A Computational Reconstruction of Homotopy Type Theory for Finite Types

Abstract

Homotopy type theory (HoTT) relates some aspects of topology, algebra, geometry, physics, logic, and type theory, in a unique novel way that promises a new and foundational perspective on mathematics and computation. The heart of HoTT is the *univalence axiom*, which informally states that isomorphic structures can be identified. One of the major open problems in HoTT is a computational interpretation of this axiom. We propose that, at least for the special case of finite types, reversible computation via type isomorphisms *is* the computational interpretation of univalence.

1. Introduction

Conventional HoTT/Agda approach We start with a computational framework: data (pairs, etc.) and functions between them. There are computational rules (beta, etc.) that explain what a function does on a given datum.

We then have a notion of identity which we view as a process that equates two things and model as a new kind of data. Initially we only have identities between beta-equivalent things.

Then we postulate a process that identifies any two functions that are extensionally equivalent. We also postulate another process that identifies any two sets that are isomorphic. This is done by adding new kinds of data for these kinds of identities.

Our approach Our approach is to start with a computational framework that has finite data and permutations as the operations between them. The computational rules apply permutations.

HoTT says id types are an inductively defined type family with refl as constructor. We say it is a family defined with pi combinators as constructors. Replace path induction with refl as base case with our induction.

Generalization How would that generalize to first-class functions? Using negative and fractionals? Groupoids?

In a computational world in which the laws of physics are embraced and resources are carefully maintained (e.g., quantum computing [Abramsky and Coecke 2004; Nielsen and Chuang 2000]), programs must be reversible. Although this is apparently a limiting idea, it turns out that conventional computation can be viewed as a special case of such resource-preserving reversible programs. This thesis has been explored for many years from different perspectives [Bennett 2003, 2010, 1973; Fredkin and Toffoli 1982; Landauer 1961, 1996; Toffoli 1980]. We build on the work of James and Sabry [2012] which expresses this thesis in a type theoretic computational framework, expressing computation via type isomorphisms.

2. Condensed Background on HoTT

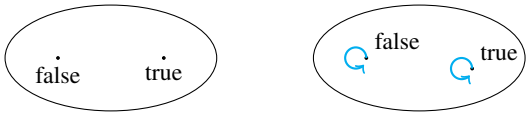
Informally, and as a first approximation, one may think of HoTT as mathematics, type theory, or computation but with all equalities replaced by isomorphisms, i.e., with equalities given computational content. We explain some of the basic ideas below.

One starts with Martin-Löf type theory, interprets the types as topological spaces or weak ∞-groupoids, and interprets identities between elements of a type as *paths*. In more detail, one interprets the witnesses of the identity $x \equiv y$ as paths from x to y. If x and y are themselves paths, then witnesses of the identity $x \equiv y$ become paths between paths, or homotopies in the topological language. In Agda notation, we can formally express this as follows:

```
data _≡_ {ℓ} {A : Set ℓ} : (a b : A) → Set ℓ where
  refl : (a : A) → (a ≡ a)

i0 : 3 ≡ 3
i0 = refl 3

i1 : (1 + 2) ≡ (3 * 1)
i1 = refl 3

i2 : ℕ ≡ ℕ
i2 = refl ℕ
```

It is important to note that the notion of propositional equality $\equiv$ relates any two terms that are *definitionally equal* as shown in example i1 above. In general, there may be *many* proofs (i.e., paths) showing that two particular values are identical and that proofs are not necessarily identical. This gives rise to a structure of great combinatorial complexity. To be explicit, we will use $\equiv_U$ to refer to the space in which the path lives.

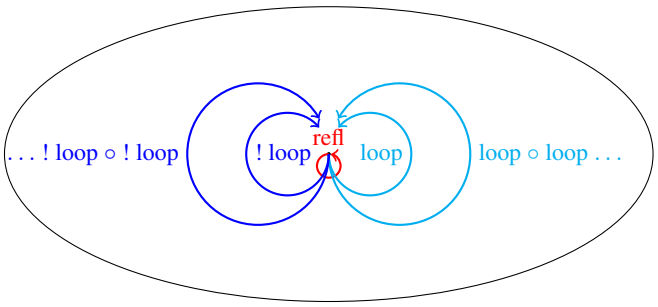
We are used to thinking of types as sets of values. So we typically view the type Bool as the figure on the left but in HoTT we should instead think about it as the figure on the right:



In this particular case, it makes no difference, but in general we may have a much more complicated path structure.

We cannot generate non-trivial groupoids starting from the usual type constructions. We need *higher-order inductive types* for that purpose. The classical example is the *circle* that is a space consisting of a point base and a path loop from base to itself. As stated, this does not amount to much. However, because paths carry

additional structure (explained below), that space has the following non-trivial structure:



The additional structure of types is formalized as follows. Let x, y, and z be elements of some U:

- For every path $p : x \equiv_U y$, there exists a path $!p : y \equiv_U x$;
- For every $p : x \equiv_U y$ and $q : y \equiv_U z$, there exists a path $p \circ q : x \equiv_U z$;
- Subject to the following conditions:

$$\begin{array}{rll} p \circ \mathit{refl}\ y & \equiv_{x \equiv_U y} & p \\ p & \equiv_{x \equiv_U y} & \mathit{refl}\ x \circ p \\ !p \circ p & \equiv_{y \equiv_U y} & \mathit{refl}\ y \\ p \circ\ !p & \equiv_{x \equiv_U x} & \mathit{refl}\ x \\ !\,(!p) & \equiv_{x \equiv_U y} & p \\ p \circ (q \circ r) & \equiv_{x \equiv_U z} & (p \circ q) \circ r \end{array}$$

- With similar conditions one level up and so on and so forth.

3. Computing with Type Isomorphisms

The main syntactic vehicle for the developments in this paper is a simple language called Π whose only computations are isomorphisms between finite types.

3.1 Syntax and Examples

The set of types τ includes the empty type 0, the unit type 1, and conventional sum and product types. The values classified by these types are the conventional ones: () of type 1, $\mathsf{inl}\ v$ and $\mathsf{inr}\ v$ for injections into sum types, and (v_1, v_2) for product types:

$$\begin{array}{lrcl} (\textit{Types}) & \tau & ::= & 0 \mid 1 \mid \tau_1 + \tau_2 \mid \tau_1 * \tau_2 \\ (\textit{Values}) & v & ::= & () \mid \mathsf{inl}\ v \mid \mathsf{inr}\ v \mid (v_1, v_2) \\ (\textit{Combinator types}) & & & \tau_1 \leftrightarrow \tau_2 \\ (\textit{Combinators}) & c & ::= & [\textit{see Table 1}] \end{array}$$

The interesting syntactic category of Π is that of *combinators* which are witnesses for type isomorphisms $\tau_1 \leftrightarrow \tau_2$. They consist of base combinators (on the left side of Table 1) and compositions (on the right side of the same table). Each line of the table on the left introduces a pair of dual constants[1] that witness the type isomorphism in the middle. This set of isomorphisms is known to be complete [Fiore 2004; Fiore et al. 2006] and the language is universal for hardware combinational circuits [James and Sabry 2012].[2]

[1] where $swap_+$ and $swap_*$ are self-dual.

[2] If recursive types and a trace operator are added, the language becomes Turing complete [Bowman et al. 2011; James and Sabry 2012]. We will not be concerned with this extension in the main body of this paper but it will be briefly discussed in the conclusion.

