

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

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.

2. Typed Isomorphisms

First, a universe of (finite) types

data **U** : **Set** **where**
ZERO : **U**

ONE : U
 PLUS : U → U → U
 TIMES : U → U → U

and its interpretation

$\llbracket _ \rrbracket : U \rightarrow \text{Set}$
 $\llbracket \text{ZERO} \rrbracket = \perp$
 $\llbracket \text{ONE} \rrbracket = \top$
 $\llbracket \text{PLUS } t_1 t_2 \rrbracket = \llbracket t_1 \rrbracket \uplus \llbracket t_2 \rrbracket$
 $\llbracket \text{TIMES } t_1 t_2 \rrbracket = \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket$

Equivalences and semirings. If we denote type equivalence by \simeq , then we can prove that

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

We also get

Theorem 2. *If $A \simeq \text{Fin } m$, $B \simeq \text{Fin } n$ and $A \simeq B$ then $m \equiv n$.*

(whose constructive proof is quite subtle).

Theorem 3. *If $A \simeq \text{Fin } m$ and $B \simeq \text{Fin } n$, then the type of all equivalences $A \simeq B$ is equivalent to the type of all permutations $\text{Perm } n$.*

Equivalences and semirings II. Semiring structures abound. We can define them on:

1. equivalences (disjoint union and cartesian product)
2. permutations (disjoint union and tensor product)

The point, of course, is that they are related:

Theorem 4. *The equivalence of Theorem 3 is an isomorphism between the semirings of equivalences of finite types, and of permutations.*

A more evocative phrasing might be:

Theorem 5.

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

A Calculus of Permutations. Syntactic theories only rely on transforming source programs to other programs, much like algebraic calculation. Since only the *syntax* of the programming language is relevant to the syntactic theory, the theory is accessible to non-specialists like programmers or students.

In more detail, it is a general problem that, despite its fundamental value, formal semantics of programming languages is generally inaccessible to the computing public. As Schmidt argues in a recent position statement on strategic directions for research on programming languages [?]:

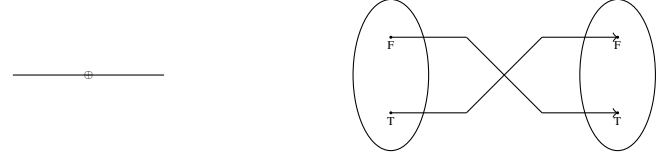
... formal semantics has fed upon increasing complexity of concepts and notation at the expense of calculational clarity. A newcomer to the area is expected to specialize in one or more of domain theory, intuitionistic type theory, category theory, linear logic, process algebra, continuation-passing style, or whatever. These specializations have generated more experts but fewer general users.

A Calculus of Permutations. First conclusion: it might be useful to *reify* a (sound and complete) set of equivalences as combinators, such as the fundamental “proof rules” of semirings:

data $\dashv \dashv \dashv$: U → U → Set where
 unite₊ : {t : U} → PLUS ZERO t ↔ t
 unite₊ : {t : U} → t ↔ PLUS ZERO t
 swap₊ : {t₁ t₂ : U} → PLUS t₁ t₂ ↔ PLUS t₂ t₁
 assocl₊ : {t₁ t₂ t₃ : U} → PLUS t₁ (PLUS t₂ t₃) ↔ PLUS (PLUS t₁ t₂) t₃
 assocr₊ : {t₁ t₂ t₃ : U} → PLUS (PLUS t₁ t₂) t₃ ↔ PLUS t₁ (PLUS t₂ t₃)
 unite* : {t : U} → TIMES ONE t ↔ t
 unite* : {t : U} → t ↔ TIMES ONE t
 swap* : {t₁ t₂ : U} → TIMES t₁ t₂ ↔ TIMES t₂ t₁
 assocl* : {t₁ t₂ t₃ : U} → TIMES t₁ (TIMES t₂ t₃) ↔ TIMES (TIMES t₁ t₂) t₃
 assocr* : {t₁ t₂ t₃ : U} → TIMES (TIMES t₁ t₂) t₃ ↔ TIMES t₁ (TIMES t₂ t₃)

