Type Inference & Polymorphism

We have spoken a lot about types in this course, and even built a type checker for the toy calculator.

Types become even more important with abstraction & application.

We saw how to assign types for those expressions too.

$$\frac{\Gamma[x : \tau_1] \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}\ \text{AbsT} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (e_1\ e_2) : \tau_2}\ \text{AppT}$$

What are τ_1 and τ_2 in these types?

Obviously we are using them as placeholders for arbitrary types.

How arbitrary is arbitrary?

Could be pretty much anything!

Can write any computable function in the lambda calculus.

The type checker needs to assign types to well-defined functions.

All functions are equal, but some functions are more equal than others!

Consider the identity function $id(x) = x$.

What is the type of id? $IntT \to IntT$? $BoolT \to BoolT$?

$(IntT \to BoolT) \to (IntT \to BoolT)$?

Any of these is a reasonable id function for that type

Polymorphic Types:

We would like a **single** id function that works for all inputs

The type of this function is **parametrized** by that of its input.

id: $T \to T$ (for **any** T, including user-defined T)

T is a generic variable.

If a type T can be obtained by substituting certain types for the generic variables in a type τ, we say that

T is a substitution instance of τ.

$IntT \to IntT$, $BoolT \to BoolT$,

$(IntT \to BoolT) \to (IntT \to BoolT)$

Are all substitution instances of $\tau \to \tau$.

One can conceive of similar functions on lists, which are agnostic of the type of the elements of the input list.

length, map, fold_right, ... How do we infer types for these?

* length [] = 0 ——— (L1)

length (x:xs) = 1 + (length xs) ——— (L2)

From (L1), the type of length is [T] → IntT for any T.

(L2) does not constrain this type in any way
(It does not further restrict the set of values T can take)

So length : [T] → IntT.

* `map f [] = []` ——— (M1)

`map f (x:xs) = f(x) : (map f xs)` ——— (M2)

`map (fun x → x+5) [1;2;3]`

$= (1+5) : (\text{map } (\text{fun } x \to x+5)\ [2;3])$

$= 6 : ((2+5) : (\text{map } (\text{fun } x \to x+5)\ [3]))$

$= 6 : (7 : ((3+5) : (\text{map } (\text{fun } x \to x+5)\ [\,])))$

$= 6 : (7 : (8 : [\,])) = [6;7;8]$

From (M1), the type of `map` is $T_1 \to [T_2] \to [T_3]$, where T_1 is the type of f.

Unlike `length`, (M2) does restrict this type!

```
map f []     = []                    ---- M1
map f (x:xs) = f(x) : (map f xs)     ---- M2
```

f takes one input, which must be of the same type as elements of the list that map is operating on.

From $T_1 \to [T_2] \to [T_3]$, we have $T_1 = T_2 \to T_4$, and so we get $(T_2 \to T_4) \to [T_2] \to [T_3]$ (for some T_4)

Also, the output of f must be of the same type as elements of the list obtained by applying map with f to a list l. So, we have $T_4 = T_3$, and so, the type of map is as follows:

$$map : (T_2 \to T_3) \to [T_2] \to [T_3].$$

* fold_right f v [] = v ——— (F1)

fold_right f v (x : xs) = f x (fold_right f v xs) ——— (F2)

fold_right (+) 5 [1;2;3]

= (+) 1 (fold_right (+) 5 [2;3])

= (+) 1 ((+) 2 (fold_right (+) 5 [3]))

= (+) 1 ((+) 2 ((+) 3 (fold_right (+) 5 [])))