3.2 Semantics

From the perspective of category theory, the language Π models what is called a *symmetric bimonoidal category* or a *commutative rig category*. These are categories with two binary operations $\oplus$ and $\otimes$ satisfying the axioms of a rig (i.e., a ring without negative elements also known as a semiring) up to coherent isomorphisms. And indeed the types of the Π-combinators are precisely the semiring axioms. A formal way of saying this is that Π is the *categorification* [Baez and Dolan 1998] of the natural numbers. A simple (slightly degenerate) example of such categories is the category of finite sets and permutations in which we interpret every Π-type as a finite set, the values as elements in these finite sets, and the combinators as permutations. Another common example of such categories is the category of finite dimensional vector spaces and linear maps over any field. Note that in this interpretation, the Π-type 0 maps to the 0-dimensional vector space which is *not* empty. Its unique element, the zero vector — which is present in every vector space — acts like a "bottom" everywhere-undefined element and hence the type behaves like the unit of addition and the annihilator of multiplication as desired.

Operationally, the semantics consists of a pair of mutually recursive evaluators that take a combinator and a value and propagate the value in the "forward" $\triangleright$ direction or in the "backwards" $\triangleleft$ direction. We show the complete forward evaluator; the backwards evaluator differs in trivial ways:

$$\begin{array}{rcl} \mathit{identl}_+ \triangleright (\mathsf{inr}\ v) & = & v \\ \mathit{identr}_+ \triangleright v & = & \mathsf{inr}\ v \\ \mathit{swap}_+ \triangleright (\mathsf{inl}\ v) & = & \mathsf{inr}\ v \\ \mathit{swap}_+ \triangleright (\mathsf{inr}\ v) & = & \mathsf{inl}\ v \\ \mathit{assocl}_+ \triangleright (\mathsf{inl}\ v) & = & \mathsf{inl}\ (\mathsf{inl}\ v) \\ \mathit{assocl}_+ \triangleright (\mathsf{inr}\ (\mathsf{inl}\ v)) & = & \mathsf{inl}\ (\mathsf{inr}\ v) \\ \mathit{assocl}_+ \triangleright (\mathsf{inr}\ (\mathsf{inr}\ v)) & = & \mathsf{inr}\ v \\ \mathit{assocr}_+ \triangleright (\mathsf{inl}\ (\mathsf{inl}\ v)) & = & \mathsf{inl}\ v \\ \mathit{assocr}_+ \triangleright (\mathsf{inl}\ (\mathsf{inr}\ v)) & = & \mathsf{inr}\ (\mathsf{inl}\ v) \\ \mathit{assocr}_+ \triangleright (\mathsf{inr}\ v) & = & \mathsf{inr}\ (\mathsf{inr}\ v) \\ \mathit{identl}_* \triangleright ((), v) & = & v \\ \mathit{identr}_* \triangleright v & = & ((), v) \\ \mathit{swap}_* \triangleright (v_1, v_2) & = & (v_2, v_1) \\ \mathit{assocl}_* \triangleright (v_1, (v_2, v_3)) & = & ((v_1, v_2), v_3) \\ \mathit{assocr}_* \triangleright ((v_1, v_2), v_3) & = & (v_1, (v_2, v_3)) \\ \mathit{dist} \triangleright (\mathsf{inl}\ v_1, v_3) & = & \mathsf{inl}\ (v_1, v_3) \\ \mathit{dist} \triangleright (\mathsf{inr}\ v_2, v_3) & = & \mathsf{inr}\ (v_2, v_3) \\ \mathit{factor} \triangleright (\mathsf{inl}\ (v_1, v_3)) & = & (\mathsf{inl}\ v_1, v_3) \\ \mathit{factor} \triangleright (\mathsf{inr}\ (v_2, v_3)) & = & (\mathsf{inr}\ v_2, v_3) \\ \mathit{id} \triangleright v & = & v \\ (\mathit{sym}\ c) \triangleright v & = & c \triangleleft v \\ (c_1 \fatsemi c_2) \triangleright v & = & c_2 \triangleright (c_1 \triangleright v) \\ (c_1 \oplus c_2) \triangleright (\mathsf{inl}\ v) & = & \mathsf{inl}\ (c_1 \triangleright v) \\ (c_1 \oplus c_2) \triangleright (\mathsf{inr}\ v) & = & \mathsf{inr}\ (c_2 \triangleright v) \\ (c_1 \otimes c_2) \triangleright (v_1, v_2) & = & (c_1 \triangleright v_1, c_2 \triangleright v_2) \end{array}$$

4. The Space of Types

Instead of modeling the semantics of Π using *permutations*, which are set-theoretic functions after all, we use *paths* from the HoTT framework. More precisely, we model the universe of Π types as a space whose points are the individual Π-types and we will consider that there is path between two points τ_1 and τ_2 if there is a Π combinator $c : \tau_1 \leftrightarrow \tau_2$. If we focus on 1-paths, this is perfect as we explain next.

Note. But first, we note that this is a significant deviation from the HoTT framework which fundamentally includes functions, which are specialized to equivalences, which are then postulated to be paths by the univalence axiom. This axiom has no satisfactory computational interpretation, however. Instead we completely bypass

$$
\begin{array}{rrcll}
identl_+ : & 0+\tau & \leftrightarrow & \tau & : identr_+ \\
swap_+ : & \tau_1+\tau_2 & \leftrightarrow & \tau_2+\tau_1 & : swap_+ \\
assocl_+ : & \tau_1+(\tau_2+\tau_3) & \leftrightarrow & (\tau_1+\tau_2)+\tau_3 & : assocr_+ \\
identl_* : & 1*\tau & \leftrightarrow & \tau & : identr_* \\
swap_* : & \tau_1*\tau_2 & \leftrightarrow & \tau_2*\tau_1 & : swap_* \\
assocl_* : & \tau_1*(\tau_2*\tau_3) & \leftrightarrow & (\tau_1*\tau_2)*\tau_3 & : assocr_* \\
dist_0 : & 0*\tau & \leftrightarrow & 0 & : factor_0 \\
dist : & (\tau_1+\tau_2)*\tau_3 & \leftrightarrow & (\tau_1*\tau_3)+(\tau_2*\tau_3) & : factor
\end{array}
$$

$$
\frac{}{\vdash id : \tau \leftrightarrow \tau} \qquad \frac{\vdash c : \tau_1 \leftrightarrow \tau_2}{\vdash sym\ c : \tau_2 \leftrightarrow \tau_1}
$$

$$
\frac{\vdash c_1 : \tau_1 \leftrightarrow \tau_2 \quad \vdash c_2 : \tau_2 \leftrightarrow \tau_3}{\vdash c_1 \fatsemi c_2 : \tau_1 \leftrightarrow \tau_3}
$$

$$
\frac{\vdash c_1 : \tau_1 \leftrightarrow \tau_2 \quad \vdash c_2 : \tau_3 \leftrightarrow \tau_4}{\vdash c_1 \oplus c_2 : \tau_1+\tau_3 \leftrightarrow \tau_2+\tau_4}
$$

$$
\frac{\vdash c_1 : \tau_1 \leftrightarrow \tau_2 \quad \vdash c_2 : \tau_3 \leftrightarrow \tau_4}{\vdash c_1 \otimes c_2 : \tau_1*\tau_3 \leftrightarrow \tau_2*\tau_4}
$$

Table 1. Π-combinators [James and Sabry 2012]

the idea of extensional functions and use paths directly. Another way to understanding what is going on is the following. In the conventional HoTT framework:

- We start with two different notions: paths and functions;
- We use extensional non-constructive methods to identify a particular class of functions that form isomorphisms;
- We postulate that this particular class of functions can be identified with paths.

