Negative Types

Abstract

...

1. Introduction

The **Int** construction or the $\mathcal{G}$ construction are neat. As Neel K. explains, 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. We "translate" that solution to a computational type-theoretic world. This has evident connections to homotopy (type) theory that remain to be investigated.

2. The Int Construction

Explain in detail perhaps with Haskell embedding and type functions etc. The key insight it to add "negative" types representing demand for values. This is how you get functions.

We lose the multiplicative structure. No evident way to define the multiplication functor. Perhaps there is a clever way. But this turns out to be a well-known open problem. Review the problem and tell the story from the algebraic topology perspective. Regular first-order types are viewed as 0-dimensional cubes. The **Int** construction generalizes types to 1-dimensional cubes. Our work here generalizes types to n-dimensional cubes.

3. The Type Structure

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:

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

The base types have dimension 0. If negative types are not used, all dimensions remain at 0. If negative types are used but no product types 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 $t_1 - t_2$ 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 $(t_1 - t_2) - (t_3 - t_4)$ 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 $(t_1 - t_2)$ and $(t_3 - t_4)$. (See Fig. 1 which is further explained after we discuss multiplication below.)

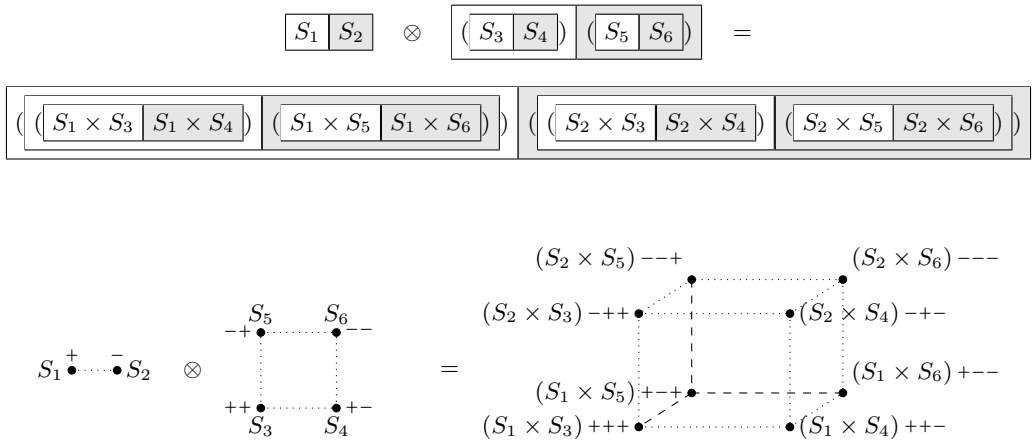


Figure 1. Example of multiplication of two cubical types.

Formally, the denotation of types is as follows:

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

where:

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

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

$$\begin{array}{rcl} \ominus S & = & \mid S \\ \ominus (\mathbb{T}_1 \mid \mathbb{T}_2) & = & \ominus \mathbb{T}_2 \mid \ominus \mathbb{T}_1 \end{array}$$

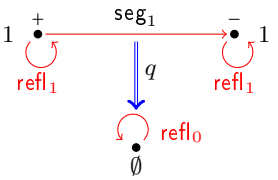
The type 0 maps to the empty set. The type 1 maps to a singleton set. The sum of 0-dimensional types is the disjoint union as usual. For cubes of higher dimensions, the subspaces are recursively added. Note that the sum of 1-dimensional types reduces to the sum used in the **Int** construction. The definition 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 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 the next section, we make the computational content of such type isomorphisms explicit. Some other isomorphic types like $(t_1 * t_2)$ and $(t_2 * t_1)$ 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 $(t_1 * t_2)$ and $(t_2 * t_1)$ requires nothing more than a pair of set-theoretic functions between the spaces 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" $\mathbb{T} \mid \mathbb{T}$ 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. (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. The path seg_1 identifies the two occurrences of 1. 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 homotopy type theory have a rich structure. We explain the details after we include a short introduction of the necessary concepts from homotopy type theory.

