SEMI-SIMPLICIAL TYPES

Astra Kolomatskaia (she/her)

20 November 2022

AN INTRODUCTION TO THE PROBLEM

Let Δ^n denote the standard simplex in $\mathbb{R}^n$ – this is defined to be the convex hull of standard basis elements $e_1, \ldots, e_n$ together with the point $e_0 = \mathbf{0}$. Δ^0 is a single point, Δ^1 is a line, Δ^2 is a triangle, Δ^3 is a tetrahedron, etc.

Given a topological space X, we can define $S[X]_n$ to be the collection of maps $\Delta^n \to X$. Thus $S[X]_0$ represents the points in X, $S[X]_1$ represents the lines in X, and $S[X]_2$ represents the triangles in X, etc.

The collections $S[X]_n$ are related by by **face maps** that encode information about the boundary of a simplex. We define affine maps $\delta_k : \Delta^{n-1} \to \Delta^n$, for for $k \in \{0, \ldots, n\}$, by mapping the points $e_0 < e_1 < \ldots < e_{n-1}$ in an order preserving way to $e_0 < e_1 < \ldots < e_n$, with the element e_k being excluded, and extending affinely. For $\mathfrak{s} : S[X]_n$, $n > 0$, we define $\partial_k\, \mathfrak{s} = \mathfrak{s} \circ \delta_k : S[X]_{n-1}$. We get thus get maps:

$$S[X]_0 \;\; S[X]_1 \;\; S[X]_2 \;\; S[X]_3 \quad \ldots$$

This quickly becomes useful for doing homotopy theory! Using the data provided by the face maps, we can talk about simplices in X with given boundary. Then, for example, given a map on the boundary of Δ^n to X, we can ask if this map is nullhomotopic, i.e. admits a filler.

The ∂_k satisfy certain identities. We can think of each ∂_k as deleting a vertex from a simplex (by passing to the opposite face). Any lower dimensional face is obtained through some sequence of vertex deletions, and we want to assert that the order of these deletions does not matter. For example, suppose that $k < l$. Then the codimension 2 face of an n-simplex that contains neither the k-th nor the l-th vertex admits two descriptions: It is the k-th face of the $(n-1)$-simplex that arises as the l-th face of the original n-simplex. Or it is the $(l-1)$-st face of the $(n-1)$-simplex that arises as the k-th face of the original n-simplex. Thus:

$$\partial_k \circ \partial_l = \partial_{l-1} \circ \partial_k \quad \text{for } k < l$$

This condition is sufficient to show that there is a unique way of removing any particular set of veritices from a simplex, as the above rule lets us sort repeated application of ∂ to one in which one deletes the vertices in decreasing order.

The collection $S[X]_\bullet$, with the information of its face maps, is commonly known as the *singular semi-simplicial set* of X. This terminology suggests that

each $S[X]_n$ is usually thought of as a set, as opposed to a space. However, if we think of the collections $\Delta^n \to X$ as spaces with the compact-open topology (which will be well behaved since Δ^n is an exceptionally nice space), then the face maps ∂_k are continuous, so this whole story makes sense in the world of topological spaces as well.

Now, instead of talking about points, lines, and triangles in some space X, we want to move to abstractly talking about some uninterpreted notion of points, lines, and triangles. For the moment, lets work in the world of sets. A **semi-simplicial set** consists of sets X_n for $n \geqslant 0$, along with face maps $\partial_k : X_n \to X_{n-1}$, for $k \in \{0, \ldots, n\}$, satisfying the face relations above.

Note that there are lots of semi-simplicial sets that do not arise as the singular semi-simplicial set of some space X. For example, any singular semi-simplicial set will have a symmetry between lines from x to y and lines from y to x, whereas this condition is not enforced in the definition of an arbitrary semi-simplicial set. Further, the singular semi-simplicial sets are in fact **simplicial sets** in that they admit a notion of degeneracies that arise from maps $\Delta^{n+1} \to \Delta^n$ which collapse two order adjacent verticies. When given abstractly, these degeneracy maps have to satisfy some conditions of their own, as well as conditions giving the boundary of a degenerate simplex. Together, these are called the *simplicial identities*. In general, degeneracies are important because they allow one to talk about boundary preserving homotopies between simplices.

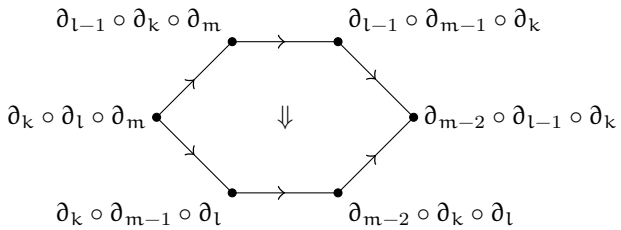
Our goal is to consider semi-simplicial objects in the context of spaces, or, more specifically, in the context of homotopy type theory. The first thing to note is that strict pointwise equality of functions is often the wrong notion to use when talking about spaces; one should instead talk about maps being homotopic. Thus, a semi-simplicial object in the homotopy category of spaces consists of spaces X_n with (continuous) maps ∂_k along with homotopies:

$$\alpha_{k,l} : \partial_k \circ \partial_l \simeq \partial_{l-1} \circ \partial_k \quad \text{for } k < l$$

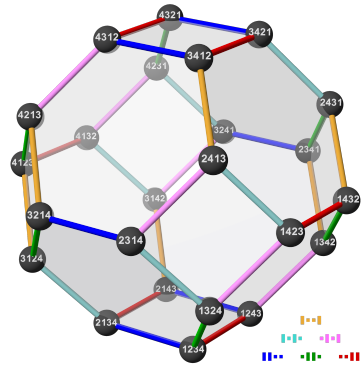
However, this is generally not satisfactory, as for $k < l < m$, we can prove that $\partial_k \circ \partial_l \circ \partial_m \simeq \partial_{m-2} \circ \partial_{l-1} \circ \partial_k$ in two different ways. We require that:

$$\alpha_{k,l} \star \partial_m \cdot \partial_{l-1} \star \alpha_{k,m} \cdot \alpha_{l-1,m-1} \star \partial_k \simeq \partial_k \star \alpha_{l,m} \cdot \alpha_{k,m-1} \star \partial_l \cdot \partial_{m-2} \star \alpha_{k,l}$$

Which we can picture as follows:



However, we're not done yet, because there are new identities for quadruples of deletions. The first identity that comes up is represented by square diagram and says that α homotopies applied to non-interacting indices commute. The second identity is given one dimension up, and describes a filler for the following figure (called a permutohedron), whose faces are the previously mentioned hexagons and squares.



