Representing, Manipulating and Optimizing Reversible Circuits

Jacques Carette
McMaster University
carette@mcmaster.ca

Amr Sabry
Indiana University
sabry@indiana.edu

Abstract

We show how a typed set of combinators for reversible computations, corresponding exactly to the semiring of permutations, is a convenient basis for representing and manipulating reversible circuits. A categorical interpretation also leads to optimization combinators, and we demonstrate their utility through an example.

1. Introduction

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

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 HoTT equivalence of types. We then connect this notion to several semiring structures on finite types with the goal of reducing the notion of finite type equivalence to a notion of reversible computation.

2.1 HoTT Equivalences 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 (Equivalence of types)**.** *Two types A and B are equivalent $A \simeq B$ if there exists a* bi-invertible *$f : A \to B$, i.e., if there exists an f that has both a left-inverse and a right-inverse. A function $f : A \to$ has a left-inverse if there exists a function $g : B \to A$ such that $g \circ f = \mathrm{id}_A$. A function $f : A \to$ has a right-inverse if there exists a function $g : B \to A$ such that $f \circ g = \mathrm{id}_B$.*

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

2.2 Instance I: Universe of Types

The first commutative semiring instance we examine is the universe of types (Set in Agda terminology). (See Appendix A for the definition of commutative rings.) The additive unit is the empty type $\bot$; the multiplicative unit is the unit type $\top$; the two binary operations are disjoint union $\uplus$ and cartesian product $\times$. The axioms are satisfied up to equivalence of types $\simeq$. For example, we have equivalences such as:

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

Formally we have the following fact.

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

2.3 Instance II: Finite Sets

The collection of all finite sets (Fin m for natural number m in Agda terminology) is another commutative semiring instance. In this case, the additive unit is Fin 0, the multiplicative unit is Fin 1, the two binary operations are still disjoint union $\uplus$ and cartesian product $\times$, and the axioms are also satisfied up to equivalence of types $\simeq$.

The reason finite sets are interesting is that each finite type A constructed from $\bot$, $\top$, $\uplus$, and $\times$ is equivalent (in $|A|$! ways) to Fin $|A|$ where $|A|$ is the size of A defined as follows:

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

Each of the $|A|$! equivalences of A with Fin $|A|$ corresponds to a *particular* enumeration of the elements of A. For example, we have two equivalences:

$$\top \uplus \top \quad \simeq \quad \mathsf{Fin}\ 2$$

corresponding to the identity and boolean negation.

Thus, as we prove next, up to equivalence, the only interesting property of a finite type is its size. In other words, 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 Fin 6 and use the latter for further reasoning. Indeed, as the next section demonstrate, this result allows us to characterize equivalences between finite types in a canonical way as permutations between finite sets.

The following theorem precisely characterizes the relationship between finite types and finite sets.

Theorem 2. *If $A \simeq$ Fin m, $B \simeq$ 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 Fin $(suc\ m')$ and Fin $(suc\ n')$, implies an equivalence between Fin m' and 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 Fin $(suc\ m')$ and analyze every position this element could be mapped to by the larger equivalence and in each case construct an equivalence that excludes this element. □

In the remainder of the paper, we will refer to the type of all equivalences between types A and B as EQ_{AB}. As explained above, this type is inhabited only if $|A| = |B|$ in which case it has $|A|$! elements witnessing the various ways in which we can have $A \simeq B$. We note that this type of all equivalences is itself a commutative semiring with the additive unit being the vacuous equivalence $\bot \simeq \bot$, the multiplicative unit being the trivial equivalence $\top \simeq \top$, the two binary operations essentially map $\uplus$ and $\times$ over equivalences, and the axioms are satisfied up to extensional equality of the functions underlying the equivalences.

Theorem 3. *The collection of all equivalences EQ_{AB} for finite types A and B forms a commutative semiring.*

2.4 Permutations on Finite Sets

Given the correspondence between finite types and finite sets, we will prove that equivalences on finite types are equivalent to permutations on finite sets. Formalizing the notion of permutations is delicate however: straightforward attempts turn out not to capture enough of the properties of permutations for our purposes. We therefore formalize a permutation using two sizes: m for the size of the input finite set and n 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 permutations as follows. A permutation CPerm m n consists of four components. The first two components are:

- a vector of size n containing elements drawn from the finite set Fin m;
- a dual vector of size m containing elements drawn from the finite set 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.

In the remainder of the paper, we will refer to the type of all permutations between finite sets $\mathsf{Fin}\ m$ and $\mathsf{Fin}\ n$ as PERM_{mn}. This type is only inhabited if $m = n$ in which case it has $m!$ elements, each of which witnesses one of the possible permutations $\mathsf{CPerm}\ m\ n$. We note that this type of all permutations is itself a commutative semiring with the additive unit being the vacuous permutations $\mathsf{CPerm}\ 0\ 0$, the multiplicative unit being the trivial permutations $\mathsf{CPerm}\ 1\ 1$, the two binary operations essentially map $\uplus$ and $\times$ over permutations, and the axioms are satisfied up to strict equality of the vectors underlying the permutations.

Theorem 4. *The collection of all permutations* PERM_{mn} *for natural numbers* m *and* n *forms a commutative semiring.*

2.5 Equivalences of Equivalences

The main result of this section is that the type of all equivalences between finite types A and B, EQ_{AB}, is equivalent to the type of all permutations PERM_{mn} where $m = |A|$ and $n = |B|$.

Theorem 5. *If* $A \simeq \mathsf{Fin}\ m$ *and* $B \simeq \mathsf{Fin}\ n$*, then the type of all equivalences* EQ_{AB} *is equivalent to the type of all permutations* $\text{PERM}\ m\ n$*.*

Proof. Although long and tedious, this proof is straightforward. □

With the proper Agda definitions, we can rephrase this theorem in a more evocative way. We will discuss the relevance of this theorem to the *univalence* postulate in the conclusion.

Theorem 6.

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

To summarize the result of this section: if we are interested in studying type equivalences, up to equivalence, it suffices to study permutations on finite sets. This will prove quite handy as, unlike the former, the latter notion can be inductively defined which gives it a natural computational interpretation.

Before concluding this section, we recall that both the type of all equivalences and the type of all permutations are commutative semirings and in fact the previous theorem can be generalized to a stronger theorem asserting that these two commutative semiring structures are *isomorphic*.

