Part I

Inferring a Function Type

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

```
let ident :: ? -> ? = (fun (x :: ?): x):
  ident(10)
```

Answer: `Int`

Inferring a Function Type

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

```
let ident :: ? -> ? = (fun (x :: ?): x):
  ident(fun (y :: Int): y)
```

Answer: `Int -> Int`

Inferring a Function Type

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

```
let ident :: ? -> ? = (fun (x :: ?): x):
  if …
  | ident(10)
  | ident(fun (y :: Int): y)(8)
```

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

Polymorphism

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

```
let ident :: ? = (FUN [?a]:
                    fun (x :: ?a): x):
  if …
  | ident[Int](10)
  | ident[Int -> Int](fun (y :: Int): y)(8)
```

This `ident` is ***polymorphic***

- The `FUN` form parameterizes over a type
- The `e[τ]` form picks a type

Polymorphic Types

What is the type of this expression?

```
FUN [?a]:
  fun (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?

```
FUN [?a]:
  FUN [?b]:
    fun (x :: ?a): x
```

It should be something like `?a -> ?a`, but picking `?a` gives something that still needs another type

New type form: `forall (?<Symbol>): <Type>`

```
forall (?a): ?a -> ?a
```

```
forall (?a): forall (?b): ?a -> ?a
```

Polymorphic Types

What is the type of this expression?

```
FUN [?a]:
  FUN [?b]:
    fun (x :: ?a): x
```

It should be something like `?a -> ?a`, but picking `?a` gives something that still needs another type

New type form: `forall (?<Symbol>): <Type>`

```
forall (?a): ?a -> ?a
```

```
forall (?a): forall (?b): ?a -> ?a
```

Grammar with Polymorphism

```
<Exp>   ::=  <Int>
          |  <Exp> + <Exp>
          |  <Exp> * <Exp>
          |  <Symbol>
          |  fun (<Symbol> :: <Type>): <Exp>
          |  <Exp>(<Exp>)
          |  FUN [?<Symbol>]: <Exp>                    NEW
          |  <Exp>[<Type>]                             NEW

<Type>  ::=  Int
          |  <Type> -> <Type>
          |  forall (?<Symbol>): <Type>                NEW
          |  ?<Symbol>                                 NEW
```

Part 2

Datatypes

```
type Exp
....
| tyfunE(n :: Symbol,
          body :: Exp)
| tyappE(tyfun :: Exp,
          tyarg :: Type)

type Value
....
| polyV(body :: Exp,
         env :: Env)

type Type
....
| idT(n :: Symbol)
| forallT(n :: Symbol,
           body :: Type)
```

Examples

```
check: interp(parse('FUN [?a]:
                       fun (x :: ?a): x'),
              mt_env)
       ~is polyV(funE(#'x, idT(#'a), idE(#'x)),
                 mt_env)
```

Examples

```
check: interp(parse('(FUN [?a]:
                        fun (x :: ?a): x)[Int]'),
              mt_env)
       ~is closV(#'x, idE(#'x), mt_env)
```

Examples

```
check: interp(parse('(FUN [?a]:
                        fun (x :: ?a): x)[Int](8)'),
              mt_env)
       ~is intV(8)
```

Examples

```
check: parse_type('forall (?a): ?a -> ?a')
       ~is forallT(#'a, arrowT(idT(#'a), idT(#'a)))
```

Examples

```
check: typecheck(parse('FUN [?a]:
                          fun (x :: ?a): x'),
                mt_env)
       ~is forallT(#'a, arrowT(idT(#'a), idT(#'a)))
```

Examples

```
check: typecheck(parse('fun (x :: ?a): x'),
                 mt_env)
       ~raises "no type"
```

Examples

```
check: typecheck(parse('fun (x :: ?a): x'),
                 extend_env(tid(#'a),
                            mt_env))
       ~is arrowT(idT(#'a), idT(#'a))
```

Examples

```
check: typecheck(parse('FUN [?a]:
                          fun (x :: ?a): x'),
                mt_env)
       ~is forallT(#'a, arrowT(idT(#'a), idT(#'a)))
```

Examples

```
check: typecheck(parse('(FUN [?a]:
                           fun (x :: ?a): x)[Int]'),
                 mt_env)
       ~is arrowT(intT(), intT())
```

Part 3

Type Checking

$$\frac{\Gamma[\,\mathbf{a}\,] \vdash \mathbf{e} : \tau}{\Gamma \vdash \texttt{FUN [? a] : e : forall [? a] : } \tau}$$

$$\frac{\Gamma \vdash \tau_0 \qquad \Gamma \vdash \mathbf{e} : \texttt{forall [? a] : } \tau_1}{\Gamma \vdash \mathbf{e}[\tau_0] : \tau_1[\,\mathbf{a} \leftarrow \tau_0\,]}$$

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

$$\frac{\Gamma[\,\mathbf{a}\,] \vdash \tau}{\Gamma \vdash \texttt{forall [? a] : } \tau}$$

Part 4

Polymorphism and Type Definitions

If we mix **FUN** with **let_type**, then we can write

