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?

\[
\{ \text{with } \{ f : (? \to ?) \ { \text{fun } \{ x : ? \} \ x } \} \\
\{ f \ { \text{fun } \{ x : \text{num} \} \ x } \} \}
\]

**Answer:** \((\text{num} \to \text{num})\)
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 \( \tau \) 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:

```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?

```
[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 \((\alpha \to \alpha)\), but picking \(\alpha\) gives something that still needs another type.

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

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

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

\[
\begin{align*}
<TPFAE> & ::= \ <num> \\
& \mid \{ + \ TPFAE \ TPFAE \}\ \\
& \mid \{- \ TPFAE \ TPFAE\}\ \\
& \mid <id> \\
& \mid \{ \text{fun} \{<id> : <TE>\} \ TPFAE \}\ \\
& \mid \{TPFAE \ TPFAE\}\ \\
& \mid \{\text{if0} \ TPFAE \ TPFAE \ TPFAE\}\ \\
& \mid \text{tyfun} \ [tyid] \ TPFAE\] \\
& \mid [@ \ TPFAE \ TE]\ \\
\end{align*}
\]

\[
<TE> ::= \ \text{num} \\
& \mid (<TE> \rightarrow <TE>) \\
& \mid (\text{forall} \ <tyid> <TE>)\ \\
& \mid <tyid>
\]

\[
\]
TPFAE Type Checking

\[
\begin{align*}
\Gamma[\text{<tyid>}] & \vdash e : \tau \\
\hline \\
\Gamma & \vdash [\text{tyfun} [\text{<tyid>}] e] : \forall<\text{tyid}>().\tau \\
\hline \\
\Gamma & \vdash \tau_0 \\
\Gamma & \vdash e : \forall<\text{tyid}>().\tau_1 \\
\hline \\
\Gamma & \vdash [\text{@} e \ \tau_0] : \tau_1[<\text{tyid}>\mapsto \tau_0] \\
\hline \\
[...<\text{tyid}>...] & \vdash <\text{tyid}> \\
\hline \\
\Gamma[<\text{tyid}>] & \vdash \tau \\
\hline \\
\Gamma & \vdash \forall<\text{tyid}>().\tau
\end{align*}
\]
Polymorphism and Type Definitions

If we mix \texttt{tyfun} with \texttt{withtype}, then we can write

\begin{verbatim}
{with \{f : (forall alpha (alpha \rightarrow num))
    [tyfun [alpha]
        {fun \{v : alpha\}
            {withtype \{list \{empty num\}
                {cons \(alpha \ast list\)}}}
        {rec \{len : (list \rightarrow num)\}
            {fun \{l : list\}
                {cases list l
                    {empty \{n\} 0}
                    {cons \{fxr\}
                        {+ 1 \{len \{snd fxr\}\}}}}}}}
    [+ \{[@ f num] 10\}
        {[@ f (num \rightarrow num)] \{fun \{x : num\} x\}}]}
\end{verbatim}

This is a kind of polymorphic list definition

\textbf{Problem:} everything must be under a \texttt{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

```ocaml
{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?

\[
\{ \text{fun } \{ x : ? \} \{ x \ x \} \}\]

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

\[
\{ \text{fun } \{ x : (\text{forall alpha } (\text{alpha } \to \text{ alpha})) \}\}
\{ [\@ \ x \ (\text{num } \to \text{ num})] \ [\@ \ x \ \text{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 their right-hand side of a `with` or `rec` binding

• This works:

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

• This doesn’t:

```haskell
{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:

```plaintext
{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:

```plaintext
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 {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`:

```
{{[@ [@ 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?

```latex
{with \{f : ? \{fun \{v : ?\}
    \{fun \{g : (forall \alpha (\alpha \to \alpha))\}
    \{g v\}\}\}\}
{with \{g : ? \{fun \{x : ?\} x\}\}
{\{f 10\} g}\}}
```

One type application must be used inside \texttt{f}:

```latex
[tyfun \{\beta\} \{fun \{v : \beta\}
    \{fun \{g : (forall \alpha (\alpha \to \alpha))\}
    \{[@ g \beta] v\}\}\}]
```
Polymorphism and Values

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

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

but due to inference constraints,

\[
{\text{fun}} \{g : ?\}
{\text{if}} \{g \text{ false}\}
{\text{g 0}}
{\text{g 1}}\}
\]

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

\[(\text{forall alpha (forall beta (alpha \rightarrow beta))})\]

can be abbreviated

\[(\alpha \rightarrow \beta)\]

without loss of information