Higher inductive types.

Homotopy type theory = MLTT + UA + higher inductive types

Recall: inductive types are generated by their constructors (terms)/ canonical terms

Since we now consider types as having
- terms
- equalities
- equalities between equalities

We can consider higher inductive types, whose constructors can be terms, equalities, equalities between equalities, etc.

Ex. $S^0 :=$ bool has constructors
- true : bool
- false : bool

Def. D^1 (the interval) has constructors
- true : D^1
- false : D^1
- $p : \text{true} =_{D^1} \text{false}$

We could define D^1 with four rules:

$$\frac{}{D' \text{ type}} \quad D'\text{-form}$$

$$\frac{}{\text{true} : D'} \qquad \frac{}{\text{false} : D'} \qquad \frac{}{p : \text{true} =_{D'} \text{false}} \qquad D'\text{-intro}$$

$$\frac{\begin{array}{l} d : D' \vdash E(d) \text{ Type} \\ \vdash t : E(\text{true}) \\ \vdash f : E(\text{false}) \\ \vdash \pi : p_* t = f : E(\text{false}) \end{array}}{d : D' \vdash \text{ind}_{D',t,f,\pi}(d) : E(d)} \quad \Big\} \; D'\text{-elim}$$

$$\left.\begin{array}{l} \vdash \text{ind}_{D',t,f,\pi}(\text{true}) \doteq t : E(\text{true}) \\ \vdash \text{ind}_{D',t,f,\pi}(\text{false}) \doteq f : E(\text{false}) \\ \vdash \text{ind}_{D',t,f,\pi}(p) \doteq \pi : p_* t = f : E(\text{false}) \end{array}\right\} \; D'\text{-comp}$$

__Def__. S^1 (the circle) has constructors

- $\text{base} : S^1$
- $\text{loop} : \text{base} = \text{base}$

__Thm__. $\pi_1(S^1) = \mathbb{Z}$ where

$\pi_1 : \mathcal{T} \to \mathcal{S}et$ is defined as

$S^1 \to \mathcal{T} \ldots ?$

- We want to make this a set.
- We also want to make thing into propositions.

<u>Ex.</u> Given $P, Q : Prop$

$P + Q$ is not a proposition in general

<u>Thm.</u> • If $P, Q : Prop$, $P \times Q : Prop$ (write $P \wedge Q$)

• If $P : Prop$, $E : P \to Prop$,

$\sum_{p:P} e(p) \quad \prod_{p:P} e(p) \quad \left(\text{write } \exists_{p:P} e(p) \quad \forall_{p:P} e(p) \right)$

are in Prop.

<u>Def.</u> Given a type T, the <u>propositional truncation $\|T\|_1$ of T</u> is the higher inductive type with constructors

- $|-|_1 : T \to \|T\|_1$
- $\prod_{x,y:T} |x|_1 =_{\|T\|_1} |y|_1$

<u>Ex.</u> Show that for any type T, $\|T\|_1$ is a proposition.

<u>Def.</u> Define $P \vee Q := \|P + Q\|_1$ for $P, Q : Prop$.

<u>Ex.</u> Functions $(T \to P) \simeq (\|T\|_1 \to P)$ for any type T, proposition P.

Def. Given a type T, the <u>set truncation $\|T\|_2$ of T</u> is the higher inductive type with constructor

- $|-|_2 : T \to \|T\|_2$
- $\prod_{\substack{x,y:T \\ p,q:x=y}} |p|_2 = |q|_2$.

Ex. Show that for any type T, $\|T\|_2$ is a set.

Def. $\pi_1 : U \to Set$
$:= \lambda T.\ \|S^1 \to T\|_2$.

Thm. $\pi_1(S^1) = \mathbb{Z}$.

Interfaces and the Rezk completion

- The result by Coquand-Danielsson says that
$$(A =_{alg} B) \simeq (A \cong B)$$
where alg is some type of sets with algebraic structure.

