Scribe: Asif-ul Haque, Xin Zheng | Lecturer: Andrew Myers

1 Introduction

In the programming languages we use often, we can define recursive data types. For example in Java we can define a binary tree like this.

```
class Tree –
    Tree left, right;
    int value;
"
```

In ML we can write something like this.

```
datatype tree = Leaf  |  Node of tree * tree * int
```

In typed-lambda calculus, we have not yet developed any mechanism to allow such recursive types. Intuitively, we can write the type tree like this.

$$tree = 1 + tree * tree * int$$

The tree on the right-hand side can be replaced by the definition itself and in this way we can get a binary tree of any size. The 1 signifies an empty tree analogous to the null value in Java. The definition is recursive and not surprisingly we need to take some sort of fixed point – fixed point on types. So we define a function F whose fixed point we need to find.

$$F(X) = \lambda X :: type.\ 1 + X * X * int$$

$$tree = F(tree)$$

2 Least vs Greatest Fixed Point

A relevant question we can ask is what fixed point are we looking for. There are two answers that interest us.

- **Least fixed point:** Such fixed points are inductive in nature and are for finite height trees.
- **Greatest fixed point:** Such fixed points are for both finite and infinite height trees. As an example of infinite height trees, some languages allow us to write

  $$rec\ y : tree.\ inr(y, y, 0)$$

  This constructs a binary tree of infinite height initialized with the value 0 at each node. In implementation, there is only one node that has value 0 with both its left and right pointers pointing to itself.

3 The μ Constructor

Here we define a new construct which we call **Fixed-point type constructor**. If $F(X)$ is the function whose fixed-point we are trying to find, we denote the fixed-point as $\mu X.\ F(X)$. By definition

$$F(\mu X.\ F(X)) = \mu X.\ F(X)$$

So we can now use this construct to define $tree = \mu X.\ 1 + X * X * int$. If we use τ for $F(X)$, then fixed-point would be $\mu X.\ \tau$. Thus

$$F(\mu X.\ \tau) = \tau\{\mu X.\ \tau/X\} = \mu X.\ \tau$$

This step of substituting the fixed-point for X inside τ "unfolds" τ. Going from one side to another of the following equality we get the **fold** and **unfold** operations.

$$\mu X.\ \tau = \tau\{\mu X.\ \tau/X\}$$

$$unfold : \mu X.\ \tau \longrightarrow \tau\{\mu X.\ \tau/X\}$$

$$fold : \mu X.\ \tau \longleftarrow \tau\{\mu X.\ \tau/X\}$$

When we write programs in Java or ML, we never write this μ construct. Instead the languages do this for us. In most programming languages this equality is an "isomorphism" rather than an equality.

$$\mu X.\ \tau \cong \tau\{\mu X.\ \tau/X\}$$

Because of this isomorphism, we have to change the views through fold and unfold operations. Based on how the conversion is handled by a programming language, we get two types.

- **Iso-recursive types** where $\mu X.\ X$ and $\tau\{\mu X.\ \tau/X\}$ are isomorphic and the fold and unfold operations are explicit. Most programming languages handle it this way.
- **Equi-recursive types** where either there are no fold/unfold operations or they are automatic. Modula-3 is one such rare language to support this.

In ML *case* and *let* constructs are examples of unfold operations. The constructors for datatypes are examples of fold operations.

4 Typing Rules

We can now write the typing rules for folds and unfolds. Like many of the rules we have seen so far, these will simply be a pair of "introduction" and "elimination" rules for μ-types.

$$\frac{\Gamma \vdash e : \tau\{\mu X.\tau/X\}}{\Gamma \vdash \mathsf{fold}_{\mu X.\tau}\ e : \mu X.\tau}\ (\mu \text{ introduction})$$

$$\frac{\Gamma \vdash e : \mu X \tau}{\Gamma \vdash \mathsf{unfold}\ e : \tau\{\mu X.\tau/X\}}\ (\mu \text{ elimination})$$

Suppose we want to type check a fold expression (annotated so we know what the fold is meant to do). We can think of it as function that gives you a $\mu X.\tau$, given that its argument e has type $\tau\{\mu X.\tau/X\}$, which is really just the same type. This is what we have in the first rule.

The second rule simply says that unfold will do the opposite. If e has type $\mu X.\tau$, then unfold e has type $\tau\{\mu X.\tau/X\}$.

These two rules say that fold and unfold really do behave just like functions as we have described them.

