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

$$
\begin{aligned}
&[\![\_]\!] : \mathsf{U} \to \mathsf{Set}\\
&[\![\ \mathsf{ZERO}\ ]\!] && = \bot\\
&[\![\ \mathsf{ONE}\ ]\!] && = \top\\
&[\![\ \mathsf{PLUS}\ t_1\ t_2\ ]\!] && = [\![\ t_1\ ]\!] \uplus [\![\ t_2\ ]\!]\\
&[\![\ \mathsf{TIMES}\ t_1\ t_2\ ]\!] && = [\![\ t_1\ ]\!] \times [\![\ t_2\ ]\!]
\end{aligned}
$$

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

(whose *constructive* proof is quite subtle).

Theorem 3.

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

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.

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

A Calculus of Permutations

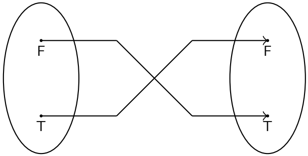
First conclusion: it might be useful to *reify* a (sound and complete) set of equivalences as combinators, such as the fundamental "proof rules" of semirings:

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 _⟷_ : 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)
```

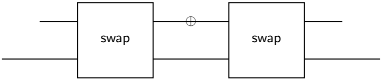
Example Circuit: Simple Negation

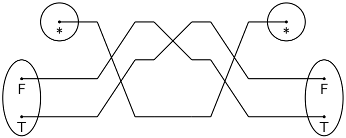


$\mathsf{BOOL} : \mathsf{U}$
$\mathsf{BOOL} = \mathsf{PLUS\ ONE\ ONE}$

$\mathsf{n}_1 : \mathsf{BOOL} \longleftrightarrow \mathsf{BOOL}$
$\mathsf{n}_1 = \mathsf{swap}_+$

Example Circuit: Not So Simple Negation





$$\begin{aligned} n_2 &: \mathsf{BOOL} \longleftrightarrow \mathsf{BOOL} \\ n_2 &= \ \mathsf{uniti}\star \odot \\ &\quad\ \mathsf{swap}\star \odot \\ &\quad\ (\mathsf{swap}_+ \otimes \mathsf{id}{\longleftrightarrow}) \odot \\ &\quad\ \mathsf{swap}\star \odot \\ &\quad\ \mathsf{unite}\star \end{aligned}$$

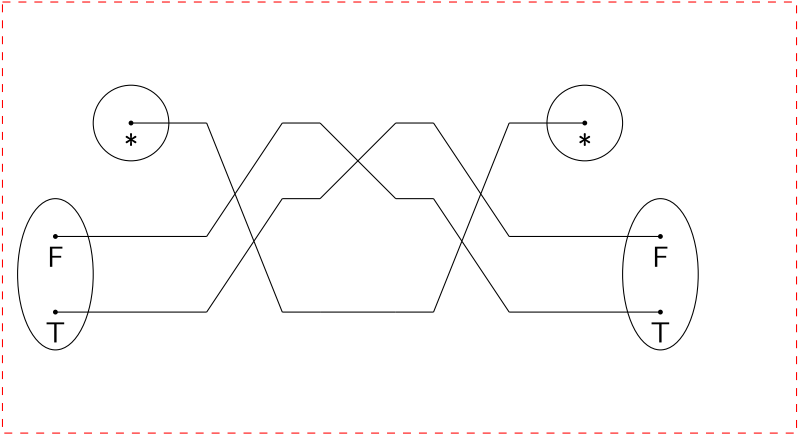
Reasoning about Example Circuits

Algebraic manipulation of one circuit to the other:

$$
\begin{array}{l}
\mathsf{negEx} : \mathsf{n_2} \Leftrightarrow \mathsf{n_1} \\
\mathsf{negEx} = \mathsf{uniti}\star \odot (\mathsf{swap}\star \odot ((\mathsf{swap}_+ \otimes \mathsf{id}\longleftrightarrow) \odot (\mathsf{swap}\star \odot \mathsf{unite}\star))) \\
