Intro

Last time we met our basic tools (judgements as rules) and how to prove things with/about them (derivations + induction)

~min sheet~

This week we are going to introduce our little language, and because the language we are going to build this unit is "ideal" based on research in Programming Languages (PL), of course we are going to start with types!

As we enter the second week you can start to appreciate the cadence of the unit:

- last week in lecture 1, we introduced rules and judgements, then in lecture 2 we proved things about them (induction)
- this week, we will intro new rules that describe our language in lecture 1, then in lecture 2 (soon if we are fast) we will prove things about them

Statics vs. Dynamics

We are going to split the life of a compile program into two phases

- static = any thing that happens before running the program (compile time)
  e.g. parsing...

> Q Can you think of other things?

  ... lexing, staged computation, type-checking, static analysis, optimisations

- dynamic = everything that happens when a program is running (runtime)
  e.g. calculating the result...

> Q Can you think of more?

  ... exceptions, side effects

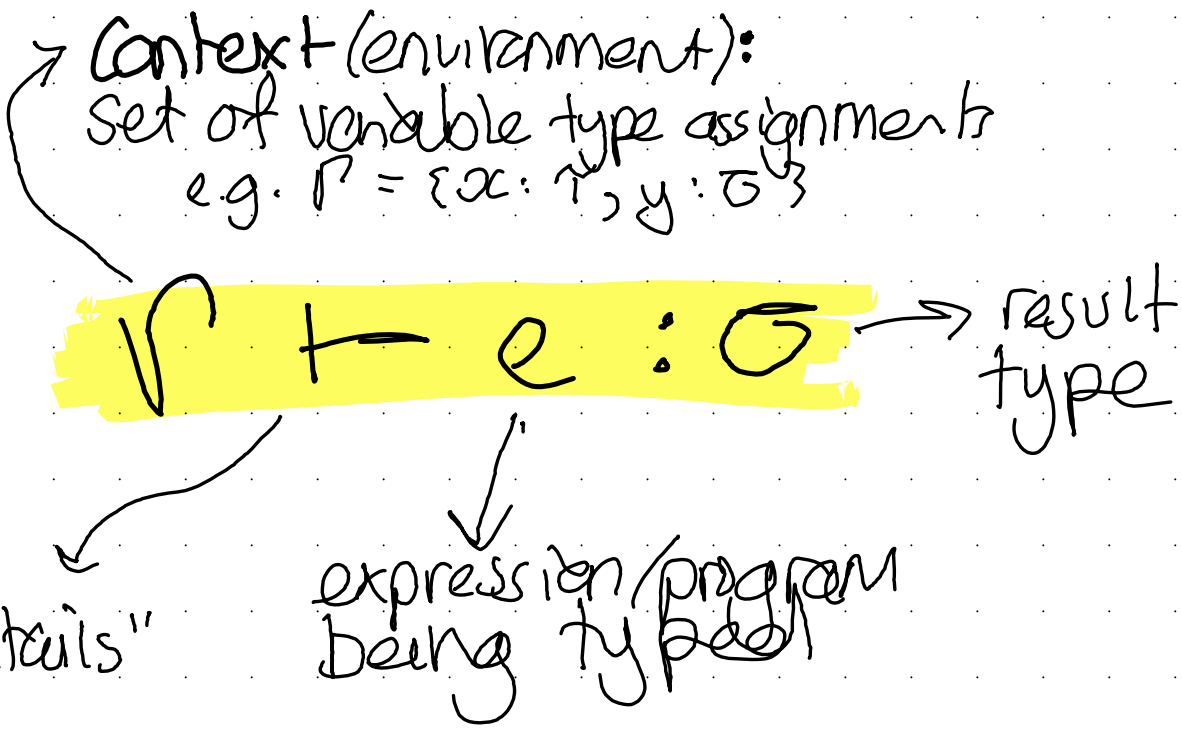
Like I said, as PL people, we love the statics of a language since we believe it's the best foundation (better to get errors at compile-time instead of runtime)

So we will start there

Typing Judgements

The syntax of a typing judgement is more involved than the nat judgement we met last time. This is because there are more components.

Here is the syntax with components labelled:



Note that in this course, we will only call well typed programs terms. Anything of the same shape that is not necessarily well-typed will be refered to as a preterm.