Theorem 7. *The equivalence of Theorem 5 is an* isomorphism *between the commutative semiring of equivalences of finite types and the commutative semiring of permutations.*

3. Typed Isormorphisms

In the previous section, we argued that, up to equivalence, the equivalence of types reduces to permutations on finite sets. The former notion relies on function equivalence and cannot be defined inductively. The second notion is easy to define in a computational framework but is too level from a programmer perspective. We propose a middle ground: a computational framework for expressing, computing, and optimizing equivalences between finite types. We will then relate this calculus to equivalences on one hand and to permutations on the other hand.

3.1 Pi

We introduce a simple language called Π whose only computations are isomorphisms between finite types [**?**].

The Π family of languages is based on type isomorphisms. In the variant we consider, the set of types τ includes 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.

3.2 Soundness

Show that pi combinators are valid equivalences

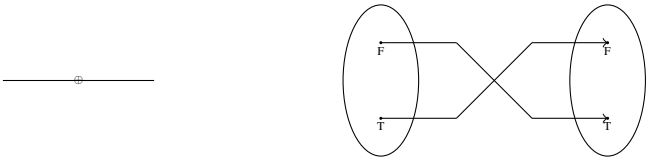
Show that pi combinators are also valid permutations on finite sets

We don't show completeness but pi combinators are designed by *reifying* the fundamental "proof rules" of semirings as combinators. Also they correspond to the type isos that Fiore et al prove are sound and complete for finite types. So they are canonical enough.

This set of isomorphisms is known to be complete [**? ?**] and the language is universal for hardware combinational circuits [**?**].[2]

From the perspective of category theory, the language Π models what is called a *symmetric bimonoidal category* or a *commutative rig category*. These are categories with two binary operations and satisfying the axioms of a commutative rig (i.e., a commutative ring without negative elements also known as a commutative semiring) up to coherent isomorphisms. And indeed the types of the Π-combinators are precisely the commutative semiring axioms. A formal way of saying this is that Π is the *categorification* [**?**] of the natural numbers. A simple (slightly degenerate) example of such categories is the category of finite sets and permutations in which we interpret every Π-type as a finite set, interpret the values as elements in these finite sets, and interpret the combinators as permutations.

4. Example Circuit: Simple Negation




```
BOOL : U
BOOL = PLUS ONE ONE

n1 : BOOL ⟷ BOOL
n1 = swap+
```

Example Circuit: Not So Simple Negation.

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

[2] If recursive types and a trace operator are added, the language becomes Turing complete [**? ?**]. We will not be concerned with this extension in the main body of this paper but it will be briefly discussed in the conclusion.[but don't we need trace for the Int construction? —JC]

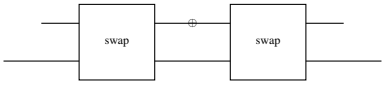
$$
\begin{array}{rrcll}
identl_+ : & 0+\tau & \leftrightarrow & \tau & : identr_+ \\
swap_+ : & \tau_1+\tau_2 & \leftrightarrow & \tau_2+\tau_1 & : swap_+ \\
assocl_+ : & \tau_1+(\tau_2+\tau_3) & \leftrightarrow & (\tau_1+\tau_2)+\tau_3 & : assocr_+ \\
identl_* : & 1*\tau & \leftrightarrow & \tau & : identr_* \\
swap_* : & \tau_1*\tau_2 & \leftrightarrow & \tau_2*\tau_1 & : swap_* \\
assocl_* : & \tau_1*(\tau_2*\tau_3) & \leftrightarrow & (\tau_1*\tau_2)*\tau_3 & : assocr_* \\
dist_0 : & 0*\tau & \leftrightarrow & 0 & : factor_0 \\
dist : & (\tau_1+\tau_2)*\tau_3 & \leftrightarrow & (\tau_1*\tau_3)+(\tau_2*\tau_3) & : factor
\end{array}
$$

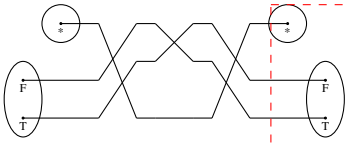
$$
\frac{}{\vdash id : \tau \leftrightarrow \tau} \qquad \frac{\vdash c_1 : \tau_1 \leftrightarrow \tau_2 \quad \vdash c_2 : \tau_2 \leftrightarrow \tau_3}{\vdash c_1 \fatsemi c_2 : \tau_1 \leftrightarrow \tau_3}
$$

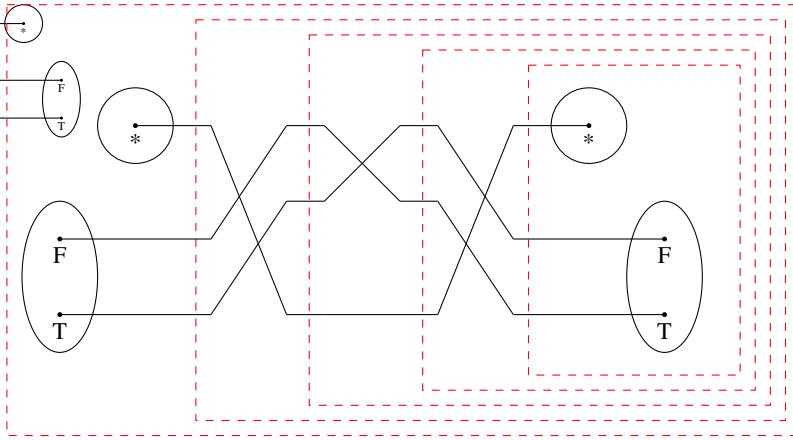
$$
\frac{\vdash c_1 : \tau_1 \leftrightarrow \tau_2 \quad \vdash c_2 : \tau_3 \leftrightarrow \tau_4}{\vdash c_1 \oplus c_2 : \tau_1+\tau_3 \leftrightarrow \tau_2+\tau_4}
$$

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

Figure 1. Π-combinators [**?**]







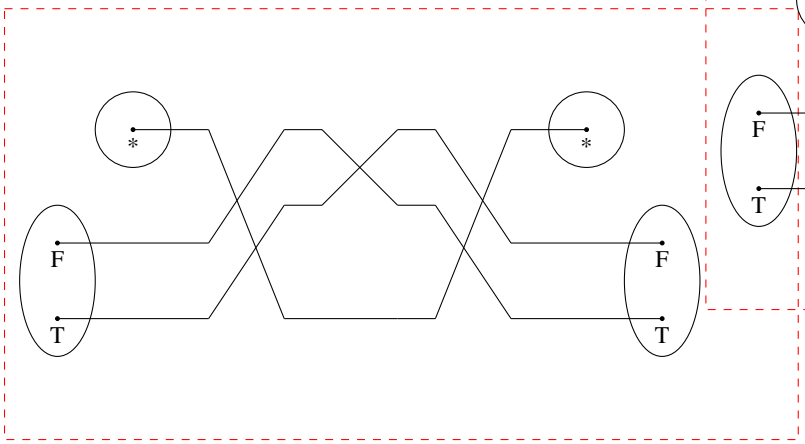

```
n2 : BOOL ⟷ BOOL
n2 =  uniti⋆ ⊙
      swap⋆ ⊙
      (swap+ ⊗ id⟷) ⊙
      swap⋆ ⊙
      unite⋆
```

