Polymorphism

Concepts of Programming Languages
Lecture 20

CAS CS 320

Outline

- Discuss **polymorphism** in general
- Discuss **System F**, a type system with parametric polymorphism
- Demo an implementation of **System F**

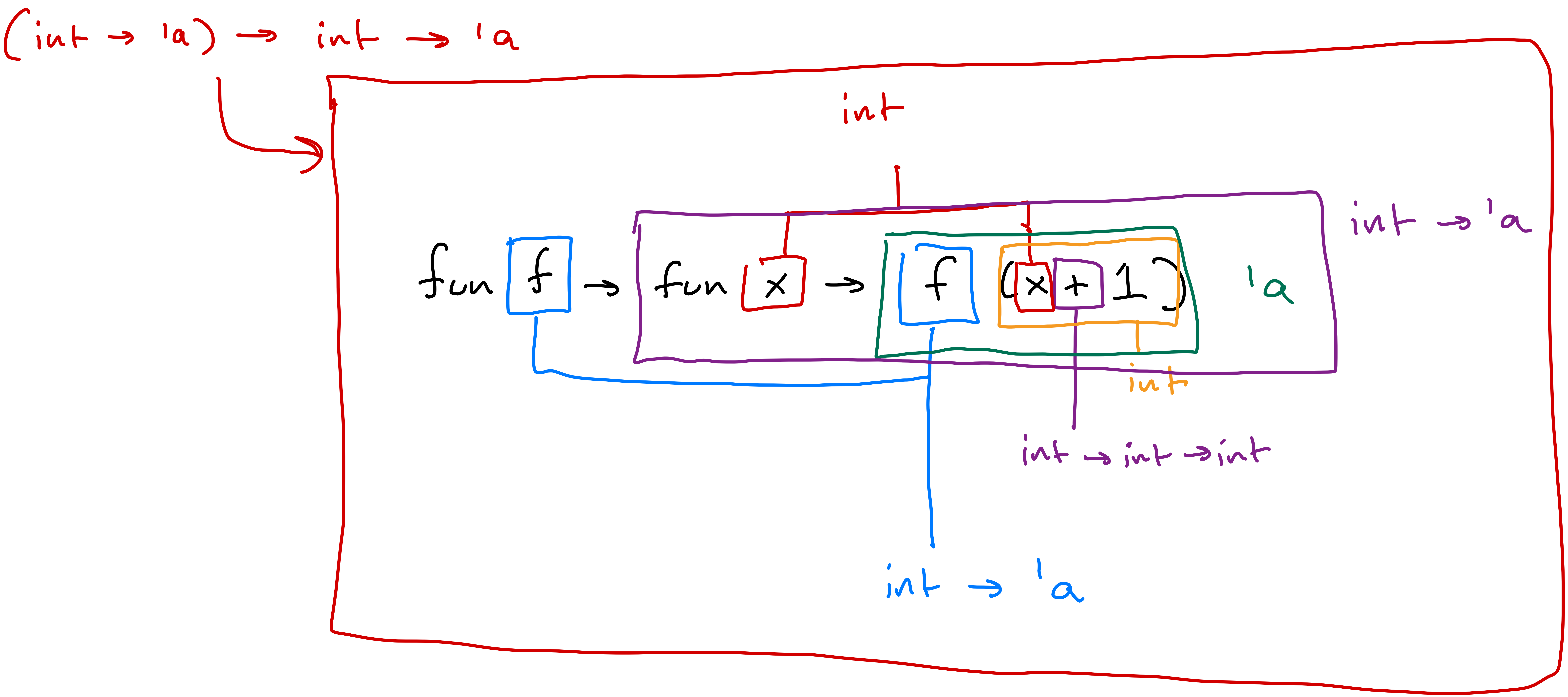
Practice Problem

```
fun f -> fun x -> f (x + 1)

let rec f x = f (f (x + 1)) in f
```

What are the types of the above OCaml expressions?

