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

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].

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 these propositions as shown in the two examples below:

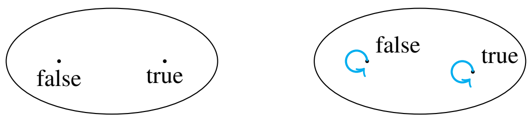
```
i0 : 3 ≡ 3
i0 = refl 3
```

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

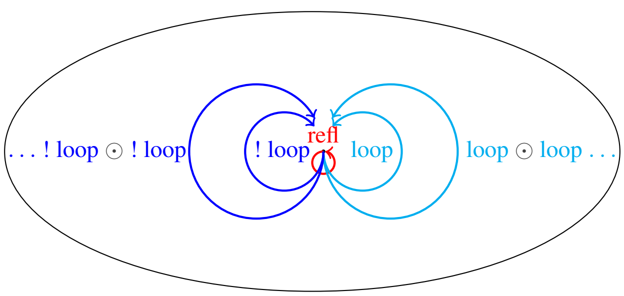
More generally, given two values m and n of type $\mathbb{N}$, it is possible to construct an element refl k of the type $m \equiv n$ if and only if m, n, and k are all "equal." As shown in example i1, this notion of *propositional equality* is not just syntactic equality but generalizes to *definitional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

An 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. This therefore suggests that the exact sequence of logical steps in the proof is irrelevant, and ultimately that all proofs of the same proposition are equivalent. This is however neither necessary nor desirable. A twist that dates back to a 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 (2paths), or homotopies in the topological language. To be explicit, we will often refer to types as *spaces* which consist of *points*, paths, 2paths, 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.

A space that satisfies the properties above for n levels is called an n-groupoid.

2.2 Univalence

In addition to paths between the points within a space like 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 the *univalence* **axiom**. 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 two spaces A and B are equivalent, $A \simeq B$, 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)
```

For our purposes, the important consequence of the univalence axiom is that equivalence of spaces implies the existence of a path between the spaces. In other words, in order to assert the existence of a path notpath between Bool and itself, we need to prove that the boolean negation function is an equivalence between the space Bool and itself. This is easy to show:

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

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

notpath : Bool ≡ Bool
notpath with univalence
... | (_ , eq) = isequiv.g eq notequiv
```

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. In the next section, we analyze this situation from the perspective of reversible programming languages based on type isomorphisms [Bowman et al. 2011; James and Sabry 2012a,b].

3. Computing with Type Isomorphisms

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]. After reviewing the motivation for this language and its relevance to HoTT, we present its syntax and semantics.

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

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

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

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

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

3.1 Reversibility

The relevance of reversibility to HoTT is based on the following analysis. 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, some functions like `not` correspond to paths. Most functions, however, are evidently unrelated to paths. In particular, any function of type $A \rightarrow B$ that does not have an inverse of type $B \rightarrow A$ cannot have any direct correspondence to paths as all paths have inverses. An interesting question then poses itself: 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 potentially correspond to paths and the distinction between the two notions would vanish making the univalence postulate unnecessary. This is the precise technical idea we investigate in detail in the remainder of the paper.

3.2 Syntax and Semantics of Π

The Π family of languages is based on type isomorphisms. In the variant we consider, 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:

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

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 Fig. 1) and compositions (on the right side of the same figure). Each line of the figure 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]

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 and satisfying the axioms of a commutative rig (i.e., a commutative ring without negative elements also known as a commutative semiring) up to coherent isomorphisms. And indeed the types of the Π-combinators are precisely the commutative 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, interpret the values as elements in these finite sets, and interpret the combinators as permutations.