In our case,

- We start with a constructive characterization of *reversible functions* or *isomorphisms* built using inductively defined combinators;
- We blur the distinction between such combinators and paths from the beginning. We view computation as nothing more than *following paths*! As explained earlier, although this appears limiting, it is universal and regular computation can be viewed as a special case of that.

Construction. We have a universe U viewed as a groupoid whose points are the types Π-types τ. The Π-combinators of Table 1 are viewed as syntax for the paths in the space U. We need to show that the groupoid path structure is faithfully represented. The combinator id introduces all the refl $\tau : \tau \equiv \tau$ paths in U. The adjoint $sym\ c$ introduces an inverse path $!p$ for each path p introduced by c. The composition operator $\fatsemi$ introduce a path $p \circ q$ for every pair of paths whose endpoints match. In addition, we get paths like $swap_+$ between $\tau_1+\tau_2$ and $\tau_2+\tau_1$. The existence of such paths in the conventional HoTT developed is *postulated* by the univalence axiom. The $\otimes$-composition gives a path $(p,q) : (\tau_1*\tau_2) \equiv (\tau_3*\tau_4)$ whenever we have paths $p : \tau_1 \equiv \tau_3$ and $q : \tau_2 \equiv \tau_4$. A similar situation for the $\oplus$-composition. The structure of these paths must be discovered and these paths must be *proved* to exist using path induction in the conventional HoTT development. So far, this appears too good to be true, and it is. The problem is that paths in HoTT are subject to rules discussed at the end of Sec. 2. For example, it must be the case that if $p : \tau_1 \equiv_U \tau_2$ that $(p \circ \text{refl}\ \tau_2) \equiv_{\tau_1 \equiv_U \tau_2} p$. This path lives in a higher universe: nothing in our Π-combinators would justify adding such a path as all our combinators map types to types. No combinator works one level up at the space of combinators and there is no such space in the first place. Clearly we are stuck unless we manage to express a notion of higher-order functions in Π. This would allow us to internalize the type $\tau_1 \leftrightarrow \tau_2$ as a Π-type which is then manipulated by the same combinators one level higher and so on.

To make the correspondence between Π and the HoTT concepts more apparent we will, in the remainder of the paper, use refl instead of id and ! instead of sym when referring to Π combinators when viewed as paths. Similarly we will use $\rightarrow$ instead of the Π-notation $\leftrightarrow$ or the HoTT notation $\equiv$ to refer to paths.

