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 [The Univalent Foundations Program 2013] 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 [2012a] 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 a variation on Martin-Löf type theory in which all equalities are given *computational content*. We explain the basic ideas below.

2.1 Paths

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. Indeed, if a proposition P is true, the corresponding type is inhabited and it is possible to provide evidence or proof for P using one of the elements of the type P. If, however, a proposition P is false, the corresponding type is empty and it is impossible to provide a proof for P. The type theory is rich enough to express the standard logical propositions denoting conjunction, disjunction, implication, and existential and universal quantifications. In addition, 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, one may write proofs of this proposition as shown in the two small examples below:

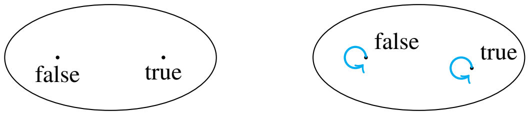
```
i0 : 3 ≡ 3
i0 = refl 3

i1 : (1 + 2) ≡ (3 * 1)
i1 = refl 3
```

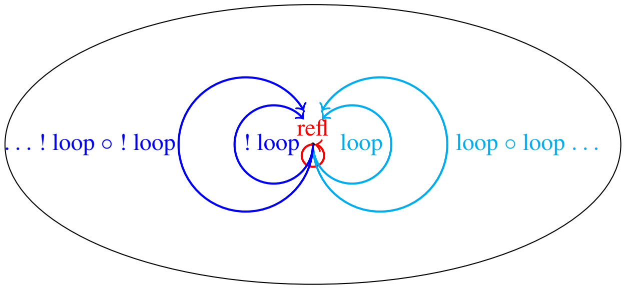
More generally, given two values m and n of type $\mathbb{N}$, it is possible to construct an element refl k of the type $m \equiv n$ if and only if m, n, and k are all "equal." As shown in example i1, this notion of *propositional equality* is not just syntactic equality but generalizes to *definitional equality*, i.e., to equality that can be established by normalizing the two values to their normal forms.

The important question from the HoTT perspective is the following: given two elements p and q of some type $x \equiv y$ with $x\, y : A$, what can we say about the elements of type $p \equiv q$. Or, in more familiar terms, given two proofs of some proposition P, are these two proofs themselves "equal." In some situations, the only interesting property of proofs is their existence, i.e., all proofs of the same proposition are considered equivalent. The twist that dates back to the paper by Hofmann and Streicher [1996] is that proofs actually possess a structure of great combinatorial complexity. HoTT builds on this idea by interpreting types as topological spaces or weak ∞-groupoids, and interpreting identities between elements of a type $x \equiv y$ as *paths* from the point x to the point y. If x and y are themselves paths, the elements of $x \equiv y$ become paths between paths, or homotopies in the topological language. To be explicit, we will often refer to types as *spaces* composed of *points*, paths, 2-paths, etc. and write $\equiv_A$ for the type of paths in space A.

As a simple example, 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 where there is a (trivial) path refl b from each point b to itself:



In this particular case, it makes no difference, but in general we may have a much more complicated path structure. The classical such example is the topological *circle* which is a space consisting of a point base and a *non trivial* 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 space A:

- For every path $p : x \equiv_A y$, there exists a path $!\,p : y \equiv_A x$;
- For every pair of paths $p : x \equiv_A y$ and $q : y \equiv_A z$, there exists a path $p \odot q : x \equiv_A z$;
- Subject to the following conditions:
  - $p \odot \mathsf{refl}\ y \equiv_{(x \equiv_A y)} p$;
  - $p \equiv_{(x \equiv_A y)} \mathsf{refl}\ x \odot p$
  - $!\,p \odot p \equiv_{(y \equiv_A y)} \mathsf{refl}\ y$
  - $p \odot !\,p \equiv_{(x \equiv_A x)} \mathsf{refl}\ x$
  - $!\,(!\,p) \equiv_{(x \equiv_A y)} p$
  - $p \odot (q \odot r) \equiv_{(x \equiv_A z)} (p \odot q) \odot r$
- This structure repeats one level up and so on ad infinitum.