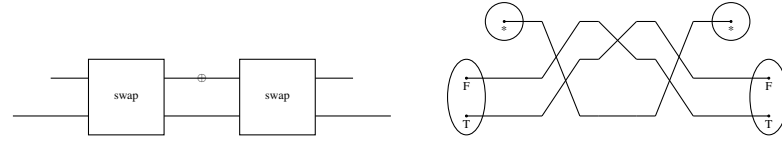
absorbr : {t : U} → TIMES ZERO t ↔ ZERO
 absorbl : {t : U} → TIMES t ZERO ↔ ZERO
 factorzr : {t : U} → ZERO ↔ TIMES t ZERO
 factorzl : {t : U} → ZERO ↔ TIMES ZERO t
 dist : {t₁ t₂ t₃ : U} → TIMES (PLUS t₁ t₂) t₃ ↔ PLUS (TIMES t₁ t₃) (TIMES t₂ t₃)
 factor : {t₁ t₂ t₃ : U} → PLUS (TIMES t₁ t₂) (TIMES t₂ t₃) ↔ TIMES (PLUS t₁ t₂) t₃
 id ↔ : {t : U} → t ↔ t
 ⊖ : {t₁ t₂ t₃ : U} → (t₁ ↔ t₂) → (t₂ ↔ t₃) → (t₁ ↔ t₃)
 ⊕ : {t₁ t₂ t₃ t₄ : U} → (t₁ ↔ t₂) → (t₂ ↔ t₃) → (t₃ ↔ t₄) → (PLUS t₁ t₂ ↔ PLUS t₃ t₄)
 ⊗ : {t₁ t₂ t₃ t₄ : U} → (t₁ ↔ t₂) → (t₂ ↔ t₃) → (t₃ ↔ t₄) → (TIMES t₁ t₂ ↔ TIMES t₃ t₄)

3. Example Circuit: Simple Negation



BOOL : U
BOOL = PLUS ONE ONE
 n₁ : BOOL ↔ BOOL
 n₁ = swap₊

Example Circuit: Not So Simple Negation.

