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 mechnically 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

Amr says:

- type equivalences are a commutative semiring
- permutations on finite sets are another commutative semiring
- these two structures are themselves equivalent

SO if we are interested in studying type equivalences, we can study permutations on finite sets; the latter can be axiomatized which is nice

Semiring structures abound. We can define them on type equivalences (disjoint union and cartesian product), and on permutations of finite sets (disjoint union and tensor product).

2.1 Type Equivalences

Two types are considered *equivalent* if there exists a pair of mediating maps between them that compose to the identity function in both directions. If we denote type equivalence by $\simeq$, then we can prove the following theorem.

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

For example, we have equivalences such as:

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

These equivalences are based on the facts that the empty type $\bot$ is the additive unit for the commutative and associative sum type $\uplus$, that the unit type $\top$ is the multiplicative unit for the commutative and associative product type $\times$, and that $\times$ distributes over $\uplus$. In addition, we have equivalences such as $\top \uplus (\top \uplus \top) \simeq \mathsf{Fin}\ 3$ and $(\top \uplus \top) \times (\top \uplus \top) \simeq \mathsf{Fin}\ 4$ which establish that every type constructed from sums and products over the empty type and the unit type is, up to $\simeq$, equivalent to a finite set $\mathsf{Fin}\ m$ for some natural number m. More generally, we can prove the following theorem.

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

This theorem, whose *constructive* proof is quite subtle, establishes that, up to equivalence, the only interesting property of a type constructed from sums and products over the empty type and the unit type is its size. This result allows us to characterize equivalences between types in a canonical way as permutations between finite sets.

2.2 Permutations on Finite Sets

2.3 Equivalences of Equivalences

The point, of course, is that the type of all type equivalences is itself equivalent to the type of all permutations on finite sets. Formally, we have the following theorem.

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

In fact we have the following stronger theorem.

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.

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

3. A Calculus of Permutations

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, inuitionistic type theory, category theory, linear logic, process algebra, continuation-passing style, or whatever. These specializations have generated more experts but fewer general users.

We are concerned, not just with 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: identity and negation. Each of these equivalences can be used to "transport" properties of Bool in a different way.

Typed Isomorphisms

First, a universe of (finite) types

```
data U : Set where
  ZERO  : U
  ONE   : U
  PLUS  : U → U → U
  TIMES : U → U → U
```

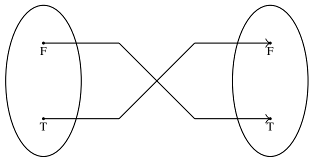
and its interpretation

```
⟦_⟧ : U → Set
⟦ ZERO ⟧        = ⊥
⟦ ONE ⟧         = ⊤
⟦ PLUS t₁ t₂ ⟧  = ⟦ t₁ ⟧ ⊎ ⟦ t₂ ⟧
⟦ TIMES t₁ t₂ ⟧ = ⟦ t₁ ⟧ × ⟦ t₂ ⟧
```

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 _⟷_ : U → U → Set where
  unite₊   : {t : U} → PLUS ZERO t ⟷ t
  uniti₊   : {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
  uniti⋆   : {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)
  assocr⋆  : {t₁ t₂ t₃ : U} → TIMES (TIMES t₁ t₂) t₃ ⟷ TIMES t₁ (TIMES t₂
  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)
  factor   : {t₁ t₂ t₃ : U} → PLUS (TIMES t₁ t₃) (TIMES t₂ t₃) ⟷ TIMES (
  id⟷      : {t : U} → t ⟷ t
  _⊙_      : {t₁ t₂ t₃ : U}    → (t₁ ⟷ t₂) → (t₂ ⟷ t₃) → (t₁ ⟷ t₃)
  _⊕_      : {t₁ t₂ t₃ t₄ : U} → (t₁ ⟷ t₃) → (t₂ ⟷ t₄) → (PLUS t₁ t₂ ⟷
  _⊗_      : {t₁ t₂ t₃ t₄ : U} → (t₁ ⟷ t₃) → (t₂ ⟷ t₄) → (TIMES t₁ t₂ ⟷
```

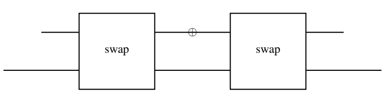
4. Example Circuit: Simple Negation

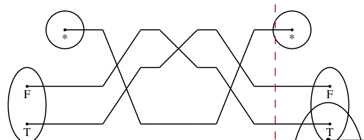



