

Representing, Manipulating and Optimizing Reversible Circuits

Jacques Carette Amr Sabry

McMaster University

Indiana University

June 11, 2015

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**.

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.

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

Starting Point

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).

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)

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.*

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

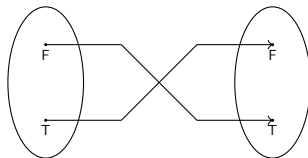
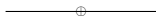
First conclusion: it might be useful to *reify* a (sound and complete) set of equivalences as combinators, such as the fundamental “proof rules” of semirings:

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  $\longleftrightarrow$  :  $\mathbf{U} \rightarrow \mathbf{U} \rightarrow \mathbf{Set}$  where
  unite+ : {t :  $\mathbf{U}$ }  $\rightarrow$  PLUS ZERO t  $\longleftrightarrow$  t
  uniti+ : {t :  $\mathbf{U}$ }  $\rightarrow$  t  $\longleftrightarrow$  PLUS ZERO t
  swap+ : {t1 t2 :  $\mathbf{U}$ }  $\rightarrow$  PLUS t1 t2  $\longleftrightarrow$  PLUS t2 t1
  associ+ : {t1 t2 t3 :  $\mathbf{U}$ }  $\rightarrow$  PLUS t1 (PLUS t2 t3)  $\longleftrightarrow$  PLUS (PLUS t1 t2) t3
  associ+* : {t1 t2 t3 :  $\mathbf{U}$ }  $\rightarrow$  PLUS (PLUS t1 t2) t3  $\longleftrightarrow$  PLUS t1 (PLUS t2 t3)
  unite* : {t :  $\mathbf{U}$ }  $\rightarrow$  TIMES ONE t  $\longleftrightarrow$  t
  uniti* : {t :  $\mathbf{U}$ }  $\rightarrow$  t  $\longleftrightarrow$  TIMES ONE t
  swap* : {t1 t2 :  $\mathbf{U}$ }  $\rightarrow$  TIMES t1 t2  $\longleftrightarrow$  TIMES t2 t1
  associ* : {t1 t2 t3 :  $\mathbf{U}$ }  $\rightarrow$  TIMES t1 (TIMES t2 t3)  $\longleftrightarrow$  TIMES (TIMES t1 t2) t3
  associ** : {t1 t2 t3 :  $\mathbf{U}$ }  $\rightarrow$  TIMES (TIMES t1 t2) t3  $\longleftrightarrow$  TIMES t1 (TIMES t2 t3)
  absorbr : {t :  $\mathbf{U}$ }  $\rightarrow$  TIMES ZERO t  $\longleftrightarrow$  ZERO
