Part I

Quiz

What type is inferred for `?` in the following expression?

```
{let {[f : (? -> ?)
           {lambda {[x : ?]} x}]}
  {f 10}}
```

Answer: *num*

Quiz

What type is inferred for `?` in the following expression?

```
{let {[f : (? -> ?)
          {lambda {[x : ?]} x}]}
  {f {lambda {[x : num]} x}}}
```

Answer: (*num* → *num*)

Quiz

What type is inferred for `?` in the following expression?

```
{let {[f : (? -> ?)
          {lambda {[x : ?]} x}]}
  {if ...
      {f 10}
      {{f {lambda {[x : num]} x}}
       8}}}
```

Answer: None; no single type works — but it's a perfectly good program for any ... of type `bool`

Polymorphism

We'd like a way to write a type that the caller chooses:

```
{let {[f : ?
          [LAMBDA ['a]
            {lambda {[x : 'a]} x}]]}
  {if …
      {[@ f num] 10}
      {{[@ f (num -> num)] {lambda {[x : num]} x}}
       8}}}
```

This `f` is ***polymorphic***

- The **`LAMBDA`** form parameterizes over a type
- The `@` form picks a type

Polymorphic Types

What is the type of this expression?

```
[LAMBDA ['a]
  {lambda {[x : 'a]} x}]
```

It should be something like (`'a` → `'a`), but it needs a specific type before it can be used as a function

Polymorphic Types

What is the type of this expression?

```
[LAMBDA ['a]
  [LAMBDA ['b]
    {lambda {[x : 'a]} x}]]
```

It should be something like (`'a` → `'a`), but picking `'a` gives something that still needs another type

New type form: (∀ (`'`**`<Sym>`**) **`<Type>`**)

(∀ (`'a`) (`'a` → `'a`))

(∀ (`'a`) (∀ (`'b`) (`'a` → `'a`)))

Polymorphic Types

What is the type of this expression?

```
[LAMBDA ['a]
  [LAMBDA ['b]
    {lambda {[x : 'a]} x}]]
```

It should be something like (`'a` → `'a`), but picking `'a` gives something that still needs another type

New type form: (∀ (**a**) τ)

(∀ (`'a`) (`'a` → `'a`))

(∀ (`'a`) (∀ (`'b`) (`'a` → `'a`)))

Grammar with Polymorphism

```
<Expr>  ::=  <Num>
          |  {+ <Expr> <Expr>}
          |  {* <Expr> <Expr>}
          |  <Sym>
          |  {lambda {[<Sym> : <Type>]} <Expr>}
          |  {<Expr> <Expr>}
          |  [LAMBDA ['<Sym>] <Expr>]                NEW
          |  [@ <Expr> <Type>]                       NEW

<Type>  ::=  num
          |  (<Type> -> <Type>)
          |  (forall ('<Sym>) <Type>)               NEW
          |  '<Sym>                                 NEW
```

Part 2

Datatypes

```
(define-type ExprC
  ....
  [tylamC (n : symbol)
          (body : ExprC)]
  [tyappC (tyfun : ExprC)
          (tyarg : Type)])

(define-type Value
  ....
  [polyV (body : ExprC)
         (env : Env)])

(define-type Type
  ....
  [varT (n : symbol)]
  [forallT (n : symbol)
           (body : Type)])
```

Examples

```
(test (interp (parse '[LAMBDA ['a]
                              {lambda {[x : 'a]} x}])
              mt-env)
      (polyV (lamC 'x (varT 'a) (idC 'x))
             mt-env))
```

Examples

```
(test (interp (parse '[@ [LAMBDA ['a]
                           {lambda {[x : 'a]} x}]
                         num])
              mt-env)
      (closV 'x (idC 'x) mt-env))
```

Examples

```
(test (interp (parse '{[@ [LAMBDA ['a]
                            {lambda {[x : 'a]} x}]
                          num]
                       8})
              mt-env)
      (numV 8))
```

Examples

```
(test (parse-type '(forall ('a) ('a -> 'a)))
      (forallT 'a (arrowT (varT 'a) (varT 'a))))
```

Examples

```
(test (typecheck (parse '[LAMBDA ['a]
                                 {lambda {[x : 'a]} x}])
                 mt-env)
      (forallT 'a (arrowT (varT 'a) (varT 'a))))
```