\quad\quad \Leftrightarrow\langle\ \mathsf{id}\Leftrightarrow \boxdot \mathsf{assoc}\odot\mathsf{l}\ \rangle \\
\quad \mathsf{uniti}\star \odot ((\mathsf{swap}\star \odot (\mathsf{swap}_+ \otimes \mathsf{id}\longleftrightarrow)) \odot (\mathsf{swap}\star \odot \mathsf{unite}\star)) \\
\quad\quad \Leftrightarrow\langle\ \mathsf{id}\Leftrightarrow \boxdot (\mathsf{swapl}\star\Leftrightarrow \boxdot \mathsf{id}\Leftrightarrow)\ \rangle \\
\quad \mathsf{uniti}\star \odot (((\mathsf{id}\longleftrightarrow \otimes \mathsf{swap}_+) \odot \mathsf{swap}\star) \odot (\mathsf{swap}\star \odot \mathsf{unite}\star)) \\
\quad\quad \Leftrightarrow\langle\ \mathsf{id}\Leftrightarrow \boxdot \mathsf{assoc}\odot\mathsf{r}\ \rangle \\
\quad \mathsf{uniti}\star \odot ((\mathsf{id}\longleftrightarrow \otimes \mathsf{swap}_+) \odot (\mathsf{swap}\star \odot (\mathsf{swap}\star \odot \mathsf{unite}\star))) \\
