Matthew S. Harris

Last time we started talking about existential types, which were a way to describe the encapsulation or information hiding that occurs in object-oriented languages and also occurs in other languages that have module systems, like ML. In ML we declare a module which has some type but we don't really say what it is, and we provide an implementation that explains what that type is. And we had a typing rule for existential types:

$$\frac{\Gamma \vdash e : \sigma[\tau/\alpha]}{\Gamma \vdash e : \exists\alpha.\sigma}$$

In our Howard-Curry isomorphism discussion, this corresponds directly to the rule that you would use for concluding a "there exists" clause in logic. In constructive logic, you can conclude that something exists if you can show an actual example, so

$$\frac{A[B/\alpha]}{\exists\alpha.A}$$

What about the rule for getting rid of the existential? If we want to be able to manipulate values of this type, we need to be able to somehow take this thing away. In logic we have a rule for doing that:

$$\frac{\exists\alpha.A \qquad \begin{array}{c}[A]\\ \vdots\\ C\end{array}}{C}$$

We need to place a restriction on this: α cannot appear in C or in any other assumptions of the proof of C. I'm using a slightly different notation than I did earlier for logic. These brackets that I'm introducing indicate an assumption that we're introducing. Earlier I was using the variable Γ to keep track of all the assumptions that we were allowed to make in constructing the proof. The idea here is that because we use this particular rule, this particular rule allows us to introduce an assumption of the form A, so we'll often write a little subscript next to the corresponding rule, showing that the application of this rule is the reason that we're able to introduce the assumption.

Let me go back and show you how we construct the rule for implication using this notation:

$$\frac{\begin{array}{c}[A]\\ \vdots\\ B\end{array}}{A \Rightarrow B}$$

This is absolutely equivalent to the rules I gave you earlier, where we had Γ keeping track of the list of assumptions that we're allowed to make. But this is the more standard notation for logic.

Now we have a rule for getting rid of $\exists$. So assuming that our isomorphism holds, we ought to be able to construct an expression whose typing corresponds to this rule. So we'll introduce an expression that looks like this (where e is an existential type):

$$\frac{\Delta; \Gamma \vdash e : \exists\alpha.\sigma \qquad \Delta \cup \{\alpha\}; \Gamma[x \mapsto \sigma] \vdash e' : \tau'}{\Delta; \Gamma \vdash \mathsf{with}\ e\ \mathsf{as}\ x : \sigma\ \mathsf{in}\ e' : \tau'}$$

In this code, we're allowed to treat the variable α as though it were a type. We can handle that by introducing a context Δ that keeps track of the legal type variables.

The way to think about this is that this is basically letting us unwrap the existential type e and letting us perform some computations on it. Now, we do have to have a restriction on this because once we get outside this expression, α isn't going to have any meaning. So as a side condition we need to require that $\alpha \notin \mathrm{FTV}(\tau') \cup \mathrm{FTV}(\Gamma)$. This corresponds exactly to the side condition in our logic case.

These existential types don't look that much like the modules we've been using. The existential type is introducing a single value and a hidden abstract type α. We can actually model a more conventional module system in terms of it. In a typical module system, we can introduce some number of named values and some number of named abstract types. It turns out that we can basically desugar a module system into existential types. I'll show you how we can do that.

These are modules that are a little bit fancier than the modules you've been looking at in the homework because they've got abstract types as well as bindings of names to values. So let's extend our notion of modules. We can define a series of types $\alpha_1, \ldots, \alpha_n$ and bind them to particular type expressions. Then we'll also have a set of bindings where we bind particular values to names.

$$(\textsf{module}\ (\textsf{deftype}\ \alpha_1 = \tau'_1, \ldots, \alpha_n = \tau'_n)\ (x_1 : \tau_1 = e_1, \ldots, x_m : \tau_m : e_m))$$

This is just a simple extension of the module mechanisms we've been playing with on the homework except that we get to introduce abstract types. We also have an extended notion of the module type. Our module types will look like

$$(\textsf{moduleof}\ (\textsf{type}\ \alpha_1, \ldots, \alpha_n)\ (x_1 : \tau_1, \ldots, x_m : \tau_m))$$

How can we express this notion of modules in terms of existential types? We need expressions that will allow us to unbundle the modules. So let's have an expression

$$(\textsf{import}\ e_m\ \textsf{in}\ e_b)$$

We're importing a module and acting as its internals on the expression e_v. The more typical word is with, but I'm writing it this way to emphasize that this is what's going on in, say, Java when you import a class.

I claim that these modules can be thought of as existential types. Can you see how to make that transformation? Basically we think about the moduleof expression as an existential type

$$\exists \alpha_1, \ldots, \alpha_n.\ \{x_1 : \tau_1, \ldots, x_m : \tau_m\}$$

In C, I can write a `.h` file, and I can declare in it a `struct foo`. Then we can write code in which I use the type `struct foo*`, and at no time do we actually know what type `foo` is. So the `struct foo` declaration is analagous to the declaration in the second part of the module expression. The deftype is analagous to actually seeing the definition when we look inside the `.c` file.

