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 and on permutations with the goal of reducing the notion of finite type equivalence to a calculus of permutations.

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 from 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 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). 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 $m = \mathit{suc}\ m'$ and $n = \mathit{suc}\ n'$. 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 $(\mathit{suc}\ m')$ and Fin $(\mathit{suc}\ n')$, 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 $(\mathit{suc}\ m')$ 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.

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 Fin m and 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.

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 3. *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 $\mathrm{PERM}\ m\ n$.*

Proof. Although long and tedious, this proof is really straightforward. Amr says: say more □

With the proper Agda definitions, we can rephrase the theorem in a way that is more evocative of the phrasing of the *univalence* axiom.

Theorem 4.

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

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 5. *The equivalence of Theorem 4 is an* isomorphism *between the commutative semiring of equivalences of finite types and the commutative semiring of permutations.*

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).

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

Typed Isomorphisms

First, a universe of (finite) types

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

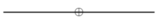
and its interpretation

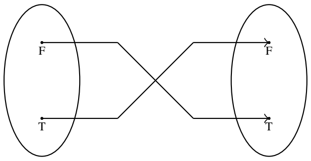
```
⟦_⟧ : U → Set
⟦ ZERO ⟧          = ⊥
⟦ ONE ⟧           = ⊤
⟦ PLUS t₁ t₂ ⟧    = ⟦ t₁ ⟧ ⊎ ⟦ t₂ ⟧
⟦ TIMES t₁ t₂ ⟧ = ⟦ t₁ ⟧ × ⟦ t₂ ⟧
```

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

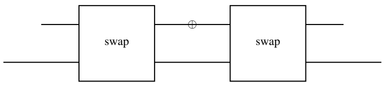


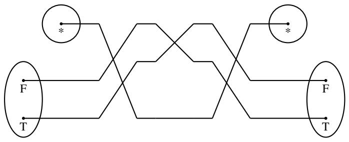


```
BOOL : U
BOOL = PLUS ONE ONE

n₁ : BOOL ⟷ BOOL
n₁ = swap₊
```

Example Circuit: Not So Simple Negation.





```
n₂ : BOOL ⟷ BOOL
n₂ =     uniti⋆ ⊙
```

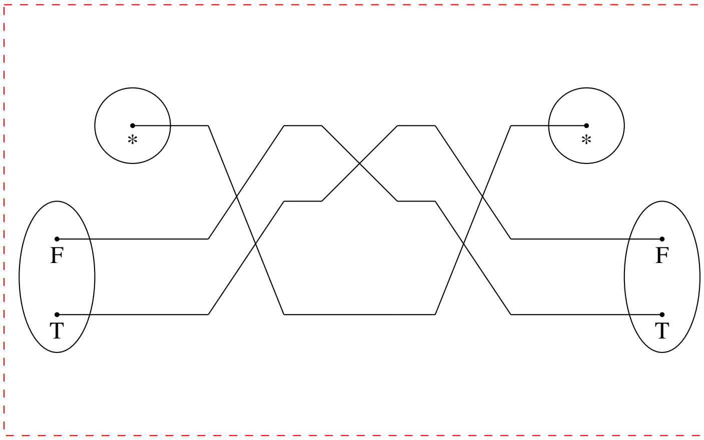
```
swap⋆ ⊙
(swap₊ ⊗ id⟷) ⊙
swap⋆ ⊙
unite⋆
```