2.2 Univalence

In addition to paths between the points false and true in the space Bool, it is also possible to consider paths between the space Bool and itself by considering Bool as a "point" in the universe Set of types. As usual, we have the trivial path which is given by the constructor refl:

```
p : Bool ≡ Bool
p = refl Bool
```

There are, however, other non trivial paths between Bool and itself and they are justified by *univalence* ***postulate***. As an example, the remainder of this section justifies that there is a path between Bool and itself corresponding to the boolean negation function.

We begin by formalizing the equivalence of functions $\sim$. Intuitively, two functions are equivalent if their results are propositionally equal for all inputs. A function $f : A \longrightarrow B$ is called an *equivalence* if there are functions g and h with whom its composition is the identity. Finally we say $A \simeq B$ if there is an equivalence between them:

```
_∼_ : ∀ {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

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

_≃_ : ∀ {ℓ ℓ'} (A : Set ℓ) (B : Set ℓ') → Set (ℓ ⊔ ℓ')
A ≃ B = Σ (A → B) isequiv
```

We can now formally state the univalence axiom:

```
postulate univalence : {A B : Set} → (A ≡ B) ≃ (A ≃ B)
```

A consequence of univalence is that equivalence of spaces implies a path between the spaces. In other words, in order to assert the existence of a path other than the trivial refl between Bool and itself, we need to find an equivalence between the space Bool and itself:

```
not2∼id : (not ∘ not) ∼ id
not2∼id false   =   refl false
not2∼id true    =   refl true

notequiv : Bool ≃ Bool
notequiv = (not , record {
    g = not ;
    α = λ b → not2∼id b ;
    h = not ;
    β = λ a → not2∼id a
                        })

not≡ : Bool ≡ Bool
not≡ with univalence
... | (_ , eq) = isequiv.g eq notequiv
```

2.3 Reversible Functions

Although the code asserting the existence of a non trivial path between Bool and itself "compiles," it is no longer executable as it relies on an Agda postulate. We analyze the situation from the perspective of reversible programming languages based on type isomorphisms [Bowman et al. 2011; James and Sabry 2012a,b].

The conventional HoTT approach starts with two, a priori, different notions: functions and paths, and then postulates an equivalence between a particular class of functions and paths. As illustrated above, functions like *not* correspond to a path like *not≡*. Most functions, however, are evidently unrelated to paths. In particular, any function $A \longrightarrow B$ that does not have an inverse $B \longrightarrow A$ cannot have any direct correspondence to paths. An interesting question poses itself though: since reversible computational models — in which all functions have inverses — are known to be universal computational models, what would happen if we considered a variant of HoTT based exclusively on reversible functions? Presumably in such a variant, all functions being reversible, would correspond to paths and the distinction between the two notions would vanish making the univalence postulate unnecessary. This is the precise idea we investigate in detail in the remainder of the paper.

3. Computing with Type Isomorphisms

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]. The main syntactic vehicle for the technical developments in this paper is a simple language called Π whose only computations are isomorphisms between finite types [2012a].

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:

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

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 2012a].[2]

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}{rlcl}
\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 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

[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 2012a]. We will not be concerned with this extension in the main body of this paper but it will be briefly discussed in the conclusion.

$identl_+ :$	$0 + \tau$	$\leftrightarrow$	τ	$: identr_+$
$swap_+ :$	$\tau_1 + \tau_2$	$\leftrightarrow$	$\tau_2 + \tau_1$	$: swap_+$
$assocl_+ :$	$\tau_1 + (\tau_2 + \tau_3)$	$\leftrightarrow$	$(\tau_1 + \tau_2) + \tau_3$	$: assocr_+$
$identl_* :$	$1 * \tau$	$\leftrightarrow$	τ	$: identr_*$
$swap_* :$	$\tau_1 * \tau_2$	$\leftrightarrow$	$\tau_2 * \tau_1$	$: swap_*$
$assocl_* :$	$\tau_1 * (\tau_2 * \tau_3)$	$\leftrightarrow$	$(\tau_1 * \tau_2) * \tau_3$	$: assocr_*$
$dist_0 :$	$0 * \tau$	$\leftrightarrow$	0	$: factor_0$
$dist :$	$(\tau_1 + \tau_2) * \tau_3$	$\leftrightarrow$	$(\tau_1 * \tau_3) + (\tau_2 * \tau_3)$	$: factor$

