Sneed sub little low.

→ what barriers?

The Simply Typed Lambda Calc (STLC)

STLC = constants + sums + products + exponentials

constants → current lang

sums → Either in Haskell

products → corresponds to tuples

exponentials → functions

Products

("Hello", 2) :: (String, Int)
() :: ()

syntactic chart

types $\tau ::= \ldots$
$\quad \tau_1 \times \tau_2$ -- product
$\quad 1$ -- unit

pre-term $e ::= \ldots$
$\quad \langle e_1, e_2 \rangle$ -- pair
$\quad \pi_1(e)$ -- fst proj
$\quad \pi_2(e)$ -- snd proj
$\quad \langle \rangle$

- introduction rules create instances of a type
- elimination rules destruct / project from instances of a type

Statics:

UNIT $\dfrac{}{\Gamma \vdash \langle \rangle : 1}$

PAIR (PROD) $\dfrac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}$

PROJ-1 $\dfrac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_1(e) : \tau_1}$

PROJ-2 $\dfrac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_2(e) : \tau_2}$

Dynamics:

VAL-UNIT $\dfrac{}{\langle \rangle \text{ val}}$

VAL-PROD $\dfrac{(e_1 \text{ val} \quad e_2 \text{ val})}{\langle e_1, e_2 \rangle \text{ val}}$

$\pi_1(\langle e_1, e_2 \rangle)$
$\downarrow$
e_1

This is an axiom cos our language is lazy.
With the dynamics we will specify that we must fully eval a pair before we can proj from it

$\dfrac{}{\pi_1(\langle e_1, e_2 \rangle) \longmapsto e_1}$ D-PROJ-TUPLE-1

$\dfrac{}{\pi_2\langle e_1, e_2 \rangle \longmapsto e_2}$ D-PROJ-TUPLE-2

$\dfrac{e \longmapsto e'}{\pi_1(e) \longmapsto \pi_1(e')}$ D-PROJ-1

$\dfrac{e \longmapsto e'}{\pi_2(e) \longmapsto \pi_2(e')}$ D-PROJ-2

Sums

```
> data Either a b = Left a
>                 | Right b

> data Void

> abort :: Void -> ()
> abort _ = ()

> abort :: Void -> Bool
> abort _ = True
```

types $T ::= \ldots$
$\quad T_1 + T_2$
$\quad 0$

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

```
fromEither :: Either a b -> Maybe a
fromEither (Left a) = Just a
fromEither (Right b) = Nothing

fromEither e = case e of
    (Left x) -> e1
    (Right y) -> e2
```