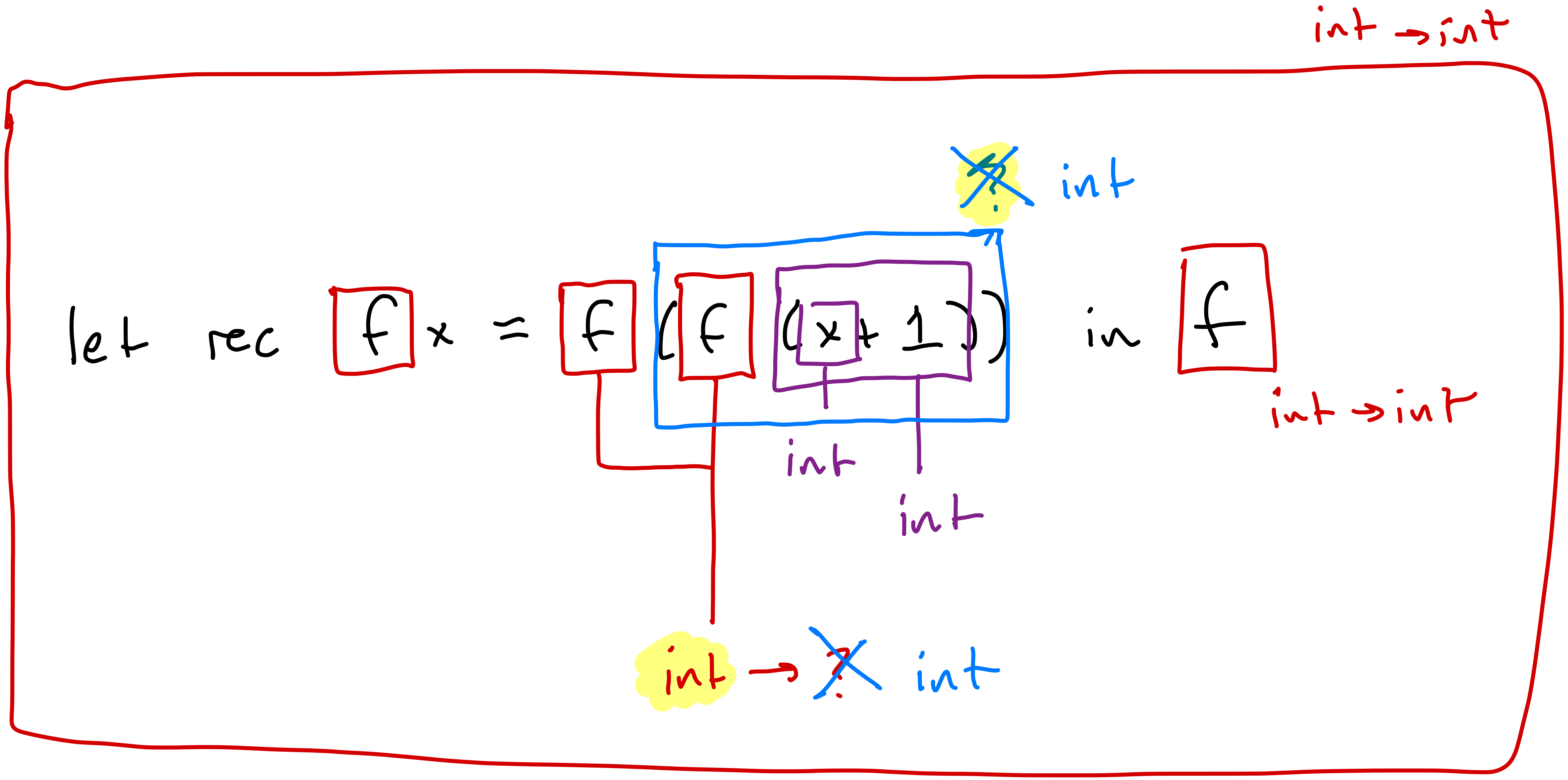
Answer

```
fun f -> fun x -> f (x + 1)

let rec f x = f (f (x + 1)) in f
```



Answer

```
fun f -> fun x -> f (x + 1)

let rec f x = f (f (x + 1)) in f
```



Polymorphism

High Level

```
let rec rev_int (l : int list) : int list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let rec rev_string (l : string list) : string list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let _ = assert (rev_int [1;2;3] = [3;2;1])
let _ = assert (rev_string ["1";"2";"3"] = ["3";"2";"1"])
```

High Level

```
let rec rev_int (l : int list) : int list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let rec rev_string (l : string list) : string list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let _ = assert (rev_int [1;2;3] = [3;2;1])
let _ = assert (rev_string ["1";"2";"3"] = ["3";"2";"1"])
```