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

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

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

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

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

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 $(p \circ \mathsf{refl}\ \tau_2) \equiv_{\tau_1 \equiv_U \tau_2} p$. 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 ⟶ instead of the Π-notation ↔ or the HoTT notation ≡ to refer to paths.

5. Agda Model

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

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

⟦_⟧ : U → Set
⟦ ZERO ⟧         = ⊥
⟦ ONE ⟧          = ⊤
⟦ PLUS t₁ t₂ ⟧    = ⟦ t₁ ⟧ ⊎ ⟦ t₂ ⟧
⟦ TIMES t₁ t₂ ⟧ = ⟦ t₁ ⟧ × ⟦ t₂ ⟧

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

record U• : Set where
  constructor •[_,_]
  field
    |_| : U
    • : ⟦ |_| ⟧

open U•

Space : (t• : U•) → Set
Space •[ t , v ] = ⟦ t ⟧

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

- examples of plain types, values, and pointed types

ONE• : U•
ONE• = •[ ONE , tt ]

BOOL : U
BOOL = PLUS ONE ONE

BOOL² : U
BOOL² = TIMES BOOL BOOL

TRUE : ⟦ BOOL ⟧
TRUE = inj₁ tt

FALSE : ⟦ BOOL ⟧
FALSE = inj₂ tt

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

BOOL•T : U•
BOOL•T = •[ BOOL , TRUE ]

- The actual programs are the commutative semiring isomorp
- pointed types.

data _↔_ : U• → U• → Set where
  unite₊    : ∀ {t v} → •[ PLUS ZERO t , inj₂ v ] ↔ •[ t , v ]
  uniti₊    : ∀ {t v} → •[ t , v ] ↔ •[ PLUS ZERO t , inj₂ v ]
  swap1₊    : ∀ {t₁ t₂ v₁} → •[ PLUS t₁ t₂ , inj₁ v₁ ] ↔ •[ PLUS t₂ t₁ , inj₂ v₁
  swap2₊    : ∀ {t₁ t₂ v₂} → •[ PLUS t₁ t₂ , inj₂ v₂ ] ↔ •[ PLUS t₂ t₁ , inj₁ v₂
  assocl1₊ : ∀ {t₁ t₂ t₃ v₁} →
    •[ PLUS t₁ (PLUS t₂ t₃) , inj₁ v₁ ] ↔
    •[ PLUS (PLUS t₁ t₂) t₃ , inj₁ (inj₁ v₁) ]
  assocl2₊ : ∀ {t₁ t₂ t₃ v₂} →
    •[ PLUS t₁ (PLUS t₂ t₃) , inj₂ (inj₁ v₂) ] ↔
    •[ PLUS (PLUS t₁ t₂) t₃ , inj₁ (inj₂ v₂) ]
  assocl3₊ : ∀ {t₁ t₂ t₃ v₃} →
    •[ PLUS t₁ (PLUS t₂ t₃) , inj₂ (inj₂ v₃) ] ↔
    •[ PLUS (PLUS t₁ t₂) t₃ , inj₂ v₃ ]
  assocr1₊ : ∀ {t₁ t₂ t₃ v₁} →
```

