Lecturer: Andrew Myers

1 Introduction

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

class Tree {
  Tree left, right;
  int value;
}

Similarly, in ML we can define recursive datatypes, with declarations like:

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

In the simply typed-lambda calculus, we have no corresponding mechanism.
Intuitively, we would like a tree type that satisfies the following equation:

tree = unit + 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 **unit** signifies an empty tree, analogous to the **null** value in Java.

2 Recursive Types as Regular Labeled Trees

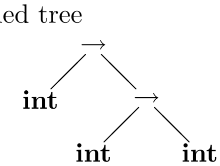
By expanding the equation for **tree**, we can see that

$$
\begin{aligned}
\alpha &= \mathbf{unit} + \alpha * \alpha * \mathbf{int} \\
&= \mathbf{unit} + (\mathbf{unit} + \alpha * \alpha * \mathbf{int}) * (\mathbf{unit} + \alpha * \alpha * \mathbf{int}) * \mathbf{int} \\
&= \mathbf{unit} + (\mathbf{unit} + (\mathbf{unit} + \alpha * \alpha * \mathbf{int}) * (\mathbf{unit} + \alpha * \alpha * \mathbf{int}) * \mathbf{int}) * (\mathbf{unit} + (\mathbf{unit} + \alpha * \alpha * \mathbf{int}) * (\mathbf{unit} + \alpha * \alpha * \mathbf{int}) \\
&= \cdots
\end{aligned}
$$

At each level, we have a finite type with the type variable α appearing at some of the leaves, and we obtain the next level by substituting the right-hand side of the equation for α. This gives a sequence of deeper and deeper finite trees, where each successive tree is a substitution instance of the previous tree.

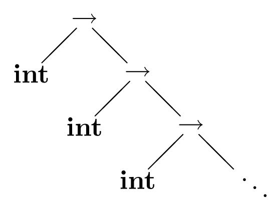
If we take the limit of this process, we have an infinite tree. We can think of this as an infinite labeled graph whose nodes are labeled with the type constructors *, +, **int**, and **unit**. This is very much like an ordinary type expression, except that it is infinite. There are no more α's, because we have substituted for all of them all the way down. This infinite tree is a solution of the tree equation, and this is what we take as the type **tree**.

In general, let Σ be a signature consisting of several type constructors of various arities. For example, Σ might consist of the type constructors $\rightarrow$, *****, **+**, **unit**, and **int**. We can form the set of (finite) types over Σ inductively in the usual way. Each such type can be regarded as a finite labeled tree. For example, the type $\mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int}$ can be viewed as the labeled tree



Now let us add some infinite types. These are infinite labeled trees that respect the arities of the constructors in Σ; that is, if the constructor is binary (such as * or $\rightarrow$), any node labeled with that constructor must have exactly two children; and if the constructor is nullary, such at **unit**, then any node labeled with that symbol must be a leaf. Within these constraints, the tree may be infinite.

A (finite or infinite) expression with only finitely many subexpressions up to isomorphism is called *regular*. For example, the infinite type



is regular, since it has only two subexpressions up to isomorphism, namely itself and **int**. The limit of the unrolling of the equation above, which we took to be the type **tree**, is also regular; it has exactly five subexpressions up to isomorphism, namely **tree**, **unit**, **tree * tree * int**, **tree * tree**, and **int**.

Regular trees are all we need to provide solutions to finite systems of type equations. Suppose we have n type equations in n variables:

$$\begin{aligned}\alpha_1 &= \tau_1 \\ &\vdots \\ \alpha_n &= \tau_n,\end{aligned} \tag{1}$$

where each τ_i is a finite type over the type constructors Σ and type variables $\alpha_1, \ldots, \alpha_n$. This system has a solution $\sigma_1, \ldots, \sigma_n$ in which each σ_i is a regular tree. Moreover, if no right-hand side is a variable, then the solution is unique.