$$
\begin{array}{rlcl}
identl_+ & \triangleright (\mathsf{inr}\ v) & = & v \\
identr_+ & \triangleright v & = & \mathsf{inr}\ v \\
swap_+ & \triangleright (\mathsf{inl}\ v) & = & \mathsf{inr}\ v \\
swap_+ & \triangleright (\mathsf{inr}\ v) & = & \mathsf{inl}\ v \\
assocl_+ & \triangleright (\mathsf{inl}\ v) & = & \mathsf{inl}\ (\mathsf{inl}\ v) \\
assocl_+ & \triangleright (\mathsf{inr}\ (\mathsf{inl}\ v)) & = & \mathsf{inl}\ (\mathsf{inr}\ v) \\
assocl_+ & \triangleright (\mathsf{inr}\ (\mathsf{inr}\ v)) & = & \mathsf{inr}\ v \\
assocr_+ & \triangleright (\mathsf{inl}\ (\mathsf{inl}\ v)) & = & \mathsf{inl}\ v \\
assocr_+ & \triangleright (\mathsf{inl}\ (\mathsf{inr}\ v)) & = & \mathsf{inr}\ (\mathsf{inl}\ v) \\
assocr_+ & \triangleright (\mathsf{inr}\ v) & = & \mathsf{inr}\ (\mathsf{inr}\ v) \\
identl_* & \triangleright ((), v) & = & v \\
identr_* & \triangleright v & = & ((), v) \\
swap_* & \triangleright (v_1, v_2) & = & (v_2, v_1) \\
assocl_* & \triangleright (v_1, (v_2, v_3)) & = & ((v_1, v_2), v_3) \\
assocr_* & \triangleright ((v_1, v_2), v_3) & = & (v_1, (v_2, v_3)) \\
dist & \triangleright (\mathsf{inl}\ v_1, v_3) & = & \mathsf{inl}\ (v_1, v_3) \\
dist & \triangleright (\mathsf{inr}\ v_2, v_3) & = & \mathsf{inr}\ (v_2, v_3) \\
factor & \triangleright (\mathsf{inl}\ (v_1, v_3)) & = & (\mathsf{inl}\ v_1, v_3) \\
factor & \triangleright (\mathsf{inr}\ (v_2, v_3)) & = & (\mathsf{inr}\ v_2, v_3) \\
id & \triangleright v & = & v \\
(c_1 \fatsemi c_2) & \triangleright v & = & c_2 \triangleright (c_1 \triangleright v) \\
(c_1 \oplus c_2) & \triangleright (\mathsf{inl}\ v) & = & \mathsf{inl}\ (c_1 \triangleright v) \\
(c_1 \oplus c_2) & \triangleright (\mathsf{inr}\ v) & = & \mathsf{inr}\ (c_2 \triangleright v) \\
(c_1 \otimes c_2) & \triangleright (v_1, v_2) & = & (c_1 \triangleright v_1, c_2 \triangleright v_2)
\end{array}
$$

Figure 2. Operational Semantics

In the remainder of this paper, we will be more interested in a model based on groupoids. But first, we give an operational semantics for Π. 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 in Fig. 2; the backwards evaluator differs in trivial ways.

3.3 Groupoid Model

Instead of modeling the types of Π using sets and the combinators using permutations we use a semantics that identifies Π-combinators with *paths*. More precisely, we model the universe of Π-types as a space `U` whose points are the individual Π-types (which are themselves spaces t containing points). We then postulate that there is path between the spaces t_1 and t_2 if there is a Π-combinator $c : t_1 \leftrightarrow t_2$. Our postulate is similar in spirit to the univalence axiom but, unlike the latter, it has a simple computa-

[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.

tional interpretation. A path directly corresponds to a type isomorphism with a clear operational semantics as presented in the previous section. As we will explain in more detail below, this approach replaces the datatype $\equiv$ modeling propositional equality with the datatype $\leftrightarrow$ modeling type isomorphisms. With this switch, the Π-combinators of Fig. 1 become *syntax* for the paths in the space U. Put differently, instead of having exactly one constructor refl for paths with all other paths discovered by proofs (see Secs. 2.5–2.12 of the HoTT book [2013]) or postulated by the univalence axiom, we have an *inductive definition* that completely specifies all the paths in the space U.

We begin with the datatype definition of the universe U of finite types which are constructed using ZERO, ONE, PLUS, and TIMES. Each of these finite types will correspond to a set of points with paths connecting some of the points. The underlying set of points is computed by ⟦_⟧ as follows: ZERO maps to the empty set $\bot$, ONE maps to the singleton set $\top$, PLUS maps to the disjoint union $\uplus$, and TIMES maps to the cartesian product $\times$.

```
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 ⟧
```

We want to identify paths with Π-combinators. There is a small complication however: paths are ultimately defined between points but the Π-combinators of Fig. 1 are defined between spaces. We can bridge this gap using a popular HoTT concept, that of *pointed spaces*. A pointed space •[t , v] is a space t : U with a distinguished value v : ⟦ t ⟧:

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

Given pointed spaces, it is possible to re-express the Π-combinators as shown in Fig. 3. The new presentation of combinators directly relates points to points and in fact subsumes the operational semantics of Fig. 2. For example $\mathsf{swap1}_{+}$ is still an operation from the space PLUS t_1 t_2 to itself but in addition it specifies that, within that spaces, it maps the point inj_1 v_1 to the point inj_2 v_1.