Copy/pasting code is *time consuming* and *error prone*

High Level

```
                               'a              'a
let rec rev_int (l : int list) : int list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let rec rev_string (l : string list) : string list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let _ = assert (rev_int [1;2;3] = [3;2;1])
let _ = assert (rev_string ["1";"2";"3"] = ["3";"2";"1"])
```

Copy/pasting code is *time consuming* and *error prone*

Polymorphism allows for better code reuse. The *same* function can be applied in multiple contexts

Basic Example

```
let id = fun x -> x
let a = id 0
let b = id (0 = 0)
let c = id id
```

Basic Example

```
let id = fun x -> x
let a = id 0
let b = id (0 = 0)
let c = id id
```

We want to be able to define functions that can be used in multiple contexts *and* that we can type check

Basic Example

```
let id = fun x -> x
let a = id 0
let b = id (0 = 0)
let c = id id
```

We want to be able to define functions that can be used in multiple contexts *and* that we can type check

Important: We can evaluate this if we *don't* type check

Basic Example

```
let id = fun x -> x
let a = id 0
let b = id (0 = 0)
let c = id id
```

We want to be able to define functions that can be used in multiple contexts *and* that we can type check

<u>Important:</u> We can evaluate this if we *don't* type check

*But if we type-check, what should be the type of **id**?*

Polymorphism

Polymorphism

There are two common kinds of polymorphism

Polymorphism

There are two common kinds of polymorphism

1. **Ad Hoc Polymorphism:** The ability to overload function names so that different types can share interfaces

Polymorphism

There are two common kinds of polymorphism

1. **Ad Hoc Polymorphism:** The ability to overload function names so that different types can share interfaces

2. **Parametric polymorphism:** The ability to define functions that are *agnostic* to (parts of) the types, giving it more reusability

Polymorphism

There are two common kinds of polymorphism

1. **Ad Hoc Polymorphism:** The ability to overload function names so that different types can share interfaces

our focus

2. **Parametric polymorphism:** The ability to define functions that are *agnostic* to (parts of) the types, giving it more reusability

Aside: Ad Hoc Polymorphism

```
let add (x : float) (y : float) = x +. y
let add (x : string) (y : string) = x ^ y
(* This doesn't work in OCaml... *)
```

Aside: Ad Hoc Polymorphism

```
let add (x : float) (y : float) = x +. y
let add (x : string) (y : string) = x ^ y
(* This doesn't work in OCaml... *)
```

Ad hoc polymorphism is essentially **function overloading**

Aside: Ad Hoc Polymorphism

```
let add (x : float) (y : float) = x +. y
let add (x : string) (y : string) = x ^ y
(* This doesn't work in OCaml... *)
```

Ad hoc polymorphism is essentially **function overloading**

Functions can be defined and used for different types of inputs

Aside: Ad Hoc Polymorphism

```
let add (x : float) (y : float) = x +. y
let add (x : string) (y : string) = x ^ y
(* This doesn't work in OCaml... *)
```

Ad hoc polymorphism is essentially **function overloading**

Functions can be defined and used for different types of inputs

Then we can define code against *interfaces* (this is common in object oriented programming)

Parametric Polymorphism

```
let id = fun x -> x
let a = id 0
let b = id (0 = 0)
let c = id id
```

Parametric Polymorphism

```
let id = fun x -> x
let a = id 0
let b = id (0 = 0)
let c = id id
```

Parametric polymorphism allows for functions which are agnostic to the types of its inputs (this is what OCaml does)

Parametric Polymorphism

```
let id = fun x -> x
let a = id 0
let b = id (0 = 0)
let c = id id
```

Parametric polymorphism allows for functions which are agnostic to the types of its inputs (this is what OCaml does)

For example, we can write a single identity function and use it in multiple contexts

There are many subtleties to this...