Reasoning about Example Circuits. Algebraic manipulation of one circuit to the other:

```
negEx : n2 ⇔ n1
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+ □
```

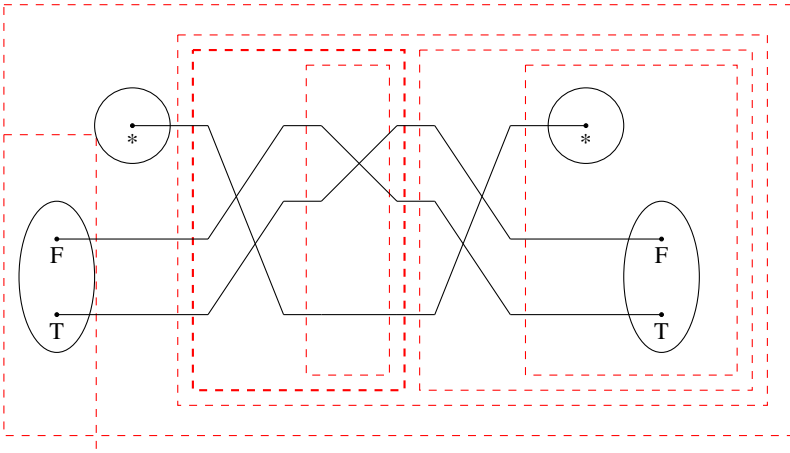
Visually.

Original circuit:

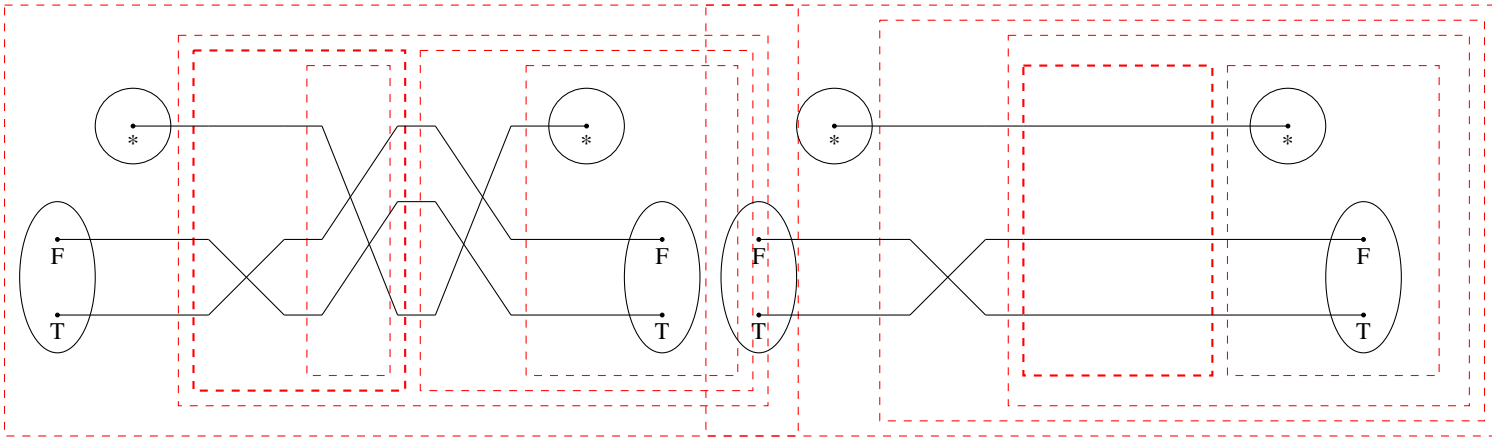


Making grouping explicit:

By associativity:

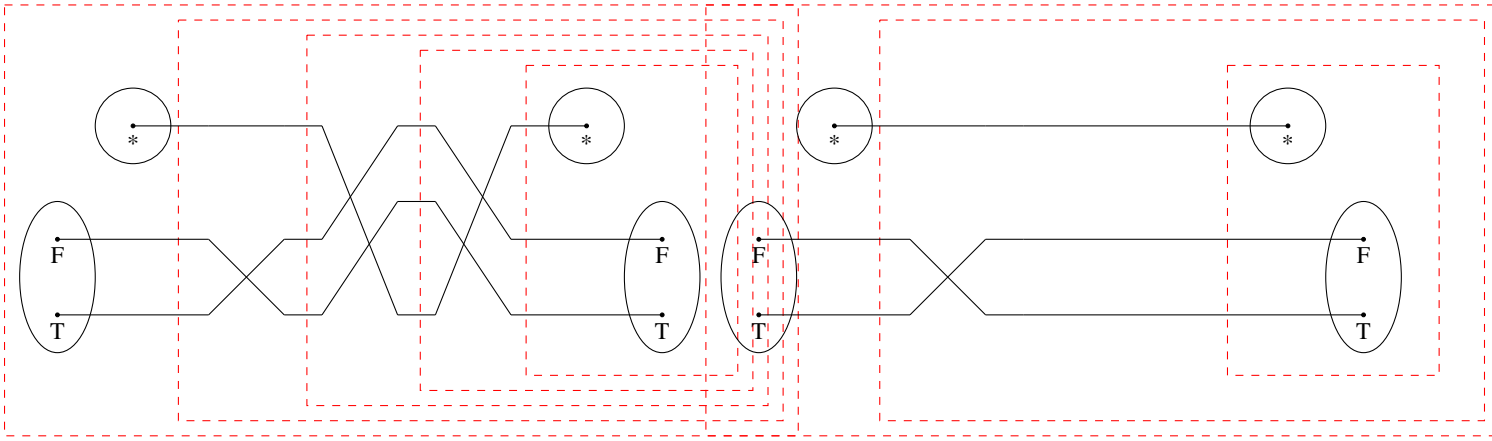


By pre-post-swap:



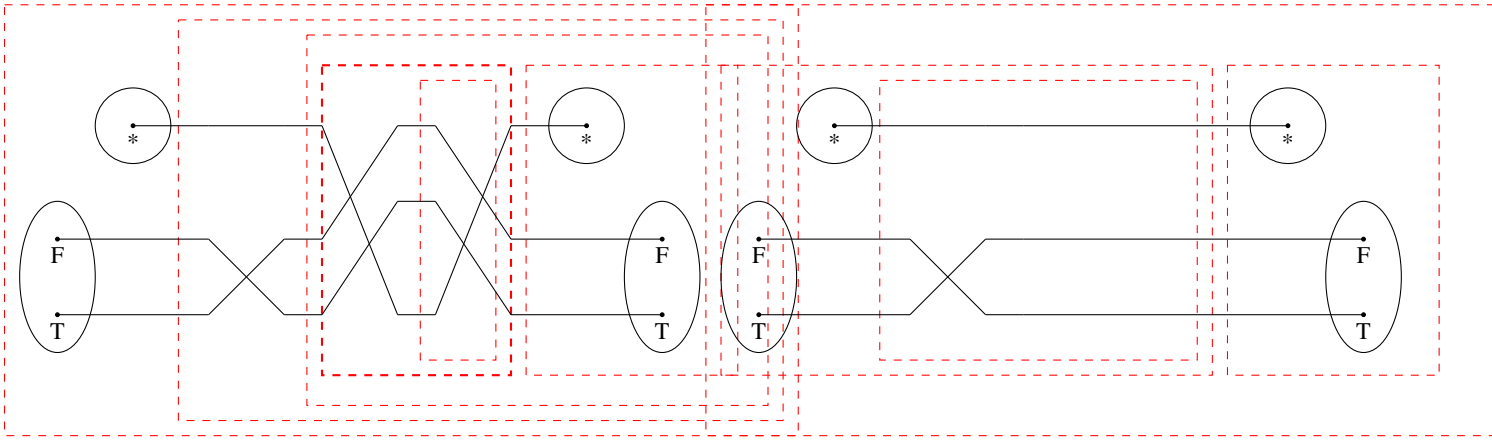
By associativity:

By id-compose-left:



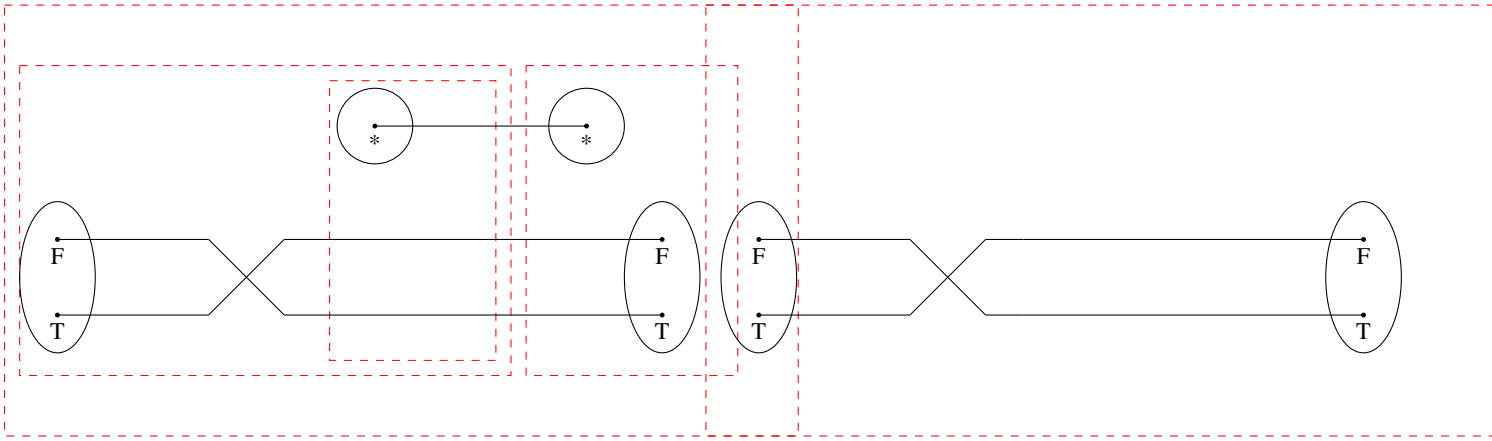
By associativity:

By associativity:

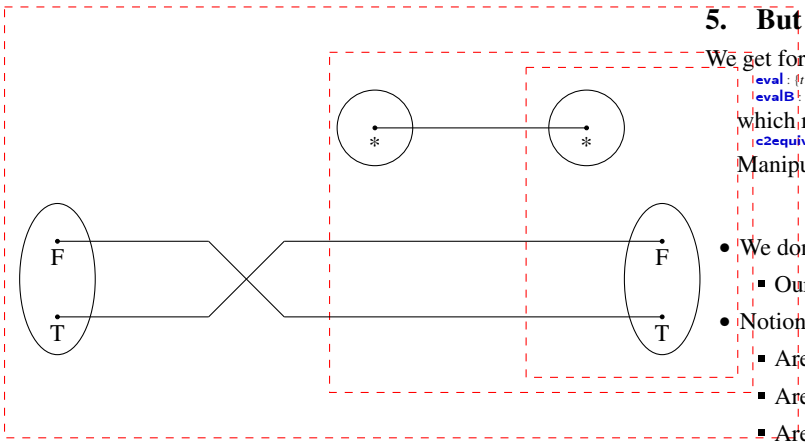


By swap-swap:

By swap-unit:



By associativity:



By unit-unit:

5. But is this a programming language?

We get forward and backward evaluators

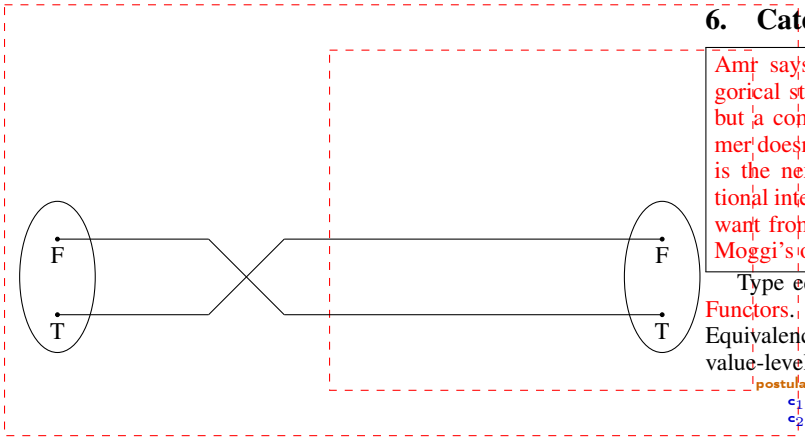
```
eval : {t1 t2 : U} → (t1 ⟷ t2) → ⟦ t1 ⟧ → ⟦ t2 ⟧
evalB : {t1 t2 : U} → (t1 ⟷ t2) → ⟦ t2 ⟧ → ⟦ t1 ⟧
```

which really do behave as expected

```
c2equiv : {t1 t2 : U} → (c : t1 ⟷ t2) → ⟦ t1 ⟧ ≃ ⟦ t2 ⟧
```

Manipulating circuits. Nice framework, but:

- We don't want ad hoc rewriting rules.
  - Our current set has 76 rules!
- Notions of soundness; completeness; canonicity in some sense.
  - Are all the rules valid? (yes)
  - Are they enough? (next topic)
  - Are there canonical representations of circuits? (open)



By id-unit-right:

6. Categorification I

Amr says: 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).

