CS611 Lecture 36 | Existential types | 19 November 2007
Lecturer: Andrew Myers

1 Modeling objects with recursive types

We've been exploring language semantics in a largely reductionist way, by breaking apart complex mechanisms into simpler components. Objects are an example of a complex mechanism that we'd hope our studies would shed some light on. However, if we try to encode objects in terms of the simpler constructs we have seen so far, we see that there is something missing.

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;
}
```

One of the challenges of modeling objects is that they can refer to themselves. For example, the code of the **contains** method is implicitly recursive with respect to the object **this**, because 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:

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

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

We can construct "objects" of this type, assuming we can take a fixed point over objects (which is possible as long as only methods can refer to the fixed point):

```
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:unit. 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

While we can encode objects currently, we are missing one of the key concepts of object-oriented programming: *encapsulation*, which in the OO world sometimes is called *information hiding*. Encapsulation is a feature in which the type system hides internals of objects, enforces an abstraction barrier between the implementer and the clients of the class. This abstraction barrier helps keep different parts of the system loosely coupled, so they can be updated and maintained without close coordination.

Encapsulation is offered in its purest form by *existential types*. The idea is that we can hide part of a type τ and replace it with a type variable α. We write $\exists\alpha.\tau$ to represent this type, where α may be mentioned inside τ. But because this type doesn't say what α 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:

$$\begin{array}{l} \exists\alpha.\ \{\ \textbf{union:}\quad \textbf{S} \rightarrow \textbf{S} \\ \qquad \textbf{contains: int} \rightarrow \textbf{bool} \\ \qquad \textbf{private:}\ \ \alpha\ \} \end{array}$$

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 $[\tau, v] : \exists\alpha.\sigma$ where $v : \sigma\{\tau/\alpha\}$. 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:

$$\frac{\Delta;\Gamma \vdash e\{\tau/\alpha\} : \sigma\{\tau/\alpha\} \quad \Delta \vdash \exists\alpha.\sigma}{\Delta;\Gamma \vdash \textbf{pack}\ _{\exists\alpha.\sigma}[\tau, e] : \exists\alpha.\sigma}$$

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

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:

$$\textbf{unpack}\ (\textbf{pack}\ _{\exists\alpha.\sigma}[\tau, v])\ \textbf{as}\ [Y, x]\ \textbf{in}\ e \rightarrow e\{\tau/Y, v/x\}$$

There are also an additional evaluation contexts:

$$E[\bullet] ::= \ldots\ \mid \textbf{pack}\ [\tau, [\bullet]] \mid \textbf{unpack}\ [\bullet]\ \textbf{as}\ [\beta, x]\ \textbf{in}\ e$$

Here is a simple example illustrating that we can pack different types into an implementation of a value without the client being able to tell:

$$\begin{array}{l} \textbf{let}\ p_1 = \textbf{pack}_{\exists\alpha.\alpha*(\alpha\rightarrow\textbf{bool})}[\textbf{int}, (5, \lambda n : \textbf{int}.(n = 1))]\ \textbf{in} \\ \qquad \textbf{unpack}\ p\ \textbf{as}\ [\beta, x]\ \textbf{in}\ ((\textbf{right}\ x)\ (\textbf{left}\ x)) \end{array}$$

$$\begin{array}{l} \textbf{let}\ p_2 = \textbf{pack}_{\exists\alpha.\alpha*(\alpha\rightarrow\textbf{bool})}[\textbf{bool}, (\textbf{true}, \lambda b : \textbf{bool}.\neg b)]\ \textbf{in} \\ \qquad \textbf{unpack}\ p\ \textbf{as}\ [\beta, x]\ \textbf{in}\ ((\textbf{right}\ x)\ (\textbf{left}\ x)) \end{array}$$

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:

$$\frac{\Delta;\Gamma \vdash e\{\tau/\alpha\} : \sigma\{\tau/\alpha\} \quad \Delta \vdash \exists\alpha.\sigma}{\Delta;\Gamma \vdash \textbf{pack}\ _{\exists\alpha.\sigma}[\tau, e] : \exists\alpha.\sigma} \quad \leftrightarrow \quad \frac{\Gamma \vdash \phi\{A/X\} \quad \Gamma \vdash A \in S}{\Gamma \vdash \exists X \in S.\phi}$$

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

Note that there is nothing directly corresponding to the set S in the typing rules because the type system currently considered does not distinguish between multiple kinds.

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 $\exists\alpha.\alpha \rightarrow \textbf{bool}$. The **unpack** primitive is similar to the **open** operation on modules.