

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

[do remember that the code which is embedded in this file is not the one in the sources below (anymore). Not sure if it matters? —JC]

[Also: double-blind this year! I guess we need to omit our names. I don’t think the text will need changed for that. —JC]

<https://github.com/JacquesCarette/pi-dual>

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.

[is any of this stuff useful for the introduction, or should it all go? —JC] 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 (\perp), 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:

$$\begin{aligned} |\perp| &= 0 \\ |\top| &= 1 \\ |A \uplus B| &= |A| + |B| \\ |A \times B| &= |A| * |B| \end{aligned}$$

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

Definition 2 (Equivalence of types). *Two types A and B are equivalent $A \simeq B$ if there exists a function $f : A \rightarrow B$ 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 \perp , the multiplicative unit is \top , and the two binary operations are \uplus and \times . \square

[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:

$$\begin{aligned} \perp \uplus A &\simeq A \\ \top \times A &\simeq A \\ A \times (B \times C) &\simeq (A \times B) \times C \\ A \times \perp &\simeq \perp \\ A \times (B \uplus C) &\simeq (A \times B) \uplus (A \times C) \end{aligned}$$

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 $\text{EQ}_{A,B}$. For a given A and B , the elements of $\text{EQ}_{A,B}$ are all the ways in which we can prove $A \simeq B$. For example, $\text{EQ}_{\text{Bool}, \text{Bool}}$ 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 $\text{EQ}_{A,B}$ 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 $\text{EQ}_{A,B}$ 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 $e_1, e_2 : \text{EQ}_{A,B}$ 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 $\perp \simeq \perp$, 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. \square

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 $B = \top \uplus (\top \uplus (\top \uplus (\top \uplus (\top \uplus (\top \uplus \perp))))$, we can find equivalences from either type to the finite set $\text{Fin } 6$ 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 $A \simeq \text{Fin } m$, $B \simeq \text{Fin } n$ 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 $m = \text{succ } m'$ and $n = \text{succ } n'$. The result follows in this case by induction assuming we can establish that the equivalence between A and B , i.e., the equivalence between $\text{Fin } (\text{succ } m')$ and $\text{Fin } (\text{succ } n')$, implies an equivalence between $\text{Fin } m'$ and $\text{Fin } n'$. 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 $\text{Fin } (\text{succ } m')$ 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. \square

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 ($\text{Fin } m$ for natural number m) forms a commutative semiring (up to \simeq).*

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

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

\square

Theorem 5. *The collection of all permutations $\text{PERM}_{m,n}$ between finite sets $\text{Fin } m$ and $\text{Fin } n$ 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 $\text{CPerm } m\ n$ using four components. The first two components are:

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

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 $\text{CPerm } 0\ 0$ for the additive unit and the trivial permutation $\text{CPerm } 1\ 1$ for the multiplicative unit. [we should detail sum and product as well, as they are non-trivial. —JC] \square

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

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

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.

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

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 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), the Π family of languages was introduced, 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, $\text{inl } v$ and $\text{inr } v$ for injections into sum types, and (v_1, v_2) for product types:

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

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 constants¹ 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 combinator in sequential composition.

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

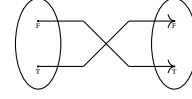
NOT1 : BOOL ↔ BOOL
NOT1 = swap+

```

¹ where swap_+ and swap_* are self-dual.

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).

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



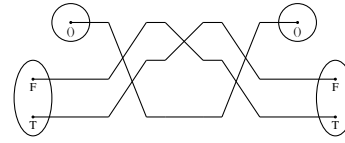
Naturally there are many ways of encoding boolean negation. The following example is a more convoluted circuit that computes the same function:

```

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, 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:

```

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

