

Quiz

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

```
{with {f : (? -> ?) {fun {x : ?} x}}  
  {f 10}}
```

- **Answer:** *num*

Quiz

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

```
{with {f : (? -> ?) {fun {x : ?} x}}
  {f {fun {x : num} x}}}
```

- **Answer:** $(num \rightarrow num)$

Quiz

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

```
{with {f : (? -> ?) {fun {x : ?} x}}
  {if0 ...
    {f 10}
    {{f {fun {x : num} x}} 8}}}
```

- **Answer:** None; no single τ works – but it's a perfectly good program for any ...

Polymorphism

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

```
{with {f : ?
      [tyfun [alpha]
            {fun {x : alpha} x}]}]
  {if0 ...
    {[@ f num] 10}
    {{[@ f (num -> num)] {fun {x : num} x}} 8}}}}
```

This `f` is *polymorphic*

- The `tyfun` form parameterizes over a type
- The `@` form picks a type

Polymorphic Types

What is the type of this expression?

```
[tyfun [alpha]  
      {fun {x : alpha} x}]
```

It should be something like ($\text{alpha} \rightarrow \text{alpha}$), but it needs a specific type before it can be used as a function

Polymorphic Types

What is the type of this expression?

```
[tyfun [alpha]  
  [tyfun [beta]  
    {fun {x : alpha} x}]]]
```

It should be something like ($\mathbf{alpha} \rightarrow \mathbf{alpha}$), but picking \mathbf{alpha} gives something that still needs another type

New type form: $\forall \langle \mathbf{tyid} \rangle . \langle \mathbf{tyexp} \rangle$

$$\forall \mathbf{alpha} . (\mathbf{alpha} \rightarrow \mathbf{alpha})$$
$$\forall \mathbf{alpha} . \forall \mathbf{beta} . (\mathbf{alpha} \rightarrow \mathbf{alpha})$$

TPFAE Grammar

```
<TPFAE> ::= <num>
          | {+ <TPFAE> <TPFAE>}
          | {- <TPFAE> <TPFAE>}
          | <id>
          | {fun {<id> : <tyexp>} <TPFAE>}
          | {<TPFAE> <TPFAE>}
          | {if0 <TPFAE> <TPFAE> <TPFAE>}
          | [tyfun [<tyid>] <TPFAE>]
          | [@ <TPFAE> <tyexp>]
<tyexp> ::= num
          | (<tyexp> -> <tyexp>)
          | (forall <tyid> <tyexp>)
          | <tyid>
```

NEW

NEW

NEW

NEW

TPFAE Type Checking

$$\frac{\Gamma[\langle \text{tyid} \rangle] \vdash e : \tau}{\Gamma \vdash [\text{tyfun } [\langle \text{tyid} \rangle] e] : \forall \langle \text{tyid} \rangle . \tau}$$

$$\frac{\Gamma \vdash \tau_0 \quad \Gamma \vdash e : \forall \langle \text{tyid} \rangle . \tau_1}{\Gamma \vdash [@ e \tau_0] : \tau_1 [\langle \text{tyid} \rangle \leftarrow \tau_0]}$$

$$[\dots \langle \text{tyid} \rangle \dots] \vdash \langle \text{tyid} \rangle$$

$$\frac{\Gamma[\langle \text{tyid} \rangle] \vdash \tau}{\Gamma \vdash \forall \langle \text{tyid} \rangle . \tau}$$

Polymorphism and Type Definitions

If we mix `tyfun` with `withtype`, then we can write

```
{with {f : (forall alpha (alpha -> num))
      [tyfun [alpha]
            {fun {v : alpha}
              {withtype {list {empty num}
                        {cons (alpha * list)}}
                {rec {len : (list -> num)
                    {fun {l : list}
                      {cases list l
                        {empty {n} 0}
                        {cons {fxr}
                          {+ 1 {len {snd fxr}}}}}}}}
                  {len {cons {pair v
                            {cons {pair v
                                  {empty 0}}}}}}}}1}
      {+ {[@ f num] 10}
        {[@ f (num -> num)] {fun {x : num} x}}}}
```

This is a kind of polymorphic list definition

Problem: everything must be under a `tyfun`

Polymorphism and Type Definitions

Solution: build `tyfun`-like abstraction into `withtype`

```
{withtype {{alpha list} {empty num}
           {cons (alpha * {alpha list})}}
 {rec {len : (forall alpha ({alpha list} -> num))
      [tyfun [alpha]
            {fun {l : {alpha list}}
              {cases {alpha list} l
                    {empty {n} 0}
                    {cons {fxr}
                        {+ 1 {len {snd fxr}}}}}}]}
      {+ {[@ len num] {[@ cons num] {pair 1 {[@ empty num] 0}}}}
         {[@ len (num -> num)] {[@ empty (num -> num)] 0}}}}}
```

Polymorphism and Inference

```
{with {f : (forall alpha (alpha -> alpha))
      [tyfun [alpha]
             {fun {x : alpha}
               x}]]}
  {[@ f (num -> num)] {fun {y : num} y}}}
```

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

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

```
{with {f : (forall alpha (alpha -> alpha))
      [tyfun [alpha]
             {fun {x : alpha}
               x}]]}
  {f {fun {y : num} y}}}
```

Polymorphism and Inference

```
{with {f : ?  
      {fun {x : ?}  
          x}}  
  {f {fun {y : num} {f 10}}}}
```

How about inferring a `tyfun` around the value of `f`?

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 alpha (alpha -> alpha))}  
  {[@ 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 `tyfun`) for the right-hand side of a `with` or `rec` binding

- This works:

```
{with {f : ?
      {fun {x : ?}
        {x}}}
      {f {fun {y : num} {f 10}}}}
```

- This doesn't:

```
{fun {x : ?} {x x}}
```

Note: makes `with` a core form

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

Polymorphism and Inference and Type Definitions

All three together make a practical programming system:

```
{withtype {{alpha list} {empty num}
           {cons (alpha * {alpha list})}}
{rec {len : ?
     {fun {l : {alpha list}}
        {cases {alpha list} l
              {empty {n} 0}
              {cons {fxr}
                    {+ 1 {len {snd fxr}}}}}}}
     {+ {len {cons {pair 1 {empty 0}}}}
        {len {cons {pair {fun {x : num} x} {empty 0}}}}}}}
```

Caml example:

```
type 'a tree = Leaf of 'a
             | Fork of 'a tree * 'a tree
```

Polymorphism and Values

A *polymorphic function* is not quite a function:

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

What happens if you write the following?

```
{with {f : ? {fun {v : ?}
              {fun {g : ?}
                  {g v}}}}}
  {with {g : ? {fun {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:

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

What happens if you write the following?

```
{with {f : ? {fun {v : ?}
              {fun {g : (forall alpha (alpha -> alpha))}
                  {g v}}}}}
  {with {g : ? {fun {x : ?} x}}
    {{f 10} g}}}
```

One type application must be used inside **f**:

```
[tyfun {beta} {fun {v : beta}
                {fun {g : (forall alpha (alpha -> alpha))}
                    {[@ g beta] v}}}]
```

Polymorphism and Values

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

```
{fun {g : (forall alpha (alpha -> alpha))}
  {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

ML prohibits polymorphic values, so that

```
{fun {g : (forall alpha (alpha -> alpha))}
  {if {g false}
     {g 0}
     {g 1}}}
```

is not allowed

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

```
(forall alpha (forall beta (alpha -> beta)))
```

can be abbreviated

```
(alpha -> beta)
```

without loss of information