Writing down a formula for this is prohibitively complicated, and things only become worse when you consider sequences of five or more deletions! However, note that, whatever all of these conditions may be, in the case of the singular semi-simplicial object associated to a space X, all of these coherences can be taken to be identity homotopies, because the results of applying iterated boundaries in different orders are strictly equal. So the most important example is evidently infinitely coherent.

The approach that we have been taking up to this point is known as **fibred**. This is in contrast to working in an **indexed** setting, where we can do away with boundary maps entirely. For example, instead of talking about the collection of lines all together, we talk about separate collections of lines joining specific points x and y. Similarly, triangles would be indexed by three points x, y, z and three lines: from x to y, from x to z, and from y to z.

Now working in the setting of type theory, we define a **semi-simplicial type** to consist of the the following infinite list of data:

$A : \mathsf{Type}$

$B : A \to A \to \mathsf{Type}$

$C : \{x\ y\ z : A\} \to B\ x\ y \to B\ x\ z \to B\ y\ z \to \mathsf{Type}$

$D : \{x\ y\ z\ w : A\}\{\alpha : B\ x\ y\}\{\beta : B\ x\ z\}\{\gamma : B\ x\ w\}\{\delta : B\ y\ z\}\{\epsilon : B\ y\ w\}\{\zeta : B\ z\ w\}$
$\quad \to C\ \alpha\ \beta\ \delta \to C\ \alpha\ \gamma\ \epsilon\ \to C\ \beta\ \gamma\ \zeta \to C\ \delta\ \epsilon\ \zeta \to \mathsf{Type}$

$\ldots$

Here, we are using Agda curly brace notation to indicate that we are leaving some of the indices as implicit. We thus write $C\ \alpha\ \beta\ \epsilon$ instead of $C\ x\ y\ z\ \alpha\ \beta\ \epsilon$.

If we had this infinite collection of data, we could sigma-up each of these notions over all of its indices to obtain a total type, as before. We could then define face maps. It is evident that these face maps would be infinitely coherent. Consequently, this makes a very strong case that the definition that we take in the fibred setting should include all possible coherences of any order, and indeed, one can argue that the indexed and fibred definitions are equivalent when all coherences are included in the latter.

For example, given the data $\partial_0, \partial_1 : X_1 \to X_0$, and $\partial_0, \partial_1, \partial_2 : X_3 \to X_2$, we can define:

$$\begin{aligned}\widetilde{X}_0 &= X_0\\ \widetilde{X}_1\,(x, y) &= \sum_{\alpha : X_1} (\partial_1\ \alpha \equiv x) \times (\partial_0\ \alpha \equiv y)\end{aligned}$$

For the next case, we are defining $\widetilde{X}_2\,(x, y, z, (\alpha, p_0, q_0), (\beta, p_1, q_1), (\gamma, p_2, q_2))$.

One may be tempted to say that this consists of $f : X_2$ with boundary data α, β, γ, by asserting, for example, that $\partial_2\ f \equiv \alpha$. However this equality in X_1 leaves the endpoints free. For example, in the case of the singular semi-simplicial type of a type X, so long as the lines $\partial_2\ f$ and α lived in the same connected component of X, they could be identified by this criterion. We see, then, that this comparison should be performed in the type $\widetilde{X}_1(x, y)$.

We begin by asking for an element $f : X_2$ which has the right 0-simplex boundary. This requires the data: $r_0 : \partial_1\ \partial_2\ f \equiv x$, $r_1 : \partial_0\ \partial_2\ f \equiv y$, and $r_2 : \partial_0\ \partial_1\ f \equiv z$. Now, we can create an element of $\widetilde{X}_1(x, y)$ to compare with (α, p_0, q_0) via $(\partial_2\ f, r_0, r_1)$.

In order to create an element of $\widetilde{X}_1(x, z)$, we want to use $\partial_1\ f$. We then have that $r_2 : \partial_0\ \partial_1\ f \equiv z$, giving us a proof that the right endpoint is z. However, for the left endpoint, we only have $r_0 : \partial_1\ \partial_2\ f \equiv x$, and we need to concatenate this on the left with an equality $\partial_1\ \partial_1\ f \equiv \partial_1\ \partial_2\ f$ in order to show that $\partial_1\ \partial_1\ f \equiv x$, as required. A similar analysis applies for $\widetilde{X}_1(y, z)$. Thus we require the commutation identities $\alpha_{k,l} : \partial_k \circ \partial_l \equiv \partial_{l-1} \circ \partial_k$ for $k < l$.

Provided these identities as part of our starting data, we would then complete the definition as follows:

$$\begin{aligned}&\widetilde{X}_2\,(x, y, z, (\alpha, p_0, q_0), (\beta, p_1, q_1), (\gamma, p_2, q_2)) =\\ &\quad\sum_{(f\,:\,X_2)}\ \sum_{(r_0\,:\,\partial_1\partial_2 f \equiv x)}\ \sum_{(r_1\,:\,\partial_0\partial_2 f \equiv y)}\ \sum_{(r_2\,:\,\partial_0\partial_1 f \equiv z)}\\ &\qquad(\partial_2\ f\ ,\ r_0\ ,\ r_1) \equiv_{\widetilde{X}_1(x,y)} (\alpha\ ,\ p_0\ ,\ q_0)\\ &\qquad\times (\partial_1\ f\ ,\ \mathsf{apeq}\ \alpha_{1,2}\ f \cdot r_0\ ,\ r_2) \equiv_{\widetilde{X}_1(x,z)} (\beta\ ,\ p_1\ ,\ q_1)\\ &\qquad\times (\partial_0\ f\ ,\ \mathsf{apeq}\ \alpha_{0,2}\ f \cdot r_1\ ,\ \mathsf{apeq}\ \alpha_{0,1}\ f \cdot r_2) \equiv_{\widetilde{X}_1(y,z)} (\gamma\ ,\ p_2\ ,\ q_2)\end{aligned}$$

Using contractible singletons and path algebra, one can show that forming the total spaces of the resulting indexed types leads to types equivalent to X_0, X_1, X_2. Similarly, starting off with indexed types $\widetilde{X}_0, \widetilde{X}_1, \widetilde{X}_2$, forming their total spaces, and then performing the above construction results in equivalent types. This demonstrates an equivalence between the indexed and fibred definitions up to the second stage. One can continue this analysis to the third stage, although writing out the details would be exceptionally painful.