```
BOOL : U
BOOL = PLUS ONE ONE

n₁ : BOOL ⟷ BOOL
n₁ = swap₊
```

Example Circuit: Not So Simple Negation.





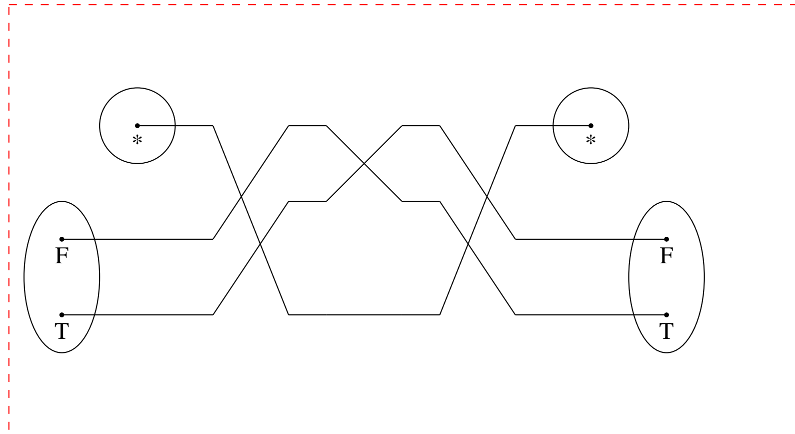

```
n₂ : BOOL ⟷ BOOL
n₂ =   uniti⋆ ⊙
       swap⋆ ⊙
       (swap₊ ⊗ id⟷) ⊙
       swap⋆ ⊙
       unite⋆
```

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

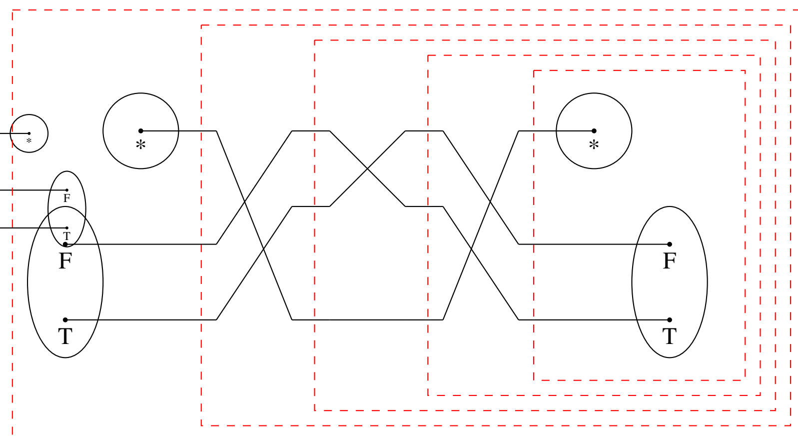
```
negEx : n₂ ⇔ n₁
negEx = uniti⋆ ⊙ (swap⋆ ⊙ ((swap₊ ⊗ id⟷) ⊙ (swap⋆ ⊙ unite⋆)))
        ⇔⟨ id⇔ ⊡ assoc⊙l ⟩
    uniti⋆ ⊙ ((swap⋆ ⊙ (swap₊ ⊗ id⟷)) ⊙ (swap⋆ ⊙ unite⋆))
        ⇔⟨ id⇔ ⊡ (swapl⋆⇔ ⊡ id⇔) ⟩
    uniti⋆ ⊙ (((id⟷ ⊗ swap₊) ⊙ swap⋆) ⊙ (swap⋆ ⊙ unite⋆))
        ⇔⟨ id⇔ ⊡ assoc⊙r ⟩
    uniti⋆ ⊙ ((id⟷ ⊗ 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.

