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:

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

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 = fold_intset inl (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: *data abstraction*, which in the OO world sometimes is called *information hiding* or *encapsulation*. This is 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.

Data abstraction 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:

$\exists\alpha.\ \mu S.\ \{$
    union: $S \to S$
    contains: int $\to$ bool
    private: α
$\}$

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:

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

$$\frac{\Delta;\Gamma \vdash e : \exists\alpha.\sigma \quad \Delta, \alpha::\textbf{type};\Gamma, x{:}\sigma \vdash e' : \tau' \quad \Delta \vdash \tau' :: \textbf{type} \quad \alpha \notin \Delta}{\Delta;\Gamma \vdash \textbf{let}\ [\alpha, x] = e\ \textbf{in}\ e' : \tau'}$$

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.

We need a reduction that unpacks an existentially quantified term:

$$\textbf{let}\ [\alpha, x] = [\tau, v]_{\exists\alpha.\sigma}\ \textbf{in}\ e \to e\{\tau/\alpha, v/x\}$$

There are also additional evaluation contexts:

$$E ::= \ldots \mid [\tau, E] \mid \textbf{let}\ [\alpha, x] = E\ \textbf{in}\ e$$

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:

$$\textbf{let}\ p_1 = [\textbf{int}, (5, \lambda n{:}\textbf{int}.\ n = 1)]_{\exists\alpha.\alpha*(\alpha\to\textbf{bool})}\ \textbf{in}$$
$$\textbf{let}\ [\alpha, x] = p_1\ \textbf{in}\ ((\textbf{right}\ x)\ (\textbf{left}\ x))$$

$$\textbf{let}\ p_2 = [\textbf{bool}, (\textbf{true}, \lambda b{:}\textbf{bool}.\ \neg b)]_{\exists\alpha.\alpha*(\alpha\to\textbf{bool})}\ \textbf{in}$$
$$\textbf{let}\ [\alpha, x] = p_2\ \textbf{in}\ ((\textbf{right}\ x)\ (\textbf{left}\ x))$$

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:

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

$$\frac{\Delta;\Gamma \vdash e : \exists\alpha.\sigma \quad \Delta, \alpha::\textbf{type};\Gamma, x{:}\sigma \vdash e' : \tau' \quad \Delta \vdash \tau' :: \textbf{type} \quad \alpha \notin \Delta}{\Delta;\Gamma \vdash \textbf{let}\ [\alpha, x] = e\ \textbf{in}\ e' : \tau'} \quad \leftrightarrow \quad \frac{\Gamma \vdash \exists X.\phi \quad \Gamma, X{:}\textbf{Prop}, \phi \vdash \phi' \quad \beta \notin FV(\phi')}{\Gamma \vdash \phi'}$$

4 Existentials and modules in ML

There is a rough correspondence between existential types and the SML module mechanism. For example, a ML signature **Rational** defined as:

module type Rational =
sig
  type t
  val plus: t $\rightarrow$ t $\rightarrow$ t
  ...
end

corresponds to the existential type $\exists \alpha.\{plus : \alpha \rightarrow \alpha \rightarrow \alpha, \dots \}$. A ML module that implements this signature is similar to an extensional value:

struct
  type t = int * int
  let plus x y = ...
  ...
end

corresponds to $[\mathbf{int} * \mathbf{int}, \{plus = \lambda x : \mathbf{int} * \mathbf{int}.\, \lambda y : \mathbf{int} * \mathbf{int}.\, \dots, \; \dots \}]_{\exists \alpha.\{plus:\alpha\rightarrow\alpha\rightarrow\alpha,\dots\}}$.

Unpacking is similar to the **open** operation on modules.

5 Strong existentials

So far the existential values we have seen are "weak" in the sense that the values of the abstract type α cannot really escape the module; they can exist only within an unpack term. This means we can't write code corresponding to this ML code:

let x : Rational.t = Rational.zero in
  Rational.plus(x,x)

To get access to several modules represented as existentials, it will be necessary to unpack them all at once at the top of the program, to create a scope in which the abstract types can be mentioned.

The idea of *strong existentials* is to allow the hidden type to be mentioned outside. We extend the type language with new *dependent module types* with the syntax $e.\alpha$:

$$\tau ::= \dots \mid \exists \alpha.\tau \mid e.\alpha \qquad \text{(e is pure)}$$

What makes the type $e.\alpha$ a *dependent type* is that it is a type that mentions a term, something we haven't seen before. The meaning of the type depends on something that is not necessarily known until run time. In general we will want to place some restrictions on what terms can be used in this position, to ensure soundness and to make type checking tractable. Here we use the description "pure" to capture these restrictions, though there is a whole spectrum of choices about what terms can be permitted, involving different tradeoffs in expressive power and tractability.

Certainly we don't want e to be something that has side effects. A simple choice is that e must be a variable name x. A more complicated choice is to allow record selector expressions of the form $x.y.z$. Some limited use of function application is the next step up, bringing us to roughly the expressive power of the SML and OCaml module systems, with its nested modules and functors.

In any case, we can now use dependent module types to express the typing rule for unpack:

$$\frac{\Delta; \Gamma \vdash e : \exists \alpha.\sigma \quad \Delta, \alpha :: \mathbf{type}; \Gamma, x : \sigma \vdash e' : \tau' \quad \Delta \vdash \tau' :: \mathbf{type} \quad \alpha \notin \Delta \quad pure(e)}{\Delta; \Gamma \vdash \mathbf{let}\ [\alpha, x] = e\ \mathbf{in}\ e' : \tau'\{e.\alpha/\alpha\}}$$

For example, if R is a variable containing the existential encoding of **Rational**, above, and we read $R.x$ as syntactic sugar for $\mathbf{let}\ [\alpha, m] = R\ \mathbf{in}\ m.x$, we can now type-check a term such as $\mathbf{let}\ c : R.t \rightarrow R.t \rightarrow R.t = R.plus\ \mathbf{in}\ c(R.zero)(R.zero)$, which has the type $R.t$.

One difficulty with allowing expressive module terms e is that it becomes hard to determine whether two types $e_1.\alpha$ and $e_2.\alpha$ are in fact the same type. It's hard, of course, because determining equality of terms in an expressive language is undecidable.

6 DeMorgan's laws

Constructively, $\exists X.\phi \Rightarrow \neg\forall X.\neg\phi$. This suggests that there should be a translation from existentials to universals, which is in fact true. Weak existential types can be encoded using universal types in System F, showing that there is an interesting duality between data abstraction and polymorphism.