```


$identl_+ :$	$0 + \tau \leftrightarrow \tau$	$: identr_+$		
$swap_+ :$	$\tau_1 + \tau_2 \leftrightarrow \tau_2 + \tau_1$	$: swap_+$		
$assocl_+ :$	$\tau_1 + (\tau_2 + \tau_3) \leftrightarrow (\tau_1 + \tau_2) + \tau_3$	$: assocr_+$		
$identl_* :$	$1 * \tau \leftrightarrow \tau$	$: identr_*$		
$swap_* :$	$\tau_1 * \tau_2 \leftrightarrow \tau_2 * \tau_1$	$: swap_*$		
$assocl_* :$	$\tau_1 * (\tau_2 * \tau_3) \leftrightarrow (\tau_1 * \tau_2) * \tau_3$	$: assocr_*$		
$dist :$	$(\tau_1 + \tau_2) * \tau_3 \leftrightarrow (\tau_1 * \tau_3) + (\tau_2 * \tau_3)$	$: factor$		
$dist_0 :$	$0 * \tau \leftrightarrow 0$	$: factorl_0$		

	$\vdash id : \tau \leftrightarrow \tau$	$\vdash c_1 : \tau_1 \leftrightarrow \tau_2 \quad \vdash c_2 : \tau_2 \leftrightarrow \tau_3$
		$\vdash c_1 \circ c_2 : \tau_1 \leftrightarrow \tau_3$
	$\vdash c_1 : \tau_1 \leftrightarrow \tau_2 \quad \vdash c_2 : \tau_3 \leftrightarrow \tau_4$	
	$\vdash c_1 \oplus c_2 : \tau_1 + \tau_3 \leftrightarrow \tau_2 + \tau_4$	
	$\vdash c_1 : \tau_1 \leftrightarrow \tau_2 \quad \vdash c_2 : \tau_3 \leftrightarrow \tau_4$	
	$\vdash c_1 \otimes c_2 : \tau_1 * \tau_3 \leftrightarrow \tau_2 * \tau_4$	

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

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:

$eval : \{t_1 \ t_2 : U\} \rightarrow (t_1 \leftrightarrow t_2) \rightarrow \llbracket t_1 \rrbracket \rightarrow \llbracket t_2 \rrbracket$
 $evalB : \{t_1 \ t_2 : U\} \rightarrow (t_1 \leftrightarrow t_2) \rightarrow \llbracket t_2 \rrbracket \rightarrow \llbracket t_1 \rrbracket$

In the definition, the function $\llbracket \cdot \rrbracket$ maps each type constructor to its Agda denotation, e.g., it maps the type 0 to \perp , 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 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:

$c2equiv : \{t_1 \ t_2 : U\} \rightarrow (c : t_1 \leftrightarrow t_2) \rightarrow \llbracket t_1 \rrbracket \simeq \llbracket t_2 \rrbracket$
 $c2perm : \{t_1 \ t_2 : U\} \rightarrow (c : t_1 \leftrightarrow t_2) \rightarrow$
 $\quad CPerm \ (size \ t_2) \ (size \ t_1)$

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:

```
[gobble=2]
id⇔ : {t1 t2 : U} {c : t1 ↔ t2} → c ⇔' c

trans⇔ : {t1 t2 : U} {c1 c2 c3 : t1 ↔ t2} →
  (c1 ⇔' c2) → (c2 ⇔' c3) → (c1 ⇔' c3)

assoc⇔ : {t1 t2 t3 t4 : U}
  {c1 : t1 ↔ t2} {c2 : t2 ↔ t3} {c3 : t3 ↔ t4} →
  (c1 ∘ (c2 ∘ c3)) ⇔' ((c1 ∘ c2) ∘ c3)

id⊙⇔ : {t1 t2 : U} {c : t1 ↔ t2} → (id↔ ∘ c) ⇔'

swap+⇔ : {t1 t2 t3 t4 : U} {c1 : t1 ↔ t2} {c2 : t3 ↔ t4} →
  (swap+ ∘ (c1 ⊕ c2)) ⇔' ((c2 ⊕ c1) ∘ swap+)
```

