A Computational Reconstruction of Homotopy Type Theory for Finite Types

Abstract

Homotopy type theory (HoTT) relates some aspects of topology, algebra, 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

Homotopy type theory (HoTT) [The Univalent Foundations Program 2013] has a convoluted treatment of functions. It starts with a class of arbitrary functions, singles out a smaller class of "equivalences" via extensional methods, and then asserts via the *univalence* **axiom** that the class of functions just singled out is equivalent to paths. Why not start with functions that are, by construction, equivalences?

The idea that computation should be based on "equivalences" is an old one and is motivated by physical considerations. Because physics requires various conservation principles (including conservation of information) and because computation is fundamentally a physical process, every computation is fundamentally an equivalence that preserves information. This idea fits well with the HoTT philosophy that emphasizes equalities, isomorphisms, equivalences, and their computational content.

In more detail, 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] and more recently in the context of type isomorphisms [James and Sabry 2012a].

This paper explores the basic ingredients of HoTT from the perspective that computation is all about type isomorphisms. Because the issues involved are quite subtle, the paper is an executable `Agda 2.4.0` file with the global `without-K` option enabled. The main body of the paper reconstructs the main features of HoTT for the limited universe of finite types consisting of the empty type, the unit type, and sums and products of types. Sec. 5 outlines directions for extending the result to richer types.

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[1]. 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. We encode this type in Agda as follows,

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

where we make the evidence explicit. In Agda, one may write proofs of such propositions as shown in the two examples below:

```
i0 : 3 ≡ 3
i0 = refl 3
```

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

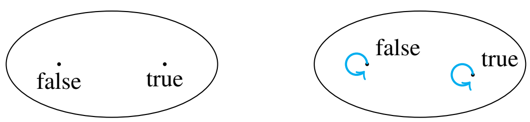
More generally, given two values m and n of type $\mathbb{N}$, it is possible to construct an element refl k of the type $m \equiv n$ if and only if m, n, and k are all "equal." As shown in example i1, this notion of *propositional equality* is not just syntactic equality but generalizes to *definitional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

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

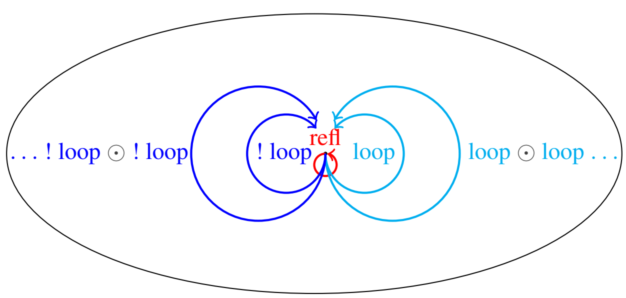
[1] but the converse is not part of the principle. This is frequently misunderstood.

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. [Surely proof theory predates this by decades? Lukasiewicz and Gentzen? —JC] 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 (2-paths), or homotopies in topological language. To be explicit, we will often refer to types as *spaces* which consist of *points*, paths, 2-paths, etc. and write $\equiv_A$ for the type of paths in space A.

[We know that once we have polymorphism, we have no interpretations in Set anymore – should perhaps put more warnings in the next paragraph? —JC] As a simple example, we are used to thinking of (simple) types as sets of values. So we typically view the type Bool as the figure on the left. 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

[Do you mean Set or U for the universe? In HoTT the latter is standard, while Set is really a weird Agda-ism —JC] 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.

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 $A \to B$ that does not have an inverse of type $B \to A$ 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:

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

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[2] 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].[3]

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.

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 computational 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 now 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
    •    : ⟦ |_| ⟧
```

Pointed spaces are often necessary in homotopy theory as various important properties of spaces depend on the chosen basepoint. In our setting, pointed spaces allow us to re-express the Π-combinators in a way that unifies their status as isomorphisms be-

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

[3] 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.[but don't we need trace for the Int construction? —JC]

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

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

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

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

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

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

Figure 2. Operational Semantics

tween *types* and as paths between *points* 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, note that the Π-combinator $swap_+ : \tau_1 + \tau_2 \leftrightarrow \tau_2 + \tau_1$ requires two clauses in the interpreter:

$$
\begin{array}{rcl}
swap_+ \triangleright (\mathsf{inl}\ v) & = & \mathsf{inr}\ v \\
swap_+ \triangleright (\mathsf{inr}\ v) & = & \mathsf{inl}\ v
\end{array}
$$

These two clauses give rise to two path constructors $\mathsf{swap1}_+$ and $\mathsf{swap2}_+$. When viewed as maps between unpointed spaces, both constructors map from PLUS t_1 t_2 to PLUS t_2 t_1. When, however, viewed as maps between points spaces, each constructor specifies in addition its action on the point in a way that mirrors the semantic evaluation rule. The situation is the same for all other Π-constructors. [The operational semantics have 24 rules, while the groupoid model has 26. This is because of the 2 rules with *absurd* in them. How shall they be explained? —JC]

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↔ in the figure). Among the new constructors we have ⊚ 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 [point to the proof in some accompanying full Agda code? —JC]. 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

ZERO• : {absurd : ⟦ ZERO ⟧} → U•
ZERO• {absurd} = •[ ZERO , absurd ]

ONE• : U•
ONE• = •[ ONE , tt ]

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.

[Shouldn't we also show that Bool contains exactly 2 things, and that TRUE and FALSE are "different" ? —JC] [The other thing is, whereas not used to be a path between Bool and Bool, we no longer have that. Shouldn't we show that, somehow, BOOL and

```
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

BOOL.F 'union' BOOL.T are somehow "equivalent"? And there there is a coherent notpath built the same way? Especially since I am sure it is quite easy to build incoherent sets of paths! —JC]

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.