3 The μ Constructor

The definition of **tree** 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 (we use a double colon to show that it's a function operating on types.

$$F(X) = \lambda\alpha :: \mathit{type}.\ 1 + \alpha * \alpha * \mathit{int}$$

$$\mathbf{tree} = F(\mathbf{tree})$$

Suppose we have a **fixed-point type constructor** that solves these equations. If $F(\alpha)$ is the function whose fixed-point we are trying to find, we denote the fixed-point as $\mu\alpha.\ F(\alpha)$. By definition, $F(\mu\alpha.\ F(\alpha)) = \mu\alpha.\ F(\alpha)$. So we can use this construct to define $\mathbf{tree} = \mu\alpha.\ \mathbf{unit} + \alpha * \alpha * \mathbf{int}$. If we write τ for $F(\alpha)$, then fixed point would be $\mu\alpha.\ \tau$. Thus,

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

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

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

$$\mathbf{unfold} : \mu\alpha.\ \tau \longrightarrow \tau\{\mu\alpha.\ \tau/\alpha\}$$

$$\mathbf{fold} : \mu\alpha.\ \tau \longleftarrow \tau\{\mu\alpha.\ \tau/\alpha\}$$

The solutions $\sigma_1, \ldots, \sigma_n$ to any finite system of the form (1) can be expressed in terms of μ. For example, suppose τ_1 and τ_2 are finite type expressions over the type variables α_1, α_2 such that neither τ_1 nor τ_2 is a variable. The system

$$\alpha_1 = \tau_1 \qquad\qquad \alpha_2 = \tau_2$$

has a unique solution σ_1, σ_2 specified by

$$\sigma_1 = \mu\alpha_1.\,(\tau_1\{\mu\alpha_2.\,\tau_2/\alpha_2\}) \qquad\qquad \sigma_2 = \mu\alpha_2.\,(\tau_2\{\mu\alpha_1.\,\tau_1/\alpha_1\}).$$

Mutually recursive type declarations arise quite often in practice. For example, consider the following Java class definitions for **Node** and **Edge**:

```
class Node {
   Edge[] inEdges, outEdges;
}

class Edge {
   Node source, sink;
}
```

Note that **Node** refers to **Edge** and vice versa. So we must take a mutual fixed point when assigning a meaning to such types.

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:

$$\mu\alpha.\,\tau \cong \tau\{\mu\alpha.\,\tau/\alpha\}$$

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 approaches to recursive types:

- **Iso-recursive types** where $\mu\alpha.\,\tau$ and $\tau\{\mu\alpha.\,\tau/\alpha\}$ are isomorphic and the fold and unfold operations are explicit. Most programming languages handle it roughly in this way.

  For example, ML has recursive datatypes, but a type is not equal to its unfolding. The *case* construct signals to the compiler that a recursive datatype needs to be unfolded. The constructors for datatypes are examples of operations that signal the need to fold.

- **Equi-recursive types** where either there are no operations indicating when to fold and unfold recursive types. A few languages do support this, such as Modula-3. For example, in Modula-3, the following two types are interchangeable in any context:

```
TYPE foo = RECORD x: INTEGER, y: REF foo END
TYPE bar = RECORD x: INTEGER,
                  y: REF RECORD
                    x: INTEGER integer
                    y: REF bar
                   END
           END
```

4 Typing Rules

Let's explore the isorecursive approach. We can write the typing rules for **fold** and **unfold**. Like many of the rules we have seen so far, these will simply be a pair of "introduction" and "elimination" rules for μ-types.

$$\frac{\Gamma \vdash e : \tau\{\mu\alpha.\,\tau/\alpha\}}{\Gamma \vdash \mathbf{fold}_{\mu\alpha.\,\tau}\ e : \mu\alpha.\,\tau}\ (\mu \text{ introduction})$$

$$\frac{\Gamma \vdash e : \mu\alpha\tau}{\Gamma \vdash \textbf{unfold}\ e : \tau\{\mu\alpha.\,\tau/\alpha\}}\ (\mu \text{ elimination})$$

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 $\mu\alpha.\,\tau$, given that its argument e has type $\tau\{mut\alpha\tau/\Box\}$, 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 $\mu\alpha.\,\tau$, then **unfold** e has type $\tau\{\mu\alpha.\,\tau/\alpha\}$.

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

5 An Example

Let's now write code to demonstrate recursive types. 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 $\mu\alpha.$. We'll need a **unit** to represent the empty list, and the general case is that we have an **int** followed by the rest of the list, i.e., $\textbf{int} * \alpha$. So we can define

$$\textbf{intlist} \quad \triangleq \quad \mu\alpha.\,\textbf{unit} + \textbf{int} * \alpha$$

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:

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

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

$$\textbf{let}\ l' : \textbf{unit} + \textbf{int} * \textbf{intlist} = \textbf{unfold}\ l\ \textbf{in}$$

And now we can do our typed **case**.