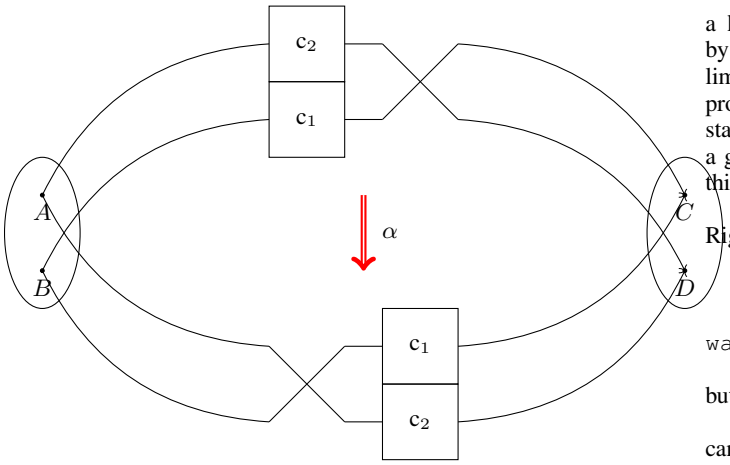
Type equivalences (such as between $A \times B$ and $B \times A$) are Functors.

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 8. *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 9 (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.*

7. Emails

Reminder of `http://mathoverflow.net/questions/106070/int-construction-traced-monoidal-categories-and-grothendieck-group`

Also, `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.163.334` seems relevant

I had checked and found no traced categories or Int constructions in the categories library. I'll think about that and see how best to proceed.

The story without trace and without the Int construction is boring as a PL story but not hopeless from a more general perspective.

I don't know, that a "symmetric rig" (never mind higher up) is a programming language, even if only for "straight line programs" is interesting! ;)

But it really does depend on the venue you'd like to send this to. If POPL, then I agree, we need the Int construction. The more generic that can be made, the better.

It might be in 'categories' already! Have you looked?

In the meantime, I will try to finish the Rig part. Those coherence conditions are non-trivial.

I am thinking that our story can only be compelling if we have a hint that h.o. functions might work. We can make that case by "just" implementing the Int Construction and showing that a limited notion of h.o. functions emerges and leave the big open problem of high to get the multiplication etc. for later work. I can start working on that: will require adding traced categories and then a generic Int Construction in the categories library. What do you think?

I have the braiding, and symmetric structures done. Most of the RigCategory as well, but very close.

Of course, we're still missing the coherence conditions for Rig.

Can you make sense of how this relates to us?

`https://pigworker.wordpress.com/2015/04/01/warming-up-to-homotopy-type-theory/`

Unfortunately not. Yes, there is a general feeling of relatedness, but I can't pin it down.

I do believe that all our terms have computational rules, so we can't get stuck.

Note that at level 1, we have equivalences between Perm(A,B) and Perm(A,B), not Perm(C,D) [look at the signature of <=>]. That said, we can probably use a combination of levels 0 and 1 to get there.

Yes, we should dig into the Licata/Harper work and adapt to our setting.

Though I think we have some short-term work that we simply need to do to ensure our work will rest on a solid basis. If that crumbles, all the rest of the edifice will too.

Trying to get a handle on what we can transport or more precisely if we can transport things that HoTT can only do with univalence.

(I use permutation for level 0 to avoid too many uses of 'equivalence' which gets confusing.)

Level 0: Given two types A and B, if we have a permutation between them then we can transport something of type P(A) to something of type P(B).

For example: take P = . + C; we can build a permutation between A+C and B+C from the given permutation between A and B

Level 1: Given types A, B, C, and D. let Perm(A,B) be the type of permutations between A and B and Perm(C,D) be the type of permutations between C and D. If we have a 2d-path between Perm(A,B) and Perm(C,D) then we can transport something of type P(Perm(A,B)) to something of type P(Perm(C,D)).

This is more interesting. What's a good example though of a property P that we can implement?

In think that in HoTT the only way to do this transport is via univalence. First you find an equivalence between the spaces of permutations, then you use univalence to postulate the existence of a path, and then you transport along that path. Is that right?

In HoTT this is exhibited by the failure of canonicity: closed terms that are stuck. We can't get closed terms that are stuck: we don't have any axioms with no computational rules, right?

Perhaps we can adapt the discussion/example in `http://homotopytypetheory.org/2011/07/27/canonicity-for-2-dime` to our setting and build something executable?

I hope not! [only partly joking]

Actually, there is a fair bit about this that I dislike: it seems to over-simplify by arbitrarily saying some things are equal when they are not. They might be equivalent, but not equal.

This came up in a different context but looks like it might be useful to us too.

`http://arxiv.org/pdf/gr-qc/9905020`

Separate. The Grothendieck construction in this case is about fibrations, and is not actually related to the "Grothendieck Group" construction, which is related to the Int construction.

Yes. The categories library has a Grothendieck construction. Not sure if we can use that or if we need to define a separate Int construction?

Reminder of http://mathoverflow.net/questions/106070/int-construction-traced-monoidal-categories and paper on that.

Also, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.163.334 seems relevant

I had checked and found no traced categories or Int constructions in the categories library. I'll think about that and see how best to proceed.

The story without trace and without the Int construction is boring as a PL story but not hopeless from a more general perspective.

I don't know, that a "symmetric rig" (never mind higher up) is a programming language, even if only for "straight line programs" is interesting! ;)

But it really does depend on the venue you'd like to send this to. If POPL, then I agree, we need the Int construction. The more generic that can be made, the better.

It might be in 'categories' already! Have you looked?

In the meantime, I will try to finish the Rig part. Those coherence conditions are non-trivial.

I am thinking that our story can only be compelling if we have a hint that h.o. functions might work. We can make that case by "just" implementing the Int Construction and showing that a limited notion of h.o. functions emerges and leave the big open problem of high to get the multiplication etc. for later work. I can start working on that: will require adding traced categories and then a generic Int Construction in the categories library. What do you think? —Amr

I have the braiding, and symmetric structures done. Most of the RigCategory as well, but very close.

