Negative Types

Abstract

. . .

1. Introduction

The **Int** construction or the $\mathcal{G}$ construction are neat. As Neel K. explains [Krishnaswami 2012], given first-order types and feedback you get higher-order functions. But if you do the construction on the additive structure, you lose the multiplicative structure. It turns out that this is related to a deep open problem in algebraic topology and homotopy theory that was recently solved [Baas et al. 2012]. We "translate" that solution to a computational type-theoretic world. This has evident connections to homotopy (type) theory that remain to be investigated in more depth.

Make sure we introduce the abbreviation HoTT in the introduction [The Univalent Foundations Program 2013].

2. The Int Construction

We may or may not want to explain the construction using Haskell. In case we do, the most relevant code is below. The key insight it to enrich types to forms of the shape $(t_1 - t_2)$ which represent a sum type of t_1 flowing in the "normal" direction (from producers to consumers) and t_2 flowing backwards (representing *demands* for values). This is expressive enough to represent functions which are viewed as expressions that convert a demand for an argument to the production of a result. The problem is that the obvious definition of multiplication is not functorial. This turns out to be intimately related to a well-known open problem in algebraic topology that goes back at least thirty years [Thomason 1980].

```
class GT p where
  type Pos p       :: *  -- a type has a positive component
  type Neg p       :: *  -- and a negative component
  type ZeroG       :: *  -- we want all the usual type
  type OneG        :: *  -- constructors including 0, 1,
  type PlusG p q   :: *  -- sums, and
  type ProdG p q   :: *  -- products
  type DualG p     :: *  -- as a bonus we get negation and
  type LolliG p q  :: *  -- linear functions

-- A definition of the composite types with a
-- positive and negative components

data a :- b = a :- b

instance GT (ap :- am) where
  type Pos (ap :- am) = ap
  type Neg (ap :- am) = am
  type ZeroG = Void :- Void
  type OneG = () :- Void
  type PlusG (ap :- am) (bp :- bm) =
    (Either ap bp) :- (Either am bm)
  type TimesG (ap :- am) (bp :- bm) =
    -- the "obvious" but broken multiplication
    (Either (ap,bp) (am,bm)) :- (Either (am,bp) (ap,bm))
  type DualG (ap :- am) = am :- ap
  type LolliG (ap :- am) (bp :- bm) =
    (Either am bp) :- (Either ap bm)

-- Functions between composite types with positive
-- and negative components; implemented using
-- resumptions (i.e., feedback)
newtype GM a b =
  GM { rg :: R (Either (Pos a) (Neg b))
               (Either (Neg a) (Pos b)) }

data R i o = R { r  :: i → (o, R i o),
                 rr :: o → (i, R o i) }

plusG :: (a ~ (ap :- am), b ~ (bp :- bm),
          c ~ (cp :- cm), d ~ (dp :- dm)) ⇒
  GM a b → GM c d → GM (PlusG a c) (PlusG b d)
plusG (GM f) (GM g) = -- short definition omitted

timesG :: (a ~ (ap :- am), b ~ (bp :- bm),
           c ~ (cp :- cm), d ~ (dp :- dm)) ⇒
  GM a b → GM c d → GM (TimesG a c) (TimesG b d)
timesG = -- IMPOSSIBLE
```

The main ingredient of the recent solution to this problem can intuitively explained as follows. We regard conventional types as 0-dimensional cubes. By adding composite types consisting of two types, the **Int** construction effectively creates 1-dimensional cubes (i.e., lines). The key to the general solution, and the approach we adopt here, is to generalize types to n-dimensional cubes.

3. Cubes

We first define the syntax and then present a simple semantic model of types which is then refined.