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

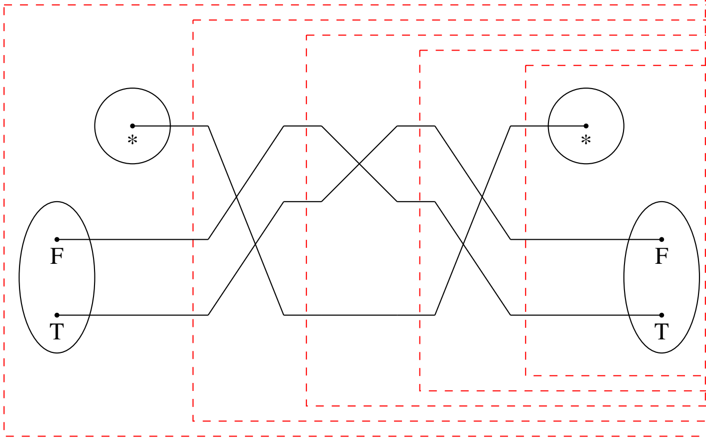
```
negEx : n₂ ⇔ n₁
negEx = uniti⋆ ⊙ (swap⋆ ⊙ ((swap₊ ⊗ id⟷) ⊙ (swap⋆ ⊙ unite⋆)))
          ⇔⟨ id⇔ ⊡ assoc⊙l ⟩
      uniti⋆ ⊙ ((swap⋆ ⊙ (swap₊ ⊗ id⟷)) ⊙ (swap⋆ ⊙ unite⋆))
          ⇔⟨ id⇔ ⊡ (swapl⋆⇔ ⊡ id⇔) ⟩
      uniti⋆ ⊙ (((id⟷ ⊗ swap₊) ⊙ swap⋆) ⊙ (swap⋆ ⊙ unite⋆))
          ⇔⟨ id⇔ ⊡ assoc⊙r ⟩
   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

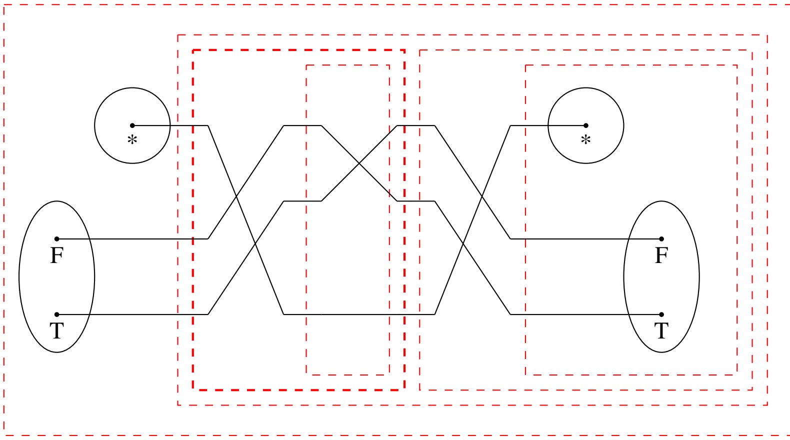
Original circuit:



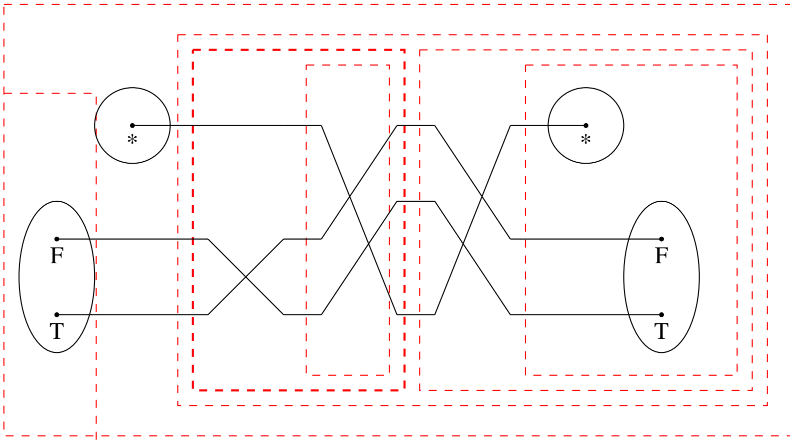
Making grouping explicit:



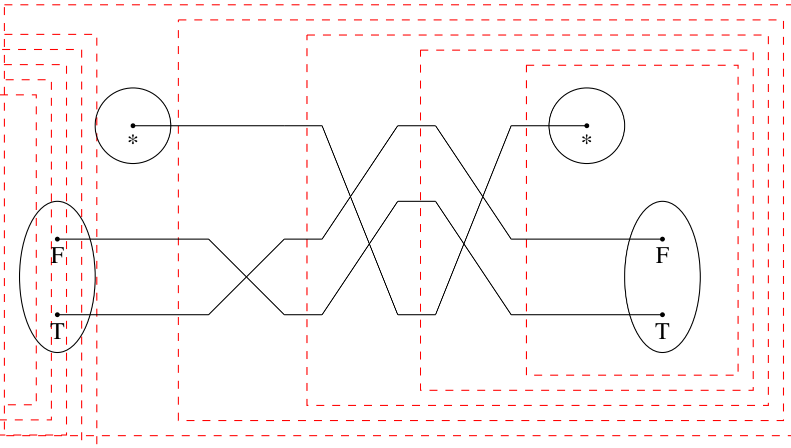
By associativity:



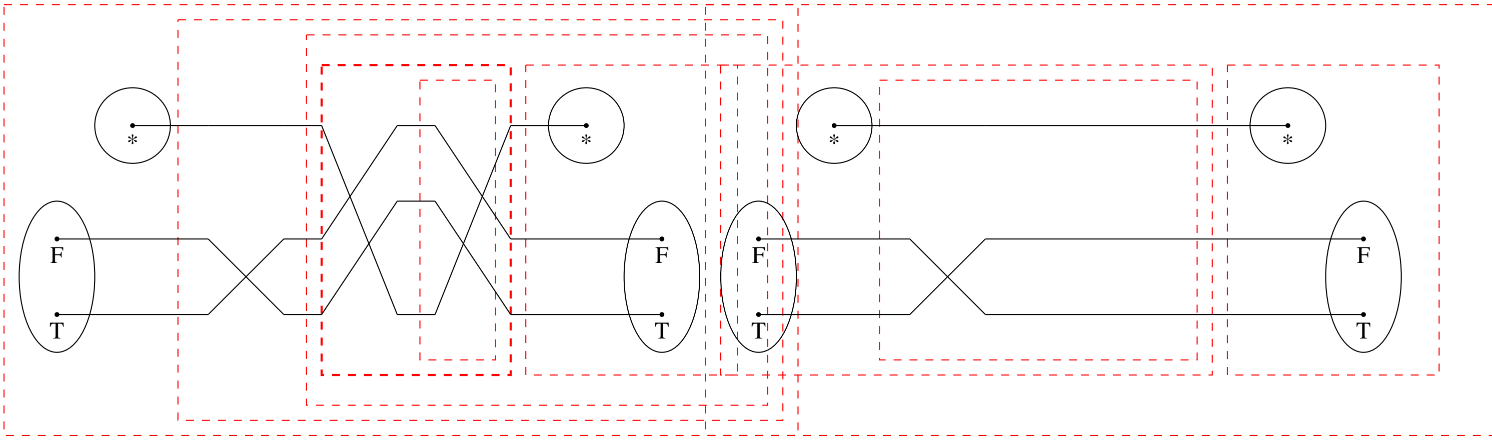
By pre-post-swap:



By associativity:

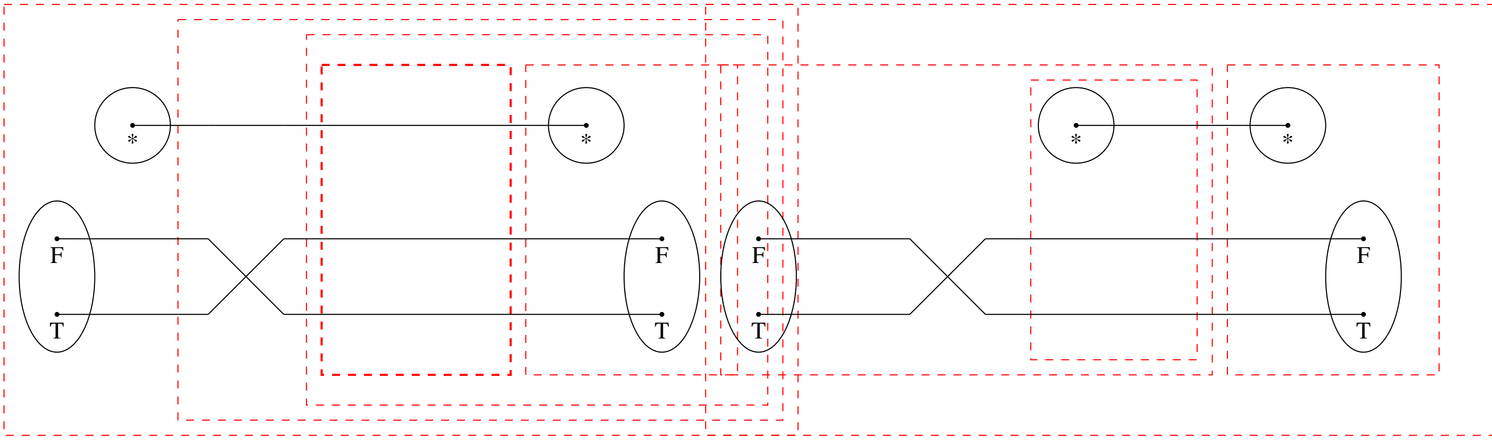


By associativity:



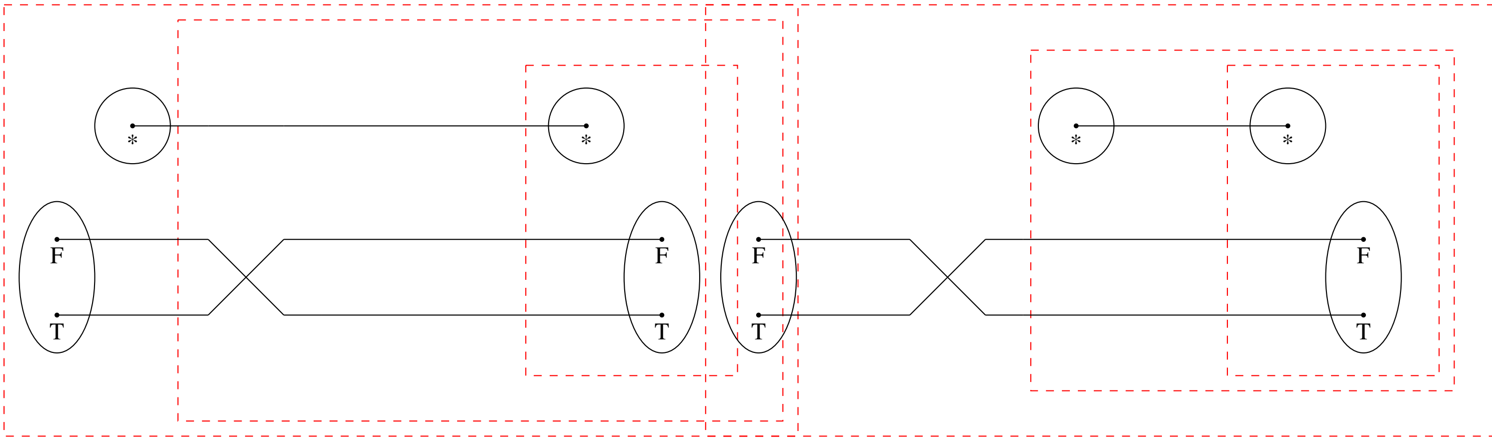
By swap-swap:

By swap-unit:



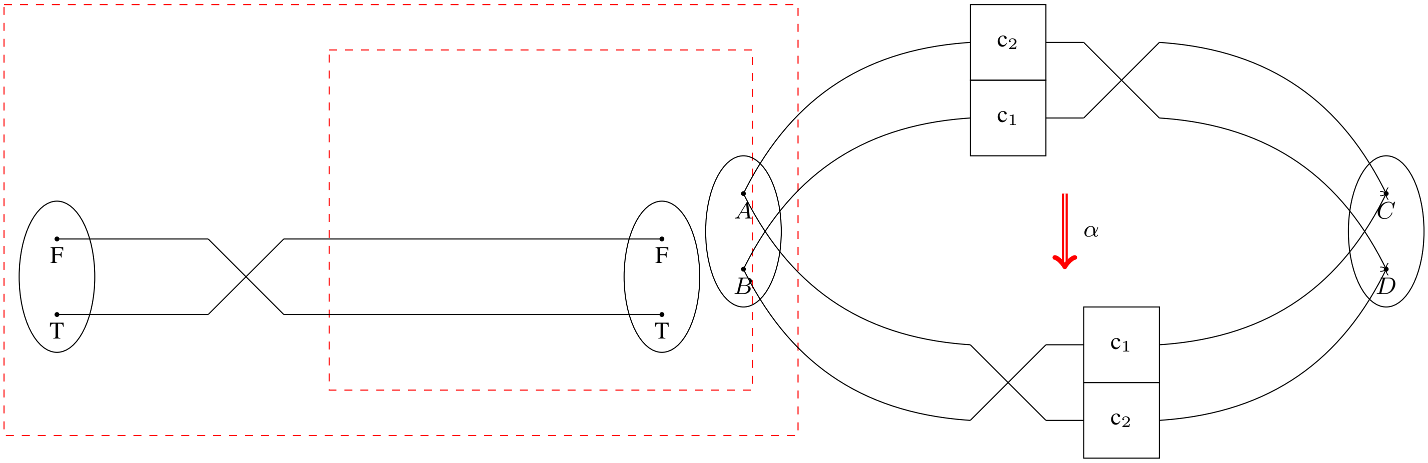
By id-compose-left:

By associativity:

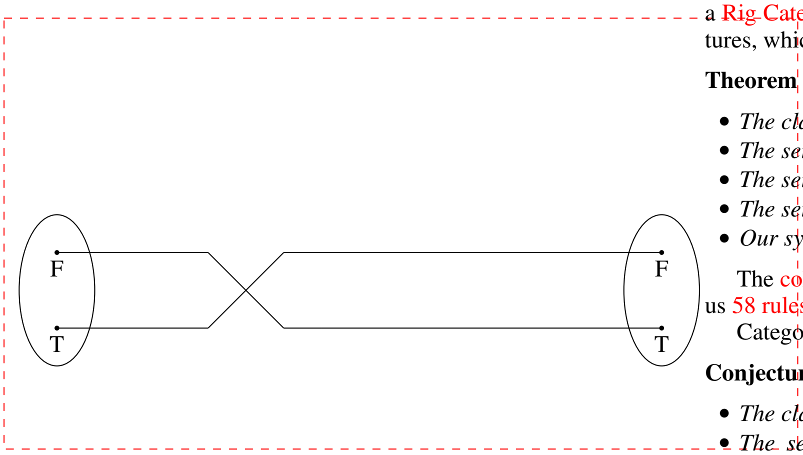


By associativity:

By unit-unit:



By id-unit-right:



5. But is this a programming language?

We get forward and backward evaluators $\mathsf{eval} : \{t_1\ t_2 : \mathsf{U}\} \to (t_1 \longleftrightarrow t_2) \to [\![\ t_1\ ]\!] \to [\![\ t_2\ ]\!]$

$\mathsf{evalB} : \{t_1\ t_2 : \mathsf{U}\} \to (t_1 \longleftrightarrow t_2) \to [\![\ t_2\ ]\!] \to [\![\ t_1\ ]\!]$

which really do behave as expected $\mathsf{c2equiv} : \{t_1\ t_2 : \mathsf{U}\} \to (c : t_1 \longleftrightarrow t_2) \to [\![\ t_1\ ]\!] \simeq [\![\ t_2\ ]\!]$

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-constructi

Also,
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1
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 con

The story without trace and without the Int constructio

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 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 can show 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

