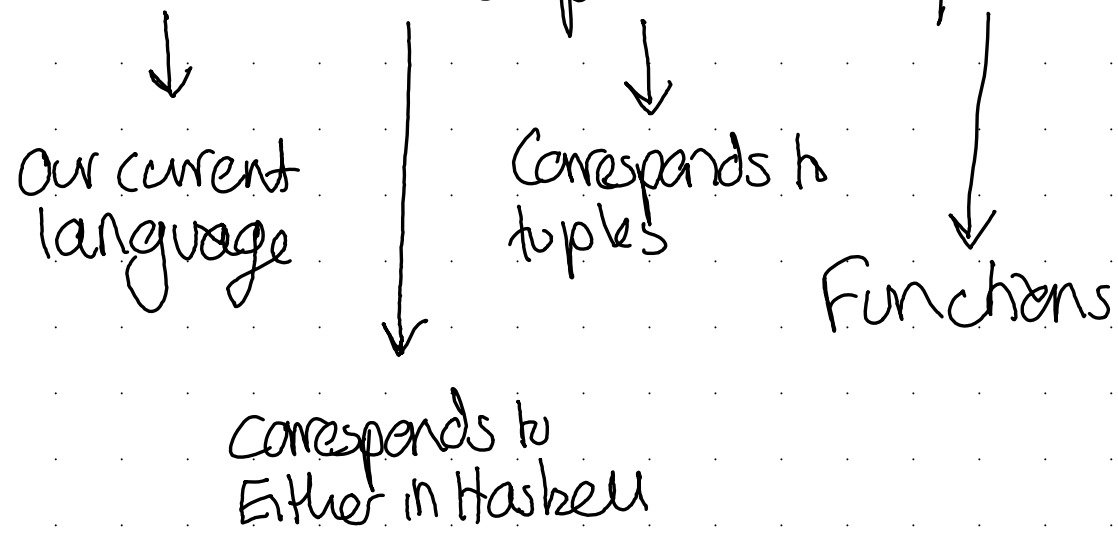
~minsheet~

The Simply Typed Lambda Calc (STLC)

This week, we will expand our little language to the STLC, which consists of:

STLC = constants + sums + products + exponentials



For sums and products, we will do the expansion together as a guided consolidation of all that we have learnt so far (syntax, statics, dynamics, and making them "good")

When it comes to functions, I will be prescriptive so that you can see the standard function rules that you will see time and again in PL papers.

The key exercise in the sheet paired with this content is proving type safety for the new functions fragment.

Products

You've met products in many programming languages. In Haskell, for example, they look like this:

("Hello", 2) :: (Str, Num) -- pair / binary product

() : () -- unit / nullary product.

Pairs are useful because they allow us to write functions that return two values.

Unit is useful because it allows us to write functions that return nothing

~check comfort with product construct~

We will extend our syntactic chart as follows to include products:

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

pre-terms $e ::= \ldots$
$\langle e_1, e_2 \rangle$ pair constructor
$\pi_1(e)$ first projection
$\pi_2(e)$ second projection
$\langle \rangle$ unit

I'll now get you to help me write the statics / typing judgements for products.

As we do so, we can consider two categories of statics:

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

Q what should the introduction rule be for unit?

$$\text{UNIT}\ \frac{}{\Gamma \vdash \langle \rangle : 1}$$

Q Intro rule for pairs?

$$\text{PROD}\ \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}$$

Q Elim rule(s) for pairs

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

$$\text{PROJ-1}\ \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_2(e) : \tau_2}$$

Now lets do the dynamics, remembering that we want to control the eval order so that it is deterministic.

Let's start with extending our value judgement.

Q What new things, if any do we want to classify as a value? (A done computation, a result.)

$\langle \rangle$ and $\langle e_1, e_2 \rangle$ because we want to return both tuples and units as results

Q what does the value judgement look like for $\langle \rangle$?

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

Q Pair?

$$\text{VAL-PAIR}\ \frac{}{\langle e_1, e_2 \rangle\ \text{val}}$$

Now for the dynamics, since we have specified $\langle \rangle$ and pairs as values, our dynamics need only specify how they interact with projections π_1 and π_2, and we want to ensure that it is deterministic as to when to project, and when to evaluate the term we are projecting from. We can do this by only projecting from fully evaluated pairs

Q what might projecting the first elem from a fully evaluated pair look like in ruleform?

$$\text{D-PROJ-TUPLE-1}\ \frac{}{\pi_1(\langle e_1, e_2 \rangle) \mapsto e_1}$$

using pattern matching required to name e_1 and signal value

Q π_2?

$$\text{D-PROJ-TUPLE-2}\ \frac{}{\pi_2(\langle e_1, e_2 \rangle) \mapsto e_2}$$

Q what about getting to this state, working on the term : $\tau_1 \times \tau_2$ to get a value that we can project from?

$$\text{D-PROJ-1}\ \frac{e \mapsto e'}{\pi_1(e) \mapsto \pi_1(e')}$$

$$\text{D-PROJ-2}\ \frac{e \mapsto e'}{\pi_2(e) \mapsto \pi_2(e')}$$

Q Can you think of a (small) term involving product types?

Help me with its typing derivation.

Help me eval it

Sums

You have met sum types before, in Either from Haskell, where it is a binary sum, representing a choice between two values

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

These will be an exciting addition to our language, because they will allow us to pattern match.

~check comfort with sum construct~ (motivate further with errors)

The sum analogue of Unit is interesting. In Haskell, it is Void

```
> data Void
```

It has no constructors, meaning that you cannot create something of type Void. It represents no choice.