Subtlety 1: Type Annotations

```
let rec rev ('a list) : 'a list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let id : 'a -> 'a = fun x -> x
```

Subtlety 1: Type Annotations

```
let rec rev ('a list) : 'a list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let id : 'a -> 'a = fun x -> x
```

Parametric polymorphism is *not* just removing type annotations

Subtlety 1: Type Annotations

```
let rec rev ('a list) : 'a list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let id : 'a -> 'a = fun x -> x
```

Parametric polymorphism is *not* just removing type annotations

There are type systems *without* polymorphism *or* type annotations

Subtlety 1: Type Annotations

```
let rec rev ('a list) : 'a list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let id : 'a -> 'a = fun x -> x
```

Parametric polymorphism is *not* just removing type annotations

There are type systems *without* polymorphism *or* type annotations

There are type systems *with* polymorphism that *require* type annotations

Subtlety 2: Type Inference

```
let rec rev ('a list) : 'a list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let id : 'a -> 'a = fun x -> x
```

Subtlety 2: Type Inference

```
let rec rev ('a list) : 'a list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let id : 'a -> 'a = fun x -> x
```

Polymorphism is *not* the same has having type inference

Subtlety 2: Type Inference

```
let rec rev ('a list) : 'a list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let id : 'a -> 'a = fun x -> x
```

Polymorphism is *not* the same has having type inference

In OCaml, polymorphism is deeply connected with it's type inference system, but they are distinct (we can choose to annotated all our OCaml code)

Subtlety 2: Type Inference

```
let rec rev ('a list) : 'a list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let id : 'a -> 'a = fun x -> x
```

Polymorphism is *not* the same has having type inference

In OCaml, polymorphism is deeply connected with it's type inference system, but they are distinct (we can choose to annotated all our OCaml code)

We will take up this topic next week

Subtlety 3: Dispatch

```
let to_string (x : 'a) : string = ...
(* This is not possible in OCaml *)
```

Parametric polymorphism cannot be used for *dispatch*

We can't write a polymorphic function that "checks the type" to see what to do

The point: Implementing polymorphism means fundamentally changing the type system

System F

Implementing Polymorphism

Implementing Polymorphism

There are a couple approaches to implementing parametric polymorphism:

Implementing Polymorphism

There are a couple approaches to implementing parametric polymorphism:

» **OCaml (Hindley–Milner):** Infer the "most general" polymorphic type

Implementing Polymorphism

There are a couple approaches to implementing parametric polymorphism:

- **OCaml (Hindley-Milner):** Infer the "most general" polymorphic type
- **System F (2nd-Order λ-Calculus):** take types as arguments!

Implementing Polymorphism

There are a couple approaches to implementing parametric polymorphism:

- **OCaml (Hindley-Milner):** Infer the "most general" polymorphic type
- **System F (2nd-Order λ-Calculus):** take types as arguments!

Either way, we have to introduce the notion of a *type variable*

Type Variables

```
let id : 'a -> 'a = fun x -> x
```

Type Variables

```
let id : 'a -> 'a = fun x -> x
```

The "parametric" part is the fact that types have *variables*

Type Variables

```
let id : 'a -> 'a = fun x -> x
```

The "parametric" part is the fact that types have *variables*

Type variables are instantiated at particular types according to the context

Type Variables

```
let id : 'a -> 'a = fun x -> x
```

The "parametric" part is the fact that types have *variables*

Type variables are instantiated at particular types according to the context

They are very similar to expression variables, e.g., we need to define *type-level capture avoiding substitution*

Quantification

```
let id : 'a . 'a -> 'a = fun x -> x
```

Quantification

```
let id : 'a . 'a -> 'a = fun x -> x
```

In reality, types variables in OCaml are **quantified**

Quantification

```
let id : 'a . 'a -> 'a = fun x -> x
```

In reality, types variables in OCaml are **quantified**

Just like with expression variables, we don't like *unbound* type variables

Quantification

```
let id : 'a . 'a -> 'a = fun x -> x
```

In reality, types variables in OCaml are **quantified**

Just like with expression variables, we don't like *unbound* type variables

We read this "**id** has type **t -> t** for any type **t**"

System F

```
let id_int : int -> int = fun (x : int) -> x
let id : 'a . 'a -> 'a = fun 'a -> fun (x : 'a) -> x

let test1 = id_int 2
let test2 = id int 2
let test3 = id string "two"
```

System F

```
let id_int : int -> int = fun (x : int) -> x
let id : 'a . 'a -> 'a = fun 'a -> fun (x : 'a) -> x

let test1 = id_int 2
let test2 = id int 2
let test3 = id string "two"
```

System F (**second-order lambda calculus**) was introduced by Jean-Yves Girard and John C. Reynolds in the 1970s

System F

```
let id_int : int -> int = fun (x : int) -> x
let id : 'a . 'a -> 'a = fun 'a -> fun (x : 'a) -> x

let test1 = id_int 2
let test2 = id int 2
let test3 = id string "two"
```

System F (**second-order lambda calculus**) was introduced by Jean-Yves Girard and John C. Reynolds in the 1970s

As usual the motivations for introducing this systems were quite different from our ideas about polymorphism now

System F

$\forall \alpha (\alpha \to \alpha)$

```
let id_int : int -> int = fun (x : int) -> x
let id : 'a . 'a -> 'a = fun 'a -> fun (x : 'a) -> x

let test1 = id_int 2
let test2 = id int 2
let test3 = id string "two"
```

type quant.

type abs.