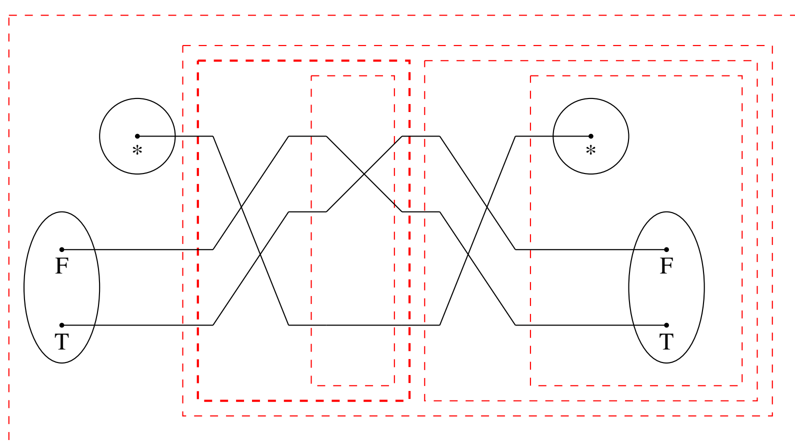
Original circuit:



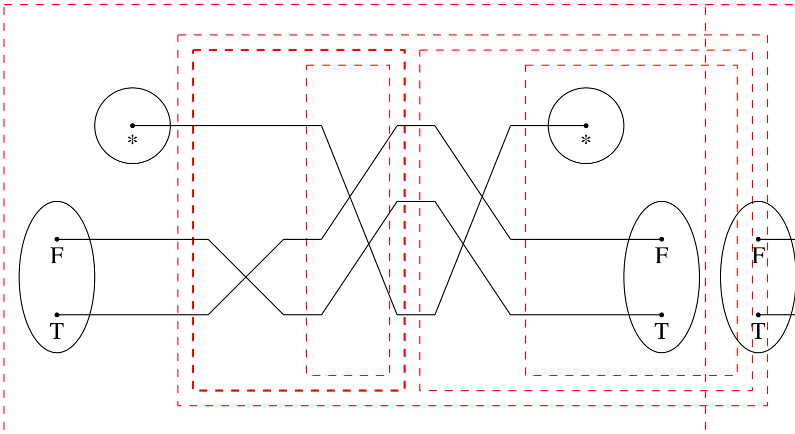
Making grouping explicit:



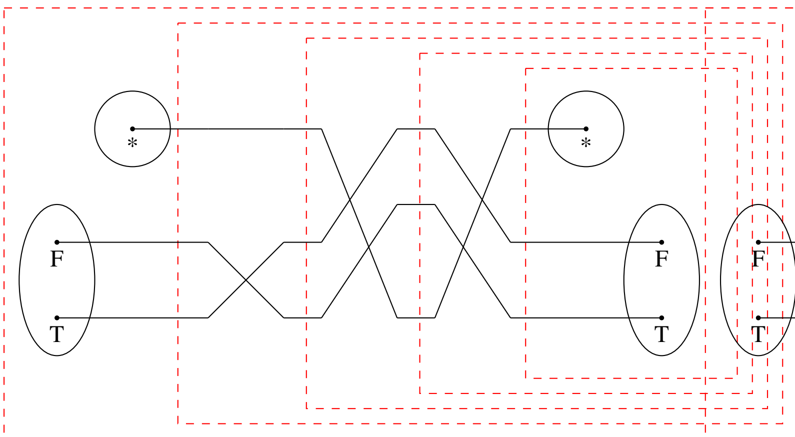
By associativity:



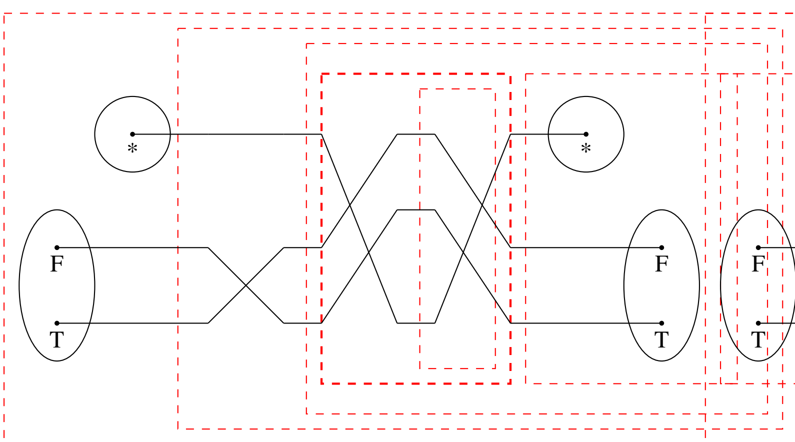
By pre-post-swap:



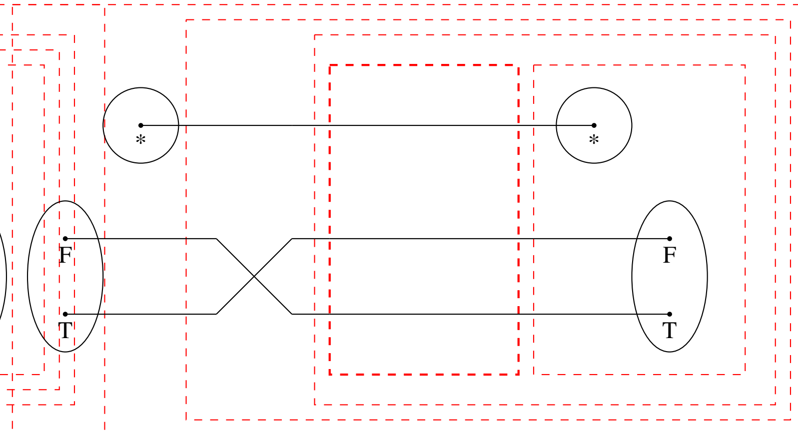
By associativity:



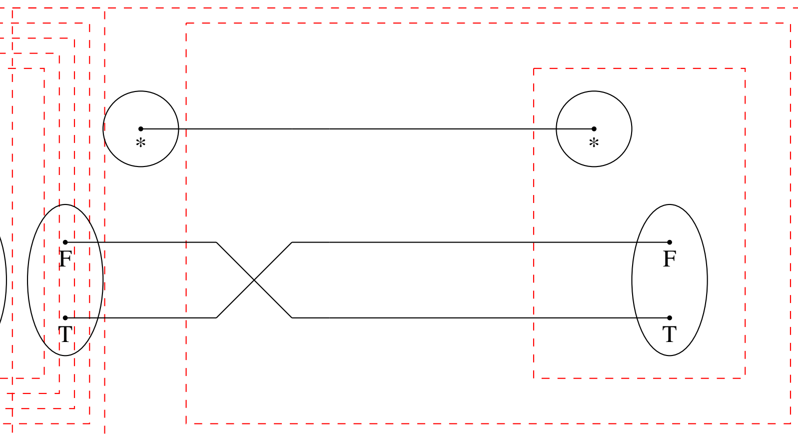
By associativity:



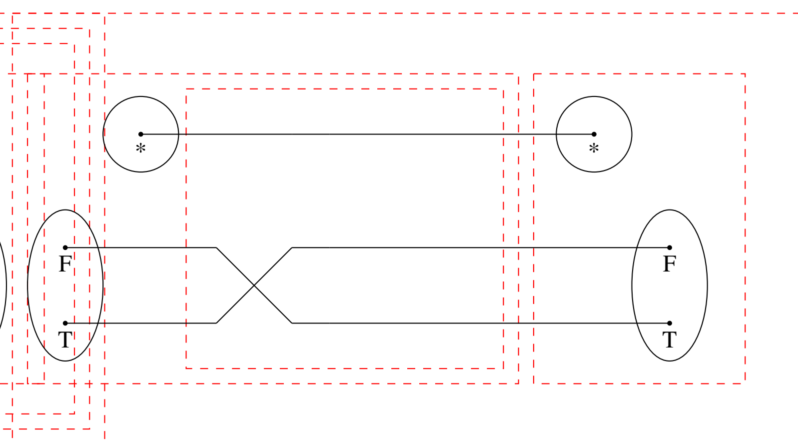
By swap-swap:



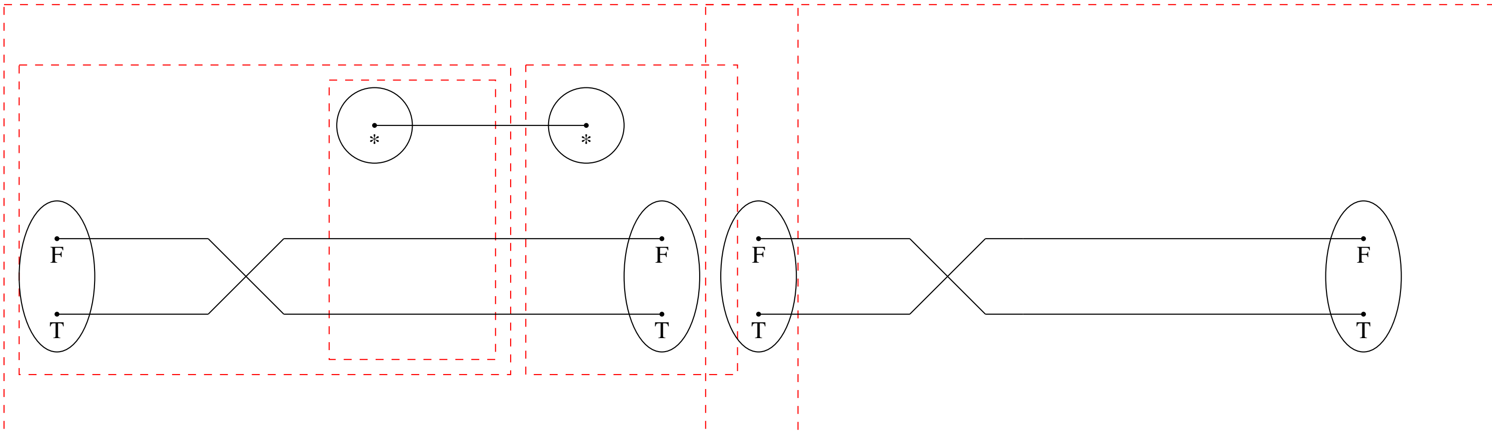
By id-compose-left:



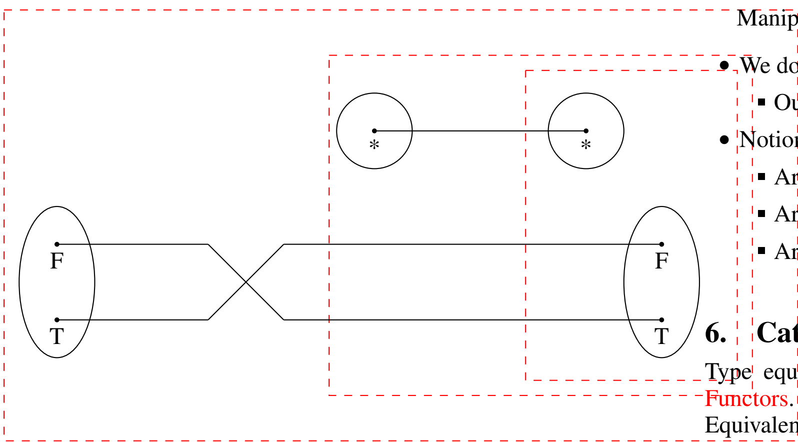
By associativity:



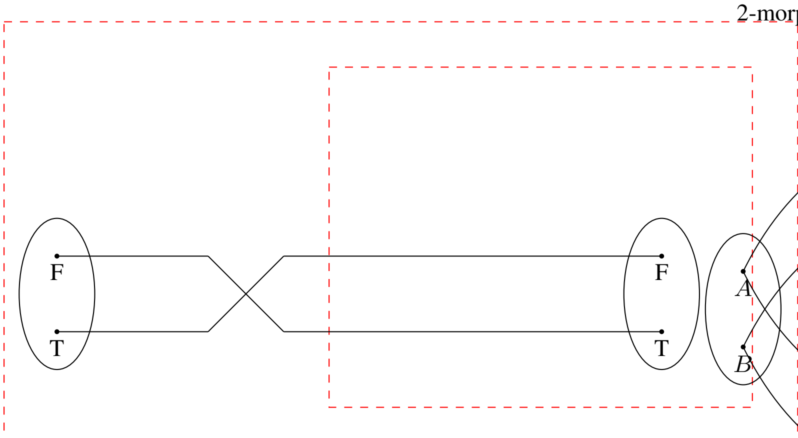
By swap-unit:



By associativity:



By unit-unit:



By id-unit-right:

5. But is this a programming language?

We get forward and backward evaluators **eval** : $\{t_1\ t_2 : \mathbf{U}\} \to (t_1 \longleftrightarrow t_2) \to [\![t_1]\!] \to [\![t_2]\!]$
evalB : $\{t_1\ t_2 : \mathbf{U}\} \to (t_1 \longleftrightarrow t_2) \to [\![t_2]\!] \to [\![t_1]\!]$
which really do behave as expected **c2equiv** : $\{t_1\ t_2 : \mathbf{U}\} \to (c : t_1 \longleftrightarrow t_2) \to [\![t_1]\!] \simeq [\![t_2]\!]$

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)

6. Categorification I

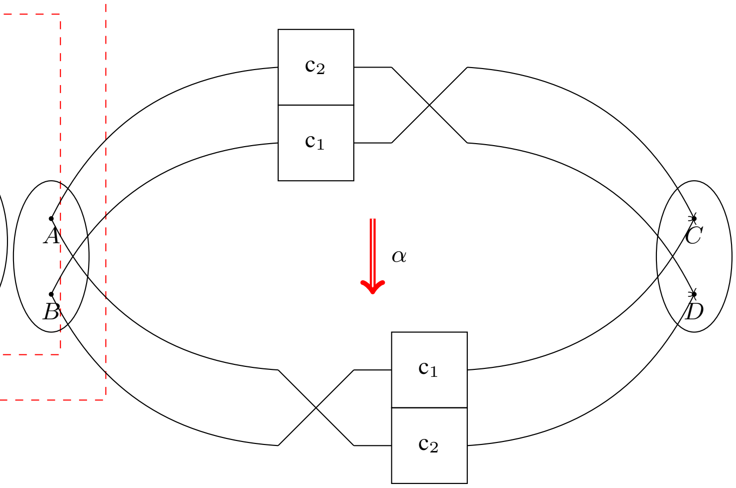
Type equivalences (such as between $A \times B$ and $B \times A$) are Functors.