One can ask for a proof of this equivalence in general, which brings us to the punchline: *neither the indexed nor fibred formulations of semi-simplicial types have ever been formally defined inside of type theory!* (As such, the general theorem would involve an equivalence where neither the RHS nor LHS currently have definitions.)

The Pattern

The problem of defining semi-simplicial types (hereafter simply SSTs) in the indexed formulation was posed by Vladimir Voevodsky over ten years ago and has remained one of the most important open problems of homotopy type theory. At first glance, this seems to be an odd state of affairs, because there clearly is a pattern in play. Empirically, you can understand this pattern by writing out the next entry or two; that a line is indexed by its endpoints, and that a triangle is indexed by its bounding lines, clearly generalises.

For this reason, many researchers in homotopy type theory have made their attempts at constructing SSTs, ideally in a proof assistant. As they discover, all direct attacks of the problem are doomed to end in **higher coherence issues**. Despite the relative simplicity of the problem statement in the indexed formulation relative to the fibred formulation, coherences of presicely the same form will eventually come into play.

Of course, just being told that this will be the outcome of your attempt will serve to provide little intuition, and you generally have to go through the process of making your own failed implementation. The reason for this is that the problem appears very tractable at first glance. One thing that may help your intuition in the meanwhile, though, is seeing a precise scientific description, internal to type theory, of the evident pattern.

To see this, we will make all arguments explicit, reorder the terms, and add binary labels:

$$A : \mathsf{Type}$$

$$B : \underbrace{(x : A)}_{1}\ \underbrace{(y : A)}_{10} \to \mathsf{Type}$$

$$C : \underbrace{(x : A)}_{1}\ \underbrace{(y : A)}_{10}\ \underbrace{(\alpha : B\ x\ y)}_{11}\ \underbrace{(z : A)}_{100}\ \underbrace{(\beta : B\ x\ z)}_{101}\ \underbrace{(\gamma : B\ y\ z)}_{110} \to \mathsf{Type}$$

$$
\begin{aligned}
D : {} & \underbrace{(x : A)}_{1}\ \underbrace{(y : A)}_{10}\ \underbrace{(\alpha : B\ x\ y)}_{11}\ \underbrace{(z : A)}_{100}\ \underbrace{(\beta : B\ x\ z)}_{101}\ \underbrace{(\gamma : B\ y\ z)}_{110}\ \underbrace{(f_0 : C\ x\ y\ \alpha\ z\ \beta\ \gamma)}_{111} \\
& \underbrace{(w : A)}_{1000}\ \underbrace{(\delta : B\ x\ w)}_{1001}\ \underbrace{(\epsilon : B\ y\ w)}_{1010}\ \underbrace{(f_1 : C\ x\ y\ \alpha\ w\ \delta\ \epsilon)}_{1011}\ \underbrace{(\zeta : B\ z\ w)}_{1100} \\
& \underbrace{(f_2 : C\ x\ z\ \beta\ w\ \delta\ \zeta)}_{1101}\ \underbrace{(f_3 : C\ y\ z\ \gamma\ w\ \epsilon\ \zeta)}_{1110} \to \mathsf{Type}
\end{aligned}
$$

This ordering of terms is hereby known as **the binary ordering**.

We can think of the canonical n-simplices as being represented by a string of all 1s. So a point would be 1, a line would be 11, and a triangle would be 111, etc. Then the dependencies of a canonical n-simplex are given by all non-zero binary arguments strictly less than the indexing number.

We say that the **dimension** of a binary number is one less than the number of 1s that it has. The dimension of a given labelled argument determines whether it indexes a point (dimension 0), or line (dimension 1), or triangle (dimension 2), etc. To understand the ordering that this gives, imagine that you're filling out an infinite dimensional simplex by adding, at each stage, a facet of highest possible dimension. The order would go as follows: *point, point, line, point, line, line, triangle, point, line, line, triangle, line, triangle, triangle, tetrahedron, etc.*

Dimensions are not the whole story, though, as, when adding two lines in a row, you have a choice of which line to add first. We say that one binary number is a **subset** of another binary number, if, when aligned to the right, the 1s digits of the first lie in a subset of the position of the 1s digits of the second. The dependencies of a particular argument are precisely those indexed by its non-zero proper subsets, in increasing order. This tells us precisely which facet we're indexing at each stage.

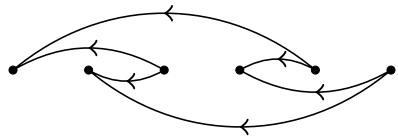
Simple Inverse Categories and Diagrams

The pattern described above is very satisfying. The question on our plate now is whether on not this observation is sufficient to solve the problem of constructing SSTs. In order to address this, we should scientifically qualify the form of the pattern in play, and then go from there.

The type of each $A, B, C, \ldots$ is given by some telescope mapping to Type, and each of the items in the telescope is of the form of one of $A, B, C, \ldots$ with previously occurring variables applied to it. The variables being applied are free of duplicates and are listed in the order in which they occur in the telescope. We also think of each $A, B, C, \ldots$ as having a dimension: A defines the 0-simplices, B defines the 1-simplices, etc.

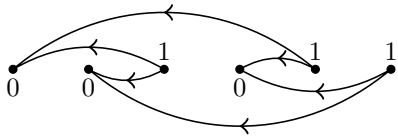
The dependency structure of each telescope can be given by a *dependency list*. A dependency list is either empty, or you can cons on an item to an existing dependency list by specifying on which of the previously occurring items it

depends. For example, the dependency list of a 2-simplex may be pictured as follows:



One will note that a dependency list is precisely equivalent to the data of a *linearised directed acyclic graph*.

Finally, in addition to giving the structure of the dependencies between items in the telescope, we must also label each item with a dimension:



To take an itinerary then, to the type of each of $A, B, C, \ldots$ we associate a particular telescope. *(Or, more precisely, we have a telescope for each dimension.)* The data of each such telescope is precisely specified by a **dimension labelled dependency list**.

In `Agda`, I have constructed such data corresponding to the pattern seen in the binary ordering of SSTs. Moreover, I have proven the following three properties of this data:

(a) **Dimension Boundedness**: All items in the labelled dependency list of an n-simplex have dimension strictly less than n.

(b) **Transitivity**: If an item depends on some other item, then it also depends on all of its dependencies.

(c) **The Shape Property**: If you take an item of dimension n in some labelled dependency list and *prune* the items preceding it by discarding all items on which it does not depend, then the resulting labelled dependency list is precisely equal to that of the dependencies of an n-simplex.