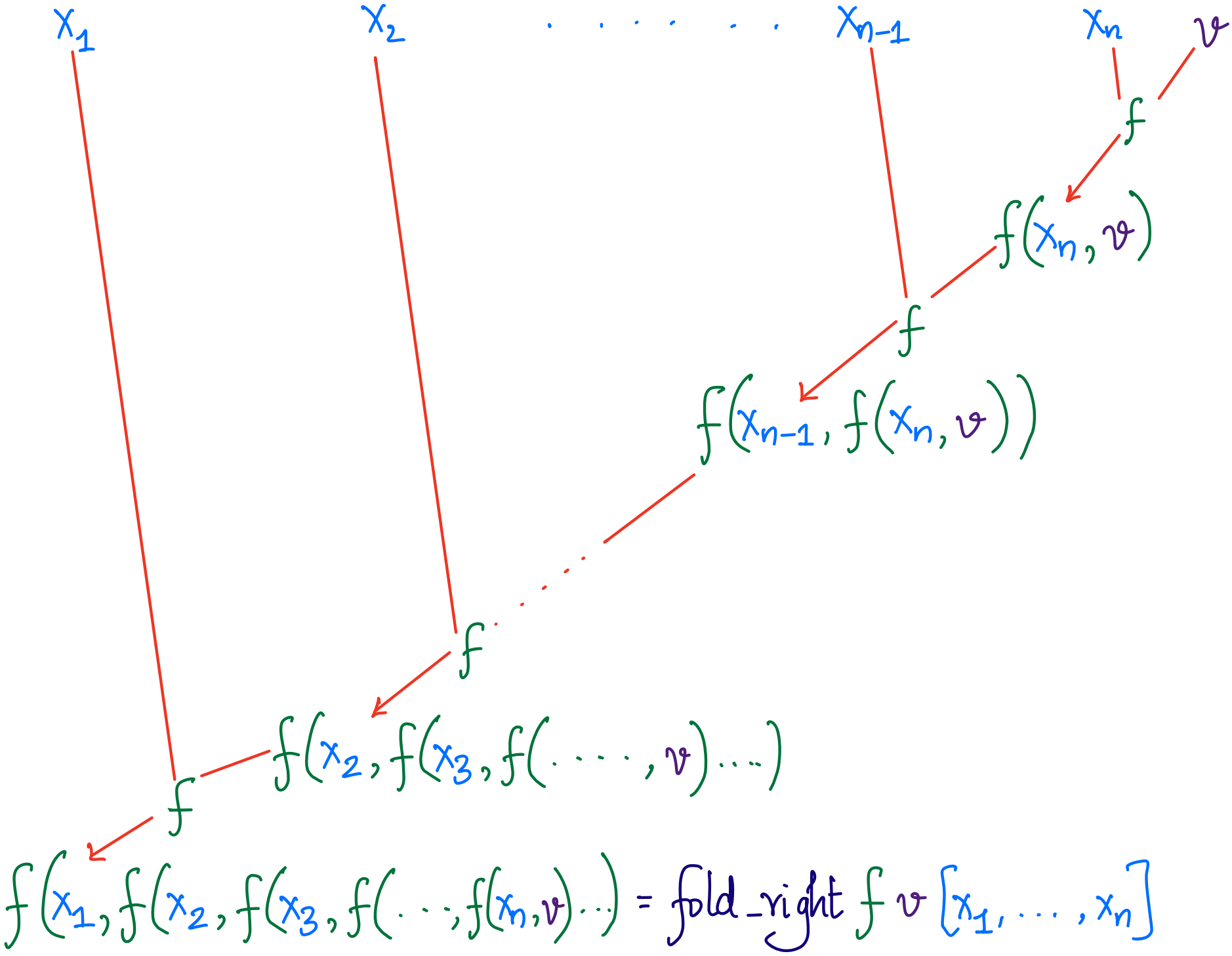
= (+) 1 ((+) 2 ((+) 3 5))

= (+) 1 ((+) 2 8)

= (+) 1 (10)

= 11

Essentially, 1 + (2 + (3 + 5))


x_1
x_2
x_{n-1}
x_n
v
f
$f(x_n, v)$
f
$f(x_{n-1}, f(x_n, v))$
f
$f(x_2, f(x_3, f(\ldots, v)\ldots))$
f
$f(x_1, f(x_2, f(x_3, f(\ldots, f(x_n, v)\ldots)) = \text{fold\_right}\ f\ v\ [x_1, \ldots, x_n]$

fold_right (+) 0 l ≡ sum l

fold_right (@) [] l ≡ flatten l . . . .

fold_right f v [] = v ——— (F1)

fold_right f v (x : xs) = f x (fold_right f v xs) ——— (F2)

From (F1), we get a type for fold_right as $T_0 \to T_2 \to [T_1] \to T_2$ where T_0 is the type of f and T_2 that of v.