$\Lambda \alpha. \lambda f^{int \to \alpha}. \lambda x^{int}. f(x+1) :$

$\forall \alpha. (int \to \alpha) \to int \to \alpha$

System F (**second-order lambda calculus**) was introduced by Jean-Yves Girard and John C. Reynolds in the 1970s

As usual the motivations for introducing this systems were quite different from our ideas about polymorphism now

The basic idea: Introduce types into the language itself so we can *pass them as arguments to functions*

System F (Caution)

```
let id_int : int -> int = fun (x : int) -> x
let id : 'a . 'a -> 'a = fun 'a -> fun (x : 'a) -> x

let test1 = id_int 2
let test2 = id int 2
let test3 = id string "two"
```

System F (Caution)

```
let id_int : int -> int = fun (x : int) -> x
let id : 'a . 'a -> 'a = fun 'a -> fun (x : 'a) -> x

let test1 = id_int 2
let test2 = id int 2
let test3 = id string "two"
```

Not valid OCaml

System F (Caution)

```
let id_int : int -> int = fun (x : int) -> x
let id : 'a . 'a -> 'a = fun 'a -> fun (x : 'a) -> x

let test1 = id_int 2
let test2 = id int 2
let test3 = id string "two"
```

Not valid OCaml

This is *not* what OCaml does

System F (Caution)

```
let id_int : int -> int = fun (x : int) -> x
let id : 'a . 'a -> 'a = fun 'a -> fun (x : 'a) -> x

let test1 = id_int 2
let test2 = id int 2
let test3 = id string "two"
```

Not valid OCaml

This is *not* what OCaml does

This is *not* what we'll be implementing in mini-project 3

System F (Caution)

```
let id_int : int -> int = fun (x : int) -> x
let id : 'a . 'a -> 'a = fun 'a -> fun (x : 'a) -> x

let test1 = id_int 2
let test2 = id int 2
let test3 = id string "two"
```

Not valid OCaml

This is *not* what OCaml does

This is *not* what we'll be implementing in mini-project 3

There are very few languages that implement this kind of polymorphism

System F (Syntax)

bound term var.

type abstraction

fun α → e

STLC

$$
\begin{aligned}
e &::= \boxed{\bullet \mid x \mid \lambda x^{\tau}.\, e \mid ee} \mid \boxed{\Lambda \alpha .\, e} \mid \boxed{e\tau} \\
\tau &::= \top \mid \tau \to \tau \mid \boxed{\alpha} \mid \boxed{\forall \alpha .\, \tau} \\
x &::= \textit{variables} \\
\alpha &::= \textit{type variables}
\end{aligned}
$$

typ. app.

quantification

type variables

bound type variable

The syntax for SOLC is the same as the that of STLC but with:

» constructs for abstracting over and applying to *types*

» constructs for quantifying (or generalizing) over type variables

System F (Typing)

System F (Typing)

STLC

$$\frac{}{\Gamma \vdash \bullet : \top} \qquad \frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x^{\tau} . e : \tau \to \tau'} \qquad \frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'}$$

We add <u>two new rules</u> to STLC to deal with our new constructs for polymorphism:

System F (Typing)

STLC

$$\frac{}{\Gamma \vdash \bullet : \top} \qquad \frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x^{\tau} . e : \tau \to \tau'} \qquad \frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'}$$

$$\frac{\Gamma \vdash e : \tau \qquad \alpha \text{ not free in } \Gamma}{\Gamma \vdash \Lambda \alpha . e : \forall \alpha . \tau}$$

We add <u>two new rules</u> to STLC to deal with our new constructs for polymorphism:

1. We can generalize over a type variable if our context doesn't depend on it

System F (Typing)