5. Agda Model

```
------------------------------------------------------
- Level 0:
- Types at this level are just plain sets with no i
- The path structure is defined at levels 1 and bey

data U : Set where
  ZERO  : U
  ONE   : U
  PLUS  : U → U → U
  TIMES : U → U → U

10214_10215 : U → Set
10214 ZERO 10215        = ⊥
10214 ONE 10215         = ⊤
10214 PLUS t₁ t₂ 10215  = 10214 t₁ 10215 ⊎ 10214 t₂ 10215
10214 TIMES t₁ t₂ 10215 = 10214 t₁ 10215 × 10214 t₂ 10215

- Programs
- We use pointed types; programs map a pointed type
- In other words, each program takes one particular
- want to work on another value, we generally use a

record U• : Set where
  constructor •[_,_]
  field
    |_| : U
    • : 10214 |_| 10215

open U•

Space : (t• : U•) → Set
Space •[ t , v ] = 10214 t 10215

point : (t• : U•) → Space t•
point •[ t , v ] = v

- examples of plain types, values, and pointed type

ONE• : U•
ONE• = •[ ONE , tt ]

BOOL : U
BOOL = PLUS ONE ONE

BOOL² : U
BOOL² = TIMES BOOL BOOL
```

```
TRUE : 10214 BOOL 10215
TRUE = inj₁ tt

FALSE : 10214 BOOL 10215
FALSE = inj₂ tt

BOOL•F : U•
BOOL•F = •[ BOOL , FALSE ]

BOOL•T : U•
BOOL•T = •[ BOOL , TRUE ]

- The actual programs are the commutative semiring isomorphisms between
- pointed types.

data _10231_ : U• → U• → Set where
  unite₊    : ∀ {t v} → •[ PLUS ZERO t , inj₂ v ] 10231 •[ t , v ]
  uniti₊    : ∀ {t v} → •[ t , v ] 10231 •[ PLUS ZERO t , inj₂ v ]
  swap1₊    : ∀ {t₁ t₂ v₁} → •[ PLUS t₁ t₂ , inj₁ v₁ ] 10231 •[ PLUS t₂ t₁ , inj₂ v₁ ]
  swap2₊    : ∀ {t₁ t₂ v₂} → •[ PLUS t₁ t₂ , inj₂ v₂ ] 10231 •[ PLUS t₂ t₁ , inj₁ v₂ ]
  assocl1₊  : ∀ {t₁ t₂ t₃ v₁} →
              •[ PLUS t₁ (PLUS t₂ t₃) , inj₁ v₁ ] 10231
              •[ PLUS (PLUS t₁ t₂) t₃ , inj₁ (inj₁ v₁) ]
  assocl2₊  : ∀ {t₁ t₂ t₃ v₂} →
              •[ PLUS t₁ (PLUS t₂ t₃) , inj₂ (inj₁ v₂) ] 10231
              •[ PLUS (PLUS t₁ t₂) t₃ , inj₁ (inj₂ v₂) ]
  assocl3₊  : ∀ {t₁ t₂ t₃ v₃} →
              •[ PLUS t₁ (PLUS t₂ t₃) , inj₂ (inj₂ v₃) ] 10231
              •[ PLUS (PLUS t₁ t₂) t₃ , inj₂ v₃ ]
  assocr1₊  : ∀ {t₁ t₂ t₃ v₁} →
              •[ PLUS (PLUS t₁ t₂) t₃ , inj₁ (inj₁ v₁) ] 10231
              •[ PLUS t₁ (PLUS t₂ t₃) , inj₁ v₁ ]
  assocr2₊  : ∀ {t₁ t₂ t₃ v₂} →
              •[ PLUS (PLUS t₁ t₂) t₃ , inj₁ (inj₂ v₂) ] 10231
              •[ PLUS t₁ (PLUS t₂ t₃) , inj₂ (inj₁ v₂) ]
  assocr3₊  : ∀ {t₁ t₂ t₃ v₃} →
              •[ PLUS (PLUS t₁ t₂) t₃ , inj₂ v₃ ] 10231
              •[ PLUS t₁ (PLUS t₂ t₃) , inj₂ (inj₂ v₃) ]
  unite⋆    : ∀ {t v} → •[ TIMES ONE t , (tt , v) ] 10231 •[ t , v ]
  uniti⋆    : ∀ {t v} → •[ t , v ] 10231 •[ TIMES ONE t , (tt , v) ]
  swap⋆     : ∀ {t₁ t₂ v₁ v₂} →
              •[ TIMES t₁ t₂ , (v₁ , v₂) ] 10231 •[ TIMES t₂ t₁ , (v₂ , v₁) ]
  assocl⋆   : ∀ {t₁ t₂ t₃ v₁ v₂ v₃} →
              •[ TIMES t₁ (TIMES t₂ t₃) , (v₁ , (v₂ , v₃)) ] 10231
              •[ TIMES (TIMES t₁ t₂) t₃ , ((v₁ , v₂) , v₃) ]
  assocr⋆   : ∀ {t₁ t₂ t₃ v₁ v₂ v₃} →
              •[ TIMES (TIMES t₁ t₂) t₃ , ((v₁ , v₂) , v₃) ] 10231
              •[ TIMES t₁ (TIMES t₂ t₃) , (v₁ , (v₂ , v₃)) ]
  distz     : ∀ {t v absurd} →
              •[ TIMES ZERO t , (absurd , v) ] 10231 •[ ZERO , absurd ]
  factorz   : ∀ {t v absurd} →
              •[ ZERO , absurd ] 10231 •[ TIMES ZERO t , (absurd , v) ]
  dist1     : ∀ {t₁ t₂ t₃ v₁ v₃} →
              •[ TIMES (PLUS t₁ t₂) t₃ , (inj₁ v₁ , v₃) ] 10231
              •[ PLUS (TIMES t₁ t₃) (TIMES t₂ t₃) , inj₁ (v₁ , v₃) ]
  dist2     : ∀ {t₁ t₂ t₃ v₂ v₃} →
              •[ TIMES (PLUS t₁ t₂) t₃ , (inj₂ v₂ , v₃) ] 10231
              •[ PLUS (TIMES t₁ t₃) (TIMES t₂ t₃) , inj₂ (v₂ , v₃) ]
  factor1   : ∀ {t₁ t₂ t₃ v₁ v₃} →
              •[ PLUS (TIMES t₁ t₃) (TIMES t₂ t₃) , inj₁ (v₁ , v₃) ] 10231
              •[ TIMES (PLUS t₁ t₂) t₃ , (inj₁ v₁ , v₃) ]
  factor2   : ∀ {t₁ t₂ t₃ v₂ v₃} →
              •[ PLUS (TIMES t₁ t₃) (TIMES t₂ t₃) , inj₂ (v₂ , v₃) ] 10231
              •[ TIMES (PLUS t₁ t₂) t₃ , (inj₂ v₂ , v₃) ]
  id10231   : ∀ {t v} → •[ t , v ] 10231 •[ t , v ]
  sym10231  : ∀ {t₁ t₂ v₁ v₂} → (•[ t₁ , v₁ ] 10231 •[ t₂ , v₂ ]) →
              (•[ t₂ , v₂ ] 10231 •[ t₁ , v₁ ])
  _9678_    : ∀ {t₁ t₂ t₃ v₁ v₂ v₃} → (•[ t₁ , v₁ ] 10231 •[ t₂ , v₂ ]) →
              (•[ t₂ , v₂ ] 10231 •[ t₃ , v₃ ]) →
              (•[ t₁ , v₁ ] 10231 •[ t₃ , v₃ ])
  _⊕1_      : ∀ {t₁ t₂ t₃ t₄ v₁ v₂ v₃ v₄} →
              (•[ t₁ , v₁ ] 10231 •[ t₃ , v₃ ]) → (•[ t₂ , v₂ ] 10231 •[ t₄ , v₄ ]) →
              (•[ PLUS t₁ t₂ , inj₁ v₁ ] 10231 •[ PLUS t₃ t₄ , inj₁ v₃ ])
  _⊕2_      : ∀ {t₁ t₂ t₃ t₄ v₁ v₂ v₃ v₄} →
              (•[ t₁ , v₁ ] 10231 •[ t₃ , v₃ ]) → (•[ t₂ , v₂ ] 10231 •[ t₄ , v₄ ]) →
              (•[ PLUS t₁ t₂ , inj₂ v₂ ] 10231 •[ PLUS t₃ t₄ , inj₂ v₄ ])
  _⊗_       : ∀ {t₁ t₂ t₃ t₄ v₁ v₂ v₃ v₄} →
              (•[ t₁ , v₁ ] 10231 •[ t₃ , v₃ ]) → (•[ t₂ , v₂ ] 10231 •[ t₄ , v₄ ]) →
              (•[ TIMES t₁ t₂ , (v₁ , v₂) ] 10231 •[ TIMES t₃ t₄ , (v₃ , v₄) ])

- Some example programs
NOT•T : •[ BOOL , TRUE ] 10231 •[ BOOL , FALSE ]
NOT•T = swap1₊

NOT•F : •[ BOOL , FALSE ] 10231 •[ BOOL , TRUE ]
NOT•F = swap2₊

CNOT•Fx : {b : 10214 BOOL 10215} →
          •[ BOOL² , (FALSE , b) ] 10231 •[ BOOL² , (FALSE , b) ]
CNOT•Fx = dist2 9678 ((id10231 ⊗ NOT•F) ⊕2 id10231) 9678 factor2

CNOT•TF : •[ BOOL² , (TRUE , FALSE) ] 10231 •[ BOOL² , (TRUE , T
CNOT•TF = dist1 9678
          ((id10231 ⊗ NOT•F) ⊕1 (id10231 {TIMES ONE BOOL} {
          factor1

CNOT•TT : •[ BOOL² , (TRUE , TRUE) ] 10231 •[ BOOL² , (TRUE , FA
CNOT•TT = dist1 9678
          ((id10231 ⊗ NOT•T) ⊕1 (id10231 {TIMES ONE BOOL} {
          factor1

- The evaluation of a program is not done in order t
- value. Both the input and output values are encode
- program; what the evaluation does is follow the pa
- reach the ouput value from the input value. Even t
- same pointed types are, by definition, observation
- may follow different paths. At this point, we simp
- programs are "the same." At the next level, we wil
- irrelevant" equivalence and reason about which pat
- other paths via 2paths etc.

- Even though individual types are sets, the univers
- groupoid. The objects of this groupoid are the poi
- morphisms are the programs; and the equivalence of
- degenerate observational equivalence that equates
- are extensionally equivalent.

_obs≅_ : {t₁ t₂ : U•} → (c₁ c₂ : t₁ 10231 t₂) → Set
c₁ obs≅ c₂ = ⊤

UG : 1Groupoid
UG = record
  { set = U•
```

```
  ; _⇝_ = _10231_
  ; _≈_ = _obs≅_
  ; id = id10231
  ; _∘_ = λ y10231z x10231y → x10231y 9678 y10231z
  ; _⁻¹ = sym10231
  ; lneutr = λ _ → tt
  ; rneutr = λ _ → tt
  ; assoc = λ _ _ _ → tt
  ; equiv = record { refl = tt
      ; sym = λ _ → tt
      ; trans = λ _ _ → tt
      }
  ; linv = λ _ → tt
  ; rinv = λ _ → tt
  ; ∘-resp-≈ = λ _ _ → tt
  }

-------------------------------------------------------------------
- Simplifiy various compositions

simplifySym : {t1 t2 : U•} → (c1 : t1 10231 t2) → (t2 10231 t1)
simplifySym unite+ = uniti+
simplifySym uniti+ = unite+
simplifySym swap1+ = swap2+
simplifySym swap2+ = swap1+
simplifySym assocl1+ = assocr1+
simplifySym assocl2+ = assocr2+
simplifySym assocl3+ = assocr3+
simplifySym assocr1+ = assocl1+
simplifySym assocr2+ = assocl2+
simplifySym assocr3+ = assocl3+
simplifySym unite⋆ = uniti⋆
simplifySym uniti⋆ = unite⋆
simplifySym swap⋆ = swap⋆
simplifySym assocl⋆ = assocr⋆
simplifySym assocr⋆ = assocl⋆
simplifySym distz = factorz
simplifySym factorz = distz
simplifySym dist1 = factor1
simplifySym dist2 = factor2
simplifySym factor1 = dist1
simplifySym factor2 = dist2
simplifySym id10231 = id10231
simplifySym (sym10231 c) = c
simplifySym (c1 9678 c2) = simplifySym c2 9678 simplifySym c1
simplifySym (c1 ⊕1 c2) = simplifySym c1 ⊕1 simplifySym c2
simplifySym (c1 ⊕2 c2) = simplifySym c1 ⊕2 simplifySym c2
simplifySym (c1 ⊗ c2) = simplifySym c1 ⊗ simplifySym c2

simplifyl9678 : {t1 t2 t3 : U•} → (c1 : t1 10231 t2) → (c2 : t2 10231 t3) → (
simplifyl9678 id10231 c = c
simplifyl9678 unite+ uniti+ = id10231
simplifyl9678 uniti+ unite+ = id10231
simplifyl9678 swap1+ swap2+ = id10231
simplifyl9678 swap2+ swap1+ = id10231
simplifyl9678 assocl1+ assocr1+ = id10231
simplifyl9678 assocl2+ assocr2+ = id10231
simplifyl9678 assocl3+ assocr3+ = id10231
simplifyl9678 assocr1+ assocl1+ = id10231
simplifyl9678 assocr2+ assocl2+ = id10231
simplifyl9678 assocr3+ assocl3+ = id10231
simplifyl9678 unite⋆ uniti⋆ = id10231
simplifyl9678 uniti⋆ unite⋆ = id10231
simplifyl9678 swap⋆ swap⋆ = id10231
simplifyl9678 assocl⋆ assocr⋆ = id10231
simplifyl9678 assocr⋆ assocl⋆ = id10231
simplifyl9678 factorz distz = id10231
simplifyl9678 dist1 factor1 = id10231
simplifyl9678 dist2 factor2 = id10231
simplifyl9678 factor1 dist1 = id10231
simplifyl9678 factor2 dist2 = id10231
simplifyl9678 (c1 9678 c2) c3 = c1 9678 (c2 9678 c3)
simplifyl9678 (c1 ⊕1 c2) swap1+ = swap1+ 9678 (c2 ⊕2 c1)
simplifyl9678 (c1 ⊕2 c2) swap2+ = swap2+ 9678 (c2 ⊕1 c1)
simplifyl9678 (_⊗_ {ONE} {ONE} c1 c2) unite⋆ = unite⋆ 9678 c2
simplifyl9678 (c1 ⊗ c2) swap⋆ = swap⋆ 9678 (c2 ⊗ c1)
simplifyl9678 (c1 ⊗ c2) (c3 ⊗ c4) = (c1 9678 c3) ⊗ (c2 9678 c4)
simplifyl9678 c1 c2 = c1 9678 c2

simplifyr9678 : {t1 t2 t3 : U•} → (c1 : t1 10231 t2) → (c2 : t2 10231 t3) → (
simplifyr9678 c id10231 = c
simplifyr9678 unite+ uniti+ = id10231
simplifyr9678 uniti+ unite+ = id10231
simplifyr9678 swap1+ swap2+ = id10231
simplifyr9678 swap2+ swap1+ = id10231
simplifyr9678 assocl1+ assocr1+ = id10231
simplifyr9678 assocl2+ assocr2+ = id10231
simplifyr9678 assocl3+ assocr3+ = id10231
simplifyr9678 assocr1+ assocl1+ = id10231
simplifyr9678 assocr2+ assocl2+ = id10231
simplifyr9678 assocr3+ assocl3+ = id10231
simplifyr9678 unite⋆ uniti⋆ = id10231
simplifyr9678 uniti⋆ unite⋆ = id10231
simplifyr9678 swap⋆ swap⋆ = id10231
simplifyr9678 assocl⋆ assocr⋆ = id10231
simplifyr9678 assocr⋆ assocl⋆ = id10231
simplifyr9678 factorz distz = id10231
simplifyr9678 dist1 factor1 = id10231
simplifyr9678 dist2 factor2 = id10231
simplifyr9678 factor1 dist1 = id10231
simplifyr9678 factor2 dist2 = id10231
simplifyr9678 (c1 9678 c2) c3 = c1 9678 (c2 9678 c3)
simplifyr9678 (c1 ⊕1 c2) swap1+ = swap1+ 9678 (c2 ⊕2 c1)
simplifyr9678 (c1 ⊕2 c2) swap2+ = swap2+ 9678 (c2 ⊕1 c1)
simplifyr9678 (_⊗_ {ONE} {ONE} c1 c2) unite⋆ = unite⋆ 9678 c2
simplifyr9678 (c1 ⊗ c2) swap⋆ = swap⋆ 9678 (c2 ⊗ c1)
simplifyr9678 (c1 ⊗ c2) (c3 ⊗ c4) = (c1 9678 c3) ⊗ (c2 9678 c4)
simplifyr9678 c1 c2 = c1 9678 c2
```