Thus $\to$ one cannot say anything that distinguishes isomorphic A, B

- Consider a converse problem:

  Suppose we specify a limited language / interface with which we can talk about an algebraic structure. (Ahrens, North, Shulman, Tsementzis)

Ex. $p\mathcal{U} := \sum_{T:\mathcal{U}} T \to \mathrm{Prop}$

Have $\mathrm{Mon} \to p\mathcal{U}$

$$(M, e, m, -, -) \mapsto (M, \lambda x. x = e).$$

Given $(T, p) : p\mathcal{U}$, think of p as the language or interface we use to reason about T.

Then we can only tell the difference between $s, t : T$ if we can tell the difference between $p(s), p(t)$.

Say s, t are <u>indiscernable</u> if

$$(s \asymp t) :\equiv (p(s) = p(t)).$$

Can quotient T by this relation:

$T/\asymp$ has constructors $i : T \to T/\asymp$,

$$j : \prod_{s,t:T} s \asymp t \to is =_{T/\asymp} it$$

<u>NB!</u> When we quotient types like this we are <u>adding</u> things to the types.

(A normal quotient in set theory results in less things — equivalence classes — and are very hard to work with formally.)

Quotients in functional programming languages haven't been successful until now — quotients in HoTT and adjacent languages, even Haskell by importing these ideas.

- So, can define $\mathbb{Q}$ as a quotient of $\mathbb{N} \times \mathbb{N}$, $\mathbb{R}$ as a quotient of sequences of $\mathbb{Q}$, etc.
- Programming languages are also complicated quotients of inductive data types, so a lot of research is about understanding/justifying these types of quotients so that we can formally reason about programming languages.

<u>NB continued</u>: We add things, so we don't change the underlying terms.
- At the computer/$\equiv$ level: nothing changes
- At the human/$=$ level: quotient

Ex. (Angiuli - Cavallo - Mörtberg - Zeuner)

Consider queues:

$$\mathrm{Queues}(A) := \sum_{Q:\mathcal{U}} Q \times (A \to Q \to Q) \times (Q \to \mathrm{Maybe}\,(Q \times A))$$

(record type)
$$\begin{cases} Q : \mathcal{U} \\ \mathrm{empty} : Q \\ \mathrm{enqueue} : A \to Q \to Q \\ \mathrm{dequeue} : Q \to \mathrm{Maybe}\,(Q \times A) \end{cases}$$

This is an <u>interface</u>.

$\mathrm{List}(A)$ is the inductive type with constructors

$\mathrm{empty}_L : \mathrm{List}(A)$

$\mathrm{enqueue}_L : A \to Q \to Q$

($\mathrm{dequeue}_L$ can be constructed to return (list w/o last element, last element))

So $(\mathrm{List}(A), \mathrm{empty}_L, \mathrm{enqueue}_L, \mathrm{dequeue}_L) : \mathrm{Queues}(A)$
and this type is easy to prove things about.

But there are better representations in terms of efficiency:

$\mathrm{BatchedList}(A) := \mathrm{List}(A) \times \mathrm{List}(A)$

$\quad \mathrm{empty}_B := (\mathrm{empty}, \mathrm{empty})$

$\quad \mathrm{enqueue}_B\,(a, (\ell, m)) := (\mathrm{enqueue}_B\,(a, \ell), m)$

$\quad \mathrm{dequeue}_B\,(\ell, m) := ((\ell, m \text{ w/o first element}), \text{ first element of } m)$

$\qquad\qquad\qquad\qquad$ o/w $\quad (\mathrm{dequeue}_L\ \ell, \mathrm{empty})$

So this is also a term of Queues(A).

There's a surjection $\mathrm{BatchedList}(A) \xrightarrow{s} \mathrm{List}(A)$

$$(\ell, m) \longmapsto \text{concat } \ell\ (\text{reverse } m)$$

that respects the operations given by the interface.

We can quotient BatchedList(A) by $(b \sim c) := (s(b) = s(c))$ ged

Then

- at the computer/efficiency/$\equiv$ level: nothing has chan
- at the human/$=$ level, we get $\mathrm{BatchedList}(A) = \mathrm{List}(A)$.

So proofs about the correctness of programs using List(A), etc can be applied to BatchedList(A).