Negative Types

Abstract

. . .

1. Introduction

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

2. Computing with Type Isomorphisms

In a computational model in which resources are carefully maintained, programs are reduced to type isomorphisms. Although this is apparently a limiting idea, it turns out that conventional computation can be viewed as a special case of such resource-preserving isomorphisms that silently consumes some resources and discards others. This thesis has been explored for many years from different perspectives [Bennett 2003, 2010, 1973; Fredkin and Toffoli 1982; Landauer 1961, 1996; Toffoli 1980]. We build on the work of James and Sabry [2012] which expresses this thesis in a type theoretic computational framework. Unlike the case for their development, we will however not consider recursive types in this paper but we will develop an extension with higher-order *linear* functions.

The main syntactic vehicle for the developments in this paper is a simple language called Π whose only computations are isomorphisms between finite types. The set of types τ includes the empty type 0, the unit type 1, and conventional sum and product types:

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

Values v are the expected ones: $()$ of type 1, $\mathsf{inl}\ v$ and $\mathsf{inr}\ v$ for injections into sum types, and (v_1, v_2) for product types.

The interesting syntactic category of Π is that of *combinators* which are witnesses for type isomorphisms of the form $b \leftrightarrow b$. They consist of base isomorphisms (on the left side of Table 1) and compositions (on the right side of the same table). Each line of the table on the left introduces a pair of dual constants[1] that witness the type isomorphism in the middle. This set of isomorphisms is known to be complete [Fiore 2004; Fiore et al. 2006] and the language is universal for hardware combinational circuits [James and Sabry 2012]. Note that we have *trace* c which in the current setting is a bounded iteration construct: it adds no expressiveness for now but will be important to model functions later. If recursive types are added, the bounded iteration becomes Turing-complete [Bowman et al. 2011; James and Sabry 2012] but we will not be concerned with recursive types in this paper.

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

From the perspective of category theory, the language Π models what is called a *symmetric bimonoidal category* or a *commutative rig category*. These are categories with two binary operations $\oplus$ and $\otimes$ satisfying the axioms of a rig (i.e., a ring without negative elements also known as a semiring) up to coherent isomorphisms. And indeed the combinators of Π are precisely the semiring axioms. A simple (slightly degenerate) example of such categories is the category of finite sets and permutations. Indeed, it is possible to interpret every Π-type as a finite set, the values as elements in these finite sets, and the combinators as permutations.

The previous interpretation of Π, although valid, misses the point of taking isomorphisms seriously as *the* essence of computation. Luckily, an impressive amount of work has been happening in HoTT that builds around the computational content of equalities, equivalences, and isomorphisms. We discuss our HoTT reinterpretation of Π semantics after we briefly review some of the essential concepts. The definitive reference is the recently published comprehensive book [The Univalent Foundations Program 2013].

3. 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.

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. To be explicit, we will use $\equiv_U$ to refer to the space in which the path lives.

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

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

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

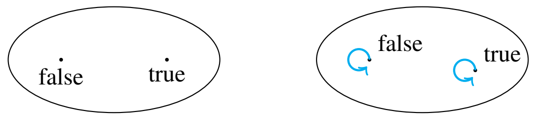
$$\frac{\vdash c_1 : \tau_1 \leftrightarrow \tau_2 \quad c_2 : \tau_3 \leftrightarrow \tau_4}{\vdash c_1 \oplus c_2 : \tau_1 + \tau_3 \leftrightarrow \tau_2 + \tau_4}$$

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

$$\frac{\vdash c : \tau_1 \oplus \tau \leftrightarrow \tau_2 \oplus \tau}{\vdash trace\ c :: \tau_1 \leftrightarrow \tau_2}$$

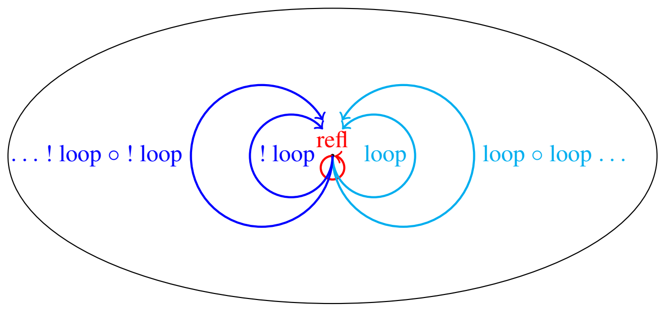
Table 1. Π-combinators [James and Sabry 2012]

