CS611 Lecture 37 Existential Types and Modules 22 November 2000

Scribe: Jed Liu, Brandon Bray Lecturer: Andrew Myers

Existential types $\exists X.\sigma$ introduced in class provide a convenient way of modeling objects and modules with private members. These notes will discuss the following topics:

1. Type rules for creating and eliminating existential types
2. Operational semantics of existential types
3. Modeling objects and modules with existential types
4. Strong existential types
5. Dependent module types
6. First-class vs. second-class modules

1 Existential Types (continued from last lecture)

Last lecture, existential types were introduced. To recap, existential types hide a part of a type. As will be seen later, this makes existential types almost perfect for modeling language constructs such as objects (with only static members) and modules that provide information hiding through abstract types.

If u has type $\exists X.\sigma$, then the value u is a pair $[\tau, v]$ where τ is the hidden part of u. We write $[\tau, v] : \exists X.\sigma$ if v has type $\sigma\{\tau/X\}$.

1.1 Creation and elimination

Existential types are created through the pack rule:

$$\frac{\Delta; \Gamma \vdash e\{\tau/X\} : \sigma\{\tau/X\}}{\Delta; \Gamma \vdash \mathsf{pack}[X = \tau, e] : \exists X.\sigma}$$

and existential values are used via unpack. If e_1 evaluates to an existential value $[\tau, v] : \exists X.\sigma_1$, the contents of e_1 can be exposed within another expression e_2 by binding τ to T and v to x in e_2 by writing "$\mathsf{unpack}\ e_1\ \mathsf{as}\ [T, x]\ \mathsf{in}\ e_2$".

$$\frac{\Delta; \Gamma \vdash e_1 : \exists X.\sigma_1 \qquad \Delta; \Gamma, v : \sigma_1\{T/X\} \vdash e_2 : \sigma_2 \qquad T \notin \Delta \qquad \Delta \vdash \sigma_2}{\Delta; \Gamma \vdash \mathsf{unpack}\ e_1\ \mathsf{as}\ [T, v]\ \mathsf{in}\ e_2 : \sigma_2}$$

Note that T must be "fresh" (i.e., $T \notin \Delta$) and cannot escape from unpack ($\Delta \vdash \sigma_2$). For example, if the expression $\mathsf{unpack}\ p\ \mathsf{as}\ [T, x]\ \mathsf{in}\ x$ were legal, then a value of type T escapes into the surrounding context where its type has no meaning.

For an intuitive explanation, suppose we have

$$\mathsf{pack}[X = \mathsf{int}, \langle 5, \lambda x \in \mathsf{int}\,.\,\#t\rangle] : \exists X.X * (X \rightarrow \mathsf{bool}).$$

This gives us the tuple $[\mathsf{int}, \langle 5, \lambda x \in \mathsf{int}\,.\,\#t\rangle]$ that hides away the fact that $X = \mathsf{int}$ from the outside world. Now, if we let p be the pack expression above and then unpack it as $[X, v]$ in $(\pi_2 v)(\pi_1 v)$, we get

$$\mathsf{unpack}\ p\ \mathsf{as}\ [X, v]\ \mathsf{in}\ (\pi_2 v)(\pi_1 v) : \mathsf{bool},$$

which takes the $\langle 5, \lambda x \in \mathsf{int}\,.\,\#t\rangle$ value part of the existential and applies the int in the first component to the $\mathsf{int} \rightarrow \mathsf{bool}$ function in the second component, returning $\#t$.

1.2 Operational semantics

Like fold and unfold, pack and unpack are in a realistic implementation purely compile-time operations that change our view of the type of an expression: they have no computational significance. However, for the purpose of showing the soundness of the type system, we define the operational semantics so that unpack can only be applied to the result of a pack:

$$\mathsf{unpack}\ (\mathsf{pack}[X = \tau, e])\ \mathsf{as}\ [X, x]\ \mathsf{in}\ e' \rightarrow e'\{\tau/X, e/x\}.$$

We also extend the evaluation context with pack and unpack:

$$C ::= \ldots \quad | \quad \mathsf{pack}[X = \tau, C] \quad | \quad \mathsf{unpack}\ C\ \mathsf{as}\ [X, x]\ \mathsf{in}\ e$$

as well as the valid values with existential values:

$$v ::= \ldots \quad | \quad \mathsf{pack}[X = \tau, v]$$

2 Modeling Objects and Modules

2.1 Modeling objects

We can combine existentials with recursion to model the encapsulation and information-hiding features of objects. That is, objects without subtyping or inheritance. To do this, we translate objects into recursively-defined existential record types. For example, consider the following pseudo-Java code.

```
class charlist {
    public charlist cons(char, charlist);
    public char car();
    public charlist cdr();

    private char head;
    private charlist tail;
}
```

This code would be translated into our 611 language as:

$$\begin{aligned}
\mathsf{charlist} = \mu T.\exists P.\{ & \\
& \mathsf{cons} : \mathsf{char} * T \rightarrow T, \\
& \mathsf{car} : \mathsf{unit} \rightarrow \mathsf{char}, \\
& \mathsf{cdr} : \mathsf{unit} \rightarrow T, \\
& \mathsf{fields} : P \\
\} &
\end{aligned}$$