Of course, we're still missing the coherence conditions for Rig.

solutions to quintic equations proof by arnold is all about hott... paths and higher degree path etc.

I thought we'd gotten at least one version, but could never prove it sound or complete.

Didn't we get stuck in the reverse direction. We never had it fully, or am I misremembering?

Right. We have one direction, from Pi combinators to FinVec permutations – c2perm in PiPerm.agda.

Note that quite a bit of the code has (already!!) bit-rotted. I changed the definition of PiLevel0 to make the categorical structure better, and that broke many things. I am in the process of fixing – which means introducing combinators all the way back in FinEquiv!!! I split the 0 absorption laws into a right and left law, and so have to do the right version; turns out they are non-trivial, because Agda only reduces the left law for free. Should be done this morning.

We do not have the other direction currently in the code. That may not be too bad, as we do have LeftCancellation to allow us to define things by recursion.

That's obsolete for now.

By the way, do we have a complement to thm2 that connects to Pi. Ideally what we want to say is what I started writing: thm2 gives us a semantic bridge between equivalences and FinVec permutations; we also need a bridge between FinVec permutations and Pi combinators, right?

Is that going somewhere, or is it an experiment that should be put into Obsolete/ ?

Thanks. I like that idea ;).

I have a bunch of things I need to do, so I won't really put my head into this until the weekend.

I understand the desire to not want to rely on the full coherence conditions. I also don't know how to really understand them until we've implemented all of them, and see what they actually say!

As I was trying really hard to come up with a single story, I am a little confused as to what "my" story seems to be... Can you give me your best recollection of what I seem to be pushing, and how that is different? Then I would gladly flesh it out for us to do a second paper on that.

Instead of discussing this over and over, I think it is clear that thm2 will be an important part of any story we will tell. So I think what I am going to start doing is to write an explanation of thm2 in a way that would be usable in a paper.

I wasn't too worried about the symmetric vs. non-symmetric notion of equivalence. The HoTT book has various equivalent definitions of equivalence and the one below is one of them.

I do recall the other discussion about extensionality. That discussion concluded with the idea that the strongest statement that can be made is that HoTT equivalence between finite *enumerated* types is equivalent to Vec-based permutations. This is thm2 and it is essentially univalence as we noted earlier. My concern however is what happens at the next level: once we start talking about equivalences between equivalences. We should be to use thm2 to say that this the same as talking about equivalences between Vec-based permutations, which as you noted earlier is equivalence of categories.

I just really want to avoid the full reliance on the coherence conditions. I also noted you have a different story and I am willing to go along with your story if you sketch a paper outline for say one of the conferences/workshops at http://cstheory.stackexchange.com/questions/7900/list-of-tcs-conferences-and-workshops

Did you see my "HoTT-agda" question on the Agda mailing list on March 11th, and Dan Licata's reply?

What you wrote reduces to our definition of *equivalence*, not permutation. To prove that equivalence, we would need funext – see my question of February 18th on the Agda mailing list.

Another way to think about it is that this is EXACTLY what thm2 provides: a proof that for finite A and B, equivalence between A and B (as below) is equivalent to permutations implemented as (Vect, Vect, pf, pf).

Now, we may want another representation of permutations which uses functions (qua bijections) internally instead of vectors. Then the answer to your question would be "yes", modulo the question/answer about which encoding of equivalence to use.

Thought a bit more about this. We need a little bridge from HoTT to our code and we're good to go I think.

In HoTT we have several notions of equivalence that are equivalent (in the technical sense). The one that seems easiest to work with is the following:

A $\simeq$ B if exists f : A $\to$ B such that: (exists g : B $\to$ A with g o f idA) X (exists h : B $\to$ A with f o h idB)

Does this definition reduce to our semantic notion of permutation if A and B are finite sets?

I'm ok with a HoTT bias, but concerned that our code does not really match that. But since we have no specific deadline, I guess taking a bit more time isn't too bad.

Since propositional equivalence is really HoTT equivalence too, then I am not too concerned about that side of things – our concrete permutations should be the same whether in HoTT or in raw Agda. Same with various notions of equivalence, especially since most of the code was lifted from a previous HoTT-based attempt at things.

I would certainly agree with the not-not-statement: using a notion of equivalence known to be incompatible with HoTT is not a good idea.

I think that I should start trying to write down a more technical story so that we can see how things fit together. I am biased towards a HoTT-related story which is what I started. If you think we should have a different initial bias let me know.

What is there is just one paragraph for now but it already opens a question: if we are pursuing that HoTT story we should be able to prove that the HoTT notion of equivalence when specialized to finite types reduces to permutations. That should be a strong foundation to the rest and the precise notion of permutation we get (parameterized by enumerations or not should help quite a bit).

More generally always keeping our notions of equivalences (at higher levels too) in sync with the HoTT definitions seems to be a good thing to do.

... and if these coherence conditions are really complete then it should be the case the two pi-combinators are equal iff their canonical forms are identical.

So to sum up we would get a nice language for expressing equivalences between finite types and a normalization process that transforms each equivalence to a canonical form. The latter yield a simple decision procedure for comparing equivalences.

Here is a nice idea: we need a canonical form for every pi-combinator. Our previous approach gave us something but it was hard to work with. I think we can use the coherence conditions to reach a canonical form by simply picking a convention that chooses one side of every commuting diagram. What do you think? —Amr

Indeed! Good idea.

However, it may not give us a normal form. This is because quite a few 'simplifications' require to use 'the other' side of a commuting diagram first, to expose a combination which simplifies. Think $(x.y^{-}1).(y.z)\ \ x.z$.

In other words, because we have associativity and commutativity, we need to deal with those specially. Diagram with sides not all the same length are easy to deal with.

However, I think it is not that bad: we can use the objects to help. We also had put the objects [aka types] in normal form before (i.e. 1 + (1 + (1 + ...))))). The good thing about that is that there are very few pi-combinators which preserve that shape, so those ought to be the only ones to worry about? We did get ourselves in the mess there too, so I am not sure that's right either!

Here is another thought: 1. think of the combinators as polynomials in 3 operators, +, * and . (composition). 2. expand things out, with + being outer, * middle, . inner. 3. within each . term, use combinators to re-order things [we would need to pick a canonical order for all single combinators] 4. show this terminates

the issue is that the re-ordering could produce new * and/or + terms. But with a well-crafted term order, I think this could be shown terminating.