Intuitively, for any dimension indexed family of telescopes, you can write a program that will output Agda code that folds up these telescopes (in the form of a postulate context sequentially defining several of the symbols $A, B, C, \ldots$). The question is then: *When will the output code be well-typed?*, and the answer is precisely when these three properties hold!

These three properties have important category theoretic significance; they precisely assert that our dimension indexed family of telescopes defines the

structure of a **simple inverse category**[1] (SIC for short). The problem of constructing a corresponding postulate context $A, B, C, \ldots$ is known as that of constructing a family of **diagrams** for this SIC.

We may thus now completely disentangle the problem of constructing SSTs into two steps: First, constructing the corresponding SIC. Second, building the corresponding diagrams.

As alluded to before, the first step has already been completed in Agda. We call the resulting object the *binary SIC*. Everybody's proximal reaction to being introduced to the problem of SSTs is to think that there is an obvious pattern, and the most fitting response to this impression is: *yes, and here is that pattern, formally, in Agda*.

Note that data defining a SIC lives in the world of h-sets (after all, we're working with discrete graphs, with natural number labels, plus some propositions about this data). Diagrams, on the other hand, will involve arbitrary types, hence being the place where the possibility of higher coherence issues creeps in.

Over ten years of failed attempts to construct SSTs suggests to experts in the field that the general problem of building diagrams for all simple inverse categories is impossible in Martin-Löf Type Theory, as with the particular case of the binary SIC.

On the other hand, the problem of constructing diagrams for all simple inverse categories is solvable in MLTT with Axiom K, as truncatedness dispenses of higher coherences. Similarly, without Axiom K, one can construct *semi-simplicial sets* and *semi-simplicial groupoids* (indeed, up to any level of truncatedness, depending on how much rote work one is willing to put into proving coherences).

Additionally, the general problem is also solvable in two level type theory (indeed, this has historically been the main application of the theory). The amount of additional assumptions made in that theory, though, far exceeds the minimal amount of additional proof theoretic strength needed to make the problem tractable, so this state of affairs still leaves the problem of constructing SSTs wide open.

I claim that a resolution of the problem of constructing SSTs would involve constructing a new type theory, extending MLTT, in which the power added is, in some way, evidently the least amount of power necessary for constructing SSTs. Essentially, what we want to is to come to the crux of the issue, and identify precisely what proof theoretic strength is necessary for having SSTs internal to a type theory. Preliminary work in this direction would then pave a pathway for a later proof of the impossibility of constructing SSTs in MLTT.

[1] We can define a *simple inverse category* to be such a collection of data, except that we also want to quotient out by particular choices of linearisation in the dependency DAG.

TYPE STRUCTURES

The work that we are about to present emerged from my previous project on formalising the categorical logic of simply typed lambda calculus in `Agda`. The central definition in that work was that of a *simple contextual category*, which is like a category, except that the notions of *objects* and *morphisms* is derived from more primitive notions of *types* and *terms*.

In simple type theory, there is a notion (i.e. type) of types. We then say that the objects of our theory, *contexts*, are defined to be lists of such types. Upon making these declarations, we have firmly ensconced ourselves in working with a simple, as opposed to dependent, type theory. What then is the structure of types in a dependent type theory?

Egbert Rijke and Paige North have a joint paper on *B-systems*, which are an adequate model of dependent type theories internal to MLTT. In this paper, the structure of types in a dependent type theory is specified by giving the collections of length n contexts, along with fundamental context projections:

$$\{\star\} = \mathsf{E}_0 \xleftarrow{\partial} \mathsf{E}_1 \xleftarrow{\partial} \mathsf{E}_2 \xleftarrow{\partial} \mathsf{E}_3 \xleftarrow{\partial} \dots$$

Egbert and I were independently working on modelling dependent type theory in `Agda`, with my eventual goal being optimising for ergonomics in formalising metatheory. The formulation above is fibred, as opposed to indexed, and is thus essentially unsuitable for formalisation (for example, saying that you have a type in a context involves postulating and transporting around equalities). Egbert and I independently arrived at using an equivalent indexed formulation in our code, using coinductive types, which we shall call a **type structure**, or **TyStr**, for short:

```
codata TyStr where
  ty : TyStr → Type
  ex : (𝒯 : TyStr) → ty 𝒯 → TyStr
```

In words, a TyStr consists of a notion of types, and for every such type, another TyStr representing the types depending on it. From a TyStr, we are able to define, using induction-recursion, the types of length n contexts:

```
data Ctx (𝒯 : TyStr) (n : ℕ) where
  ∅ : Ctx zero
  _+_ : {n : ℕ}
    (Γ : Ctx 𝒯 n) → ty (exs 𝒯 Γ) → Ctx 𝒯 (suc n)

exs : (𝒯 : TyStr) {n : ℕ} → Ctx 𝒯 n → TyStr
exs 𝒯 ∅ = 𝒯
exs 𝒯 (Γ + A) = ex (exs 𝒯 Γ) A
```

The above definition say that, to extend a context Γ, you add a type depending on Γ. In order to make sense of types depending on a context, this has to be defined mutually with an operation of extending a type structure by a context.

There is a neat way of thinking about TyStrs: We define the *linear SIC* to consist of the $\mathbb{N}$ indexed family of labelled dependency lists where the n-th list consists on n-items of increasing dimensions, and such that each item depends on all of the previous items. If we built a postulate context out of the linear SIC, it would look like:

$A : \mathsf{Type}$

$B : (x : A) \to \mathsf{Type}$

$C : (x : A)\ (y : B\ x) \to \mathsf{Type}$

$D : (x : A)\ (y : B\ x)\ (z : C\ x\ y) \to \mathsf{Type}$

$\ldots$

Then the type of data of defining each element of the infinite list $A, B, C, \ldots$ is exactly given by the type of TyStrs. Thus, in a slightly different sense than how we were describing the problem before, the type TyStr is a construction of the type of diagrams for the linear SIC.

In the case of TyStrs, the coinductive formulation is really expressing something profound about the type of linear diagrams that we do not directly see in the fibred construction described above. We can express this as a mapping-in universal property:

```
UP : (A : Type) (Z : A → Type)
  (S : (x : A) → Z x → A) → (A → TyStr)
ty (UP A Z S x) = Z x
ex (UP A Z S x) x' = UP A Z S (S x x')
```

Thus, to construct a function $A \to \mathsf{TyStr}$, you need to give a *ZS structure* on the type A. This consists of Z, a type dependent on A, and S, a family of functions $Z\ x \to A$. Note that the types of Z and S are read directly off of the fields of TyStr. With this data, every element of A defines a TyStr.