Terms = well-typed pre-terms

Before we meet our language, and see its typing rules, we must first chat about binders.

Binders

Binder = a language construct that closes over (binds) variables

e.g. $\forall$ or $\exists$ from maths...

Q Can you think of any other binders? Perhaps some from programming languages

... λ or let or where from Haskell

Bound variables = variables bound by a binder

Q Can you guess what a free variable is

Free variables = variables not bound by a binder

Q In this expr which variables are free?
Q Which are bound?

$\lambda x.xy$

binder, bound, free

Closed expr = one with no free variables
Open expr = one with free variables

Q Can you give me an example open expr.?
Q A closed one?

Open expr: $\lambda x.y$ ← free

Closed expr: $\lambda x.x$ ← bound

α-equivalence/convertibility states that the name of a bound variable doesn't matter

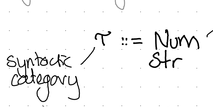
i.e. $(\lambda x.x) =_\alpha (\lambda y.y)$

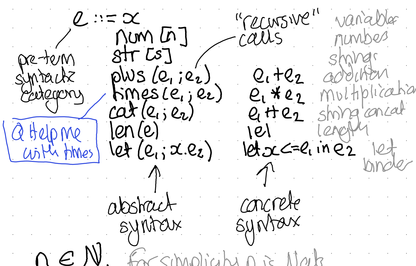
Q Can anyone suggest more alpha equiv expressions?

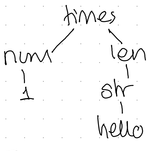
$\lambda xy.x =_\alpha \lambda yx.y$

$\lambda xy.y =_\alpha \lambda yx.x$

It is just the occurance/position of the named var that matters, not its name.







A little language of numbers and strings - Syntax

Syntax can be specified using syntactic charts. The type we will use is called:

Backus-Naur form (BNF)

Our little language has two syntactic categories: the syntax of types, and the syntax of pre-terms

There are two types: numbers and strings, and we write the type of numbers as Num and the type of strings is written Str.

$$\tau ::= \begin{array}{l} \text{Num} \\ \text{Str} \end{array}$$

syntactic category

concrete syntax

Our pre-terms are more complex as they have both concrete and abstract syntax, and "recursive" calls

Abstract syntax = the syntax of the term as it should appear in the abstract syntax tree

Concrete syntax = a user-friendly abbreviation of the abstract syntax

$e ::= x$ — variables

$\quad$ num [n] — numbers

$\quad$ str [s] — strings

$\quad$ plus $(e_1; e_2)$ — $e_1 + e_2$ — addition

$\quad$ times $(e_1; e_2)$ — $e_1 * e_2$ — multiplication

$\quad$ cat $(e_1; e_2)$ — $e_1 \mathbin{+\!\!+} e_2$ — string concat

$\quad$ len (e) — $|e|$ — length

$\quad$ let $(e_1; x.e_2)$ — let $x <= e_1$ in e_2 — let binder

pre-term syntactic category

"recursive" calls

Q Help me with times

abstract syntax

concrete syntax

$n \in \mathbb{N}$
$s \in \Sigma^*$

for simplicity n is Nats and strings is a sequence of letters drawn from the alphabet Σ

Σ normally stands for some alphabet
e.g $\Sigma = \{a, b, c \cdots x\}$

And Σ^* is typically used to describe words of that alphabet, specifically zero or more occurances of members of the Σ set strung together
e.g $"" \in \Sigma^*$ $\quad$ $"hello" \in \Sigma^*$

We call these Abstract Syntax Trees because syntactically correct expressions can be written as a tree:

times(num [1]; len(str[hello]))

times

For your reference, this BNF corresponds to the following Haskell deep embedding of our little language, where we are using Haskell's data types to represent the syntax.

```
> data Types = Num | Str

> data Exp = V Char
>          | N Nat
>          | S String
>          | Plus Exp Exp
>          | Times Exp Exp
>          | Cat Exp Exp
>          | Len Exp
>          | Let Char Exp
```

If you've super been on your Haskell, and to enrich your PL research knowledge, here is a more advanced and typed embedding using my eDSL knowledge, specifically the DataKinds and GADT language extensions and Higher Order Abstract Syntax (HOAS)