We note that the refinement of the Π-combinators to combinators on pointed spaces is given by an inductive family for *heterogeneous* equality that generalizes the usual inductive family for propositional equality. Put differently, what used to be the only constructor for paths refl is now just one of the many constructors (named id$\leftrightarrow$ in the figure). Among the new constructors and we have $\odot$ that constructs path compositions. By construction, every combinator has an inverse calculated as shown in Fig. 4. These constructions are sufficient to guarantee that the universe U is a groupoid. Additionally, we have paths that connect values in different but isomorphic spaces like •[TIMES t_1 t_2 , (v_1 , v_2)] and •[TIMES t_2 t_1 , (v_2 , v_1)].

The example notpath which earlier required the use of the univalence axiom can now be directly defined using $\mathsf{swap1}_{+}$ and $\mathsf{swap2}_{+}$. To see this, note that Bool can be viewed as a shorthand for PLUS ONE ONE with true and false as shorthands for inj_1 tt and inj_2 tt. With this in mind, the path corresponding to boolean negation consists of two "fibers", one for each boolean value as shown below:

```
data Path (t₁ t₂ : U•) : Set where
  path : (c : t₁ ↔ t₂) → Path t₁ t₂

BOOL : U
BOOL = PLUS ONE ONE

TRUE FALSE : ⟦ BOOL ⟧
TRUE    = inj₁ tt
FALSE   = inj₂ tt

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

BOOL•T : U•
BOOL•T = •[ BOOL , TRUE ]

NOT•T : BOOL•T ↔ BOOL•F
NOT•T = swap1₊

NOT•F : BOOL•F ↔ BOOL•T
NOT•F = swap2₊

notpath•T : Path BOOL•T BOOL•F
notpath•T = path NOT•T

notpath•F : Path BOOL•F BOOL•T
notpath•F = path NOT•F
```

In other words, a path between spaces is really a collection of paths connecting the various points. Note however that we never need to "collect" these paths using a universal quantification.

4. Computing with Paths

The previous section presented a language Π whose computations are all the possible isomorphisms between finite types. (Recall that the commutative semiring structure is sound and complete for isomorphisms between finite types [Fiore 2004; Fiore et al. 2006].) Instead of working with arbitrary functions, then restricting them to equivalences, and then postulating that these equivalences give rise to paths, the approach based on Π starts directly with the full set of possible isomorphisms and encodes it as an inductive datatype of paths between pointed spaces. The resulting structure is evidently a 1-groupoid as the isomorphisms are closed under inverses and composition. We now investigate the higher groupoid structure.

The pleasant result will be that the higher groupoid structure will result from another "lifted" version of Π in which computations manipulate paths. The lifted version will have all the combinators from Fig. 3 to manipulate collections of paths (e.g., sums of products of paths) in addition to combinators that work on individual paths. The latter combinators will capture the higher groupoid structure relating for example, the path id$\leftrightarrow$ $\odot$ c with c. Computations in the lifted Π will therefore correspond to 2paths and the entire scheme can be repeated over and over to capture the concept of weak ∞-groupoids.

4.1 Examples

We start with a few examples where we define a collection of paths p_1 to p_5 all from the pointed space •[BOOL , TRUE] to the pointed space •[BOOL , FALSE]:

```
T↔F : Set
T↔F = Path BOOL•T BOOL•F

p₁ p₂ p₃ p₄ p₅ : T↔F
```

```
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 ]
  _◎_ : ∀ {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) ])
```

Figure 3. Pointed version of Π-combinators or inductive definition of paths

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

Figure 4. Pointed Combinators or paths inverses

```
p1 = path NOT•T
p2 = path (id↔ ◎ NOT•T)
p3 = path (NOT•T ◎ NOT•F ◎ NOT•T)
p4 = path (NOT•T ◎ id↔)
p5 = path   (uniti⋆ ◎ swap⋆ ◎ (NOT•T ⊗ id↔) ◎
            swap⋆ ◎ unite⋆)
```