4.1 Examples

Given that all paths are between pointed spaces, i.e., are between particular values, it might appear that all paths between the same pointed spaces are extensionally equivalent. Consider first the following simple examples, which are all paths 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
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 steps along the way. Thinking extensionally, the paths are equivalent. But they are also equivalent if we look at their internal structure using a few simple groupoid identities. In particular, paths p_2 and p_4 sequentially compose the boolean negation with the trivial path, and hence by the groupoid laws are equivalent to p_1. Similarly, the first two steps in path p_3 sequentially compose a combinator with its inverse which is equivalent to the trivial path by the groupoid laws. For path p_5, the groupoid laws are not sufficient to prove its equivalence with any of the other paths but one can argue, as shown below, that it is also equivalent to the others.

Ultimately, the question of whether path p_5 should be considered equivalent to the others should be based on whether there is a "smooth deformation" between the paths. This question can be addressed from a purely categorical approach thanks to the various *coherence theorems* connecting the categorical wiring diagrams to special cases of homotopies called isotopies. (See Selinger's paper [2011] for an excellent survey and the papers by Joyal and Street [1988; 1991] for the original development.) We will not pursue this in detail except for a short discussion in the next section.

But first, we address the important question of whether all paths from a given pointed space to another should be considered equivalent. We answer this question negatively using the following two examples:

```
BOOL² : U
BOOL² = TIMES BOOL BOOL

NOT•T2 CNOT•TT :
  •[ BOOL² , (TRUE , TRUE) ] ↔ •[ BOOL² , (TRUE , FALSE) ]

NOT•T2 = id↔ ⊗ NOT•T

CNOT•TT =
  dist1 ⊚
  ((id↔ ⊗ NOT•T) ⊕1 (id↔ {v = (tt , TRUE)})) ⊚
  factor1
```

The path NOT•T2 just negates the second component of the pair. The path CNOT•TT is the conditional-not reversible gate which only negates the second component of the pair if the first component is TRUE. Although the two paths have the same endpoints, they should not be considered equivalent. The simple reason is that the paths can be given different more general types, i.e., they connect different families of endpoints:

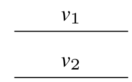
```
NOT•T2' : ∀ {v} →
  •[ BOOL² , (v , TRUE) ] ↔ •[ BOOL² , (v , FALSE) ]
NOT•T2' = id↔ ⊗ NOT•T

CNOT•TT' : ∀ {v} →
  •[ BOOL² , (inj₁ v , TRUE) ] ↔ •[ BOOL² , (inj₁ v , FALSE) ]
CNOT•TT' =
  dist1 ⊚
  ((id↔ ⊗ NOT•T) ⊕1 (id↔ {v = (tt , TRUE)})) ⊚
  factor1
```

Indeed it is possible to use NOT•T2' with the value v bound to FALSE but this is not possible for the other path. We should therefore be careful not to introduce 2-paths between arbitrary paths just because they agree on some endpoints.

4.2 Isotopies

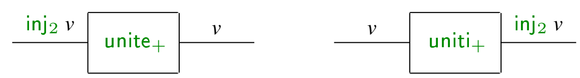
Returning to idea of "smooth deformations" of paths, we first introduce a graphical notation for paths which is the "space" in which the deformations happen. The conventional presentation of wiring diagrams is for unpointed spaces. We adapt it for pointed spaces. First we show how to represent each possible pointed space as a collection of "wires" and then we show how each combinator "shuffles" or "transforms" the wires:

- It is not possible to produce a pointed space •[ZERO , v] for any v.
- The pointed space •[ONE , tt] is invisible in the graphical notation.
- The pointed space •[TIMES t_1 t_2 , (v_1 , v_2)] is represented using two parallel wires labeled v_1 and v_2:



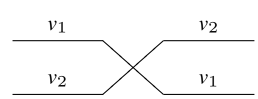
  Note that if one of the types is ONE, the corresponding wire disappears. If both the wires are ONE, they both disappear.
- The pointed space •[PLUS t_1 t_2 , inj₁ v_1] is represented by a wire labeled with inj₁ v_1. The pointed space •[PLUS t_1 t_2 , inj₂ v_2] is similarly represented by a wire labeled with inj₂ v_2.

