

Kindergarten Shapes



cube



simplex



globe



Semi-Simplicial Types

Astra Kolomatskaia

Joint work with Michael Shulman

Contents

α Why Are Semi-Simplicial Types Hard?

β A Proposed Constrution

↳ *(involves an extension of type theory)*



Why Are Semi-Simplicial Types Hard?

Types in Type Theory

Simple type theory:

There is a collection of types; contexts are defined to be lists of types

Dependent type theory:

There are collections of length n contexts, along with fundamental projections:

$$\{\star\} = E_0 \xleftarrow{\partial} E_1 \xleftarrow{\partial} E_2 \xleftarrow{\partial} E_3 \xleftarrow{\partial} \dots$$

Type Structures

The previous formulation is fibred, and thus unsuitable for formalisation

We instead work with an indexed equivalent:

```
codata TyStr where  
  ty : TyStr → Type  
  ex : ( $\mathcal{T}$  : TyStr) → ty  $\mathcal{T}$  → TyStr
```

In Agda:

```
record TyStr  $\ell$  : Type (lsuc  $\ell$ ) where  
  coinductive  
  field  
    ty : Type  $\ell$   
    ex : ty → TyStr  $\ell$ 
```

Contexts

```
codata TyStr where  
  ty : TyStr → Type  
  ex : ( $\mathcal{T}$  : TyStr) → ty  $\mathcal{T}$  → TyStr
```

From this definition, we are able to recover the type of length n contexts:

```
data Ctx ( $\mathcal{T}$  : TyStr) ( $n$  :  $\mathbb{N}$ ) where  
   $\emptyset$  : Ctx zero  
   $\_+ \_$  : { $n$  :  $\mathbb{N}$ }  
    ( $\Gamma$  : Ctx  $\mathcal{T}$   $n$ ) → ty (exs  $\mathcal{T}$   $\Gamma$ ) → Ctx  $\mathcal{T}$  (suc  $n$ )  
  
exs : ( $\mathcal{T}$  : TyStr) { $n$  :  $\mathbb{N}$ } → Ctx  $\mathcal{T}$   $n$  → TyStr  
exs  $\mathcal{T}$   $\emptyset$  =  $\mathcal{T}$   
exs  $\mathcal{T}$  ( $\Gamma + A$ ) = ex (exs  $\mathcal{T}$   $\Gamma$ ) A
```

Semi-Simplicial Types

Informally, a *semi-simplicial type* consists of the following infinite list of data:

$A_0 : \text{Type}$

$A_1 : A_0 \rightarrow A_0 \rightarrow \text{Type}$

$A_2 : \{x\ y\ z : A_0\} \rightarrow A_1\ x\ y \rightarrow A_1\ x\ z \rightarrow A_1\ y\ z \rightarrow \text{Type}$

$A_3 : \{x\ y\ z\ w : A_0\} \{ \alpha : A_1\ x\ y \} \{ \beta : A_1\ x\ z \} \{ \gamma : A_1\ x\ w \} \{ \delta : A_1\ y\ z \}$
 $\{ \epsilon : A_1\ y\ w \} \{ \zeta : A_1\ z\ w \} \rightarrow A_2\ \alpha\ \beta\ \delta \rightarrow A_2\ \alpha\ \gamma\ \epsilon \rightarrow A_2\ \beta\ \gamma\ \zeta \rightarrow A_2\ \delta\ \epsilon\ \zeta$
 $\rightarrow \text{Type}$

...

A_0 is a notion of *points*, A_1 is a notion of *lines*, A_2 is a notion of *triangles*, A_3 is a notion of *tetrahedra*, etc. – we are defining all of the *n-simplices*

Problem Statement

Constructing SSTs is one of the most important open problems in HoTT

The problem is usually posed as constructing a function

$$\text{SST} : \mathbb{N} \rightarrow \text{Type}$$

such that $\text{SST } n$ is the type of n -truncated SSTs

Conceptual Refinement 1:

It is better to ask for

$$\text{SST} : \text{TyStr}$$

Problem Statement [cont.]

