Existential Types and Modules

Lecture 18 Thursday, April 4, 2013

---

1 Modules

Some languages, including C and FORTRAN, have a single global namespace. This can cause problems. With large programs, a name collision is likely, that is, two different programmers (or pieces of code) attempting to use the same name for different purposes. Also, components of a program may be more tightly coupled, since two components are coupled simply by one using 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/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 $m.x$. However, if we are writing a program that uses module m heavily, it is convenient to have a shorter way to refer to names. Many languages provide a mechanism to use all exported names of a module using shorter notation. For example "with m do e" allows the names exported by m to be used in e. You may have seen languages with syntax like "Open m", or "import m", or "using m".

We will first introduce *existential types*, a type mechanism that will help us understand modules. We will then use existential types to help us understand a simple module system.

2 Existential types

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

Thinking about the Curry-Howard isomorphism may provide some intuition for existential types. As the notation and name suggest, the logical formula that corresponds to an existential type $\exists X.\ \tau$ is an existential formula $\exists X.\ \phi$, where X may occur in ϕ. In intuitionist logic, would would it mean for the statement "there exists some X such that ϕ is true" to be true? In intuitionist logic, a statement is true only if there is a proof for it. To prove "there exists some X such that ϕ is true" we must actually provide a *witness* ψ, an entity that is a suitable replacement for X, and also, a proof that ϕ is true when we replace X with witness ψ.

A value $\{\tau', v\}$ of type $\exists X.\ \tau$ exactly corresponds to a proof of an existential statement: type τ' is the witness type, and v is a value with 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.

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

Note that in this grammar, we annotate existential values with their existential type. The construct to create an existential value, $\mathsf{pack}\ \{\tau_1, e\}\ \mathsf{as}\ \exists X.\ \tau_2$, 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 some examples to get an intuition for packing and unpacking.

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

mentation.

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

The abstract type name is **Counter**, and its concrete representation is **int**. The type of the variable $counterADT$ is $\exists$**Counter**. $\{$ new : **Counter**, get : **Counter** $\to$ **int**, inc : **Counter** $\to$ **Counter** $\}$.

We can use the existential value $counterADT$ as follows.

$$\mathsf{unpack}\ \{C, x\} = counterADT\ \mathsf{in}\ \mathsf{let}\ y\!:\!C = x.\mathsf{new}\ \mathsf{in}\ x.\mathsf{get}\ (x.\mathsf{inc}\ (x.\mathsf{inc}\ y))$$

Note that we annotate the pack construct with the existential type. That is, we explicitly state the type $\exists$**Counter**. .... Why is 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 **int** $\to$ **int**, but one is the implementation of get, of type **Counter** $\to$ **int** and the other is the implementation of inc, of type **Counter** $\to$ **Counter**.

We now define the operational semantics. 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 new typing rules make sure that existential values are used correctly. Note that code using an existential value (e in unpack $\{X, x\} = e_1$ in e_2) does not know the witness type of the existential value of type $\exists X.\ \tau_1$.

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

$$\frac{\Delta \cup \{X\} \vdash \tau \text{ ok}}{\Delta \vdash \exists X.\, \tau \text{ ok}}$$

Note that we define well-formedness of existential types, similar to well-formedness of universal types. In the typing rule for unpack $\{X, x\} = e_1$ in e_2, why do we need the premise $\Delta \vdash \tau_2$ ok?

3 A simple module mechanism

Let's see a simple module mechanism.

$$\begin{aligned}
e ::= {} & x \mid \lambda x{:}\tau.\, e \mid e_1\ e_2 \mid n \mid e_1 + e_2 \\
& \mid \text{module implements } \tau \{ \text{ type } X_1 = \tau_1, \ldots, X_m = \tau_m;\ \text{val } x_1 = e_1, \ldots, x_n = e_n \} \\
& \mid e.x \mid \text{open } e_1 \text{ in } e_2 \\
v ::= {} & n \mid \lambda x{:}\tau.\, e \mid \text{module implements } \tau \{ \text{ type } X_1 = \tau_1, \ldots, X_m = \tau_m;\ \text{val } x_1 = v_1, \ldots, x_n = v_n \} \\
\tau ::= {} & \textbf{int} \mid x{:}\tau_1 \to \tau_2 \mid X \mid \textbf{interface}\ \{\ \textbf{type}\ X_1, \ldots, X_m;\ \textbf{val}\ x_1{:}\tau_1, \ldots, x_n{:}\tau_n\ \} \mid e.X
\end{aligned}$$

This module system provides a way to define modules module implements τ { ... }. A module definition declares a number of type variables $X_1, \ldots, X_m$, along with witness types for these type variables, and also associates values with names $x_1, \ldots, x_n$. A module can export the type variables and (a subset of) the named values, by declaring its interface using the interface type **interface** { ... }.

We also provide a way to access the values associated with names in a module: in the expression $e.x$, expression e is intended to evaluate to a module, and $e.x$ will evaluate to whatever value the module associates with the name x. Similarly, type variables declared in a module can be referred to using the dependent type $e.X$, where e is intended to denote a module.