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

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-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)

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 that our work will rest on the

Jacques

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

(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

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

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

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

Separately, 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 construc

On Apr 10, 2015, at 11:04 AM, Jacques Carette <carette@

Reminder of
http://mathoverflow.net/questions/106070/int-constructi

Also,
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1
seems relevant

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

On 2015-04-10 10:52 AM, Amr Sabry wrote:
I just checked and found no traced categories or Int con

The story is weaker without the Int constructio

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 se
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. Th
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 b
implementing the Int Construction and showing that a li
h.o. functions emerges and leave the big open problem o
the multiplication etc. for later work. I can start wor
will require adding traced categories and then a generi
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. 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 hott... 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, or am I misremembering?? --Amr

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

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 m

We do not have the other direction currently in the code. That may not be too bad, as we do have LeftCa

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 connects to Pi. Ideally what we want to say is what I s

On Apr 24, 2015, at 5:25 PM, Jacques Carette <carette@mcmaster.ca> wrote:

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

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

As I was trying really hard to come up with a single story, I am a little confused as to what "my" story

On 2015-04-23 9:07 PM, Sabry, Amr A. wrote:
Instead of discussing this over and over, I think it is clear that thm2 will be an important part of any

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

I wasn’t too worried about the symmetric vs. non-symmetric notions of equivalence. The HoTT book has more

I do recall the other discussion about extensionality. That is also concluded with the idea that the

I just really want to avoid the full reliance on the coherence conditions. I also noted you have a diffe

--Amr

On 04/23/2015 12:23 PM, Jacques Carette wrote:
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*, all
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
(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 encoding of equivalence we use.

Jacques

On 2015-04-23 10:32 AM, Sabry, Amr A. wrote:
Thought 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 seems easiest to work with 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 p
and B are finite sets?

--Amr

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

I’m ok with a HoTT bias, but concerned that our code do
match that. But since we have no specific deadline, I
bit more time isn’t too bad.

Since propositional equivalence is really HoTT equivale
I am not too concerned about that side of things -- our
permutations should be the same whether in HoTT or in r
with various notions of equivalence, especially since som
code was lifted from a previous HoTT-based attempt at t
I would certainly agree with the not-not-statement: usi
