Lecture 28
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.
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.

$$\tau ::= \cdots \mid \alpha \mid \forall \alpha. \tau$$
Existential Types

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

$$\tau ::= \cdots \mid \alpha \mid \forall \alpha. \tau$$

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

$$\tau ::= \cdots \mid \alpha \mid \exists \alpha. \tau$$
Existential Types

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

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

$$\exists \text{Counter.}$$

\[
\begin{array}{l}
\{ \text{new : Counter}, \\
\text{get : Counter} \rightarrow \text{int}, \\
\text{inc : Counter} \rightarrow \text{Counter} \}
\end{array}
\]
Existential Types

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

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

Here, the witness type might be int:

\{ \text{new : int},
\text{get : int} \to \text{int},
\text{inc : int} \to \text{int} \}
Existential Types

Let’s extend our STLC with existential types:

\[ \tau ::= \text{int} \]

\[ | \tau_1 \rightarrow \tau_2 \]

\[ | \{ l_1 : \tau_1, \ldots, l_n : \tau_n \} \]

\[ | \exists \alpha. \tau \]

\[ | \alpha \]
We’ll tag the values of existential types with the witness type.
We’ll tag the values of existential types with the witness type.

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

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

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

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

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

\[ E ::= \ldots \]
\[ \quad | \text{pack } \{\tau_1, E\} \text{ as } \exists \alpha. \tau_2 \]
\[ \quad | \text{unpack } \{\alpha, x\} = E \text{ in } e \]

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

\[
\Delta, \Gamma \vdash e_1 : \exists \alpha. \tau_1 \quad \Delta \cup \{\alpha\}, \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \quad \Delta \vdash \tau_2 \text{ ok}
\]

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

The side condition \( \Delta \vdash \tau_2 \text{ ok} \) ensures that the existentially quantified type variable \( \alpha \) does not appear free in \( \tau_2 \).
Example

let \textit{counterADT} =
pack \{ \textbf{int},
{ new = 0,
get = \lambda i: \textbf{int}. i,
inc = \lambda i: \textbf{int}. i + 1 } \}

as
\exists \textbf{Counter}.
{ new : \textbf{Counter},
get : \textbf{Counter} \rightarrow \textbf{int},
inc : \textbf{Counter} \rightarrow \textbf{Counter} }
Here’s how to use the existential value `counterADT`:

```haskell
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...

```plaintext
let counterADT =
pack {{x:int},
  { new = {x = 0},
    get = \lambda r:{x:int}. r.x,
    inc = \lambda r:{x:int}. r.x + 1  }
}

as
∃Counter.
{ new : Counter,
  get : Counter → int,
  inc : Counter → 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:

\[
\text{let } m = \\
\quad \text{pack } \{ \text{int}, \{ a = 5, f = \lambda x: \text{int}. x + 1 \} \} \text{ as } \exists \alpha. \{ a: \alpha, f: \alpha \rightarrow \alpha \} \in \\
\quad \text{unpack } \{ T, x \} = m \text{ in } x.f x.a
\]

where the type of $x.f x.a$ is just $T$. 
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.
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.

\[
\exists \alpha. \tau \triangleq \forall \beta. (\forall \alpha. \tau \rightarrow \beta) \rightarrow \beta
\]

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

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

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