```
      •[ PLUS (PLUS t₁ t₂) t₃ , inj₁ (inj₁ v₁) ] ↔
      •[ PLUS t₁ (PLUS t₂ t₃) , inj₁ v₁ ]
  assocr2₊ : ∀ {t₁ t₂ t₃ v₂} →
      •[ PLUS (PLUS t₁ t₂) t₃ , inj₁ (inj₂ v₂) ] ↔
      •[ PLUS t₁ (PLUS t₂ t₃) , inj₂ (inj₁ v₂) ]
  assocr3₊ : ∀ {t₁ t₂ t₃ v₃} →
      •[ PLUS (PLUS t₁ t₂) t₃ , inj₂ v₃ ] ↔
      •[ PLUS t₁ (PLUS t₂ t₃) , inj₂ (inj₂ v₃) ]
  unite⋆    : ∀ {t v} → •[ TIMES ONE t , (tt , v) ] ↔ •[ t , v ]
  uniti⋆    : ∀ {t v} → •[ t , v ] ↔ •[ TIMES ONE t , (tt , v) ]
  swap⋆     : ∀ {t₁ t₂ v₁ v₂} →
      •[ TIMES t₁ t₂ , (v₁ , v₂) ] ↔ •[ TIMES t₂ t₁ , (v₂ , v₁) ]
  assocl⋆ : ∀ {t₁ t₂ t₃ v₁ v₂ v₃} →
            •[ TIMES t₁ (TIMES t₂ t₃) , (v₁ , (v₂ , v₃)) ] ↔
            •[ TIMES (TIMES t₁ t₂) t₃ , ((v₁ , v₂) , v₃) ]
  assocr⋆ : ∀ {t₁ t₂ t₃ v₁ v₂ v₃} →
            •[ TIMES (TIMES t₁ t₂) t₃ , ((v₁ , v₂) , v₃) ] ↔
            •[ TIMES t₁ (TIMES t₂ t₃) , (v₁ , (v₂ , v₃)) ]
  distz : ∀ {t v absurd} →
            •[ TIMES ZERO t , (absurd , v) ] ↔ •[ ZERO , absurd ]
  factorz : ∀ {t v absurd} →
            •[ ZERO , absurd ] ↔ •[ TIMES ZERO t , (absurd , v) ]
  dist1     : ∀ {t₁ t₂ t₃ v₁ v₃} →
            •[ TIMES (PLUS t₁ t₂) t₃ , (inj₁ v₁ , v₃) ] ↔
            •[ PLUS (TIMES t₁ t₃) (TIMES t₂ t₃) , inj₁ (v₁ , v₃) ]
  dist2     : ∀ {t₁ t₂ t₃ v₂ v₃} →
            •[ TIMES (PLUS t₁ t₂) t₃ , (inj₂ v₂ , v₃) ] ↔
            •[ 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) ] ↔
            •[ TIMES (PLUS t₁ t₂) t₃ , (inj₁ v₁ , v₃) ]
  factor2   : ∀ {t₁ t₂ t₃ v₂ v₃} →
            •[ PLUS (TIMES t₁ t₃) (TIMES t₂ t₃) , inj₂ (v₂ , v₃) ] ↔
            •[ TIMES (PLUS t₁ t₂) t₃ , (inj₂ v₂ , v₃) ]
  id↔       : ∀ {t v} → •[ t , v ] ↔ •[ t , v ]
  sym↔      : ∀ {t₁ t₂ v₁ v₂} → (•[ t₁ , v₁ ] ↔ •[ t₂ , v₂ ]) →
      (•[ t₂ , v₂ ] ↔ •[ t₁ , v₁ ])
  _◎_       : ∀ {t₁ t₂ t₃ v₁ v₂ v₃} → (•[ t₁ , v₁ ] ↔ •[ t₂ , v₂ ]) →
      (•[ t₂ , v₂ ] ↔ •[ t₃ , v₃ ]) →
      (•[ t₁ , v₁ ] ↔ •[ t₃ , v₃ ])
  _⊕1_      : ∀ {t₁ t₂ t₃ t₄ v₁ v₂ v₃ v₄} →
      (•[ t₁ , v₁ ] ↔ •[ t₃ , v₃ ]) → (•[ t₂ , v₂ ] ↔ •[ t₄ , v₄ ]) →
      (•[ PLUS t₁ t₂ , inj₁ v₁ ] ↔ •[ PLUS t₃ t₄ , inj₁ v₃ ])
  _⊕2_      : ∀ {t₁ t₂ t₃ t₄ v₁ v₂ v₃ v₄} →
      (•[ t₁ , v₁ ] ↔ •[ t₃ , v₃ ]) → (•[ t₂ , v₂ ] ↔ •[ t₄ , v₄ ]) →
      (•[ PLUS t₁ t₂ , inj₂ v₂ ] ↔ •[ PLUS t₃ t₄ , inj₂ v₄ ])
  _⊗_        : ∀ {t₁ t₂ t₃ t₄ v₁ v₂ v₃ v₄} →
      (•[ t₁ , v₁ ] ↔ •[ t₃ , v₃ ]) → (•[ t₂ , v₂ ] ↔ •[ t₄ , v₄ ]) →
            (•[ TIMES t₁ t₂ , (v₁ , v₂) ] ↔ •[ TIMES t₃ t₄ , (v₃ , v₄) ])