For convenience, we provide an easier way to access names exported by a module. In the expression open e_1 in e_2, expression e_1 evaluates to a module, and in expression e_2, all names exported by e_1 are in scope, meaning that e_2 can refer to a name x without needing to prefix it with the module, $e_1.x$.

Because a module may export only a subset of the names it defines, the module allow information hiding: details of the module implementation are not available to clients of the module.

Let's consider our example for a simple counter in our new module language.

$$
\begin{array}{l}
\mathsf{let}\ cm\ = \\
\quad \mathsf{module} \\
\quad\quad \mathsf{implements}\ \textbf{interface}\ \{\ \textbf{type}\ \textbf{CTR};\ \mathsf{val\ new}:\textbf{CTR}, \mathsf{get}:\textbf{CTR}\to\textbf{int}, \mathsf{inc}:\textbf{CTR}\to\textbf{CTR}\ \} \\
\quad \{\ \mathsf{type}\ \textbf{CTR}=\textbf{int};\ \mathsf{val\ new}=1, \mathsf{get}=\lambda i:\textbf{CTR}.\, i, \mathsf{inc}=\lambda i{:}CTR.\, i+1\ \} \\
\mathsf{in} \\
\quad \mathsf{let}\ c:cm.\textbf{CTR}=cm.\mathsf{new}\ \mathsf{in}\ cm.\mathsf{get}\ (cm.\mathsf{inc}\ c)
\end{array}
$$

In the code above, we are accessing the names exported by the module cm using the $e.x$ notation. We can use the open construct to make this a little more convenient.

$$
\begin{array}{l}
\mathsf{let}\ cm\ = \\
\quad \ldots \\
\mathsf{in\ open}\ cm\ \mathsf{in} \\
\quad \mathsf{let}\ c:\textbf{CTR}=\mathsf{new}\ \mathsf{in}\ \mathsf{get}\ (\mathsf{inc}\ c)
\end{array}
$$

3.1 Operational semantics

We define a large-step operational semantics for the language. Because interesting things are going on with names and scope, (due to the open construct), we use an environment semantics. An environment ρ maps variables to values. The operational semantics maps $\langle e, \rho\rangle$, an expression e and environment ρ to a value v.

The evaluation of functions deserves special mention. Configuration $\langle \lambda x:\tau.\, e, \rho\rangle$ is a function $\lambda x:\tau.\, e$, defined in environment ρ, and evaluates to the *closure* $(\lambda x:\tau.\, e, \rho)$. A closure consists of code along with values for all free variables that appear in the code. Note than when we apply a function, we evaluate the function body using the environment from the closure (i.e., the lexical environment), as opposed to the environment in use at the function application (the dynamic environment).

The evaluation of a module simply evaluates the expressions that names in the module are mapped to. Accessing a module name $e.x$ just evaluates to the value that the module e associated with the name x.

Opening a module, open e_1 in e_2, evaluates e_1 to a module, and then adds all of the names that module exports to the environment in which e_2 is evaluated.

$$
\frac{}{\langle x, \rho\rangle \Downarrow \rho(x)} \qquad \frac{}{\langle n, \rho\rangle \Downarrow n} \qquad \frac{\langle e_1, \rho\rangle \Downarrow n_1 \quad \langle e_2, \rho\rangle \Downarrow n_2}{\langle e_1 + e_2, \rho\rangle \Downarrow n}\ n = n_1 + n_2
$$

$$\frac{}{\langle \lambda x{:}\tau.\, e, \rho\rangle \Downarrow (\lambda x{:}\tau.\, e, \rho)} \qquad \frac{\langle e_1, \rho\rangle \Downarrow (\lambda x{:}\tau.\, e, \rho_{lex}) \quad \langle e_2, \rho\rangle \Downarrow v_2 \quad \langle e, \rho_{lex}[x \mapsto v_2]\rangle \Downarrow v}{\langle e_1\ e_2, \rho\rangle \Downarrow v}$$

$$\frac{\langle e_i, \rho[x_1 \mapsto v_1, \ldots, x_{i-1} \mapsto v_{i-1}]\rangle \Downarrow v_i}{\begin{array}{l}\langle \textsf{module implements } \tau \textsf{ \{ type } X_1 = \tau_1, \ldots, X_m = \tau_m\textsf{; val } x_1 = e_1, \ldots, x_n = e_n \textsf{ \}}, \rho\rangle \Downarrow \\ \qquad \textsf{module implements } \tau \textsf{ \{ type } X_1 = \tau_1, \ldots, X_m = \tau_m\textsf{; val } x_1 = v_1, \ldots, x_n = v_n \textsf{ \}}\end{array}}$$

$$\frac{\langle e, \rho\rangle \Downarrow \textsf{module implements } \tau \textsf{ \{ type } X_1 = \tau_1, \ldots, X_m = \tau_m\textsf{; val } x_1 = v_1, \ldots, x_n = v_n \textsf{ \}}}{\langle e.x_i, \rho\rangle \Downarrow v_i}$$

$$\frac{\begin{array}{r}\langle e_1, \rho\rangle \Downarrow \textsf{module implements } \textbf{interface } \textsf{\{ } \textbf{type } X_1, \ldots, X_m\textsf{; } \textbf{val } y_1{:}\tau_1, \ldots, y_o{:}\tau_o \textsf{ \}} \\ \textsf{\{ type } X_1 = \tau_1, \ldots, X_m = \tau_m\textsf{; val } x_1 = v_1, \ldots, x_n = v_n \textsf{ \}} \\ \forall j, k.\ y_j = x_k \Rightarrow w_j = v_k \\ \langle e_2, \rho[y_1 \mapsto w_1, \ldots, y_o \mapsto w_o]\rangle \Downarrow v\end{array}}{\langle \textsf{open } e_1 \textsf{ in } e_2, \rho\rangle \Downarrow v}$$