equivalence known to be incompatible with HoTT is not a
Jacques

On 2015-04-21 10:38 AM, Sabry, Amr A. wrote:
I think that I should start trying to write down a more
story so that we can see how things fit together. I am
towards a HoTT-related story which is what I started. If
we should have a different initial bias let me know.
What is there is just one paragraph for now but it alre
question: if we are pursuing that HoTT story we should
prove that the HoTT notion of equivalence when speciali
types reduces to permutations. That should be a strong
as to the precise notion of permutation we get (
by enumerations or not should help quite a bit).

More generally always keeping our notions of equivalenc
levels in-sync with the HoTT definitions seems to
thing to do. --Amr

...and if these coherence conditions are really comple

So to sum up we would get a nice language for ex[illegible]

--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 o[illegible]
2. expand things out, with + being outer, * middle, . inner.
3. within each . term, use combinators to re-orde[illegible]
4. show this terminates

the issue is that the re-ordering could produce ne[illegible]

Jacques

On 2015-04-27 6:16 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

I've been thinking about this some more. I can't help but think that, somehow, Laplaza has already work

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 d

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

Well enough. Last talk on the last day, so peopl[illegible]

I think the idea that (reversible circuits == pro[illegible]

If we had a similar story for Caley+T (as they li[illegible]

Note that I've pushed quite a few things forward [illegible]

Yes, I think this can make a full paper -- especi[illegible]

I think the details are fine. A little bit of po[illegible]

Writing it up actually forced me to add PiEquiv.a[illegible]