All the paths start at TRUE and end at FALSE but follow different intermediate paths along the way. Informally p_1 is the most "efficient" canonical way of connecting TRUE to FALSE via the appropriate fiber of the boolean negation. Path p_2 starts with the trivial path from TRUE to itself and then uses the boolean negation. The first step is clearly superfluous and hence we expect, via the groupoid laws, to have a 2path connecting p_2 to p_1. Path p_3 does not syntactically refer to a trivial path but instead uses what is effectively a trivial path that follows a negation path and then its inverse. We also expect to have a 2path between this path and the other ones. Path p_4 is also evidently equivalent to the others but the situation with path p_5 is more subtle. We defer the discussion until we formally define 2paths. For now, we note that — viewed extensionally — each path connects TRUE to FALSE and hence all the paths are extensionally equivalent. In the conventional approach to programming language semantics, which is also followed in the current formalization of HoTT, this extensional equivalence would then be used to justify the existence of the 2paths. In our setting, we do *not* need to reason using extensional methods since all functions (paths) are between pointed spaces (i.e., are point to point) and we have an inductive definition of paths which can be used to define computational rules that simplify paths as shown next.

4.2 Path Induction

only REVERSIBLE functions from paths to paths are allowed. we could write these functions as functions and prove inverses etc but then we are back to the extensional view. better to axiomatize the groupoid laws using combinators

In the conventional formalization to HoTT, the groupoid laws are a consequence of *path induction*. The situation is similar in our case but with an important difference: our inductively defined type family of paths includes many more constructors than just refl. Consider for example, the inductively defined function ! in Fig. 4 which shows that each path has an inverse by giving the computational rule for calculating that inverse. We will use this definition to postulate a 2path between ?? c and ! c. More generally, we can postulate a 2-path between any path c : ($t_1 \leftrightarrow t_2$) and the result of replacing any constructor cons in c by an inductively defined function fcons. The intuitive reason this is correct is that the functional version of the constructor fcons must respect the types which means it is extensionally equivalent to the constructor itself as all the types are pointed. For example, if ?? c maps FALSE to TRUE, the result of ! c must also do the same.

In the following we will use, without giving the full definitions, the following inductive functions that can be used to replace various patterns of constructors:

- simplifyl⊚
- simplifyr⊚
-
-
-

We can similarly perform nested induction to simplify path composition. We omit this large function but show a couple of interesting cases:

4.3 Level 1 Π

regular pi is level 0

```
data 2U : Set where
  2ZERO  : 2U
  2ONE   : 2U
  2PLUS  : 2U → 2U → 2U
  2TIMES : 2U → 2U → 2U
  PATH   : {t1 t2 : U•} → (t1 ⟷ t2) → 2U

2⟦_⟧ : 2U → Set
2⟦ 2ZERO ⟧          = ⊥
2⟦ 2ONE ⟧           = ⊤
2⟦ 2PLUS t1 t2 ⟧    = 2⟦ t1 ⟧ ⊎ 2⟦ t2 ⟧
2⟦ 2TIMES t1 t2 ⟧   = 2⟦ t1 ⟧ × 2⟦ t2 ⟧
2⟦ PATH {t1} {t2} c ⟧ = Path t1 t2
```

– empty set of paths – a trivial path – disjoint union of paths – pairs of paths – level 0 paths between values groupoid laws not enough

