Quiz

What type is inferred for ? in the following expression?

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

```latex
\{\textbf{with } \{f : (? \rightarrow ?) \ {\textbf{fun } \{x : ?\} \ x}\} \n\{f \ {\textbf{fun } \{x : \textbf{num}\} \ x}\}\} \n```
Quiz

What type is inferred for ? in the following expression?

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

Answer: \((\text{num} \rightarrow \text{num})\)
Quiz

What type is inferred for ? in the following expression?

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

What type is inferred for \( ? \) in the following expression?

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

**Answer:** None; no single \( \tau \) works — but it’s a perfectly good program for any \( ... \) of type `num`
Polymorphism

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

```ocaml
{with {f : ?
   [tyfun [alpha]
      {fun {x : alpha} x}]}

{if0 ...
  {[@ f num] 10}
  {[[@ f (num -> num)] {fun {x : num} x}] 8}}
```
Polymorphism

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

```plaintext
{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?

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

\[
\begin{align*}
\text{tyfun} & \quad [\alpha] \\
\text{tyfun} & \quad [\beta] \\
\{ \text{fun} \ {x : \alpha} \ x \} & \]
\end{align*}
\]
Polymorphic Types

What is the type of this expression?

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

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

What is the type of this expression?

\[
\text{[tyfun \ [alpha]}
\text{[tyfun \ [beta]}
\text{\{fun \ \{x : \alpha\} \ x\}]]}
\]

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

New type form: \(\forall <\text{tyid}>.<\text{TE}>\)

\(\forall \alpha. (\alpha \rightarrow \alpha)\)

\(\forall \alpha. \forall \beta. (\alpha \rightarrow \alpha)\)
TPFAE Grammar

\[
\begin{align*}
\text{<TPFAE>} & ::= \text{<num>} \\
& \mid \{+ \text{<TPFAE>} \text{<TPFAE>}\} \\
& \mid \{- \text{<TPFAE>} \text{<TPFAE>}\} \\
& \mid \text{<id>} \\
& \mid \{\text{fun}\ \{\text{id} : \text{<TE>}\} \text{<TPFAE>}\} \\
& \mid \{\text{<TPFAE>} \text{<TPFAE>}\} \\
& \mid [\text{tyfun}\ [\text{<tyid>}] \text{<TPFAE>}] \\
& \mid [@ \text{<TPFAE>} \text{<TE>}] \\
\text{<TE>} & ::= \text{num} \\
& \mid (\text{<TE>} \rightarrow \text{<TE>}) \\
& \mid (\text{forall} \ \text{<tyid>} \ \text{<TE>}) \\
& \mid \text{<tyid>}
\end{align*}
\]
TPFAE Type Checking

\[ \Gamma[\text{<tyid>}] \vdash e : \tau \]

\[ \Gamma \vdash \text{tyfun } [\text{<tyid>]} e : \forall\text{<tyid>}.\tau \]

\[ \Gamma \vdash \tau_0 \quad \Gamma \vdash e : \forall\text{<tyid>}.\tau_1 \]

\[ \Gamma \vdash \text{@ e } \tau_0 : \tau_1[<\text{tyid}> \leftarrow \tau_0] \]

\[ [...\text{<tyid>...}] \vdash <\text{tyid}> \]

\[ \Gamma[\text{<tyid>}] \vdash \tau \]

\[ \Gamma \vdash \forall\text{<tyid>}.\tau \]
Polymorphism and Type Definitions

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

```plaintext
{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}}}}}]]}}}
    {[@ 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`