STLC

$$\frac{}{\Gamma \vdash \bullet : \top} \qquad \frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x^{\tau} . e : \tau \to \tau'} \qquad \frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'}$$

$$\frac{\Gamma \vdash e : \tau \qquad \alpha \text{ not free in } \Gamma}{\Gamma \vdash \Lambda \alpha . e : \forall \alpha . \tau} \qquad \frac{\Gamma \vdash e : \forall \alpha . \tau \qquad \tau' \text{ is a type}}{\Gamma \vdash e\tau' : \boxed{[\tau'/\alpha]\tau}}$$

instantiate types

We add <u>two new rules</u> to STLC to deal with our new constructs for polymorphism:

1. We can generalize over a type variable if our context doesn't depend on it

2. We can apply an expression e to a type τ, but we have to *substitute* the type into the type of e

Type Substitution

$$[\tau/\alpha]\,\top \;=\; \top$$

$$[\tau/\alpha]\alpha' = \begin{cases} \tau & \alpha' = \alpha \\ \alpha' & \texttt{else} \end{cases}$$

$$[\tau/\alpha](\tau_1 \to \tau_2) = [\tau/\alpha]\tau_1 \to [\tau/\alpha]\tau_2$$

$$[\tau/\alpha](\forall \alpha' .\, \tau') = \begin{cases} \forall \alpha' .\, \tau' & \alpha' = \alpha \\ \forall \beta .\, [\tau/\alpha][\beta/\alpha']\tau' & \texttt{else (}\beta\texttt{ is fresh)} \end{cases}$$

α-renaming

If we have variables in types, we also need to define *substitution* in types

And we have to deal with capture avoidance!

Example (Substitution)

$$[(\top \to \alpha)/\beta](\forall \alpha. \beta \to \alpha) =$$

$$[\top \to \alpha/\beta](\forall \eta. \beta \to \eta) =$$

$$\forall \eta. (\top \to \alpha) \to \eta$$

Example (Derivation)

$$(T \to T) \to (T \to T) = [T \to T/\alpha]\,\alpha \to \alpha$$

$$\dfrac{\overline{\{x:\alpha\} \vdash x:\alpha}}{\{x:\alpha\} \vdash \Lambda\alpha.\,x : \forall\alpha.\,\alpha}$$

not possible

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{x:\alpha\} \vdash x:\alpha}}{\cdot \vdash \lambda x^{\alpha}.\,x : \alpha \to \alpha}}{\cdot \vdash \Lambda\alpha.\,\lambda x^{\alpha}.\,x : \forall\alpha.\,\alpha \to \alpha}}{\cdot \vdash (\Lambda\alpha.\,\lambda x^{\alpha}.\,x)(T \to T) : (T \to T) \to (T \to T)} \qquad \dfrac{\dfrac{}{\{x:T\} \vdash x:T}}{\cdot \vdash \lambda x^{T}.\,x : T \to T}$$

$$\cdot \vdash \underbrace{\boxed{(\Lambda\alpha.\,\underbrace{\boxed{\lambda x^{\alpha} x}}_{\text{identity}})}}_{\text{polymorphic id.}}(T \to T)\underbrace{\boxed{\lambda x^{T}.\,x}}_{\text{identity}} : T \to T$$

Drawbacks

```
let k = fun 'a 'b (x : 'a) (y : 'b) -> x
let out = k int (bool -> int) 4 (fun b -> if b then 0 else 1)
```

Explicitly passing types as arguments is *clunky*

And maybe we should be able to "tell from context" what the instantiated types are...

<u>OCaml's approach:</u> *we'll figure out the "most general" type you need to pass in from context*

demo

(System F)

Comparison with Curry-Typing

Does dropping type annotations automatically give use polymorphism? (No)

Comparison with Curry-Typing

STLC (Curry)

$$\frac{}{\Gamma \vdash \bullet : \top} \qquad \frac{(x:\tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x:\tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'} \qquad \frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'}$$

no type annotations

$\cdot \vdash \lambda x.x : \text{int} \to \text{int}$ ✓

$\cdot \vdash \lambda x.x : \text{bool} \to \text{bool}$

design this

let id = λx.x in
let a = id 0 in
let b = id • in
•

✗

Does dropping type annotations automatically give use polymorphism? (No)

Summary

- Implementing **parametric polymorphism** means fundamentally changing our type system
- Polymorphism requires the introduction of **type variables** and **type quantification** in order to *generalize* over possible types