- Knowing how points are represented, we now show how various combinators act on the wires. The combinator id↔ is invisible. The combinator ⊚ connects the outgoing wires of one diagram to the input wires of the other. The associativity of ⊚ is implicit in the graphical notation.
- The combinators unite₊ and uniti₊ are represented as follows:

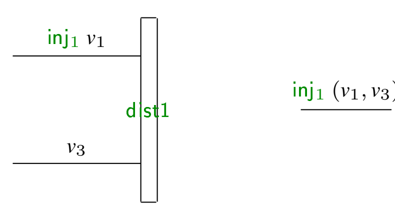


- All other combinators that just re-label a value are similarly represented as one box with one incoming wire labeled by the input value and one outgoing wires labeled by the resulting value.
- The combinators that operate on TIMES types are a bit more involved as shown below. First, although the unit value tt is invisible in the graphical notation, the combinators unite⋆ and uniti⋆ are still represented as boxes as shown below:

  The combinator swap⋆ is represented by crisscrossing wires:

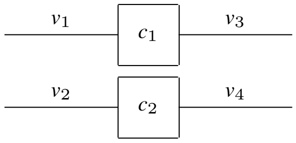


  As discussed below, it is possible to consider a 3d variation which makes explicit which of the wires is on top and which is on bottom. The combinators assocl⋆ and assocr⋆ are invisible in the graphical notation as associativity of parallel wires is implicit. In other words, three parallel wires could be seen as representing $((v_1 , v_2) , v_3)$ or $(v_1 , (v_2 , v_3))$.
- The combinators dist1, dist2, factor1, and factor2 have the following representation:

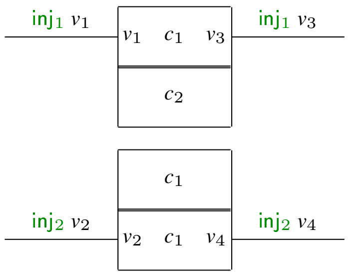


[fix this diagram and add the other 3 diagrams —AS]

- The composite combinator $c_1 \otimes c_2$ is the parallel composition shown below:

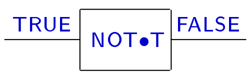


- The combinators $c_1 \oplus 1\ c_2$ and $c_1 \oplus 2\ c_2$ are represented as follows:

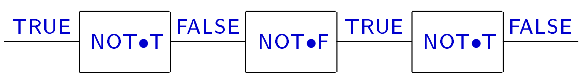


- Finally, when a box c is sequentially composed with its mirror image $!\ c$ (in either order), both boxes disappear.

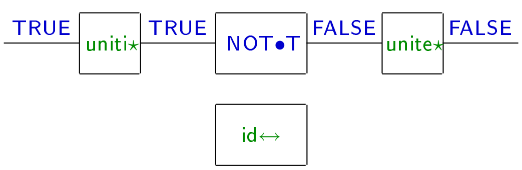
Let us draw the five paths p_1 to p_5 introduced in the previous section. Since id⟷ is invisible, the three paths p_1, p_2, and p_4, are all represented as follows:



Path p_3 would be represented as:



but then we notice that any two of the adjacent boxes are mirror images and erase them to produce the same wiring diagram as the previous three paths. For p_5, we have the following representation:



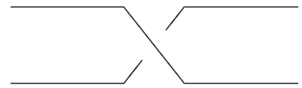
where the occurrences of swap⋆ have disappeared since one of the wires is invisible. The occurrences of id⟷ that acts on the invisible wire does *not*, however, disappear.

The graphical notation is justified by various *coherence theorems*. We quote one of these basic theorems (originally due to Joyal and Street).

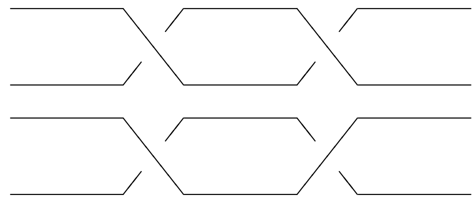
Theorem 1. *A well-formed equation between morphisms in the language of monoidal categories follows from the axioms of monoidal categories if and only if it holds, up to planar isotopy, in the graphical language.*

Translating to our setting, the theorem says the following. In the special case of diagrams involving just one monoid (say ZERO and PLUS types only) and no uses of swapping combinators, the two combinators represented by the diagrams are equivalent if the diagrams can be transformed to each other by moving wires and boxes around without crossing, cutting, or gluing any wires and without detaching them from the plane. Using similar theorems, it is possible, under certain assumptions, to prove that path p_5 is equivalent to the other paths.

Looking at the various coherence theorems for special cases of monoidal categories, we note an interesting subtlety that should be further investigated in detail: in our presentation of Π, we have assumed that swap⋆ is its self-inverse. But thinking of the categorical wiring diagrams more geometrically suggests that two wires crossing each other requires a third dimension. In other words, a possible diagram for swap⋆ would be:



where it is explicit which path is crossing over which during the swap operation. Technically we have moved from a symmetric monoidal category to a *braided* one. From this idea, it follows that a sequence of two swaps might represent one of the following two diagrams:



In 3 dimensions, the first diagram creates a "knot" but the second reduces to trivial identity paths. In the context of symmetric monoidal categories, i.e., in the context of our original presentation of Π, the diagrams are forced to be 2 dimensional: the distinction between them vanishes and they become equivalent.