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



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 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 U:

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

$$\begin{array}{rcl} p \circ refl\ y & \equiv_{x \equiv_U y} & p \\ p & \equiv_{x \equiv_U y} & refl\ x \circ p \\ !p \circ p & \equiv_{y \equiv_U y} & refl\ y \\ p \circ\ !p & \equiv_{x \equiv_U x} & refl\ x \\ !\ (!p) & \equiv_{x \equiv_U y} & p \\ p \circ (q \circ r) & \equiv_{x \equiv_U z} & (p \circ q) \circ r \end{array}$$

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

4. The Space of Π-Types

Instead of modeling the semantics of Π using *permutations*, which are are set-theoretic functions after all, we use *paths* from the HoTT framework. More precisely, we model the universe of Π types as a space whose points are the individual Π-types and we will consider that there is path between two points τ_1 and τ_2 if there is a Π combinator $c : \tau_1 \leftrightarrow \tau_2$. If we focus on 1-paths, this is perfect as we explain next.

Note. But first, we note that this is a significant deviation from the HoTT framework which fundamentally includes functions, which are specialized to equivalences, which are then postulated to be paths by the univalence axiom. This axiom has no satisfactory computational interpretation, however. Instead we completely bypass the idea of extensional functions and use paths directly. Another way to understanding what is going on is the following. In the conventional HoTT framework:

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

In our case,

- We start with a constructive characterization of *reversible functions* or *isomorphisms* built using inductively defined combinators;
- We blur the distinction between such combinators and paths from the beginning. We view computation as nothing more than *following paths*! As explained earlier, although this appears limiting, it is universal and regular computation can be viewed as a special case of that.

Construction. We have a universe U viewed as a groupoid whose points are the types Π-types τ. The Π-combinators of Table 1 are viewed as syntax for the paths in the space U. We need to show that the groupoid path structure is faithfully represented. The combinator id introduces all the $\mathsf{refl}_\tau : \tau \equiv \tau$ paths in U. The adjoint $sym\ c$ introduces an inverse path $!p$ for each path p introduced by c. The composition operator $\fatsemi$ introduce a path $p \circ q$ for every pair of paths whose endpoints match. In addition, we get paths like $swap_+$ between $\tau_1 + \tau_2$ and $\tau_2 + \tau_1$. The existence of such paths in the conventional HoTT developed is *postulated* by the univalence axiom. The $\otimes$-composition gives a path (p, q) : $(\tau_1 * \tau_2) \equiv (\tau_3 * \tau_4)$ whenever we have paths $p : \tau_1 \equiv \tau_3$ and $q : \tau_2 \equiv \tau_4$. A similar situation for the $\oplus$-composition. The structure of these paths must be discovered and these paths must be *proved* to exist using path induction in the conventional HoTT development. So far, this appears too good to be true, and it is. The problem is that paths in HoTT are subject to rules discussed at the end of Sec. 3. For example, it must be the case that if $p : \tau_1 \equiv_U \tau_2$ that $(p \circ \mathsf{refl}_{\tau_2}) \equiv_{\tau_1 \equiv_U \tau_2} p$. This path lives in a higher universe: nothing in our Π-combinators would justify adding such a path as all our combinators map types to types. No combinator works one level up at the space of combinators and there is no such space in the first place. Clearly we are stuck unless we manage to express a notion of higher-order functions in Π. This would allow us to internalize the type $\tau_1 \leftrightarrow \tau_2$ as a Π-type which is then manipulated by the same combinators one level higher and so on.