4. Homotopy Type Theory

Informally, and as a first approximation, one may think of *homotopy type theory* (HoTT) as mathematics, type theory, or computation but with all equalities replaced by isomorphisms, i.e., with equalities given computational content. A bit more formally, one starts with Martin-Löf type theory, interprets the types as topological spaces or weak ∞-groupoids, and interprets identities between elements of a type as *paths*. In more detail, one interprets the witnesses of the identity $x \equiv y$ as paths from x to y. If x and y are themselves paths, then witnesses of the identity $x \equiv y$ become paths between paths, or homotopies in the topological language.

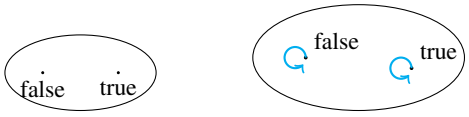
Formally, Martin-Löf type theory, is based on the principle that every proposition, i.e., every statement that is susceptible to proof, can be viewed as a type. The correspondence is validated by the following properties: if a proposition P is true, the corresponding type is inhabited, i.e., it is possible to provide evidence for P using one of the elements of the type P. If, however, the proposition P is false, the corresponding type is empty, i.e., it is impossible to provide evidence for P. The type theory is rich enough to allow propositions denoting conjunction, disjunction, implication, and existential and universal quantifications.

It is clear that the question of whether two elements of a type are equal is a proposition, and hence that this proposition must correspond to a type. In Agda notation, we can formally express this as follows:

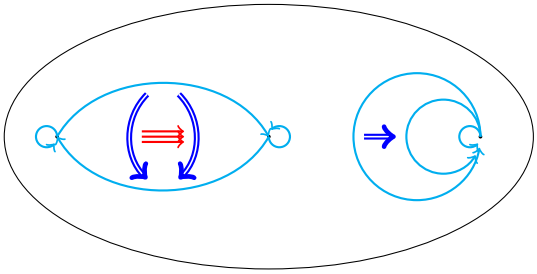
```
data _≡_ {ℓ} {A : Set ℓ} : (a b : A) → Set ℓ where
  refl : (a : A) → (a ≡ a)

i0 : 3 ≡ 3
i0 = refl 3

i1 : (1 + 2) ≡ (3 * 1)
i1 = refl 3

i2 : ℕ ≡ ℕ
i2 = refl ℕ
```

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

We are used to think of types as sets of values. So we think of the type `Bool` as the figure on the left but 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 might have something like which shows that types are to be viewed as topological spaces or groupoids:



The additional structure of types is formalized as follows:

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

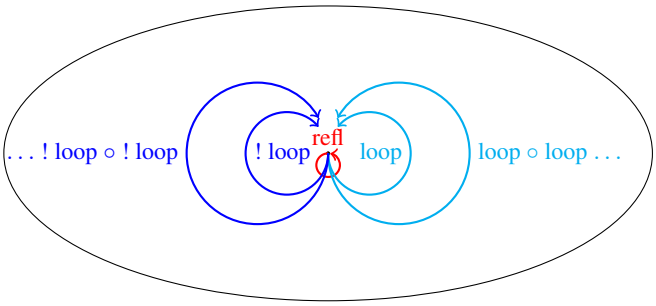
$$\begin{array}{rcl} p \circ \mathit{refl}\ y & \equiv & p \\ p & \equiv & \mathit{refl}\ x \circ p \\ !p \circ p & \equiv & \mathit{refl}\ y \\ p \circ\ !p & \equiv & \mathit{refl}\ x \\ !\,(!p) & \equiv & p \\ p \circ (q \circ r) & \equiv & (p \circ q) \circ r \end{array}$$

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

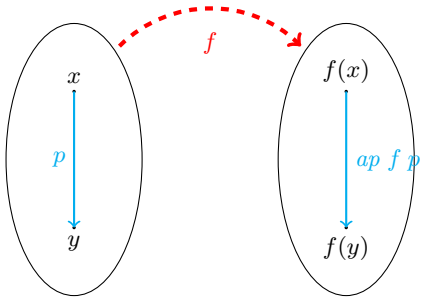
We cannot generate non-trivial groupoids starting from the usual type constructions. We need *higher-order inductive types* for that purpose. Example:

```
- data Circle : Set where
- base : Circle
- loop : base ≡ base

module Circle where
  private data S¹* : Set where base* : S¹*

  S¹ : Set
  S¹ = S¹*

  base : S¹
  base = base*

  postulate loop : base ≡ base
```