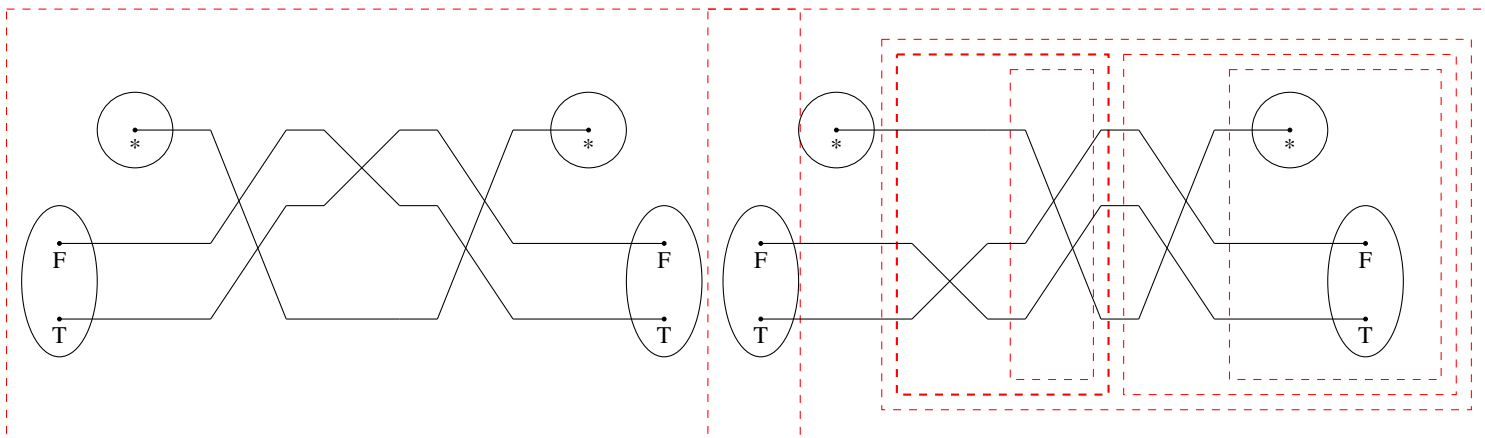


n₂ : BOOL ↔ BOOL
 n₂ = unite* ⊙
 swap* ⊙
 (swap₊ ⊗ id ↔) ⊙
 swap* ⊙
 unite*

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

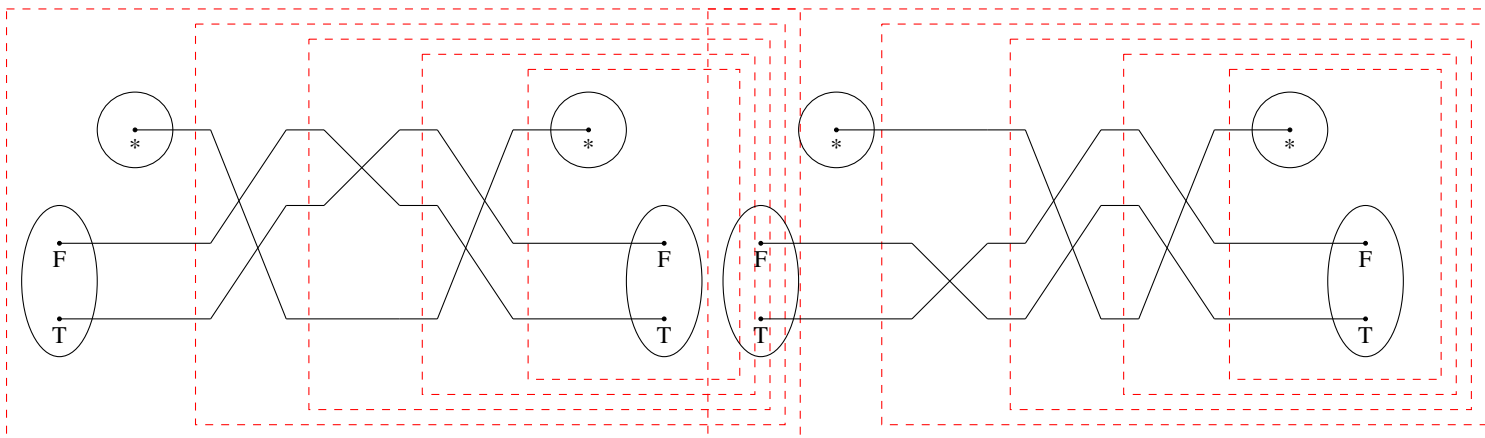
negEx : n₂ ↔ n₁
 negEx = unite* ⊙ (swap* ⊙ ((swap₊ ⊗ id ↔) ⊙ (swap* ⊙ unite*)))
 ↔ (id ↔ ⊠ assoc ⊠ l)
 unite* ⊙ ((swap* ⊙ (swap₊ ⊗ id ↔)) ⊙ (swap* ⊙ unite*))
 ↔ (id ↔ ⊠ (swap₊ ⊗ id ↔))
 unite* ⊙ (((id ↔ ⊗ swap₊) ⊙ swap*) ⊙ (swap* ⊙ unite*))
 ↔ (id ↔ ⊠ assoc ⊠ r)
 unite* ⊙ ((id ↔ ⊗ swap₊) ⊙ (swap* ⊙ (swap* ⊙ unite*)))
 ↔ (id ↔ ⊠ (id ↔ ⊠ assoc ⊠ l))
 unite* ⊙ ((id ↔ ⊗ swap₊) ⊙ ((swap* ⊙ swap*) ⊙ unite*))
 ↔ (id ↔ ⊠ (id ↔ ⊠ (linv ⊠ id ↔)))
 unite* ⊙ ((id ↔ ⊗ swap₊) ⊙ (id ↔ ⊗ unite*))
 ↔ (id ↔ ⊠ (id ↔ ⊠ idl ⊠ l))
 unite* ⊙ ((id ↔ ⊗ swap₊) ⊙ unite*)
 ↔ (assoc ⊠ l)
 (unite* ⊙ (id ↔ ⊗ swap₊)) ⊙ unite*
 ↔ (unite* ⊗ id ↔)
 (swap₊ ⊙ unite*) ⊙ unite*
 ↔ (assoc ⊠ r)
 swap₊ ⊙ (unite* ⊙ unite*)
 ↔ (id ↔ ⊠ linv ⊠ l)
 swap₊ ⊙ id ↔
 ↔ (idr ⊠ l)
 swap₊ ⊠

