On Computational Higher-Dimensional Type Theory

An essay submitted as part of coursework for
Advanced Topics in Programming Languages

Pritam Choudhury
December 18, 2016

1 Motivation

The search for foundations of mathematics was at its peak in the early 20^{th} century. But then there were the paradoxes. Many of the paradoxes arose because of comparison between incomparable things, like set and the set of all sets. To overcome the paradoxes, Bertrand Russell formulated his theory of hierarchical types and the idea was that all the individuals of a type shared some common property and as such were comparable with respect to that property. Over the years, type theory has developed into several different forms but the initial aim of type theory providing a foundation for all of mathematics has not been realized till date. One of the reasons why this is so is because of the treatment of equality in type theories. Type theories, in general, are austere when it comes to equality, with the dictum being individuals of a type are equal if and only if they are exactly the same. As the 20^{th} century progressed, the search for foundations of mathematics fell out of fashion and mathematicians were more or less satisfied with set-theoretic or category-theoretic foundations.

Another important contribution from the foundational crisis period is that it led to the development of the intuitionist school of mathematics championed by Brouwer, a strong rival of the traditional formalist school led by Hilbert. The intuitionists believed that the foundations of mathematics rest on human intuition and not on formal proofs. As such, they equated truth of a mathematical proposition with a mental construction realizing that proposition. This constructive approach towards mathematics brought it closer to computer-programming because in a constructive framework, propositions can be thought of as specifications of programs and constructions as programs satisfying those specifications. This also opened up the possibility of using computers to search for and check constructive mathematical proofs. On the other hand, the classical approach does not correspond so well with programming since in general, it is not possible to construct a program from a classical proof that uses the law of the excluded middle. For the intuitionists however, the benefits of any formalization (machine-level or otherwise) lay only in the fact that it helps one to keep track of things and make sure that mistakes are not being made as one progresses.

The close correspondence between programming and constructive mathematics turns constructive logics into programming languages. In these programming languages, we can develop constructive theories. When implemented on a machine, working in such programming

languages would force us to make sure that we indeed do what we say. In addition, the machine can also come up with occasional hints and perform some low-level computations by itself. It is imperative that when our theories become complex and involve a lot of cases to consider, we use such a proof-assistant not just to make the task easy, but also to ensure that we have not ignored some corner cases, which might have seemed trivial but later turned out to be non-trivial and irreconcilable. This has prompted many computer-scientists and mathematicians alike to look towards proof-assistants to formally verify their systems and theories.

But then the proof-assistants are based on constructive type-theoretic logics and the treatment of equality in these logics is too strict to work with. Ideas which are very easy to express on paper might become quite verbose when expressed in these proof-assistants. One of the way out might be to do mathematics modulo an equivalence relation, this is the well-known setoid approach. But then the proof of the equivalence has to be carried all around, making the proofs long and verbose. Another way might be to say that 'I know this is equal to that' and make the proof assistant accept this as a fact, the axiomatic approach. But this is blatantly antagonistic to the idea of intuitionism and constructivism. What we need is not a modification of the way we work with these proof assistants, but a modification of the assistants themselves, because the problem lies in the treatment of equality, by the underlying type-theoretic logics. And this can only be done if we can develop a type-theory with a richer notion of equality.

The connection between homotopy theory and type theory, observed somewhere around 2005-06, provided an insight into how such a thing might be done. In homotopy theory, one does mathematics upto homotopy equivalence. Two spaces A and B are said to be homotopy equivalent if there exists continuous functions $f : A \to B$ and $g : B \to A$ such that $g \circ f$ and $f \circ g$ are respectively homotopic to the identity functions on A and B. Two continuous functions $p, q : X \to Y$ are said to be homotopic if there exists a continuous function $H : X \times [0, 1] \to Y$ such that $H(x, 0) = p(x)$ and $H(x, 1) = q(x)$. The idea of equality as a continuous transformation from the first (the LHS) to the second (the RHS) is a quite powerful one and we frequently use this both formally and informally. In our daily lives, when we say that a baby and the grown-up adult are the same person, what we essentially mean is that we have a continuous transformation linking the baby to the adult, often the transformation is provided by the train of images we have in our mind of the person in the intermediate years, as he or she grew up from a baby to an adult. This idea is equally useful in formal mathematics; our proofs (the chain of steps) can be seen as continuous transformations from the LHS to the RHS. Applying this homotopy idea to type theory, we can now think of types as spaces, individuals of types as inhabitants of those spaces and equality between two individuals of a type as paths between them. This correspondence between type theory and homotopy theory enables one to work in type theory modulo homotopy equivalence, rather than modulo the strong exact equality.

With this, it seems that the time is again ripe to open up the question of foundations for mathematics. Now it might be possible to do more, if not all, of mathematics comparatively easily in a type-theoretic framework. And this time, it comes with an added advantage, the possibility of having machine-checked proofs for mathematical theories. The search for

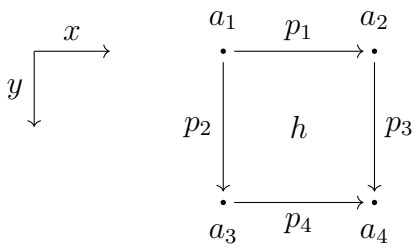
foundations this time is called 'Univalent Foundations' [Uni13], since it rests on the idea of univalence, which says that isomorphic things are equal. Now if we switch our perspective from that of a mathematician to one of a type-theorist, we find that our types do not have to be the way we thought them to be till now, but can be far more richer with interesting internal structures, thanks to our now-rich notion of equality, borrowed from homotopy theory. In fact, we can present a new avatar of type theory with this enriched notion of equality. This is called computational higher-dimensional type theory, with the prefix 'computational' affirming our commitment to a constructive and computational outlook in its development. The following work is my attempt at understanding this theory, as presented in [AHW17].

2 Introduction

In higher-dimensional type theory, types have inner structure. There are paths between terms representing equality of the connected terms, and there are higher paths between those paths representing equality of the connected paths, and so on and so forth ad infinitum. This gives us a hierarchical view of a type, where a type has elements at different levels, at level-0, they are the ordinary terms, at level-1 they are equality proofs (paths) of those terms, at level-2 they are equality proofs (higher paths) between those equality proofs, and so on. Our type-theoretic operations now operate on these structures. To get a meaningful theory out of this, we need to ensure that our type-theoretic operations respect those structures, i.e. any operation must produce equal structures on equal structures. Also, since we have weakened the notion of equality, now equality proofs become all the more important because earlier just by observing the syntactic forms, we could say whether two terms are equal or not, but now we may not be able to do so. This weakening also opens the possibility of having several equality proofs for the same two terms. And these proofs may themselves be equal, which would then be witnessed by some proof in the next higher level. So we see that there may be non-trivial interactions between the levels or dimensions. As such, we need a way to organize this structural information so that we may express the intra and inter-dimensional interactions easily and efficiently. Cubical sets provide us with such a framework where we can express the judgements of higher-dimensional type theory.

3 Cubical sets

Cubical sets can be thought of as sets containing n-dimensional abstract cubes as members. The dimensions are named from a countably infinite set $\mathbb{A}$ of variables x, y, z, etc. The dimensions themselves range over an abstract interval $[0, 1]$. So a 0-cube is a point, an 1-cube is a line over $[0, 1]$, a 2-cube is a square over $[0, 1] \times [0, 1]$ and so on. The 0-cubes can be seen as the ordinary terms of a type, the 1-cubes as paths between those terms, the 2-cubes as higher paths and so on. To understand better, let us look at a 2-cube.