5 An Example

Let's now write a little piece of code to demonstrate what we have talked about so far. Suppose we want to write a program to add up a list of numbers. How would we define a recursive list type? It's a recursive type, so there'll be a $\mu X.$. We'll need a 1 to represent the empty list, and the general case is that we have an int followed by the rest of the list, i.e., $\mathsf{int} * X$. So we can define

$$\mathsf{intlist} \triangleq \mu X.\, 1 + \mathsf{int} * X$$

Now we can write our function to add up an intlist, which we'll call sum. This is going to be a recursive function, so we'll need to take a fix point and declare it as

$$\mathsf{let\ sum} = \mathsf{rec}\ f : \mathsf{intlist} \to \mathsf{int}.\ \lambda l : \mathsf{intlist}.$$

And what do we do in the body of this function? We want to do a case on l, but we need a sum, and l is a μ-type. So we need to first unfold l (and this is what ML will do for you automatically when it sees a case). So we have

$$\mathsf{let}\ l' : 1 + \mathsf{int} * \mathsf{intlist} = \mathsf{unfold}\ l\ \mathsf{in}$$

And now we can do our typed λ-calculus case.

$$\begin{array}{ll} \mathsf{case}\ l'\ \mathsf{of} & \\ & \lambda u : 1.\ 0 \\ \mid & \lambda p : \mathsf{int} * \mathsf{intlist}.\ (\#1\ p) + f\ (\#2\ p) \end{array}$$

This is just the same code that you would write in ML, except we have broken out some of the things that ML hides for you. In particular, we've explicitly shown that there is recursion happening with our definition of the intlist type, and the unfold that needs to happen to get the different views of our type.

6 Structural Operational Semantics

This is actually pretty simple. We have our introduction form (fold), and our elimination form (unfold), and by now, you know that all structural operational semantics is just the elimination form wrapped around the introduction form and some magic happens. So the left hand side would be unfold ($\mathsf{fold}_{\mu X.\,\tau}\ e$). And what does this step to? No surprises here: just e.

$$\mathsf{unfold}\ (\mathsf{fold}_{\mu X.\,\tau}\ e) \longrightarrow e$$

This is showing that there's nothing really interesting happening here with these fold and unfold operations. They are just shifting views, and the fold and unfold mark which view we are looking at.

7 Self Application and Ω

Recall the self application and Ω terms that we saw in previous lectures.

$$\begin{array}{rcl} SA & \triangleq & \lambda x.\ x\ x \\ \Omega & \triangleq & SA\ SA \end{array}$$

We can now actually give these terms types, by sticking in some judicious folding. We know that x has to be some kind of function type, where it can take itself as its argument. So x must have a type like

$$x : \mu Y.\ Y \to \tau$$

for some τ that we have not yet defined.

So what will be the type of unfold x? We simply do one step of unfold for our μ-type.

$$\mathsf{unfold}\ x : (\mu Y.\ Y \to \tau) \to \tau$$

Now unfold x looks like a function that takes in something of the type of x as an argument. So we can take unfold x and apply it to x. And what's the type of (unfold x) x?

$$(\mathsf{unfold}\ x)\ x : \tau$$

since $\mathsf{unfold}\ x$ gives us a τ, and we gave it an appropriate argument.

So now we can write the fully typed SA term.

$$SA \triangleq \lambda x : \mu Y.\, Y \to \tau.\, (\mathsf{unfold}\ x)\ x : (\mu Y.\, Y \to \tau) \to \tau$$

What if we folded the SA type? We unfolded x from $\mu Y.\, Y \to \tau$ to $(\mu Y.\, Y \to \tau) \to \tau$, so we can fold back in the opposite direction.

$$\mathsf{fold}_{\mu Y.\, Y \to \tau}\ SA : \mu Y.\, Y \to \tau$$

Therefore, we can take SA and apply it to $\mathsf{fold}\ SA$, and get a τ.

$$SA\ (\mathsf{fold}\ SA) : \tau$$

And this, you'll notice, is just the same as the Ω term. If we run our operational semantics, we'll discover that this expression is just going to go off into an infinite loop, just like it's supposed to do. And yet, we have everything well-typed.

We never picked what τ was, and indeed, we can choose anything we want. In particular, $\mathsf{0}$ is a good choice, so we can write

$$\Omega : \mathsf{0}$$

which is appealing since we know Ω will never return.

8 Numbers as a Recursive Type

What else can we do with recursive types? Turns out we don't need to have natural numbers anymore. Recall that the typed λ-calculus started out with $\mathsf{1}$, $\mathsf{boolean}$, and int. We already got rid of $\mathsf{boolean}$, since it can be represented as a sum, $\mathsf{1} + \mathsf{1}$.