3.1 Negative and Cubical Types

Our types τ include the empty type 0, the unit type 1, conventional sum and product types, as well as *negative* types:

$$\tau \quad ::= \quad 0 \mid 1 \mid \tau_1 + \tau_2 \mid \tau_1 * \tau_2 \mid -\tau$$

We use $\tau_1 - \tau_2$ to abbreviate $\tau_1 + (-\tau_2)$ and more interestingly $\tau_1 \multimap \tau_2$ to abbreviate $(-\tau_1) + \tau_2$. The *dimension* of a type is defined as follows:

$$\begin{array}{rcl}
\mathsf{dim}(\cdot) & :: & \tau \to \mathbb{N} \\
\mathsf{dim}(0) & = & 0 \\
\mathsf{dim}(1) & = & 0 \\
\mathsf{dim}(\tau_1 + \tau_2) & = & \max(\mathsf{dim}(\tau_1), \mathsf{dim}(\tau_2)) \\
\mathsf{dim}(\tau_1 * \tau_2) & = & \mathsf{dim}(\tau_1) + \mathsf{dim}(\tau_2) \\
\mathsf{dim}(-\tau) & = & \max(1, \mathsf{dim}(\tau))
\end{array}$$

$$\boxed{S_1 \mid S_2} \quad \otimes \quad \boxed{(\boxed{S_3 \mid S_4}) \mid (\boxed{S_5 \mid S_6})} \quad =$$

$$\boxed{(\,(\boxed{S_1 \times S_3 \mid S_1 \times S_4}) \mid (\boxed{S_1 \times S_5 \mid S_1 \times S_6})\,) \mid (\,(\boxed{S_2 \times S_3 \mid S_2 \times S_4}) \mid (\boxed{S_2 \times S_5 \mid S_2 \times S_6})\,)}$$

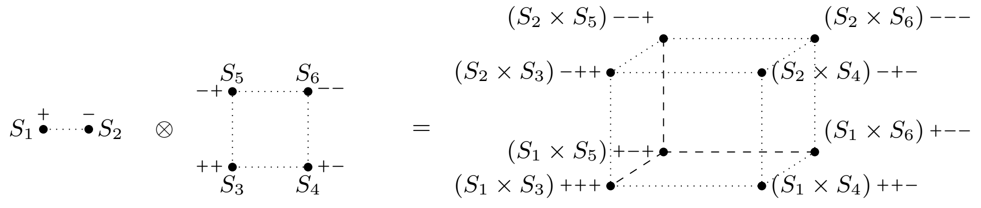


Figure 1. Example of multiplication of two cubical types.

The base types have dimension 0. If negative types are not used, all dimensions remain at 0. If negative types are used but no products of negative types appear anywhere, the dimension is raised to 1. This is the situation with the **Int** or $\mathcal{G}$ construction. Once negative and product types are freely used, the dimension can increase without bounds.

This point is made precise in the following tentative denotation of types (to be refined in the next section) which maps a type of dimension n to an n-dimensional cube. We represent such a cube syntactically as a binary tree of maximum depth n with nodes of the form $\boxed{\mathbb{T}_1 \mid \mathbb{T}_2}$. In such a node, $\mathbb{T}_1$ is the positive subspace and $\mathbb{T}_2$ (shaded in gray) is the negative subspace along the first dimension. Each of these subspaces is itself a cube of a lower dimension. The 0-dimensional cubes are plain sets representing the denotation of conventional first-order types. We use S to denote the denotations of these plain types. A 1-dimensional cube, $\boxed{S_1 \mid S_2}$, intuitively corresponds to the difference $\tau_1 - \tau_2$ of the two types whose denotations are S_1 and S_2 respectively. The type can be visualized as a "line" with polarized endpoints connecting the two points S_1 and S_2. A full 2-dimensional cube, $\boxed{(\boxed{S_1 \mid S_2}) \mid (\boxed{S_3 \mid S_4})}$, intuitively corresponds to the iterated difference of the appropriate types $(\tau_1 - \tau_2) - (\tau_3 - \tau_4)$ where the successive "colors" from the outermost box encode the sign. The type can be visualized as a "square" with polarized corners connecting the two lines corresponding to $(\tau_1 - \tau_2)$ and $(\tau_3 - \tau_4)$. (See Fig. 1 which is further explained after we discuss multiplication below.)