4.3 Π Lifted and with Groupoid Axioms

To summarize, there is a spectrum of possibilities to be explored for when paths should be considered equivalent. The minimum requirement is that paths that can be related using the groupoid laws should be considered equivalent and hence should be related by a 2-path. In the sequel, we will adopt this conservative approach and leave further investigations to future work.

Formally, we lift the entire Π language to compute with paths instead of with points. The lifted version of Π will have all the combinators of Fig. 3 as well as additional combinators witnessing the groupoid laws. The groupoid combinators will allow us to relate paths like p_1 and p_2 and the combinators from Fig. 3 will allow us to compute with sums and products of paths up to the commutative semiring isomorphisms. What is pleasant about this design is that 2-paths inherit a similar structure to 1-paths, and hence the entire scheme can be repeated over and over lifting Π to higher and higher levels to capture the concept of weak ∞-groupoids.

We now present the detailed construction of the next level of Π.

```
data 1U : Set where
    1ZERO   : 1U
    1ONE    : 1U
    1PLUS   : 1U → 1U → 1U
    1TIMES  : 1U → 1U → 1U
    PATH    : U• → U• → 1U

1⟦_⟧ : 1U → Set
1⟦ 1ZERO ⟧          = ⊥
1⟦ 1ONE ⟧           = ⊤
1⟦ 1PLUS t₁ t₂ ⟧    = 1⟦ t₁ ⟧ ⊎ 1⟦ t₂ ⟧
1⟦ 1TIMES t₁ t₂ ⟧   = 1⟦ t₁ ⟧ × 1⟦ t₂ ⟧
1⟦ PATH t₁ t₂ ⟧     = Path t₁ t₂
```

The new universe 1U is a universe whose spaces are path spaces. The space 1ZERO is the empty set of paths. The space 1ONE is the space of paths containing one path that is the identity for path products. Sums and products of paths are representing using disjoint union and cartesian products. In addition, all paths from Fig. 3 are reified as values in 1U.

As before, we define pointed spaces (now of paths instead of points) and introduce Π combinators on these pointed path spaces. In addition to the commutative semiring combinators, there are also combinators that witness the groupoid equivalences. (See Fig. 5.)

```
record 1U• : Set where
  constructor 1•[_,_]
  field
    1|_| : 1U
    1• : 1⟦ 1|_| ⟧
```

To verify that the universe U• with ↔ as 1-paths and ⇔ as 2-paths is indeed a groupoid, we have developed a small library inspired by Thorsten Altenkirch's definition of groupoids (see `http://github.com/txa/OmegaCats`) and copumpkin's definition of category (see `http://github.com/copumpkin/categories`). The proof is shown below:

```
G : 1Groupoid
G = record
  { set = U•
  ; _↝_ = _↔_
  ; _≈_ = λ {t₁} {t₂} c₀ c₁ →
      1•[ PATH t₁ t₂ , path c₀ ] ⇔ 1•[ PATH t₁ t₂ , path c₁ ]
  ; id = id↔
  ; _∘_ = λ c₀ c₁ → c₁ ⊚ c₀
  ; _⁻¹ = !
  ; lneutr = λ _ → ridl
  ; rneutr = λ _ → lidl
  ; assoc = λ _ _ _ → assocl
  ; equiv = record { refl = id⇔
      ; sym = λ c → 1! c
      ; trans = λ c₀ c₁ → c₀ ⊚ c₁ }
  ; linv = λ {t₁} {t₂} c → linv c
  ; rinv = λ {t₁} {t₂} c → rinv c
  ; ∘-resp-≈ = λ f⟷h g⟷i → resp⊚ g⟷i f⟷h
  }
```

The proof refers to two simple functions linv and rinv with the following types:

```
linv : {t₁ t₂ : U•} → (c : t₁ ↔ t₂) →
  1•[ PATH t₁ t₁ , path (c ⊚ ! c) ] ⇔ 1•[ PATH t₁ t₁ , path id↔ ]
rinv : {t₁ t₂ : U•} → (c : t₁ ↔ t₂) →
  1•[ PATH t₂ t₂ , path (! c ⊚ c) ] ⇔ 1•[ PATH t₂ t₂ , path id↔ ]
```

[TODO: show a few cases?]

5. The Int Construction

[TODO: transition]

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 [1996] and the closely related $\mathcal{G}$ construction of linear logic [Abramsky 1996] 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 $\boxed{\tau_1 \mid \tau_2}$ 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.

5.1 Conventionl Construction on Unpointed Types

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

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

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. 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:

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

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 [Di Cosmo 2005], 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:

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

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.[4] There are however a few interesting cases whose appreciation is essential for the remainder of the paper.

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

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

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.***

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

[4] See Krishnaswami's [2012] excellent blog post implementing this construction in OCaml.