GADT DataKinds

```
> data Exp a where
>     N :: Nat -> Exp Num
>     S :: String -> Exp Str
>     Plus  :: Exp Num -> Exp Num -> Exp Num
>     Times :: Exp Num -> Exp Num -> Exp Num
>     Cat   :: Exp Str -> Exp Str -> Exp Str
>     Len   :: Exp Str -> Exp Num
>     Let   :: Exp a -> (Exp a -> Exp b) -> Exp b
```

HOAS

We can even achieve the concrete syntax with smart constructors

```
> (+) = Plus
> (*) = Times
> (++) = Cat
```

assuming we hide these from Prelude

Len too hard for Haskell
HOAS gives us nice let syntax

Here is another BNF example. This time for a small robot language

$n \in \mathbb{N}$ $s \in \Sigma^*$ where $\Sigma = \{a, b, c \cdots x\}$

```
c ::= stop
      fwd n (c)
      bwd n (c)
      left (c)
      right (c)
      say s (c)
```

Example command:

```
fwd 3 (say [hello] stop)
```

A little language of numbers + strings — typing

Now we know how to write pre-terms, we will specify what it means to be a term: a well typed pre-term

$$\text{VAR}\ \frac{}{\Gamma, x:\sigma \vdash x:\sigma}$$

The Var rule says that a variable x is typeable if it is in the context

$$\text{Num}\ \frac{n \in \mathbb{N}}{\Gamma \vdash \text{num}[n] : \text{Num}}$$

The Num rule says that num[n] is typeable as a Num under any Γ so long as n is a $\mathbb{N}$

Q What do you think the Str rule is?

$$\text{Str}\ \frac{s \in \Sigma^*}{\Gamma \vdash \text{str}[s] : \text{Str}}$$

The Str rule says that str[s] has type Str under any Γ so long as $s \in \Sigma^*$

$$\text{PLUS}\ \frac{\Gamma \vdash e_1 : \text{Num} \quad \Gamma \vdash e_2 : \text{Num}}{\Gamma \vdash \text{plus}(e_1; e_2) : \text{Num}}$$

The Plus rule says: if Γ types e_1 and e_2 as Num, then we can add them to produce a Num (under the same context).

Q Can you specify times?

$$\text{TIMES}\ \frac{\Gamma \vdash e_1 : \text{Num} \quad \Gamma \vdash e_2 : \text{Num}}{\Gamma \vdash \text{times}(e_1; e_2) : \text{Num}}$$

The Times rule says: if Γ types e_1 and e_2 as Num, then we can times them to produce a Num (under the same context).

(Q what about Cat?)

$$\text{CAT}\ \frac{\Gamma \vdash e_1 : \text{Str} \quad \Gamma \vdash e_2 : \text{Str}}{\Gamma \vdash \text{cat}(e_1; e_2) :}$$

The Cat rule says: if Γ types e_1 and e_2 as Str, then we can append them to produce a Str (under the same context).

Q If this is the Len rule, what does it say?

$$\text{LEN}\ \frac{\Gamma \vdash e : \text{Str}}{\Gamma \vdash \text{len}(e) : \text{Num}}$$

The Len rule says if e is a Str under Γ, then len(e) has type Num under Γ.

The Let rule is the most interesting because it actually adds variables to the context:

$$\text{Let}\ \frac{\Gamma \vdash e_1 : \sigma_1 \qquad \Gamma, x:\sigma_1 \vdash e_2 : \sigma_2}{\Gamma \vdash \text{let}(e_1; x.e_2) : \sigma_2}$$

The let rule says that if the expression that gets bound (e_1) has type σ_1 under Γ, and the body of the let (e_2) has type σ_2 under the context $\Gamma, x:\sigma_1$ (a variable of the same type as the bound expression) then the whole let expression can be typed with σ_2.

Robot lang typing rules (no context cos it's a silly small lang) :)

$$\text{STOP}\ \frac{}{\vdash \text{stop} : \text{Cmd}}$$

$$\text{FWD}\ \frac{n \in \mathbb{N} \qquad \vdash c : \text{Cmd}}{\vdash \text{fwd}\ n\ c : \text{Cmd}}$$

$$\text{BWD}\ \frac{n \in \mathbb{N} \qquad \vdash c : \text{Cmd}}{\vdash \text{bwd}\ n\ c : \text{Cmd}}$$

$$\text{LEFT}\ \frac{\vdash c : \text{Cmd}}{\vdash \text{left}\ c : \text{Cmd}} \qquad \text{RIGHT}\ \frac{\vdash c : \text{Cmd}}{\vdash \text{right}\ c : \text{Cmd}}$$

$$\text{SAY}\ \frac{s \in \Sigma^*}{\vdash \text{say}\ s : \text{Cmd}}$$

Now we have the basics of our little language, lets explore what good properties we can and want to prove of it, and prove it!