6. Homotopy Type Theory

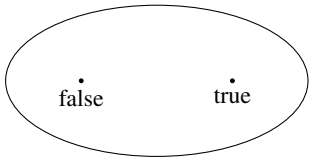
Informally, and as a first approximation, one may think of *homotopy type theory* (HoTT) as mathematics, type theory, or computation but with all equalities replaced by isomorphisms, i.e., with equalities given computational content. A bit more formally, one starts with Martin-Löf type theory, interprets the types as topological spaces or weak ∞-groupoids, and interprets identities between elements of a type as *paths*. In more detail, one interprets the witnesses of the identity $x \equiv y$ as paths from x to y. If x and y are themselves paths, then witnesses of the identity $x \equiv y$ become paths between paths, or homotopies in the topological language.

Formally, Martin-Löf type theory, is based on the principle that every proposition, i.e., every statement that is susceptible to proof, can be viewed as a type. The correspondence is validated by the following properties: if a proposition P is true, the corresponding

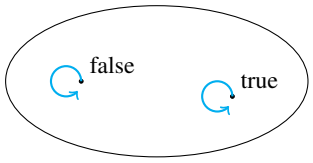
type is inhabited, i.e., it is possible to provide evidence for P using one of the elements of the type P. If, however, the proposition P is false, the corresponding type is empty, i.e., it is impossible to provide evidence for P. The type theory is rich enough to allow propositions denoting conjunction, disjunction, implication, and existential and universal quantifications.

It is clear that the question of whether two elements of a type are equal is a proposition, and hence that this proposition must correspond to a type. It is important to note that the notion of proposition equality $\equiv$ relates any two terms that are *definitionally equal* as shown in example $i1$ above. In general, there may be *many* proofs (i.e., paths) showing that two particular values are identical and that proofs are not necessarily identical. This gives rise to a structure of great combinatorial complexity.