We have dimensions x and y ranging over $[0, 1]$. The four corners represent four ordinary terms a_1, a_2, a_3, a_4 of some type A, the paths p_1, p_2, p_3, p_4 represent equalities between the corresponding terms and the body h is a higher y-path between p_1 and p_4 and a higher x-path between p_2 and p_3. The shape is a square because to have a higher path (h) between two paths (p_1 and p_4), we must first make sure that the corresponding end-points (a_1, a_3 and a_2, a_4) of the paths being compared are themselves equal (as witnessed by p_2 and p_3 respectively). Now this higher path h bundles together the hierarchical information of the structure. For example, substituting 0 and 1 for y in h, we know that h is a higher path from $h\langle 0/y\rangle$ to $h\langle 1/y\rangle$. Further substituting 0 and 1 for x in $h\langle 0/y\rangle$, we know that $h\langle 0/y\rangle$ is a path from $h\langle 0/y\rangle\langle 0/x\rangle$ to $h\langle 0/y\rangle\langle 1/x\rangle$. Similarly, we know that $h\langle 1/y\rangle$ is a path from $h\langle 1/y\rangle\langle 0/x\rangle$ to $h\langle 1/y\rangle\langle 1/x\rangle$. We can repeat our example by first substituting x in lieu of y in h.

We note that by our construction, $h\langle 0/x\rangle\langle 0/y\rangle$ and $h\langle 0/y\rangle\langle 0/x\rangle$ are the same point a_1. In fact, dimension substitution commutes for independent dimension names. Let us look at a few other things we can do with this 2-cube. If we substitute x for y (or y for x) in h, we get an x (or a y) path from a_1 to a_4, a diagonal of the square. We can also view this x, y square as a degenerate x, y, z-cube where $h\langle 0/z\rangle = h$ and $h\langle 1/z\rangle = h$. We may also rename the dimensions from x, y to z_1, z_2 and then we would have a z_1, z_2 cube $h\langle z_1/x\rangle\langle z_2/y\rangle$. Let us now look at these ideas more generally.

Let Ψ, Ψ' be proper subsets of the set of variable names $\mathbb{A}$. Then given a Ψ-cube c_Ψ and a function $f : \Psi' \to \Psi$, we get a Ψ'-cube $c_{\Psi'}$. When $\Psi' \subset \Psi$ and $f : \Psi' \hookrightarrow \Psi$, then $c_{\Psi'}$ is a face of c_Ψ. When $\Psi' \subset \Psi$ but f is not injective, then $c_{\Psi'}$ is a diagonal of c_Ψ. When $\Psi \subset \Psi'$ and $f(x \in \Psi) = x$, then $c_{\Psi'}$ is c_Ψ degenerated to Ψ'. When f is some other injective map, then $c_{\Psi'}$ is c_Ψ with the corresponding renaming of dimension names. These structural operations on dimension names are similar to the ones we have on ordinary variables in type theories. In higher-dimensional type theory, since both types and terms may be higher-dimensional structures, the contexts may contain dimension variables in addition to the type variables.

Once we have the structural framework of cubical sets, we can start populating them with our terms and types. Now homotopy type theory and higher-dimensional type theory are both extensions of Martin-Löf type theory and so before we delve deeper into the terms and types of higher-dimensional type theory, let us take a look at Martin-Löf type theory. In fact, higher-dimensional type theory is essentially higher-dimensional Martin-Löf type theory.

4 Martin-Löf type theory

Martin-Löf, carrying forward the tradition of Brouwer, developed his dependent type theory [ML98] with the aim of formalizing intuitionistic mathematics. The constructive nature of this theory makes it a programming language, in addition to its being an intuitionistic logic. Martin-Löf type theory has been very successful in its aim of formalization of constructive theories. This can be inferred from the fact that several state-of-the-art proof assistants are based on variants of this theory.

4.1 An overview

The four basic judgements of this theory are A is a type, a is a term of type A, types A and B are definitionally equal, a_1 and a_2 are definitionally equal terms of type A. For every construct in his theory, Martin-Löf gives a 'meaning explanation' explaining what the construct means and then he uses the 'meaning explanations' to justify his inference rules. For example, $A \wedge B$ is true means that I have a method to solve A and a method to solve B; now I can justify the inference rule 'From $A \wedge B$ is true, derive A is true' using the above meaning explanation. This is in line with Brouwer's idea that the justification of the mathematical rules is provided by our intuition or understanding. In this theory, for every construct, there are formation, introduction, elimination and equality rules. The formation 'meaning explanation' explains what it means to be a term of a type, the introduction 'meaning explanation' explains how to construct the canonical terms of a type, the elimination 'meaning explanation' explains the induction principle for a type and the equality 'meaning explanation' explains when two canonical terms of a type or two canonical types are definitionally equal.

In Martin-Löf type theory, we have types corresponding to all the connectives of first-order logic. We have product ($\times$) types, sum ($+$) types, function ($\to$) types, dependent function (Π) types and dependent product (Σ) types corresponding to $\wedge, \vee, \Rightarrow, \forall, \exists$ respectively. An interesting feature of this theory is that there are two types of equality, definitional and propositional. Definitional equality is a meta-theoretic equality and is specified by the equality rules. But propositional equality corresponds to identity types within the theory. For any two elements $a_1, a_2 : A$, we have the identity type $\mathrm{Id}(a_1, a_2)$ and we say that a_1 and a_2 are propositionally equal if the type is inhabited. Till now, we have been using the word 'equality' in the sense of propositional equality and we shall continue to do so and explicitly state if we mean otherwise. The identity types have some remarkable features. Research into the identity types led to the groupoid interpretation of this theory and this may be said to be the forerunner of homotopy type theory.

4.2 Groupoid interpretation

In Martin-Löf type theory, a given identity type has, in general, at most one term, reflexivity. But Hofmann and Streicher showed [HS98] that there is a model of the theory where for some type A and some $a_1, a_2 \in A$, $\mathrm{Id}(a_1, a_2)$ has more than one element. This means that identity proofs of two terms need not be unique. Homotopy type theory and higher-dimensional type

theory are both a generalization of this idea in that not only can terms have non-unique identity proofs, but also identity proofs between terms can have non-unique identity proofs between themselves and so on.