2.2 Modeling modules

We'd like to use existential types for modeling modules (such as they would appear in ML in the form of structures). Modules are a mechanism for encapsulating values and are themselves a value. A signature in ML is effectively the type of a module. Modules are convenient for writing abstract data types. Unfortunately, the existential types discussed thus far make it impossible to model modules correctly. This is because we have been discussing *weak* existential types, so-called because the abstract type denoted by the type variable cannot escape the uses of unpack. This limitation places severe limitations on using modules to create abstract data types. The code above might be written as a module in the following way:

```
signature CHARLIST =
sig
  type list

  val empty : list

  val cons  : char * list -> list
  val car   : list -> char
  val cdr   : list -> list
end
```

Unfortunately, outside of the module, the type `list` has no meaning. One way around this problem is to wrap any expression that might need to use the abstract type of the module with all the unpacked elements of the module:

$$\begin{array}{l} \mathsf{unpack}\ cons\ \mathsf{as}\ [T, kons]\ \mathsf{in} \\ \quad \mathsf{unpack}\ car\ \mathsf{as}\ [T, kar]\ \mathsf{in} \\ \quad\quad \mathsf{unpack}\ cdr\ \mathsf{as}\ [T, kdr]\ \mathsf{in} \\ \quad\quad\quad e \end{array}$$

We can think of the unpack operations as being like Java import statements that bring an external module into scope so it can be accessed. However, it suggests that the *weak* existential types we have been looking at are too limited to express modules as in ML or Modula-3.

3 Strong Existential Types

The method of modeling modules described thus far obviously lacks some necessary expressive power. Instead, let's try adding module types and expressions to our language and defining the semantics that we'd expect modules to have. Then we'll look at how we can extend existential types to have this power. Here is the extended language:

$$\begin{array}{l} \tau ::= \ldots \quad | \quad \{\mathsf{type}\ X_1, \ldots, X_m; \mathsf{val}\ l_1 : \tau_1, \ldots, l_n : \tau_n\} \quad | \quad e.X \\ e ::= \ldots \quad | \quad \{\mathsf{type}\ X_1 = \tau_1, \ldots, X_m = \tau_m; \mathsf{val}\ l_1 = e_1, \ldots, l_n = e_n\} \quad | \quad e.l \end{array}$$

Note that we must be careful now that expressions may be used as part of type expressions, in the type e.X, because the meaning of this type depends on the value e, which can change at run time. It would be unacceptable to have to do type-checking (i.e., decide whether the program is well-formed) at run-time. This module construct is much like ML or Modula-3 modules, or Java if we only use static methods and fields are used everywhere.

Modules look like records but they have the added capability of defining new types. Existential types only allowed one type to be hidden at a time; modules allow many types to be abstracted. The selection expression $e.l$ is used in place of unpack, and lastly the types X_i can be used outside the module once they are qualified by the module value. It is this ability to refer to the types X_i that gives new power.

We can translate a program using these module types into a language with a stronger version of existential types. We will refer to these types as *strong existential types* (they are also called *strong sum types*, as in Mitchell) Strong existential types extend weak existential types with two new kinds of expressions. The new type expression $e.T$ denotes the type that is hidden inside a packed value e, and $e.V$ denotes the corresponding value. The rule for pack remains unchanged from before; however, unpack as [c,h] in anges in an important way:

$$\frac{\Delta; \Gamma \vdash e_1 : \exists X.\sigma_1 \qquad \Delta; \Gamma, v : \sigma_1\{T/X\} \vdash e_2 : \sigma_2 \qquad T \notin \Delta}{\Delta; \Gamma \vdash \mathsf{unpack}\ e_1\ \mathsf{as}\ [T, v]\ \mathsf{in}\ e_2 : \sigma_2\{e_1.T/T}$$

The restriction $\Delta \vdash \sigma_2$ is no longer there, because that the hidden type can now be talked about in the surrounding context. However, in that context we need to call the type $e_1.T$ rather than T or X, because every existential value of type $\exists X.\sigma_1$ potentially has a different type hidden inside it. We do need to add another rule to the static semantics so that we may reason about $e.V$:

$$\frac{\Delta; \Gamma \vdash e : \exists X.\sigma \qquad X \notin \Delta}{\Delta; \Gamma \vdash e.V : \sigma\{e.T/X\}}$$

4 Dependent Module Types

Modules are values, and recall that modules can contain types. These types are referred to as dependent module types. Dependent, because the types depend on a value: the module. In order to build a sound type system with types like $e.T$, we need a very simple rule to decide when $e_1.T$ and $e_2.T$ are the same type. As a starting point, we might require syntactic equality between e_1 and e_2. But this is not enough, as expressions on the left hand side of the selector ("dot") can have different meanings in different contexts. Typically, such expressions are restricted to module names. Languages restrict the use of module values to prevent unsoundness in the type system. Specifically, any expression $x.T$ cannot escape the scope of x; also, module names cannot be shadowed. Our rules for introducing new variables into the type context must prevent this from happening.

5 First Class vs. Second Class Modules

Lastly, we have noted that modules are in fact values. In many languages, such as ML they are treated as second-class values, meaning they cannot be passed around or created at run time.

Programming in a language with first-class module values is a bit awkward because whenever we want to pass a value of type `e.V`, we also need to pass along `e` with it. In the next lecture, we will talk about object-oriented languages. We can think of objects as carrying around their implementations (module values) along with them implicitly, which is usually more convenient.