Lecture 26
Existential Types
Namespaces

It’s no fun to program in a language with a single, global namespace: C, FORTRAN, and PHP until depressingly recently.
Namespaces

It’s no fun to program in a language with a single, global namespace: C, FORTRAN, and PHP until depressingly recently.

Components of a large program have to worry about name collisions.

And components become tightly coupled: any component can use a name defined by any other.
Modularity

A module is a collection of named entities that are related.

Modules provide separate namespaces: different modules can use the same names without worrying about collisions.

Modules can:

- Choose which names to export
- Choose which names to keep hidden
- Hide implementation details
Existential Types

In the polymorphic $\lambda$-calculus, we introduced *universal* quantification for types.

\[ \tau ::= \cdots \mid X \mid \forall X. \tau \]
Existential Types

In the polymorphic \( \lambda \)-calculus, we introduced *universal* quantification for types.

\[
\tau ::= \cdots \mid X \mid \forall X. \tau
\]

If we have \( \forall \), why not \( \exists \)? What would *existential* type quantification do?

\[
\tau ::= \cdots \mid X \mid \exists X. \tau
\]
Existential Types

Together with records, existential types let us *hide* the implementation details of an interface.
Together with records, existential types let us *hide* the implementation details of an interface.

\[ \exists \text{Counter}.
\set{\text{new : Counter,}
\text{get : Counter \rightarrow int,}
\text{inc : Counter \rightarrow Counter}} \]
Existential Types

Together with records, existential types let us *hide* the implementation details of an interface.

\[
\exists \text{Counter}.
\{ \text{new : Counter},
      \text{get : Counter} \rightarrow \text{int},
      \text{inc : Counter} \rightarrow \text{Counter} \}
\]

Here, the *witness type* might be \text{int}:

\[
\{ \text{new : int},
      \text{get : int} \rightarrow \text{int},
      \text{inc : int} \rightarrow \text{int} \}
\]
Let’s extend our STLC with existential types:

$$\tau ::= \text{int}$$

$$\mid \tau_1 \to \tau_2$$

$$\mid \{ l_1 : \tau_1, \ldots, l_n : \tau_n \}$$

$$\mid \exists X. \tau$$

$$\mid X$$
We’ll tag the values of existential types with the witness type.
Syntax & Dynamic Semantics

We’ll tag the values of existential types with the witness type.

A value has type $\exists X. \tau$ is a pair $\{\tau', v\}$
where $v$ has type $\tau\{\tau'/X\}$. 
We’ll tag the values of existential types with the witness type.

A value has type $\exists X. \tau$ is a pair $\{\tau', v\}$
where $v$ has type $\tau\{\tau'/X\}$.

We’ll add new operations to construct and destruct these pairs:

$$\text{pack } \{\tau_1, e\} \text{ as } \exists X. \tau_2$$

$$\text{unpack } \{X, x\} = e_1 \text{ in } e_2$$
Syntax

\[\begin{align*}
e & ::= x \\
& \mid \lambda x: \tau. e \\
& \mid e_1 \ e_2 \\
& \mid n \\
& \mid e_1 + e_2 \\
& \mid \{ l_1 = e_1, \ldots, l_n = e_n \} \\
& \mid e. l \\
& \mid \text{pack} \ \{ \tau_1, e \} \ \text{as} \ \exists X. \ \tau_2 \\
& \mid \text{unpack} \ \{ X, x \} = e_1 \ \text{in} \ e_2
\end{align*}\]

\[\begin{align*}
v & ::= n \\
& \mid \lambda x: \tau. e \\
& \mid \{ l_1 = v_1, \ldots, l_n = v_n \} \\
& \mid \text{pack} \ \{ \tau_1, v \} \ \text{as} \ \exists X. \ \tau_2
\end{align*}\]
Dynamic Semantics

\[ E ::= \ldots \]

\[ \mid \text{pack } \{ \tau_1, E \} \text{ as } \exists X. \tau_2 \]