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* ∘ uniti*)))
    ⇔ (id↔ ∘ assoc⊙l)
  uniti* ∘ ((swap* ∘ (swap+ ∘ id↔)) ∘ (swap* ∘ uniti*))
    ⇔ (id↔ ∘ (swap+ ∘ id↔))
  uniti* ∘ (((id↔ ∘ swap+) ∘ swap*) ∘ (swap* ∘ uniti*))
    ⇔ (id↔ ∘ assoc⊙r)
  uniti* ∘ ((id↔ ∘ swap+) ∘ (swap* ∘ (swap* ∘ uniti*)))
    ⇔ (id↔ ∘ (id↔ ∘ assoc⊙l))
  uniti* ∘ ((id↔ ∘ swap+) ∘ ((swap* ∘ swap*) ∘ uniti*))
    ⇔ (id↔ ∘ (id↔ ∘ (linv⊙l ∘ id↔)))
  uniti* ∘ ((id↔ ∘ swap+) ∘ (id↔ ∘ uniti*))
    ⇔ (id↔ ∘ (id↔ ∘ idl⊙l))
  uniti* ∘ ((id↔ ∘ swap+) ∘ uniti*)
    ⇔ (assoc⊙l)
  (uniti* ∘ (id↔ ∘ swap+)) ∘ uniti*
    ⇔ (uniti* ⇔ id↔)
  (swap+ ∘ uniti*) ∘ uniti*
    ⇔ (assoc⊙r)
  swap+ ∘ (uniti* ∘ uniti*)
    ⇔ (id↔ ∘ linv⊙l)
```

$\text{swap}_+ \odot \text{id} \leftrightarrow$
 $\Leftrightarrow (\text{id} \odot \text{I})$
 $\text{swap}_+ \square$

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 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. 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 to each other.

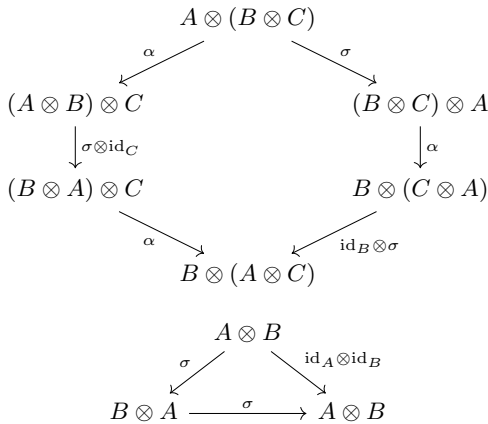
5.1 Monoidal Categories

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).

From a different perspective, Baez and Dolan (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 will be the case but, for that, we need categories that are much richer than just the symmetric monoidal categories which are only categorifications of commutative monoids. We will need to consider the categorification of commutative semirings.

5.2 Symmetric Rig Groupoids

The categorification of a commutative semiring is called a *symmetric rig category*. It is built 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 an isomorphism, the category will also be a groupoid.

There are several equivalent definitions of rig categories. We use the following definition from the ncatlab pages (<http://ncatlab.org/nlab/show/rig+category>).

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:

$$\begin{aligned}
a_\ell : x \otimes 0 &\xrightarrow{\sim} 0 \\
a_r : 0 \otimes x &\xrightarrow{\sim} 0
\end{aligned}$$

satisfying coherence conditions worked out by Laplaza (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.

The coherence conditions for rig categories were first worked out by Laplaza (Laplaza 1972). Pages 31-35 of his paper report 24

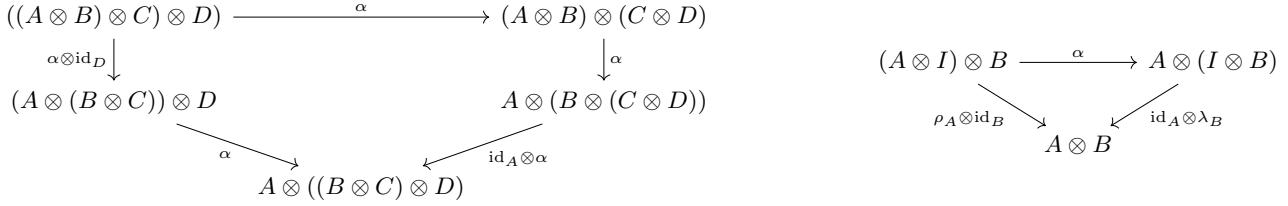


Figure 2. Pentagon and Triangle Diagrams