We have shown that the problem of constructing the type of diagrams of a SIC can be solved in certain cases. In particular, if the type of diagrams happens to have a universal property, then the question about whether or not we can construct such a type comes down to being able to construct anything satisfying that universal property (because universal properties characterise things uniquely). We can prove that the fibred formulation above satisfies the UP of linear diagrams, so it's possible to perform this construction without coinduction. However, the universal property in question is manifestly coinductive so having coinduction is imperative for having a canonical type of linear diagrams.

The plan, therefore, is exhibit a universal property associated to SSTs, and to construct a type theory capable of expressing that universal property.

SSTs Are a Dependent Type Theory

The problem of constructing SSTs has traditionally been posed as that of constructing a function $\mathsf{SST} : \mathbb{N} \to \mathsf{Type}$ such that $\mathsf{SST}\ n$ is the type of n-truncated SSTs (this is usually thought of as a sigma type specifying the first n of $A, B, C, \ldots$). A solution to this problem would lead to a definition of infinite SSTs, albeit in a messy way.

A refinement on the problem statement is to instead ask for $\mathsf{SST} : \mathsf{TyStr}$. In other words, SST naturally has the structure of a TyStr. We could manually define the first few stages of this TyStr by copattern matching:

```
SST : TyStr
ty SST = Type
ty (ex SST A) = A → A → Type
ty (ex (ex SST A) B) =
  (x y z : A) → B x y → B x z → B y z → Type
ex (ex (ex SST A) B) C = ?
```

If one had such a TyStr, then n-truncated SSTs would be given by the type $\mathit{Ctx}\ \mathsf{SST}\ n$, and infinite SSTs would analogously be infinite contexts – here Ctx^∞ can be defined coinductively in a straightforward way. Moreover, the general problem of constructing diagrams is best phrased as that of turning a SIC into a TyStr.

Surprisingly, this formulation of the problem is **not** the one that we work towards in the remainder of this note. The reason for this is that the type of diagrams $\mathit{Ctx}^\infty\ \mathsf{SST}$ does not have a universal property on its own, and we must instead work in a more general setting.

The plan is that we will describe $\mathsf{SST} : \mathsf{TyStr}$ such that $\mathit{ty}\ \mathsf{SST}$ is the type of infinite SSTs. Intuitively, then, we think of $A : \mathit{ty}\ \mathsf{SST}$ as consisting of the following infinite collection of data:

$$A_0 : \mathsf{Type}$$
$$A_1 : A_0 \to A_0 \to \mathsf{Type}$$
$$A_2 : \{x\ y\ z : A_0\} \to A_1\ x\ y \to A_1\ x\ z \to A_1\ y\ z \to \mathsf{Type}$$
$$\ldots$$

If we accept this definition, for the time being, then we must next ask for the interpretation of $B : \mathit{ty}\ (\mathit{ex}\ \mathsf{SST}\ A)$. This will consist of the data:

$$B_0 : A_0 \to \mathsf{Type}$$
$$B_1 : \{x\ y : A_0\} \to A_1\ x\ y \to B_0\ x \to B_0\ y \to \mathsf{Type}$$
$$B_2 : \{x\ y\ z : A_0\}\{\alpha : A_1\ x\ y\}\{\beta : A_1\ x\ z\}\{\gamma : A_1\ y\ z\}\{x' : B_0\ x\}\{y' : B_0\ y\}\{z' : B_0\ z\}$$
$$\to A_2\ \alpha\ \beta\ \gamma \to B_1\ \alpha\ x'\ y' \to B_1\ \beta\ x'\ z' \to B_1\ \gamma\ y'\ z' \to \mathsf{Type}$$
$$\ldots$$

B is known as a dependent semi-simplicial type over A. A_0 was just the type of points of A, whereas $B_0\ x$ is the type of points of B living over some point $x : A_0$. Similarly, the type $B_1\ \alpha\ x'\ y'$ describes lines in B living over $\alpha : A_1\ x\ y$, starting from $x' : B_0\ x$, and ending at $y' : B_0\ y$. In general, the type of dependent n-simplices in B will be indexed by a filled-in n-simplex in A, as well as a lift of its boundary to B.

Similarly, we identify $C : ty\ (ex\ \mathsf{SST}\ A)\ B)$ with the data:

$$C_0 : \{x : A_0\} \to B_0\ x \to \mathsf{Type}$$
$$C_1 : \{x\ y : A_0\}\{\alpha : A_1\ x\ y\}\{x' : B_0\ x\}\{y' : B_0\ y\}$$
$$\quad \to B_1\ \alpha\ x'\ y' \to C_0\ x' \to C_0\ y' \to \mathsf{Type}$$
$$\ldots$$

The pattern is much the same: a doubly dependent n-simplex in C is indexed by a filled in dependent n-simplex in B, as well as a lift of its boundary to C.

We have defined $\mathsf{SST} : \mathsf{TyStr}$ to capture SSTs of all levels of dependency. The remarkable observation is that SST defines the collection of types in a model of dependent type theory, in the sense of B-systems. To show this, we have to construct weakening, elements, substitution, and variables, and show that these constructs satisfy a list of properties.

The most interesting construct is that of elements. Intuitively, we can consider a notion of morphisms of SSTs, and elements of $A : ty\ \mathsf{SST}$ should be equivalent to morphisms from the terminal SST to A. The terminal SST has exactly one filler for any boundary data, and so a morphism would send $\star_0 : \top_0$, to $t_0 : A_0$. It would then have to send the only 1-simplex, $\star_1 : \top_1\ \star_0\ \star_0$ to some $t_1 : A_1\ x_0\ x_0$. Similarly, it would have to send the only 2-simplex, $\star_2 : \top_2\ \star_1\ \star_1\ \star_1$, to some $t_2 : A_2\ x_1\ x_1\ x_1$. Thus, an element of $A : ty\ \mathsf{SST}$ consists of the infinite collection of data:

$$t_0 : A_0$$
$$t_1 : A_1\ t_0\ t_0$$
$$t_2 : A_2\ t_1\ t_1\ t_1$$
$$\ldots$$

An element of $B : ty\ (ex\ \mathsf{SST}\ A)$ should correspond to a dependent morphism of SSTs from A to B (in other words, a section). Thus for every $x : A_0$, we get some $s_0\ x : B_0\ x_0$, and for every $\alpha : A_0\ x\ y$, we would get some $s_1\ \alpha : B_1\ \alpha\ (s_0\ x)\ (s_0\ y)$. Thus, an element of $B : ty\ (ex\ \mathsf{SST}\ A)$ consists of the infinite collection of data:

$$s_0 : (x : A_0) \to B_0\ x$$
$$s_1 : \{x\ y : A_0\}(\alpha : A_1\ x\ y) \to B_1\ \alpha\ (s_0\ x)\ (s_0\ y)$$
$$s_2 : \{x\ y\ z : A_0\}\{\alpha : A_1\ x\ y\}\{\beta : A_1\ x\ z\}\{\gamma : A_1\ y\ z\}(f : A_2\ \alpha\ \beta\ \gamma)$$
$$\quad \to B_2\ f\ (s_1\ \alpha)\ (s_1\ \beta)\ (s_1\ \gamma)$$
$$\ldots$$

