# CS 4110

# Programming Languages & Logics

## Lecture 26
## Existential Types

2 November 2016

# Announcements

- HW #7 due tonight at 11:59pm
- HW #8 out now
- After that, no homework until after Prelim II (and after Thanksgiving)

# 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.

# Existential Types

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

$$\begin{aligned} &\exists\, \textbf{Counter}.\\ &\quad \{\, \text{new} : \textbf{Counter},\\ &\quad\;\;\, \text{get} : \textbf{Counter} \to \textbf{int},\\ &\quad\;\;\, \text{inc} : \textbf{Counter} \to \textbf{Counter} \,\} \end{aligned}$$

# Existential Types

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

$$\begin{aligned}&\exists\, \textbf{Counter}.\\ &\quad \{\ \text{new} : \textbf{Counter},\\ &\quad\ \ \ \text{get} : \textbf{Counter} \to \textbf{int},\\ &\quad\ \ \ \text{inc} : \textbf{Counter} \to \textbf{Counter}\ \}\end{aligned}$$

Here, the *witness type* might be **int**:

$$\begin{aligned}&\{\ \text{new} : \textbf{int},\\ &\ \ \ \text{get} : \textbf{int} \to \textbf{int},\\ &\ \ \ \text{inc} : \textbf{int} \to \textbf{int}\ \}\end{aligned}$$

# Existential Types

Let's extend our STLC with existential types:

$$
\begin{aligned}
\tau ::= {} & \textbf{int} \\
\mid {} & \tau_1 \to \tau_2 \\
\mid {} & \{\, l_1 : \tau_1, \ldots, l_n : \tau_n \,\} \\
\mid {} & \exists X.\, \tau \\
\mid {} & X
\end{aligned}
$$

# Syntax & Dynamic Semantics

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 $\tau\{\tau'/X\}$.

# 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 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

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

$$
\begin{aligned}
v ::= {} & n \\
\mid {} & \lambda x{:}\tau.\, e \\
\mid {} & \{\, l_1 = v_1, \ldots, l_n = v_n \,\} \\
\mid {} & \text{pack } \{\tau_1, v\} \text{ as } \exists X.\, \tau_2
\end{aligned}
$$

# Dynamic Semantics

$$
\begin{aligned}
E ::= &\ \ldots \\
&\mid \text{pack } \{\tau_1, E\} \text{ as } \exists\, X.\ \tau_2 \\
&\mid \text{unpack } \{X, x\} = E \text{ in } e
\end{aligned}
$$

$$
\frac{}{\text{unpack } \{X, x\} = (\text{pack } \{\tau_1, v\} \text{ as } \exists\, Y.\ \tau_2) \text{ in } e \to e\{v/x\}\{\tau_1/X\}}
$$

# Static Semantics

$$\frac{\Delta, \Gamma \vdash e : \tau_2\{\tau_1/X\}}{\Delta, \Gamma \vdash \mathsf{pack}\ \{\tau_1, e\}\ \mathsf{as}\ \exists\, X.\, \tau_2 : \exists\, X.\, \tau_2}$$

# Static Semantics

$$\frac{\Delta, \Gamma \vdash e : \tau_2\{\tau_1/X\}}{\Delta, \Gamma \vdash \mathsf{pack}\ \{\tau_1, e\}\ \mathsf{as}\ \exists\, X.\ \tau_2 : \exists\, X.\ \tau_2}$$

$$\frac{\Delta, \Gamma \vdash e_1 : \exists\, X.\ \tau_1 \quad \Delta \cup \{X\}, \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \quad \Delta \vdash \tau_2\ \mathsf{ok}}{\Delta, \Gamma \vdash \mathsf{unpack}\ \{X, x\} = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

The side condition $\Delta \vdash \tau_2$ ok ensures that the existentially quantified type variable $X$ does not appear free in $\tau_2$.

# Example

$$
\begin{array}{l}
\text{let } \mathit{counterADT} = \\
\quad \text{pack}\ \{\ \textbf{int}, \\
\qquad\qquad \{\ \text{new} = 0, \\
\qquad\qquad\ \ \text{get} = \lambda \mathit{i} : \textbf{int}.\ \mathit{i}, \\
\qquad\qquad\ \ \text{inc} = \lambda \mathit{i} : \textbf{int}.\ \mathit{i} + 1\ \}\ \} \\
\quad \text{as} \\
\qquad \exists\ \textbf{Counter}. \\
\qquad\qquad \{\ \text{new} : \textbf{Counter}, \\
\qquad\qquad\ \ \text{get} : \textbf{Counter} \to \textbf{int}, \\
\qquad\qquad\ \ \text{inc} : \textbf{Counter} \to \textbf{Counter}\} \\
\text{in} \ldots
\end{array}
$$

# Example

Here's how to use the existential value *counterADT*:

$$\begin{array}{l} \text{unpack}\ \{T, c\} = \mathit{counterADT}\ \text{in} \\ \text{let}\ y = c.\text{new in} \\ c.\text{get}\ (c.\text{inc}\ (c.\text{inc}\ y)) \end{array}$$

# Representation Independence

We can define alternate, equivalent implementations of our counter...

$$
\begin{array}{l}
\text{let } \mathit{counterADT} = \\
\quad \text{pack } \{\{x\!:\!\mathbf{int}\}, \\
\qquad\qquad \{ \text{ new} = \{x = 0\}, \\
\qquad\qquad\quad \text{get} = \lambda r\!:\!\{x\!:\!\mathbf{int}\}.\, r.x, \\
\qquad\qquad\quad \text{inc} = \lambda r\!:\!\{x\!:\!\mathbf{int}\}.\, r.x + 1 \ \} \ \} \\
\quad \text{as} \\
\qquad \exists \mathbf{Counter}. \\
\qquad\qquad \{ \text{ new} : \mathbf{Counter}, \\
\qquad\qquad\quad \text{get} : \mathbf{Counter} \to \mathbf{int}, \\
\qquad\qquad\quad \text{inc} : \mathbf{Counter} \to \mathbf{Counter}\} \\
\text{in} \ldots
\end{array}
$$

# Existentials and Type Variables

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

# Existentials and Type Variables

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

This rules out programs like this:

$$
\begin{aligned}
&\text{let } m = \\
&\qquad \text{pack } \{\textbf{int}, \{a = 5, f = \lambda x\!:\!\textbf{int}.\ x + 1\}\} \text{ as } \exists\, X.\ \{a\!:\!X, f\!:\!X \to X\} \\
&\text{in} \\
&\text{unpack } \{T, x\} = m \text{ in } x.f\ x.a
\end{aligned}
$$

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.

# 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.

$$
\begin{aligned}
\exists X.\, \tau &\triangleq \forall Y.\, (\forall X.\, \tau \to Y) \to Y \\
\text{pack } \{\tau_1, e\} \text{ as } \exists X.\, \tau_2 &\triangleq \Lambda Y.\, \lambda f : (\forall X. \tau_2 \to Y).\, f\,[\tau_1]\, e \\
\text{unpack } \{X, x\} = e_1 \text{ in } e_2 &\triangleq e_1\,[\tau_2]\,(\Lambda X. \lambda x : \tau_1.\, e_2)
\end{aligned}
$$

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