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. (See Fig. 3 for the high-level proof term showing that type and type equivalences form a symmetric rig category.)

5.3 Instances of Symmetric Rig Categories

As mentioned above we have the following theorem.

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. The high-level structure of that proof is in Fig. 3. \square

More relevant for 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. 4) and then to add a whole new layer of 2-combinators (discussed in the next section) witnessing enough equivalences of Π combinators to prove the coherence laws. 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. \square

Putting the result above together with Laplaza's coherence result about rig categories, we conclude our main result.

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

[This is weak; I could get away with it in a short talk, but this should be made more precise. It is essentially what the code says, and what is encoded by the natural transformations, but there has to be a better way to say this. —JC] The general structure explaining the nature of the second level metalanguage and how the two languages fit together from a categorical perspective is the following:

- type equivalences for example between $A \oplus B$ and $B \oplus A$ are functors;
- equivalences between such functors are *natural isomorphisms*;
- at the value level, these equivalences induce 2-morphisms.

For example, assuming we are given two Π -combinators:

$$c_1 : \{B C : \mathbf{U}\} \rightarrow B \leftrightarrow C$$

$$c_2 : \{A D : \mathbf{U}\} \rightarrow A \leftrightarrow D$$

from which we build two larger combinators p_1 and p_2 below:

$$p_1 p_2 : \{A B C D : \mathbf{U}\} \rightarrow \text{PLUS } A B \leftrightarrow \text{PLUS } C D$$

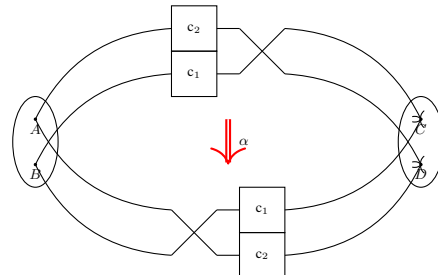
$$p_1 = \text{swap}_+ \odot (c_1 \oplus c_2)$$

$$p_2 = (c_2 \oplus c_1) \odot \text{swap}_+$$

As reversible circuits, p_1 and p_2 evaluate as follows. If p_1 is given the value $\text{inl } a$, it first transforms it to $\text{inr } a$, and then passes it to c_2 . If p_2 is given the value $\text{inl } a$, 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:

$$\text{swapl}_+ \Leftrightarrow : \{t_1 t_2 t_3 t_4 : \mathbf{U}\} \{c_1 : t_1 \leftrightarrow t_2\} \{c_2 : t_3 \leftrightarrow t_4\} \rightarrow (\text{swap}_+ \odot (c_1 \oplus c_2)) \Leftrightarrow ((c_2 \oplus c_1) \odot \text{swap}_+)$$

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:



```

TypeEquivCat : Category (Level.suc Level.zero) Level.zero Level.zero - omitted

TypeEquivGroupoid : Groupoid TypeEquivCat
TypeEquivGroupoid = record
  {  $\_^{-1}$  = sym $\simeq$ 
  ; iso =  $\lambda \_ \{ \_ \} \{ \_ \} \{ f, mkqinv \ g \ \alpha \ \beta \} \rightarrow$  record {
      iso $^l$  = eq  $\beta \ \beta$ 
      ; iso $^r$  = eq  $\alpha \ \alpha$  } } }

CPM $\oplus$  : Monoidal TypeEquivCat - omitted
CPM $\times$  : Monoidal TypeEquivCat - omitted

BM $\oplus$  : Braided CPM $\oplus$  - omitted
BM $\times$  : Braided CPM $\times$  - omitted

SBM $\oplus$  : Symmetric BM $\oplus$ 
SBM $\oplus$  = record { symmetry = eq swapswap $_+$  swapswap $_+$  }

SBM $\times$  : Symmetric BM $\times$ 
SBM $\times$  = record { symmetry = eq swapswap $\star$  swapswap $\star$  }

TERig : RigCategory SBM $\oplus$  SBM $\times$ 
TERig = record
  { distrib $_l$  =  $x \otimes [y \oplus z] \equiv [x \otimes y] \oplus [x \otimes z]$ 
  ; distrib $_r$  =  $[x \oplus y] \otimes z \equiv [x \otimes z] \oplus [y \otimes z]$ 
  ; ann $_l$  =  $0 \otimes x \equiv 0$ 
  ; ann $_r$  =  $x \otimes 0 \equiv 0$ 
  ; laplazaI = eq distl-swap $_+$ -lemma factorl-swap $_+$ -lemma
  ; laplazaII = eq dist-swap $\star$ -lemma factor-swap $\star$ -lemma
  ; laplazaIV = eq dist-dist-assoc-lemma assoc-factor-factor-lemma
  ; laplazaVI = eq distl-assoc-lemma assoc-factorl-lemma
  ; laplazaIX = eq fully-distribute fully-factor
  ; laplazaX = eq distz0 $\equiv$ distrz0 factorz0 $\equiv$ factorzr0
  ; laplazaXI = eq distz0 $\equiv$ unite $_+$  $\circ$ [distz,distz] $\circ$ distl factorz0 $\equiv$ factorl $\circ$ [factorz,factorz] $\circ$ uniti $_+$ 
  ; laplazaXIII = eq unite $\star$ r0 $\equiv$ absorb1 uniti $\star$ r0 $\equiv$ factorz
  ; laplazaXV = eq absorb1 $\equiv$ absorb $\circ$ swap $\star$  factorzr $\equiv$ swap $\star$  $\circ$ factorz
  ; laplazaXVI = eq absorbr $\Leftrightarrow$ assocl $\star$  $\circ$ [absorbr $\otimes$ id] $\circ$ absorbr factorz $\Leftrightarrow$ factorz $\circ$ [factorz $\otimes$ id] $\circ$ assocr $\star$ 
  ; laplazaXVII = eq elim-middle- $\perp$  insert-middle- $\perp$ 
  ; laplazaXIX = eq elim $\perp$ -A[0 $\oplus$ B] insert $\oplus$  $\perp$ -AB
  ; laplazaXXIII = eq elim T-1[A $\oplus$ B] insert T1 $\otimes$ -A $\oplus$ B
  }

```

Figure 3. Symmetric Rig Groupoid of Type Equivalences

6.2 Revised Syntax

[The second sentence below is a repeat; where does it belong? —JC] [You’re missing 4 combinators from that picture! Plus the ones you added are the ones for absorption and distributivity, NOT the ones which are redundant, they are really needed for a proper semring. —JC]

The inspiration of symmetric rig groupoids suggested a refactoring of II with additional level-1 combinators. The added combinators 4 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. 5. To avoid clutter we omit the names of the combinators and only show the signatures.

[I suggest adding the following laplaza combinators to that table: I, IV, IX (sorry), X and XVII. Also a short paragraph explaining what they do. For some of them, I suggest digging into Data.SumProd.Properties and showing some of these too (just as signatures). The later ones are much better. Anything with a proj in the statement is "wrong" (it builds in some premature beta-redexes in the statement), so don’t pick those. But they are theorems which are not in Agda’s standard library. Most of the properties of sums and products (aka coherences) were already there, but not these, which are interesting interactions between them. —JC]