The most important part of the B-system structure on SST, for our purposes, is the weakening structure. Formally, we define:

```
codata TyMor (𝒯 𝒮 : TyStr) where
  fun : TyMor 𝒯 𝒮 → ty 𝒯 → ty 𝒮
  up : (ϱ : TyMor 𝒯 𝒮) (A : ty 𝒯)
    → TyMor (ex 𝒯 A) (ex 𝒮 (fun ϱ A))

codata WkStr (𝒯 : TyStr) where
  wk : WkStr 𝒯 → (A : ty 𝒯) → TyMor 𝒯 (ex 𝒯 A)
  up : WkStr 𝒯 → (A : ty 𝒯) → WkStr (ex 𝒯 A)
```

In the fibred formulation of the type of linear diagrams, a TyMor would be equivalent to a natural transformation of such diagrams. The key point is that TyMors define not only how to operate on top-level types, but also on arbitrary contexts. A WkStr on $\mathcal{T}$ then is a way to shift contexts in $\mathcal{T}$, or any of its iterated extensions, over by a type in what we conceptually view as adding a trivial dependency. Weakening on SST is intuitively constructed in this way – we weaken by adding trivial dependencies.

Generalised (co)Induction

Let's consider first the case of type streams. We define:

```
codata TyStream where
  head : TyStream → Type
  tail : TyStream → TyStream
```

Thus a TyStream is just an infinite list (aka stream) of types. A morphism of TyStreams is defined in the evident way as *"rungs on a ladder"*. Note that is we had streams of natural numbers, there would be no similar suitable notion of morphisms, so TyStreams are a special kind of stream.

Now, what would it mean to *inductively define a TyStream*? In other words, we want to make sense of definitions of the form:

```
data Foo : TyStream where
  Z : head Foo
  S : Foo → tail Foo
```

This is written in the internal logic of the category of TyStreams, so, for example, Z is a morphism of TyStreams.

Well, in this example, we ask ourselves whether this definition meaningfully defines Foo : TyStream? Well, we have an element Z : $head$ Foo, and a way to promote elements down the stream, so this should meaningfully define something equivalent to a stream of units.

When the possibility of such generalised TyStream-valued induction was proposed, Mike and I immediately wrote down a definition of SSTs:

```
codata SST : TyStr where
  Z : SST → Type
  S : (X : SST) → Z X → ex SST X
```

This is TyStr-valued coinduction, and is written in the internal language of TyStrs. *There is a lot to unpack here*, although the main idea is that the *type* of infinite SSTs does not on its own have a universal property, instead, you have to generalise to all notions of dependency, and only then does the TyStr of SSTs acquire a universal property.

We then spent the next seven months thinking about this fictional code fragment and working on a type theory in which it would type-check. In general, the original definition above is almost right, but what one realises after working on this for several weeks is that the setting has to be expanded to TyStrs with weakening in order to recover simplex types. Conjecturally, an analogous B-system-valued coinduction will correctly recover the B-system of SSTs described above, with extraction algorithms for both simplex types and element components.

UNPACKING THINGS

We begin with the projection Z : SST → Type. This should be a TyMor, so Type should be a TyStr. We can define the universal TyStr as follows:

```
𝒰' : Type → TyStr
ty (𝒰' A) = A → Type
ex (𝒰' A) B = 𝒰' (Σ A B)

𝒰 : TyStr
ty 𝒰 = Type
ex 𝒰 A = 𝒰' A
```

An alternative definition is given as follows:

```
𝒰' : Type → TyStr
𝒰' A = UP Type (λ A → A → Type) Σ

𝒰 : TyStr
𝒰 = 𝒰' ⊤
```

In other words, A : $ty\ \mathcal{U}$ is a type, B : $ty\ (ex\ \mathcal{U}\ A)$ is a type depending on A, and C : $ty\ (ex\ (ex\ \mathcal{U}\ A)\ B)$ is a type depending on Σ A B, etc. Thus, morally, $\mathcal{U}$ is the TyStr of types, and a TyMor $\mathcal{S} : \mathcal{T} \to \mathcal{U}$ should represent a dependent type structure over $\mathcal{T}$.

It is often easier to work with dependent type structures directly. We define:

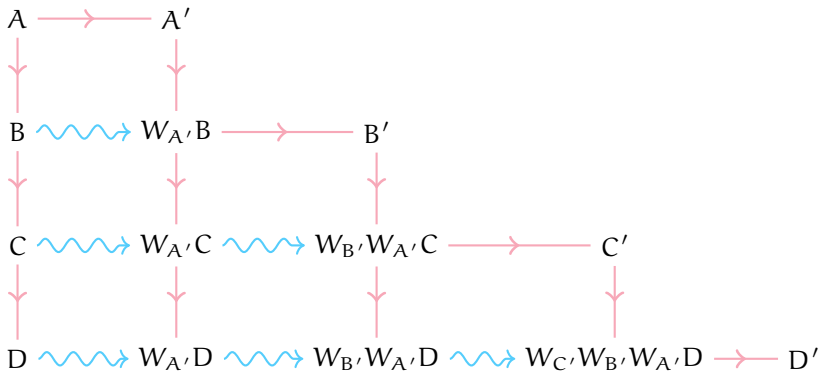
```
codata TyStr^d (𝒯 : TyStr) where
  ty^d : TyStr^d 𝒯 → ty 𝒯 → Type
  ex^d : (𝒮 : TyStr^d 𝒯) (A : ty 𝒯) → ty^d 𝒮 A
    → TyStr^d (ex 𝒯 A)
```

One also obtains a notion $Ctx^{\mathrm{d}}\ \mathcal{S}\ \Gamma$ of dependent contexts in $\mathcal{S}$ living over some $\Gamma : Ctx\ \mathcal{T}\ n$. In the same way that the fibred notion of linear diagrams is charictarised by giving all of the length n contexts, along with their projections, the fibred notion of dependent linear diagrams is given by all of the Γ-indexed contexts, and their projections.