Inversion

This first property is the formalisation of rules being "syntax directed", putting emphasis on the importance of only having one way of doing each thing, because this allows us to unlock the benefit of types that it guides the programme.

Take our PLUS rule for example: there is only one way of introducing the plus syntax in a well typed setting, and this rule insists that both arguments to plus are numbers, preventing any term featuring plus with non-Num arguments -- preventing the programme doing silly things.

$$\text{PLUS}\ \frac{\Gamma \vdash e_1 : \text{Num} \qquad \Gamma \vdash e_2 : \text{Num}}{\Gamma \vdash \text{plus}(e_1; e_2) : \text{Num}}$$

This set up also makes proofs easier because we can look at the syntax of the term we have been given, and from that know what the derivation should have been.

(as we observed last week when we did proof by induction on the nat) judgement.

We call these **inversion lemmata** (where lemmata is the plural of lemma) because they can be created for any set of (good) typing judgements.

The inversion lemma for our language includes one case per rule:

Inversion: Suppose $\Gamma \vdash e : \tau$

1. (PLUS case) If $e = \text{plus}(e_1; e_2)$, then it must be that
   - $\tau = \text{Num}$
   - $\Gamma \vdash e_1 : \text{Num}$
   - $\Gamma \vdash e_2 : \text{Num}$

i.e. plus should only ever be typed as Num and applied to Nums

Q Can you help me formally state inversion for Cat?

$$\text{CAT}\ \frac{\Gamma \vdash e_1 : \text{Str} \qquad \Gamma \vdash e_2 : \text{Str}}{\Gamma \vdash \text{cat}(e_1; e_2) : \text{Str}}$$

2. (CAT case) If $e = \text{cat}(e_1; e_2)$ then it must be that:
   - $\tau : \text{Str}$
   - $e_1 : \text{Str}$
   - $e_2 : \text{Str}$

(sheet will ask you about len and let case)

Inversion can be proven of a set of rules in two ways:

Proven by:
- Induction: maybe a bit overkill, but very formal and persuasive. Also good practice!
- Inspection: this is when you say look at the rules, there is no other way, obviously this holds. Maybe a bit cocky-wanky / risky for the inexperienced, but a nice shortcut for the experienced.

Weakening

Our next desireable property is weakening, which states that adding unrelated variables to the environment shouldn't affect typing.

This makes sense because it is sort of asserting that a small reasonable and unrelated change causes no unexpected effects or side effects.

Weakening holds for most programming languages.

In this course, it will be a very useful lemma for you because when completing your proofs you might find that you need to weaken an environment (a common proof mistake is ignoring the environments and not ensuring that they match).

Weakening: If $\Gamma \vdash e : \tau$ and x is fresh (unrelated) then $\Gamma, x:\sigma \vdash e : \tau$.

Intuitively, the reason it holds is if we have $\Gamma \vdash e : \tau$, then we have a derivation, which will still be valid if we "thread" this extra unrelated variable through

Intuition:

We have $x{:}\text{Num} \vdash \text{plus}(\text{num}[3]; \text{num}[2]) : \text{Num}$ and $z:\sigma$ is fresh

$$\dfrac{\text{Num} \quad \dfrac{3 \in \mathbb{N}}{x{:}\text{Num} \vdash \text{num}[3] : \text{Num}} \quad \dfrac{2 \in \mathbb{N}}{x{:}\text{Num} \vdash \text{num}[2] : \text{Num}}\ \text{NUM}}{x{:}\text{Num} \vdash \text{plus}(\text{num}[3]; \text{num}[2]) : \text{Num}}\ \text{PLUS}$$

$\Downarrow$

$$\dfrac{\text{Num} \quad \dfrac{3 \in \mathbb{N}}{x{:}\text{Num} \vdash \text{num}[3] : \text{Num}} \quad \dfrac{2 \in \mathbb{N}}{x{:}\text{Num} \vdash \text{num}[2] : \text{Num}}\ \text{NUM}}{x{:}\text{Num}, y{:}\sigma \vdash \text{plus}(\text{num}[3]; \text{num}[2]) : \text{Num}}\ \text{PLUS}$$

Adding y doesn't violate the derivation. (see notes for an extended example featuring LET and VAR)

Weakening is proved by induction.