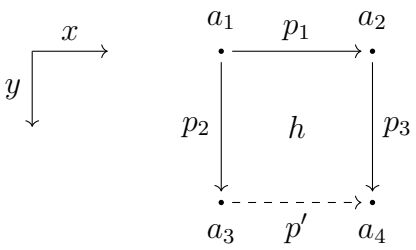
Since we may have non-unique identity proofs now, we need to make sure that they interact in a proper way. Specifically, since equality is an equivalence congruence relation, we must be able to invert, compose and lift equality proofs, in addition to having reflexivity proofs. For example, for any type A and $a_1, a_2, a_3 \in A$ and $p_1 \in \mathrm{Id}(a_1, a_2)$, $p_2 \in \mathrm{Id}(a_2, a_3)$, we know by transitivity that $\mathrm{Id}(a_1, a_3)$ is inhabited, the proof for this needs to be given by $p_1 \circ p_2$, the composition of p_1 and p_2. But now we also have to make sure that these operations themselves are well-behaved. To elaborate, given $p_1 \in \mathrm{Id}(a_1, a_2), p_2 \in \mathrm{Id}(a_2, a_3)$ and $p_3 \in \mathrm{Id}(a_3, a_4)$, we can compose p_1, p_2 and p_3 in two different ways to produce a proof of $\mathrm{Id}(a_1, a_4)$; we expect that the two composites would be equal, i.e. $\exists h, h \in \mathrm{Id}((p_1 \circ p_2) \circ p_3, p_1 \circ (p_2 \circ p_3))$. Similarly, for $p \in \mathrm{Id}(a_1, a_2)$, we would expect that $\exists h_1, h_1 \in \mathrm{Id}(p \circ p^{-1}, r_{a_1})$ and $\exists h_2, h_2 \in \mathrm{Id}(p^{-1} \circ p, r_{a_2})$ where r_a is the reflexivity proof at a. Further $\exists h_3, h_3 \in \mathrm{Id}(p \circ r_{a_2}, p)$ and $\exists h_4, h_4 \in \mathrm{Id}(r_{a_1} \circ p, p)$. These are the associativity, inverse and identity laws for groupoid. We do not need that the laws hold up to definitional equality but only up to propositional equality; the witness being some element at the next higher level. These operations of inversion, composition, etc are not just limited to equality proofs between terms, but we need to define them at each and every level, and then make sure that there are witnesses for the groupoid laws at the next higher level. In short, our types are now weak ∞-groupoids and we need to take care of this fact.

5 Kan condition

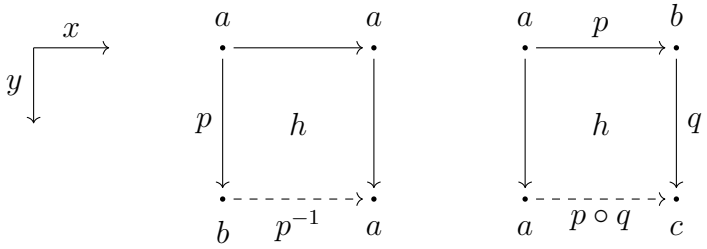
Since we are working with non-unique identity proofs in higher-dimensional type theory, we must define the inverse, composition and lift operations at every dimension and make sure that the groupoid laws hold. It is noteworthy that because of our choice of using cubical sets, we get reflexivity proofs just by weakening the dimension context. For example, given a point $a \in A$ at 0-dimension, we can get a path from a to a just by degenerating to an x-path $a \in A[x]$. Now the inverse and composition operation may involve operands of the same type or different types; i.e. it may either be homogeneous or heterogeneous. Instead of defining both together, we separate them into two operations, one called homogeneous Kan composition and the other called coercion.

5.1 Homogeneous Kan composition

We first consider homogeneous composition. Informally, Kan composition is like putting lid on open boxes. To understand the geometry, let us look at homogeneous Kan composition of equality proofs between terms or 1-cubes.

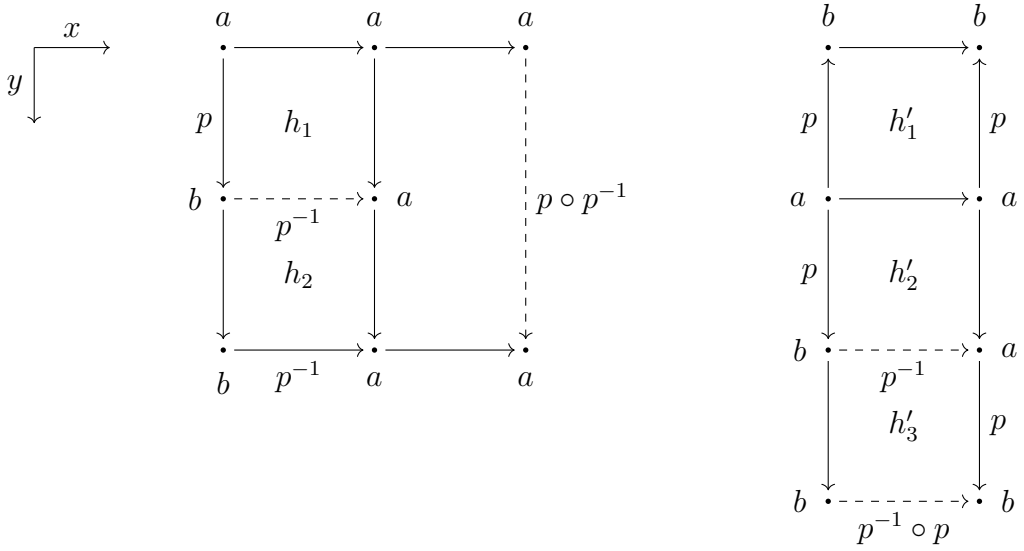


Given paths p_1, p_2, p_3, the composition gives us a path p' (the lid) and also a higher path h (the filler). We can get inverse and composition in the following way. (The paths with no names are reflexivity.)



5.2 Verification of the groupoid laws: inverse

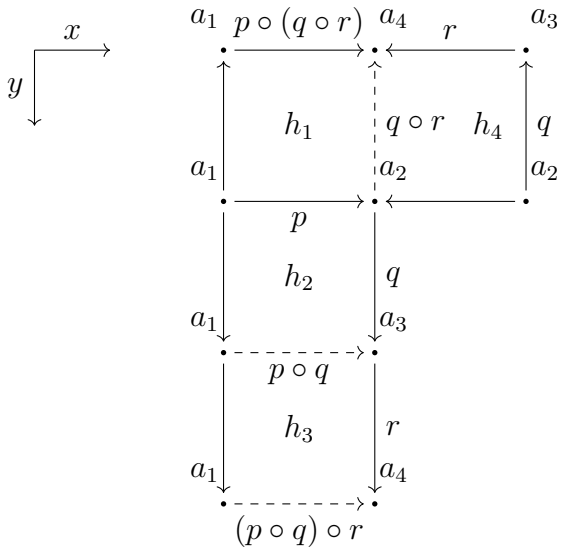
Now let us look at how the groupoid laws are satisfied by this operation. It is easy to see that the identity laws are satisfied, since reflexivity proofs are given just by weakening. So we first look at the inverse laws.



In the first figure, h_1 is for obtaining inverse of p, h_2 is just weakening of p^{-1}, and the bigger square (say h_3) is for composition of p and p^{-1}. Now a proof of $p \circ p^{-1} = r_{a_1}$ is given by $h_3^{-1} \circ (h_1 \circ h_2)$. Similarly in the other figure, h'_2 is for obtaining inverse of p, h'_1 is just weakening of p, and h'_3 is for composition of p^{-1} and p. A proof of $r_{a_2} = p^{-1} \circ p$ is then given by $h_1'^{-1} \circ (h'_2 \circ h'_3)$.

5.3 Verification of the groupoid laws: associativity

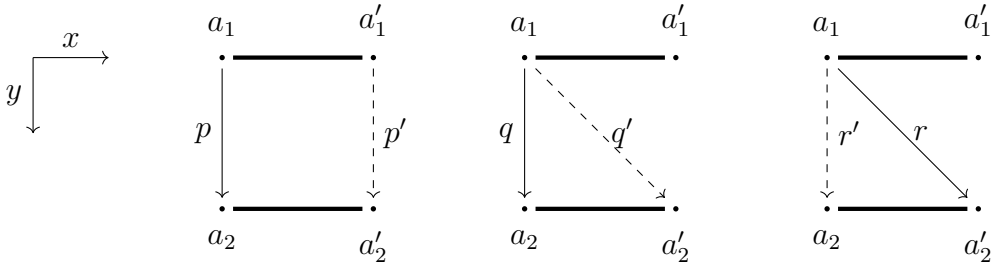
Now let us look at the associativity law.



