some of the weirder Laplaza rules]. Plus the 'minimalistic' Pi leads to much larger programs with LOTS of extra redexes.

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

6.2 Optimization Rules

What we need now is Pi plus another layer to top to optimize Pi programs; no ad hoc rules; principled rules;

6.3 Examples

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, Street, and Verity (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. The objects of the new category are of the form $\boxed{\tau_1 \mid \tau_2}$ 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, and as we explain below, a function is nothing but an object that converts a demand for an argument into the production of a result.

7.1 Conventional Construction on Unpointed Types

We begin our formal development by extending Π — at any level — with a new universe of types $\mathbb{T}$ that consists of composite types $\boxed{\tau_1 \mid \tau_2}$:

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

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 notion of isomorphisms on these 1d types.

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

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

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 explains the lifting of isomorphisms to 1d types:

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

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 Table 1.[3] There are however a few interesting cases whose appreciation is essential for the remainder of the paper.

Easy Lifting. Many of the 0d combinators lift easily to the 1d level. For example:

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

$$\begin{array}{rcl} identl_+ & : & \mathbb{0} \boxplus \mathbb{T} \Leftrightarrow \mathbb{T} \\ & = & assocr_+ \fatsemi (id \oplus swap_+) \fatsemi assocl_+ \end{array}$$

Composition using $trace$***.***

$$\begin{array}{rl} (\fatsemi) : & (\mathbb{T}_1 \Leftrightarrow \mathbb{T}_2) \to (\mathbb{T}_2 \Leftrightarrow \mathbb{T}_3) \to (\mathbb{T}_1 \Leftrightarrow \mathbb{T}_3) \\ f \fatsemi g = & trace\ (assoc_1 \fatsemi (f \oplus id) \fatsemi assoc_2 \fatsemi (g \oplus id) \fatsemi assoc_3) \end{array}$$

New combinators $curry$ ***and*** $uncurry$ ***for higher-order functions.***

$$\boxminus\left(\boxed{\tau_1}\boxed{\tau_2}\right) \triangleq \boxed{\tau_2}\boxed{\tau_1}$$

$$\boxed{\tau_1}\boxed{\tau_2} \multimap \boxed{\tau_3}\boxed{\tau_4} \triangleq \boxminus\left(\boxed{\tau_1}\boxed{\tau_2}\right) \boxplus \boxed{\tau_3}\boxed{\tau_4} \triangleq \boxed{\tau_2 + \tau_3}\boxed{\tau_1 + \tau_4}$$

$$\begin{array}{rcl} flip & : & (\mathbb{T}_1 \Leftrightarrow \mathbb{T}_2) \to (\boxminus\mathbb{T}_2 \Leftrightarrow \boxminus\mathbb{T}_1) \\ flip\ f & = & swap_+ \fatsemi f \fatsemi swap_+ \end{array}$$

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

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

The "phony" multiplication that is not a functor. The definition for the product of 1d types used above is:

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

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 which would mean 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 is less well-known that it should be. Further investigation reveals that this observation is intimately related to a well-known problem in algebraic topology and homotopy theory that was identified thirty years ago as the "phony" multiplication (Thomason 1980) in a special class categories related to ours. This problem was recently solved (Baas et al. 2012) using a technique whose fundamental ingredients are to add more dimensions and then take homotopy colimits. It remains to investigate whether this idea can be integrated with our development to get higher-order functions while retaining the multiplicative structure.

8. Conclusion

- add trace to make language Turing complete
- generalize from commutative rig to field as a way to get some notion of h.o. functions

We 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. In the case of finite types, this is just convoluted as 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 programming language that is complete for combinational circuits (James and Sabry 2012a).

To make these connections precise, we now explore permutations over finite sets as an explicit computational realization of isomorphisms between finite types and prove that the type of all permutations between finite sets is equivalent to the type of type equivalences but with better computational properties, i.e., without the reliance on function extensionality.

Our theorem shows that, in the case of finite types, reversible computation via type isomorphisms *is* the computational interpretation of univalence. The alternative presentation of the theorem exposes it as an instance of *univalence*. In the conventional HoTT setting, univalence is postulated as an axiom that lacking computational content. In more detail, the conventional HoTT approach starts with two, a priori, different notions: functions and identities (paths), and then postulates an equivalence between a particular class of functions (equivalences) and paths. Most functions are not equivalences and hence are evidently unrelated to paths. An interesting question then poses itself: since reversible computational models — in which all functions have inverses — are known to be universal computational models, what would happen if we considered a variant of HoTT based exclusively on reversible functions? Presumably in such a variant, all functions — being reversible — would potentially correspond to paths and the distinction between the two notions would vanish making the univalence postulate unnecessary. This is the precise technical idea that is captured in theorem above for the limited case of finite types.

We focused on commutative semiring structures. An obvious question is whether the entire setup can be generalized to a larger algebraic structure like a field. That requires additive and multiplicative inverses. There is evidence that this negative and fractional types are sensible and that they would give rise to some form of higher-order functions. There is also evidence for even more exotic types that are related to algebraic numbers including roots and imaginary numbers.

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

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 and B. Coecke. A categorical semantics of quantum protocols. In *LICS*, 2004.

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

M. Barr and C. Wells. *Category theory for computing science (2. ed.)*. Prentice Hall international series in computer science. Prentice Hall, 1995. ISBN 978-0-13-323809-9.

C. 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.

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

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

W. J. Bowman, R. P. James, and A. Sabry. Dagger Traced Symmetric Monoidal Categories and Reversible Programming. In *RC*, 2011.

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.

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`).

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

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.

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

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.

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.

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

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

E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93:55–92, July 1991.

M. Nielsen and I. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, Cambridge, 2000.

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

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.

A. Commutative Semirings

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

Definition 6. *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 $a, b, c \in R$:*

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

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

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

B. Diagrammatic Optimization