Firstly, thanks Spencer for setting this up.

This is partly a response to Amr, and partly my o[illegible]

One of the key ingredients to getting diagrammati[illegible]

If you ignore these theorems and insist on workin[illegible]

Of course, when it comes to computing with diagra[illegible]

(1: combinatoric) its a graph with some extra bells and whistles
(2: syntactic) its a convenient way of writing do[illegible]
(3: "lego" style) its a collection of tiles, conn[illegible]

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.

[illegible]

On 2015-06-02 7:53 PM, Sabry, Amr A. wrote:

[illegible]

There are some slightly different approaches to impleme

A category can be formalized as a kind of elementary ax

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

is used for the three place predicate.

The operations such as the binary composition of maps a

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

[illegible]

In any symmetric monoidal 2-category, we have a notion

[illegible]

Jacques

On 2015-05-17 8:01 AM, Sabry, Amr A. wrote:
[illegible] http://www.informatik.uni-

--Amr

More related work (as I encountered them, but later

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

A Homotopical Completion Procedure with Application
http://drops.dagstuhl.de/opus/frontdoor.php?source_

A really nice set of slides that illustrates both o
http://www.lix.polytechnique.fr/Labo/Samuel.Mimram/

I think there is something very important going on
http://comp.mq.edu.au/~rgarner/Papers/Glynn.pdf
which I also attach. [I googled 'Knuth Bendix cohe

There are also seems to be relevant stuff buried (v

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

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 $a, b, c \in R$:*

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

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