Equivalences between Functors are Natural Isomorphisms. At the value-level, they induce 2-morphisms:

```
postulate
    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 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*

and of course the punchline:

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.*

7. Emails

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

Also,
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.163.334
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 constructions in the categories library. I'll think

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

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

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

On Apr 9, 2015, at 10:59 PM, Jacques Carette <carette@m
wrote:

I have the braiding, and symmetric structures done.  Mo
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-t

Unfortunately not.  Yes, there is a general feeling of

I do believe that all our terms have computational rule

Note that at level 1, we have equivalences between Perm

Yes, we should dig into the Licata/Harper work and adap

Though I think we have some short-term work that we sim

Jacques

On 2015-04-09 12:05 PM, Amr Sabry wrote:
Trying to get a handle on what we can transport or more

(I use permutation for level 0 to avoid too many uses o

Level 0: Given two types A and B, if we have a permutat

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

--

Level 1: Given types A, B, C, and D. let Perm(A,B) be t
This is more interesting. What's a good example though

In think that in HoTT the only way to do this transport
In HoTT this is exhibited by the failure of canonicity:

Perhaps we can adapt the discussion/example in http://h
--Amr

I hope not! [only partly joking]

Actually, there is a fair bit about this that I dislike

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

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

Separate.  The Grothendieck construction in this case i

Jacques
```

On 2015-04-10 11:56 AM, Sabry, Amr A. wrote:
Yes. The categories library has a Grothendieck construction. Not sure if we can use that

On Apr 10, 2015, at 11:04 AM, Jacques Carette <carette@mcmaster.ca> wrote:

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

Also,
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.163.334
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 constructions in the categories library. I'll think

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

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

On 2015-04-10, 06:06 , Sabry, Amr A. wrote:
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 higher "Int" of
the multiplication etc. for later work. I can start working on that. It
will require adding traced categories and then a generic Int
Construction in the categories library. What do you think? --Amr

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

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.

Jacques

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

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

On 2015-04-25 8:37 AM, Sabry, Amr A. wrote:
Didn't we get stuck in the reverse direction. We never had it fully, but I'm not remembering it --Amr

On Apr 25, 2015, at 8:27 AM, Jacques Carette <carette@mcmaster.ca> wrote:

Right. We have one direction, from Pi combinators to (Vec) permutations. The other seems easier to wor

Note that quite a bit of the code has (already!!) bit-rotted. I changed the definition of PiLevel0 to m

We do not have the other direction currently if we need

Jacques

On 2015-04-25 7:28 AM, Sabry, Amr A. wrote:
That's obsolete for now

By the way, do we have a complement to thm2 that connec

On Apr 24, 2015, at 5:25 PM, Jacques Carette <carette@m

Is that going somewhere, or is it an experiment that sh
Jacques

Thanks. I like that idea ;).

I have a bunch of things I need to do, so I won't reall

I understand the desire to not want to rely on the full

As I was trying really hard to come up with a single st

On 2015-04-23 9:07 PM, Sabry, Amr A. wrote:
Instead of discussing this over and over, I think it is

On Apr 23, 2015, at 6:07 PM, Amr Sabry <sabry@indiana.e

I wasn't too worried about the symmetric vs. non-symmet

I do recall the other discussion about extensionality.

I just really want to avoid the full reliance on the co

On 04/23/2015 12:23 PM, Jacques Carette wrote:
Did you see my "HoTT-agda" question on the Agda mailing
list, and Dan Licata's reply?

What you wrote reduces to our definition of *equivalenc
permutation. To prove that equivalence, we would need
question of February 18th on the Agda mailing list.

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

Now, we may want another representation of permutations
functions (qua bijections) internally instead of vector
answer to your question would be "yes", modulo the ques
which boils down to equivalence of equivalences.

Jacques

On 2015-04-23 10:32 AM, Sabry, Amr A. wrote:
Though a bit more about this. We need a little bridge
our code and we're good to go I think.

In HoTT we have several notions of equivalence that are
the technical sense). The one that is Pi-ish is the
following:

A ≃ B if exists f : A → B such that:
  (exists g : B → A with g o f ~ idA) X
  (exists h : B → A with f o h ~ idB)

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

--Amr

On Apr 21, 2015, at 11:03 AM, Jacques Carette <carette@mcmaster.ca> wrote:

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, 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.