Let's say I'm writing a list package in C, and I want to hide the actual contents of lists. I can write a `.h` file with

```
struct list;
struct list *cons(head, struct list *tl);
```

So I can write a whole data structure package in C that implements pointers to some type that I don't know. This `cons` is one of these x_i's, and this list is one of these α_i's. So in the C file I would then write

```
struct list { int head;
              struct list *tail; };
```

where the `struct list *tail` is the τ'_1. I'm trying to translate this module by using existential quantification over all the type variables I introduced, and I'm producing one big record with all the values. That's how I'm able to cram this into an existential type framework. For a module it's going to be a record where I say

$$\{x_1 : \tau_1[\tau'_1/\alpha_1, \ldots, \tau'_m/\alpha_m] = e_1, \ldots\}$$

I'll be able to translate the import statement into one of these with statements, so what I would say is

$$\textsf{with}\ e_m\ \textsf{as}\ m : \{x_1 : \tau_1, \ldots, x_m : \tau_m\}\ \textsf{in let}\ x_1 = m.x_1, \ldots, x_m = m.x_m\ \textsf{in}\ e_b$$

The idea here is that the module is now represented as an existential type that contains a record, and I'm just pulling out the record and introducing a series of variables of the appropriate names so that they're all in scope when I evaluate e_b.

As you can see, we can use these module types to represent notions of modules from various languages. We can pretty much describe what's going on in C; describing what's happening in more principled languages like ML and Modula-3 is actually somewhat easier. But the reason we started talking about modules was actually that we wanted to know how to simulate objects, and this mechanism isn't going to work very well as a mechanism for talking about objects. So there's kind of a mismatch between module notions and objects. Objects don't have abstract types associated with them, so it isn't the case that when you have an object o, you can talk about an abstract type $o.t$.

There are a couple of other problems. Let's look at our point example again from last time. We said that we can write a point class in Java:

```
class Point {
  int x;
  int get_x();
  Point move_x(int dx);
}
```

If you try to translate this into a module framework, you've got a small problem because if we want to think of this point as being an existential type, then the internal `Point` reference has the same type as the module itself. So we have to have modules that refer to themselves, and we'll in fact have to use recursive types to explain what's going on. The existential type by itself is not going to give us what we want. We could imagine sticking a μt on the front and using t inside. One problem is that when we start taking recursive types and existential types together, the interactions with subtyping don't work out very well, and we want to be able to model that. Also we won't be able to handle inheritance properly in this kind of model.

1 Modeling objects without modules

Part of why we're getting into trouble here is that we're trying to cram objects, prematurely perhaps, into a framework that looked attractive because of the wonderful correspondence between logic and types. Perhaps what we need to do is actually back up and look at objects from first principles. In fact, we can look at objects in an untyped framework similar to the way we did the lambda calculus; we didn't start out by writing a type system for the lambda calculus — we built a type system and then discovered that we needed lots of fancy types to explain what was going on. In fact it's possible to develop an object calculus that corresponds to the lambda calculus but talks about objects rather than functions. I'll show how to do that, probably next time. Let's just talk about what features we'd like to see in a faithful model of objects.

Fundamentally, an object looks like a mapping from names to values. That's why we started out introducing record types — basic name-to-value mapping constructs. Here is the basic idea: an object is a name-to-value mapping. There are a lot of details about how we actually manage these things. Objects need to have some additional properties:

- Values can be functions (methods).
- Methods can return objects.
- Methods have a special variable `this` referring to the object.

There are some details though. There are actually two different classes of languages: class-based and object-based object-oriented languages. The issue here is how we actually manage inheritance. In class-based languages, objects are instances of classes, and we define new classes by inheriting from other classes. When we want to create an object, an instance of a class, then we use a constructor, which will produce a new object of that class. There shouldn't be any surprises here, since you're all probably familiar with class-based object-oriented languages. We can think of these classes as just being other objects. We normally like to think of classes as types, but they're clearly not exactly types because classes have some values associated with them: they have methods, which are function values, that their objects are going to acquire. We can think of classes as being objects that have a set of associated function values (methods) that the objects of that class will share. So class objects are repositories of shared information for all of the instance objects of that class. Then we (typically) could also associate with each class object a type, so there is a distinction between the type of a class and the class itself.

We can make that explicit in object-based languages. (Two such languages are Self and Cecil.) In object-based languages, when you want to construct a new object, you clone an existing object. Then you can modify the fields of the clone object with new values. You can actually use this cloning thing to model two different things. Cloning also is something you can use to model inheritance. The idea is that when you want to create a new subclass, you clone an existing object, and instead of modifying ordinary fields, you modify one of the fields that corresponds to a method. You can actually change one of the methods of the object you just cloned, and by doing so you effectively create a new kind of object corresponding to a new class. In object-based object-oriented languages, there's typically no distinction betwen fields and methods.