(F2) restricts this further by telling us that f takes two inputs, the first of which **must be of the same type** as elements of the list we are folding. It also tells us that the output of f **must be of the same type** as v. So, $f : T_1 \to T_2 \to T_2$, and

$$\text{fold\_right} : (T_1 \to T_2 \to T_2) \to T_2 \to [T_1] \to T_2 .$$

How do we construct a type checker that works with polymorphism?

By some manner of inference over typing rules, as usual!

Consider the S combinator. Recall that $SPQR = (PR)(QR)$.

Can write the equivalent lambda expression for the S combinator as $\lambda x.\lambda y.\lambda z.((xz)(yz))$.

We can write out the type inference tree as follows:

$$\dfrac{\dfrac{\dfrac{\Gamma' \vdash x : \tau_0 \quad \Gamma' \vdash z : \tau_1}{\Gamma' \vdash (xz) : \tau_4}\,\text{AppT} \qquad \dfrac{\Gamma' \vdash y : \tau_2 \quad \Gamma' \vdash z : \tau_3}{\Gamma' \vdash (yz) : \tau_5}\,\text{AppT}}{\Gamma' \vdash ((xz)(yz)) : \tau_6}\,\text{AppT}}{\Gamma \vdash \lambda x.\lambda y.\lambda z.((xz)(yz)) : \tau_7}\,\text{AbsT}^3$$

What equations do we get based on what we know about AppT?

$$\tau_0 = \tau_1 \to \tau_4 \qquad \tau_2 = \tau_3 \to \tau_5 \qquad \tau_4 = \tau_5 \to \tau_6$$

$$= \tau_1 \to \tau_5 \to \tau_6$$

$$\dfrac{\dfrac{\Gamma' \vdash x : \tau_1 \to \tau_5 \to \tau_6 \quad \Gamma' \vdash z : \tau_1}{\Gamma' \vdash (xz) : \tau_5 \to \tau_6}\ \text{AppT} \qquad \dfrac{\Gamma' \vdash y : \tau_3 \to \tau_5 \quad \Gamma' \vdash z : \tau_3}{\Gamma' \vdash (yz) : \tau_5}\ \text{AppT}}{\Gamma' \vdash ((xz)(yz)) : \tau_6}\ \text{AppT}$$

z gets two types τ_1 and τ_3. Are these types equal?

Recall that we are doing this derivation to end in AbsT!

So $\Gamma' = \Gamma[\underline{x}:T_0, y:T_2, \underline{z}:T_1]$

But $\Gamma' \vdash \underline{z}:T_3$, so $T_3 = T_1$

Variables in the scope of an abstraction get assigned the same type at all their occurrences.

So $\lambda x.\lambda y.\lambda z.(xz)(yz)$ has the following type (plugging in $T_3 = T_1$)

$$(T_1 \to T_5 \to T_6) \to (T_1 \to T_5) \to T_1 \to T_6$$

What about let statements?

let statements can be translated to lambda expressions.

For example, let $x = M$ in N can be translated to $(\lambda x.N)M$.

But, as above, all occurrences of x in $\lambda x.N$ must get the same type.

When we type $\lambda x.N$, we do so agnostic of the expression (if any) that it might be applied to. But a let expression has all that information available ab initio, which we can use to make typing more flexible! In let $x = M$ in N, to type N, it suffices to assume that x will only take on the value M. This allows us to type different occurrences of x with different substitution instances of the type of M.

Consider the expression

$$\text{let } e = \lambda x.\, x \text{ in } (e\ e).$$

Do both occurrences of e in $(e\ e)$ get the same type? Obviously not

$$\frac{\Gamma \vdash e : T_1 \to T_2 \qquad \Gamma \vdash e : T_1}{\Gamma \vdash (e\ e) : T_2}\ \text{AppT}$$

Unlike abstraction!

Note that there is no substitution which equates $T_1 \to T_2$ with T_1.
So we **cannot** assign the same type to the two occurrences of e.

However, since e is the identity function, we know that the types for the inputs and outputs of e are the **same**.

Thus, we need to assign to the two occurrences of e two different substitution instances of the $T \to T$ type!

Consider marking the es in the expression as $(e^l\ e^r)$.