the equivalent of path induction is the induction principle for the type family defining paths

```
record 2U• : Set where
  constructor 2•[_,_]
  field
    2|_| : 2U
    2• : 2⟦ 2|_| ⟧

Path• : {t1 t2 : U•} → (c : t1 ⟷ t2) → 2U•
Path• c = 2•[ PATH c , path c ]

data _⇔_ : 2U• → 2U• → Set where
  id⟷ : {t : 2U•} → t ⇔ t
  _⊚_ : {t1 t2 t3 : 2U•} → (t1 ⇔ t2) → (t2 ⇔ t3) → (t1 ⇔ t3)

  simplifyl⊚l : ∀ {t1 t2 t3 v1 v2 v3}
    {c1 : •[ t1 , v1 ] ⟷ •[ t2 , v2 ]}
    {c2 : •[ t2 , v2 ] ⟷ •[ t3 , v3 ]} →
    Path• (c1 ⊚ c2) ⇔ Path• (simplifyl⊚ c1 c2)
  simplifyl⊚r : {t1 t2 t3 : U•} {c1 : t1 ⟷ t2} {c2 : t2 ⟷ t3} →
    Path• (simplifyl⊚ c1 c2) ⇔ Path• (c1 ⊚ c2)
  simplifyr⊚l : ∀ {t1 t2 t3 v1 v2 v3}
    {c1 : •[ t1 , v1 ] ⟷ •[ t2 , v2 ]}
    {c2 : •[ t2 , v2 ] ⟷ •[ t3 , v3 ]} →
    Path• (c1 ⊚ c2) ⇔ Path• (simplifyr⊚ c1 c2)
  simplifyr⊚r : {t1 t2 t3 : U•} {c1 : t1 ⟷ t2} {c2 : t2 ⟷ t3} →
    Path• (simplifyr⊚ c1 c2) ⇔ Path• (c1 ⊚ c2)
  simplifySyml : {t1 t2 : U•} {c : t1 ⟷ t2} →
    Path• (! c) ⇔ Path• (! c)
  simplifySymr : {t1 t2 : U•} {c : t1 ⟷ t2} →
    Path• (! c) ⇔ Path• (! c)
  invll   : ∀ {t1 t2 v1 v2} → {c : •[ t1 , v1 ] ⟷ •[ t2 , v2 ]} →
    Path• (! c ⊚ c) ⇔ Path• (id⟷ {t2} {v2})
  invlr   : ∀ {t1 t2 v1 v2} → {c : •[ t1 , v1 ] ⟷ •[ t2 , v2 ]} →
    Path• (id⟷ {t2} {v2}) ⇔ Path• (! c ⊚ c)
  invrl   : ∀ {t1 t2 v1 v2} → {c : •[ t1 , v1 ] ⟷ •[ t2 , v2 ]} →
    Path• (c ⊚ ! c) ⇔ Path• (id⟷ {t1} {v1})
  invrr   : ∀ {t1 t2 v1 v2} → {c : •[ t1 , v1 ] ⟷ •[ t2 , v2 ]} →
    Path• (id⟷ {t1} {v1}) ⇔ Path• (c ⊚ ! c)
  resp⊚   : {t1 t2 t3 : U•}
    {c1 : t1 ⟷ t2} {c2 : t2 ⟷ t3}
    {c3 : t1 ⟷ t2} {c4 : t2 ⟷ t3} →
    (Path• c1 ⇔ Path• c3) →
    (Path• c2 ⇔ Path• c4) →
    Path• (c1 ⊚ c2) ⇔ Path• (c3 ⊚ c4)
```

Simplify various compositions

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 introduces an inverse path $!p$ for each path p introduced by c. The composition operator $⨟$ introduces a path $p \odot q$ 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 needs to proved from first principles for some types and *postulated* for the universe type 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.

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 $A \rightarrow B$ 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.

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:

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.

for examples of 2 paths look at proofs of path assoc; triangle and pentagon rules

the idea I guess is that instead of having the usual evaluator where values flow, we want an evaluator that rewrites the circuit to primitive isos; for that we need some normal form for permutations and a proof that we can rewrite any circuit to this normal form

plan after that: add trace; this make obs equiv much more interesting and allows a limited form of h.o. functions via the int construction and then do the ring completion to get more complete notion of h.o. functions

Level 1: Types are sets of paths. The paths are defined at the previous level (level 0). At level 1, there is no interesting 2path structure. From the perspective of level 0, we have points with non-trivial paths between them, i.e., we have a groupoid. The paths cross type boundaries, i.e., we have heterogeneous equality