But we can also treat numbers as a recursive type. A natural number is either 0, which we will represent a $\mathsf{1}$ (unit), or it's a successor of a natural number. We can write that as

$$\mathsf{nat} \triangleq \mu X.\, \mathsf{1} + X$$

So our representation of 0 will be a folded *null* that has been injected into the sum. And 1 would basically be a folded 0.

$$\begin{aligned} 0 &\triangleq \mathsf{fold}_{\mathsf{nat}}\ \mathsf{inl}(null) \\ 1 &\triangleq \mathsf{fold}_{\mathsf{nat}}\ \mathsf{inr}(0) \end{aligned}$$

And so on for $2, 3, 4, \ldots$

We can use nat to code up all of the usual arithmetic that we want. For example, successor would be

$$SUCC \triangleq \lambda x : \mathsf{nat}.\, \mathsf{fold}_{\mathsf{nat}}\ \mathsf{inr}(x) : \mathsf{nat}$$

So all we really need is the $\mathsf{1}$ type. If we have recursive types, products, and sums, we can build all of the other types like natural numbers, integers, floating point numbers, and so on from the $\mathsf{1}$ type.

9 Untyped to Typed λ-Calculus

Now that we have all the expressive power we want in types, we can encode the (untyped) λ-calculus with types. Taking an arbitrary λ-calculus term that does not say what its type is, we can write down a type for it. So what's the type of a term in the λ-calculus?

Let's call that type U. We know that terms in the λ-calculus are all functions that can take in other terms of the λ-calculus and give you back another term in the λ-calculus as a result. So it must be the case that U is isomorphic to $\mathsf{U} \to \mathsf{U}$. That means we can define it as

$$\mathsf{U} \triangleq \mu X.\, X \to X$$

which will satisfy the isomorphism.

And we can in fact write a translation from the λ-calculus to the typed λ-calculus with recursive types.

$$\begin{array}{rcl} [\![x]\!] & = & x \\ [\![e_0\ e_1]\!] & = & (\mathsf{unfold}\ e_0)\ e_1 \\ [\![\lambda x.\, e]\!] & = & \mathsf{fold}_{\mathsf{U}}\ \lambda x : \mathsf{U}.\, [\![e]\!] \end{array}$$

Note that every translation gives us something of type U.

10 Closed vs Open Recursion

There is one thing that is limiting about our take on recursive types so far. What we have been looking at is a *closed* recursion mechanism. That is, if you look at a type like $\mu X.\, X \to X$, the scope of the recursion is very clearly defined. There is a $\mu X.$, and X can only be mentioned in its body, and you therefore know exactly where you are taking a fixed point. That's nice, since the meaning of types is locally defined in some sense.

If you look at a lot of languages though, you'll see that they give a more open recursion. For example, in Java, it's not the case that a class can only refer to itself. It can also refer to other classes. So you can have a class A that refers to a class B, where B also refers to A.

```
class A {
  B x;
}

class B {
  A x;
}
```

Notice that nowhere did we have to say that there's a fixed point. But depending on how we define the meaning of these types, we need to take a big fixed point over all of them. In general, if you want to understand the meaning of a bunch of types in say a Java virtual machine, it's really a big fixed point over all of the classes in the system, since they can all refer to each other in arbitrarily complicated ways.

Mechanisms that give you open recursion have this idea that you use names for fixed points, and the meaning of names is not really defined in a local way. In Java, we can create these names A and B, and they are referring to particular components of a big fixed point that we took over the entire system, but we didn't have to explicitly invoke that fixed point constructor.

The plus of this is that it makes sense for extensibility and reuse. It says that you can grab a bunch of classes that you have sitting around, pull them into a system, and you don't have to define where the fixed point is taken. In fact, the classes that you are taking off the shelf can even refer to classes that are not yet defined, to be supplied by you. The fixed point will then be automatically taken across the additional classes.

Open recursion gives you a lot of flexibility, but it does come with a price. There are some semantic issues. Suppose we have the following.

```
class A {
  static final int x = B.x + 1;
}

class B {
  static final int x = A.x + 1;
}
```

What does this code mean? In Java lexicon, static final fields are supposed to look like compile-time constants. But there is no actual fixed point solution to this code. There are no integers that we can assign to A.x and B.x that can make both of the equations come out true. So open recursion mechanisms are a bit problematic.

What do you get if you actually did this in Java? You'll get a 1 and a 2, but which is which, you don't know. It depends on the order that these two classes are initialized.