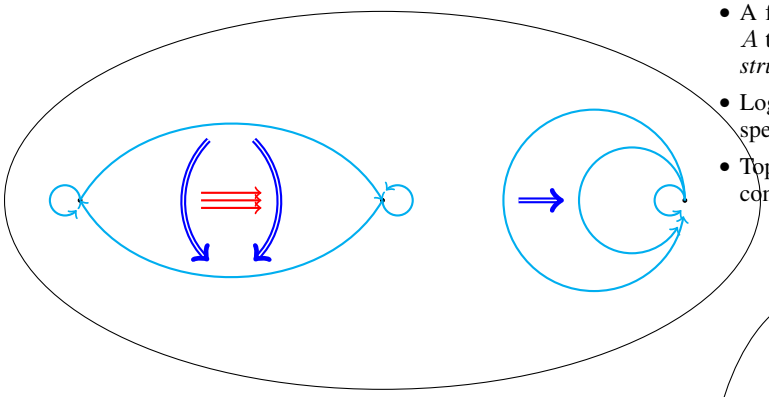
We are used to think of types as sets of values. So we think of the type `Bool` as:



In HoTT, we should instead think about it as:



In this particular case, it makes no difference, but in general we might have something like which shows that types are to be viewed as topological spaces or groupoids:



The additional structure of types is formalized as follows:

- For every path $p : x \equiv y$, there exists a path $!p : y \equiv x$;
- For every $p : x \equiv y$ and $q : y \equiv z$, there exists a path $p \circ q : x \equiv z$;
- Subject to the following conditions:

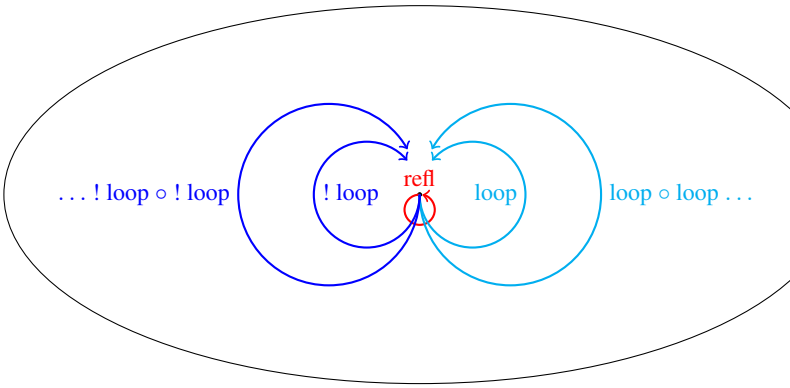
$$\begin{array}{rcl} p \circ \mathit{refl}\ y & \equiv & p \\ p & \equiv & \mathit{refl}\ x \circ p \\ !p \circ p & \equiv & \mathit{refl}\ y \\ p \circ\ !p & \equiv & \mathit{refl}\ x \\ !\,(!p) & \equiv & p \\ p \circ (q \circ r) & \equiv & (p \circ q) \circ r \end{array}$$

- With similar conditions one level up and so on and so forth.

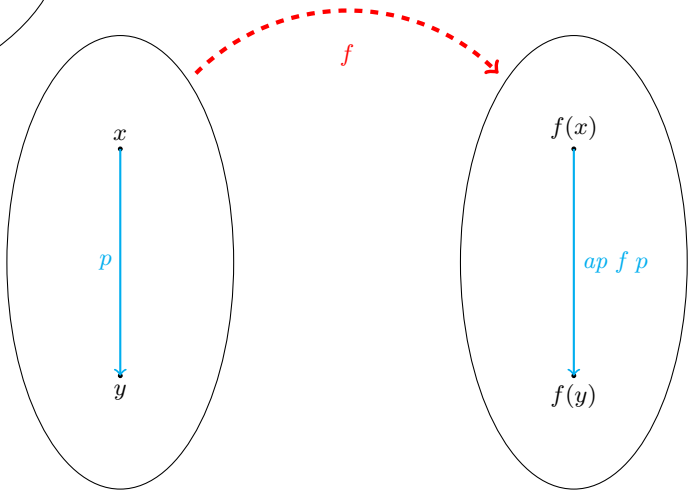
We cannot generate non-trivial groupoids starting from the usual type constructions. We need *higher-order inductive types* for that purpose. Example:

```
- data Circle : Set where
- base : Circle
- loop : base ≡ base

module Circle where
  private data S¹* : Set where base* : S¹*

  S¹ : Set
  S¹ = S¹*

  base : S¹
  base = base*

  postulate loop : base ≡ base
```

Here is the non-trivial structure of this example:



- A function from space A to space B must map the points of A to the points of B as usual but it must also *respect the path structure*;
- Logically, this corresponds to saying that every function respects equality;
- Topologically, this corresponds to saying that every function is continuous.



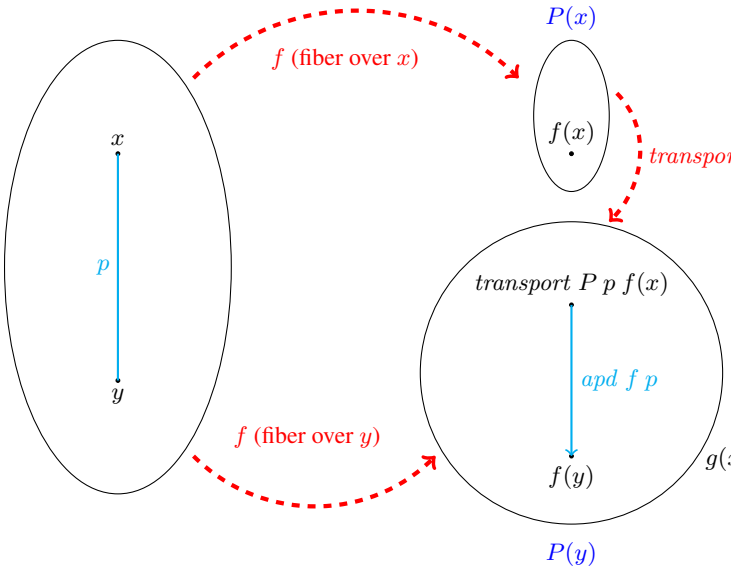
- $ap\ f\ p$ is the action of f on a path p;

- This satisfies the following properties:

$$\begin{aligned} ap\ f\ (p \circ q) &\equiv (ap\ f\ p) \circ (ap\ f\ q) \\ ap\ f\ (!\ p) &\equiv !\ (ap\ f\ p) \\ ap\ g\ (ap\ f\ p) &\equiv ap\ (g \circ f)\ p \\ ap\ id\ p &\equiv p \end{aligned}$$

Type families as fibrations.

- A more complicated version of the previous idea for dependent functions;
- The problem is that for dependent functions, $f(x)$ and $f(y)$ may not be in the same type, i.e., they live in different spaces;
- Idea is to *transport* $f(x)$ to the space of $f(y)$;
- Because everything is "continuous", the path p induces a transport function that does the right thing: the action of f on p becomes a path between *transport* $(f(x))$ and $f(y)$.



- Let $x, y, z : A$, $p : x \equiv y$, $q : y \equiv z$, $f : A \to B$, $g : \Pi_{a \in A} P(a) \to P'(a)$, $P : A \to Set$, $P' : A \to Set$, $Q : B \to Set$, $u : P(x)$, and $w : Q(f(x))$.
- The function *transport* $P\ p$ satisfies the following properties:

$$\begin{aligned} transport\ P\ q\ (transport\ P\ p\ u) &\equiv transport\ P\ (p \circ q)\ u \\ transport\ (Q \circ f)\ p\ w &\equiv transport\ Q\ (ap\ f\ p)\ w \\ transport\ P'\ p\ (g\ x\ u) &\equiv g\ y\ (transport\ P\ p\ u) \end{aligned}$$

- Let $x, y : A$, $p : x \equiv y$, $P : A \to Set$, and $f : \Pi_{a \in A} P(a)$;
- We know we have a path in $P(y)$ between *transport* $P\ p\ (f(x))$ and $f(y)$.
- We do not generally know how the point *transport* $P\ p\ (f(x)) : P(y)$ relates to x;
- We do not generally know how the paths in $P(y)$ are related to the paths in A.
- First "crack" in the theory.