Formally, the denotation of types discussed so far is as follows:

$$\begin{aligned}
[\![0]\!] &= \emptyset \\
[\![1]\!] &= \{\star\} \\
[\![\tau_1 + \tau_2]\!] &= [\![\tau_1]\!] \oplus [\![\tau_2]\!] \\
[\![\tau_1 * \tau_2]\!] &= [\![\tau_1]\!] \otimes [\![\tau_2]\!] \\
[\![-\tau]\!] &= \ominus[\![\tau]\!]
\end{aligned}$$

where:

$$\begin{aligned}
S_1 \oplus S_2 &= S_1 \uplus S_2 \\
S \oplus (\boxed{\mathbb{T}_1 \mid \mathbb{T}_2}) &= \boxed{S \oplus \mathbb{T}_1 \mid \mathbb{T}_2} \\
(\boxed{\mathbb{T}_1 \mid \mathbb{T}_2}) \oplus S &= \boxed{\mathbb{T}_1 \oplus S \mid \mathbb{T}_2} \\
(\boxed{\mathbb{T}_1 \mid \mathbb{T}_2}) \oplus (\boxed{\mathbb{T}_3 \mid \mathbb{T}_4}) &= \boxed{\mathbb{T}_1 \oplus \mathbb{T}_3 \mid \mathbb{T}_2 \oplus \mathbb{T}_4}
\end{aligned}$$

$$\begin{aligned}
S_1 \otimes S_2 &= S_1 \times S_2 \\
S \otimes (\boxed{\mathbb{T}_1 \mid \mathbb{T}_2}) &= \boxed{S \otimes \mathbb{T}_1 \mid S \otimes \mathbb{T}_2} \\
(\boxed{\mathbb{T}_1 \mid \mathbb{T}_2}) \otimes \mathbb{T} &= \boxed{\mathbb{T}_1 \otimes \mathbb{T} \mid \mathbb{T}_2 \otimes \mathbb{T}}
\end{aligned}$$

$$\begin{aligned}
\ominus S &= \boxed{ \mid S} \\
\ominus \boxed{\mathbb{T}_1 \mid \mathbb{T}_2} &= \boxed{\ominus \mathbb{T}_2 \mid \ominus \mathbb{T}_1}
\end{aligned}$$

The type 0 maps to the empty set. The type 1 maps to a singleton set. The sum of 0-dimensional types is the disjoint union as usual. For cubes of higher dimensions, the subspaces are recursively added. Note that the sum of 1-dimensional types reduces to the sum used in the **Int** construction. The definition of negation is natural: it recursively swaps the positive and negative subspaces. The product of 0-dimensional types is the cartesian product of sets. For cubes of higher-dimensions n and m, the result is of dimension $(n + m)$. The example in Fig. 1 illustrates the idea using the product of 1-dimensional cube (i.e., a line) with a 2-dimensional cube (i.e., a square). The result is a 3-dimensional cube as illustrated.

3.2 Higher-Order Functions

In the **Int** construction a function from $T_1 = (t_1 - t_2)$ to $T_2 = (t_3 - t_4)$ is represented as an object of type $-T_1 + T_2$. Expanding the definitions, we get:

$$\begin{aligned}
-T_1 + T_2 &= -(t_1 - t_2) + (t_3 - t_4) \\
&= (t_2 - t_1) + (t_3 - t_4) \\
&= (t_2 + t_3) - (t_1 + t_4)
\end{aligned}$$

The above calculation is consistent with our definitions specialized to 1-dimensional types. Note that the function is represented as an object of the same dimension as its input and output types. The situation generalizes to higher-dimensions. For example, consider a function of type

$$\boxed{\boxed{\boxed{\tau_1}\boxed{\tau_2}}\boxed{\boxed{\tau_3}\boxed{\tau_4}}} \multimap \boxed{\boxed{\boxed{\tau_5}\boxed{\tau_6}}\boxed{\boxed{\tau_7}\boxed{\tau_8}}}$$

This function is represent by an object of dimension 2 as the calculation below shows:

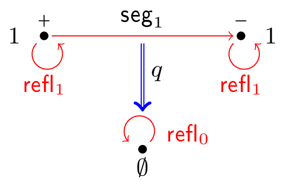
$$\begin{aligned}
& \boxed{\boxed{\boxed{\tau_1}\boxed{\tau_2}}\boxed{\boxed{\tau_3}\boxed{\tau_4}}} \multimap \boxed{\boxed{\boxed{\tau_5}\boxed{\tau_6}}\boxed{\boxed{\tau_7}\boxed{\tau_8}}} \\
= \; & (\ominus \boxed{\boxed{\boxed{\tau_1}\boxed{\tau_2}}\boxed{\boxed{\tau_3}\boxed{\tau_4}}}) \oplus (\boxed{\boxed{\boxed{\tau_5}\boxed{\tau_6}}\boxed{\boxed{\tau_7}\boxed{\tau_8}}}) \\
= \; & (\boxed{\boxed{\ominus(\boxed{\tau_3}\boxed{\tau_4})}\boxed{\ominus(\boxed{\tau_1}\boxed{\tau_2})}}) \oplus (\boxed{\boxed{\boxed{\tau_5}\boxed{\tau_6}}\boxed{\boxed{\tau_7}\boxed{\tau_8}}}) \\
= \; & (\boxed{\boxed{\boxed{\tau_4}\boxed{\tau_3}}\boxed{\boxed{\tau_2}\boxed{\tau_1}}}) \oplus (\boxed{\boxed{\boxed{\tau_5}\boxed{\tau_6}}\boxed{\boxed{\tau_7}\boxed{\tau_8}}}) \\
= \; & \boxed{\boxed{(\boxed{\tau_4}\boxed{\tau_3}) \oplus (\boxed{\tau_5}\boxed{\tau_6})}\boxed{(\boxed{\tau_2}\boxed{\tau_1}) \oplus (\boxed{\tau_7}\boxed{\tau_8})}} \\
= \; & \boxed{\boxed{(\boxed{\tau_4 \oplus \tau_5}\boxed{\tau_3 \oplus \tau_6})}\boxed{(\boxed{\tau_2 \oplus \tau_7}\boxed{\tau_1 \oplus \tau_8})}} \\
= \; & \boxed{\boxed{(\boxed{\tau_4 \uplus \tau_5}\boxed{\tau_3 \uplus \tau_6})}\boxed{(\boxed{\tau_2 \uplus \tau_7}\boxed{\tau_1 \uplus \tau_8})}}
\end{aligned}$$

This may be better understood by visualizing each of the argument type and result types as two squares. The square representing the argument type is flipped in both dimensions effectively swapping the labels on both diagonals. The resulting square is then superimposed on the square for the result type to give the representation of the function as a first-class object.

3.3 Type Isomorphisms: Paths to the Rescue

Our proposed semantics of types identifies several structurally different types such as $(1+(1+1))$ and $((1+1)+1)$. In some sense, this is innocent as the types are isomorphic. However, in the operational semantics discussed in Sec. 6, we make the computational content of such type isomorphisms explicit. Some other isomorphic types like $(\tau_1 * \tau_2)$ and $(\tau_2 * \tau_1)$ map to different cubes and are *not* identified: explicit isomorphisms are needed to mediate between them. We therefore need to enrich our model of types with isomorphisms connecting types we deem equivalent. So far, our types are modeled as cubes which are really sets indexed by polarities. An isomorphism between $(\tau_1 * \tau_2)$ and $(\tau_2 * \tau_1)$ requires nothing more than a pair of set-theoretic functions between the spaces, and that compose to the identity. What is much more interesting are the isomorphisms involving the empty type 0. In particular, if negative types are to be interpreted as their name suggests, we must have an isomorphism between $(t - t)$ and the empty type 0. Semantically the former denotes the "line" $\boxed{\mathbb{T}}\boxed{\mathbb{T}}$ and the latter denotes the empty set. Their denotations are different and there is no way, in the world of plain sets, to express the fact that these two spaces should be identified. What is needed is the ability to *contract* the *path* between the endpoints of the line to the trivial path on the empty type. This is, of course, where the ideas of homotopy (type) theory enter the development.