Visually.
 Original circuit:



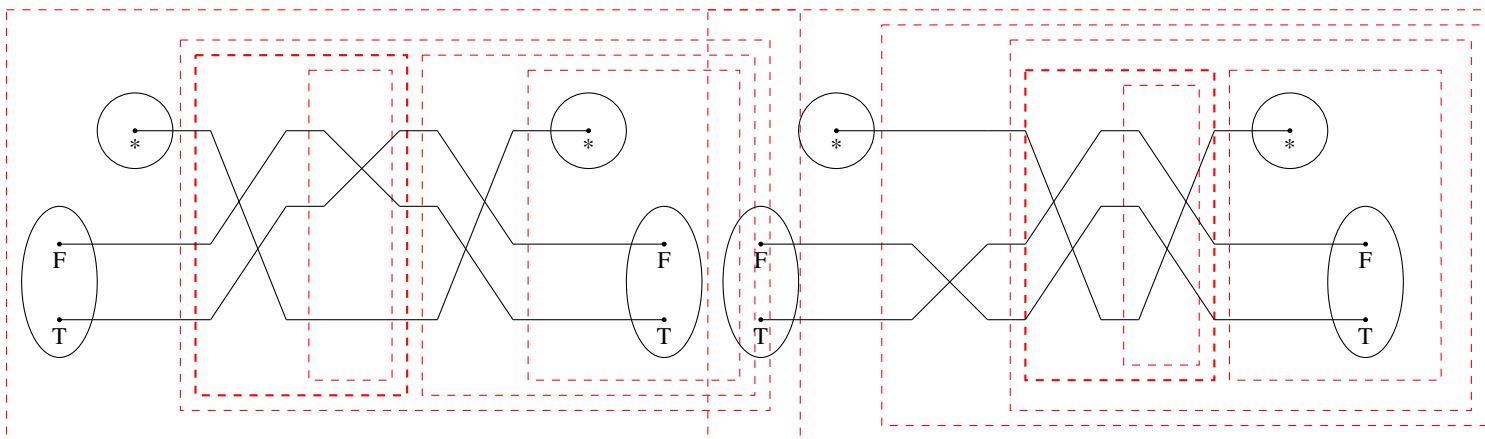
Making grouping explicit:

By associativity:



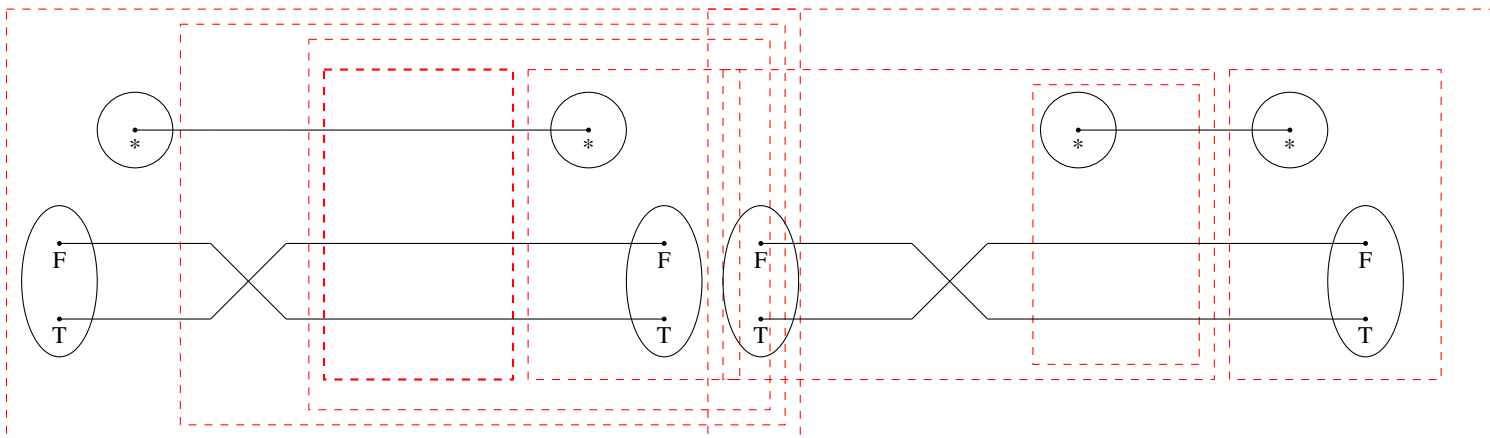
By associativity:

By associativity:



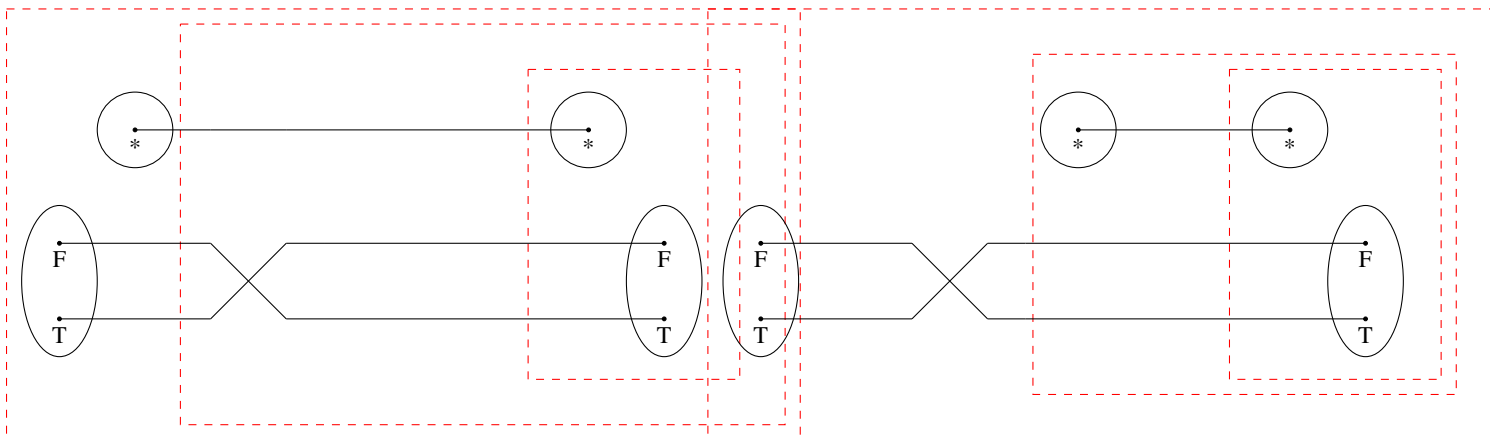
By pre-post-swap:

By swap-swap:



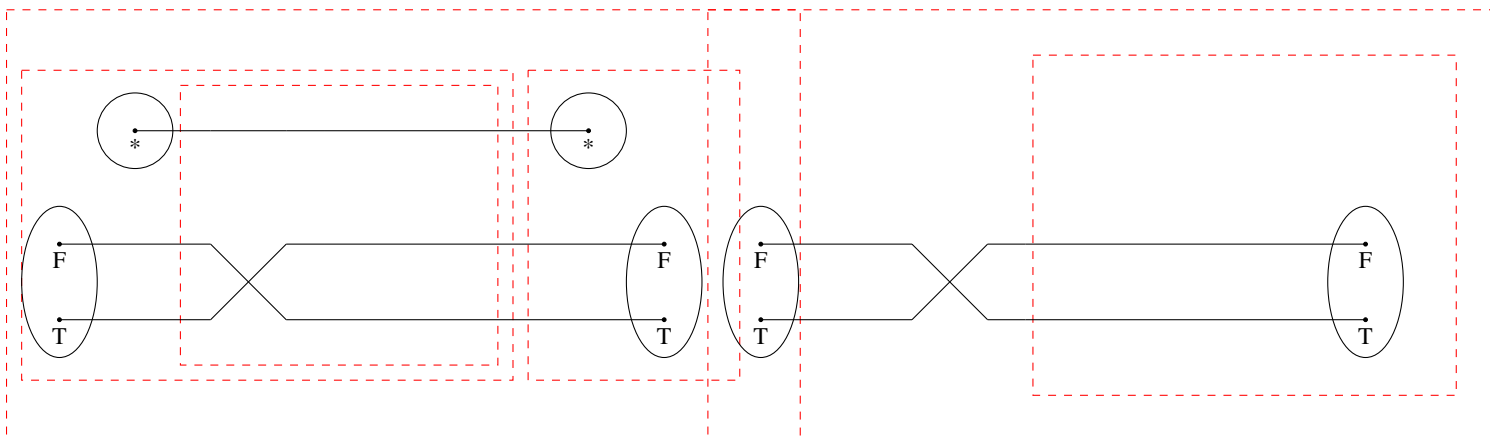
By id-compose-left:

By associativity:



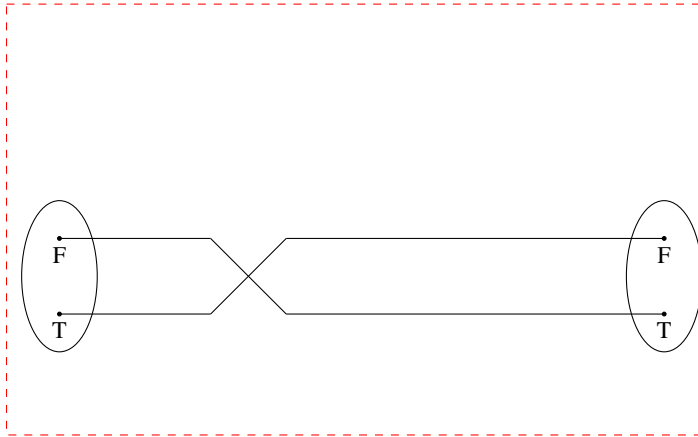
By associativity:

By unit-unit:



By swap-unit:

By id-unit-right:



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 6. 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

4. But is this a programming language?

We get forward and backward evaluators $\text{eval} : \{t_1 \ t_2 : \mathbf{U}\} \rightarrow (t_1 \longleftrightarrow t_2) \rightarrow \llbracket t_1 \rrbracket \rightarrow \llbracket t_2 \rrbracket$ and of course the punchline:

$\text{evalB} : \{t_1 \ t_2 : \mathbf{U}\} \rightarrow (t_1 \longleftrightarrow t_2) \rightarrow \llbracket t_2 \rrbracket \rightarrow \llbracket t_1 \rrbracket$ which really do behave as expected

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)