3.2 Typing rules

We provide, but do not discuss, typing rules for the modules.

$$\frac{}{\Delta, \Gamma \vdash x{:}\Gamma(x)} \qquad \frac{}{\Delta, \Gamma \vdash n{:}\textbf{int}} \qquad \frac{\Delta, \Gamma \vdash e_1{:}\textbf{int} \quad \Delta, \Gamma \vdash e_2{:}\textbf{int}}{\Delta, \Gamma \vdash e_1 + e_2{:}\textbf{int}}$$

$$\frac{\Delta, \Gamma, x{:}\tau \vdash e{:}\tau_2}{\Delta, \Gamma \vdash \lambda x{:}\tau.\, e{:}x{:}\tau_1 \to \tau_2} \qquad \frac{\Delta, \Gamma \vdash e_1{:}x{:}\tau_1 \to \tau_2 \quad \Delta, \Gamma \vdash e_2{:}\tau_1}{\Delta, \Gamma \vdash e_1\ e_2{:}\tau_2\{e_2/x\}}$$

$$\frac{\begin{array}{c}\tau \equiv \textbf{interface } \textsf{\{ } \textbf{type } X_1, \ldots, X_m\textsf{; } \textbf{val } y_1{:}\tau_1'', \ldots, y_o{:}\tau_o'' \textsf{ \}} \\ \forall i. \qquad \Delta, \Gamma[x_1 \mapsto \tau_1', \ldots, x_{i-1} \mapsto \tau_{i-1}'] \vdash e_i\{\tau_1/X_1\}\ldots\{\tau_m/X_m\}{:}\tau_i' \\ \forall j, k.\ x_j = y_k \Rightarrow \tau_k''\{\tau_1/X_1\}\ldots\{\tau_m/X_m\} = \tau_j'\end{array}}{\Delta, \Gamma \vdash \textsf{module implements } \tau \textsf{ \{ type } X_1 = \tau_1, \ldots, X_m = \tau_m\textsf{; val } x_1 = e_1, \ldots, x_n = e_n \textsf{ \}}{:}\tau}$$

$$\frac{\Delta, \Gamma \vdash e{:}\textbf{interface } \textsf{\{ } \textbf{type } X_1, \ldots, X_m\textsf{; } \textbf{val } x_1{:}\tau_1, \ldots, x_n{:}\tau_n \textsf{ \}}}{\Delta, \Gamma \vdash e.x_i{:}\tau_i\{e.X_1/X_1\}\ldots\{e.X_m/X_m\}}$$

$$\frac{\Delta, \Gamma \vdash e_1 : \textbf{interface}\ \{\ \textbf{type}\ X_1, \ldots, X_m;\ \textbf{val}\ x_1 : \tau_1, \ldots, x_n : \tau_n\ \} \qquad \Delta \cup \{X_1, \ldots, X_m\}, \Gamma[x_1 : \tau_1, \ldots, x_n : \tau_n] \vdash e_2 : \tau}{\Delta, \Gamma \vdash \mathsf{open}\ e_1\ \mathsf{in}\ e_2 : \tau}$$

$$\frac{}{\Delta, \Gamma \vdash X\ \mathsf{ok}}\ X \in \Delta \qquad \frac{}{\Delta, \Gamma \vdash \textbf{int}\ \mathsf{ok}} \qquad \frac{\Delta, \Gamma \vdash \tau_1\ \mathsf{ok} \quad \Delta, \Gamma[x : \tau_1] \vdash \tau_2\ \mathsf{ok}}{\Delta, \Gamma \vdash x : \tau_1 \to \tau_2\ \mathsf{ok}}$$

$$\frac{\Delta \cup \{X_1, \ldots, X_m\}, \Gamma[x_1 : \tau_1, \ldots, x_{i-1} : \tau_{i-1}] \vdash \tau_i\ \mathsf{ok}}{\Delta, \Gamma \vdash \textbf{interface}\ \{\ \textbf{type}\ X_1, \ldots, X_m;\ \textbf{val}\ x_1 : \tau_1, \ldots, x_n : \tau_n\ \}\ \mathsf{ok}}$$

$$\frac{\Delta, \Gamma \vdash e : \textbf{interface}\ \{\ \textbf{type}\ X_1, \ldots, X_m;\ \textbf{val}\ x_1 : \tau_1, \ldots, x_n : \tau_n\ \}}{\Delta, \Gamma \vdash e.X_i\ \mathsf{ok}}$$