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

-- Groupoid combinators
lidl : ∀ {t₁ t₂} → {c : t₁ ⟷ t₂} →
   1•[ PATH t₁ t₂ , path (id⟷ ◎ c) ] ⇔ 1•[ PATH t₁ t₂ , path c ]
lidr : ∀ {t₁ t₂} → {c : t₁ ⟷ t₂} →
   1•[ PATH t₁ t₂ , path c ] ⇔ 1•[ PATH t₁ t₂ , path (id⟷ ◎ c) ]
ridl : ∀ {t₁ t₂} → {c : t₁ ⟷ t₂} →
   1•[ PATH t₁ t₂ , path (c ◎ id⟷) ] ⇔ 1•[ PATH t₁ t₂ , path c ]
ridr : ∀ {t₁ t₂} → {c : t₁ ⟷ t₂} →
   1•[ PATH t₁ t₂ , path c ] ⇔ 1•[ PATH t₁ t₂ , path (c ◎ id⟷) ]
assocl : ∀ {t₁ t₂ t₃ t₄} → {c₁ : t₁ ⟷ t₂} {c₂ : t₂ ⟷ t₃} {c₃ : t₃ ⟷ t₄} →
   1•[ PATH t₁ t₄ , path (c₁ ◎ (c₂ ◎ c₃)) ] ⇔ 1•[ PATH t₁ t₄ , path ((c₁ ◎ c₂) ◎ c₃) ]
assocr : ∀ {t₁ t₂ t₃ t₄} → {c₁ : t₁ ⟷ t₂} {c₂ : t₂ ⟷ t₃} {c₃ : t₃ ⟷ t₄} →
   1•[ PATH t₁ t₄ , path ((c₁ ◎ c₂) ◎ c₃) ] ⇔ 1•[ PATH t₁ t₄ , path (c₁ ◎ (c₂ ◎ c₃)) ]
unite₊l : ∀ {t v} →
   1•[ PATH (•[ PLUS ZERO t , inj₂ v ]) (•[ PLUS ZERO t , inj₂ v ]) ,
     path (unite₊ ◎ uniti₊) ] ⇔
   1•[ PATH (•[ PLUS ZERO t , inj₂ v ]) (•[ PLUS ZERO t , inj₂ v ]) , path id⟷ ]
unite₊r : ∀ {t v} →
   1•[ PATH (•[ PLUS ZERO t , inj₂ v ]) (•[ PLUS ZERO t , inj₂ v ]) , path id⟷ ] ⇔
   1•[ PATH (•[ PLUS ZERO t , inj₂ v ]) (•[ PLUS ZERO t , inj₂ v ]) ,
     path (unite₊ ◎ uniti₊) ]
uniti₊l : ∀ {t v} →
   1•[ PATH (•[ t , v ]) (•[ t , v ]) , path (uniti₊ ◎ unite₊) ] ⇔
   1•[ PATH (•[ t , v ]) (•[ t , v ]) , path id⟷ ]
uniti₊r : ∀ {t v} →
   1•[ PATH (•[ t , v ]) (•[ t , v ]) , path id⟷ ] ⇔
   1•[ PATH (•[ t , v ]) (•[ t , v ]) , path (uniti₊ ◎ unite₊) ]
swap1₊l : ∀ {t₁ t₂ v₁} →
   1•[ PATH •[ PLUS t₁ t₂ , inj₁ v₁ ] •[ PLUS t₁ t₂ , inj₁ v₁ ] ,
     path (swap1₊ ◎ ! swap1₊) ] ⇔
   1•[ PATH •[ PLUS t₁ t₂ , inj₁ v₁ ] •[ PLUS t₁ t₂ , inj₁ v₁ ] , path id⟷ ]
swap1₊r : ∀ {t₁ t₂ v₁} →
   1•[ PATH •[ PLUS t₁ t₂ , inj₁ v₁ ] •[ PLUS t₁ t₂ , inj₁ v₁ ] , path id⟷ ] ⇔
   1•[ PATH •[ PLUS t₁ t₂ , inj₁ v₁ ] •[ PLUS t₁ t₂ , inj₁ v₁ ] ,
     path (swap1₊ ◎ ! swap1₊) ]
swap2₊l : ∀ {t₁ t₂ v₂} →
   1•[ PATH •[ PLUS t₁ t₂ , inj₂ v₂ ] •[ PLUS t₁ t₂ , inj₂ v₂ ] ,
     path (swap2₊ ◎ ! swap2₊) ] ⇔
   1•[ PATH •[ PLUS t₁ t₂ , inj₂ v₂ ] •[ PLUS t₁ t₂ , inj₂ v₂ ] , path id⟷ ]
swap2₊r : ∀ {t₁ t₂ v₂} →
   1•[ PATH •[ PLUS t₁ t₂ , inj₂ v₂ ] •[ PLUS t₁ t₂ , inj₂ v₂ ] , path id⟷ ] ⇔
   1•[ PATH •[ PLUS t₁ t₂ , inj₂ v₂ ] •[ PLUS t₁ t₂ , inj₂ v₂ ] ,
     path (swap2₊ ◎ ! swap2₊) ]
assocl1₊l : ∀ {t₁ t₂ t₃ v₁} →
   1•[ PATH •[ PLUS t₁ (PLUS t₂ t₃) , inj₁ v₁ ] •[ PLUS t₁ (PLUS t₂ t₃) , inj₁ v₁ ] ,
     path (assocl1₊ ◎ ! assocl1₊) ] ⇔
   1•[ PATH •[ PLUS t₁ (PLUS t₂ t₃) , inj₁ v₁ ] •[ PLUS t₁ (PLUS t₂ t₃) , inj₁ v₁ ] ,
     path id⟷ ]
assocl1₊r : ∀ {t₁ t₂ t₃ v₁} →
   1•[ PATH •[ PLUS t₁ (PLUS t₂ t₃) , inj₁ v₁ ] •[ PLUS t₁ (PLUS t₂ t₃) , inj₁ v₁ ] ,
     path id⟷ ] ⇔
   1•[ PATH •[ PLUS t₁ (PLUS t₂ t₃) , inj₁ v₁ ] •[ PLUS t₁ (PLUS t₂ t₃) , inj₁ v₁ ] ,
   path (assocl1₊ ◎ ! assocl1₊) ]
assocl2₊l : ∀ {t₁ t₂ t₃ v₂} →
   1•[ PATH •[ PLUS t₁ (PLUS t₂ t₃) , inj₂ (inj₁ v₂) ]