Jacques

On 2015-04-21 10:38 AM, Sabry, Amr A. wrote:
I think that I should start trying to write down a more detailed 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 source of theory for 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 equivalence (at higher levels too) in sync with the HoTT definitions seems to be a good thing to do. --Amr

... and if these coherence conditions are really complete then it should be the case of the two, pi-combinat

So to sum up we would get a nice language for expressing equivalences between finite types and a normal

--Amr

On 04/27/2015 06:16 AM, Sabry, Amr A. wrote:
Here is a nice idea: we need a canonical form for every pi-combinator. Our previous approach gave us som

Indeed! Good idea.

However, it may not give us a normal form. This is because quite a few 'simplifications' require to use

In other words, because we have associativity and commutativity, we need to deal with those specially.

However, I think it is not that bad: we can use the objects to help. We also had put the objects [aka t

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 thin
4. show this terminates

the issue is that the re-ordering could produce new * a
Jacques

On 2015-04-27 6:16 AM, Sabry, Amr A. wrote:
Here is a nice idea: we need a canonical form for every

I've been thinking about this some more. I can't help

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

Another place to look is in Fiore (et al?)'s proof of c

On 2015-04-26 6:34 AM, Sabry, Amr A. wrote:
What's the proof strategy for establishing that a CPerm

Well enough. Last talk on the last day, so people are

I think the idea that (reversible circuits == proof ter

If we had a similar story for Caley+T (as they like to

Note that I've pushed quite a few things forward in the

Yes, I think this can make a full paper -- especially o

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

Writing it up actually forced me to add PiEquiv.agda to

Firstly, thanks Spencer for setting this up.

This is partly a response to Amr, and partly my own tak

One of the key ingredients to getting diagrammatic lang

by enumerations or not should help quite a bit). If you ignore these theorems and insist on working with

Of course, when it comes to computing with diagrams, th

(1: combinatoric) its a graph with some extra bells and
(2: syntactic) its a convenient way of writing down som
(3: "lego" style) its a collection of tiles, connected

Point of view (1) is basically what Quantomatic is buil

Naiively, point of view (2) is that a diagram represent

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

This eliminates the need for the interchange law, but k

This is a very good example of CCT. As I am sure that y

My primary CCT interest, so far, has been with what I c

There's also the perspective that string diagrams of va

From that perspective, the string diagrams for traced m

Yes, I am sure this observation has been made before.

[And since monoidal categories are involved in knot theory, this is un-surprising from that angle as wel
Also, Tarmo Uustalu's "Coherence for skew-monoidal cate
On 2015-06-02 7:53 PM, Sabry, Amr A. wrote:
looking at that 2path picture... if these were physical wires and boxes, we could twist [illegible]
[Apparently I could have saved myself some of that [illegible]

There are some slightly different approaches to implementing a category as an axiom system which
Somehow, at the end of the day, it seems we're looking [illegible]

A category can be formalized as a kind of elementary axiom system using a language with two sorts, map a

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. C

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 a

For most of the systems that I have looked at the axioms are often " rules", such as the category axioms

A morphism of an axiom set using constructors is a functor. When the axioms include products and powers

With this representation of a category using axioms in the "constructor" logic, the axioms and their the

'm writing you offline for the moment, just to see whether I am understanding what you would like. In sh

We are in some sense categorifying the notion of "commutative rig". The role of commutative monoid is ca

I believe there is a canonical candidate for the categorification of tensor product of commutative monoi

If S is the 2-category of symmetric monoidal categories, strong symmetric monoidal functors, and monoida

In any symmetric monoidal 2-category, we have a notion of "pseudo-commutative pseudomonoid", which gener

(\otimes: C @ C --> C, U: I --> C, etc.)

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

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].
Jacques

On 2015-05-17 8:01 AM, Sabry, Amr A. wrote:
Something closer to our work http://www.informatik.uni-bremen.de/agra/doc/konf/rc15_ricercar.pdf

--Amr

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 a


2