4.4 2Paths

```
-- let's try to prove that p₁ = p₂ = p₃ = p₄ = p₅

-- p₁ ~> p₂
α₄ : 2●[ PATH NOT•T , p₁ ] ⇔ 2●[ PATH (id⟷ ◎ NOT•T) , p₂ ]
α₄ = simplifyl◎r

-- p₂ ~> p₃
α₅ : 2●[ PATH (id⟷ ◎ NOT•T) , p₂ ] ⇔
     2●[ PATH (NOT•T ◎ NOT•F ◎ NOT•T) , p₃ ]
α₅ = id⟷ ◎ NOT•T
       ≡⟨ simplifyl◎l ⟩
     NOT•T
       ≡⟨ simplifyr◎r ⟩
     NOT•T ◎ id⟷
       ≡⟨ resp◎ id⟷ simplifyl◎r ⟩
     NOT•T ◎ NOT•F ◎ NOT•T ∎

-- p₃ ~> p₄
α₆ : 2●[ PATH (NOT•T ◎ NOT•F ◎ NOT•T) , p₃ ] ⇔
     2●[ PATH (NOT•T ◎ id⟷) , p₄ ]
α₆ = resp◎ id⟷ simplifyl◎l

-- p₅ ~> p₁

α₈ : 2●[ PATH (uniti⋆ ◎ swap⋆ ◎
          (NOT•T ⊗ id⟷) ◎ swap⋆ ◎ unite⋆) ,
     p₅ ] ⇔
     2●[ PATH NOT•T , p₁ ]
α₈ = uniti⋆ ◎ swap⋆ ◎ (NOT•T ⊗ id⟷) ◎ swap⋆ ◎ unite⋆
       ≡⟨ resp◎ id⟷ (resp◎ id⟷ simplifyl◎r) ⟩
     uniti⋆ ◎ (swap⋆ ◎ ((NOT•T ⊗ id⟷) ◎ (swap⋆ ◎ unite⋆)))
       ≡⟨ resp◎ id⟷ (resp◎ id⟷ simplifyl◎r) ⟩
     uniti⋆ ◎ (swap⋆ ◎ (((NOT•T ⊗ id⟷) ◎ swap⋆) ◎ unite⋆))
       ≡⟨ resp◎ id⟷ (resp◎ id⟷ (resp◎ simplifyl◎l id⟷ )) ⟩
     uniti⋆ ◎ (swap⋆ ◎ ((swap⋆ ◎ (id⟷ ⊗ NOT•T)) ◎ unite⋆))
       ≡⟨ resp◎ id⟷ (resp◎ id⟷ simplifyl◎l) ⟩
     uniti⋆ ◎ (swap⋆ ◎ (swap⋆ ◎ ((id⟷ ⊗ NOT•T) ◎ unite⋆)))
       ≡⟨ resp◎ id⟷ simplifyl◎r ⟩
     uniti⋆ ◎ ((swap⋆ ◎ swap⋆) ◎ ((id⟷ ⊗ NOT•T) ◎ unite⋆))
       ≡⟨ resp◎ id⟷ (resp◎ simplifyl◎l id⟷) ⟩
     uniti⋆ ◎ (id⟷ ◎ ((id⟷ ⊗ NOT•T) ◎ unite⋆))
       ≡⟨ resp◎ id⟷ simplifyl◎l ⟩
     uniti⋆ ◎ ((id⟷ ⊗ NOT•T) ◎ unite⋆)
       ≡⟨ resp◎ id⟷ simplifyl◎l    ⟩
     uniti⋆ ◎ (unite⋆ ◎ NOT•T)
       ≡⟨ simplifyl◎r ⟩
     (uniti⋆ ◎ unite⋆) ◎ NOT•T
       ≡⟨ resp◎ simplifyl◎l id⟷ ⟩
     id⟷ ◎ NOT•T
       ≡⟨ simplifyl◎l ⟩
     NOT•T ∎

-- p₄ ~> p₅

α₇ : 2●[ PATH (NOT•T ◎ id⟷) , p₄ ] ⇔
     2●[ PATH (uniti⋆ ◎ swap⋆ ◎
          (NOT•T ⊗ id⟷) ◎ swap⋆ ◎ unite⋆) ,
     p₅ ]
α₇ = simplifyr◎l ◎ {!!} -- (! α₈)
G : 1Groupoid
G = record
    { set = U•
    ; _↝_ = _⟷_
    ; _≈_ = λ c₀ c₁ → Path• c₀ ⇔ Path• c₁
    ; id = id⟷
    ; _∘_ = λ c₀ c₁ → c₁ ◎ c₀
    ; _⁻¹ = !
    ; lneutr = λ _ → simplifyr◎l
    ; rneutr = λ _ → simplifyl◎l
    ; assoc = λ _ _ _ → simplifyl◎r
    ; equiv = record { refl = id⟷
                     ; sym = λ c → {!!} -- ! c
                     ; trans = λ c₀ c₁ → c₀ ◎ c₁ }
    ; linv = λ _ → invrl
    ; rinv = λ _ → invll
    ; ∘-resp-≈ = λ f⟷h g⟷i → resp◎ g⟷i f⟷h
    }
```