- example programs

NOT•T : •[ BOOL , TRUE ] ↔ •[ BOOL , FALSE ]
NOT•T = swap1₊

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

CNOT•Fx : {b : ⟦ BOOL ⟧} →
            •[ BOOL² , (FALSE , b) ] ↔ •[ BOOL² , (FALSE , b) ]
CNOT•Fx = dist2 ◎ ((id↔ ⊗ NOT•F) ⊕2 id↔) ◎ factor2

CNOT•TF : •[ BOOL² , (TRUE , FALSE) ] ↔ •[ BOOL² , (TRUE , TRUE) ]
CNOT•TF = dist1 ◎
            ((id↔ ⊗ NOT•F) ⊕1 (id↔ {TIMES ONE BOOL} {(tt , TRUE)})) ◎
            factor1

CNOT•TT : •[ BOOL² , (TRUE , TRUE) ] ↔ •[ BOOL² , (TRUE , FALSE) ]
```

```
CNOT•TT = dist1 ◎
            ((id↔ ⊗ NOT•T) ⊕1 (id↔ {TIMES ONE BOOL} {(tt , TRUE)})
            factor1

- The evaluation of a program is not done in order to figu
- value. Both the input and output values are encoded in t
- program; what the evaluation does is follow the path to
- reach the output value from the input value. Even though
- same pointed types are, by definition, observationally e
- may follow different paths. At this point, we simply dec
- programs are "the same." At the next level, we will weak
- irrelevant" equivalence and reason about which paths can
- other paths via 2paths etc.

- Even though individual types are sets, the universe of t
- groupoid. The objects of this groupoid are the pointed t
- morphisms are the programs; and the equivalence of progr
- degenerate observational equivalence that equates every
- are extensionally equivalent.

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

UG : 1Groupoid
UG = record
    { set = U•
    ; _↝_ = _↔_
    ; _≈_ = _obs≅_
    ; id = id↔
    ; _∘_ = λ y↔z x↔y → x↔y ◎ y↔z
    ; _⁻¹ = sym↔
    ; 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
    }

------------------------------------------------------
- Simplify various compositions

simplifySym : {t₁ t₂ : U•} → (c₁ : t₁ ↔ t₂) → (t₂ ↔ 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 id↔ = id↔
simplifySym (sym↔ c) = c
simplifySym (c₁ ◎ c₂) = simplifySym c₂ ◎ 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

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

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

6. 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:

7. Theory

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.

M. Hofmann and T. Streicher. The groupoid interpretation of type theory. In *Venice Festschrift*, pages 83–111, 1996.

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

R. P. James and A. Sabry. Isomorphic interpreters from logically reversible abstract machines. In *RC*, 2012b.

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.

The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `http://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

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

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

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

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

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

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

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

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

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

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

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

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

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

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