1 Modeling Objects with Recursive Types

We have 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 had hoped 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 (left != null) && left.contains(n);
      else if (n > value) return (right != null) && right.contains(n);
      else return n == value;
   }
   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 λ-calculus. First, there is a type Intset being declared:

$$\mathsf{Intset} \;=\; \mu S.\{\mathsf{union} : S \to S,\, \mathsf{contains} : \mathsf{int} \to \mathsf{bool},\, \mathsf{value} : \mathsf{int},\, \mathsf{left} : S,\, \mathsf{right} : S\}$$

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

```
let s = fold_Intset (rec this: { union : Intset → Intset, ... }. // the unfolding of Intset
{
   union = λs' : Intset. ...
   contains = λn : int.
      if m < this.value then case this.left of
         λs':Intset.((unfold s').contains) n
      else if m > this.value then case this.right of
         λs':Intset.((unfold s').contains) n
      else n = this.value
}
```

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 will 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 object-oriented programming world sometimes is called *information hiding* or *encapsulation*. This is a feature in which the type system hides internals of objects, enforcing 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 does not 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 typecheck, as follows:

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

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

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

We need a reduction that unpacks an existentially quantified term:

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

There are also additional evaluation contexts:

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

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:

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

$$\begin{array}{l}\mathsf{let}\ p_2 = [\mathsf{bool}, (\mathit{true}, \lambda b : \mathsf{bool}.\, \neg b)]_{\exists\alpha.\alpha*(\alpha\to\mathsf{bool})}\ \mathsf{in}\\ \quad \mathsf{let}\ [\alpha, x] = p_2\ \mathsf{in}\ ((\mathsf{right}\ x)\ (\mathsf{left}\ x))\end{array}$$

3 Existential Types and Constructive Logic

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

Typing:

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

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

Constructive Logic:

$$\frac{\Gamma \vdash \varphi\{A/X\} \qquad \Gamma \vdash A : \mathsf{Prop}}{\Gamma \vdash \exists X.\varphi} \qquad\qquad \frac{\Gamma \vdash \exists X.\varphi \qquad \Gamma, X : \mathsf{Prop}, \varphi \vdash \varphi'}{\Gamma \vdash \varphi'}\ (X \notin FV(\varphi'))$$

4 Existentials and Modules in ML

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

```
module type Rational =
sig
  type t
  val plus: t → t → t
  ...
end
```

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

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

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

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 cannot 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 $e.\alpha$:

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

What makes the type $e.\alpha$ a *dependent type* is that it is a type that mentions a term, something we have not 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 do not 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 their nested modules and functors.

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

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

For example, if R is a variable containing the existential encoding of Rational, above, and we read $R.x$ as syntactic sugar for $\mathsf{let}\ [\alpha, m] = R\ \mathsf{in}\ m.x$, we can now typecheck a term such as $\mathsf{let}\ c : R.t \to R.t \to R.t = R.plus\ \mathsf{in}\ c(R.zero)(R.zero)$, 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 is hard, of course, because determining equality of terms in an expressive language is undecidable.

6 DeMorgan's Laws

Constructively, $\exists X.\varphi \Rightarrow \neg\forall X.\neg\varphi$ (but not conversely). 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.