  absorbl : {t :  $\mathbf{U}$ }  $\rightarrow$  TIMES t ZERO  $\longleftrightarrow$  ZERO
  factorzr : {t :  $\mathbf{U}$ }  $\rightarrow$  ZERO  $\longleftrightarrow$  TIMES t ZERO
  factorzl : {t :  $\mathbf{U}$ }  $\rightarrow$  ZERO  $\longleftrightarrow$  TIMES ZERO t
  dist : {t1 t2 t3 :  $\mathbf{U}$ }  $\rightarrow$  TIMES (PLUS t1 t2) t3  $\longleftrightarrow$  PLUS (TIMES t1 t3) (TIMES t2 t3)
  factor : {t1 t2 t3 :  $\mathbf{U}$ }  $\rightarrow$  PLUS (TIMES t1 t3) (TIMES t2 t3)  $\longleftrightarrow$  TIMES (PLUS t1 t2) t3
  id  $\longleftrightarrow$  : {t :  $\mathbf{U}$ }  $\rightarrow$  t  $\longleftrightarrow$  t
   $\ominus$  : {t1 t2 t3 :  $\mathbf{U}$ }  $\rightarrow$  (t1  $\longleftrightarrow$  t2)  $\rightarrow$  (t2  $\longleftrightarrow$  t3)  $\rightarrow$  (t1  $\longleftrightarrow$  t3)
   $\oplus$  : {t1 t2 t3 t4 :  $\mathbf{U}$ }  $\rightarrow$  (t1  $\longleftrightarrow$  t3)  $\rightarrow$  (t2  $\longleftrightarrow$  t4)  $\rightarrow$  (PLUS t1 t2  $\longleftrightarrow$  PLUS t3 t4)
   $\otimes$  : {t1 t2 t3 t4 :  $\mathbf{U}$ }  $\rightarrow$  (t1  $\longleftrightarrow$  t3)  $\rightarrow$  (t2  $\longleftrightarrow$  t4)  $\rightarrow$  (TIMES t1 t2  $\longleftrightarrow$  TIMES t3 t4)
```

Example Circuit: Simple Negation



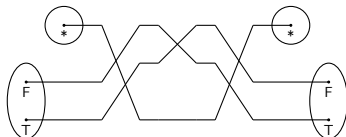
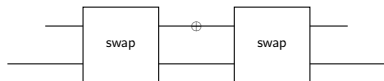
BOOL : U

BOOL = PLUS ONE ONE

$n_1 : \text{BOOL} \longleftrightarrow \text{BOOL}$

$n_1 = \text{swap}_+$

Example Circuit: Not So Simple Negation



$n_2 : \text{BOOL} \longleftrightarrow \text{BOOL}$

$n_2 =$ $\text{uniti} \star \odot$
 $\text{swap} \star \odot$
 $(\text{swap}_+ \otimes \text{id} \longleftrightarrow) \odot$
 $\text{swap} \star \odot$
 $\text{unite} \star$

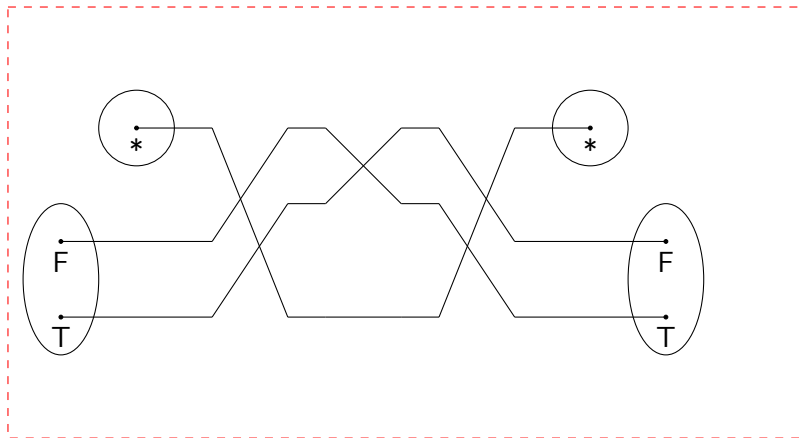
Reasoning about Example Circuits

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*
      ⇔ ( uniti* ⇔ □ id ⇔ )
      (swap+ ∘ uniti*) ∘ unite*
      ⇔ ( assoc⊙r )
      swap+ ∘ (uniti* ∘ unite*)
      ⇔ ( id ⇔ □ linv⊙l )
      swap+ ∘ id⟷
      ⇔ ( idr⊙l )
      swap+ □
```