     •[ PLUS t₁ (PLUS t₂ t₃) , inj₂ (inj₁ v₂) ] , path (assocl2₊ ◎ ! assocl2₊) ] ⇔
   1•[ PATH •[ PLUS t₁ (PLUS t₂ t₃) , inj₂ (inj₁ v₂) ]
     •[ PLUS t₁ (PLUS t₂ t₃) , inj₂ (inj₁ v₂) ] , path id⟷ ]
assocl2₊r : ∀ {t₁ t₂ t₃ v₂} →
   1•[ PATH •[ PLUS t₁ (PLUS t₂ t₃) , inj₂ (inj₁ v₂) ]
     •[ PLUS t₁ (PLUS t₂ t₃) , inj₂ (inj₁ v₂) ] , path id⟷ ] ⇔
   1•[ PATH •[ PLUS t₁ (PLUS t₂ t₃) , inj₂ (inj₁ v₂) ]
     •[ PLUS t₁ (PLUS t₂ t₃) , inj₂ (inj₁ v₂) ] ,
   path (assocl2₊ ◎ ! assocl2₊) ]
assocl3₊l : ∀ {t₁ t₂ t₃ v₃} →
   1•[ PATH •[ PLUS t₁ (PLUS t₂ t₃) , inj₂ (inj₂ v₃) ]
     •[ PLUS t₁ (PLUS t₂ t₃) , inj₂ (inj₂ v₃) ] ,
   path (assocl3₊ ◎ ! assocl3₊) ] ⇔
   1•[ PATH •[ PLUS t₁ (PLUS t₂ t₃) , inj₂ (inj₂ v₃) ]
     •[ PLUS t₁ (PLUS t₂ t₃) , inj₂ (inj₂ v₃) ] , path id⟷ ]
assocl3₊r : ∀ {t₁ t₂ t₃ v₃} →
   1•[ PATH •[ PLUS t₁ (PLUS t₂ t₃) , inj₂ (inj₂ v₃) ]
     •[ PLUS t₁ (PLUS t₂ t₃) , inj₂ (inj₂ v₃) ] , path id⟷ ] ⇔
   1•[ PATH •[ PLUS t₁ (PLUS t₂ t₃) , inj₂ (inj₂ v₃) ]
     •[ PLUS t₁ (PLUS t₂ t₃) , inj₂ (inj₂ v₃) ] ,
   path (assocl3₊ ◎ ! assocl3₊) ]
resp◎ : ∀ {t₁ t₂ t₃} → {c₁ : t₁ ⟷ t₂} {c₂ : t₂ ⟷ t₃} {c₃ : t₁ ⟷ t₂} {c₄ : t₂ ⟷ t₃} →
   (1•[ PATH t₁ t₂ , path c₁ ] ⇔ 1•[ PATH t₁ t₂ , path c₃ ]) →
   (1•[ PATH t₂ t₃ , path c₂ ] ⇔ 1•[ PATH t₂ t₃ , path c₄ ]) →
     1•[ PATH t₁ t₃ , path (c₁ ◎ c₂) ] ⇔ 1•[ PATH t₁ t₃ , path (c₃ ◎ c₄) ]
```

Figure 5. Inductive definition of 2-paths

$$
\begin{array}{rcl}
\mathit{flip} & : & (\mathbb{T}_1 \Leftrightarrow \mathbb{T}_2) \rightarrow (\boxminus\mathbb{T}_2 \Leftrightarrow \boxminus\mathbb{T}_1) \\
\mathit{flip}\ f & = & \mathit{swap}_+ \fatsemi f \fatsemi \mathit{swap}_+ \\
\\
\mathit{curry} & : & ((\mathbb{T}_1 \boxplus \mathbb{T}_2) \Leftrightarrow \mathbb{T}_3) \rightarrow (\mathbb{T}_1 \Leftrightarrow (\mathbb{T}_2 \multimap \mathbb{T}_3)) \\
\mathit{curry}\ f & = & \mathit{assocl}_+ \fatsemi f \fatsemi \mathit{assocr}_+ \\
\\
\mathit{uncurry} & : & (\mathbb{T}_1 \Leftrightarrow (\mathbb{T}_2 \multimap \mathbb{T}_3)) \rightarrow ((\mathbb{T}_1 \boxplus \mathbb{T}_2) \Leftrightarrow \mathbb{T}_3) \\
\mathit{uncurry}\ f & = & \mathit{assocr}_+ \fatsemi f \fatsemi \mathit{assocl}_+
\end{array}
$$

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:

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

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 [Thomason 1980] in a special class categories related to ours. This problem was recently solved [Baas et al. 2012] using a technique whose fundamental ingredients are to add more dimensions and then take homotopy colimits. It remains to investigate whether this idea can be integrated with our development to get higher-order functions while retaining the multiplicative structure.

5.2 Pointed Int Construction

Since our development is done using pointed spaces, we adapt the conventional construction as follows.