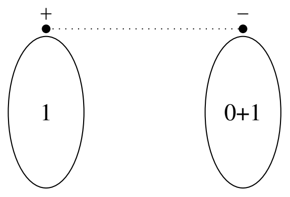
5. The Int Construction

The **Int** construction [Joyal et al. 1996] or the closely related $\mathcal{G}$ construction [Abramsky 1996] are fascinating ideas. As Krishnaswami [2012] explains, given first-order types and feedback, it is possible to higher-order *linear* functions. The key insight it to enrich types to be of the shape $(\tau_1 - \tau_2)$ which represent a sum type of τ_1 flowing in the "normal" direction (from producers to consumers) and τ_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. Since we already have *trace c* to provide feedback, the next immediate step is extend the Π types as follows:

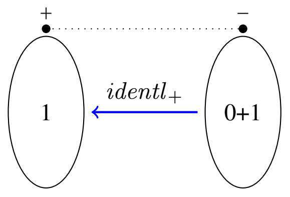
$$\begin{array}{rcl} \tau & ::= & 0 \mid 1 \mid \tau_1 + \tau_2 \mid \tau_1 * \tau_2 \\ \mathbb{T} & ::= & \tau_1 - \tau_2 \end{array}$$

In anticipation of future developments, we will call the original types τ 0-dimensional and the new types $\mathbb{T}$ 1-dimensional. The previous combinators all work on 0-dimensional types. The punchline will be that a 0-level combinator $c : \tau_1 \leftrightarrow \tau_2$ on 0-dimensional types can be expressed as a value v_c of the 1-dimensional type $\tau_2 - \tau_1$. Furthermore, we will have lifted versions of most of the combinators to work on the 1-dimensional types. These lifted versions will allow us to manipulate combinators on 0-dimensional types as first-class values.

Before we define the construction in any detail, let's take one simple example $identl_+ : 0 + 1 \leftrightarrow 1$ and see how to represent as a value of type $1 - (0 + 1)$. We visually represent the type itself as a line with 0-dimensional types attached at the endpoints (which are distinguished by polarities):



The value representing $identl_+$ is simply:



This entire package is a value, an atomic entity with invisible internal structure and that can only be manipulated via the level 1 combinators described next.

The next order of business is to define a few abbreviations of 1-dimensional types:

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

The level 1 combinators are now exactly identical to the combinators in Table 1 changing all the 0 dimensional types τ to 1 dimensional types $\mathbb{T}$ and hence replacing all occurrences of 0 by $\mathbb{0}$, 1 by $\mathbb{1}$, $+$ by $\boxplus$, and $*$ by $\boxtimes$.

Most of the time, the level 1 combinators are oblivious to the fact that they are manipulating first class functions. Formally, the action of a level 1 combinator of type $(\tau_1 - \tau_2) \overset{1}{\leftrightarrow} (\tau_3 - \tau_4)$ is derived from the action of 0 level combinators of type $(\tau_1 + \tau_4) \leftrightarrow (\tau_3 + \tau_2)$. Thus to take a trivial example $identl^1_+ : \mathbb{0} \boxplus \mathbb{T} \overset{1}{\leftrightarrow} \mathbb{T}$ is realized as $assocr_+ \fatsemi (id \oplus swap_+) \fatsemi assocl_+$ which evidently knows nothing specific about higher order functions. The interesting idea is that it is possible to define a new level 1 combinator that exposes the internal structure of values as functions:

$$\eta : \quad 0 - 0 \quad \overset{1}{\leftrightarrow} \quad \tau - \tau \quad : \varepsilon$$

What this does is essentially provide an input and output port which are connected by the internal hidden function.

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].

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 problem in algebraic topology and homotopy theory identified in 1980 [Thomason 1980] and 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 we investigate in some depth.

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.