Consider the situation above in which we want to identify the spaces corresponding to the types $(1 - 1)$ and the empty type:



The top of the figure is the 1-dimensional cube representing the type $(1 - 1)$ as before except that we now add a path seg_1 to connect the two endpoints. This path identifies the two occurrences of 1. (Note that previously, the dotted lines in the figures were a visualization aid and were *not* meant to represent paths.) We also make explicit the trivial identity paths from every space to itself. The bottom of the figure is the 0-dimensional cube representing the empty type. To express the equivalence of $(1 - 1)$ and 0, we add a 2-path q, i.e. a path between paths, that connects the path seg_1 to the trivial path refl_0. That effectively makes the two points "disappear." Surprisingly, that is everything that we need. The extension to higher dimensions just "works" because paths in HoTT have a rich structure. We explain the details after we include a short introduction of the necessary concepts from HoTT.

4. Condensed Background on HoTT

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

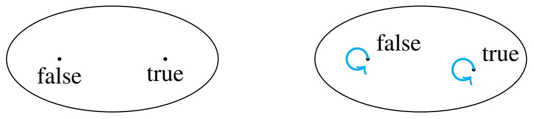
4.1 Types as Spaces

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

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

i0 : 3 ≡ 3
i0 = refl 3

i1 : (1 + 2) ≡ (3 * 1)
i1 = refl 3

i2 : ℕ ≡ ℕ
i2 = refl ℕ
```

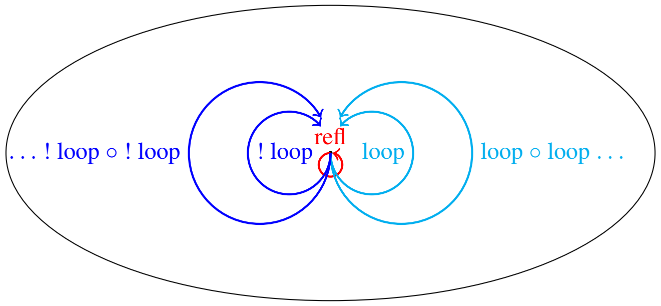
It is important to note that the notion of proposition equality $\equiv$ relates any two terms that are *definitionally equal* as shown in example i1 above. In general, there may be *many* proofs (i.e., paths) showing that two particular values are identical and that proofs are not necessarily identical.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

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



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

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



The additional structure of types is formalized as follows:

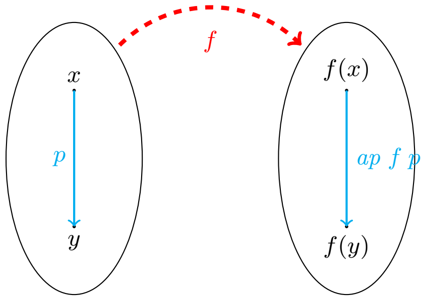
- For every path $p : x \equiv y$, there exists a path $!p : y \equiv x$;
- For every $p : x \equiv y$ and $q : y \equiv z$, there exists a path $p \circ q : x \equiv z$;
- Subject to the following conditions:

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

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

4.2 Functions

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

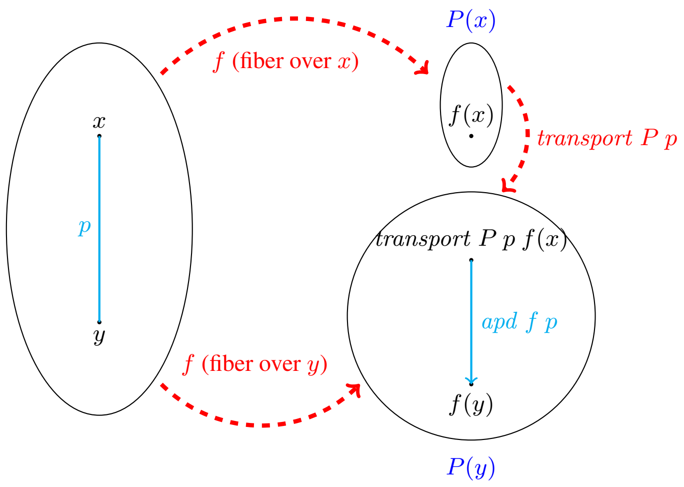


- $ap\ f\ p$ is the action of f on a path p;
- This satisfies the following properties:

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

Type families as fibrations.

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



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

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

- Let $x, y : A$, $p : x \equiv y$, $P : A \to \mathit{Set}$, and $f : \Pi_{a \in A} P(a)$;
- We know we have a path in $P(y)$ between $transport\ P\ p\ (f(x))$ and $f(y)$.
- We do not generally know how the point $transport\ P\ p\ (f(x)) : P(y)$ relates to x;
- We do not generally know how the paths in $P(y)$ are related to the paths in A.
- We know that paths in $A \times B$ are pairs of paths, i.e., 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.
- We know that paths in $A_1 \uplus A_2$ are tagged, i.e., 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.

5. Homotopy Types and Univalence

We describe the construction of our universe of types.

5.1 Homotopy Types