In the figure above, h_2, h_3 are for obtaining $p \circ q$ and $(p \circ q) \circ r$ respectively, whereas h_4, h_1 are for obtaining $q \circ r$ and $p \circ (q \circ r)$ respectively. Now a proof of $p \circ (q \circ r) = (p \circ q) \circ r$ is given by $h_1^{-1} \circ (h_2 \circ h_3)$. Thus we see that Kan operations allow us to define inverse and composition operations and also makes sure that the groupoid laws hold.

5.4 Coercion

Let us now look at coercion. If we have two equal types and an element of one of them, we would expect that there exists an element in the other type equal to this element. The operation of coercion gets that element, the transport of the first, for us. Two equal types can be represented in our cubical setting as a type-line. So given a type-line and a term, we can coerce it from one dimension to another. Let us look at a few examples.



In the figures above the thick lines represent type-line say $x.A$. Let's suppose $A\langle 0/x\rangle = A_0$ and $A\langle 1/x\rangle = A_1$. Now, in the first figure, p is a path in A_0 and using the type-line as a guide, we can transport from dimension-0 to dimension-1 (along x) to get a path p' in A_1. In the second figure, we coerce q from dimension-0 to dimension-y to get a heterogeneous equality between a_1 and a'_2. In the third figure, we coerce a heterogeneous equality r from

dimension-y to dimension-0 to get a homogeneous equality r' in A_0. So coercion helps us to move back and forth between equal elements of equal types. Now that we have got the basic idea of homogeneous composition and coercion, let's look at their formal syntax.

5.5 Formal presentation

The homogeneous composition is given by $\text{hcom}_A^{\overrightarrow{r_i}}(r \rightsquigarrow r', M; \overrightarrow{y.N_i^\epsilon})$. Here A is a type, $M \in A$ is called the cap of the composition, $\overrightarrow{r_i}$ given by a list of dimension terms $r_1, r_2, \ldots r_n$ is called the extent of the composition, $\overrightarrow{y.N_i^\epsilon}$ is a list of faces forming the tube of the composition with $y.N_i^\epsilon$ being the face corresponding to the equation $r_i = \epsilon$ (where $\epsilon = \{0, 1\}$) and r, r' are the starting and ending dimensions respectively. In the case of our example, $p' = \text{hcom}_A^x(0 \rightsquigarrow 1, p_1; y.p_2, y.p_3)$ and $h = \text{hcom}_A^x(0 \rightsquigarrow y, p_1; y.p_2, y.p_3)$. The coercion operation is given by $\text{coe}_{x.A}^{r \rightsquigarrow r'}(M)$, where $x.A$ is an x type-line in A and M is coerced from dimension r to r' along $x.A$. We can say a few things about hcom and coe already. The trivial case when $r = r'$ in hcom or coe, we get back M. And when some of the r_i in the extent is ϵ, the hcom operation gives us $N_i^\epsilon\langle r'/y\rangle$. Once we have motivated ourselves for these extra constructs of higher-dimensional type theory, let us look at the formal syntax and semantics of the language we shall consider here.

6 Syntax and operational semantics

The terms of the language are given below.

$$
\begin{aligned}
M := {} & (a : A) \to B \mid (a : A) \times B \mid \text{Id}_{x.A}(M, N) \mid \lambda a.M \mid \text{app}(M, N) \mid \langle M, N\rangle \mid \pi_1(M) \\
& \mid \pi_2(M) \mid \text{inl}(M) \mid \text{inr}(M) \mid (\text{match } M \text{ with inl } a \Rightarrow M | \text{inr } a \Rightarrow M) \mid \langle x\rangle M \\
& \mid M@r \mid \mathbb{S}^1 \mid \text{base} \mid \text{loop}_r \mid \mathbb{S}^1\text{-elim}_{a.A}(M; N, x.N_2) \mid \text{hcom}_A^{\overrightarrow{r_i}}(r \rightsquigarrow r', M; \overrightarrow{y.N_i^\epsilon}) \\
& \mid \text{coe}_{x.A}^{r \rightsquigarrow r'}(M)
\end{aligned}
$$

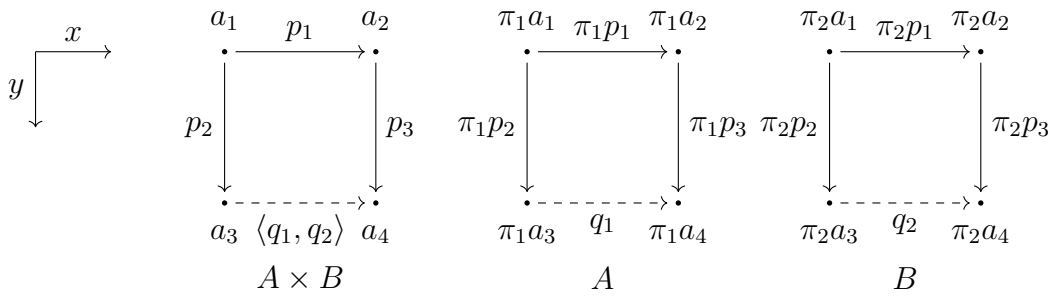
In addition to the terms of ordinary type-theory, we have dimension abstraction $\langle x\rangle M$ and dimension application $M@r$. We also have the higher inductive type circle $\mathbb{S}^1$, which has a single 0-dimensional term base and a path loop_r from base to itself. Now let us look at the operational semantics of this language.

As we have discussed earlier, the interesting feature of higher-dimensional type theory is the Kan condition, i.e. hcom and coe. In fact, the small-step semantics for other ordinary type-theoretic terms remain almost the same and so we skip them and focus on just hcom and coe. In this section, we look at the stepping behaviour of hcom and coe at different types.

6.1 Product types

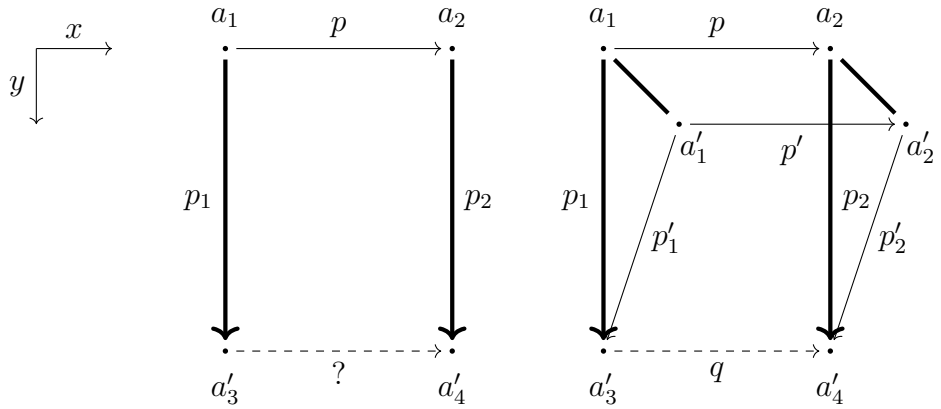
Before looking at hcom for dependent products, let's look at hcom for ordinary products. Equality at product types is component-wise equality. For example, $M = N \in A \times B$, if $\pi_1(M) = \pi_1(N) \in A$ and $\pi_2(M) = \pi_2(N) \in B$. So equality proofs between terms of product

types are pairs of equality proofs at the component types. The hcom operation at product types then needs to compose the proofs pair-wise. The figure below shows an example.