6. Cubes

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

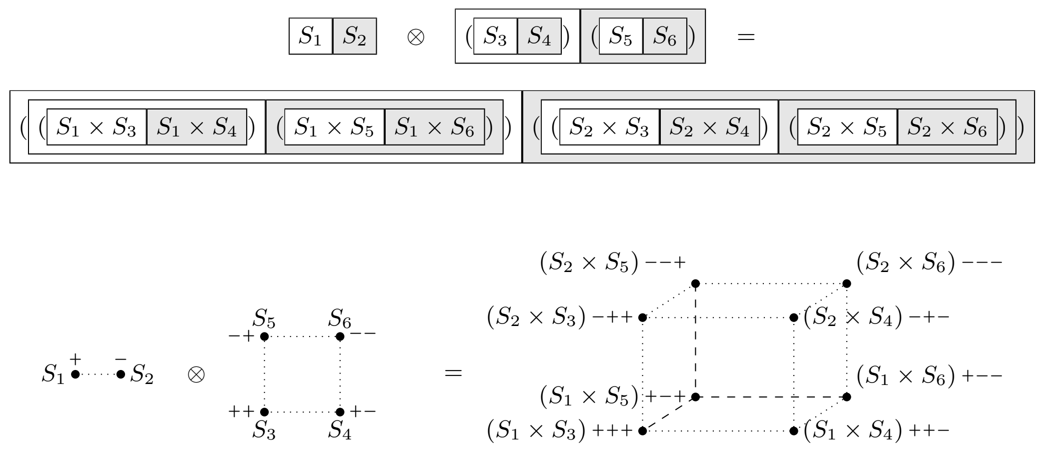
6.1 Negative and Cubical Types

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:

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



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

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 Sec. 7) 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:

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

where:

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

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

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

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.

6.2 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:

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

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

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

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

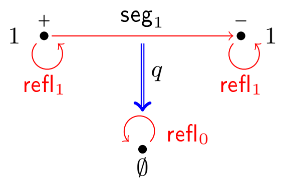
$$\begin{aligned}
& \boxed{\boxed{\tau_1 \mid \tau_2}}\boxed{\boxed{\tau_3 \mid \tau_4}} \multimap \boxed{\boxed{\tau_5 \mid \tau_6}}\boxed{\boxed{\tau_7 \mid \tau_8}} \\
= \; & (\ominus \boxed{\boxed{\tau_1 \mid \tau_2}}\boxed{\boxed{\tau_3 \mid \tau_4}}) \oplus (\boxed{\boxed{\tau_5 \mid \tau_6}}\boxed{\boxed{\tau_7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

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.

6.3 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. 8, 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" $\boxed{\mathbb{T} \mid \mathbb{T}}$ 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. 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.

7. Homotopy Types and Univalence

We describe the construction of our universe of types.

7.1 Homotopy Types

Our starting point is the construction in Sec. 6 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. 6.
- 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:

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

- 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.

7.2 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)
```

8. A Reversible Language with Cubical Types

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

8.1 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:

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

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

$$\frac{v_1^{s1} : \tau_1 \quad v_2^{s2} : \tau_2}{(v_1, v_2)^{s_1 \cdot s_2} : \tau_1 * \tau_2}\ \mathit{prod}$$

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.

9. 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.

10. Conclusion

References

S. Abramsky. Retracing some paths in process algebra. In U. Montanari and V. Sassone, editors, *CONCUR '96: Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin Heidelberg, 1996. ISBN 978-3-540-61604-7. doi: 10.1007/3-540-61604-7_44. URL http://dx.doi.org/10.1007/3-540-61604-7_44.

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.

C. Bennett. Notes on Landauer's principle, reversible computation, and Maxwell's Demon. *Studies In History and Philosophy of Science Part B: Studies In History and Philosophy of Modern Physics*, 34(3):501–510, 2003.

C. Bennett. Notes on the history of reversible computation. *IBM Journal of Research and Development*, 32(1):16–23, 2010.

C. H. Bennett. Logical reversibility of computation. *IBM J. Res. Dev.*, 17: 525–532, November 1973.

W. J. Bowman, R. P. James, and A. Sabry. Dagger Traced Symmetric Monoidal Categories and Reversible Programming. In *RC*, 2011.

M. Fiore. Isomorphisms of generic recursive polynomial types. In *POPL*, pages 77–88. ACM, 2004.

M. P. Fiore, R. Di Cosmo, and V. Balat. Remarks on isomorphisms in typed calculi with empty and sum types. *Annals of Pure and Applied Logic*, 141(1-2):35–50, 2006.

E. Fredkin and T. Toffoli. Conservative logic. *International Journal of Theoretical Physics*, 21(3):219–253, 1982.

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

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.

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.