Here is the non-trivial structure of this example:



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

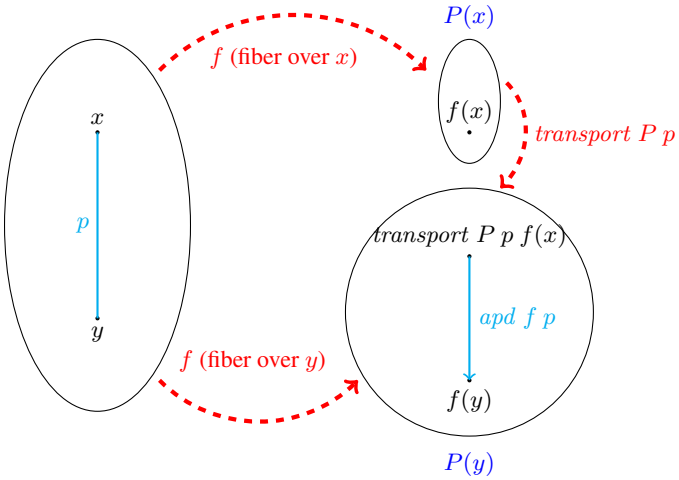


- $ap\ f\ p$ is the action of f on a path p;
- This satisfies the following properties:

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

Type families as fibrations.

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



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

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

- Let $x, y : A$, $p : x \equiv y$, $P : A \to Set$, and $f : \Pi_{a \in A} P(a)$;
- We know we have a path in $P(y)$ between $transport\ P\ p\ (f(x))$ and $f(y)$.
- We do not generally know how the point $transport\ P\ p\ (f(x)) : P(y)$ relates to x;
- We do not generally know how the paths in $P(y)$ are related to the paths in A.
- First "crack" in the theory.

Structure of Paths:

- What do paths in $A \times B$ look like? We can prove that $(a_1, b_1) \equiv (a_2, b_2)$ in $A \times B$ iff $a_1 \equiv a_2$ in A and $b_1 \equiv b_2$ in B.
- What do paths in $A_1 \uplus A_2$ look like? We can prove that $inj_i\ x \equiv inj_j\ y$ in $A_1 \uplus A_2$ iff $i = j$ and $x \equiv y$ in A_i.
- What do paths in $A \to B$ look like? We cannot prove anything. Postulate function extensionality axiom.
- What do paths in Set_ℓ look like? We cannot prove anything. Postulate univalence axiom.

5. The Universe of Types

We describe the construction of our universe of types. Our starting point is the construction in the previous section 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 $\boxed{S_1}\boxed{S_2}$ 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 at the end of the previous section;
- 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 detail in the recent book on Homotopy Type Theory (The Univalent Foundations Program 2013). In particular, for every path p, the groupoid structure requires an inverse path p^{-1} in the opposite direction; and for every two paths with common ending and starting point, a path corresponding to the composition of the two paths is included and so on. These paths obey various coherence conditions that ensure for example that composing a path and its inverse is equivalent to the trivial path, and that composition is associative, up to path equivalences at higher levels.

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

```
be careful about that 2-path. Are we
```

allowed to do that given that the
empty type is, well, empty.

identical to a single point. It should
then follow that a square where one edge
contracts to a point would itself contract
to a line, and a 3-dimensional cube
in which one face contracts to a point
would itself contract to a square, etc.

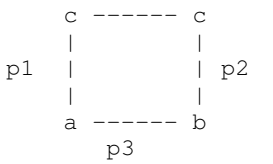
So the story is that as we are building
the denotations of types, we add these
paths making up the cubes. The only
case where we add new paths is multiplication
(see Fig.1 again). We take the paths p in S1
and the paths q in S2 and build paths (p,q)
in the product space.

Once we have this structure, we do this
homotopy completion which turns our inductive
type into a higher-inductive type. We
start adding 2-paths between any path
connecting (S-S) and the refl path on 0.