We can begin defining SST by copattern matching:

```
SST : TyStr
  ty SST =
    Type
  ty (ex SST A0) =
    A0 → A0 → Type
  ty (ex (ex SST A0) A1) =
    {x y z : A0} → A1 x y → A1 x z → A1 y z → Type
  ex (ex (ex SST A0) A1) A2 = ?
```

We recover the n -truncated SSTs by taking Ctx SST n ,
and infinite SSTs may be defined as Ctx^∞ SST

Morally, we have passed to an indexed formulation of a problem from a fibred one

The Pattern

To discover the pattern, as observed by *Tim Campion*, we make all arguments explicit, and annotate items in the telescopes with binary numbers:

$A_0 : \text{Type}$

$A_1 : \underbrace{(x : A_0)}_1 \underbrace{(y : A_0)}_{10} \rightarrow \text{Type}$

$A_2 : \underbrace{(x : A_0)}_1 \underbrace{(y : A_0)}_{10} \underbrace{(\alpha : A_1 x y)}_{11} \underbrace{(z : A_0)}_{100} \underbrace{(\beta : A_1 x z)}_{101} \underbrace{(\gamma : A_1 y z)}_{110} \rightarrow \text{Type}$

$A_3 : \underbrace{(x : A_0)}_1 \underbrace{(y : A_0)}_{10} \underbrace{(\alpha : A_1 x y)}_{11} \underbrace{(z : A_0)}_{100} \underbrace{(\beta : A_1 x z)}_{101} \underbrace{(\gamma : A_1 y z)}_{110}$
 $\underbrace{(f_0 : A_2 x y \alpha z \beta \gamma)}_{111} \underbrace{(w : A_0)}_{1000} \underbrace{(\delta : A_1 x w)}_{1001} \underbrace{(\epsilon : A_1 y w)}_{1010} \underbrace{(f_1 : A_2 x y \alpha w \delta \epsilon)}_{1011}$
 $\underbrace{(\zeta : A_1 z w)}_{1100} \underbrace{(f_2 : A_2 x z \beta w \delta \zeta)}_{1101} \underbrace{(f_3 : A_2 y z \gamma w \epsilon \zeta)}_{1110} \rightarrow \text{Type}$

The Pattern [cont.]

The type of each A_n is a telescope of *items* mapping to Type

- ▶ Each item has type $A_m \dots$ with zero or more variables applied
- ▶ The variable applications are unique and appear in increasing order
- ▶ Each item is indexed by a binary number, ranging from 1 to $\underbrace{1 \dots 10}_{n \text{ ones}}$

Given an item indexed by binary number b , we have to answer two questions:

- Which A_m appears in its type?
- Which variables are applied to this A_m ?

The Pattern [cont.]

We start with some item, indexed by a binary number b :

(a) Which A_m appears in its type?

The *dimension* of a binary number is one less than the number of 1s that it has
e.g. 1101 has dimension 2

A dimension m item uses A_m

(b) Which variables are applied to this A_m ?

One binary number is a *subset* of another, 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
e.g. the non-zero proper subsets of 1101 are 1, 100, 101, 1000, 1001, 1100

The variables applied to A_m are indexed by the non-zero proper subsets of b

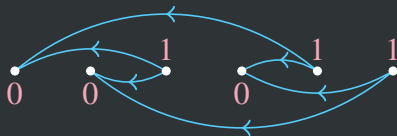
Dependency Lists

For each item, we need to say *yes* or *no* to depending on the previous items
We also need to know the dimension of the item

To each n we associated a *dimension labelled dependency list*

These will be linearised directed acyclic graphs with \mathbb{N} -valued vertex labels

Here is the labelled dependency list of a 2-simplex:



Properties of the Binary Ordering

In Agda, I have constructed a family of labelled dependency lists 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

Simple Inverse Categories

In general, given any \mathbb{N} indexed family of labelled dependency lists, we can construct untyped syntactic expressions A_1 , A_2 , A_3 , etc.

One asks: *When will these expressions be well-typed?*

The answer is: *Exactly when the above three properties hold!*

Such data presents a *Simple Inverse Category*

In the case of semi-simplicial types, we have completely scientifically qualified the nature of the pattern that is apparent when first encountering the problem

However, does this solve the problem of constructing semi-simplicial types?