We thus think of Z as a $\mathsf{TyStr}^{\mathrm{d}}$ over SST. The projection $S : (X : \mathsf{SST}) \to Z\ X \to ex\ \mathsf{SST}\ X$ is then to be thought of as a morphism between two dependent type structures over SST. We thus need to define, for every type structure $\mathcal{T}$, a dependent type structure $EX\ \mathcal{T} : \mathsf{TyStr}^{\mathrm{d}}\ \mathcal{T}$.

We will now introduce a notation for contexts. If $\mathcal{T}$ is a TyStr, we will write $\Gamma = A \to B \to C \to D : Ctx\ \mathcal{T}\ 4$ to mean that $A : ty\ \mathcal{T}$, and that B depends on A, etc. Further, if $\mathcal{S} : \mathsf{TyStr}^{\mathrm{d}}\ \mathcal{T}$, we will write $\tilde{\Gamma} = A' \Rightarrow B' \Rightarrow C' \Rightarrow D' : Ctx^{\mathrm{d}}\ \mathcal{S}\ \Gamma$ to indicate that $A' : ty^{\mathrm{d}}\ \mathcal{S}\ A$, $B' : ty^{\mathrm{d}}\ (ex^{\mathrm{d}}\ \mathcal{S}\ A\ A')B$, etc. Thus, in order to define a dependent type structure over $\mathcal{T}$, me must give meaning to dependent contexts of the form $\tilde{\Gamma} = A' \Rightarrow B' \Rightarrow C' \Rightarrow D' : Ctx^{\mathrm{d}}\ \mathcal{S}\ \Gamma$.

Now, it turns out that in order to define $EX\ \mathcal{T}$, we need to assume that $\mathcal{T}$ is equipped with a weakening structure. Under this assumption, the dependent contexts $\tilde{\Gamma} = A' \Rightarrow B' \Rightarrow C' \Rightarrow D' : Ctx^{\mathrm{d}}\ \mathcal{S}\ \Gamma$ living over $\Gamma = A \to B \to C \to D : Ctx\ \mathcal{T}\ 4$ are described as consisting of the data of A', B', C', and D' fitting into the following diagram (called the EX diagram):



Given a context $\Gamma : Ctx\ \mathsf{SST}\ n$, we refer to a dependent context $\tilde{\Gamma} : Ctx^{\mathrm{d}}\ Z\ \Gamma$ as a Z-section of Γ. S is therefore is a gadget that, given any context $\Gamma : Ctx\ \mathsf{SST}\ n$, transforms Z-sections of Γ into sections of $EX\ \mathsf{SST}$ over Γ.

Suppose that we have $\mathcal{T}$: TyStr, equipped with a $\mathcal{W}$: WkStr $\mathcal{T}$. Then a ZS-structure on $(\mathcal{T}, \mathcal{W})$ consists of Z : TyStr$^{\mathrm{d}}$ $\mathcal{T}$, and S, a morphism of TyStr$^{\mathrm{d}}$s over $\mathcal{T}$ from Z to $EX\ \mathcal{T}$.

From the previous discussion, this data should be sufficient to define a TyMor $\mathcal{T} \to$ SST. So every A : $ty\ \mathcal{T}$ should give rise to an infinite semi-simplicial type, and every B : $ty\ (ex\ \mathcal{T}\ \mathsf{A})$ should give rise to an infinite semi-simplicial type depending on the previous one, etc. In order to see that this is the case, we present an algorithm for extracting simplex types $\mathsf{A}_0, \mathsf{A}_1, \mathsf{A}_2, \ldots$ from A.

Given A : $ty\ \mathcal{T}$, this can be viewed as the length one context

$$\Gamma_0 = \varnothing\ +\ \mathsf{A} : Ctx\ \mathcal{T}\ 1.$$

The type A_0 of 0-simplices is given as $ty^{\mathrm{d}}\ \mathsf{Z}\ \mathsf{A}$: Type. If we now choose a 0-simplex x : $ty^{\mathrm{d}}\ \mathsf{Z}\ \mathsf{A}$, then we obtain a Z-section $\varnothing\ +\ \mathsf{x} : Ctx^{\mathrm{d}}\ \mathsf{Z}\ \Gamma_0$. Applying S to this Z-section turns it into $\varnothing\ +\ \mathsf{S}\ \mathsf{x} : Ctx^{\mathrm{d}}\ (EX\ \mathcal{T})\ \Gamma_0$.

Now, the key thing to note is that length $\mathfrak{n}$ contexts in $EX\ \mathcal{T}$ translate into length $2\mathfrak{n}$ contexts in $\mathcal{T}$, given by the outer stair shaped sequence of dependencies in the EX diagram. Thus, what we really have now is a context:

$$\Gamma_1 = \varnothing\ +\ \mathsf{A}\ +\mathsf{S}\ \mathsf{x} : Ctx\ \mathcal{T}\ 2.$$

We can now form a Z-section of this context: $\varnothing\ +\ \mathsf{y} +\ \alpha : Ctx^{\mathrm{d}}\ \mathsf{Z}\ \Gamma_1$. Note that $\mathsf{y} : ty^{\mathrm{d}}\ \mathsf{Z}\ \mathsf{A}$, so it's a 0-simplex as well, and then $\alpha : ty^{\mathrm{d}}\ (ex^{\mathrm{d}}\ \mathsf{Z}\ \mathsf{A}\ \mathsf{y})\ (\mathsf{S}\ \mathsf{x})$ is the type of 1-simplices whose boundary consists of the points x and y. We then apply S to this to obtain the length 4 context:

$$\Gamma_2 = \varnothing\ +\ \mathsf{A}\ +\mathsf{S}\ \mathsf{y}\ +\ W_{\mathsf{S}\ \mathsf{y}}(\mathsf{S}\ \mathsf{x})\ +\ \mathsf{S}\ \alpha : Ctx\ \mathcal{T}\ 4.$$