Structure of Paths:

- What do paths in $A \times B$ look like? We can prove that $(a_1, b_1) \equiv (a_2, b_2)$ in $A \times B$ iff $a_1 \equiv a_2$ in A and $b_1 \equiv b_2$ in B.
- What do paths in $A_1 \uplus A_2$ look like? We can prove that $inj_i\ x \equiv inj_j\ y$ in $A_1 \uplus A_2$ iff $i = j$ and $x \equiv y$ in A_i.
- What do paths in $A \to B$ look like? We cannot prove anything. Postulate function extensionality axiom.
- What do paths in Set_ℓ look like? We cannot prove anything. Postulate univalence axiom.

Function Extensionality:

```
- f ∼ g iff ∀ x. f x ≡ g x
_∼_ : ∀ {ℓ ℓ'} → {A : Set ℓ} {P : A → Set ℓ'} →
      (f g : (x : A) → P x) → Set (ℓ ⊔ ℓ')
_∼_ {ℓ} {ℓ'} {A} {P} f g = (x : A) → f x ≡ g x

- f is an equivalence if we have g and h such that
- the compositions with f in both ways are ∼ id
record isequiv {ℓ ℓ'} {A : Set ℓ} {B : Set ℓ'} (f : A → B) :
    Set (ℓ ⊔ ℓ') where
    constructor mkisequiv
    field
      g : B → A
      α : (f ∘ g) ∼ id
      h : B → A
      β : (h ∘ f) ∼ id
- a path between f and g implies f ∼ g
happly : ∀ {ℓ ℓ'} {A : Set ℓ} {B : A → Set ℓ'} {f g : (a : A) → B a} →
         (f ≡ g) → (f ∼ g)
happly {ℓ} {ℓ'} {A} {B} {f} {g} p = {!!}

postulate - that f ∼ g implies a path between f and g
  funextP :  {A : Set} {B : A → Set} {f g : (a : A) → B a} →
             isequiv {A = f ≡ g} {B = f ∼ g} happly

funext :  {A : Set} {B : A → Set} {f g : (a : A) → B a} →
          (f ∼ g) → (f ≡ g)
funext = isequiv.g funextP
```

A path between f and g is a collection of paths from $f(x)$ to $g(x)$. We are no longer executable!

Univalence:

```
- Two spaces are equivalent if we have functions
- f, g, and h that compose to id
_≃_ : ∀ {ℓ ℓ'} (A : Set ℓ) (B : Set ℓ') → Set (ℓ ⊔ ℓ')
A ≃ B = Σ (A → B) isequiv

- A path between spaces implies their equivalence
idtoeqv : {A B : Set} → (A ≡ B) → (A ≃ B)
idtoeqv {A} {B} p = {!!}

postulate - that equivalence of spaces implies a path
  univalence : {A B : Set} → (A ≡ B) ≃ (A ≃ B)
```

Again, we are no longer executable!

Analysis:

- We start with two different notions: paths and functions;
- We use extensional non-constructive methods to identify a particular class of functions that form isomorphisms;
- We postulate that this particular class of functions can be identified with paths.

Insight:

- Start with a constructive characterization of *reversible functions* or *isomorphisms*;
- Blur the distinction between such reversible functions and paths from the beginning.

Note that:

- Reversible functions are computationally universal (Bennett's reversible Turing Machine from 1973!)
- *First-order* reversible functions can be inductively defined in type theory (James and Sabry, POPL 2012).

7. Examples

Let's start with a few simple types built from the empty type, the unit type, sums, and products, and let's study the paths postulated by HoTT.

For every value in a type (point in a space) we have a trivial path from the value to itself:

In addition to all these trivial paths, there are structured paths. In particular, paths in product spaces can be viewed as pair of paths. So in addition to the path above, we also have:

8. Theory

9. Pi

9.1 Base isomorphisms

$$
\begin{array}{rrcll}
identl_+ : & 0 + b & \leftrightarrow & b & : identr_+ \\
swap_+ : & b_1 + b_2 & \leftrightarrow & b_2 + b_1 & : swap_+ \\
assocl_+ : & b_1 + (b_2 + b_3) & \leftrightarrow & (b_1 + b_2) + b_3 & : assocr_+ \\
identl_* : & 1 * b & \leftrightarrow & b & : identr_* \\
swap_* : & b_1 * b_2 & \leftrightarrow & b_2 * b_1 & : swap_* \\
assocl_* : & b_1 * (b_2 * b_3) & \leftrightarrow & (b_1 * b_2) * b_3 & : assocr_* \\
dist_0 : & 0 * b & \leftrightarrow & 0 & : factor_0 \\
dist : & (b_1 + b_2) * b_3 & \leftrightarrow & (b_1 * b_3) + (b_2 * b_3) & : factor
\end{array}
$$

$$\frac{}{\vdash id : b \leftrightarrow b} \qquad \frac{\vdash c : b_1 \leftrightarrow b_2}{\vdash sym\ c : b_2 \leftrightarrow b_1}$$

$$\frac{\vdash c_1 : b_1 \leftrightarrow b_2 \quad c_2 : b_2 \leftrightarrow b_3}{\vdash c_1 \fatsemi c_2 : b_1 \leftrightarrow b_3}$$

$$\frac{\vdash c_1 : b_1 \leftrightarrow b_2 \quad c_2 : b_3 \leftrightarrow b_4}{\vdash c_1 \oplus c_2 : b_1 + b_3 \leftrightarrow b_2 + b_4}$$

$$\frac{\vdash c_1 : b_1 \leftrightarrow b_2 \quad c_2 : b_3 \leftrightarrow b_4}{\vdash c_1 \otimes c_2 : b_1 * b_3 \leftrightarrow b_2 * b_4}$$

These isomorphisms:

- Form an inductive type
- Identify each isomorphism with a collection of paths
- For example:

$$swap_+ : \quad b_1 + b_2 \quad \leftrightarrow \quad b_2 + b_1$$

becomes:

$$
\begin{array}{rrcl}
swap_+^1 : & inj_1 v & \equiv & inj_2 v \\
swap_+^2 : & inj_2 v & \equiv & inj_1 v
\end{array}
$$

References

S. Abramsky and B. Coecke. A categorical semantics of quantum protocols. In *LICS*, 2004.

J. C. Baez and J. Dolan. Categorification. In Higher Category Theory, Contemp. Math. 230, 1998, pp. 1-36., 1998.

C. Bennett. Notes on Landauer's principle, reversible computation, and Maxwell's Demon. *Studies In History and Philosophy of Science Part B: Studies In History and Philosophy of Modern Physics*, 34(3):501–510, 2003.

C. Bennett. Notes on the history of reversible computation. *IBM Journal of Research and Development*, 32(1):16–23, 2010.

C. H. Bennett. Logical reversibility of computation. *IBM J. Res. Dev.*, 17: 525–532, November 1973.

W. J. Bowman, R. P. James, and A. Sabry. Dagger Traced Symmetric Monoidal Categories and Reversible Programming. In *RC*, 2011.

M. Fiore. Isomorphisms of generic recursive polynomial types. In *POPL*, pages 77–88. ACM, 2004.

M. P. Fiore, R. Di Cosmo, and V. Balat. Remarks on isomorphisms in typed calculi with empty and sum types. *Annals of Pure and Applied Logic*, 141(1-2):35–50, 2006.