```
-- Types are of the form t - t'

record U- : Set where
  constructor _-_
  field
    pos : U
    neg : U

open U-

ZERO- ONE- : U-
ZERO- = ZERO - ZERO
ONE-  = ONE  - ZERO

PLUS- : U- → U- → U-
PLUS- (pos1 - neg1) (pos2 - neg2) =
  PLUS pos1 pos2 - PLUS neg1 neg2

TIMES- : U- → U- → U-
TIMES- (pos1 - neg1) (pos2 - neg2) =
  PLUS (TIMES pos1 pos2) (TIMES neg1 neg2) -
  PLUS (TIMES pos1 neg2) (TIMES neg1 pos2)

FLIP- : U- → U-
FLIP- (pos - neg) = neg - pos

LOLLI- : U- → U- → U-
LOLLI- (pos1 - neg1) (pos2 - neg2) =
  PLUS neg1 pos2 - PLUS pos1 neg2

-- Pointed types are of the form (t , pt) - (t' , pt')

data U-• : Set where
  both• : (t : U-) → ⟦ pos t ⟧ → ⟦ neg t ⟧ → U-•

ZERO-• : {absurd : ⟦ ZERO ⟧} → U-•
ZERO-• {absurd} = both• ZERO- absurd absurd

ONE-• : {absurd : ⟦ ZERO ⟧} → U-•
ONE-• {absurd} = both• ONE- tt absurd

FLIP-• : U-• → U-•
FLIP-• (both• t p n) = both• (FLIP- t) n p

PLUS-11• : U-• → U-• → U-•
PLUS-11• (both• t1 p1 n1) (both• t2 p2 n2) =
  both• (PLUS- t1 t2) (inj1 p1) (inj1 n1)

PLUS-12• : U-• → U-• → U-•
PLUS-12• (both• t1 p1 n1) (both• t2 p2 n2) =
  both• (PLUS- t1 t2) (inj1 p1) (inj2 n2)

PLUS-21• : U-• → U-• → U-•
PLUS-21• (both• t1 p1 n1) (both• t2 p2 n2) =
  both• (PLUS- t1 t2) (inj2 p2) (inj1 n1)

PLUS-22• : U-• → U-• → U-•
PLUS-22• (both• t1 p1 n1) (both• t2 p2 n2) =
  both• (PLUS- t1 t2) (inj2 p2) (inj2 n2)

LOLLI-11• : U-• → U-• → U-•
LOLLI-11• (both• t1 p1 n1) (both• t2 p2 n2) =
  both• (LOLLI- t1 t2) (inj1 n1) (inj1 p1)

LOLLI-12• : U-• → U-• → U-•
LOLLI-12• (both• t1 p1 n1) (both• t2 p2 n2) =
  both• (LOLLI- t1 t2) (inj1 n1) (inj2 n2)

LOLLI-21• : U-• → U-• → U-•
LOLLI-21• (both• t1 p1 n1) (both• t2 p2 n2) =
  both• (LOLLI- t1 t2) (inj2 p2) (inj1 p1)

LOLLI-22• : U-• → U-• → U-•
LOLLI-22• (both• t1 p1 n1) (both• t2 p2 n2) =
  both• (LOLLI- t1 t2) (inj2 p2) (inj2 n2)