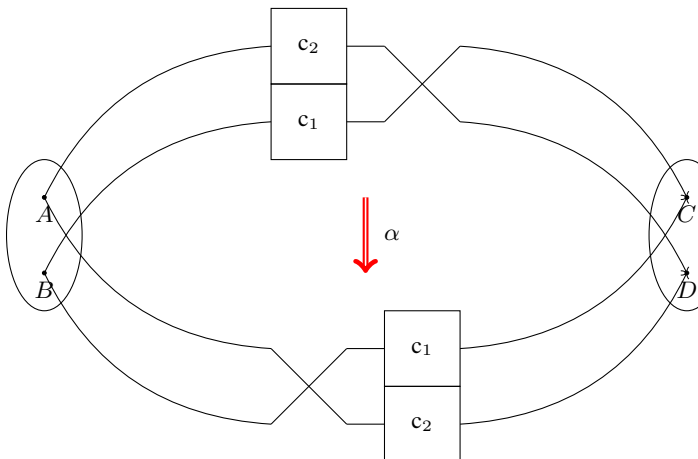
5. Categorification I

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
 $\mathbf{c}_1 : \{B \ C : \mathbf{U}\} \rightarrow B \longleftrightarrow C$
 $\mathbf{c}_2 : \{A \ D : \mathbf{U}\} \rightarrow A \longleftrightarrow D$
 $\mathbf{p}_1 \ \mathbf{p}_2 : \{A \ B \ C \ D : \mathbf{U}\} \rightarrow \text{PLUS } A \ B \longleftrightarrow \text{PLUS } C \ D$
 $\mathbf{p}_1 = \text{swap}_+ \odot (\mathbf{c}_1 \oplus \mathbf{c}_2)$
 $\mathbf{p}_2 = (\mathbf{c}_2 \oplus \mathbf{c}_1) \odot \text{swap}_+$

2-morphism of circuits



Theorem 7 (Laplaza 1972). There is a sound and complete set of **coherence rules** for Symmetric Rig Categories.

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

6. Emails

Reminder of

<http://mathoverflow.net/questions/106070/int-construction>

Also,

<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.1.1.1.1.1.1.1> seems relevant

Indeed, this does not seem to be in the library.

On 2015-04-10 10:52 AM, Amr Sabry wrote:

I had checked and found no traced categories or Int construction

The story without trace and without the Int construction

On 04/10/2015 09:06 AM, Jacques Carette wrote:

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

But it really does depend on the venue you'd like to see POPL, then I agree, we need the Int construction. The 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. The conditions are non-trivial.
 Jacques

On 2015-04-10, 06:06 , Sabry, Amr A. wrote:

I am thinking that our story can only be compelling if that h.o. functions might work. We can make that case by implementing the Int Construction and showing that a li h.o. functions emerges and leave the big open problem of the multiplication etc. for later work. I can start work will require adding traced categories and then a generi

Construction in the categories library. What do you think?

On Apr 9, 2015, at 10:59 PM, Jacques Carette <carette@mcmaster.ca> wrote:

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

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

Jacques

On 2015-04-09 11:41 AM, Sabry, Amr A. wrote: Can you make sense of how this relates to us?

<https://pigworker.wordpress.com/2015/04/01/warming-up-to-checked-objects/>

Unfortunately not. Yes, there is a general feeling that the Int construction is not quite right.

I do believe that all our terms have computational content.

Note that at level 1, we have equivalences between `Perm(A, B)` and `Perm(B, A)`.

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 can do.

Jacques

On 2015-04-09 12:05 PM, Amr Sabry wrote: Trying to get a handle on what we can transport from HoTT to Int.

(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, we can transport something from A to B.

For example: take $P = . + C$; we can build a permutation between P and P .

--

Level 1: Given types A, B, C, and D. let $\text{Perm}(A, B)$ be the type of permutations between A and B.

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 A and B.

In HoTT this is exhibited by the failure of canonicity.

Perhaps we can adapt the discussion/example in <http://homotopytypetheory.org/2011/07/27/canonicity-for-2/>

--Amr

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 that the Int construction is not quite right.

On 2015-04-09 12:36 PM, Amr Sabry wrote: This came up in a different context but looks like it might be related.

<http://arxiv.org/pdf/gr-qc/9905020>

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

Jacques

On 2015-04-10 11:04 AM, Sabry, Amr A. wrote: Yes. The categories library has a Grothendieck construction.

On Apr 10, 2015, at 11:04 AM, Jacques Carette <carette@mcmaster.ca> wrote: Reminder of the <http://mathoverflow.net/questions/106070/int-construction>

<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.1.1.1.1.1.1.1> seems relevant

Indeed, this does not seem to be in the library.

On 2015-04-10 10:52 AM, Amr Sabry wrote: I checked the Int construction and it seems to be in the library.

I don't know, that a "symmetric rig" (never mind higher) is not in the library.

But it really does depend on the venue you'd like to see it.