The Typing Puzzle

I have formalised untyped syntax, typing derivations, and the construction of untyped expressions from families of labelled dependency lists

AUDIENCE CHALLENGE: Prove, in Cubical Agda, that any simple inverse category results in well-typed syntactic expressions, hence constructing semi-simplicial types “externally”

On GitHub: The result of ~35h of formalisation work setting up this puzzle

The main difficulty was proving things about the binary ordering

Conversations with [Emily Riehl](#) prompted formalising the binary ordering

Conversations with [Ophelia Bauckholt](#) prompted exploring typing

[Reed Mullanix](#) helped me with the formalisation for ~6h

Infinite Coherence Issues

Well-typedness derivations are terms of some inductive datatype

If we could construct a function from derivations to `Type`, we would be done

ISSUE: *HoTT can't eat itself! (as far as we know...)*

In the course of constructing such a function, one requires a coherence

In the course of proving the coherence, one requires a higher coherence

This continues...



A Proposed Construction

Dependent Semi-Simplicial Types

We think of a semi-simplicial type A as the following infinite list of data:

$$A_0 : \text{Type}$$
$$A_1 : A_0 \rightarrow A_0 \rightarrow \text{Type}$$
$$A_2 : \{x\ y\ z : A_0\} \rightarrow A_1\ x\ y \rightarrow A_1\ x\ z \rightarrow A_1\ y\ z \rightarrow \text{Type}$$

...

Conceptual Refinement 2:

We instead seek to construct $\text{SST} : \text{TyStr}$ such that each $A : \text{ty SST}$ is an (infinite) semi-simplicial type

Dependent Semi-Simplicial Types [cont.]

Suppose that we have that $A : \text{ty}$ SST represents a semi-simplicial type

What does $B : \text{ty}$ (ex SST A) represent?

We think of a dependent semi-simplicial type B over A as the following infinite list of data:

$$B_0 : A_0 \rightarrow \text{Type}$$

$$B_1 : \{x\ y : A_0\} \rightarrow A_1\ x\ y \rightarrow B_0\ x \rightarrow B_0\ y \rightarrow \text{Type}$$

$$\begin{aligned} B_2 : & \{x\ y\ z : A_0\} \{ \alpha : A_1\ x\ y \} \{ \beta : A_1\ x\ z \} \{ \gamma : A_1\ y\ z \} \\ & \{ x' : B_0\ x \} \{ y' : B_0\ y \} \{ z' : B_0\ z \} \rightarrow A_2\ \alpha\ \beta\ \gamma \\ & \rightarrow B_1\ \alpha\ x'\ y' \rightarrow B_1\ \beta\ x'\ z' \rightarrow B_1\ \gamma\ y'\ z' \rightarrow \text{Type} \end{aligned}$$

...

Dependent Semi-Simplicial Types [cont.]

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

$$C_0 : \{x : A_0\} \rightarrow B_0 \ x \rightarrow \text{Type}$$

$$C_1 : \{x \ y : A_0\} \{ \alpha : A_1 \ x \ y \} \{ x' : B_0 \ x \} \{ y' : B_0 \ y \} \\ \rightarrow B_1 \ \alpha \ x' \ y' \rightarrow C_0 \ x' \rightarrow C_0 \ y' \rightarrow \text{Type}$$

...

A dependent n -simplex in B is indexed by a filled in dependent n -simplex in A , as well as a lift of its boundary to B

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

etc.

SSTs are a Dependent Type Theory

By *dependent type theories*, I mean *B-systems*

B-systems have: *weakening, elements, shifts (substitution), and zero-variables*

This data is subject to nine axioms

On paper, we can show that the TyStr SST described above can be extended to the full structure of a B-system

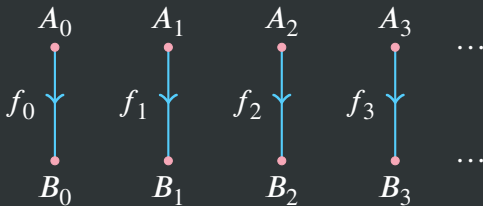
KEY OBSERVATION: *To construct SSTs, you need to work in greater generality!*

Generalised Induction

Consider the type of *type streams*:

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

Note that type streams have an appropriate notion of morphisms (*ladder rungs*)



Generalised Induction [cont.]

What would it mean to inductively define $\text{Foo} : \text{TyStream}$?