\quad\quad \Leftrightarrow\langle\ \mathsf{id}\Leftrightarrow \boxdot (\mathsf{id}\Leftrightarrow \boxdot \mathsf{assoc}\odot\mathsf{l})\ \rangle \\
\quad \mathsf{uniti}\star \odot ((\mathsf{id}\longleftrightarrow \otimes \mathsf{swap}_+) \odot ((\mathsf{swap}\star \odot \mathsf{swap}\star) \odot \mathsf{unite}\star)) \\
\quad\quad \Leftrightarrow\langle\ \mathsf{id}\Leftrightarrow \boxdot (\mathsf{id}\Leftrightarrow \boxdot (\mathsf{linv}\odot\mathsf{l} \boxdot \mathsf{id}\Leftrightarrow))\ \rangle \\
\quad \mathsf{uniti}\star \odot ((\mathsf{id}\longleftrightarrow \otimes \mathsf{swap}_+) \odot (\mathsf{id}\longleftrightarrow \odot \mathsf{unite}\star)) \\
\quad\quad \Leftrightarrow\langle\ \mathsf{id}\Leftrightarrow \boxdot (\mathsf{id}\Leftrightarrow \boxdot \mathsf{idl}\odot\mathsf{l})\ \rangle \\
\quad \mathsf{uniti}\star \odot ((\mathsf{id}\longleftrightarrow \otimes \mathsf{swap}_+) \odot \mathsf{unite}\star) \\
\quad\quad \Leftrightarrow\langle\ \mathsf{assoc}\odot\mathsf{l}\ \rangle \\