\[ \mid \text{unpack } \{ X, x \} = E \text{ in } e \]

\[
\text{unpack } \{ X, x \} = (\text{pack } \{ \tau_1, v \} \text{ as } \exists Y. \tau_2) \text{ in } e \rightarrow e\{v/x\}\{\tau_1/X\}
\]
\[ \Delta, \Gamma \vdash e : \tau_2\{\tau_1/X\} \]

\[ \Delta, \Gamma \vdash \text{pack } \{\tau_1, e\} \text{ as } \exists X. \tau_2 : \exists X. \tau_2 \]
\[
\begin{align*}
\Delta, \Gamma \vdash e : \tau_2 \{ \tau_1 / X \} \\
\Delta, \Gamma \vdash \text{pack} \{ \tau_1, e \} \text{ as } \exists X. \tau_2 : \exists X. \tau_2
\end{align*}
\]

\[
\begin{align*}
\Delta, \Gamma \vdash e_1 : \exists X. \tau_1 \\
\Delta \cup \{ X \}, \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \\
\Delta \vdash \tau_2 \text{ ok}
\end{align*}
\]

\[
\Delta, \Gamma \vdash \text{unpack} \{ X, x \} = e_1 \text{ in } e_2 : \tau_2
\]

The side condition \( \Delta \vdash \tau_2 \text{ ok} \) ensures that the existentially quantified type variable \( X \) does not appear free in \( \tau_2 \).
let counterADT =
    pack \{ int, 
        \{ new = 0, 
        get = \lambda i . int . i, 
        inc = \lambda i . int . i + 1 \} \} 

as 
\exists \text{Counter}.

\{ new : \text{Counter},
    get : \text{Counter} \rightarrow \text{int},
    inc : \text{Counter} \rightarrow \text{Counter} \}

in . . .
Here’s how to use the existential value `counterADT`:

```plaintext
unpack \{ T, c \} = counterADT in
let y = c.new in
c.get (c.inc (c.inc y))
```
We can define alternate, equivalent implementations of our counter...

```
let counterADT =
  pack {{x:int},
    { new = {x = 0},
      get = \r:{x:int}. r.x,
      inc = \r:{x:int}. r.x + 1 } }
  as
  \exists Counter.
    { new : Counter,
      get : Counter \rightarrow int,
      inc : Counter \rightarrow Counter }
```

in . . .
Existentials and Type Variables

In the typing rule for unpack, the side condition $\Delta \vdash \tau_2 \text{ ok}$ prevents type variables from “leaking out” of unpack expressions.
Existentials and Type Variables

In the typing rule for unpack, the side condition $\Delta \vdash \tau_2 \text{ ok}$ prevents type variables from “leaking out” of unpack expressions.

This rules out programs like this:

```
let m =
    pack {\texttt{int}, \{a = 5, f = \lambda x: \texttt{int}. x + 1\}} as \exists X. \{a:X, f:X \rightarrow X\}
in
    unpack \{T, x\} = m in x.f x.a
```

where the type of $x.f x.a$ is just $T$. 
We can encode existentials using universals!

The idea is to use a Church encoding where an existential value is a function that takes a type and then calls a continuation.
Encoding Existentials

We can encode existentials using universals!

The idea is to use a Church encoding where an existential value is a function that takes a type and then calls a continuation.

\[
\exists X. \tau \triangleq \forall Y. (\forall X. \tau \rightarrow Y) \rightarrow Y
\]

pack \(\{\tau_1, e\}\) as \(\exists X. \tau_2\) \(\triangleq \Lambda Y. \lambda f : (\forall X. \tau_2 \rightarrow Y). f[\tau_1] e\)

unpack \(\{X, x\} = e_1\) in \(e_2\) \(\triangleq e_1[\tau_2] (\Lambda X. \lambda x : \tau_1. e_2)\)

where \(e_1\) has type \(\exists X. \tau_1\) and \(e_2\) has type \(\tau_2\)