Void is interesting because from a logical perspective (via Curry-Howard) we can view it as False. Meaning that we can conclude any thing from it. This is echoed in Haskell with how we can always define a function from void to any other type:

```
> absurd :: Void → ()
> absurd _ = ()

> absurd :: Void → Bool
> absurd _ = True
```

This is always the constant () function since we cannot match on Void See **Bonus** for more

We will extend our syntactic chart as follows to accomodate sums:

types $\tau ::= \ldots$
$\tau_1 + \tau_2$ sum
0 Void

pre-terms $e ::= \ldots$
abort(e)
inl(e) left injection
inr(e) right injection
case(e; x.e1; y.e2) case analysis (proj)

I'll give you the statics for abort, but then plz help me with the rest.

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

If I have the impossible (a value :0), I can do any thing I want. (any thing from False)

Q Typing for InL?

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

Q InR?

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

Q Case? (hardest cos binders)

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

Q What about its dynamics

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

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

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

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

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

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

Q Can you think of a (small) term involving sum types?

Help me with its typing derivation.

Help me evaluate it.

Substitution

Of course, it is not only the language that we must extend, but also our key lemmata.

We will do substitution.

It works much as you would expect, just pushing the subst down into subterms, with us assuming our key the Barendregt convention to ensure case with its binders doesn't go wrong.

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

Q what does this say?

We define subst on a pair to be performing the subst on each elem of the pair

Q what should the rest be?

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

π_i: common short hand

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

renaming key

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

Functions

I hope I don't need to motivate functions.

In our language, we won't name functions, so it will be equivalent to anonymous terms / lambda expressions in Haskell:

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

Syntax:

types $\tau ::= \ldots$
$\tau_1 \to \tau_2$ function type

pre-terms $e ::= \ldots$
$\lambda x{:}\tau . e$ abstraction
$e_1(e_2)$ application

(explicit typing — pointing to $x{:}\tau$)

Statics:

These are very standard, you will see them time and time again in the PL literature:

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

Lam says that the lambda abstraction $\lambda x{:}\sigma.e$ expands the context with $x:\sigma$, allowing the body to be typed with τ.

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

Application says that if e_1 is a function and e_2 has the correct argument type, then the application of the function e_1 to the arg e_2 will have the result type of the function.

Q How would we write a function for doubling a number in the STLC?

$$\lambda z{:}\text{Num}.\,\text{plus}(z,z)$$

For the dynamics, we want to make lambda expressions values, so we only perform the application when the function is fully reduced.

Q How should we specify this via dynamics

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

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

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

note that it is standard to call the act of applying the function (aka subbing in its (first) argument) beta-reduction ($\to_\beta$)

Of course, this use of substitution necessitates us expanding the definition of subst in the meta lang to cover the cases of lambda expressions and applications.

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

Again, we just propagate the application of the subst into the sub-expression

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

Again, we have been careful to employ the Barendregt convention + named carefully

Let's play with our function terms!

Q Can someone help me write the function add : Num → Num → Num

(it might be helpful to write it in Haskell, then translate it).

(see lecture note 8 for solution)

(Q let's justify its well-typedness with a derivation)

(Q lets evaluate it)

The fun thing about the way we have defined functions, is that it supports higher-order functions

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

HO functions = functions that take other functions as an argument

e.g.

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

Q How would we write this in STLC?

(Q Justify its well-typedness)

(Q Evaluate it with the dynamics)

Bonus - Inhabitants + Cat Theory

This is a little enrichment that might shed light on why products and sums are called that, and justify their correctness by glimpsing into the Mathematics behind them

Q In first year when you did Haskell you might have encountered inhabitants of a type.

a) Hands up if you recall that

b) Can someone with their hand up say what they recall.

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

The size of these sets, especially in relation to sums, products and functions (exponentials) explains their names.

```
data Unit = Unit    -- One inhabitant
data Bool = True | False    -- Two inhabitants
data Tri  = One | Two | Three    -- Three inhabs
```

Q How many inhabitants does Void have?

```
data Void    -- Zero inhabitants.
```

So already, we can see why Void is 0 and why Unit is 1.

What about sums?

Unit + Bool has 3 inhabitants:

```
Inl ()
Inr True
Inr False
```

→ we add the inhabitants

Q What do you think happens for products? Justify via example.

(Unit, Bool) has Two inhabitants

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

→ we multiply

Functions are the most interesting because they are exponentials!

```
example1 :: Unit → Bool
example1 _ = True          2 = 2^1

example1' _ = False

example2 :: Bool → Unit
example2 _ = ()            1 = 1^2

example3 :: Tri → Void     0 = 0^3
-- NA --

example4 :: Void → Tri
example4  _ = one      ┐
example4' _ = two      ├ all equiv (we
example4" _ = three    ┘ can never run them)

example4 _ = absurd        1 = 3^0
```

So as we can see, there is a strong correspondence between types and arithmetic

```
Void = 0
Unit = 1
Either = +
pairs = ×
functions = ^
```

In fact, rules that apply to arithmetic also apply to our types and their inhabitants e.g. Void is the unit of Either.

Also operations we do on types such as carrying respect inhabitance in the sense that these equivalent functions have the same number of inhabitants.

This is because Haskell and STLC are mathematically based, and their products come from the maths of maths: category theory. Sums and products and exponentials don't just appear in types and arithmetic, but everywhere. They are very common structures and category theory works at such a high level of abstraction that it unifies them.

```
Unit = terminal object of Hask
Void = initial object of Hask
Either = coproducts of Hask
pairs = product objects in Hask
functions = Exponential objects in Hask
```

See Bartosz Milewski's "Function Types" post on his Programming Cafe blog for more

Also of interest might be Conor McBride's extended abstract "The Derivative of a Regular Type is its Type of One-Hole Contexts", which is an example of the sort of paper that this unit hopes to equip you to read.