\quad (\mathsf{uniti}\star \odot (\mathsf{id}\longleftrightarrow \otimes \mathsf{swap}_+)) \odot \mathsf{unite}\star \\
\quad\quad \Leftrightarrow\langle\ \mathsf{unitil}\star\Leftrightarrow \boxdot \mathsf{id}\Leftrightarrow\ \rangle \\
\quad (\mathsf{swap}_+ \odot \mathsf{uniti}\star) \odot \mathsf{unite}\star \\
\quad\quad \Leftrightarrow\langle\ \mathsf{assoc}\odot\mathsf{r}\ \rangle \\
\quad \mathsf{swap}_+ \odot (\mathsf{uniti}\star \odot \mathsf{unite}\star) \\
\quad\quad \Leftrightarrow\langle\ \mathsf{id}\Leftrightarrow \boxdot \mathsf{linv}\odot\mathsf{l}\ \rangle \\
\quad \mathsf{swap}_+ \odot \mathsf{id}\longleftrightarrow \\
\quad\quad \Leftrightarrow\langle\ \mathsf{idr}\odot\mathsf{l}\ \rangle \\
\quad \mathsf{swap}_+\ \square
\end{array}
$$

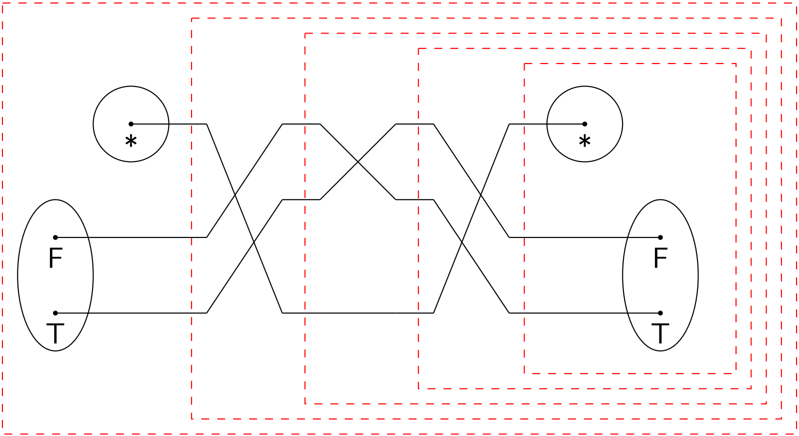
Visually

Original circuit:



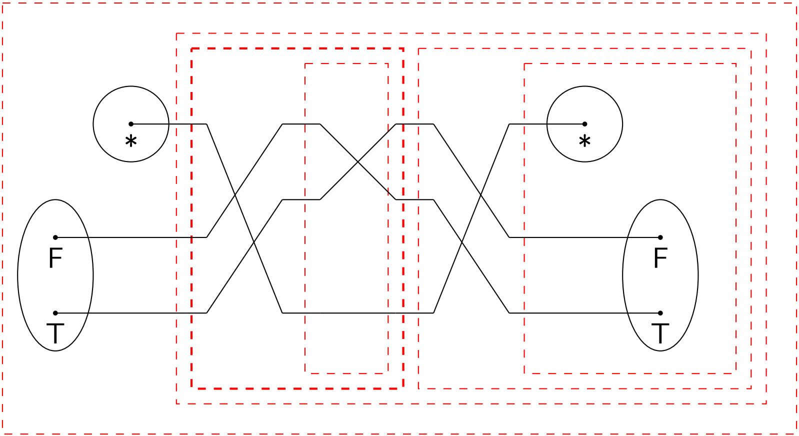
Visually

Making grouping explicit:



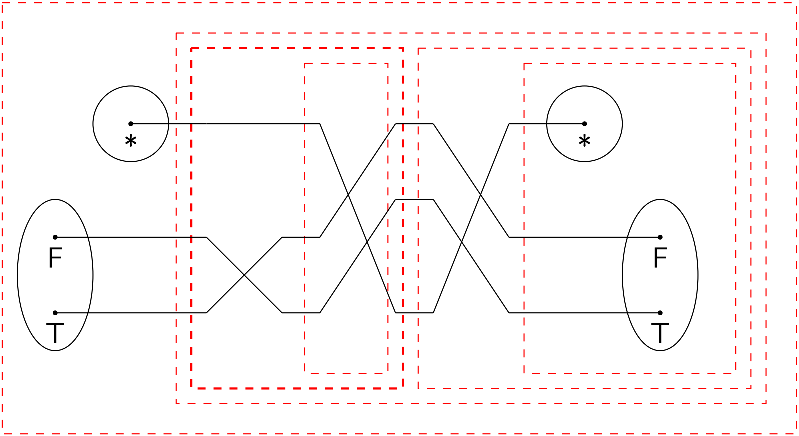
Visually

By associativity:



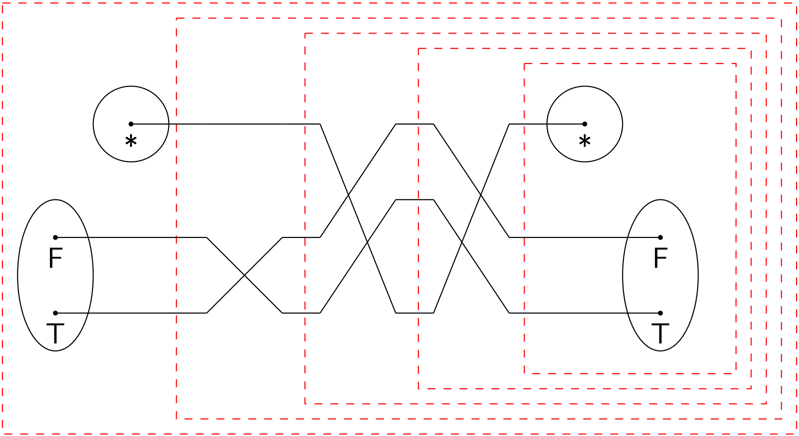