Examples

```
(test/exn (typecheck (parse '{lambda {[x : 'a]} x})
                     mt-env)
          "no type")
```

Examples

```
(test (typecheck (parse '{lambda {[x : 'a]} x})
                 (extend-env
                  (tvar 'a)
                  mt-env))
      (arrowT (varT 'a) (varT 'a)))
```

Examples

```
(test (typecheck (parse '[LAMBDA ['a]
                                 {lambda {[x : 'a]} x}])
                 mt-env)
      (forallT 'a (arrowT (varT 'a) (varT 'a))))
```

Examples

```
(test (typecheck (parse '[@ [LAMBDA ['a]
                              {lambda {[x : 'a]} x}]
                            num])
                 mt-env)
      (arrowT (numT) (numT)))
```

Part 3

Type Checking

$$\frac{\Gamma[\,\mathbf{a}\,] \vdash \mathbf{e} : \tau}{\Gamma \vdash [\mathbf{LAMBDA}\ [\mathbf{a}]\ \mathbf{e}] : (\forall\ (\mathbf{a})\ \tau)}$$

$$\frac{\Gamma \vdash \tau_0 \qquad \Gamma \vdash \mathbf{e} : (\forall\ (\mathbf{a})\ \tau_1)}{\Gamma \vdash [@\ \mathbf{e}\ \tau_0] : \tau_1[\,\mathbf{a} \leftarrow \tau_0\,]}$$

$$[\,\ldots\mathbf{a}\ldots\,] \vdash \mathbf{a}$$

$$\frac{\Gamma[\,\mathbf{a}\,] \vdash \tau}{\Gamma \vdash (\forall\ (\mathbf{a})\ \tau)}$$

Part 4

Polymorphism and Type Definitions

If we mix **LAMBDA** with **let-type**, then we can write

```
{let {[f : (forall ('a) ('a -> num))
           [LAMBDA ['a]
             {lambda {[v : 'a]}
               {let-type {List {empty num}
                               {cons ('a * List)}}
                 {letrec {[len : (List -> num)
                                 {lambda {[l : List]}
                                   {cases List l
                                     {empty {n} 0}
                                     {cons {p}
                                           {+ 1 {len {snd p}}}}}}]}
                   {len {cons {pair v
                                    {cons {pair v
                                                {empty 0}}}}}}}}}]}
  {+ {[@ f num] 10}
     {[@ f (num -> num)] {lambda {[x : num]} x}}}}
```

This is a kind of polymorphic list Definition

Problem: everything must be under a **LAMBDA**

Polymorphism and Type Definitions

Solution: build **`LAMBDA`**-like abstraction into **`let-type`**

```
{let-type {(List 'a) {empty num}
                     {cons ('a * (List 'a))}}
  {letrec {[len : ((List num) -> num)
                {lambda {[l : (List num)]}
                  {cases (List num) l
                    {empty {n} 0}
                    {cons {p}
                          {+ 1 {len {snd p}}}}}}]}
    {len {[@ cons num] {pair 1 {[@ empty num] 0}}}}}}
```

Polymorphism and Type Definitions

Solution: build **`LAMBDA`**-like abstraction into **`let-type`**

```
{let-type {(List 'a) {empty num}
                     {cons ('a * (List 'a))}}
  {letrec {[len : (forall ('a) ((List 'a) -> num))
                  [LAMBDA ['a]
                    {lambda {[l : (List 'a)]}
                      {cases (List 'a) l
                        {empty {n} 0}
                        {cons {p}
                              {+ 1 {len {snd p}}}}}}}]]}
    {+ {[@ len num] {[@ cons num] {pair 1 {[@ empty num] 0}}}}
       {[@ len (num -> num)] {[@ empty (num -> num)] 0}}}}}
```

Part 5

Polymorphism and Inference

```
{let {[f : (forall ('a) ('a -> 'a))
          [LAMBDA ['a]
            {lambda {[x : 'a]}
              x}]]}
  {[@ f (num -> num)] {lambda {[y : num]} y}}}
```

The type application `[@ f (num -> num)]` is obvious, since we can get the type of `{lambda {[y : num]} y}`

With polymorphism, type inference is usually combined with type-application inference:

```
{let {[f : (forall ('a) ('a -> 'a))
          [LAMBDA ['a]
            {lambda {[x : 'a]}
              x}]]}
  {f {lambda {[y : num]} y}}}
```

Polymorphism and Inference

```
{let {[f : ?
          {lambda {[x : ?]}
            x}]}
  {f {lambda {[y : num]} {f 10}}}}
```

How about inferring a **LAMBDA** around the value of **f**?