```plaintext
{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

```haskell
{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}
Polymorphism and Inference

\[
\text{with } \{f : (\forall \alpha. (\alpha \to \alpha))\} \\
[\text{tyfun} \ [\alpha] \\
\{\text{fun} \ \{x : \alpha\} \\
\quad x\}]\}
\]

\[@ f (\text{num} \to \text{num})] \ \{\text{fun} \ \{y : \text{num}\} \ y\}\}

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

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

\[
\text{with } \{f : (\forall \alpha. (\alpha \to \alpha))\} \\
[\text{tyfun} \ [\alpha] \\
\{\text{fun} \ \{x : \alpha\} \\
\quad x\}]\}
\]

\{f \ \{\text{fun} \ \{y : \text{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`?
Polymorphism and Inference

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

How about inferring a tyfun around the value of \texttt{f}?

Yes, with some caveats...
Polymorphism and Inference

Does the following expression have a type?

\{ \texttt{fun} \ \{ x : ? \} \ \{ x \ x \} \}
Polymorphism and Inference

Does the following expression have a type?

\{
fun \{x : ?\} \{x x\}\n\}

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

\{
fun \{x : (forall alpha (alpha -> alpha))\}
[@ x (num -> num)] [@ x num]\}
Polymorphism and Inference

Does the following expression have a type?

\{\texttt{fun } \{x : ?\} \{x x\}\}

Yes, if we infer \texttt{forall} types and type applications:

\{\texttt{fun } \{x : (forall alpha (alpha -> alpha))\}\}
\{\texttt{[@ x (num -> num)]} \texttt{[@ 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 \texttt{tyfun}) for ther right-hand side of a \texttt{with} or \texttt{rec} binding

• This works:

\[
\begin{align*}
\{ & \textbf{with} \ \{ f : ? \} \\
& \{ \textbf{fun} \ \{ x : ? \} \\
& \quad x \} \}
\{ & \textbf{fun} \ \{ y : \text{num} \} \ \{ f \ 10 \} \} \}
\end{align*}
\]

• This doesn’t:

\[
\{ \textbf{fun} \ \{ x : ? \} \ \{ x \ x \} \}
\]
Let-Based Polymorphism

Inference constraint: only infer a polymorphic type (and insert \texttt{tyfun}) for ther right-hand side of a \texttt{with} or \texttt{rec} binding

- This works:

\[
\begin{align*}
\{\textbf{with} & \ \{f : ?\} \\
\{\textbf{fun} & \ \{x : ?\} \\
& x\}\}
\end{align*}
\]

- This doesn’t:

\[
\begin{align*}
\{\textbf{fun} & \ \{x : ?\} \ \{x \ x\}\}
\end{align*}
\]

\textbf{Note}: makes \texttt{with} a core form

\textbf{Implementation}: check right-hand side, add a \texttt{forall} and \texttt{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}}}}}}}
```
Polymorphism and Inference and Type Definitions

All three together make a practical programming system:

```ocaml
{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:

```ocaml
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
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?

```ocaml
{with {f : ?} {fun {g : ?} {fun {v : ?} {g v}}}}
{with {g : ?} {fun {x : ?} x}}
{{f g} 10}}
```
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?

```plaintext
{with {f : ? {fun {g : ?}
    {fun {v : ?}
        {g v}}}}
{with {g : ? {fun {x : ?} x}}
{{f g} 10}}}
```

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

```plaintext
{[[@ [@ 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?

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

```plaintext
{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 \( f \):

```plaintext
[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:

\[
\begin{align*}
\text{fun } & \{g : (\forall \alpha. \alpha \to \alpha)\} \\
\text{if } & \{g \ false\} \\
& \{g \ 0\} \\
& \{g \ 1\}\}
\end{align*}
\]
Polymorphism and Values

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

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

but due to inference constraints,

```plaintext
{fun {g : ?}
   {if {g false}
       {g 0}
       {g 1}}}
```

would be rejected!
Polymorphism and Values

ML prohibits polymorphic values, so that

```
{fun \textbf{g} : (forall \alpha \ (\alpha \to \alpha))}
{\textbf{if} \ g \ \textbf{false}}
{\textbf{g} \ 0}
{\textbf{g} \ 1}}
```

is not allowed
Polymorphism and Values

ML prohibits polymorphic values, so that

\[
\{ \text{fun } \{g : (\forall \alpha. \alpha \to \alpha)\} \\
\{ \text{if } \{g \text{ false}\} \\
\{g \ 0\} \\
\{g \ 1\}\}\}
\]

is not allowed

• Consistent with inference

• Every \texttt{forall} appears at the beginning of a type, so

\[
(\forall \alpha. (\forall \beta. \alpha \to \beta))
\]

can be abbreviated

\[
(\alpha \to \beta)
\]

without loss of information