Questions from yesterday:

- Discussions around value judgements and dynamics
  - → this is indeed making our language lazy, which we will explore further in Week 7
- why rules have one conclusion
  - → standard
  - → 2 conclusions is valid, but can lead to messy proofs so the standard is 1 conclusion
  - → speak to Tom if you are interested

(found no examples of 2 conclusions in the literature, probs cos non-standard / not my specialty)

~Min sheet~

Sums

types $\tau ::= \ldots \mid \tau_1 + \tau_2 \mid \mathbf{0}$

pre-terms $e ::= \ldots \mid \mathsf{abort}(e) \mid \mathsf{inl}(e) \mid \mathsf{inr}(e) \mid \mathsf{case}(e; x.e_1; y.e_2)$

Statics:

$$\text{ABORT}\ \frac{\Gamma \vdash e : \mathbf{0}}{\Gamma \vdash \mathsf{abort}(e) : \tau}$$

$$\text{INL}\ \frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{inl}(e) : \tau_1 + \tau_2}$$

$$\text{INR}\ \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{inr}(e) : \tau_1 + \tau_2}$$

$$\text{CASE}\ \frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x : \tau_1 \vdash e_1 : \sigma \quad \Gamma, y : \tau_2 \vdash e_2 : \sigma}{\Gamma \vdash \mathsf{case}(e; x.e_1; y.e_2) : \sigma}$$

Dynamics:

$$\text{VAL-INL}\ \frac{}{\mathsf{inl}(e)\ \mathsf{val}}$$

$$\text{VAL-INR}\ \frac{}{\mathsf{inr}(e)\ \mathsf{val}}$$

$$\text{D-ABORT-1}\ \frac{e \longmapsto e'}{\mathsf{abort}(e) \longmapsto \mathsf{abort}(e')}$$

$$\text{D-CASE-INL}\ \frac{}{\mathsf{case}(\mathsf{inl}(e); x.e_1; y.e_2) \longmapsto e_1[e/x]}$$

$$\text{D-CASE-INR}\ \frac{}{\mathsf{case}(\mathsf{inr}(e); x.e_1; y.e_2) \longmapsto e_2[e/y]}$$

$$\text{D-CASE-1}\ \frac{e \longmapsto e'}{\mathsf{case}(e; x.e_1; y.e_2) \longmapsto \mathsf{case}(e'; x.e_1; y.e_2)}$$

Substitution

$$\langle e_1, e_2 \rangle [e/x] \stackrel{def}{\equiv} \langle e_1[e/x], e_2[e/x] \rangle$$

$$\pi_i(u)[e/x] \stackrel{def}{\equiv} \pi_i(u[e/x]) \quad i \in [1,2]$$

$$inl(u)[e/x] \stackrel{def}{\equiv} inl(u[e/x])$$

$$inr(u)[e/x] \stackrel{def}{\equiv} inr(u[e/x])$$

$$case(u; z.e_1; y.e_2)[e/x]$$

$$\stackrel{def}{\equiv} case(u[e/x]; z.e_1[e/x]; y.e_2[e/x])$$

Functions

$$(\lambda x.x) : a \to a$$

Syntax:

types $\tau ::= \ldots \mid \tau_1 \to \tau_2$

preterms $e ::= \ldots \mid \lambda x{:}\tau.e \mid e_1(e_2)$

Statics:

$$\text{LAM}\ \frac{\Gamma, x{:}\sigma \vdash e : \tau}{\Gamma \vdash \lambda x{:}\sigma.e : \sigma \to \tau}$$

$$\text{APP}\ \frac{\Gamma \vdash e_1 : \sigma \to \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1(e_2) : \tau}$$

Example:

$\lambda x{:}\text{Num}.\ x + x$

plus(x,x)

Dynamics:

$$\text{VAL-LAM}\ \frac{}{\lambda x{:}\tau.e\ \text{val}}$$

$$\text{D-APP-1}\ \frac{e_1 \mapsto e_1'}{e_1(e_2) \mapsto e_1'(e_2)}$$

$$\text{D-BETA}\ \frac{}{(\lambda x{:}\tau.e)(e_2) \mapsto e_1[e_2/x]}$$

$\to_\beta$

Subst:

$$(e_1(e_2))[e/x] \stackrel{\text{def}}{=} (e_1[e/x](e_2[e/x]))$$

$$(\lambda y{:}\tau.u)[e/x] \stackrel{\text{def}}{=} (\lambda y{:}\tau.u[e/x])$$

$$\text{add} : \text{Num} \to \text{Num} \to \text{Num}$$

$\lambda x{:}\text{Num}.\ \lambda y{:}\text{Num}.\ x + y$

$\tau = \text{Num} \to \text{Num} \qquad \Gamma = \emptyset$

$\sigma = \text{Num}$

$\sigma' = \text{Num}$

$$\text{VAR}\ \frac{}{\Gamma, x{:}\sigma, y{:}\sigma' \vdash x : \text{Num}} \qquad \text{VAR}\ \frac{}{\Gamma, x{:}\sigma, y{:}\sigma' \vdash y : \text{Num}}$$

$$\text{PLUS}\ \frac{\Gamma, x{:}\sigma, y{:}\sigma' \vdash x : \text{Num} \quad \Gamma, x{:}\sigma, y{:}\sigma' \vdash y : \text{Num}}{\Gamma, x{:}\sigma, y{:}\sigma' \vdash \text{plus}(x,y) : \text{Num}}$$

$$\text{LAM}\ \frac{\Gamma, x{:}\sigma \vdash \lambda y{:}\text{Num}.\text{plus}(x,y) : \tau}{\Gamma \vdash \lambda x{:}\text{Num}.\lambda y{:}\text{Num}.\text{plus}(x,y) : \sigma \to \tau}$$

$$\text{LAM}$$

This specification of functions supports higher-order (HO) functions

HO functions = functions that take other functions as arguments

e.g.

```
twice :: (a->a) -> a -> a
twice f x = f (f x)
```

$\Lambda a.\ \text{twice}_a := \lambda f : a \to a.\ \lambda x : a.\ f(f(x))$

not simply-typed

Bonus - Inhabitants + Cat Theory

Inhabitants of a type
= the set of values that have that type

```
>data Unit = Unit
>data Bool = True | False
>data Tri = One | Two | Three
>data Void
```

Unit + Bool = 3

Inl ()
Inr True
Inr False

(Unit , Bool) = 2

((), False)
((), True)