Our starting point is the construction in Sec. 3 which we summarize again. We start with all finite sets built from the empty set, a singleton set, disjoint unions, and cartesian products. All these sets are thought of being indexed by the empty sequence of polarities ϵ. We then constructs new spaces that consist of pairs of finite sets $\boxed{S_1 \mid S_2}$ indexed by positive and negative polarities. These are the 1-dimensional spaces. We iterate this construction to the limit to get n-dimensional cubes for all natural numbers n.

At this point, we switch from viewing the universe of types as an unstructured collection of spaces to a viewing it as a *groupoid* with explicit *paths* connecting spaces that we want to consider as equivalent. We itemize the paths we add next:

- The first step is to identify each space with itself by adding trivial identity paths $\mathsf{refl}_{\mathbb{T}}$ for each space $\mathbb{T}$;
- Then we add paths $\mathsf{seg}_{\mathbb{T}}$ that connect all occurrences of the same type $\mathbb{T}$ in various positions in n-dimensional cubes. For example, the 1-dimensional space corresponding to $(1 - 1)$ would now include a path connecting the two endpoints as illustrated at the end of Sec. 3.
- We then add paths for witnessing the usual type isomorphisms between finite types such as associativity and commutativity of sums and products. The complete list of these isomorphisms is given in the next section.
- Finally, we add *2-paths* between refl_0 and any path $\mathsf{seg}_{\mathbb{T}}$ whose endpoints are of opposite polarities, i.e., of polarities s and $neg(s)$ where:

$$\begin{array}{rcl} neg(\epsilon) & = & \epsilon \\ neg(+s) & = & -neg(s) \\ neg(-s) & = & +neg(s) \end{array}$$

- The groupoid structure forces other paths to be added as described in the previous section.

Now the structure of path spaces is complicated in general. Let's look at some examples.

5.2 Univalence

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 *is* the computational interpretation of univalence. Specifically, in the context of finite types, univalence specializes to a relationship between type isomorphisms on the side of syntactic identities and permutations in the symmetric group on the side of semantic equivalences.

In conventional HoTT:

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

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

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

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

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

In the conventional setting, this is not executable!
Analysis:

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

Insight:

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

Note that:

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

6. A Reversible Language with Cubical Types

We first define values then combinators that manipulate the values to witness the type isomorphisms.

6.1 Values

Now that the type structure is defined, we turn our attention to the notion of values. Intuitively, a value of the n-dimensional type τ is an element of one of the sets located in one of the corners of the n-dimensional cube denoted by τ (taking into that there are non-trivial paths relating these sets to other sets, etc.) Thus to specify a value, we must first specify one of the corners of the cube (or equivalently one of the leaves in the binary tree representation) which can easily be done using a sequence s of $+$ and $-$ polarities indicating how to navigate the cube in each successive dimension starting from a fixed origin to reach the desired corner. We write v^s for the value v located at corner s of the cube associated with its type. We use ϵ for the empty sequence of polarities and identify v with v^ϵ. Note that the polarities doesn't completely specify the type since different types like $(1 + (1 + 1))$ and $((1 + 1) + 1)$ are assigned the same denotation. What the path s specifies is the *polarity* of the value, or its "orientation" in the space denoted by its type. Formally:

$$\frac{}{() : 1} \qquad \frac{v^s : \tau}{v^{neg(s)} : -\tau}\ neg$$

$$\frac{v^s : \tau_1}{(\mathsf{inl}\ v)^s : \tau_1 + \tau_2}\ left \qquad \frac{v^s : \tau_2}{(\mathsf{inr}\ v)^s : \tau_1 + \tau_2}\ right$$

$$\frac{v_1^{s_1} : \tau_1 \quad v_2^{s_2} : \tau_2}{(v_1, v_2)^{s_1 \cdot s_2} : \tau_1 * \tau_2}\ prod$$

The rules *left* and *right* reflect the fact that sums do not increase the dimension. Note that when s is ϵ, we get the conventional values for the 0-dimensional sum type. The rule *prod* is the most involved one: it increases the dimension by *concatenating* the two dimensions of its arguments. For example, if we pair v_1^+ and v_2^{-+} we get $(v_1, v_2)^{+-+}$. (See Fig. 1 for the illustration.) Note again that if both components are 0-dimensional, the pair remains 0-dimensional and we recover the usual rule for typing values of product types. The rule *neg* uses the function below which states

$identl_+ :$	$0 + b$	$\leftrightarrow$	b	$: identr_+$
$swap_+ :$	$b_1 + b_2$	$\leftrightarrow$	$b_2 + b_1$	$: swap_+$
$assocl_+ :$	$b_1 + (b_2 + b_3)$	$\leftrightarrow$	$(b_1 + b_2) + b_3$	$: assocr_+$
$identl_* :$	$1 * b$	$\leftrightarrow$	b	$: identr_*$
$swap_* :$	$b_1 * b_2$	$\leftrightarrow$	$b_2 * b_1$	$: swap_*$
$assocl_* :$	$b_1 * (b_2 * b_3)$	$\leftrightarrow$	$(b_1 * b_2) * b_3$	$: assocr_*$
$dist_0 :$	$0 * b$	$\leftrightarrow$	0	$: factor_0$
$dist :$	$(b_1 + b_2) * b_3$	$\leftrightarrow$	$(b_1 * b_3) + (b_2 * b_3)$	$: factor$
$\eta :$	0	$\leftrightarrow$	$b + (-b)$	$: \epsilon$

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

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

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

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

Table 1. Combinators

that the negation of a value v is the same value v located at the "opposite" corner of the cube.

6.2 Π Combinators

The terms of Π witness type isomorphisms of the form $b \leftrightarrow b$. They consist of base isomorphisms, as defined in Table 1 and their composition. Each line of the table introduces a pair of dual constants[1] that witness the type isomorphism in the middle. These are the base (non-reducible) terms of the second, principal level of Π. Note how the above has two readings: first as a set of typing relations for a set of constants. Second, if these axioms are seen as universally quantified, orientable statements, they also induce transformations of the (traditional) values. The (categorical or homotopical) intuition here is that these axioms have computational content because they witness isomorphisms rather than merely stating an extensional equality. The isomorphisms are extended to form a congruence relation by adding constructors that witness equivalence and compatible closure.

It is important to note that "values" and "isomorphisms" are completely separate syntactic categories which do not intermix. The semantics of the language come when these are made to interact at the "top level" via *application*:

$$\textit{top level term}, l \quad ::= \quad c\ v$$

7. Related Work and Context

A ton of stuff here.

Connection to our work on univalence for finite types. We didn't have to rely on sets for 0-dimensional types. We could have used groupoids again.

8. Conclusion

References

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.

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.

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.

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