Representing, Manipulating and Optimizing Reversible Circuits

Jacques Carette Amr Sabry

McMaster University

Indiana University

June 11, 2015

Reversible Computing

The “obvious” intersection between quantum computing and programming languages is reversible computing.

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? —JC]

[important remark: these are all *Boolean* circuits! —JC]

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.

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 $[\![\_]\!] : \mathsf{U} \to \mathsf{Set}$

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

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 $\simeq$).

Much more meaningfully, we also get

Theorem 2.

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

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 $\simeq$).

Much more meaningfully, we also get

Theorem 2.

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

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 $\mathrm{Perm}\, n$.

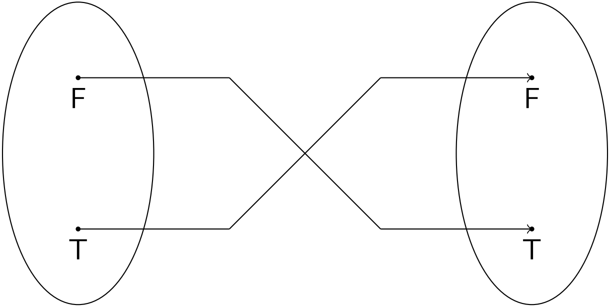
A Calculus of Permutations

In fact, we can say even more. Defining the usual disjoint union and tensor product of permutations, we can prove

Theorem 4.

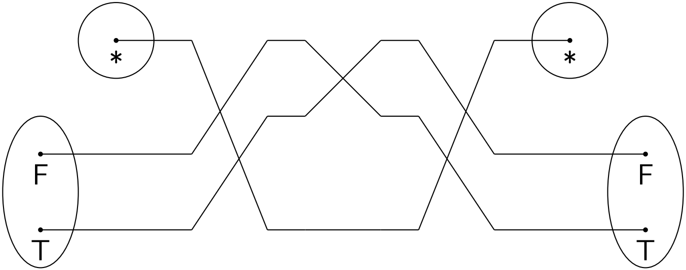
The equivalence of Theorem 3 is an isomorphism of semirings.

Example Circuit: Simple Negation



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

Example Circuit: Not So Simple Negation



$$
\begin{aligned}
\mathsf{n}_2 &: \mathsf{BOOL} \longleftrightarrow \mathsf{BOOL} \\
\mathsf{n}_2 &= \ \mathsf{uniti}\star \odot \\
&\quad\ \mathsf{swap}\star \odot \\
&\quad\ (\mathsf{swap}_+ \otimes \mathsf{id}\longleftrightarrow) \odot \\
&\quad\ \mathsf{swap}\star \odot \\
&\quad\ \mathsf{unite}\star
\end{aligned}
$$

Reasoning about Example Circuits

Algebraic manipulation of one circuit to the other:

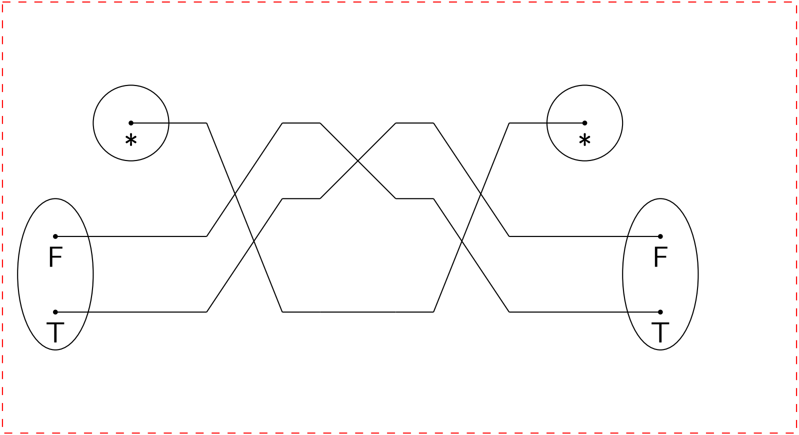
```
open import PiLevel1 hiding (negEx)

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⇔ ⟩
```

Reasoning about Example Circuits

foo

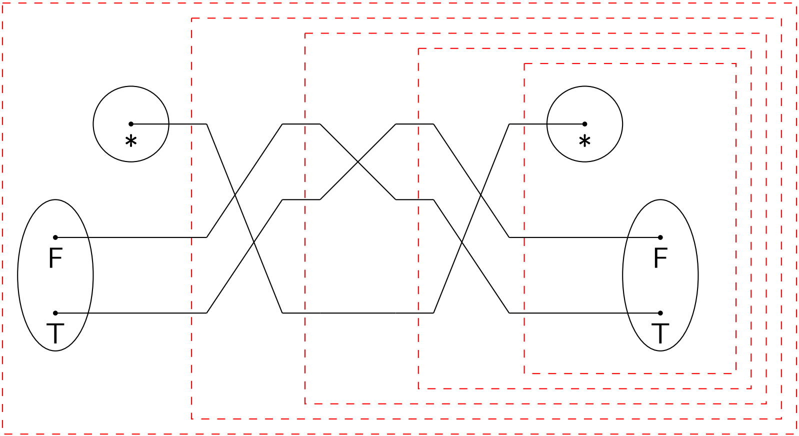
Visually

Original circuit:



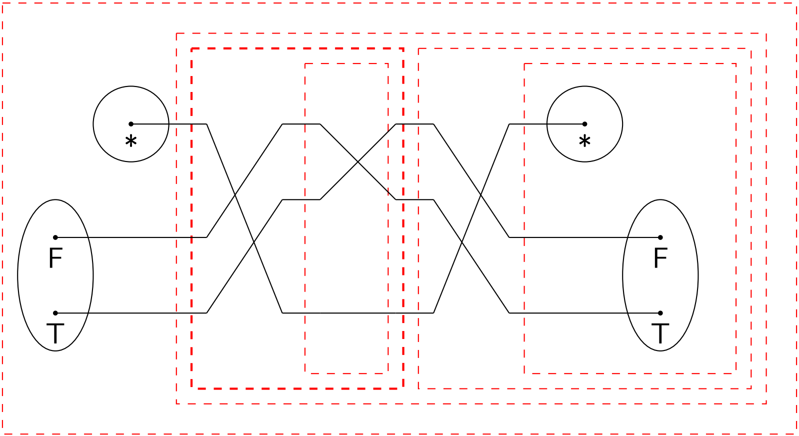
Visually

Making grouping explicit:



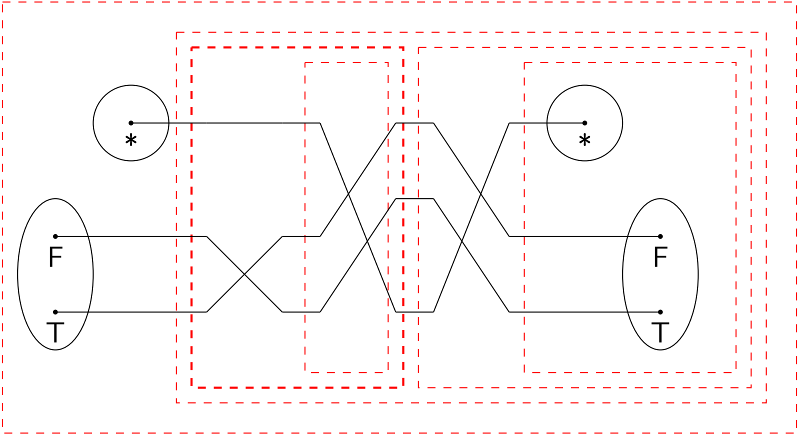
Visually

By associativity:



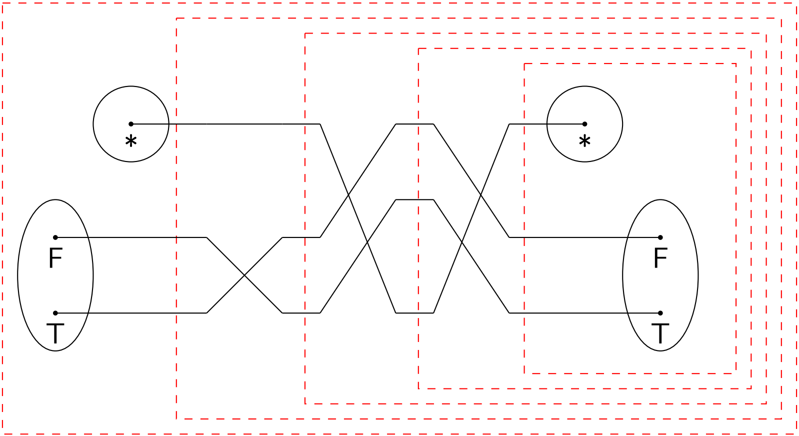
Visually

By pre-post-swap:



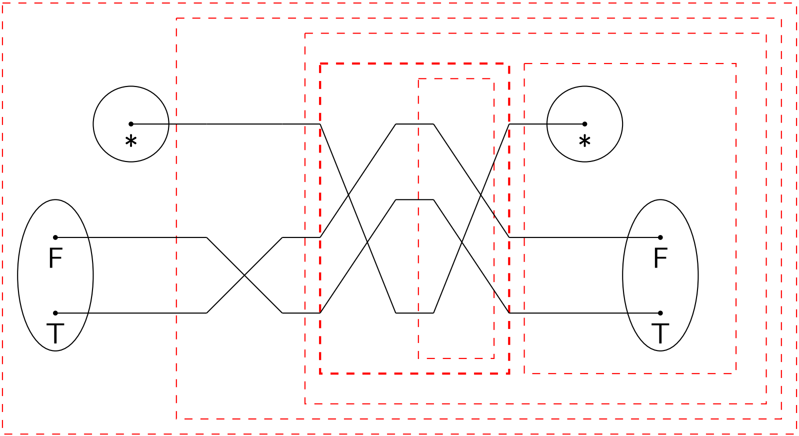
Visually

By associativity:



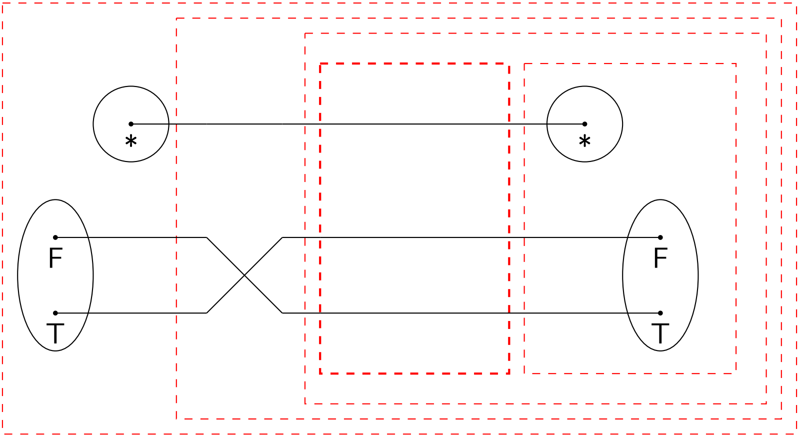
Visually

By associativity:



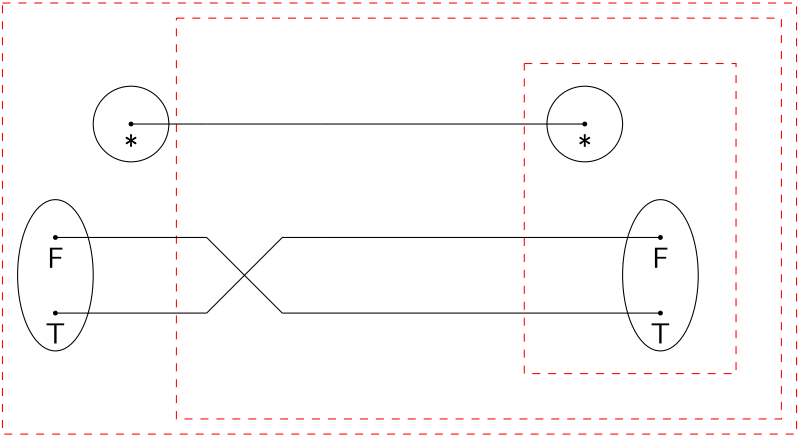
Visually

By swap-swap:



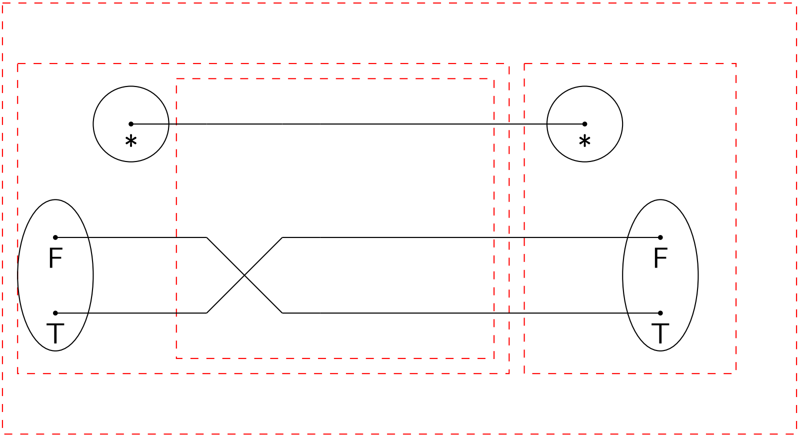
Visually

By id-compose-left:



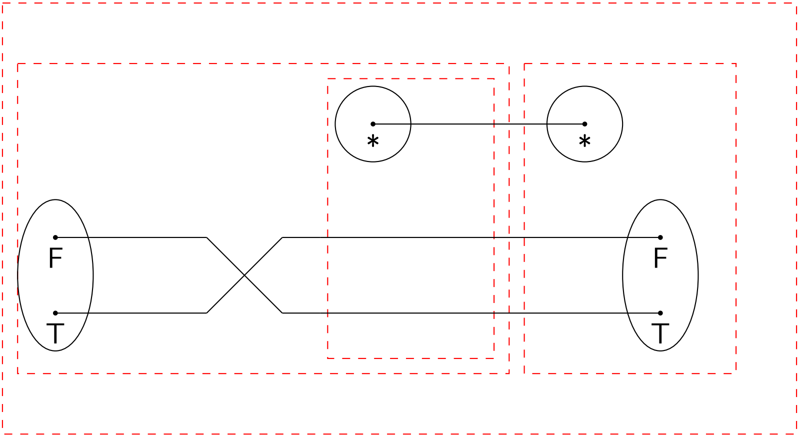
Visually

By associativity:



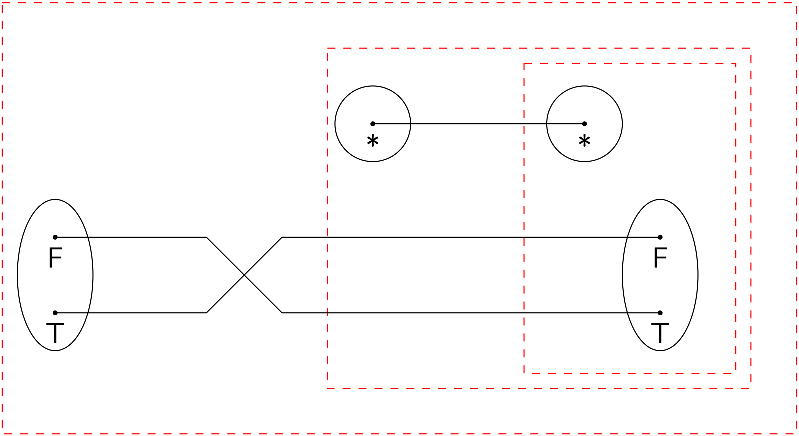
Visually

By swap-unit:



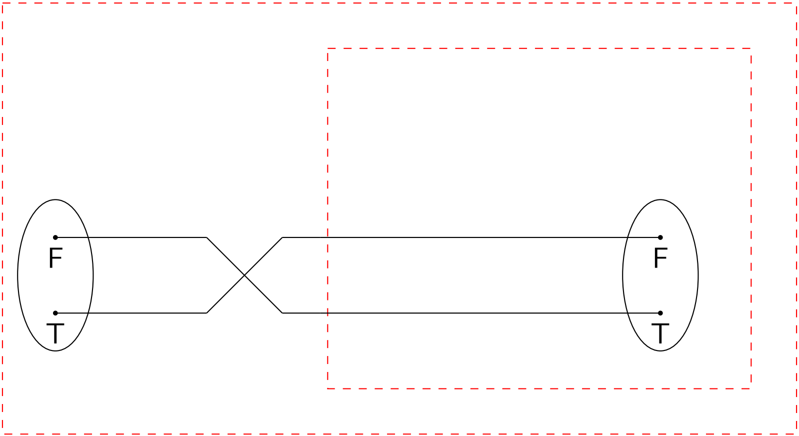
Visually

By associativity:



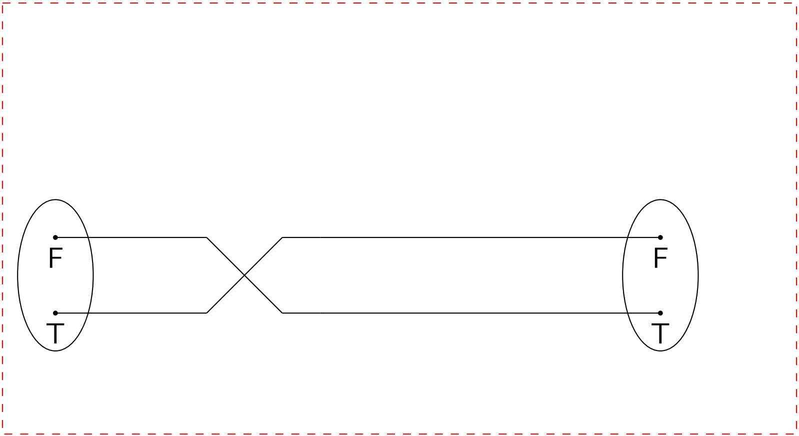
Visually

By unit-unit:



Visually

By id-unit-right:



Questions

- We don't want an ad hoc notation with ad hoc rewriting rules
- Notions of soundness; completeness; canonicity in some sense; what can we say?

1-paths vs. 2-paths

1-paths are between isomorphic types, e.g., $A * B$ and $B * A$. List them all.

1-paths vs. 2-paths

2-paths are between 1-paths, e.g.,
%endcode

1-paths vs. 2-paths