Yes, with some caveats...

Polymorphism and Inference

Does the following expression have a type?

```
{lambda {[x : ?]} {x x}}
```

Yes, if we infer `forall` types and type applications:

```
{lambda {[x : (forall ('a) ('a -> 'a))]}
  {[@ x (num -> num)] [@ x num]}}
```

Inferring types like this is arbitrarily difficult (i.e., undecidable), so type systems generally don't

Let-Based Polymorphism

Inference constraint: only infer a polymorphic type (and insert **`LAMBDA`**) for ther right-hand side of a **`let`** or **`letrec`** binding

- This works:

```
{let {[f : ?
          {lambda {[x : ?]}
            x}]}
  {f {lambda {[y : num]} {f 10}}}}
```

- This doesn't:

```
{lambda {[x : ?]} {x x}}
```

Note: makes **`let`** a core form

Implementation: check right-hand side, add a **`forall`** and **`LAMBDA`** for each unconstrained *new* type variable

Polymorphism and Inference and Type Definitions

All three together make a practical programming system:

```
{let-type {(List 'a) {empty num}
                     {cons ('a * (List 'a))}}
  {letrec {[len : ?
                {lambda {[l : (List 'a)]}
                  {cases (List 'a) l
                    {empty {n} 0}
                    {cons {p}
                          {+ 1 {len {snd p}}}}}}]}
    {+ {len {cons {pair 1 {empty 0}}}}
       {len {cons {pair {lambda {[x : num]} x} {empty 0}}}}}}}
```

Part 6

Polymorphic Types

```
(lambda (x) x)
```

Why does **plai-typed** show

```
('a -> 'a)
```

instead of

```
(forall ('a) ('a -> 'a))
```

?

Polymorphism and Values

A ***polymorphic function*** is not quite a function

- A **function** is applied to a value to get a new value

```
f : (num -> num)
{f 1}
```

- A **polymorphic function** is applied to a type to get a function

```
pf : (forall ('a) ('a -> 'a))
{[@ pf num] 1}

{pf 1} no type
```

Polymorphism and Values

A ***polymorphic function*** is not quite a function

What happens if you write the following?

```
{let {[f : ? {lambda {[g : ?]}
                {lambda {[v : ?]}
                  {g v}}}]}
  {let {[g : ? {lambda {[x : ?]} x}]}
    {{f g} 10}}}
```

A type application must be used at the function call, not in `f`:

```
{{[@ [@ f num] num] 10} [@ g num]}
```

Polymorphism and Values

A ***polymorphic function*** is not quite a function

What happens if you write the following?

```
{let {[f : ? {lambda {[v : ?]}
               {lambda {[g : (forall ('a) ('a -> 'a))]}
                 {g v}}}]}
  {let {[g : ? {lambda {[x : ?]} x}]}
    {{f 10} g}}}
```

One type application must be used inside `f`:

```
[LAMBDA {'b} {lambda {[v : 'b]}
               {lambda {[g : (forall ('a) ('a -> 'a))]}
                 {[@ g 'b] v}}}]
```

Polymorphism and Values

An argument that is a polymorphic value can be used in multiple ways:

```
{lambda {[g : (forall ('a) ('a -> 'a))]}
  {if {g false}
      {g 0}
      {g 1}}}
```

but due to inference constraints,

```
{lambda {[g : ?]}
  {if {g false}
      {g 0}
      {g 1}}}
```

would be rejected!

Polymorphism and Values

ML and **`plai-typed`** prohibit polymorphic values, so that

```
{lambda {[g : (forall ('a) ('a -> 'a))]}
  {if {g false}
      {g 0}
      {g 1}}}
```

is not allowed

- Consistent with inference
- Every **`forall`** appears at the beginning of a type, so

```
(forall ('a) (forall ('b) ('a -> 'b)))
```

  can be abbreviated

```
('a -> 'b)
```

  without loss of information