5. The Int Construction

2paths are functions on paths; the int construction reifies these functions/2paths as 1paths

In the context of monoidal categories, it is known that a notion of higher-order functions emerges from having an additional degree of *symmetry*. In particular, both the **Int** construction of Joyal, Street, and Verity [?] and the closely related $\mathcal{G}$ construction of linear logic [?] construct higher-order *linear* functions by considering a new category built on top of a given base traced monoidal category. The objects of the new category are of the form $\boxed{\tau_1}\boxed{\tau_2}$ where τ_1 and τ_2 are objects in the base category. Intuitively, this object represents the *difference* $\tau_1 - \tau_2$ with the component τ_1 viewed as conventional type whose elements represent values flowing, as usual, from producers to consumers, and the component τ_2 viewed as a *negative type* whose elements represent demands for values or equivalently values flowing backwards. Under this interpretation, and as we explain below, a function is nothing but an object that converts a demand for an argument into the production of a result.

We begin our formal development by extending Π with a new universe of types $\mathbb{T}$ that consists of composite types $\boxed{\tau_1}\boxed{\tau_2}$:

$$(\mathit{1d\ types}) \quad \mathbb{T} \quad ::= \quad \boxed{\tau_1}\boxed{\tau_2}$$

In anticipation of future developments, we will refer to the original types τ as 0-dimensional (0d) types and to the new types $\mathbb{T}$ as 1-dimensional (1d) types. It turns out that, except for one case discussed below, the 1d level is a "lifted" instance of Π with its own notions of empty, unit, sum, and product types, and its corresponding notion of isomorphisms on these 1d types.

Our next step is to define lifted versions of the 0d types:

$$\begin{array}{rcl}
\mathbb{0} & \triangleq & \boxed{0}\boxed{0} \\
\mathbb{1} & \triangleq & \boxed{1}\boxed{0} \\
\boxed{\tau_1}\boxed{\tau_2} \boxplus \boxed{\tau_3}\boxed{\tau_4} & \triangleq & \boxed{\tau_1 + \tau_3}\boxed{\tau_2 + \tau_4} \\
\boxed{\tau_1}\boxed{\tau_2} \boxtimes \boxed{\tau_3}\boxed{\tau_4} & \triangleq & \\
 & & \boxed{(\tau_1 * \tau_3) + (\tau_2 * \tau_4)}\boxed{(\tau_1 * \tau_4) + (\tau_2 * \tau_3)}
\end{array}$$

Building on the idea that Π is a categorification of the natural numbers and following a long tradition that relates type isomorphisms and arithmetic identities [**?**], one is tempted to think that the **Int** construction (as its name suggests) produces a categorification of the integers. Based on this hypothesis, the definitions above can be intuitively understood as arithmetic identities. The same arithmetic intuition explains the lifting of isomorphisms to 1d types:

$$\boxed{\tau_1}\boxed{\tau_2} \Leftrightarrow \boxed{\tau_3}\boxed{\tau_4} \quad \triangleq \quad (\tau_1 + \tau_4) \leftrightarrow (\tau_2 + \tau_3)$$

In other words, an isomorphism between 1d types is really an isomorphism between "re-arranged" 0d types where the negative input τ_2 is viewed as an output and the negative output τ_4 is viewed as an input. Using these ideas, it is now a fairly standard exercise to define the lifted versions of most of the combinators in Table 1.[3] There are however a few interesting cases whose appreciation is essential for the remainder of the paper that we discuss below.

Easy Lifting. Many of the 0d combinators lift easily to the 1d level. For example:

$$\begin{array}{rcl}
id & : & \mathbb{T} \Leftrightarrow \mathbb{T} \\
 & : & \boxed{\tau_1}\boxed{\tau_2} \Leftrightarrow \boxed{\tau_1}\boxed{\tau_2} \\
 & \triangleq & (\tau_1 + \tau_2) \leftrightarrow (\tau_2 + \tau_1) \\
id & = & swap_+ \\
 & & \\
identl_+ & : & \mathbb{0} \boxplus \mathbb{T} \Leftrightarrow \mathbb{T} \\
 & = & assocr_+ \fatsemi (id \oplus swap_+) \fatsemi assocl_+
\end{array}$$