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

On 2015-04-10 11:04 AM, Sabry, Amr A. wrote: I checked the Int construction and it seems to be in the library.

I don't know, that a "symmetric rig" (never mind higher) is not in the library.

But it really does depend on the venue you'd like to see it.

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

On 2015-04-10 11:04 AM, Sabry, Amr A. wrote: I checked the Int construction and it seems to be in the library.

I don't know, that a "symmetric rig" (never mind higher) is not in the library.

But it really does depend on the venue you'd like to see it.

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

3. within each . term, use combinators to re-order and sign so we would like to get a canonical order of the
4. show this terminates

On 2015-06-02 7:53 PM, Sabry, Amr A. wrote:
the issue is that the re-ordering could produce new kinds of paths. But with a well-chosen detour, physical
Jacques
There are some slightly different approaches to impleme

On 2015-04-27 6:16 AM, Sabry, Amr A. wrote: A category can be formalized as a kind of elementary axiomatic theory. Here is a nice idea: we need a canonical form for every pi-combinator. Our previous approach gave us some $f: X \rightarrow Y$ equiv $\text{Domain}(f) = X$ and $\text{Range}(f) = Y$. I've been thinking about this some more. I can't help but think that, somehow, Laplaza has already worked out how this is used for the three place predicate. 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 case $f:Z$ to Y , $g:Y$ to X implies $g(f):Z$ to X

On 2015-04-26 6:34 AM, Sabry, Amr A. wrote:
 What's the proof strategy for establishing that $\text{aACFermion} \leq \text{aP}$ that always produces the original aP with was fine.
 Well enough. Last talk on the last day, so people were most tired. The system's behavior caused a few people to be confused.

I think the idea that (reversible circuits == proof of this is just a little too using with foructors and func

If we had a similar story for Caley+T (as they likely do), we would have made garb usage slashes iBu

Note that I've pushed quite a few things forward in what you might call the "near future" (the next few months).

Yes, I think this can make a full paper -- especially if we emphasize that the base configuration is not dependent on the

I think the details are fine. A little bit of pol

Writing it up actually forced me to add $\mathrm{PiEquiv.alphaStoSther2}$ to the $\mathrm{ecosystocrv}$ - ef why not $\mathrm{isicrmonald}$ $\mathrm{ahowate}$ buries

Firstly, thanks Spencer for setting this up. In any symmetric monoidal 2-category, we have a notion

This is partly a response to Amr, and partly my own takes on C (Computing with Imperative languages for monads).

One of the key ingredients to getting diagrammatical squares to work for every \mathbb{A}^1 -isomorphism is the existence of

If you ignore these theorems and insist on working with the type of thing ideals for varieties other than

Of course, when it comes to computing with diagrams there are indeed you have to make a choice is exactly

(1: combinatoric) its a graph with some extra bells and whistles

(2) syntactic) its a convenient way of writing down something that is true on 2015-11-27 of C. Teram, Sabry, Amr A. wrote:

Point of view (1) is basically what Quantomatic is built on. "String graphs" aka "open-graphs" give a co-

Naively, point of view (2) is that a diagram represents a downward case (as opposed to an upward one) in the argument.

Point of view (3) is the one espoused by the 2D/hypermedia/represen-

This eliminates the need for the interchange law, but keeps pretty much everything else "rigid". This be-

This is a very good example of CCT. As I am sure [http://vondra.daher.shandehellisfrderdoorDnnh3aoR9sslokn3](#)

My primary CCT interest so far has been with what's called *incommensurable values*. This is a place where

There's also the perspective that string diagrams of various flavors are morphisms in some operad (the

From that perspective, the string diagrams for $\text{trbraid} : \text{braid} \rightarrow \text{braid}$ and $\text{trbraid} : \text{braid} \rightarrow \text{braid}$ are morphisms in some operad (the one for braiding). I think there is something very important going on in some of these examples.

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

There are also seems to be relevant stuff buried (very

Also, Tarmo Uustalu's "Coherence for skew-monoidal
[Apparently I could have saved myself some of that
Somehow, at the end of the day, it seems we're look