E. Fredkin and T. Toffoli. Conservative logic. *International Journal of Theoretical Physics*, 21(3):219–253, 1982.

R. P. James and A. Sabry. Information effects. In *POPL*, pages 73–84. ACM, 2012.

R. Landauer. Irreversibility and heat generation in the computing process. *IBM J. Res. Dev.*, 5:183–191, July 1961.

R. Landauer. The physical nature of information. *Physics Letters A*, 1996.

M. Nielsen and I. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, Cambridge, 2000.

T. Toffoli. Reversible computing. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 632–644. Springer-Verlag, 1980.

$$\frac{}{() : 1} \qquad \frac{v_1 : t_1}{\mathsf{inl}\ v_1 : t_1 + t_2} \qquad \frac{v_2 : t_2}{\mathsf{inr}\ v_2 : t_1 + t_2} \qquad \frac{v_1 : t_1 \quad v_2 : t_2}{(v_1, v_2) : t_1 * t_2}$$

$$\frac{}{\mathsf{inr}\ v \xrightarrow{identl_+} v : \mathsf{inr}\ v \equiv_{identl_+} v}$$

$$\frac{}{v \xrightarrow{identr_+} \mathsf{inr}\ v : v \equiv_{identr_+} \mathsf{inr}\ v} \qquad \frac{}{\mathsf{inl}\ v \xrightarrow{swap_+} \mathsf{inr}\ v : \mathsf{inl}\ v \equiv_{swap_+} \mathsf{inr}\ v} \qquad \frac{}{\mathsf{inr}\ v \xrightarrow{swap_+} \mathsf{inl}\ v : \mathsf{inr}\ v \equiv_{swap_+} \mathsf{inl}\ v}$$

$$\frac{}{\mathsf{inl}\ v \xrightarrow{assocl_+} \mathsf{inl}\ (\mathsf{inl}\ v) : \mathsf{inl}\ v \equiv_{assocl_+} \mathsf{inl}\ (\mathsf{inl}\ v)} \qquad \frac{}{\mathsf{inr}\ (\mathsf{inl}\ v) \xrightarrow{assocl_+} \mathsf{inl}\ (\mathsf{inr}\ v) : \mathsf{inr}\ (\mathsf{inl}\ v) \equiv_{assocl_+} \mathsf{inl}\ (\mathsf{inr}\ v)}$$

$$\frac{}{\mathsf{inr}\ (\mathsf{inr}\ v) \xrightarrow{assocl_+} \mathsf{inr}\ v : \mathsf{inr}\ (\mathsf{inr}\ v) \equiv_{assocl_+} \mathsf{inr}\ v} \qquad \frac{}{\mathsf{inl}\ (\mathsf{inl}\ v) \xrightarrow{assocr_+} \mathsf{inl}\ v : \mathsf{inl}\ (\mathsf{inl}\ v) \equiv_{assocr_+} \mathsf{inl}\ v}$$

$$\frac{}{\mathsf{inl}\ (\mathsf{inr}\ v) \xrightarrow{assocr_+} \mathsf{inr}\ (\mathsf{inl}\ v) : \mathsf{inl}\ (\mathsf{inr}\ v) \equiv_{assocr_+} \mathsf{inr}\ (\mathsf{inl}\ v)} \qquad \frac{}{\mathsf{inr}\ v \xrightarrow{assocr_+} \mathsf{inr}\ (\mathsf{inr}\ v) : \mathsf{inr}\ v \equiv_{assocr_+} \mathsf{inr}\ (\mathsf{inr}\ v)}$$

$$\frac{}{((), v) \xrightarrow{identl_*} v : ((), v) \equiv_{identl_*} v} \qquad \frac{}{v \xrightarrow{identr_*} ((), v) : v \equiv_{identr_*} ((), v)} \qquad \frac{}{((v_1, v_2) \xrightarrow{swap_*} (v_2, v_1) : (v_1, v_2) \equiv_{swap_*} (v_2, v_1)}$$

$$\frac{}{(v_1, (v_2, v_3)) \xrightarrow{assocl_*} ((v_1, v_2), v_3) : (v_1, (v_2, v_3)) \equiv_{assocl_*} ((v_1, v_2), v_3)} \qquad \frac{}{((v_1, v_2), v_3) \xrightarrow{assocr_*} (v_1, (v_2, v_3)) : ((v_1, v_2), v_3) \equiv_{assocr_*} (v_1,}$$

$$\frac{}{(\mathsf{inl}\ v_1, v_2) \xrightarrow{dist} \mathsf{inl}\ (v_1, v_2) : (\mathsf{inl}\ v_1, v_2) \equiv_{dist} \mathsf{inl}\ (v_1, v_2)} \qquad \frac{}{(\mathsf{inr}\ v_1, v_2) \xrightarrow{dist} \mathsf{inr}\ (v_1, v_2) : (\mathsf{inr}\ v_1, v_2) \equiv_{dist} \mathsf{inr}\ (v_1, v_2)}$$

$$\frac{}{\mathsf{inl}\ (v_1, v_2) \xrightarrow{factor} (\mathsf{inl}\ v_1, v_2) : \mathsf{inl}\ (v_1, v_2) \equiv_{factor} (\mathsf{inl}\ v_1, v_2)} \qquad \frac{}{\mathsf{inr}\ (v_1, v_2) \xrightarrow{factor} (\mathsf{inr}\ v_1, v_2) : \mathsf{inr}\ (v_1, v_2) \equiv_{factor} (\mathsf{inr}\ v_1, v_2)}$$

$$\frac{}{v \xrightarrow{id} v : v \equiv_{id} v} \qquad \frac{p : v_2 \equiv_c v_1}{!p : v_1 \equiv_{sym\ c} v_2} \qquad \frac{p : v_1 \equiv_{c_1} v_2 \quad q : v_2 \equiv_{c_2} v_3}{p \overset{v_2}{\bullet} q : v_1 \equiv_{c_1 \S c_2} v_3} \qquad \frac{p : v \equiv_{c_1} v'}{\mathsf{inl}\ p : \mathsf{inl}\ v \equiv_{c_1 \oplus c_2} \mathsf{inl}\ v'}$$

$$\frac{p : v \equiv_{c_2} v'}{\mathsf{inr}\ p : \mathsf{inr}\ v \equiv_{c_1 \oplus c_2} \mathsf{inr}\ v'} \qquad \frac{p : v_1 \equiv_{c_1} v_1' \quad q : v_2 \equiv_{c_2} v_2'}{(p, q) : (v_1, v_2) \equiv_{c_1 \otimes c_2} (v_1', v_2')} \qquad \frac{p : v \equiv_c v'}{(v \xrightarrow{id} v) \overset{v}{\bullet} p \xrightarrow{\mathsf{lid}} p : (v \xrightarrow{id} v) \overset{v}{\bullet} p \equiv_{\mathsf{lid}} p}$$

$$\frac{p : v' \equiv_c v}{p \overset{v}{\bullet} (v \xrightarrow{id} v) \xrightarrow{\mathsf{rid}} p : p \overset{v}{\bullet} (v \xrightarrow{id} v) \equiv_{\mathsf{rid}} p} \qquad \frac{}{!(\mathsf{inr}\ v \xrightarrow{identl_+} v) \xrightarrow{!1} v \xrightarrow{identr_+} \mathsf{inr}\ v} \qquad \frac{p : v' \equiv_c v}{(!p \overset{v'}{\bullet} p) \xrightarrow{l!} (v \xrightarrow{id} v) : (!p \overset{v'}{\bullet} p) \equiv_{l!} (v \xrightarrow{id} v)}$$

$$\frac{}{? :? \equiv_{r!} ?} \qquad \frac{}{? :? \equiv_{!!} ?} \qquad \frac{}{? :? \equiv_{\circ} ?}$$