Visually

By pre-post-swap:



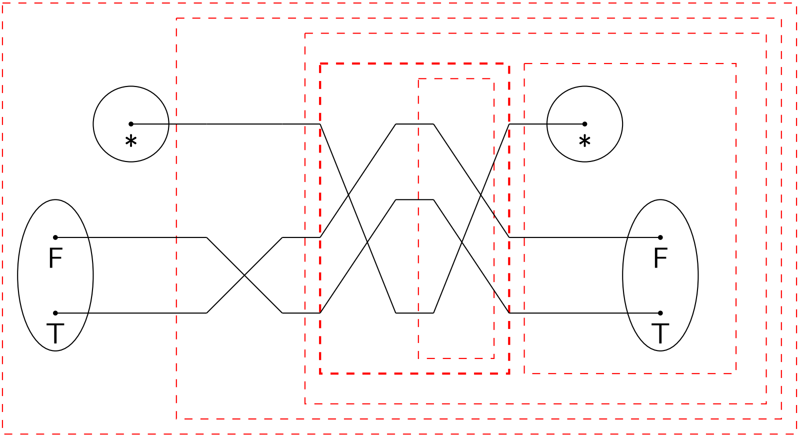
Visually

By associativity:



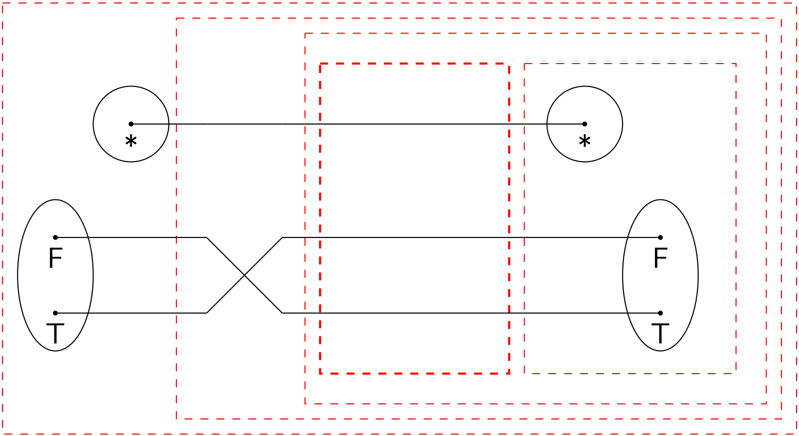
Visually

By associativity:



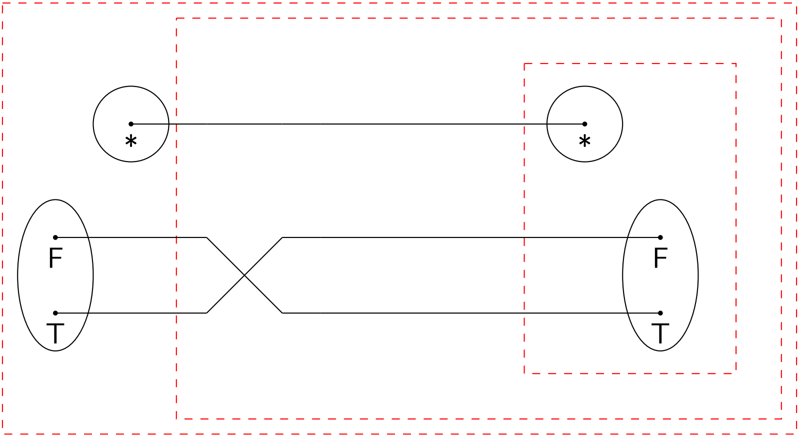
Visually

By swap-swap:



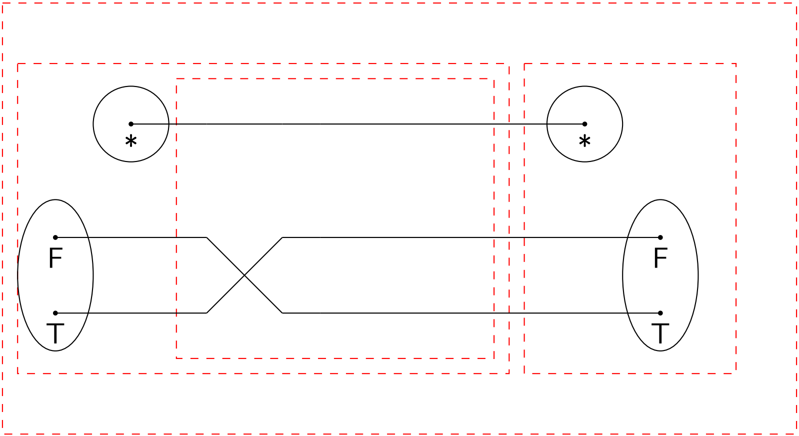
Visually

By id-compose-left:



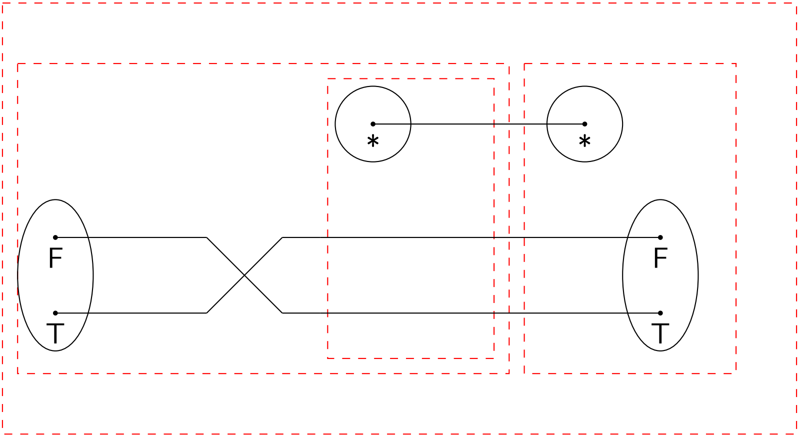
Visually

By associativity:



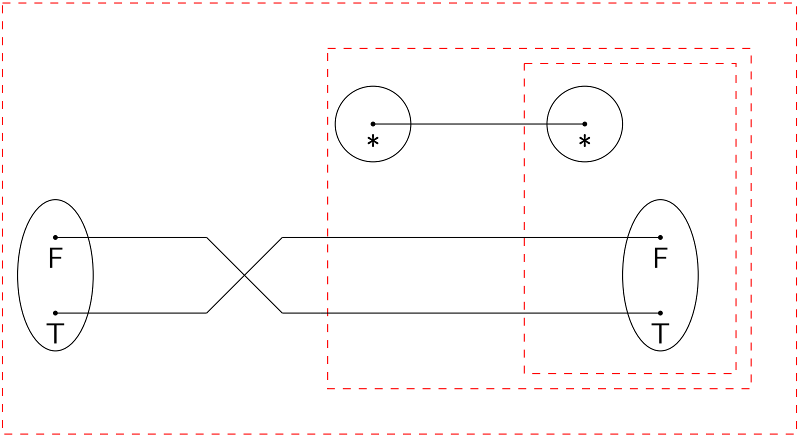
Visually

By swap-unit:



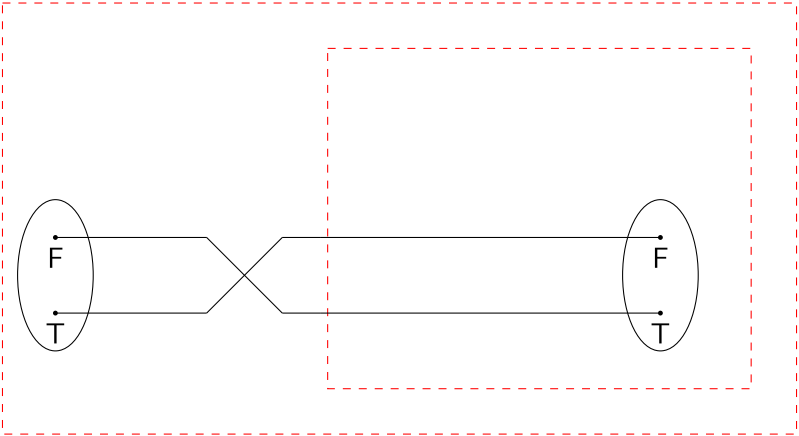
Visually

By associativity:



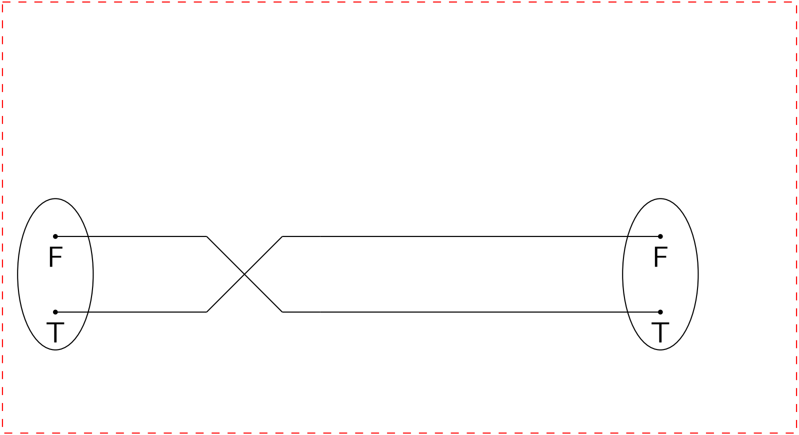
Visually

By unit-unit:



Visually

By id-unit-right:



But is this a programming language?

We get forward and backward evaluators

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

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)

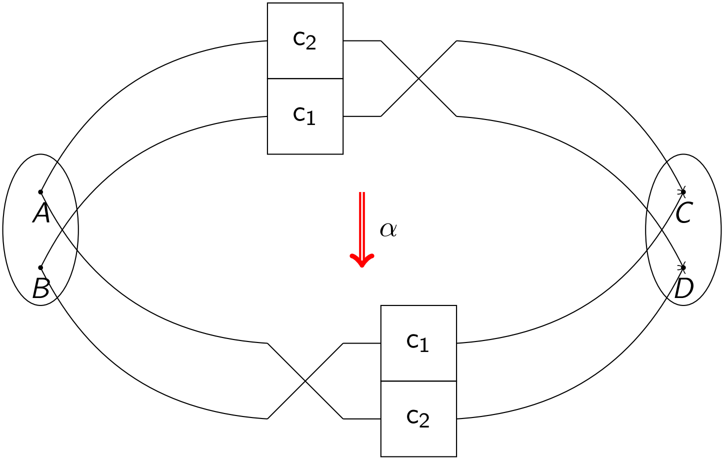
Categorification I

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*

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.