-- Combinators are between
-- (t , pt) - (t' , pt') and
-- (u , pu) - (u' , pu')
-- this re-arranges to a level 0 combinator
-- (t , pt) + (u' , pu') <-> (t' , pt') + (u , pu)
-- when combining the pointed sets in the last line
-- we have two options on the source: left pt, right pu'
-- and two options on the target: left pt', and right pu

data _⇄_ : U-• → U-• → Set where
  NN : ∀ {P1 N1 P2 N2 p1 n1 p2 n2} →
    •[ PLUS P1 N2 , inj2 n2 ] ↔ •[ PLUS N1 P2 , inj1 n1 ] →
    (both• (P1 - N1) p1 n1) ⇄ (both• (P2 - N2) p2 n2)
  NP : ∀ {P1 N1 P2 N2 p1 n1 p2 n2} →
    •[ PLUS P1 N2 , inj2 n2 ] ↔ •[ PLUS N1 P2 , inj2 p2 ] →
    (both• (P1 - N1) p1 n1) ⇄ (both• (P2 - N2) p2 n2)
  PN : ∀ {P1 N1 P2 N2 p1 n1 p2 n2} →
    •[ PLUS P1 N2 , inj1 p1 ] ↔ •[ PLUS N1 P2 , inj1 n1 ] →
    (both• (P1 - N1) p1 n1) ⇄ (both• (P2 - N2) p2 n2)
  PP : ∀ {P1 N1 P2 N2 p1 n1 p2 n2} →
```

```
•[ PLUS P₁ N₂ , inj₁ p₁ ] ↔ •[ PLUS N₁ P₂ , inj₂ p₂ ] →
(both• (P₁ - N₁) p₁ n₁) ⇄ (both• (P₂ - N₂) p₂ n₂)

- there are two fibers for id in the int category

id⇄NN : {t : U-•} → t ⇄ t
id⇄NN {both• t p n} = NN swap2₊

id⇄PP : {t : U-•} → t ⇄ t
id⇄PP {both• t p n} = PP swap1₊

identl₊ : {t : U-•} → PLUS-22• ZERO-• t ⇄ t
identl₊ = {!!} - PP (assocr2₊ ⊚ (id↔ ⊕2 swap1₊) ⊚ assocl2₊)

- define trace and composition

flip⇄ : {t₁ t₂ : U-•} → (t₁ ⇄ t₂) → (FLIP-• t₂ ⇄ FLIP-• t₁)
flip⇄ (NN c) = PP (swap1₊ ⊚ c ⊚ swap1₊)
flip⇄ (NP c) = PN (swap1₊ ⊚ c ⊚ swap2₊)
flip⇄ (PN c) = NP (swap2₊ ⊚ c ⊚ swap1₊)
flip⇄ (PP c) = NN (swap2₊ ⊚ c ⊚ swap2₊)

curry1111⇄ : {t₁ t₂ t₃ : U-•} → (PLUS-11• t₁ t₂ ⇄ t₃) → (t₁ ⇄ LOLLI-11• t₂ t₃)
curry1111⇄ {t₁} {t₂} {t₃} f = {!!}

- define small example:

- given in plain Pi level 0
- c1 c2 : t1 + t4 <-> t2 + t3
- given in plain Pi level 1
- alpha : c1 <-> c2

- in the int category
- c1 and c2 are maps between (t1 - t2) and (t3 - t4)
- 'name c1' and 'name c2' are maps between (0 - 0) and ((t1 - t2) o (t3 - t4))
- c1 and c2 themselves become elements of ((t1 - t2) -o (t3 - t4))
- i.e., they become values
- alpha that used to be a 2path, i.e., a path between paths, is now
- a path between the values c1 and c2 in the -o type
```

5.3 Example

6. Conclusion

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

Add eta/epsilon and trace to Int category

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

triangle; pentagon rules; eckmann-hilton

References

S. Abramsky. Retracing some paths in process algebra. In *CONCUR '96: Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin Heidelberg, 1996.

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

N. Baas, B. Dundas, B. Richter, and J. Rognes. Ring completion of rig categories. *Journal für die reine und angewandte Mathematik (Crelles Journal)*, 2013(674):43–80, Mar. 2012.

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.

R. Di Cosmo. A short survey of isomorphisms of types. *Mathematical. Structures in Comp. Sci.*, 15(5):825–838, Oct. 2005.

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.

A. Joyal and R. Street. Planar diagrams and tensor algebra. `http://maths.mq.edu.au/~street/Publications.html`, Sep. 1988.

A. Joyal and R. Street. The geometry of tensor calculus I. *Advances in Mathematics*, 88(1):55–112, 1991.

A. Joyal, R. Street, and D. Verity. Traced monoidal categories. In *Mathematical Proceedings of the Cambridge Philosophical Society*. Cambridge Univ Press, 1996.

N. Krishnaswami. The geometry of interaction, as an OCaml program. `http://semantic-domain.blogspot.com/2012/11/in-this-post-ill-show-how-to-turn.html`, 2012.

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

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

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

P. Selinger. A survey of graphical languages for monoidal categories. In B. Coecke, editor, *New Structures for Physics*, volume 813 of *Lecture Notes in Physics*, pages 289–355. Springer Berlin / Heidelberg, 2011.

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

R. Thomason. Beware the phony multiplication on Quillen's $\mathcal{A}^{-1}\mathcal{A}$. *Proc. Amer. Math. Soc.*, 80(4):569–573, 1980.

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