As Fig. 5 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

$identls_+ :$	$\tau + 0 \leftrightarrow \tau$	$: identrs_+$
$identls_* :$	$\tau * 1 \leftrightarrow \tau$	$: identrs_*$
$absorbr_0 :$	$0 * \tau \leftrightarrow 0$	$: factorl_0$
$absorbl_0 :$	$\tau * 0 \leftrightarrow 0$	$: factort_0$
$distl :$	$\tau_1 * (\tau_2 + \tau_3) \leftrightarrow (\tau_1 * \tau_2) + (\tau_1 * \tau_3)$	$: factorl$

Figure 4. Additional Π -combinators

many them 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 (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.

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 $\boxed{\tau_1 \tau_2}$ 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 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 (James and Sabry 2012a). We then extend Π with a new universe of types \mathbb{T} that consists of composite types $\boxed{\tau_1 \tau_2}$:

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

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.

[should define ; here too —JC] Our next step is to define lifted versions of the 0d types:

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

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 defini-

tions above can be intuitively understood as arithmetic identities. The same arithmetic intuition explains the lifting of isomorphisms to 1d types:

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

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.

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

$$\begin{aligned}
id &: \mathbb{T} \Leftrightarrow \mathbb{T} \\
&: \boxed{\tau_1 \tau_2} \Leftrightarrow \boxed{\tau_1 \tau_2} \\
&\triangleq (\tau_1 + \tau_2) \leftrightarrow (\tau_2 + \tau_1) \\
id &= swap_+ \\
identl_+ &: 0 \boxplus \mathbb{T} \Leftrightarrow \mathbb{T} \\
&= assocr_+ \circ (id \oplus swap_+) \circ assocr_+
\end{aligned}$$

[are you sure about identl? The combinator for this in PiWith-Level/Pi1.agda is longer and (not surprisingly) uses unite and uniti. —JC]

Composition using trace. [assoc1 and assoc2 do not exist, should rename —JC]

$$\begin{aligned}
(\circ) &: (\mathbb{T}_1 \Leftrightarrow \mathbb{T}_2) \rightarrow (\mathbb{T}_2 \Leftrightarrow \mathbb{T}_3) \rightarrow (\mathbb{T}_1 \Leftrightarrow \mathbb{T}_3) \\
f \circ g &= trace (assoc_1 \circ (f \oplus id) \circ assoc_2 \circ (g \oplus id) \circ assoc_3)
\end{aligned}$$

Higher-order functions.

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

$$\begin{aligned}
flip &: (\mathbb{T}_1 \Leftrightarrow \mathbb{T}_2) \rightarrow (\boxminus \mathbb{T}_2 \Leftrightarrow \boxminus \mathbb{T}_1) \\
flip f &= swap_+ \circ f \circ swap_+
\end{aligned}$$

$$\begin{aligned}
curry &: ((\mathbb{T}_1 \boxplus \mathbb{T}_2) \Leftrightarrow \mathbb{T}_3) \rightarrow (\mathbb{T}_1 \Leftrightarrow (\mathbb{T}_2 \multimap \mathbb{T}_3)) \\
curry f &= assocr_+ \circ f \circ assocr_+
\end{aligned}$$

$$\begin{aligned}
uncurry &: (\mathbb{T}_1 \Leftrightarrow (\mathbb{T}_2 \multimap \mathbb{T}_3)) \rightarrow ((\mathbb{T}_1 \boxplus \mathbb{T}_2) \Leftrightarrow \mathbb{T}_3) \\
uncurry f &= assocr_+ \circ f \circ assocr_+
\end{aligned}$$