Here $q_1 = \text{hcom}^x_A(0 \rightsquigarrow 1, \pi_1 p_1; y.\pi_1 p_2, y.\pi_1 p_3)$ and $q_2 = \text{hcom}^x_B(0 \rightsquigarrow 1, \pi_2 p_1; y.\pi_2 p_2, y.\pi_2 p_3)$.
Generalizing this, we have,
$\text{hcom}^{\overrightarrow{r_i}}_{A \times B}(r \rightsquigarrow r', M, \overrightarrow{y.N_i^\epsilon}) \longmapsto \langle \text{hcom}^{\overrightarrow{r_i}}_A(r \rightsquigarrow r', \pi_1 M, \overrightarrow{y.\pi_1 N_i^\epsilon}), \text{hcom}^{\overrightarrow{r_i}}_B(r \rightsquigarrow r', \pi_2 M, \overrightarrow{y.\pi_2 N_i^\epsilon}) \rangle$.
The coercion operation also behaves similarly.
$\text{coe}^{r \rightsquigarrow r'}_{x.A \times B}(M) \longmapsto \langle \text{coe}^{r \rightsquigarrow r'}_{x.A}(\pi_1 M), \text{coe}^{r \rightsquigarrow r'}_{x.B}(\pi_2 M) \rangle$.

Now let us look at hcom and coe at dependent product types. We can't use the rules above for dependent products because in this case the second projection may give us elements of different types, for example $\pi_2 a_1$ and $\pi_2 a_3$ may belong to different types. So in order to define hcom at dependent product types, we need to generalize our Kan composition to account for heterogeneous equality as well and for this we use coercion. To motivate ourselves, let us look at an example.



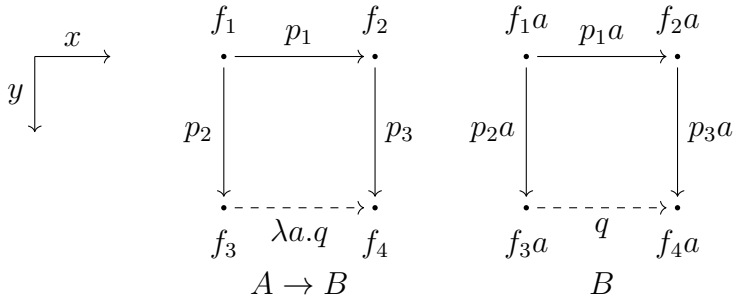
In the figures above, the thick directed lines represent heterogeneous equality while the undirected thick lines are type-lines. Let $y.A$ be a type-line and let $A\langle 0/y \rangle = A_0$ and $A\langle 1/y \rangle = A_1$. Now p is a path in A_0 and p_1, p_2 are heterogeneous equality paths. We want to compose p, p_1, p_2. For this, we first need to coerce p from dimension-0 to dimension-1 along the type-line and then we need to coerce p_1, p_2 from dimension-y to dimension-1 and then compose p', p'_1, p'_2 at A_1 to get $q = \text{hcom}^x_{A_1}(0 \rightsquigarrow 1, p'; y.p'_1, y.p'_2)$. The general composition com is given by:
$\text{com}^{\overrightarrow{r_i}}_{y.A}(r \rightsquigarrow r', M; \overrightarrow{y.N_i^\epsilon}) := \text{hcom}^{\overrightarrow{r_i}}_{A\langle r'/y \rangle}(r \rightsquigarrow r', \text{coe}^{r \rightsquigarrow r'}_{y.A}(M); \overrightarrow{y.\text{coe}^{y \rightsquigarrow r'}_{y.A}(N_i^\epsilon)})$.

Let us now continue with our discussion on hcom and coe for dependent product types. The composition for the first component remains the same, but since the composition for the second component involves heterogeneous equality, we use com. But for that we need a type-line to serve as a proof that we indeed intend to connect elements of equal types. The proof is given by the homogeneous composite from the first component. Formally, $\mathrm{hcom}^{\overrightarrow{r_i}}_{(a:A)\times B}(r \rightsquigarrow r', M, \overrightarrow{y.N_i^\epsilon}) \longmapsto \langle \mathrm{hcom}^{\overrightarrow{r_i}}_{A}(r \rightsquigarrow r', \pi_1 M, \overrightarrow{y.\pi_1 N_i^\epsilon}), \mathrm{com}^{\overrightarrow{r_i}}_{z.B[F/a]}(r \rightsquigarrow r', \pi_2 M, \overrightarrow{y.\pi_2 N_i^\epsilon})\rangle$, where $F = \mathrm{hcom}^{\overrightarrow{r_i}}_{A}(r \rightsquigarrow z, \pi_1 M, \overrightarrow{y.\pi_1 N_i^\epsilon})$.
Similarly, the coercion operation is given by
$\mathrm{coe}^{r\rightsquigarrow r'}_{x.(a:A)\times B}(M) \longmapsto \langle \mathrm{coe}^{r\rightsquigarrow r'}_{x.A}(\pi_1 M), \mathrm{coe}^{r\rightsquigarrow r'}_{x.B[F'/a]}(\pi_2 M)\rangle$, where $F' = \mathrm{coe}^{r\rightsquigarrow x}_{x.A}(\pi_1 M)$.

Now let us look at hcom and coe at function types.

6.2 Function types

First we look at ordinary functions and then at dependent functions. Equality at function types is extensional equality. Two functions are equal if they produce equal outputs on equal arguments. So the hcom operation at function types will be a function that, given an argument, produces equality proofs of the results (at that argument) of the functions being compared. We look at an example below.



Generalizing this, we have,
$\mathrm{hcom}^{\overrightarrow{r_i}}_{A\to B}(r \rightsquigarrow r', M, \overrightarrow{y.N_i^\epsilon}) \longmapsto \lambda a.\mathrm{hcom}^{\overrightarrow{r_i}}_{B}(r \rightsquigarrow r', \mathrm{app}(M, a); \overrightarrow{y.\mathrm{app}(N_i^\epsilon, a)})$. The coercion operation is interesting here because defining it like $\mathrm{coe}^{r\rightsquigarrow r'}_{x.A\to B} \longmapsto \lambda a.\mathrm{coe}^{r\rightsquigarrow r'}_{x.B}(\mathrm{app}(M, a))$ would not be correct. This is so because a is in dimension r' while M takes elements of dimension r as arguments; so we need to coerce a in the other direction and then apply M to it. So the right way would be $\mathrm{coe}^{r\rightsquigarrow r'}_{x.A\to B} \longmapsto \lambda a.\mathrm{coe}^{r\rightsquigarrow r'}_{x.B}(\mathrm{app}(M, \mathrm{coe}^{r'\rightsquigarrow r}_{x.A}(a)))$. Let us now look at hcom and coe for dependent functions.

Here even with dependence, the hcom operation remains the same because we have already fixed an a and so we just need homogeneous composition in type, say $B[a]$. The coercion operation changes a bit because now instead of just having the type B, we have types $B[a]$ and $B[\mathrm{coe}^{r'\rightsquigarrow r}_{x.A}(a)]$. So we need an explicit proof of their equality. The coercion operation is given by: $\mathrm{coe}^{r\rightsquigarrow r'}_{x.(a:A)\to B} \longmapsto \lambda a.\mathrm{coe}^{r\rightsquigarrow r'}_{x.B[F/a]}(\mathrm{app}(M, \mathrm{coe}^{r'\rightsquigarrow r}_{x.A}(a)))$, where $F = \mathrm{coe}^{r'\rightsquigarrow x}_{x.A}(a)$.