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

This definition is written in the internal logic of type streams

In this case $S : \text{Foo} \rightarrow \text{tail Foo}$ is a morphism of type streams

$Z : \text{head Foo}$, and S allows us to promote terms down the stream

Morally, Foo should be equivalent to a stream of units

This kind of inductive declaration makes sense

Defining SSTs

Recall that we have:

```
codata TyStr where  
  ty : TyStr → Type  
  ex : ( $\mathcal{T}$  : TyStr) → ty  $\mathcal{T}$  → TyStr
```

We are now able to give a preliminary definition of semi-simplicial types:

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

This is a TyStr-valued coinductive definition in the internal language of TyStr

Intuition for the Definition

Given any (possibly dependent) SST A , $Z A$ gives the 0-simplices of A

If $x : Z A$ is a 0-simplex, $S A x$ is the *slice* of A over x

The simplices of the slice are *mapping objects*

$$Z A \quad Z (S A x) \quad Z (S (S A x) (y, \alpha))$$

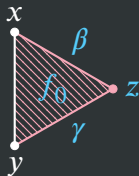
x

A single red dot representing the 0-simplex x.

x
 α
 y

A vertical red line segment with a white dot at the top labeled x and a red dot at the bottom labeled y. The segment is labeled alpha in the middle.

x
 β
 f_0
 γ
 y
 z

A triangle with vertices x (top), y (bottom left), and z (bottom right). The edges are labeled alpha (between x and y), beta (between x and z), and gamma (between y and z). The interior of the triangle is shaded with red diagonal lines and labeled f_0.

The number of new datum in successive slices goes up by a factor of two

The Z Projection

We have that $Z : \text{SST} \rightarrow \text{Type}$

This should be a morphism of TyStrs , so we need to interpret Type as a TyStr

We construct the universe as follows:

$$\begin{aligned} \mathcal{U}' &: \text{Type} \rightarrow \text{TyStr} \\ \text{ty } (\mathcal{U}' \ A) &= A \rightarrow \text{Type} \\ \text{ex } (\mathcal{U}' \ A) \ B &= \mathcal{U}' \ (\Sigma \ A \ B) \end{aligned}$$
$$\begin{aligned} \mathcal{U} &: \text{TyStr} \\ \text{ty } \mathcal{U} &= \text{Type} \\ \text{ex } \mathcal{U} \ A &= \mathcal{U}' \ A \end{aligned}$$

Dependent Type Structures

A morphism $\mathcal{T} \rightarrow \mathcal{U}$ is equivalent to a dependent type structure over \mathcal{T}

We can define this data as follows:

```
codata TyStrd ( $\mathcal{T} : \text{TyStr}$ ) where  
   $ty^d : \text{TyStr}^d \mathcal{T} \rightarrow ty \mathcal{T} \rightarrow \text{Type}$   
   $ex^d : (\mathcal{S} : \text{TyStr}^d \mathcal{T}) (A : ty \mathcal{T}) \rightarrow ty^d \mathcal{S} A$   
         $\rightarrow \text{TyStr}^d (ex \mathcal{T} A)$ 
```

We will work under the assumption that $Z : \text{TyStr}^d \text{SST}$

The S Projection

We have that $S : (X : \text{SST}) \rightarrow Z\ X \rightarrow \text{ex SST } X$

This should be a morphism of dependent type structures over SST

We thus need to define $EX\ \mathcal{T} : \text{TyStr}^d\ \mathcal{T}$

In general, suppose that we have $\Gamma = A \leftarrow B \leftarrow C \leftarrow D : \text{Ctx}\ \mathcal{T}$

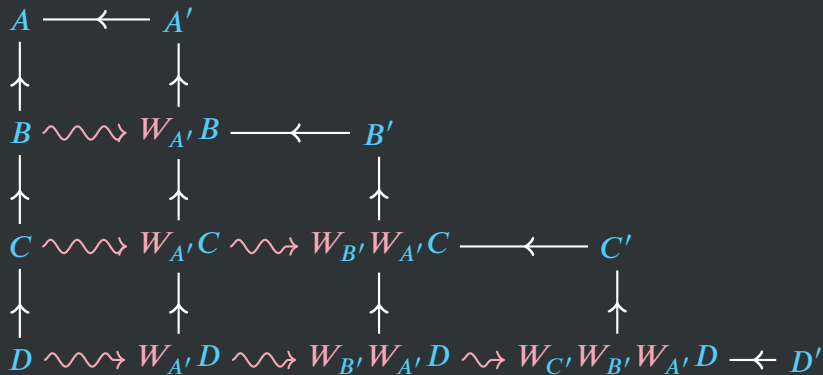
To define $\mathcal{S} : \text{TyStr}^d\ \mathcal{T}$, we have to specify the Γ -indexed dependent contexts