Here is a nice idea: we need a canonical form for every pi-combinator. Our previous approach gave us something but it was hard to work with. I think we can use the coherence conditions to reach a canonical form by simply picking a convention that chooses one side of every commuting diagram. What do you think? —Amr

I've been thinking about this some more. I can't help but think that, somehow, Laplaza has already worked that out, and that is what is actually in the 2nd half of his 1972 paper! [Well, that Rig-Category 'terms' have a canonical form, but that's what we need]

Pi-combinators might be simpler, I don't know.

Another place to look is in Fiore (et al?)'s proof of completeness of a similar case. Again, in their details might be our answer.

What's the proof strategy for establishing that a CPerm implies a Pi-combinator. The original idea was to translate each CPerm to a canonical Pi-combinator and then show that every combinator is equal to its canonical representative. Is that still the high-level idea?

Well enough. Last talk on the last day, so people are tired. Doubt we've caused a revolution in reversible computing... Though when I mentioned that the slides were literate Agda, Peter Selinger made a point of emphasizing what that meant.

I think the idea that (reversible circuits == proof terms) is just a little too wild for it to sink in quickly. Same with the idea of creating a syntactic language (i.e. Pi) out of the semantic structure of the desired denotational semantics (i.e. permutations). People understood, I think, but it might be too much to really 'get'.

If we had a similar story for Caley+T (as they like to call it), it might have made a bigger splash. But we should finish what we have first.

Note that I've pushed quite a few things forward in the code. Most are quite straightforward, but they help explain what we are doing, and the relation between some of the pieces. Ideally, there would be more of those easy ones [i.e. that evaluation is the same as the action of an equivalence which in turn is the same as the action of a permutation]. These are all 'extensional' in nature, but still an important sanity check.

Yes, I think this can make a full paper – especially once we finish those conjectures. It depends a little bit on which audience we would want to pitch it to.

I think the details are fine. A little bit of polishing is probably all that's left to do. Some of the transitions between topics might be a little abrupt. And we need to reinforce the message of "semantics drive the syntax + syntactic theory is good", which is there, but a bit lost in the sea of details. And the 'optimizing circuits' aspect could also be punched up a bit.

Writing it up actually forced me to add PiEquiv.agda to the repository – which is trivial (now), but definitely adds to the story. I think there might be some easy theorems relating PiLevel0 as a programming language, the action of equivalences, and the action of permutations. In other words, we could get that all 3 are the same 'extensionally' fairly easily. What we are still missing is a way to go back from either an equivalence or a permutation to a syntactic combinator.

Firstly, thanks Spencer for setting this up.

This is partly a response to Amr, and partly my own take on (computing with) graphical languages for monoidal categories.

One of the key ingredients to getting diagrammatic languages to do work for you is to actually take the diagrams seriously. String diagrams now have very strong coherence theorems which state that an equation holds by the axioms of (various kinds of) monoidal categories if and only if the diagrams are equal. The most notable of these are the theorems of Joyal & Street in Geometry of Tensor Calculus for monoidal, symmetric monoidal, and braided monoidal categories.

If you ignore these theorems and insist on working with the syntax of monoidal categories (rather than directly with diagrams), things become, as you put it "very painful".

Of course, when it comes to computing with diagrams, the first thing you have to make precise is exactly what you mean by "diagram". In Joyal & Street's picture, this literally a geometric object, i.e. some points and lines in space. This works very well, and pretty much exactly formalises what happens when you do a pen-and-paper proof involving string diagrams. However, when it comes to mechanising proofs, you need some way to represent a string diagram as a data structure of some kind. From here, there seem to be a few approaches:

(1: combinatoric) its a graph with some extra bells and whistles (2: syntactic) its a convenient way of writing down some kind of term (3: "lego" style) its a collection of tiles, connected together on a 2D plane

Point of view (1) is basically what Quantomatic is built on. "String graphs" aka "open-graphs" give a combinatoric way of working with string diagrams, which is sound and complete with respect to (traced) symmetric monoidal categories. See arXiv:1011.4114 for details of how we did this.

Naiively, point of view (2) is that a diagram represents an equivalence class of expressions in the syntax of a monoidal category, which is basically back to where we started. However, there

are more convenient syntaxes, which are much closer in spirit to the diagrams. Lately, we've had a lot of success in connected with abstract tensor notation, which came from Penrose. See g. arXiv:1308.3586 and arXiv:1412.8552.

Point of view (3) is the one espoused by the 2D/higher-dimensional rewriting people (e.g. Yves Lafont and Samuel Mimram). It is also (very entertainingly) used in Pawel Sobocinski's blog: http://graphicallinearalgebra.net .

This eliminates the need for the interchange law, but keeps pretty much everything else "rigid". This benefits from being able to consider more general categories, but is less well-behaved from the point of view of rewriting. For example as Lafont/Mimram point out, even finite rewrite systems can generate infinite sets of critical pairs.

This is a very good example of CCT. As I am sure that you and others on the list (e.g., Duncan Ross) know monoidal cats have been suggested for quantum mechanics, they are closely related to Petri nets, linear logic, and other "net-based" computational systems. There is considerable work on graphic syntax. It would be interesting to know more details on your cats and how you formalize them.

My primary CCT interest, so far, has been with what I call computational toposes. This is a slight strengthening of an elementary topos to make subobject classification work in a computational setting. This is very parallel to what you are doing, but aimed at engineering modeling. The corresponding graphical syntax is an enriched SysML syntax. SysML is a dialect of UML. These toposes can be used to provide a formal semantics for engineering modeling.

There's also the perspective that string diagrams of various flavors are morphisms in some operad (the composition law of which allows you to nest morphisms inside of morphisms).

From that perspective, the string diagrams for traced monoidal categories are little more than just bijections between sets. This idea, and its connection to rewriting (finding normal forms for morphisms in a traced or compact category), is something Jason Morton and I have been working on recently.

Yes, I am sure this observation has been made before. We'd have to verify it for all the 2-paths before we really claim this.

[And since monoidal categories are involved in knot theory, this is un-surprising from that angle as well]

looking at that 2path picture… if these were physical wires and boxes, we could twist the wires, flipping the c1-c2 box and having them cross on the other side. So really as we have noted before I am sure, these 2paths are homotopies in the sense of smooth transformations between paths. Not sure what to do with this observation at this point but I thought it is worth noting.

There are some slightly different approaches to implementing a category as a computational system which make more intrinsic use of logic, than the ones mentioned by Aleks. As well there is a different take on the relationship of graphical languages to the category implementation.

A category can be formalized as a kind of elementary axiom system using a language with two sorts, map and type (object), with equality for each sort. The signature contain the function symbols, Domain and Range. The arguments of both are a map and whose value is a type. The abbreviation

f:X to Y equiv Domain(f) = X and Range(f) = Y

is used for the three place predicate.