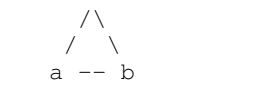
Technically this is a functor between (S-S)
and the empty type mapping S to 0, the
other S to 0, and the path between them
to refl.

Now consider (a-b)-(c-c). We have a
functor to (a-b)-0 that contracts the
path between c and -c. We want to
reduce the dimension of that space
and map to (a-b).

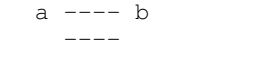
original space



after collapsing the two c's



or because path compose



where the top path is p1;p2
and the bottom path is p3

I think we should have a 2-path
between them to say that these
paths are equivalent which will
then collapse the square to a
line as desired.

A critical isomorphism that is
not reflected at all in the semantics
is that 1-1 = 0

The denotation of 1-1 is a line;
the denotation of 0 is the empty set

```
[ {*} | {*} ] --> []
```

Something like the simplification
of Conway games here perhaps?

There are two values of type
1-1

```
()
```
and
```
()^-
```

epsilon should take one and map
to the other because it can
never produce something of type 0

In general, we have several direction
not just left and right like in the
Int construction. In 3D we have
8 directions: +++, ++-, ..., ---
We have an epsilon that
takes ++- and flips it to --+
and another that flips
+++ to --- and so on

So we should have 8 interpreters
for 3D one for each direction!!!

The ++- interpreter would
manipulates values like
(v1,v2,v3) and would apply
the normal combinators to v1, v2,
and the adjoint to v3 and so on.

To complete the story we need to
define morphisms. (More on this below.)
Once we have a notion of morphism we
can check whether X + 0 is the same
as X etc. i.e., we can check all the
ring equivalences.

Ok so what are the morphisms between
these cubical objects? We know what
they are for 1-dimensional cubes:
they are the pi combinabors. We also
know what they are for the 2-dimensional
cubes: a maps (A-B) ==> (C-D)
is a Pi map between A+D <=> C+B.
How to generalize this?

Why is there no trace in the ring
completion paper??? What are
the morphisms in that paper?

The ring completion paper produces
a simplicial category.

```
p. 3 talks about the group cancellation
as subcubes along the diagonal...

We shouldn't focus on denotations. We want
an operational semantics for pi with
negatives. The same way that Neel turned
the G construction into code that does
something neat; we want to turn that
ring completion construction into code
that produces h.o. functions without
losing the multiplicative structure.

Show how to embed a square in 2D into
each of the faces of the 3D cube.

1D into 2D:

(a-b) => (a-b)-(0-0)
(a-b) => (a-0)-(b-0)
(a-b) => (0-b)-(0-a)
(a-b) => (0-0)-(b-a)
```

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. 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:

$$\frac{}{() : 1}$$

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

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

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

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

```
We only have -
which flips ALL the directions simultaneously.
So from ++-+--+ you can only go to
        --+-++-
What if you wanted to go to
        ++++--+
I think we use the * functor to flip the
direction we want.
```

6.2 Combinators: Example

Consider the following simple function on 0-dimensional sum types:

$$\begin{array}{rcl} \mathsf{swapP} & :: & (\tau_1 + \tau_2) \rightarrow (\tau_2 + \tau_1) \\ \mathsf{swapP}(\mathsf{inl}\, v) & = & inr v \\ \mathsf{swapP}(\mathsf{inr}\, v) & = & inl v \end{array}$$

Given our setup, this just works n-dimensional types. And so do most of the Π combinators for that matter. Only η and ϵ seem to need thinking. How and why would $1 - 1$ which is a 1-dimensional line be the same as the empty type which is a 0-dimensional thing. And how do we generalize for arbitrary group identities at higher dimensions. We need a mechanism for cubes with subspaces that "cancel" to map to equivalent smaller subcubes.

6.3 Π Combinators

The terms of Π witness type isomorphisms of the form $b \leftrightarrow b$. They consist of base isomorphisms, as defined below, and their composition. Each line of the above 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 the following 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

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

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

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

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

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

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

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

Table 1. Combinators