Products. The definition for the product of 1d types used above is:

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

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

$$\begin{array}{lcl}
c_1 \circ (c_2 \circ c_3) & \Leftrightarrow & (c_1 \circ c_2) \circ c_3 \\
id \circ c & \Leftrightarrow & c \\
c \circ id & \Leftrightarrow & c \\
c \circ (!c) & \Leftrightarrow & id \\
(!c) \circ c & \Leftrightarrow & id \\
\\
identl_+ \circ c_2 & \Leftrightarrow & (c_1 \oplus c_2) \circ identl_+ \\
identr_+ \circ (c_1 \oplus c_2) & \Leftrightarrow & c_2 \circ identr_+ \\
identls_+ \circ c_2 & \Leftrightarrow & (c_2 \oplus c_1) \circ identls_+ \\
identrs_+ \circ (c_2 \oplus c_1) & \Leftrightarrow & c_2 \circ identrs_+ \\
swap_+ \circ (c_1 \oplus c_2) & \Leftrightarrow & (c_2 \oplus c_1) \circ swap_+ \\
\\
identl_* \circ c_2 & \Leftrightarrow & (c_1 \otimes c_2) \circ identl_* \\
identr_* \circ (c_1 \otimes c_2) & \Leftrightarrow & c_2 \circ identr_* \\
identls_* \circ c_2 & \Leftrightarrow & (c_2 \otimes c_1) \circ identls_* \\
identrs_* \circ (c_2 \otimes c_1) & \Leftrightarrow & c_2 \circ identrs_* \\
swap_* \circ (c_1 \otimes c_2) & \Leftrightarrow & (c_2 \otimes c_1) \circ swap_* \\
\\
swap_+ \circ factor & \Leftrightarrow & factor \circ (swap_+ \otimes id) \\
absorbr_0 & \Leftrightarrow & absorbl_0 \\
identl_* & \Leftrightarrow & absorbr_0 \\
absorbl_0 & \Leftrightarrow & swap_* \circ absorbr_0 \\
absorbr_0 & \Leftrightarrow & (assocl_* \circ (absorbr_0 \otimes id)) \circ absorbr_0 \\
absorbr_0 & \Leftrightarrow & (distl \circ (absorbr_0 \oplus absorbr_0)) \circ identl_+ \\
(id \otimes absorbr_0) \circ absorbl_0 & \Leftrightarrow & (assocl_* \circ (absorbl_0 \otimes id)) \circ absorbr_0 \\
(id \otimes identl_+) & \Leftrightarrow & (distl \circ (absorbl_0 \oplus id)) \circ identl_+ \\
identl_+ & \Leftrightarrow & distl \circ (identl_+ \oplus identl_+) \\
(distl \circ (dist \oplus dist)) \circ assocl_+ & \Leftrightarrow & (((dist \circ (distl \oplus distl)) \circ assocl_+) \circ (assocr_+ \oplus id)) \circ (id \oplus swap_+ \oplus id) \circ (assocl_+ \oplus id) \\
\\
(id \otimes swap_+) \circ distl & \Leftrightarrow & distl \circ swap_+ \\
dist \circ (swap_* \oplus swap_*) & \Leftrightarrow & swap_* \circ distl \\
((assocl_+ \otimes id) \circ dist) \circ (dist \oplus id) & \Leftrightarrow & (dist \circ (id \oplus dist)) \circ assocl_+ \\
assocl_* \circ distl & \Leftrightarrow & ((id \otimes distl) \circ distl) \circ (assocl_* \oplus assocl_*) \\
\\
(c_1 \oplus (c_2 \oplus c_3)) \circ assocl_+ & \Leftrightarrow & assocl_+ \circ ((c_1 \oplus c_2) \oplus c_3) \\
((c_1 \oplus c_2) \oplus c_3) \circ assocr_+ & \Leftrightarrow & assocr_+ \circ (c_1 \oplus (c_2 \oplus c_3)) \\
(c_1 \otimes (c_2 \otimes c_3)) \circ assocl_* & \Leftrightarrow & assocl_* \circ ((c_1 \otimes c_2) \otimes c_3) \\
((c_1 \otimes c_2) \otimes c_3) \circ assocr_* & \Leftrightarrow & assocr_* \circ (c_1 \otimes (c_2 \otimes c_3)) \\
((a \oplus b) \otimes c) \circ dist & \Leftrightarrow & dist \circ ((a \otimes c) \oplus (b \otimes c)) \\
((a \otimes c) \oplus (b \otimes c)) \circ factor & \Leftrightarrow & factor \circ ((a \oplus b) \otimes c) \\
(a \otimes (b \oplus c)) \circ distl & \Leftrightarrow & distl \circ ((a \otimes b) \oplus (a \otimes c)) \\
((a \otimes b) \oplus (a \otimes c)) \circ factorl & \Leftrightarrow & factorl \circ (a \otimes (b \oplus c)) \\