The original object-oriented languages were class-based. Simula-67 was a statically-typed class-based object-oriented language, and Smalltalk is an untyped class-based object-oriented language. The notion of object-based object-oriented languages came along later.

The important thing here about this inheritance business, whether it occurs in the class-based framework or the object-based framework, is that when you inherit a method, the special variable this always has to shift so that it always refers to the object of the subclass. It's precisely that that makes modeling object-oriented languages difficult.

1.1 Recursive record interpretation

Last time I gave you a little interpretation of a point object as a recursive record. That worked pretty well as long as we didn't try to do any inheritance. Let's see what happens when we actually try to do it. We'll model our `Point` object as

$$\mathsf{rec}\ p\ \{x = \mathsf{ref}\ 1, \mathsf{get\_x} = \lambda u : \mathsf{unit}.(\hat{}p.x)\}$$

The p in get_x corresponds to this; we needed to use the rec operator so we could make the methods of p refer to p itself. That sounds pretty good so far, so now what happens when we try to create a new object that inherits from this object? I'm going to stay with the object-based approach because it's simpler — we don't have to talk about classes and objects.

Let's say we want to create a new object p_2 that is the same as p except that we want to change x. One way we can do it is to create a new record

$$p_2 = \{x = \mathsf{ref}\ 2, \mathsf{get\_x} = p.\mathsf{get\_x}, \dots\}$$

We probably want some special notation for this notion of taking a record and changing one of its components. We'll write this as $(p.x \leftarrow \mathsf{ref}\ 2)$. In this case we'd consider this construction of a new instance of an object and not inheritance per se, although we could model inheritance with exactly the same operator.

This won't work because the function get_x is still talking about p, so $p_2.\mathsf{get\_x}(\#u)$ returns the x from p, not p_2. The problem is that this p was bound away, disappearing into the object, and we weren't able to talk about it when we constructed p_2.

1.2 Self-application interpretation

Self-application corresponds to what happens in real implementations of object-oriented languages, but unfortunately it's also kind of broken. Here's the idea: we'll not bind this variable explicitly, but we'll keep it outside as an argument:

$$p = \{x = \mathsf{ref}\ 1, \mathsf{get\_x} = \lambda p'u.(\hat{}p'x)\}$$

Obviously the unit argument is gratuitous, but I'm going to leave it in for consistency. Notice that we don't even have to talk about recursive records anymore because whoever calls us does us the favor of passing in the recursive reference for us. Now the idea is that every time you say $p.\mathsf{get\_x}(\#u)$, we'll think of this as actually being a call $p.\mathsf{get\_x}(p, \#u)$. Then we can say $p_2 = (p.x \leftarrow \mathsf{ref}\ 2)$ and we're okay because whenever we call $p_2.\mathsf{get\_x}$, we're going to be passing in p_2 implicitly and we'll be able to fetch the proper reference.

This is in fact what object-oriented languages like C++ do when you invoke a method — it passes an extra argument. So what's the problem here? The problem is that we can't assign types to these things. What types would we like to assign? We'd like to say

$$p : \mu P.\{x : \mathsf{ref\ int}, \mathsf{get\_x} = P \rightarrow \mathsf{unit} \rightarrow \mathsf{int}\}$$

and then when we construct p_2, we're going to have the same type. So the problem comes in when we try to use this for inheritance. Let's say that p_2, in addition to having a field x, has a new field y. Then we'd like the type of p_2 to be a subtype of the type of p, and that's going to get us into trouble because the type P is an argument, so it should be subtyped in a contravariant manner, but we're going to require covariant subtyping. To see this, I actually need to give you a rule for subtyping of recursive types:

$$\frac{\Delta \cup \{\alpha \leq \alpha'\} \vdash \tau \leq \tau'}{\Delta \vdash \mu\alpha.\tau \leq \mu\alpha'.\tau'}$$

Now the problem is that we're supposed to show

$$P' \leq P \vdash \mathrm{type}(p) \leq \mathrm{type}(p_2)$$

but if $P' \leq P$, then we don't have the necessary contravariant relationship between P and P' that we would need for establishing the subtype relationship between the type of $\mathsf{get\_x}$ in p and p_2. That's why the self-application model doesn't work.

What we saw here is that if we try to encode objects using our conventional type system, we get into trouble. It turns out that there are reasonably faithful encodings of objects that have been developed, but they're very complicated, and it's just not worth it. You're better off just designing an object type system from the ground up rather than trying to figure out how to translate it to our existing types. In the next couple of lectures we'll talk about how to do that.