```
let f :: (forall (?a): ?a -> Int) = (FUN [?a]:
                                      fun (v :: ?a):
                                        let_type (List | empty(arg :: Int)
                                                       | cons(arg :: ?a * List)):
                                          letrec len :: List -> Int = (fun (l :: List):
                                                                         match l
                                                                         | empty(n): 0
                                                                         | cons(p):
                                                                             1 + len(snd(p))):
                                            len(cons(values(v,
                                                            cons(values(v,
                                                                        empty(0)))))))):
  f[Int](10) + f[Int -> Int](fun (x :: Int): x)
```

This is a kind of polymorphic list definition

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

Polymorphism and Type Definitions

Solution: build **`FUN`**-like abstraction into **`let_type`**

```
let_type (Listof(?a) | empty(arg :: Int)
                     | cons(arg :: ?a * Listof(?a))):
  letrec len :: Listof(Int) -> Int = (fun (l :: Listof(Int)):
                                        match l
                                        | empty(n): 0
                                        | cons(p): 1 + len(snd(p))):
    len(cons[Int](values(1, empty[Int](0))))
```

Polymorphism and Type Definitions

Solution: build **`FUN`**-like abstraction into **`let_type`**

```
let_type (Listof(?a) | empty(arg :: Int)
                     | cons(arg :: ?a * Listof(?a))):
  letrec len :: (forall (?a): Listof(?a) -> Int) = (FUN [?a]:
                                                      fun (l :: Listof(?a)):
                                                        match l
                                                        | empty(n): 0
                                                        | cons(p): 1 + len(snd(p))):
    len[Int](cons[Int](values(1, empty[Int](0))))
      + len[Int -> Int](empty[Int -> Int](0))
```

Part 5

Polymorphism and Inference

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

```
let ident = (fun (x :: ?a):
               x):
  ident(fun (y :: Int): y)

⇒

let ident :: (forall (?a): ?a -> ?a) = (FUN [?a]:
                                          fun (x :: ?a):
                                            x):
  ident[Int -> Int](fun (y :: Int): y)
```

Polymorphism and Inference

The `@` part is easy:

```
let ident :: (forall (?a): ?a -> ?a) = (FUN [?a]:
                                          fun (x :: ?a):
                                            x):
  ident(fun (y :: Int): y)
```

The type application `ident[Int -> Int]` is obvious, since we can get the type of `fun (y :: Int): y`

Polymorphism and Inference

```
let ident :: ? = (fun (x :: ?):
                    x):
  ident(fun (y :: Int): ident(10))
```

How about inferring a **FUN** around the value of **ident**?

Yes, with some caveats...

Polymorphism and Inference

Does the following expression have a type?

```
fun (x :: ?): x(x)
```

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

```
fun (x :: forall (?a): ?a -> ?a):
  x[Int -> Int](x[Int])
```

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 **`FUN`**) for the right-hand side of a **`let`** or **`letrec`** binding

- This works:

```
let ident :: ? = (fun (x :: ?):
                    x):
  ident(fun (y :: Int): ident(10))
```

- This doesn't:

```
fun (x :: ?): x(x)
```

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

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

Polymorphism and Inference and Type Definitions

All three together make a practical programming system:

```
let_type (Listof(?a) | empty(arg :: Int)
                     | cons(arg :: ?a * Listof(?a))):
  letrec len :: ? = (fun (l :: Listof(?a)):
                       match l
                       | empty(n): 0
                       | cons(p): 1 + len(snd(p))):
    len(cons(values(1, empty(0))))
      + len(cons(values(fun (x :: Int): x, empty(0))))
```

Part 6

Polymorphic Types

```
fun (x): x
```

Why does Shplait 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 :: Int -> Int
f(1)
```

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

```
pf :: forall (?a): ?a -> ?a
pf[Int](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 :: ? = (fun (v :: ?):
                fun (h :: ?):
                  h(v)):
  let g :: ? = (fun (x :: ?): x):
    f(10)(g)
```

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

```
f[Int][Int](10)(g[Int])
```

Polymorphism and Values

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

What happens if you write the following?

```
let f :: ? = (fun (v :: ?):
                fun (h :: forall (?a): ?a -> ?a):
                  h(v)):
  let g :: ? = (fun (x :: ?): x):
    f(10)(g)
```

One type application must be used inside `f`:

```
FUN [?b]: fun (v :: ?b):
            fun (h :: forall (?a): ?a -> ?a):
              h[?b](v)
```

Polymorphism and Values

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

```
fun (g :: forall (?a): ?a -> ?a):
  if g(#false)
  | g(0)
  | g(1)
```

but due to inference constraints,

```
fun (g :: ?):
  if g(#false)
  | g(0)
  | g(1)
```

would be rejected!

Polymorphism and Values

Shplait (like OCaml) prohibits polymorphic values, so that

```
fun (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