\\
id \oplus id & \Leftrightarrow & id \\
id \otimes id & \Leftrightarrow & id \\
\\
(c_1 \circ c_3) \oplus (c_2 \circ c_4) & \Leftrightarrow & (c_1 \oplus c_2) \circ (c_3 \oplus c_4) \\
(c_1 \circ c_3) \otimes (c_2 \circ c_4) & \Leftrightarrow & (c_1 \otimes c_2) \circ (c_3 \otimes c_4) \\
\\
identls_+ \oplus id & \Leftrightarrow & assocr_+ \circ (id \oplus identl_+) \\
identls_* \otimes id & \Leftrightarrow & assocr_* \circ (id \oplus identl_*) \\
\\
assocr_+ \circ assocr_+ & \Leftrightarrow & ((assocr_+ \oplus id) \circ assocr_+) \circ (id \oplus assocr_+) \\
assocr_* \circ assocr_* & \Leftrightarrow & ((assocr_* \otimes id) \circ assocr_*) \circ (id \otimes assocr_*) \\
(assocr_+ \circ swap_+) \circ assocr_+ & \Leftrightarrow & ((swap_+ \oplus id) \circ assocr_+) \circ (id \oplus swap_+) \\
(assocl_+ \circ swap_+) \circ assocl_+ & \Leftrightarrow & ((id \oplus swap_+) \circ assocl_+) \circ (swap_+ \oplus id) \\
(assocr_* \circ swap_*) \circ assocr_* & \Leftrightarrow & ((swap_* \oplus id) \circ assocr_*) \circ (id \otimes swap_*) \\
(assocl_* \circ swap_*) \circ assocl_* & \Leftrightarrow & ((id \otimes swap_*) \circ assocl_*) \circ (swap_* \otimes id) \\
\\
(c \otimes id) \circ absorbl_0 & \Leftrightarrow & absorbl_0 \circ id \\
(id \otimes c) \circ absorbr_0 & \Leftrightarrow & absorbr_0 \circ id \\
id \circ factorl_0 & \Leftrightarrow & factorl_0 \circ (id \otimes c) \\
id \circ factorr_0 & \Leftrightarrow & factorr_0 \circ (c \otimes id) \\
\\
\frac{}{\vdash c \Leftrightarrow c} \quad \frac{\vdash c_1 \Leftrightarrow c_2 \quad \vdash c_2 \Leftrightarrow c_3}{\vdash c_1 \Leftrightarrow c_3} \quad \frac{\vdash c_1 \Leftrightarrow c_3 \quad \vdash c_2 \Leftrightarrow c_4}{\vdash (c_1 \circ c_2) \Leftrightarrow (c_3 \circ c_4)} \quad \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)} \quad 10 \\
\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)}
\end{array}$$

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:

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

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 than 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. **[I would (syntactically) highlight that the Int construction does NOT allow one to lift multiplication in a straightforward manner. Put it in a “box” or something? —JC]**

8. Conclusion

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

We start with the class of all functions $A \rightarrow B$, 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 the 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.

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.
- 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-i-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.
- 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.

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

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](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>.

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 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 $a, b, c \in R$:

$$\begin{aligned} 0 + a &= a \\ a + b &= b + a \\ a + (b + c) &= (a + b) + c \\ 1 \cdot a &= a \\ a \cdot b &= b \cdot a \\ a \cdot (b \cdot c) &= (a \cdot b) \cdot c \\ 0 \cdot a &= 0 \\ (a + b) \cdot c &= (a \cdot c) + (b \cdot c) \end{aligned}$$

B. Diagrammatic Optimization

