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.

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.

A Calculus of Permutations

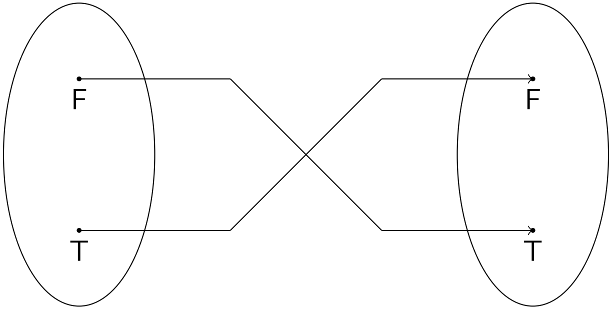
Syntactic theories only rely on transforming source programs to other programs, much like algebraic calculation. Since only the *syntax* of the programming language is relevant to the syntactic theory, the theory is accessible to non-specialists like programmers or students.
In more detail, it is a general 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.*

A Calculus of Permutations

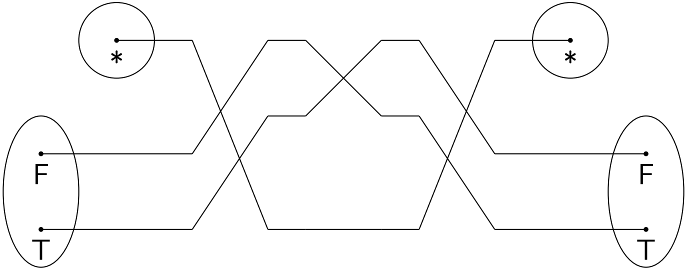
The remedy proposed by Schmidt and others is the development of a formal semantics with a strong calculational flavor. As expressed by Andrew Kennedy in a recent posting to the newsgroup `comp.lang.ml`, the most important property of such a semantics is that it "should be comprehensible to an educated programmer, in the same way that a formal grammar can be understood by a programmer with a degree in computer science." In addition, this popular semantics should be amenable to machine manipulation for proving program properties and for building software engineering tools such as debuggers and static analyzers, and should be reasonably close to implementations in order to explain some of the intensional behavior of languages like space allocation and sharing of data structures.

Example Circuit: Simple Negation



$n_1 : \text{BOOL} \longleftrightarrow \text{BOOL}$
$n_1 = \text{swap}_+$

Example Circuit: Not So Simple Negation



n_2 : BOOL $\longleftrightarrow$ BOOL

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

Reasoning about Example Circuits

Algebraic manipulation of one circuit to the other:

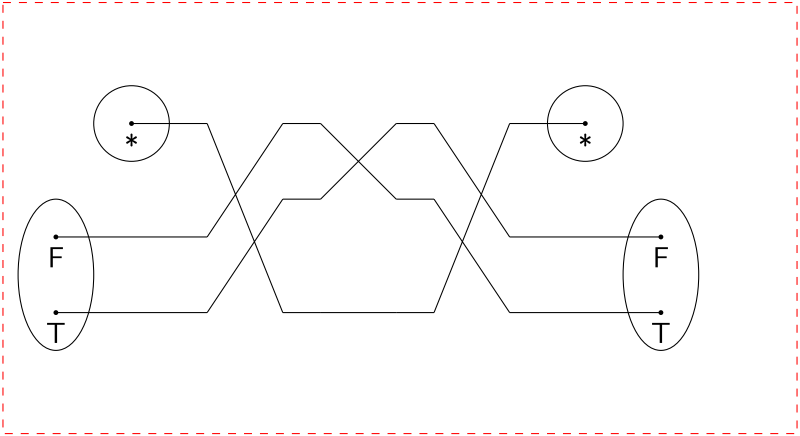
$$
\begin{aligned}
&\mathsf{negEx} : \mathsf{n}_2 \Leftrightarrow \mathsf{n}_1\\
&\mathsf{negEx} = \mathsf{uniti}\star \odot (\mathsf{swap}\star \odot ((\mathsf{swap}_+ \otimes \mathsf{id}{\longleftrightarrow}) \odot (\mathsf{swap}\star \odot \mathsf{unite}\star)))\\
&\qquad \Leftrightarrow\langle\ \mathsf{resp}{\odot}{\Leftrightarrow}\ \mathsf{id}{\Leftrightarrow}\ \mathsf{assoc}{\odot}\mathsf{l}\ \rangle\\
&\quad \mathsf{uniti}\star \odot ((\mathsf{swap}\star \odot (\mathsf{swap}_+ \otimes \mathsf{id}{\longleftrightarrow})) \odot (\mathsf{swap}\star \odot \mathsf{unite}\star))\\
&\qquad \Leftrightarrow\langle\ \mathsf{resp}{\odot}{\Leftrightarrow}\ \mathsf{id}{\Leftrightarrow}\ (\mathsf{resp}{\odot}{\Leftrightarrow}\ \mathsf{swapl}{\star}{\Leftrightarrow}\ \mathsf{id}{\Leftrightarrow})\ \rangle\\
&\quad \mathsf{uniti}\star \odot (((\mathsf{id}{\longleftrightarrow} \otimes \mathsf{swap}_+) \odot \mathsf{swap}\star) \odot (\mathsf{swap}\star \odot \mathsf{unite}\star))\\
&\qquad \Leftrightarrow\langle\ \mathsf{resp}{\odot}{\Leftrightarrow}\ \mathsf{id}{\Leftrightarrow}\ \mathsf{assoc}{\odot}\mathsf{r}\ \rangle\\
&\quad \mathsf{uniti}\star \odot ((\mathsf{id}{\longleftrightarrow} \otimes \mathsf{swap}_+) \odot (\mathsf{swap}\star \odot (\mathsf{swap}\star \odot \mathsf{unite}\star)))\\
&\qquad \Leftrightarrow\langle\ \mathsf{resp}{\odot}{\Leftrightarrow}\ \mathsf{id}{\Leftrightarrow}\ (\mathsf{resp}{\odot}{\Leftrightarrow}\ \mathsf{id}{\Leftrightarrow}\ \mathsf{assoc}{\odot}\mathsf{l})\ \rangle\\
&\quad \mathsf{uniti}\star \odot ((\mathsf{id}{\longleftrightarrow} \otimes \mathsf{swap}_+) \odot ((\mathsf{swap}\star \odot \mathsf{swap}\star) \odot \mathsf{unite}\star))\\
&\qquad \Leftrightarrow\langle\ \mathsf{resp}{\odot}{\Leftrightarrow}\ \mathsf{id}{\Leftrightarrow}\ (\mathsf{resp}{\odot}{\Leftrightarrow}\ \mathsf{id}{\Leftrightarrow}\ (\mathsf{resp}{\odot}{\Leftrightarrow}\ \mathsf{linv}{\odot}\mathsf{l}\ \mathsf{id}{\Leftrightarrow}))\ \rangle
\end{aligned}
$$

Reasoning about Example Circuits

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

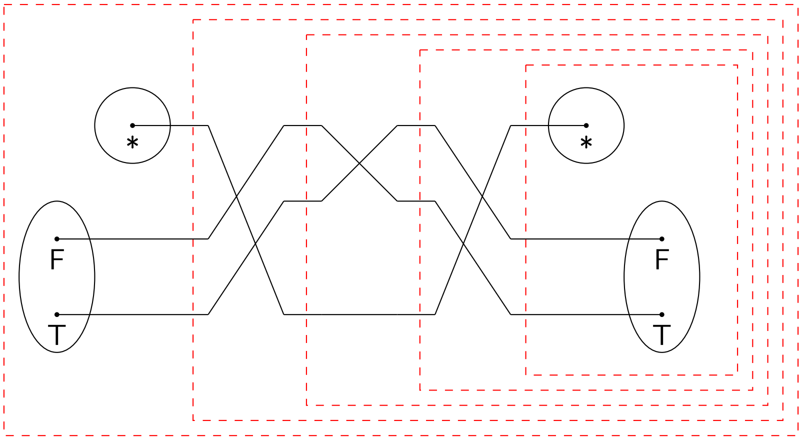
Visually

Original circuit:



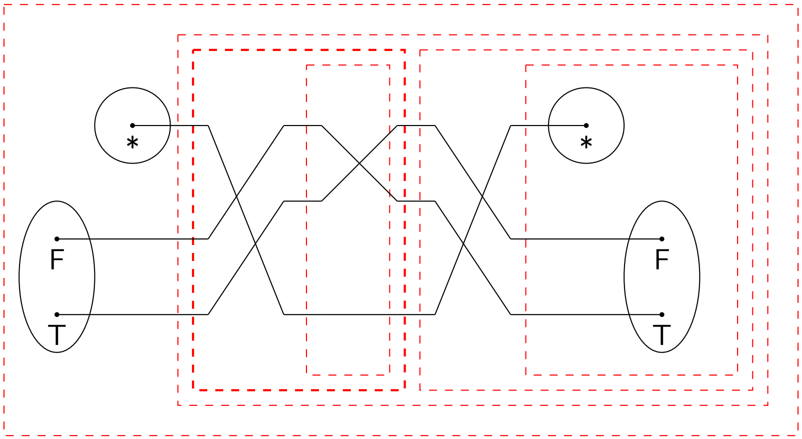
Visually

Making grouping explicit:



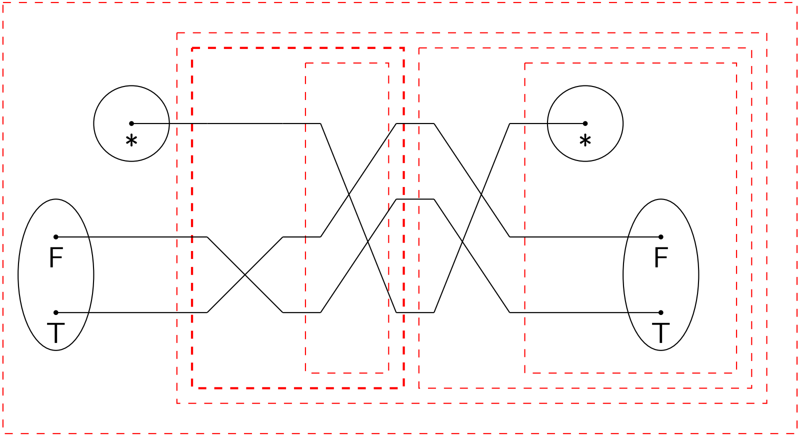
Visually

By associativity:



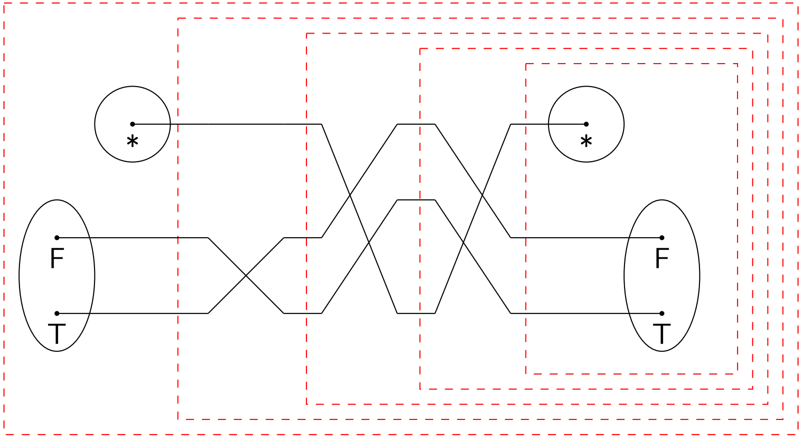
Visually

By pre-post-swap:



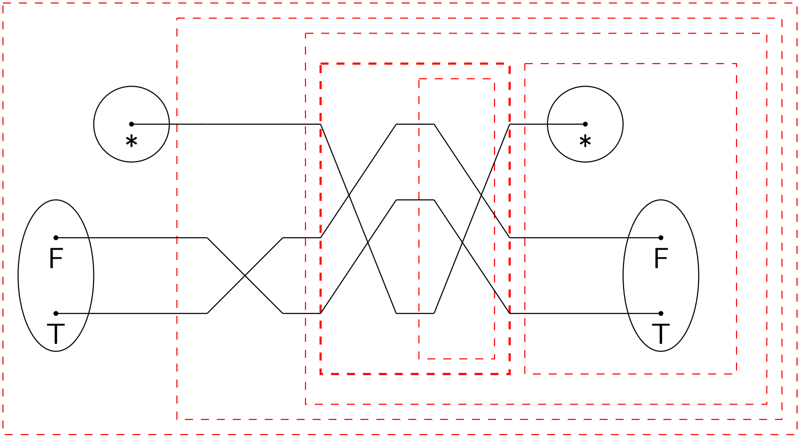
Visually

By associativity:



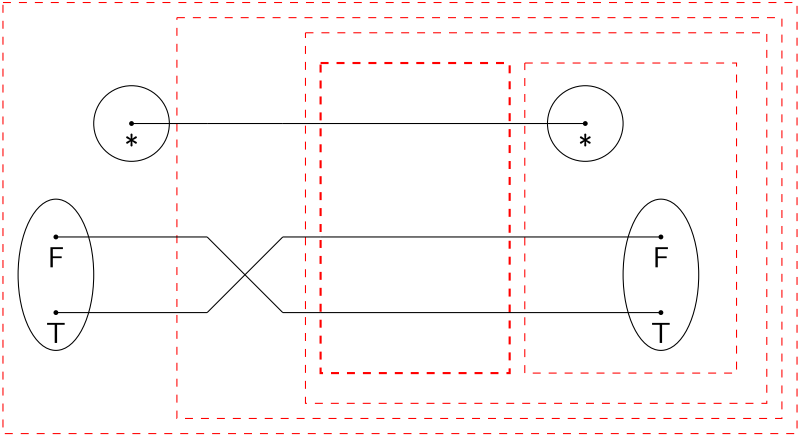
Visually

By associativity:



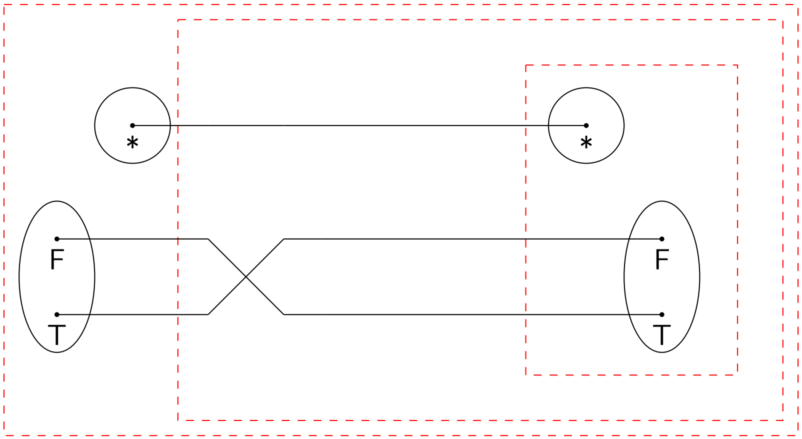
Visually

By swap-swap:



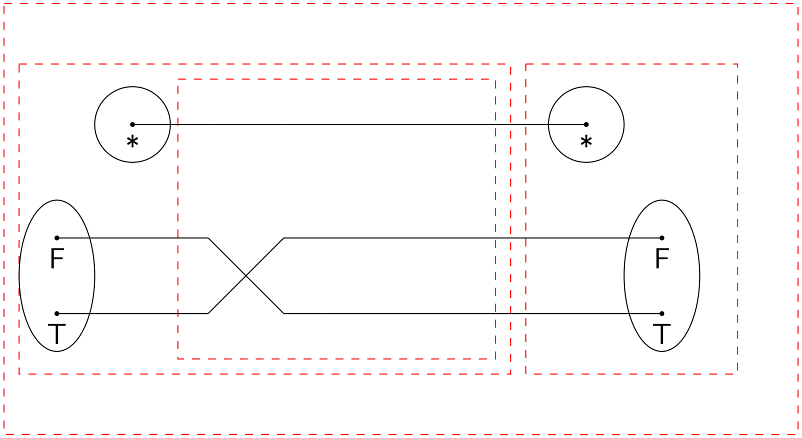
Visually

By id-compose-left:



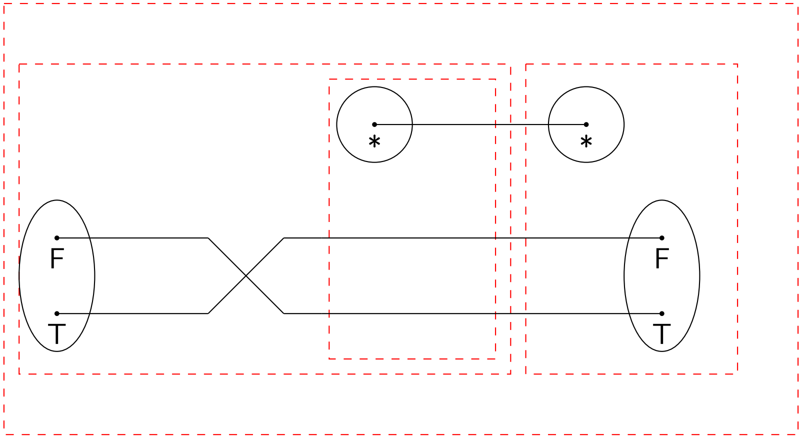
Visually

By associativity:



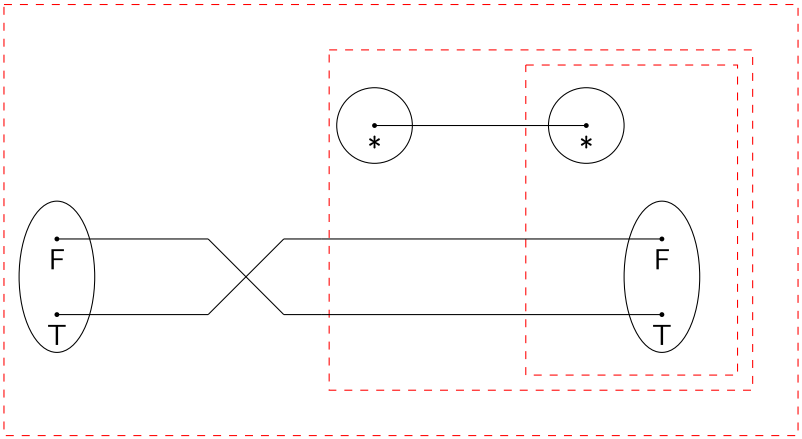
Visually

By swap-unit:



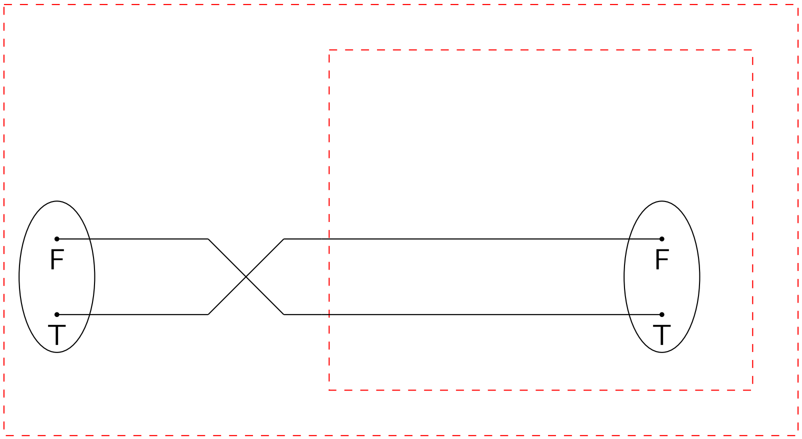
Visually

By associativity:



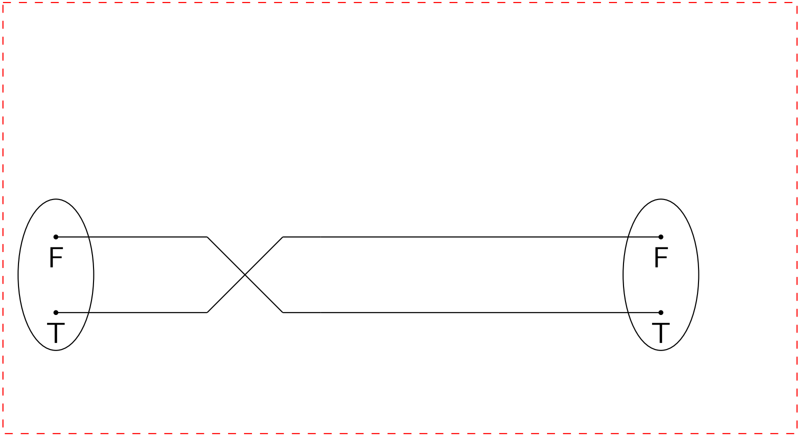
Visually

By unit-unit:



Visually

By id-unit-right:



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

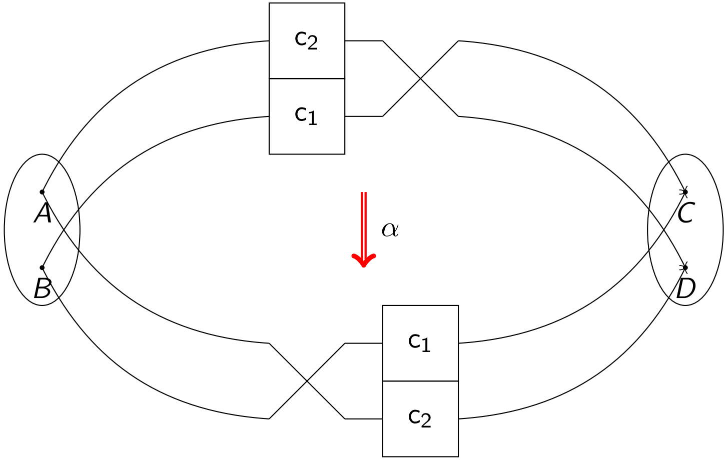
1-paths are between isomorphic types, e.g., $A * B$ and $B * A$. List them all.

1-paths vs. 2-paths

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

$$
\begin{array}{l}
\quad \text{postulate} \\
\qquad c_1 : \{B\ C : U\} \to B \longleftrightarrow C \\
\qquad c_2 : \{A\ D : U\} \to A \longleftrightarrow D \\
\\
\quad p_1\ p_2 : \{A\ B\ C\ D : U\} \to \mathsf{PLUS}\ A\ B \longleftrightarrow \mathsf{PLUS}\ C\ D \\
\quad p_1 = \mathsf{swap}_+ \odot (c_1 \oplus c_2) \\
\quad p_2 = (c_2 \oplus c_1) \odot \mathsf{swap}_+
\end{array}
$$

1-paths vs. 2-paths



?

?

?

?

?

?

?

?

?

?

?

?

?

?

?

?

?

?

?

?

?