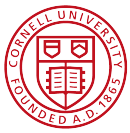
CS 4110 – Programming Languages and Logics
Lecture #26: Existential Types



1 Modules

Simple languages, such as C and FORTRAN, often have a single global namespace. This causes problems in large programs due to name collisions—i.e., two different programmers (or pieces of code) using the same name for different purposes—are likely. In addition, it often leads to situations where multiple components of a program are more tightly coupled, since one component may use a name defined by the other.

Modular programming addresses these issues. A *module* is a collection of named entities that are related to each other in some way. Modules provide separate namespaces: different modules have different name spaces, and so can freely use names without worrying about name collisions.

Typically, a module can choose what names and entities to export (i.e., which names to allow to be used outside of the module), and what to keep hidden. The exported entities are declared in an *interface*, and the interface typically does not export details of the implementation. This means that different modules can implement the same interface in different ways. Also, by hiding the details of module implementation, and preventing access to these details except through the exported interface, programmers of modules can be confident that code invariants are not broken.

Packages in Java are a form of modules. A package provides a separate namespace (we can have a class called `Foo` in package `p1` and package `p2` without any conflicts). A package can hide details of its implementation by using private and package-level visibility.

How do we access the names exported by a module? Given a module m that exports an entity names x, common syntax for accessing x is $m.x$. Many languages also provide a mechanism to use all exported names of a module using shorter notation—e.g., "Open m", or "import m", or "using m".

2 Existential types

In this section, we will extend the simply-typed lambda calculus with *existential types* (and records). An existential type is written $\exists X.\ \tau$, where type variable X may occur in τ. If a value has type $\exists X.\ \tau$, it means that it is a pair $\{\tau', v\}$ of a type τ' and a value v, such that v has type $\tau\{\tau'/X\}$.

We introduce a language construct to create existential values, and a construct to use existential values. The syntax of the new language is given by the following grammar.

$$
\begin{aligned}
e &::= x \mid \lambda x{:}\tau.\, e \mid e_1\ e_2 \mid n \mid e_1 + e_2 \\
&\quad \mid \{\ l_1 = e_1, \ldots, l_n = e_n\ \} \mid e.l \\
&\quad \mid \mathsf{pack}\ \{\tau_1, e\}\ \mathsf{as}\ \exists X.\ \tau_2 \mid \mathsf{unpack}\ \{X, x\} = e_1\ \mathsf{in}\ e_2 \\
v &::= n \mid \lambda x{:}\tau.\, e \mid \{\ l_1 = v_1, \ldots, l_n = v_n\ \} \mid \mathsf{pack}\ \{\tau_1, v\}\ \mathsf{as}\ \exists X.\ \tau_2 \\
\tau &::= \mathbf{int} \mid \tau_1 \to \tau_2 \mid \{\ l_1{:}\tau_1, \ldots, l_n{:}\tau_n\ \} \mid \exists X.\ \tau
\end{aligned}
$$

Note that in this grammar, we annotate existential values with their existential type. The construct to create an existential value, pack $\{\tau_1, e\}$ as $\exists X.\ \tau_2$, is often called *packing*, and the construct to use an existential value is called *unpacking*. Before we present the operational semantics and typing rules, let's see an example to get an intuition for packing and unpacking.

Here we create an existential value that implements a counter, without revealing details of its implementation.

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

The abstract type name is **Counter**, and its concrete representation is **int**. The type of the variable $counterADT$ is $\exists \mathbf{Counter}.\ \{\ \mathsf{new} : \mathbf{Counter}, \mathsf{get} : \mathbf{Counter} \to \mathbf{int}, \mathsf{inc} : \mathbf{Counter} \to \mathbf{Counter}\ \}$. We can use the existential value $counterADT$ as follows.

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

Note that we annotate the pack construct with the existential type. That is, we explicitly state the type

$$\exists \mathbf{Counter}.\ \{\mathsf{new} : \mathbf{Counter}, \mathsf{get} : \mathbf{Counter} \to \mathbf{int}, \mathsf{inc} : \mathbf{Counter} \to \mathbf{Counter}\}.$$

Why do we do this? Without this annotation, we would not know which occurrences of the witness type are intended to be replaced with the type variable, and which are intended to be left as the witness type.

In the counter example above, the type of expressions $\lambda i : \mathbf{int}.\ i$ and $\lambda i : \mathbf{int}.\ i + 1$ are both $\mathbf{int} \to \mathbf{int}$, but one is the implementation of get, of type $\mathbf{Counter} \to \mathbf{int}$ and the other is the implementation of inc, of type $\mathbf{Counter} \to \mathbf{Counter}$.

We now define the operational semantics for existentials. We add two new evaluation contexts, and one evaluation rule for unpacking an existential value.

$$E ::= \cdots \mid \mathsf{pack}\ \{\tau_1, E\}\ \mathsf{as}\ \exists X.\ \tau_2 \mid \mathsf{unpack}\ \{X, x\} = E\ \mathsf{in}\ e$$

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

The typing rules ensure that existential values are used correctly.

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

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

Note that in the typing rule for unpack, the side condition $\Delta \vdash \tau_2$ ok ensures that the existentially quantified type variable X does *not* appear free in τ_2. This rules out programs such as,

$$\begin{array}{l}\mathsf{let}\ m = \\ \quad \mathsf{pack}\ \{\mathbf{int}, \{a = 5, f = \lambda x{:}\mathbf{int}.x + 1\}\}\ \mathsf{as}\ \exists X.\ \{a{:}X, f{:}X \to X\} \\ \mathsf{in} \\ \mathsf{unpack}\ \{X, x\} = m\ \mathsf{in}\ x.f\ x.a\end{array}$$

where the type of $(x.f\ x.a)$ has X free.

3 Church Encoding

It turns out that we can encode existentials in System F! The idea is to use a Church encoding, where an existential value is a function that takes a type and then calls the continuation

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

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

For further details see Pierce, Chapter 24.