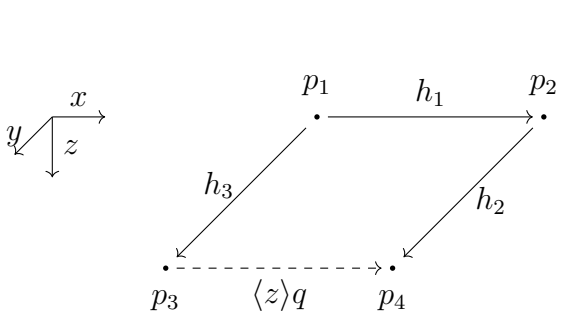
Let us now look at hcom and coe for disjoint sum types.

6.3 Disjoint sum types

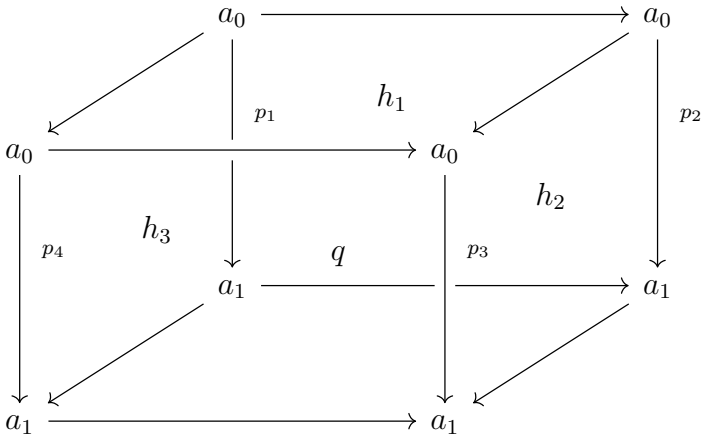
Equality at disjoint sum types is equality at either of the types. Elements $a_1, a_2 \in A + B$ are equal if they are either equal elements of A or equal elements of B. So the hcom operation at disjoint sum types is a match expression that uses the hcom operation of the base types. Formally, $\text{hcom}^{\overrightarrow{r_i}}_{A+B}(r \rightsquigarrow r', M; \overrightarrow{y.N_i^\epsilon}) \longmapsto$ match M, N_i^ϵ with
$\text{inl } M', (\text{inl } N')_i^\epsilon \Rightarrow \text{ inl } (\text{hcom}^{\overrightarrow{r_i}}_{A}(r \rightsquigarrow r', M'; \overrightarrow{y.N_i'^\epsilon}))$
$\mid \text{inr } M', (\text{inr } N')_i^\epsilon \Rightarrow \text{inr } (\text{hcom}^{\overrightarrow{r_i}}_{B}(r \rightsquigarrow r', M'; \overrightarrow{y.N_i'^\epsilon}))$.
Since we do not have equality between inl _ and inr _ , whenever M is inl _ , N_i^ϵ must be inl _ because their faces must fit together and this will be made sure by the typing rules. The coercion operation can also be defined similarly.
$\text{coe}^{r \rightsquigarrow r'}_{x.A+B}(M) \longmapsto$ match M with
$\text{inl } M' \Rightarrow \text{ inl } (\text{coe}^{r \rightsquigarrow r'}_{x.A} M')$
$\mid \text{inr } M' \Rightarrow \text{ inr } (\text{coe}^{r \rightsquigarrow r'}_{x.B} M')$
So we have looked at hcom and coe for $\wedge$ (product), $\vee$(disjoint sum), $\Rightarrow$(function), $\forall$(dependent function), $\exists$(dependent product). Now let us look at the small-step semantics for terms of identity types and the circle $\mathbb{S}^1$.

6.4 Identity types

As we have discussed earlier that two elements are equal if their identity set is inhabited. Similarly, two elements of some identity set are equal if their higher identity set is inhabited. So equality at identity types is the existence of an element in the higher identity type. Now elements of identity types are dimension-abstracted lines, where the abstraction contains information about the hierarchical structure of the end-points. To compose them, we need to unroll this information a bit and see that everything fits together, then compose them at a higher dimension and abstract back. Let us look at an example.



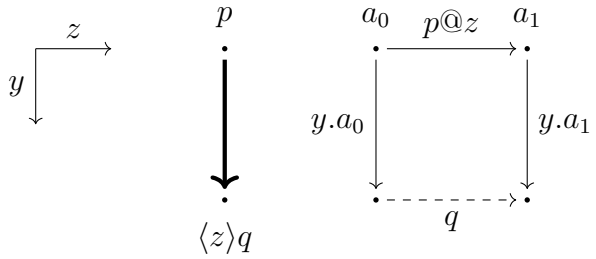
$\text{Id}_{z.A}(a_0, a_1)$



In the first figure above, h_1, h_2, h_3 are higher paths between paths p_1, p_2, p_3, p_4, which are themselves elements of the heterogeneous equality type $\text{Id}_{z.A}(a_0, a_1)$. Let $A\langle 0/z \rangle = A_0$

and $A\langle 1/z\rangle = A_1$. So p_1, p_2, p_3, p_4 are z-lines from A_0 to A_1 and h_1, h_2, h_3 are $\_, z$ squares. Then h_1, h_2, h_3 when applied to z give us three squares that form a side-laid trough, as we see in the second figure. The top and bottom faces of the cube in A_0 and A_1 respectively can be filled by reflexivity. So we have an open box and we can get its lid by hcom. Note that we do not need general composition com because our dimension context has been extended to include the dimension variable z. After taking the hcom q, we can abstract over z to get our desired composite. Generalizing this example, we have, $\mathrm{hcom}^{\overrightarrow{r_i}}_{\mathrm{Id}_{z.A}(a_0,a_1)}(r \rightsquigarrow r', M; \overrightarrow{y.N_i^\epsilon}) \longmapsto \langle z\rangle \mathrm{hcom}^{\overrightarrow{r_i},z}_{A}(r \rightsquigarrow r', M@z; \overrightarrow{y.N_i^\epsilon@z}, \_.a_0, \_.a_1)$, where $\_.a_0, \_.a_1$ are the appropriate analogues of our reflexivity squares.

Now we look at coercion for identity types. For that, we consider the following example. Let $p \in \mathrm{Id}_{z.A}(a_0, a_1)$ and we want to coerce p from 0 to 1 along the type-line $y.\mathrm{Id}_{z.A}(a_0, a_1)$. Our coercion will give us an element of $\mathrm{Id}_{z.A\langle 1/y\rangle}(a_0\langle 1/y\rangle, a_1\langle 1/y\rangle)$. To get that element, we need to unroll p, then compose it with the y-lines $y.a_0, y.a_1$ and then abstract back. This time the composition operation is com, because of the y type-line.



$y.\mathrm{Id}_{z.A}(a_0, a_1) \qquad y.A$

Formally we have $\mathrm{coe}^{r\rightsquigarrow r'}_{y.\mathrm{Id}_{z.A}(a_0,a_1)}(M) \longmapsto \langle z\rangle \mathrm{com}^{z}_{y.A}(r \rightsquigarrow r', M@z; y.a_0, y.a_1)$.
The other stepping relations for identity terms are that if $M \longmapsto M'$, then $M@r \longmapsto M'@r$ and that $(\langle x\rangle M)@r \longmapsto M\langle r/x\rangle$. Here $\langle x\rangle M$ is a value.

