CS611 Lecture 35 Existential types 11/23/05

Scribe: based on notes by H. Chao, D. Yeh, A. Montalban, Y. Zhang
edited by Sam Arbesman and Maksim Orlovich Lecturer: Andrew Myers

1 Modeling objects with recursive types

With our typing model complete, we'll now begin to explore objects and classes. Consider the following Java implementation of integer sets as binary search trees:

```
class intset {
    intset union(intset S) { ... }
    boolean contains(int n) {
        if (n == value) return true;
        if (n < value) then return (left != null) && left.contains(n);
        else return (right != null) && right.contains(n);
    }
    int value;
    intset left,right;
}
```

This code snippet is implicitly using self-reference (the values left and right are actually this.left and this.right). With recursive types and records we can approximate this in the typed lambda calculus. First, there is a type intset being declared:

$$\mathsf{intset} = \mu S.(\{\mathsf{union} : S \to S, \mathsf{contains} : \mathsf{int} \to \mathsf{bool}, \mathsf{value} : \mathsf{int}, \mathsf{left} : S, \mathsf{right} : S\} + 1)$$

Note that we need recursive types to represent the fact that union returns an object of the same type.

And we can construct "objects" of this type:

```
let s = inl fold_intset (rec this: { union : intset → intset, ... }. // the unfolding of intset
    { union = λs' : intset. ...
      contains = λn : int. if n = this.value then true
          else if m < this.value then case this.left of
                 λu:1. false
                 λs':intset. ((unfold s').contains) n
               else ...
    }
```

This whole expression has type intset and will behave mostly like an object. There are a couple of ways in which this falls short of what Java objects provide: first, there is no inheritance and we'll have trouble extending this code to support inheritance. Second, the internals of the class are fully exposed to any other objects or functions that might use it. We need some way of providing a restricted interface to our objects and classes. It is this second problem we will talk about now.

2 Encapsulation/Information Hiding

While we can encode objects currently, we are missing one of the key concepts of object-oriented programming: information hiding. Information hiding is important since it both provides an abstraction barrier as well as allowing for division of labor and assignment of blame. We can indeed encode information hiding with the use of existential types, which correspond closely to the logical equivalent.

The idea is that we can hide part of a type τ and replace it with a type variable X. We write $\exists X.\tau$ to represent this type, where X may be mentioned inside τ. But because this type doesn't say what X is, no code receiving a value of this type can make use of knowledge of the hidden part of this type.

For example, in the intset example we would write a type like this:

```
∃X. { union:     S  →  S
      contains: int → bool
      private:  X }
```

We can think of values of this type as being a kind of pair consisting of a type and a value. That is, the pair $[\tau, v] : \exists X.\sigma$ where $v : \sigma\{\tau/X\}$. To manipulate these values, we introduce two new operators, pack (the introduction form) and unpack (the elimination form).

These two forms look, and type-check, as follows:

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

$$\frac{\Delta;\Gamma \vdash e : \exists X.\sigma \qquad \Delta, Y;\Gamma, x : \sigma\{Y/X\} \vdash e' : \tau' \qquad \Delta \vdash \tau' \qquad Y \notin \Delta}{\Delta;\Gamma \vdash \texttt{unpack}\ e\ \texttt{as}\ [Y, x]\ \texttt{in}\ e' : \tau'}$$

Notice that we had to add the context Δ, just as in the case of polymorphism, in order to make sure that no types refer to unbound type variables.

The following are the operational semantics for this feature:

$$\texttt{unpack}\ (\texttt{pack}\ _{\exists X.\sigma}[\tau, v])\ \texttt{as}\ [Y, x]\ \texttt{in}\ e \to e\{\tau/Y, v/x\}$$

There are also additional evaluation contexts:

$$E ::= \ldots \mid \texttt{pack}\ [\tau, [\cdot]] \mid \texttt{unpack}\ [\cdot]\ \texttt{as}\ [Y, x]\ \texttt{in}\ e$$

Here is a simple example illustrating how these new language features can be used:

$$\begin{array}{l}
\texttt{let}\ p_1 = \texttt{pack}_{\exists X.X*(X\to\texttt{bool})}[\texttt{int}, (5, \lambda n : \texttt{int}.(n = 1))]\ \texttt{in} \\
\texttt{let}\ p_2 = \texttt{pack}_{\exists X.X*(X\to\texttt{bool})}[\texttt{bool}, (\texttt{true}, \lambda x : \texttt{bool}.(n = 1))]\ \texttt{in} \\
\texttt{let}\ \mathit{apply} = \lambda p : \exists X.X * (X \to \texttt{bool})\,.\,(\#2p)(\#1p)\ \texttt{in} \\
\qquad \mathit{apply}(p_1) + \mathit{apply}(p_2)
\end{array}$$

Notice that the function *apply* can be applied to both p_1 and p_2 because they have the same existential type, even though internally their structure differs.

3 Existential Types and Constructive Logic

The existential types get their names partly because they correspond to inference rules of constructive logic involving the $\exists$ qualifier:

$$\frac{\Delta;\Gamma \vdash e\{\tau/X\} : \sigma\{\tau/X\} \quad \Delta \vdash \tau :: \mathsf{type} \quad X \notin \Delta}{\Delta;\Gamma \vdash \texttt{pack}\ _{\exists X.\sigma}[\tau, e] : \exists X.\sigma} \quad \Leftrightarrow \quad \frac{\Gamma \vdash \phi\{A/X\} \quad \Gamma \vdash A \in S}{\Gamma \vdash \exists X \in S.\phi}$$

$$\frac{\Delta;\Gamma \vdash e : \exists X.\sigma \quad \Delta, Y;\Gamma, x : \sigma\{Y/X\} \vdash e' : \tau' \quad \Delta \vdash \tau' \quad Y \notin \Delta}{\Delta;\Gamma \vdash \texttt{unpack}\ e\ \texttt{as}\ [Y, x]\ \texttt{in}\ e' : \tau'} \Leftrightarrow \frac{\Gamma \vdash \exists X \in S.\phi \quad \Gamma, Y \in S, \phi\{Y/X\} \vdash \phi_2 \quad Y \notin FTV(\phi_2)}{\Gamma \vdash \phi'}$$

Note that the set S corresponds here to the kind type. If there were more kinds in the type system then the correspondence would be even closer.

4 Existentials and modules in ML

There is a rough correspondence between existential types and the SML module mechanism. For example, the SML signature

```
sig
  type T
  val toBool: T->bool
end
```

Is roughly the same as $\exists X.X \to \texttt{bool}$. The unpack primitive is similar to the open operation on modules.