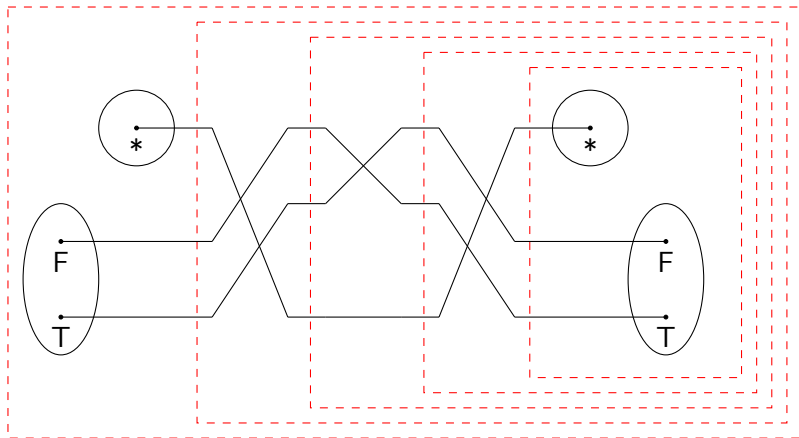
Visually

Original circuit:



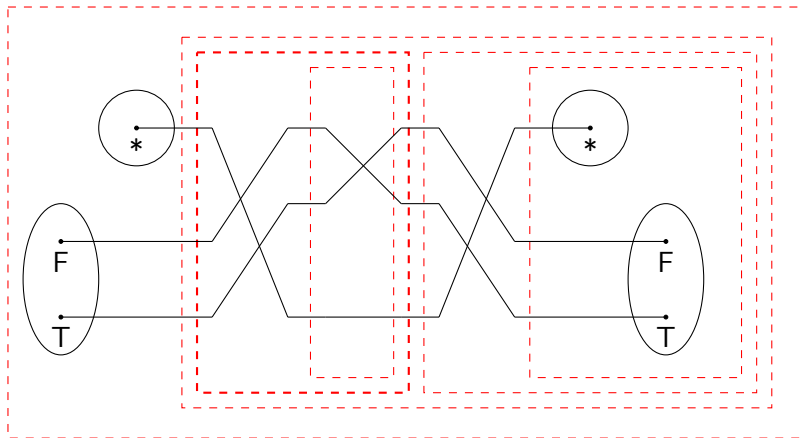
Visually

Making grouping explicit:



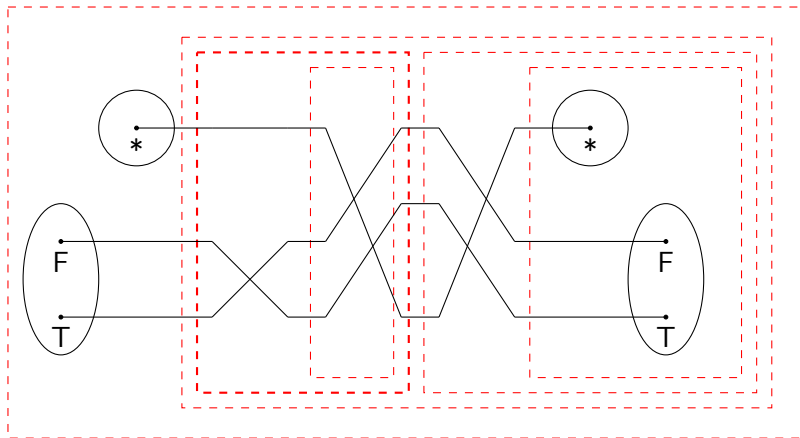
Visually

By associativity:



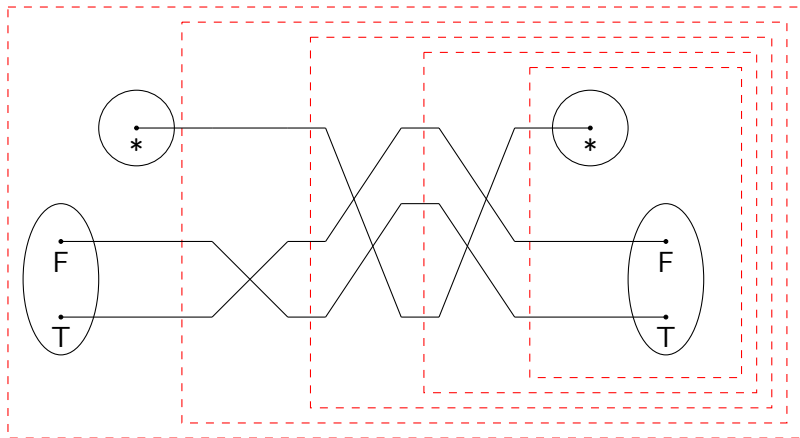
Visually

By pre-post-swap:



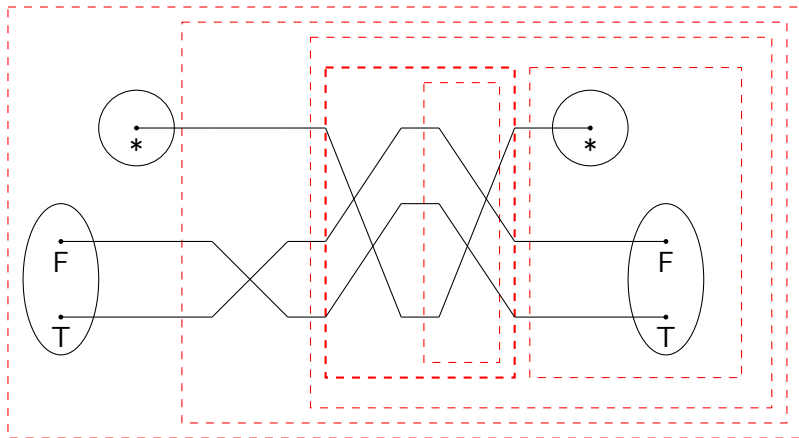
Visually

By associativity:



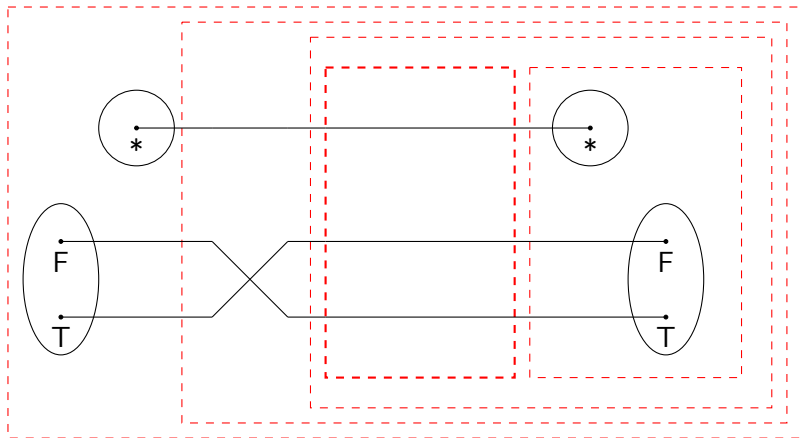
Visually

By associativity:



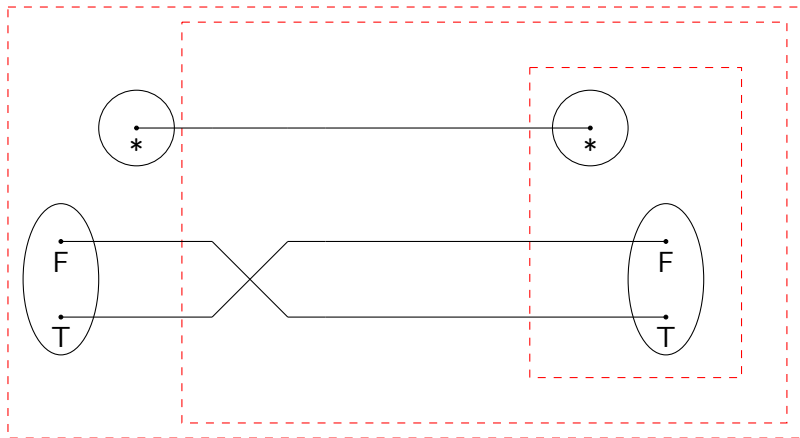
Visually

By swap-swap:



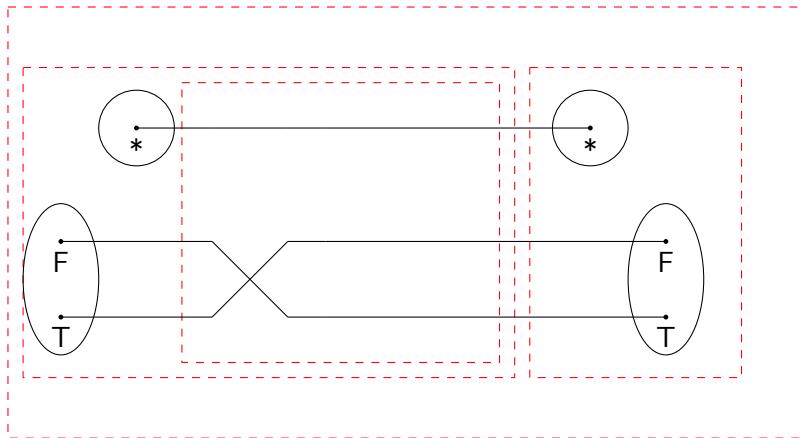
Visually

By id-compose-left:



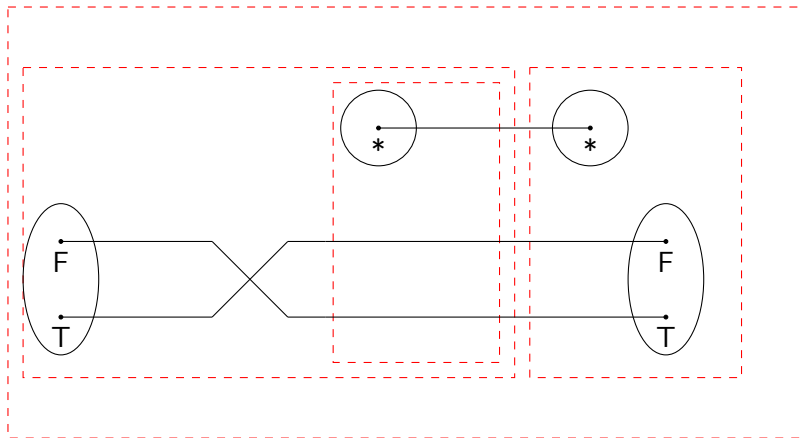
Visually

By associativity:



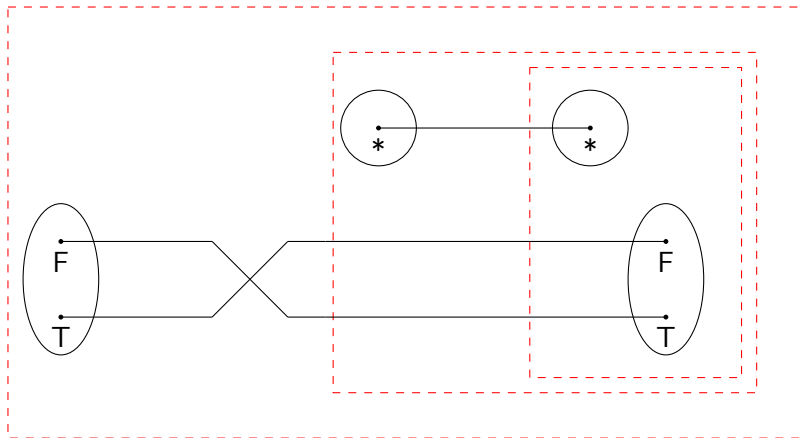
Visually

By swap-unit:



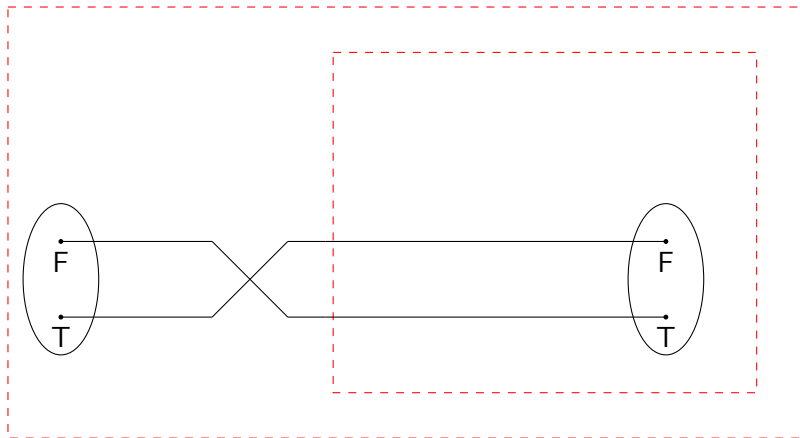
Visually

By associativity:



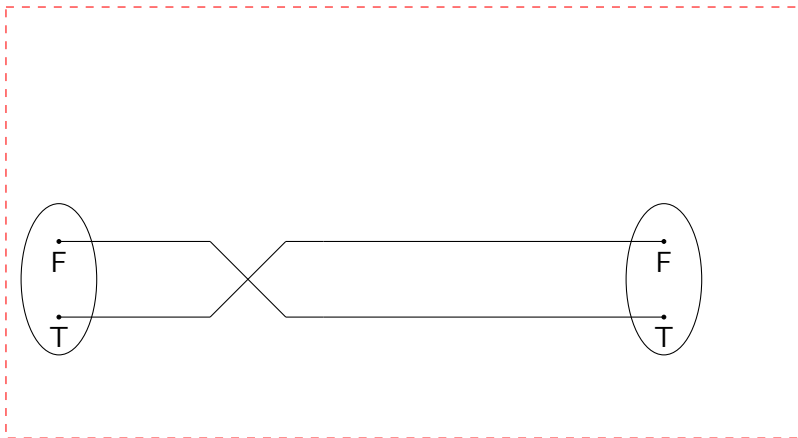
Visually

By unit-unit:



Visually

By id-unit-right:



But is this a programming language?

We get forward and backward evaluators

$$\begin{aligned}\text{eval} &: \{t_1 \ t_2 : \mathbf{U}\} \rightarrow (t_1 \longleftrightarrow t_2) \rightarrow \llbracket t_1 \rrbracket \rightarrow \llbracket t_2 \rrbracket \\ \text{evalB} &: \{t_1 \ t_2 : \mathbf{U}\} \rightarrow (t_1 \longleftrightarrow t_2) \rightarrow \llbracket t_2 \rrbracket \rightarrow \llbracket t_1 \rrbracket\end{aligned}$$

But is this a programming language?

We get forward and backward evaluators

$$\begin{aligned}\text{eval} &: \{t_1 \ t_2 : \mathbf{U}\} \rightarrow (t_1 \longleftrightarrow t_2) \rightarrow \llbracket t_1 \rrbracket \rightarrow \llbracket t_2 \rrbracket \\ \text{evalB} &: \{t_1 \ t_2 : \mathbf{U}\} \rightarrow (t_1 \longleftrightarrow t_2) \rightarrow \llbracket t_2 \rrbracket \rightarrow \llbracket t_1 \rrbracket\end{aligned}$$

which really do behave as expected

$$\text{c2equiv} : \{t_1 \ t_2 : \mathbf{U}\} \rightarrow (c : t_1 \longleftrightarrow t_2) \rightarrow \llbracket t_1 \rrbracket \simeq \llbracket t_2 \rrbracket$$

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)

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

$$c_1 : \{B \ C : \mathbf{U}\} \rightarrow B \longleftrightarrow C$$

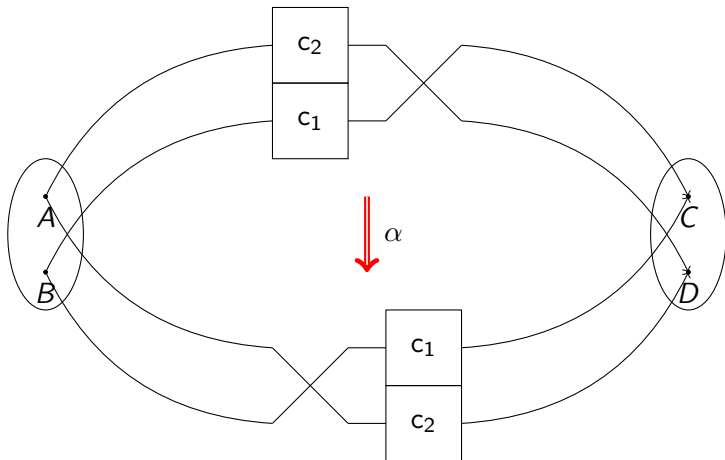
$$c_2 : \{A \ D : \mathbf{U}\} \rightarrow A \longleftrightarrow D$$

$$p_1 \ p_2 : \{A \ B \ C \ D : \mathbf{U}\} \rightarrow \text{PLUS } A \ B \longleftrightarrow \text{PLUS } C \ D$$

$$p_1 = \text{swap}_+ \odot (c_1 \oplus c_2)$$

$$p_2 = (c_2 \oplus c_1) \odot \text{swap}_+$$

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

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
- The set of permutations
- The set of equivalences between finite types

Categorification III

Conjecture 1.

The following are *Symmetric Rig Groupoids*:

- The class of all types (*Set*)
- The set of all finite types
- The set of permutations
- The set of equivalences between finite types

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*.