Representing, Manipulating and Optimizing Reversible Circuits

Jacques Carette
McMaster University
carette@mcmaster.ca

Amr Sabry
Indiana University
sabry@indiana.edu

Abstract

...

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.

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

```
⟦_⟧ : U → Set
```

```
⟦ ZERO ⟧        = ⊥
⟦ ONE ⟧         = ⊤
⟦ PLUS t₁ t₂ ⟧  = ⟦ t₁ ⟧ ⊎ ⟦ t₂ ⟧
⟦ TIMES t₁ t₂ ⟧ = ⟦ t₁ ⟧ × ⟦ t₂ ⟧
```

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

(whose *constructive* proof is quite subtle).

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 $\mathsf{Perm}\,n$.*

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 2 is an isomorphism between the semirings of equivalences of finite types, and of permutations.*

A more evocative phrasing might be:

Theorem 5.

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

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.

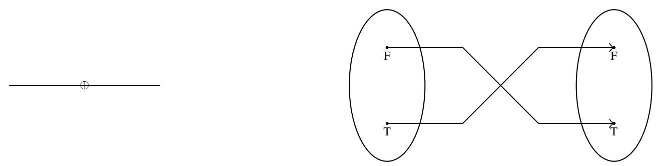
In more detail, it is a general problem that, despite its fundamental value, formal semantics of programming languages is generally inaccessible to the computing public. As Schmidt argues in a recent position statement on strategic directions for research on programming languages [**?**]:

> ...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.

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) 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 t₃) (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 ⟷ t₃) → (t₂ ⟷ t₄) → (PLUS t₁ t₂ ⟷ PLUS t₃ t₄)
  _⊗_      : {t₁ t₂ t₃ t₄ : U} → (t₁ ⟷ t₃) → (t₂ ⟷ t₄) → (TIMES t₁ t₂ ⟷ TIMES t₃ t₄)
```

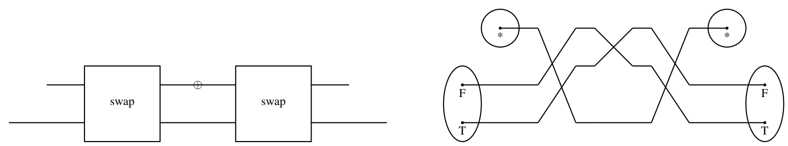
3. Example Circuit: Simple Negation




```
BOOL : U
BOOL = PLUS ONE ONE

n₁ : BOOL ⟷ BOOL
n₁ = swap₊
```

Example Circuit: Not So Simple Negation.

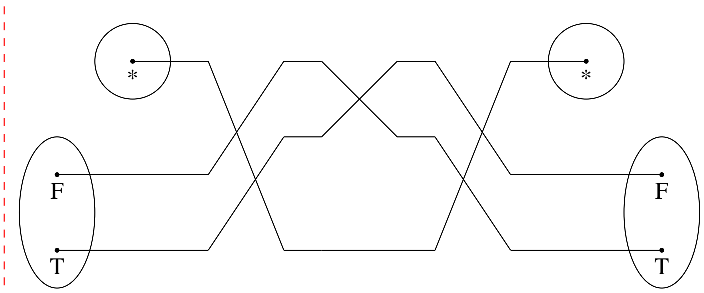



```
n₂ : BOOL ⟷ BOOL
n₂ =   uniti⋆ ⊙
       swap⋆ ⊙
       (swap₊ ⊗ id⟷) ⊙
       swap⋆ ⊙
       unite⋆
```

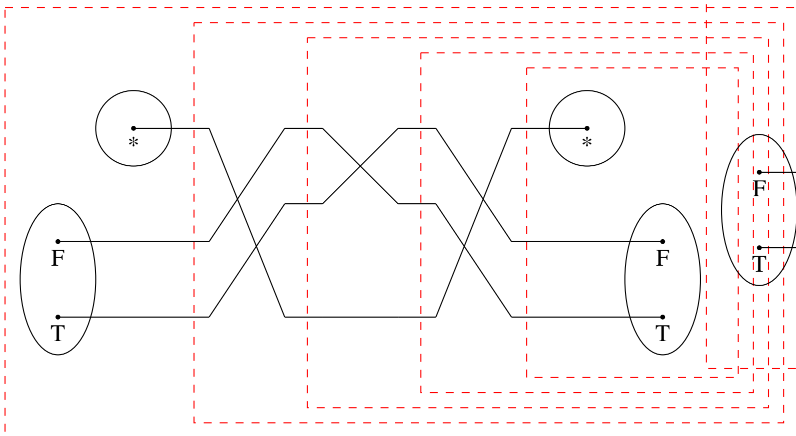
Reasoning about Example Circuits. Algebraic manipulation of one circuit to the other:

```
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₊ □
```

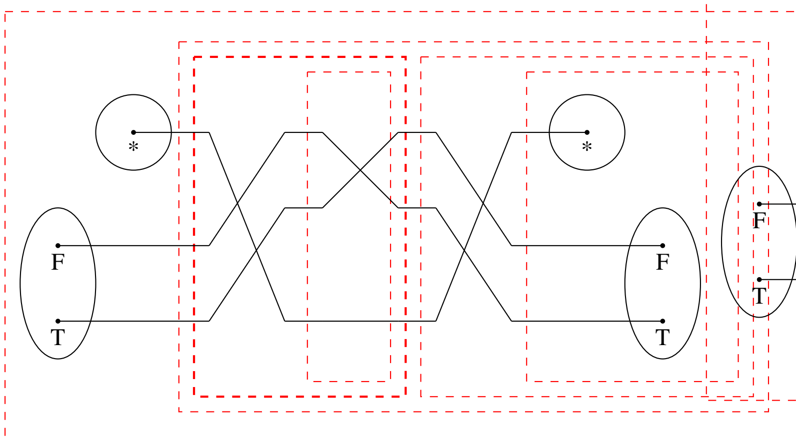
Visually.
Original circuit:



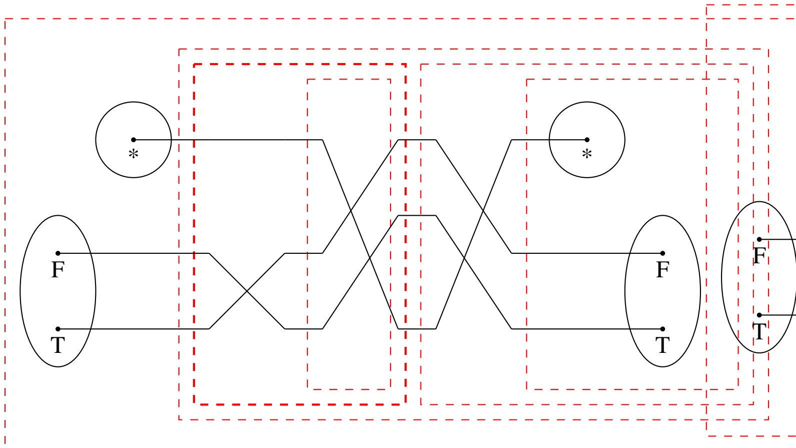
Making grouping explicit:



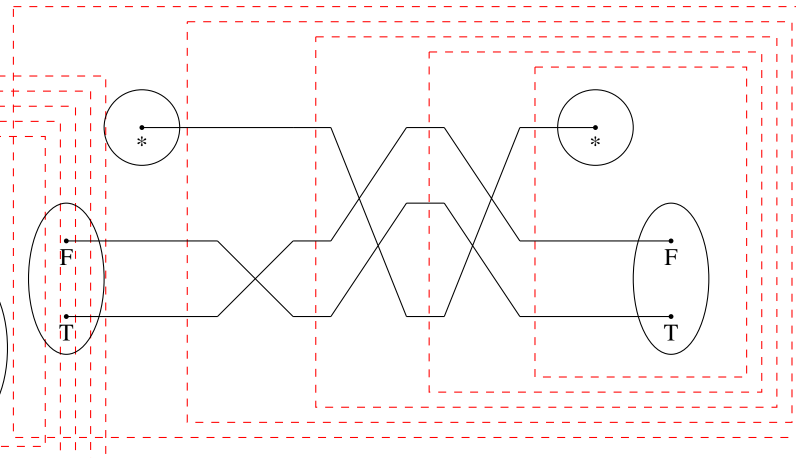
By associativity:



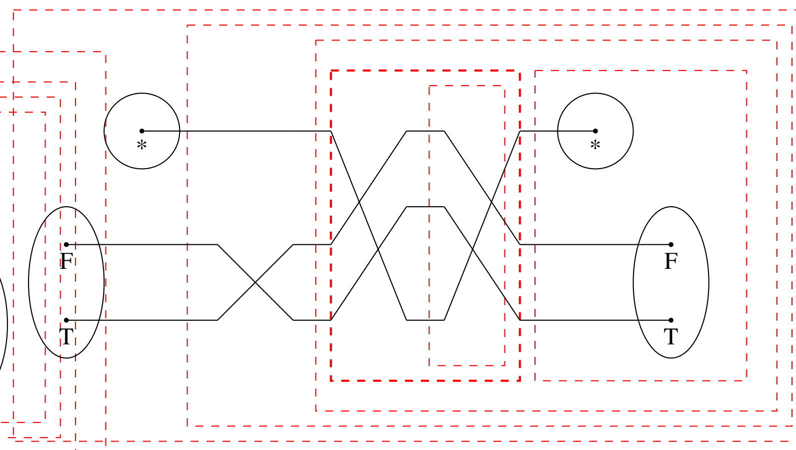
By pre-post-swap:



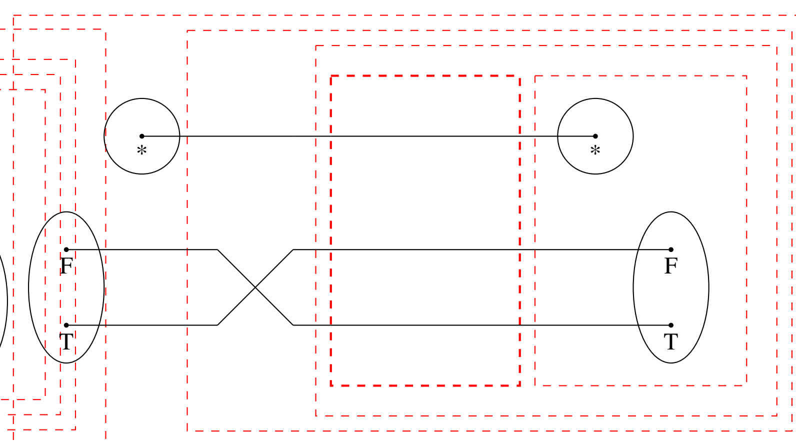