The operations such as the binary composition of maps are represented as first order function symbols. Of course the function constructions are not interpreted as total functions in the standard first order model theory. So, for example, one has axioms such as the typing condition

f:Z to Y, g:Y to X implies g(f):Z to X

A function symbol that always produces a map with a unique domain and range type, as a function of the arguments, is called a constructor. For example, id(X) is a constructor with a type argument. This same kind of logic can be used to present linear logics.

For most of the systems that I have looked at the axioms are often " rules", such as the category axioms. Sometimes one needs axioms which have rules as consequences. One can use standard first order inference together with rewrite technology to compute. The axioms for a category imply that the terms generate a directed graph. Additional axioms provide congruence relations on the graph.

A morphism of an axiom set using constructors is a functor. When the axioms include products and powers, the functors map to sets, this yields is a form of Henkin semantics. Thus, while it is not standard first order model theory, is well-known. For other kinds of axiom systems a natural semantics might be Hilbert spaces.

With this representation of a category using axioms in the "constructor" logic, the axioms and their theory serve as a kind of abstract syntax. The constructor logic approach provides standardization for categories which can be given axioms in this logic. Different axiom sets can be viewed as belonging to different profiles. The logic representation is independent of any particular graphical syntax. A graphical syntax would, of course have to interpret that axioms correctly. Possibly the Joyal and Street theorems can be interpreted as proving the graphical representation map is a structure preserving functor. Possibly the requirements for a complete graphical syntax is that it is an invertible functor.

'm writing you offline for the moment, just to see whether I am understanding what you would like. In short, I guess you want a principled understanding of where the coherence conditions come from, from the perspective of general 2-category theory perhaps (a la work of the Australian school headed by Kelly in the 1970's).

We are in some sense categorifying the notion of "commutative rig". The role of commutative monoid is categorified by symmetric monoidal category, which roughly is the next notion past commutative monoid in the stable range on the periodic table.

I believe there is a canonical candidate for the categorification of tensor product of commutative monoids. In other words, given symmetric monoidal categories A, B, C, the (symmetric monoidal) category of functors A x B –> C that are strong symmetric monoidal in separate arguments should be equivalent to the (sm) category of strong symmetric monoidal functors $A \otimes B - - > C$, for this canonical tensor product $A \otimes B$. Actually, I don't think we absolutely need this construction – we could phrase everything in terms of "multilinear" (i.e. multi-(strong sm)) functors $A_1 x ... x A_n - - > B$, but it seems a convenience worth taking advantage of. In fact, let me give this tensor product a more neutral name – I'll write @, and I for the tensor unit – because I'll want to reserve $\otimes$ for something else (consistent with Laplaza's notation).

If S is the 2-category of symmetric monoidal categories, strong symmetric monoidal functors, and monoidal natural transformations, then this @ should endow S with a structure of (symmetric) monoidal 2-category, with some other pleasant properties (such as S's being symmetric monoidal closed in the appropriate 2-categorical sense). All of these facts should be deducible on abstract grounds, by categorifying the notion of commutative monad (such as the free commutative monoid monad on Set) to an appropriate categorification to commutative 2-monad on Cat, and categorifying the work of Kock on commutative monads.

In any symmetric monoidal 2-category, we have a notion of "pseudo-commutative pseudomonoid", which generalizes the notion of symmetric monoidal category in the special case of the monoidal 2-category (Cat, x). Anyhow, if $(C, \oplus, N)$ is a symmetric monoidal category, then I my guess (I've checked some but not

all details) is that a symmetric rig category is precisely a pseudo-commutative pseudomonoid object ($\otimes : C@C --> C, U : I --> C$, etc.)

in (S, @). I would consider this is a reasonable description stemming from general 2-categorical principles and concepts.

Would this type of thing satisfy your purposes, or are you looking for something else?

Quite related indeed. But much more ad hoc, it seems [which they acknowledge].

Something closer to our work `http://www.informatik.uni-bremen.de/agra/doc/konf/rc15_ricercar.pdf`

More related work (as I encountered them, but later stuff might be more important):

Diagram Rewriting and Operads, Yves Lafont `http://iml.univ-mrs.fr/~lafont/pub/diagrams.pdf`

A Homotopical Completion Procedure with Applications to Coherence of Monoids `http://drops.dagstuhl.de/opus/frontdoor.php?source_opus=4064`

A really nice set of slides that illustrates both of the above `http://www.lix.polytechnique.fr/Labo/Samuel.Mimram/docs/mimram_kbs.pdf`

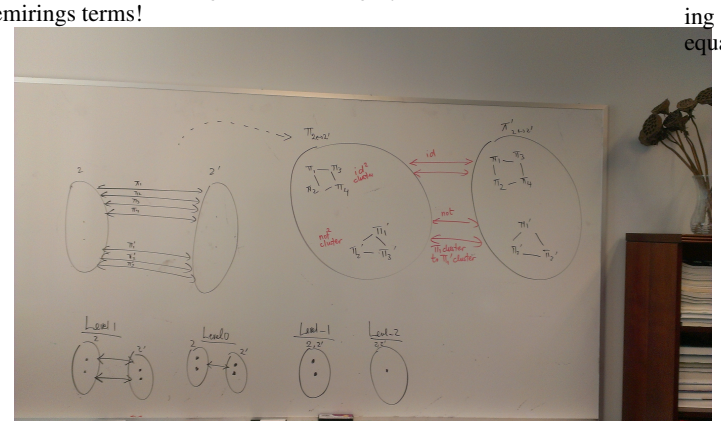
I think there is something very important going on in section 7 of `http://comp.mq.edu.au/~rgarner/Papers/Glynn.pdf` which I also attach. [I googled 'Knuth Bendix coherence' and these all came up]

There are also seems to be relevant stuff buried (very deep!) in chapter 13 of Amadio-Curiens' Domains and Lambda Calculi.

Also, Tarmo Uustalu's "Coherence for skew-monoidal categories", available on `http://cs.ioc.ee/~tarmo/papers/`

[Apparently I could have saved myself some of that searching time by going to `http://ncatlab.org/nlab/show/rewriting` ! At the bottom, the preprint by Mimram seems very relevant as well]

Somehow, at the end of the day, it seems we're looking for a confluent, terminating term-rewriting system for commutative semirings terms!



8. Conclusion

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.

A. Commutative Semirings

Given that the structure of commutative semirings is central to this paper, we recall the formal algebraic definition.

Definition 2. *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 $a, b, c \in R$:*

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

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

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

In the paper, we are interested into various commutative semiring structures up to some congruence relation instead of strict equality $=$.