I don't want to go through the proof in this lecture, because what we have taught you of proof by induction and rules should enable you to prove this statement, so it is in the sheet.

Again if you are struggling with proof by induction see us ASAP. Try this weakening proof in the sheets, talk to the TAs about it in the class. Ensure you get it, and if not, just let me know and I'll arrange office hours.

Substitution

Thus far we have been creating our language, but what have we been creating it in? I've showed you a couple of examples embedding it into Haskell, but in reality we are specifying it in mathematics.

When we do this, we call the defined/specified language the guest or embedded language and what it is specified in the host or meta language.

Often during this process, we specify meta-theoretic operations to help us along.

> Meta-theoretic ops = operations that we will use in the specification of our language that are not constructs of the language.

Substitution is an example of this.

This operation specifies how we instantiate a variable.

We define it's behavior by induction (in other words by pattern matching on each syntactic form of pre-terms:

$$\underbrace{z[e/x]}_{\text{sub } e \text{ in for } x} = \begin{cases} e & \text{if } z \equiv x \\ z & \text{if } z \not\equiv x \end{cases}$$

When performing substitution on variables, if the variable in question is the one we are substituting, we perform the substitution, swapping the variable x for its substitute e, if not the variable remains.

$$(\text{num}[n])[e/x] = \text{num}[n]$$

$$(\text{str}[s])[e/x] = \text{str}[s]$$

Substitution has no effect on constants (numbers and strings) because they don't mention/involve variables.

The "recursive" cases (plus, times, cat and len) are all very similar and boring because they just push the substitution down to their sub-expressions. It is only at the leaves (var namely) and with binders (we will do next) that substitution is interesting.

For example substitution in the plus case is just

$$\text{plus}(e_1; e_2)[e/x] = \text{plus}(e_1[e/x]; e_2[e/x])$$

Times, cat and len are <u>analygous</u>

Analygous is a word you will see often in proofs because proof writers use it when they consider the proof of another case so similar to the current case that it is not worth writing. Succinct for experts, less helpful for newbies.

All in all the definition of substitution is similar to writing a function in Haskell that pattern matches on each construct.

There is one case we have left out: let. This is because it is a binder, which complicates things. We'll get to that in a second, first I want to run through performing a substitution with you to ensure we get the idea

$$(\text{plus}(\text{num}[1]; x))[\text{num}[2]/x]$$
$$\rightarrow \text{plus}(\text{num}[1][\text{num}[2]/x]; x[\text{num}[2]/x])$$
$$\rightarrow \text{plus}(\text{num}[1]; x[\text{num}[2]/x])$$
$$\rightarrow \text{plus}(\text{num}[1]; \text{num}[2])$$

> Q What would the result have been if we were substituting for y instead?

$$(\text{plus}(\text{num}[1]; x))[\text{num}[2]/y]$$
$$\twoheadrightarrow \text{plus}(\text{num}[1]; x)$$

But what about let?
Well substituting under a binder is treacherous!

Let me show you why...

Here is the definition of substitution for let:

$$\text{let}(e_1; y.e_2)[e/x] = \text{let}(e_1[e/x]; y.e_2[e/x])$$

Looks like just another propagation of the substitution, but if we are not careful, all could go wrong.

Consider the case where $x = y$.

$$\text{let}(y; y.\text{len}(y))[\text{str}[\text{hi}]/y]$$

this y is NOT this y
→ referential clash

Only the first y should be substituted, but if we proceed blindly, both will

Consider the case where e contains y

$$\text{let}(x; y.x + y)[y/x]$$

this y is free
but if substituted in here it won't be
→ variable capture

Luckily, both of these issues can be fixed by alpha-renaming. In both cases, if we rename the y of the let to z our issues will go away.

So to avoid all of this we will adopt the

Barendregt Convention

= We assume that everything has been alpha renamed already so this doesn't happen

Easy as that. Just be aware and don't introduce such issues.