Thus we must give meaning to $\Gamma' = A' \leftarrow B' \leftarrow C' \leftarrow D' : \text{Ctx}^d\ \mathcal{S}\ \Gamma$

The EX Diagram

In order to define $EX \mathcal{T}$, we need $\mathcal{T} : \text{TyStr}$ to be equipped with *weakening*

The meaning of $\Gamma' = A' \Leftarrow B' \Leftarrow C' \Leftarrow D' : \text{Ctx}^d (EX \mathcal{T}) \Gamma$ is:



Simplex Extraction

Suppose that we have $\mathcal{T} : \text{TyStr}$, equipped with a $\mathcal{W} : \text{WkStr } \mathcal{T}$

A *ZS-structure* on $(\mathcal{T}, \mathcal{W})$ consists of $Z : \text{TyStr}^d \mathcal{T}$, and S , a morphism of TyStr^d s over \mathcal{T} from Z to $\text{EX } \mathcal{T}$

S induces, in particular, maps $\text{Ctx}^d Z \Gamma \rightarrow \text{Ctx}^d (\text{EX } \mathcal{T}) \Gamma$

With this data, every $A : \text{ty } \mathcal{T}$ becomes an (infinite) SST

Similarly, every $B : \text{ty } (\text{ex } \mathcal{T} A)$ becomes a dependent SST

How do we extract A_0, A_1, A_2 , etc. from A ?

Simplex Extraction [cont.]

We start with $A : ty \mathcal{T}$, this gives us $\Gamma_0 = \emptyset \vdash A : Ctx \mathcal{T} 1$

The type A_0 of 0-simplices is given as $ty^d \ Z A : Type$

If we choose $x : ty^d \ Z A$, then we obtain a Z -section $\emptyset \vdash x : Ctx^d \ Z \Gamma_0$

Applying S to this Z -section turns it into $\emptyset \vdash S x : Ctx^d (EX \mathcal{T}) \Gamma_0$

By the EX -diagram, this EX -section defines $\Gamma_1 = \emptyset \vdash A \vdash S x : Ctx \mathcal{T} 2$

Next we form a Z -section of this context: $\emptyset \vdash y \vdash \alpha : Ctx^d \ Z \Gamma_1$

Note that $y : ty^d \ Z A$, so it's a 0-simplex as well

$\alpha : ty^d (ex^d \ Z A y) (S x)$ has the type of 1-simplices with boundary x and y

Simplex Extraction [cont.]

Next, we apply S to $\emptyset \vdash y \vdash \alpha$ to get $\emptyset \vdash S y \vdash S \alpha : \text{Ctx}^d (EX \mathcal{T}) \Gamma_1$

By the EX -diagram, we get $\Gamma_2 = \emptyset \vdash A \vdash S y \vdash W_{S y} (S x) \vdash S \alpha : \text{Ctx} \mathcal{T} 4$

We Z -section again via $\emptyset \vdash z \vdash \beta \vdash \gamma \vdash f_0 : \text{Ctx}^d Z \Gamma_2$

We have that $z : ty^d Z A$ and $\beta : ty^d (ex^d Z A z) (S y)$

So z is a 0-simplex, and β is a 1-simplex joining y to z

We have that:

$$f_0 : ty^d (ex^d (ex^d (ex^d Z A z) (S y) \beta) (W_{S y} (S x)) \gamma) (S \alpha)$$

Then f_0 has the type of 2-simplicies with boundary $x, y, z, \alpha, \beta, \gamma$

Simplex Extraction [cont.]

We previously sectioned $\Gamma_2 = \emptyset \vdash A \vdash S y \vdash W_{S y} (S x) \vdash S \alpha$

Then, in $\emptyset \vdash z \vdash \beta \vdash \gamma \vdash f_0 : Ctx^d \vdash \Gamma_2$, we have:

$$\gamma : ty^d (ex^d (ex^d \vdash A z) (S y) \beta) (W_{S y} (S x))$$

This ought to be a 1-simplex joining x to z , i.e. of type $ty^d (ex^d \vdash A z) (S x)$

The type that we see falsely suggests that γ depends on y and β

Morally, the weakening should cancel the extension

This would follow if \vdash preserved weakening

In the category of TyStrs with weakening, morphisms should preserve weakening

Status Report

In present-day Agda, the universal property constructs *unreduced simplex types*

With rewriting, we can see that if Z and S preserve weakening definitionally, then we extract the correct reduced simplex types

We are working on a type theory which lets us work with TyStrs with weakening

Further, in any example of universal property data in which preservation is definitional, we can construct the corresponding SSTs in present-day Agda

In particular, we are able to construct the *singular semi-simplicial types*

Singular SSTs

Given a type X , we think of X as a space

A space has points, lines, triangles, etc.