$e^l : T_l \to T_l$ $\qquad$ $e^r : T_r \to T_r$

From AppT, we know that

$T_1 \to T_2 = T_l \to T_l$

$T_1 = T_r \to T_r$

using unification!

We can solve this system of equations to get $T_l = T_r \to T_r$.

Therefore,

$e^l : (T_r \to T_r) \to (T_r \to T_r)$, and

$e^r : T_r \to T_r$.

Type Inference for Recursive Functions :

Think about a recursive function as having the following general "shape": $f = (\ldots f \ldots)$

Since we will only be using this in the context of an expression, we can adopt a modified let expression:

letrec $f = M$ in N where M looks like $(\ldots f \ldots)$

How do we type this expression?

Let us start exactly as we did with let expressions.

① Assume f has some type τ_f.

② Determine a type τ_M for M using τ_f to be the type of f.

- Every occurrence of f in M needs to get type τ_f.

** Need to ensure that τ_M unifies with τ_f! **

Once this is done, however, each occurrence of f in N can be typed using a substitution instance of the type τ_M.

Throwback Thursday: Back to Combinatory Logic

We saw the I, S, and K combinators.

There is another combinator which codes up recursion: Y.

The reduction rule for Y is as follows:

$$YE \rightarrow E(YE)$$

Allows one to apply an expression E to itself multiple times.

Can write letrec expressions as let expressions + Y combinator.

If we can come up with a type for Y, then we know how to type letrec expressions using

the type for Y and

the type for let expressions.

$$* \; map\ f\ l = \text{if } (l == []) \text{ then } [] \text{ else } (f(hd\ l)) : (map\ f\ (tl\ l)).$$

$$letrec\ map = \lambda f.\lambda l.\ \text{if } (l == []) \text{ then } [] \text{ else } (f(hd\ l)) : (map\ f\ (tl\ l)).$$

$$\downarrow$$

$$let\ map = Y(\lambda m.\lambda f.\lambda l.\ \text{if } (l == []) \text{ then } [] \text{ else } (f(hd\ l)) : (m\ f\ (tl\ l))).$$

$$map \rightarrow (\lambda m.\lambda f.\lambda l.\ \text{if } (l == []) \text{ then } [] \text{ else } (f(hd\ l)) : (m\ f\ (tl\ l)))\ map$$

$$= \lambda f.\lambda l.\ \text{if } (l == []) \text{ then } [] \text{ else } (f(hd\ l)) : (map\ f\ (tl\ l)).$$

So now we only need to assign a type for Y.

letrec $Y = \lambda f.(f\ (Yf))$

$$\dfrac{\dfrac{}{\Gamma' \vdash f : \tau_2} \qquad \dfrac{\Gamma' \vdash Y : \tau_0 \qquad \Gamma' \vdash f : \tau_1}{\Gamma' \vdash Yf : \tau_3}\ \text{AppT}}{\dfrac{\Gamma' \vdash f\ (Yf) : \tau_4}{\Gamma \vdash \lambda f.(f\ (Yf)) : \tau_5}\ \text{AbsT}}\ \text{AppT}$$

$\tau_1 = \tau_2 \qquad \tau_0 = \tau_1 \to \tau_3 \qquad \tau_2 = \tau_3 \to \tau_4 \qquad \tau_5 = \tau_1 \to \tau_4$

$\tau_1 = \tau_3 \to \tau_4$

$\tau_0 = (\tau_3 \to \tau_4) \to \tau_3 \qquad \tau_5 = (\tau_3 \to \tau_4) \to \tau_4$

Y and $\lambda f.(f\ (Yf))$ must be assigned the same type,
since they are the **same expression**, occurring in the same context!

So, $\tau_0 = \tau_5 = (\tau_3 \to \tau_3) \to \tau_3$.

$$Y : (\tau \to \tau) \to \tau .$$

We can show that we can write any recursive function

$$\text{letrec } f = \lambda \bar{x} . (\ldots f \ldots \ldots)$$

as

$$\text{let } f = Y\,(\lambda g . \lambda \bar{x} . (\ldots g \ldots \ldots))$$

We can now assign a type to any recursive function by combining the type for let and Y appropriately!