Now let us look at $\mathbb{S}^1$.

6.5 Circle

First let us look at the simple stepping rules. We have $\mathrm{loop}_\epsilon \longmapsto \mathrm{base}$. This is so because loop is a path from base to base. Then $\mathbb{S}^1\text{-elim}_{a.A}(M; P, x.L)$ steps to $\mathbb{S}^1\text{-elim}_{a.A}(M'; P, x.L)$ whenever M steps to M'. When $M = \mathrm{base}$, we get P and when $M = \mathrm{loop}_r$, we get $L\langle r/x\rangle$. The coercion operation is trivial here because there is just a single point, base. Now we look at how the eliminator interacts with the hcom operation. Before considering the dependent eliminator, let's look at the ordinary one. We have,
$\mathbb{S}^1\text{-elim}_A(\mathrm{hcom}^{x_1,x_2,\ldots,x_n}_{\mathbb{S}^1}(r \rightsquigarrow r', M; \overrightarrow{y.N_i^\epsilon}); P, x.L) \longmapsto$
$\mathrm{hcom}^{x_1,x_2,\ldots,x_n}_{A}(r \rightsquigarrow r', \mathbb{S}^1\text{-elim}_A(M; P, x.L); \overrightarrow{y.\mathbb{S}^1\text{-elim}_A(N_i^\epsilon; P, x.L)})$.
Intuitively, this means that the eliminator commutes with hcom. For the dependent case, we use com instead of hcom, i.e.
$\mathbb{S}^1\text{-elim}_{a.A}(\mathrm{hcom}^{x_1,x_2,\ldots,x_n}_{\mathbb{S}^1}(r \rightsquigarrow r', M; \overrightarrow{y.N_i^\epsilon}); P, x.L) \longmapsto$

$\mathrm{com}^{x_1,x_2,\ldots,x_n}_{z.A[F/a]}(r \rightsquigarrow r', \mathbb{S}^1\text{-elim}_{a.A}(M; P, x.L); \overrightarrow{y.\mathbb{S}^1\text{-elim}_{a.A}(N_i^\epsilon; P, x.L)})$, where
$F = \mathrm{hcom}^{x_1,x_2,\ldots,x_n}_{\mathbb{S}^1}(r \rightsquigarrow z, M; \overrightarrow{y.N_i^\epsilon})$.
The values here are base, loop_r and $\mathrm{hcom}^{\overrightarrow{r_i}}_{\mathbb{S}^1}(r \rightsquigarrow r', M; \overrightarrow{y.N_i^\epsilon})$ (where $r \neq r'$).

This completes our discussion of operational semantics and we now look at what it means to be a cubical type.

7 Cubical types

The type system specifies which syntactic forms we want to assign meaning to (and as by-product which forms we don't want to assign any meaning). It is a mechanism to separate the meaningful syntactic forms from the meaningless ill-formed ones. In our higher-dimensional setting, we first need to make sure that our types respect the hierarchical nature of their terms. For example, if a type at Ψ, say A, has an element M, then $A\psi$ must contain the element $M\psi$ for any $\psi : \Psi' \to \Psi$. In addition, we would expect that evaluation and dimension substitution can be interleaved arbitrarily. Entities that satisfy this basic structural requirement are called cubical pretypes.

7.1 Pretypes

Formally, let's consider a dimension-stratified binary equality relation $\_ \approx^\Psi \_$ over values in Ψ. The relation is symmetric and transitive. And it is reflexive over canonical values. For values A_0, B_0, if $A_0 \approx^\Psi B_0$, then the value-stratified relations $\_ \approx^\Psi_{A_0} \_$ and $\_ \approx^\Psi_{B_0} \_$ are also equal, i.e. $M_0 \approx^\Psi_{A_0} N_0$ if and only if $M_0 \approx^\Psi_{B_0} N_0$. This gives us a cubical type system which we can populate with our pretypes. Let's formalize the notion of a pretype.
We say that A and B are equal pretypes in Ψ, written $A = B$ pretype $[\Psi]$, when for $\psi_1 : \Psi_1 \to \Psi, \psi_2 : \Psi_2 \to \Psi_1$, and

$$\begin{aligned} \text{given } & A\psi_1 \Downarrow A_1, A_1\psi_2 \Downarrow A_2, (A\psi_1)\psi_2 \Downarrow A_{12}, \\ & B\psi_1 \Downarrow B_1, B_1\psi_2 \Downarrow B_2, (B\psi_1)\psi_2 \Downarrow B_{12} \\ \text{then } & A_2 \approx^{\Psi_2} A_{12} \approx^{\Psi_2} B_2 \approx^{\Psi_2} B_{12}. \end{aligned} \tag{1}$$

Similarly, we can define equal elements of pretypes. We say $M = N \in A[\Psi]$, when for $\psi_1 : \Psi_1 \to \Psi, \psi_2 : \Psi_2 \to \Psi_1$, and

$$\begin{aligned} \text{given } & M\psi_1 \Downarrow M_1, M_1\psi_2 \Downarrow M_2, (M\psi_1)\psi_2 \Downarrow M_{12}, \\ & N\psi_1 \Downarrow N_1, N_1\psi_2 \Downarrow N_2, (N\psi_1)\psi_2 \Downarrow N_{12} \\ \text{then } & M_2 \approx^{\Psi_2}_{A_{12}} M_{12} \approx^{\Psi_2}_{A_{12}} N_2 \approx^{\Psi_2}_{A_{12}} N_{12}. \end{aligned} \tag{2}$$

The conditions above check that any pair of dimension substitution commute with evaluation. We call a pretype A cubical, if for any $\psi : \Psi' \to \Psi$,

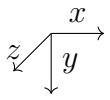
$$\begin{aligned} \text{given } & A\psi \Downarrow A_0 \text{ and } M \approx^{\Psi'}_{A_0} N, \\ \text{then } & M = N \in A\psi[\Psi']. \end{aligned} \tag{3}$$

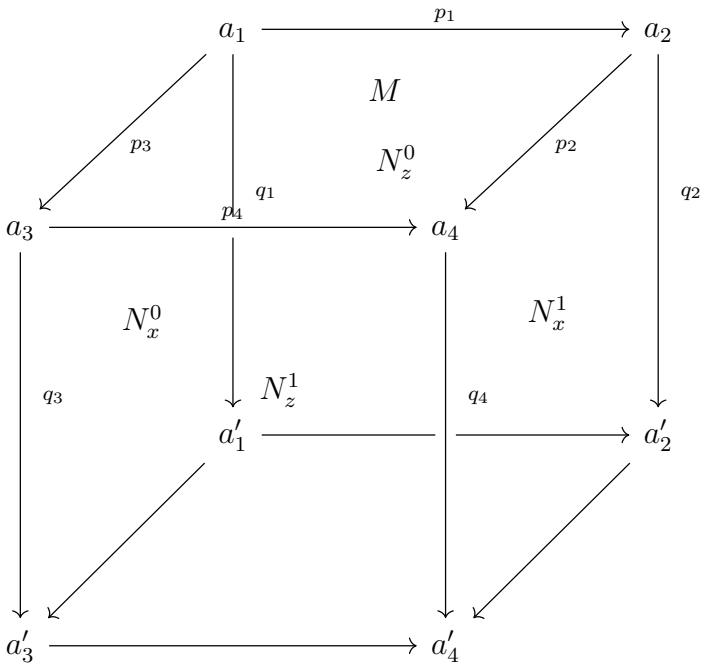
Put in other words, a pretype is cubical if the evaluation of a term in that pretype can be arbitrarily interleaved with dimension substitution. Note that we use the judgements $A = A$ pretype $[\Psi]$ and $M = M \in A[\Psi]$ in lieu of A pretype $[\Psi]$ and $M \in A[\Psi]$ respectively.