We then take another Z-section, this time of Γ_2: $\varnothing\ +\ z +\ \beta\ +\ \gamma\ +\ \mathfrak{f}_0 : Ctx^{\mathrm{d}}\ \mathsf{Z}\ \Gamma_2$. We have that $z : ty^{\mathrm{d}}\ \mathsf{Z}\ \mathsf{A}$ and $\beta : ty^{\mathrm{d}}\ (ex^{\mathrm{d}}\ \mathsf{Z}\ \mathsf{A}\ z)\ (\mathsf{S}\ \mathsf{y})$, so z is a 0-simplex and β is a 1-simplex from y to z. The difficulty, however, is that

$$\gamma : ty^{\mathrm{d}}\ (ex^{\mathrm{d}}\ (ex^{\mathrm{d}}\ \mathsf{Z}\ \mathsf{A}\ z)\ (\mathsf{S}\ \mathsf{y})\ \beta)\ (W_{\mathsf{S}\ \mathsf{y}}(\mathsf{S}\ \mathsf{x})).$$

What we want is for this type to be $ty^{\mathrm{d}}\ (ex^{\mathrm{d}}\ \mathsf{Z}\ \mathsf{A}\ z)\ (\mathsf{S}\ \mathsf{x})$, in other words, that of a 1-simplex from x to z. The appearance of β in the above expression suggests that the type of γ depends on β, which is incorrect, however, there is also a weakening, and the idea is that this weakening should cancel the extension, giving us the desired reduction.

Ignoring this problem for the moment, we obtain that

$$\mathfrak{f}_0 : ty^{\mathrm{d}}\ (ex^{\mathrm{d}}\ (ex^{\mathrm{d}}\ (ex^{\mathrm{d}}\ \mathsf{Z}\ \mathsf{A}\ z)\ (\mathsf{S}\ \mathsf{y})\ \beta)\ (W_{\mathsf{S}\ \mathsf{y}}(\mathsf{S}\ \mathsf{x}))\ \gamma)\ (\mathsf{S}\ \alpha),$$

and this should be the type of 2-simplices with boundary x, y, z, α, β, γ. In the next iteration, we would apply S to this context to obtain a length 8 context. Z sectioning this context would pass from the 2-simplex that we previously specified to a 3-simplex. The type of the last element of that Z-section would be the type of 3-simplices. We can repeat this process to extract all of the simplex types.

Fascinatingly, the above algorithm actually describes how to extract simplex types from types at all levels of dependency in $\mathcal{T}$. Instead of starting with the context $\varnothing + \mathrm{A}$, we could have started with the context $\varnothing + \mathrm{A} + \mathrm{B}$. A Z-section of this context would be $\varnothing + \mathrm{x} + \mathrm{x}'$, where $\mathrm{x} : ty^{\mathrm{d}}\ \mathsf{Z}\ \mathrm{A}$ is a 0-simplex in A, and $ty^{\mathrm{d}}\ (ex\ \mathsf{Z}\ \mathrm{A}\ \mathrm{x})\ \mathrm{B}$ is the type of dependent 0-simplices in B living over x. Repeatedly applying S and sectioning the new context would yield, as the type of the last element, all of the dependent simplex types in B.

In Agda, I have formalised the notion of a ZS-structure on a TyStr with weakening. From any such data, given $\mathrm{A} : ty\ \mathcal{T}$, I construct a derived TyStr such that its contexts look like $\varnothing + \mathrm{x} + \mathrm{y} + \alpha + z + \beta + \gamma + \mathfrak{f}_0$ as above. In other words, taking contexts in this TyStr achives the process of filling out successive facets of an infinite-dimensional simplex. (Producing such a gadget is my preferred way of solving the simplex extraction problem.)

This Agda file solves the problem of producing *unreduced simplex types*, in the sense of the problem that we alluded to above. Since we are working in the theory of type structures with weakening, the correct notion of morphism involves a weakening preservation condition. The universe $\mathcal{U}$ has a weakening structure that just weakens a type by adding a trivial dependency, so Z preserving weakening would be a proposition that would allow us to cancel the ex with the weakening in the type of γ. In the type of a 3-simplex, as well, we would see weakening under S, and, in order to get reduced types, we also have to postulate that S preserves weakening – in effect, one weakening inside of S moves to two weakenings outside of S. (Also we will need weakening to preserve weakening in order to be able to state S preservation.) In experiments with rewriting, we observe that if all of these properties were definitional, then we would have reduced simplex types on the nose. Thus the entire game of the current research is making sense of this.

Finally, note that the order in which we extracted the simplices does not match the binary ordering since, for example, the line from y to z came before the line from x to z (the ordering of the dimensions, however, is a match). In the EX diagram, we chose to have the squiggly arrows indicating weakening be horizontal, which resulted in the length $2\mathfrak{n}$ context having a staircase shape. We could have instead drawn the diagram with vertical squiggly arrows, which would result in the length $2\mathfrak{n}$ context having an L-shape. The first version has an advantage in that adding in that insertions to the stair shaped context occur at the end, unlike the L-shaped context, in which one of the insertions occurs at the middle. The remarkable observation about the second version,

though, is that the same simplex extraction process as described above would yield simplices ordered *exactly in the binary ordering!*

The Singular Semi-Simplicial Types

In certain cases, we will be able to construct ZS-structures in which the weakening preservation laws hold definitionally. When this happens, the simplex extraction algorithm above will yield *"reduced"* expressions, and the problem of constructing that family of SSTs will have a complete resolution in present-day Agda. Fortunately, in the case of the most important example, the singular semi-simplicial types, we are able to produce such a structure!

Suppose that we have a type X, and we want to produce the SST $\mathsf{Sing}\ X$. We have a mapping-in universal property that lets us construct SSTs out of types in some type structure $\mathcal{T}$. What should this $\mathcal{T}$ be?

One observation to make is that $\mathcal{T}$ cannot be trivial in several senses. First, $\mathcal{T}$ cannot be the type structure with only a single closed type and no dependent types living over it. This is because $\mathcal{T}$ has to admit a weakening structure, so we must have, in the least, all possible weakenings of any closed type. We might then be tempted to take $\mathcal{T}$ to be the next simplest case – one in which there is exactly one type at each level of dependency (so units all the way up). Unfortunately, in that case, every dependent type will be a weakening of the unique closed type, and weakening preservation on Z would imply that Z is essentially constant. For example, the type of points would equal the type of lines between any x and y, as well as the type of triangle with any given boundary. This does not generally hold in the case of singular SSTs, or indeed, in most examples of SSTs, we thus see that the choice of $\mathcal{T}$ has to be uniquely tailored to the situation.

The key is to generalise the problem of constructing $\mathsf{Sing}\ X$ for a particular type X to that of constructing Sing to any type, and indeed, to any dependent type at all levels of dependency. In other words, we will take $\mathcal{T}$ to be $\mathcal{U}$, the universal type structure constructed above.