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?

2. 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.

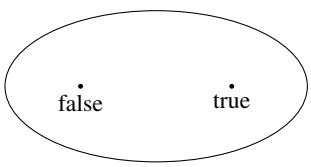
Formally, Martin-Löf type theory, is based on the principle that every proposition, i.e., every statement that is susceptible to proof, can be viewed as a type. The correspondence is validated by the following properties: if a proposition P is true, the corresponding type is inhabited, i.e., it is possible 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. In Agda notation, we can formally express this as follows:

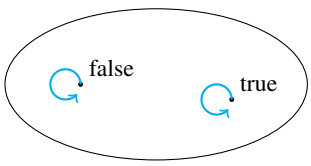
```
data _≡_ {ℓ} {A : Set ℓ} : (a b : A) → Set ℓ where
  refl : (a : A) → (a ≡ a)

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 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.

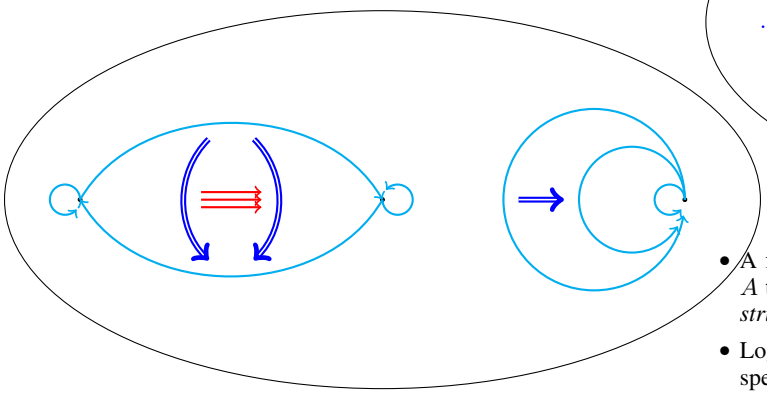
We are used to think of types as sets of values. So we think of the type `Bool` as:

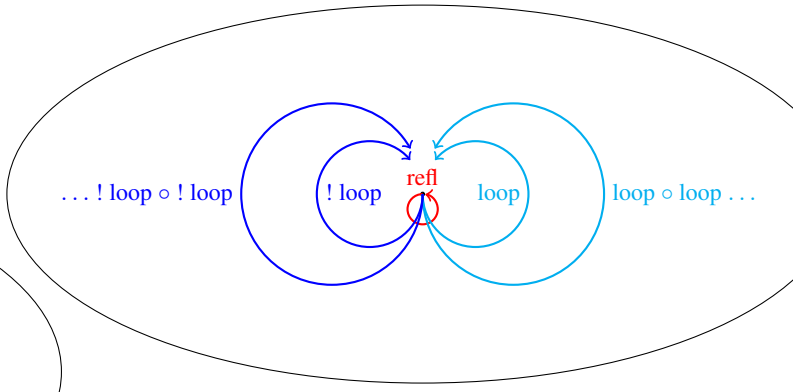


In HoTT, we should instead think about it as:



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:

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

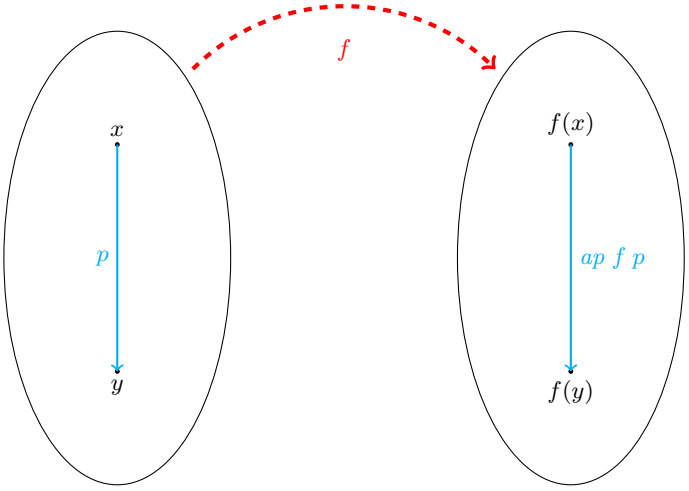
- With similar conditions one level up and so on and so forth.

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.

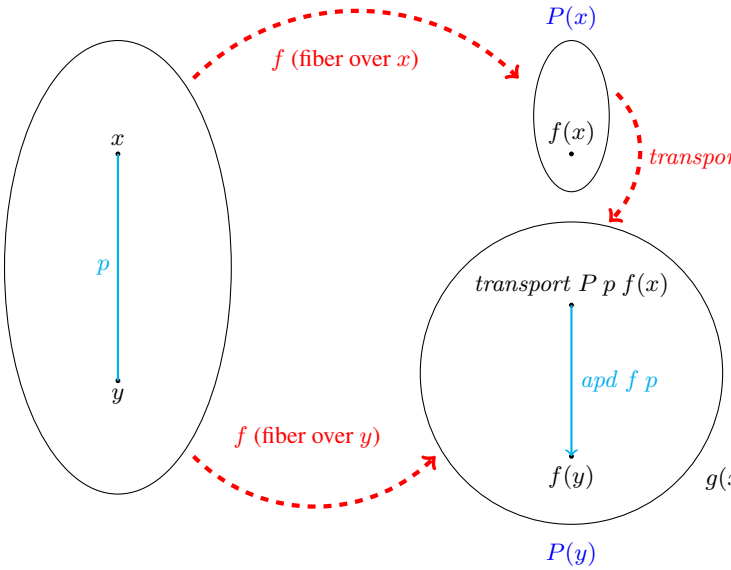


- $ap\ f\ p$ is the action of f on a path p;
- This satisfies the following properties:

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

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 $\mathit{transport}\ (f(x))$ 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:

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

- 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).

3. 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:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4. Theory

5. Pi

5.1 Base isomorphisms

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

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

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

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

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:

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