The corresponding SST is called $\text{Sing } X$

How do we use our universal property to construct this?

Singular SSTs [cont.]

We are going to construct singular SSTs from some type structure \mathcal{T}

It turns out that we have to suitably generalise by taking $\mathcal{T} = \mathcal{U}$

This has a standard weakening structure (*we're weakening types*)

Z is defined to be the *tautological dependent type structure* on \mathcal{U}

Suppose that $\Gamma = \emptyset \vdash A \vdash B \vdash C : \text{Ctx } \mathcal{U} \ 3$

Then $\emptyset \vdash x \vdash y \vdash z : \text{Ctx}^d \ Z \ \Gamma$ consists of $x : A$, $y : B \ x$, and $z : C \ (x, \ y)$

S is defined such that for $x : A$, we have $S \ x = \lambda \ y. \ x \equiv y$

For the higher cases, we use cleverness involving cubical dependent path types

Singular SSTs [cont.]

$Z' : \{X : \text{Type } \ell\} \rightarrow X \rightarrow \text{ZStr } (\mathcal{U}' X) \ell$

$ty^Z (Z' x) A = A \times$

$ex^Z (Z' x) a = Z' (x , a)$

$Z : \text{ZStr } (\mathcal{U} \ell) \ell$

$ty^Z Z A = A$

$ex^Z Z a = Z' a$

$S' : \{X Y : \text{Type } \ell\} (f : X \rightarrow Y) (y : Y) (p : (x : X) \rightarrow y \equiv f x) \rightarrow$

$\text{SStr } (\text{replace } f) (Z' y) (\mathcal{W}\text{-}\mathcal{U}' X)$

$fun^S (S' f y p) \{A\} a_0 = \lambda \{(x , a_1) \rightarrow \text{PathP } (\lambda i \rightarrow A (p \times i)) a_0 a_1\}$

$up^S (S' f y p) \{A\} a =$

$S' (\lambda x \rightarrow f (\text{fst } (\text{fst } x)) , \text{snd } (\text{fst } x)) (y , a)$

$(\lambda x i \rightarrow p (\text{fst } (\text{fst } x)) i , \text{snd } x i)$

$S : \text{SStr } (\text{idTyMor } (\mathcal{U} \ell)) Z \mathcal{W}\text{-}\mathcal{U}$

$fun^S S a_0 a_1 = a_0 \equiv a_1$

$up^S S a = S' \text{fst } a \text{snd}$

Singular SSTs [cont.]

Applying the simplex extraction algorithm in Agda yields the following output:

 $x : X$ $y : X$ $\alpha : x \equiv y$ $z : X$ $\beta : y \equiv z$ $\gamma : x \equiv z$ $f_0 : \text{PathP } (\lambda i \rightarrow x \equiv \beta i) \ \alpha \ \gamma$ $\Delta_0 : \text{PathP } (\lambda i \rightarrow \text{PathP } (\lambda j \rightarrow x \equiv f_1 i j) \ \alpha \ (f_2 i)) \ f_0 \ f_3$ $w : X$ $\delta : z \equiv w$ $\epsilon : y \equiv w$ $f_1 : \text{PathP } (\lambda i \rightarrow y \equiv \delta i) \ \beta \ \epsilon$ $\zeta : x \equiv w$ $f_2 : \text{PathP } (\lambda i \rightarrow y \equiv \delta i) \ \gamma \ \zeta$ $f_3 : \text{PathP } (\lambda i \rightarrow y \equiv \epsilon i) \ \alpha \ \zeta$

Summary

α

We can codify the pattern in SSTs via a simple inverse category

Audience Puzzle: show that this leads to externally well-typed expressions

However, it seems that these cannot be internalised

β

When working with TyStrs with weakening, SST acquires an universal property

We can express this in terms of non-Type-valued coinduction

This universal property lets us construct singular semi-simplicial types

The type theory that lets us do this in generality is under construction

Thank you for listening to my talk!

A research writeup and associated code may be found at:

<https://github.com/FrozenWinters/SSTs>