CS 4110

Programming Languages & Logics

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.

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 λ-calculus, we introduced *universal* quantification for types.

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

Existential Types

In the polymorphic λ-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.

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

Existential Types

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

$$\begin{array}{l} \exists\, \mathbf{Counter}. \\ \quad \{\ \text{new} : \mathbf{Counter}, \\ \quad\ \ \ \text{get} : \mathbf{Counter} \to \mathbf{int}, \\ \quad\ \ \ \text{inc} : \mathbf{Counter} \to \mathbf{Counter}\ \} \end{array}$$

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

$$\begin{array}{l} \{\ \text{new} : \mathbf{int}, \\ \ \ \ \text{get} : \mathbf{int} \to \mathbf{int}, \\ \ \ \ \text{inc} : \mathbf{int} \to \mathbf{int}\ \} \end{array}$$

Existential Types

Let’s extend our STLC with existential types:

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

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

Syntax & Dynamic Semantics

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:

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

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

Syntax

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

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

Dynamic Semantics

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

$$\frac{}{\text{unpack } \{\alpha, x\} = (\text{pack } \{\tau_1, v\} \text{ as } \exists\, \beta.\, \tau_2) \text{ in } e \rightarrow e\{v/x\}\{\tau_1/\alpha\}}$$

Static Semantics

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

Static Semantics

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

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

The side condition $\Delta \vdash \tau_2$ ok ensures that the existentially quantified type variable α does not appear free in τ_2.

Example

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

Example

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

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

Representation Independence

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

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

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:

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

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.

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

where e_1 has type $\exists\alpha.\tau_1$ and e_2 has type τ_2