Composition using $trace$***.***

$$\begin{array}{rl}
(\fatsemi) : & (\mathbb{T}_1 \Leftrightarrow \mathbb{T}_2) \to (\mathbb{T}_2 \Leftrightarrow \mathbb{T}_3) \to (\mathbb{T}_1 \Leftrightarrow \mathbb{T}_3) \\
f \fatsemi g = & trace\ (assoc_1 \fatsemi (f \oplus id) \fatsemi assoc_2 \fatsemi (g \oplus id) \fatsemi assoc_3)
\end{array}$$

New combinators $curry$ ***and*** $uncurry$ ***for higher-order functions.***

$$\begin{array}{rcl}
\boxminus(\boxed{\tau_1}\boxed{\tau_2}) & \triangleq & \boxed{\tau_2}\boxed{\tau_1} \\
\boxed{\tau_1}\boxed{\tau_2} \multimap \boxed{\tau_3}\boxed{\tau_4} & \triangleq & \boxminus(\boxed{\tau_1}\boxed{\tau_2}) \boxplus \boxed{\tau_3}\boxed{\tau_4} \\
 & \triangleq & \boxed{\tau_2 + \tau_3}\boxed{\tau_1 + \tau_4}
\end{array}$$

$$\begin{array}{rcl}
flip & : & (\mathbb{T}_1 \Leftrightarrow \mathbb{T}_2) \to (\boxminus\mathbb{T}_2 \Leftrightarrow \boxminus\mathbb{T}_1) \\
flip\ f & = & swap_+ \fatsemi f \fatsemi swap_+ \\
 & & \\
curry & : & ((\mathbb{T}_1 \boxplus \mathbb{T}_2) \Leftrightarrow \mathbb{T}_3) \to (\mathbb{T}_1 \Leftrightarrow (\mathbb{T}_2 \multimap \mathbb{T}_3)) \\
curry\ f & = & assocl_+ \fatsemi f \fatsemi assocr_+ \\
 & & \\
uncurry & : & (\mathbb{T}_1 \Leftrightarrow (\mathbb{T}_2 \multimap \mathbb{T}_3)) \to ((\mathbb{T}_1 \boxplus \mathbb{T}_2) \Leftrightarrow \mathbb{T}_3) \\
uncurry\ f & = & assocr_+ \fatsemi f \fatsemi assocl_+
\end{array}$$

The "phony" multiplication that is not a functor. The definition for the product of 1d types used above is:

$$\boxed{\tau_1}\boxed{\tau_2} \boxtimes \boxed{\tau_3}\boxed{\tau_4} \triangleq \boxed{(\tau_1 * \tau_3) + (\tau_2 * \tau_4)}\boxed{(\tau_1 * \tau_4) + (\tau_2 * \tau_3)}$$

That definition is "obvious" in some sense as it matches the usual understanding of types as modeling arithmetic identities. Using this definition, it is possible to lift all the 0d combinators involving products *except* the functor:

$$(\otimes) : (\mathbb{T}_1 \Leftrightarrow \mathbb{T}_2) \to (\mathbb{T}_3 \Leftrightarrow \mathbb{T}_4) \to ((\mathbb{T}_1 \boxtimes \mathbb{T}_3) \Leftrightarrow (\mathbb{T}_2 \boxtimes \mathbb{T}_4))$$

After a few failed attempts, we suspected that this definition of multiplication is not functorial which would mean that the **Int** construction only provides a limited notion of higher-order functions at the cost of losing the multiplicative structure at higher-levels. This observation is less well-known that it should be. Further investigation reveals that this observation is intimately related to a well-known problem in algebraic topology and homotopy theory that was identified thirty years ago as the "phony" multiplication [**?**] in a special class categories related to ours. This problem was recently solved [**?**] using a technique whose fundamental ingredients are to add more dimensions and then take homotopy colimits. We exploit this solution in the remainder of the paper.

```
Add eta/epsilon and trace to Int
category

Explain the definitions in this
section much better...
```

6. Conclusion

Talk about trace and recursive types

talk about h.o. functions, negative types, int construction, ring completion paper

canonicity for 2d type theory; licata harper

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.

[3] See Krishnaswami's [**?**] excellent blog post implementing this construction in OCaml.