By associativity:



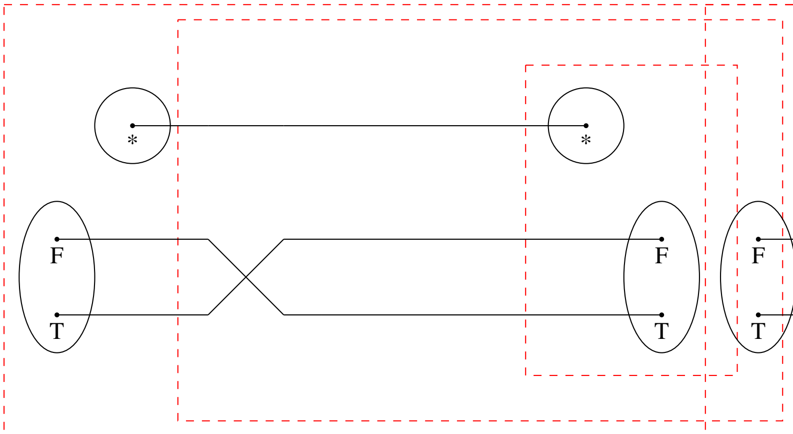
By associativity:



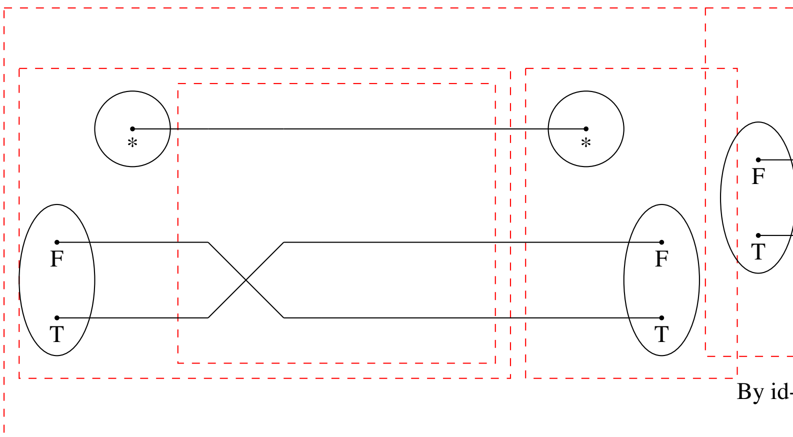
By swap-swap:



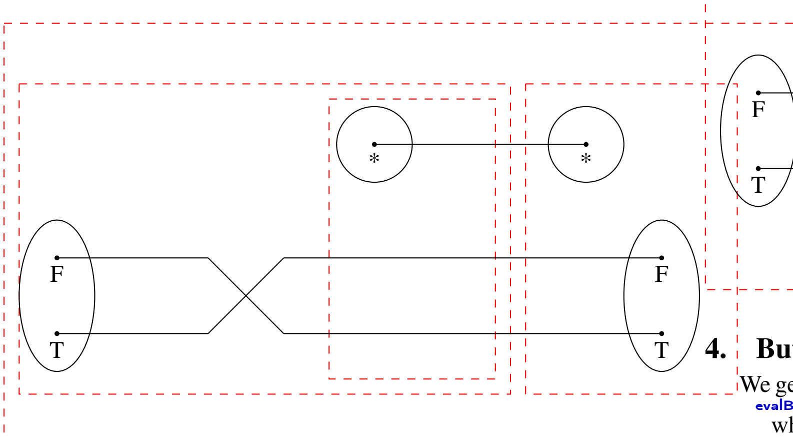
By id-compose-left:



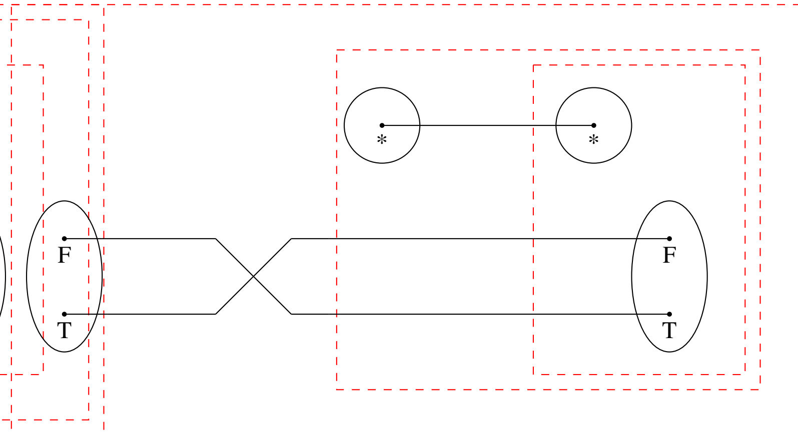
By associativity:



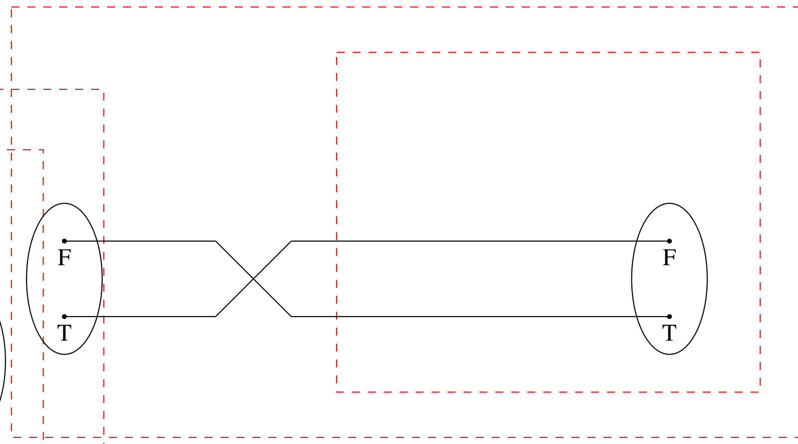
By swap-unit:



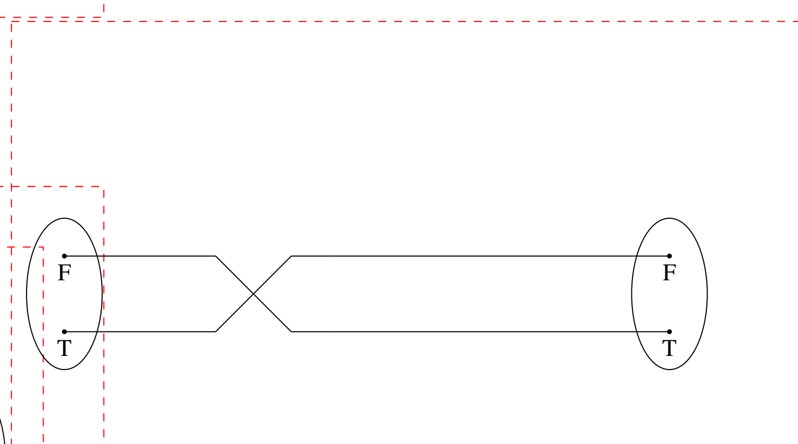
By associativity:



By unit-unit:



By id-unit-right:



4. But is this a programming language?

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

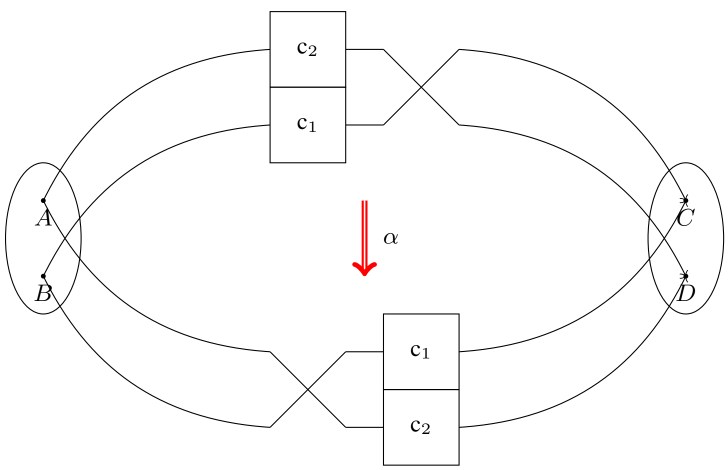
5. Categorification I

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
    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.*