Now substitution is a very useful meta-op that we will be using next week when we define our dynamics, so we better ensure it plays nicely with our statics.

We capture "playing nicely" into the substitution lemma, one of the most important lemmas of this unit.

Substitution: If $\Gamma \vdash e : \tau$ and $\Gamma, x:\tau \vdash u : \sigma$
then $\Gamma \vdash u[e/x] : \sigma$

In other words: if I sub in an expression of the correct type, the overall type will be unchanged.

This is proven by <u>induction</u> (I told you we'd be doing this a lot!)

I'll start the proof for you now, doing the most difficult case, and you'll do the rest in the sheet.

Proof by induction on $\Gamma, x:\tau \vdash u : \sigma$.

(we have chosen, the second more interesting premise to induct on)

[Case: let]

Explicitly instantiate Γ as $\Gamma, x:\tau$; u as let... and σ as σ:

GOAL = if $\Gamma \vdash e : \tau$
and
$\Gamma, x:\tau \vdash \text{let}(e_1; y.e_2) : \sigma$
then
$\Gamma \vdash \text{let}(e_1; y.e_2)[e/x] : \sigma$

Assume premises and update goal:

P1 = $\Gamma \vdash e : \tau$
P2 = $\Gamma, x:\tau \vdash \text{let}(e_1; y.e_2) : \sigma$

GOAL' = $\Gamma \vdash \text{let}(e_1; y.e_2)[e/x] : \sigma$

Now normally at this point I state the IHs, but this is really easy to get wrong in these more complex settings, because the IH is exactly the proof statement for exactly the smaller thing. So first I'm going to explore the derivation of P2 so I get the "smaller thing" exactly.

By inversion!
P2 must have had the following derivation, with sub derivations I've named for use later:

$$\dfrac{\begin{matrix}\text{SHAPE1} \\ \vdots \\ \Gamma, x:\tau \vdash e_1 : \sigma_1\end{matrix} \qquad \begin{matrix}\text{SHAPE2} \\ \vdots \\ \Gamma, x:\tau, y:\sigma_1 \vdash e_2 : \sigma\end{matrix}}{\Gamma, x:\tau \vdash \text{let}(e_1; y.e_2) : \sigma}\ \text{LET}$$

The "smaller thing" is what is on top of the rule, our recursive premises. Because we have two, we have two IHs:

IH1 = if $\Gamma \vdash e : \tau$ and $\Gamma, x:\tau \vdash e_1 : \sigma_1$
then $\Gamma \vdash e_1[e/x] : \sigma_1$

IH2 = if $\Gamma, y:\sigma_1 \vdash e : \tau$ and $\Gamma, y:\sigma_1, x:\tau \vdash e_1 : \sigma_2$
then $\Gamma, y:\sigma_1 \vdash e_2[e/x] : \sigma_2$

Notice how Γ for IH2 includes y!

To use the conclusions of the IHs, we must "unlock" them by providing their premises.

Applying IH1 to P1 and SHAPE1 allows us to conclude

UIH1 = $\Gamma \vdash e_1[e/x] : \sigma_1$

It is a little more complex for IH2 because P1 isn't quite what we need.

Luckily, we just learnt about weakening

By weakening P1 we can conclude

WP1 = $\Gamma, y:\sigma_1 \vdash e : \tau$

Applying IH2 to this and SHAPE2 gives us

UIH2 = $\Gamma, y:\sigma_1 \vdash e_2[e/x] : \sigma_2$

Combining UIH1 and UIH2 together using the let rule proves our goal:

GOAL' = $\Gamma \vdash \text{let}(e_1; y.e_2)[e/x] : \sigma$
= by def subst
$\Gamma \vdash \text{let}(e_1[e/x]; y.e_2[e/x])$

$$\dfrac{\dfrac{\vdots\ \text{UIH1}}{\Gamma \vdash e_1[e/x] : \sigma_1} \qquad \dfrac{\vdots\ \text{UIH2}}{\Gamma, y:\sigma_1 \vdash e_2[e/x] : \sigma}}{\Gamma \vdash \text{let}(e_1[e/x]; y.e_2[e/x]) : \sigma}\ \text{LET}$$

$\square_{\text{LET}}$