Now in higher-dimensional type theory, in addition to the normal type-theoretic operations, we have two special ones, hcom and coe. We need to ensure that whenever hcom and coe terms are well-defined, they respect the equality relation and behave as expected. We now look at these two conditions.

7.2 Equally Kan

To motivate ourselves, let's begin with an example. Let's say we want to compute $\mathrm{hcom}_A^{x,z}(0 \rightsquigarrow y, M; y.N_x^0, y.N_x^1, y.N_z^0, y.N_z^1)$. With respect to the figure below, we want to compute the filler of the cube. But for the hcom operation to be meaningful, the p_i's must match, i.e. the lines forming the boundary of M must match with the corresponding lines forming the boundaries of the tube faces. In addition, the q_i's must match, i.e. the corresponding boundaries of the tube faces must match. Once these are ensured, we can carry out the composition to get the filler.





Now that we have the filler, we must also make sure that what we have got is in accordance with our intuitive understanding of the composition. In other words, the filler should be the given cube whose topmost face matches with M and whose side-faces match with the N_i^ϵ's.

To ensure this, we would require our hcom operation to return the cap when the starting and the ending dimension are the same. And return the appropriate edge of the appropriate tube face when one of the dimensions in the extent is an ϵ. The latter also ensures that the edges of the composite match with the corresponding edges of the tube faces. In addition to all this, our hcom operation must also respect the equality relation.

We now look at the formal presentation. To express them succinctly, we use dimension context restrictions whereby we restrict our dimension context using equations. For example $[x, y, z|x = 0]$ would refer to the y, z face (at $x = 0$) of the terms and types in this restricted context. The conditions follow.

$$\begin{aligned}
\text{Given } A = B \text{ pretype } [\Psi] \\
\psi : \Psi' \to \Psi \\
M = O \in A\psi[\Psi'] \\
(\forall i, \epsilon) \quad N_i^\epsilon = P_i^\epsilon \in A\psi[\Psi', y|r_i = \epsilon] \\
(\forall i, j, \epsilon, \epsilon') \quad N_i^\epsilon = N_j^{\epsilon'} \in A\psi[\Psi', y|r_i = \epsilon, r_j = \epsilon'] \\
(\forall i, \epsilon) \quad N_i^\epsilon\langle r/y\rangle = M \in A\psi[\Psi'|r_i = \epsilon], \\
\text{then } \mathrm{hcom}^{\overrightarrow{r_i}}_{A\psi}(r \rightsquigarrow r', M; \overrightarrow{y.N_i^\epsilon}) = \mathrm{hcom}^{\overrightarrow{r_i}}_{B\psi}(r \rightsquigarrow r', O; \overrightarrow{y.P_i^\epsilon}) \in A\psi[\Psi'] \\
\text{and } \mathrm{hcom}^{\overrightarrow{r_i}}_{A\psi}(r \rightsquigarrow r, M; \overrightarrow{y.N_i^\epsilon}) = M \in A\psi[\Psi'] \\
\text{and } \mathrm{hcom}^{\overrightarrow{r_1, r_2, \ldots, r_{i-1}, \epsilon, r_{i+1}, \ldots, r_n}}_{A\psi}(r \rightsquigarrow r', M; \overrightarrow{y.N_i^\epsilon}) = N_i^\epsilon\langle r'/y\rangle \in A\psi[\Psi']
\end{aligned} \tag{4}$$

Similarly coe should respect equality and coerce to the same element if the starting and ending dimensions are the same. Formally,

$$\begin{aligned}
\text{Given } A = B \text{ pretype } [\Psi] \\
\psi : (\Psi', x) \to \Psi \\
M = N \in A\psi\langle r/x\rangle[\Psi'], \\
\text{then } \mathrm{coe}^{r \rightsquigarrow r'}_{x.A\psi}(M) = \mathrm{coe}^{r \rightsquigarrow r'}_{x.B\psi}(N) \in A\psi\langle r'/x\rangle[\Psi'] \\
\text{and } \mathrm{coe}^{r \rightsquigarrow r}_{x.A\psi}(M) = M \in A\psi\langle r/x\rangle[\Psi']
\end{aligned} \tag{5}$$

Two equal pretypes are said to be equally Kan if they satisfy conditions (4) and (5). And two equal pretypes are equal types if they are cubical and equally Kan. The judgement that A is a type in Ψ is expressed as $A = A$ type $[\Psi]$. So we see that to be a type in this higher-dimensional setting, all these conditions must be satisfied. In other words, every type in a cubical type system satisfies these conditions.

Now we look at the types in our language.

8 Types

In our higher-dimensional setting, types are Ψ-dimensional cubes acting as classifiers of other Ψ-dimensional cubes (their terms). Here types are more like specifications of their terms as

the latter evaluate, rather than being a predefined entity. So what we are essentially doing while defining a type A is that, we are collecting a set of Ψ-dimensional cubes for each Ψ, and then defining a partial equivalence relation between them and then checking that A is a pretype, cubical and Kan.

The types we are considering are dependent product, dependent function, disjoint sum, identity and circle, along with their equality relations. We have already discussed the equality relation at each type in section (6). Then to show that they are indeed types, for each type former, we first need to show that it is a pretype as defined in (1), then we need to prove the introduction, elimination, β and η rules according to (2), then prove that it is cubical as defined in (3) and finally show that it is Kan as defined in (4) and (5). We do not carry out this development here, as it is quite lengthy and require the development of several subsidiary lemmas.

With this, we conclude our discussion of the language.

9 Conclusion

The aim of this project has been to understand the basics of computational higher-dimensional type theory. This is an area in the making. In this essay, we looked at a few of its aspects. Throughout this essay, we tried to develop a geometrical intuition of looking at higher-dimensional constructs. We observed that equality endows types and terms with richer and more complex structures. The price we pay for this added complexity seems reasonable considering the fact that the equality relation at each type is now hard-coded once and for all, making it possible to simply work modulo this relation. This hard-coding might make our later work more manageable than that with the setoid approach, where equality is loosely coded. The systematic treatment of equality in our case also makes it easier to express complex interactions between equalities. And while doing all this, we never compromised with the constructive nature of the theory. If homotopy type theory indeed develops into a foundation for mathematics, there is prospect of using computational higher-dimensional type theory to formalize all of mathematics on proof-assistants. But for such a thing to happen, computational higher-dimensional type theory first needs to be developed into a full-fledged programming language. Towards this end, it is just the beginning. As we can see, this theory, as of now, does not have several basic programming constructs like recursion, universes, etc. Developing these is the way forward.

References

[AHW17] Carlo Angiuli, Robert Harper, and Todd Wilson. Computational higher dimensional type theory. In *Proceedings of the 44th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '17, New York, NY, USA, 2017. ACM.

[HS98] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory.

In *Twenty-five years of constructive type theory (Venice, 1995)*, volume 36 of *Oxford Logic Guides*, pages 83–111. Oxford Univ. Press, New York, 1998.

[ML98] Per Martin-Löf. An intuitionistic theory of types. In *Twenty-five years of constructive type theory (Venice, 1995)*, volume 36 of *Oxford Logic Guides*, pages 127–172. Oxford Univ. Press, New York, 1998.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.