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

This is just the same code that you would write in ML, except you can see explicitly some things that ML hides from 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

The operational semantics are straightforward. 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** ($\textbf{fold}_{\mu\alpha.\,\tau}\ e$). And what does this step to? No surprises here: just e.

$$\textbf{unfold}\ (\textbf{fold}_{\mu\alpha.\,\tau}\ e) \longrightarrow e$$

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.

$$\begin{aligned} SA &\triangleq \lambda x.\, x\ x \\ \Omega &\triangleq SA\ SA \end{aligned}$$

We can now give these terms types, by adding 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:

$$x : \mu\alpha.\, \alpha \to \tau$$

for some type τ.

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

$$\textbf{unfold}\ x : (\mu\alpha.\, \alpha \to \tau) \to \tau$$

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?

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

since **unfold** x gives us a τ, and we gave it an appropriate argument.

So now we can write the fully typed SA term.

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

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

$$\textbf{fold}_{\mu\alpha.\, \alpha\to\tau}\ SA : \mu\alpha.\, \alpha \to \tau$$

Therefore, we can take SA and apply it to **fold** SA, and get a τ.

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

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 goes into an infinite loop, just like before. And yet, it's well-typed.

We never picked what τ was, and indeed, we can choose anything we want. This is a sure sign that the term will never produce any result.

$$\Omega : \tau$$

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 **unit**, **boolean**, and **int**. We already saw how to get rid of **boolean**, by translating it to the sum **unit** + **unit**.

Using recursive types, we can encode numbers. A natural number is either 0, which we will represent as **null**, or it's the successor of a natural number. Thus, we can define a type for natural numbers:

$$\textbf{nat} \triangleq \mu\alpha.\, \textbf{unit} + \alpha$$

Our representation of 0 is a folded *null* that has been injected into the sum. And *unit* would is folded 0, and so on.

$$\begin{array}{rcl} 0 & \triangleq & \mathbf{fold_{nat}}\ \mathbf{inl}(null) \\ 1 & \triangleq & \mathbf{fold_{nat}}\ \mathbf{inr}(0) \\ 2 & \triangleq & \mathbf{fold_{nat}}\ \mathbf{inr}(1) \\ & \dots & \end{array}$$

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

$$SUCC \triangleq \lambda x : \mathbf{nat}.\, \mathbf{fold_{nat}}\ \mathbf{inr}(x) : \mathbf{nat}$$

So all we really need is the **unit** 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 **unit** 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 $\mathbf{U}$. 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 $\mathbf{U}$ is isomorphic to $\mathbf{U} \to \mathbf{U}$. That means we can define it as:

$$\mathbf{U} \triangleq \mu\alpha.\, \alpha \to \alpha$$

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.

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

Note that every translation gives us something of type $\mathbf{U}$.

10 Semantics of recursive types

One question we can ask is what fixed point we expect to get from the μ type constructor. One way to solve these equations is to consider them as inductive definitions, in which the type corresponds to the union of all sets generated by finite applications of F. This is an adequate interpretation for eager languages where we do not expect to be able to construct infinite trees by taking fixed points. However, even in an eager language, we expect to be able to take fixed points over functions. So for general F, we want the type to denote CPOs that we can take fixed points over. The inductive construction will not do that.

Fortunately, we have all the necessary machinery. We understand the μ type constructor as finding the fixed point to a domain equation that is solved according to Scott's projective limit construction (seen in an earlier lecture). This produces a CPO solution.

11 Closed vs Open Recursion

There is one thing that is limiting about our take on recursive types so far. A type like $\mu\alpha.\, \alpha \to \alpha$ is a *closed* recursion mechanism, in which the scope of the recursion is very clearly defined. There is a $\mu\alpha.$, and α 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.

Many languages, such as Java, provide an open recursion mechanism. For example, in Java, a class can refer to other classes, which can refer freely back to the original class. This is nice because you don't have to define an ordering on classes:

class A {
  B x;
}

class B {
  A x;
}

Notice that nowhere did we have to say that there's a fixed point.

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. With open recursion mechanisms, different names in the program can implicitly correspond to fixed point constructions. The meaning of names is not 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 implicitly over the entire system.

The plus is that it supports extensibility and reuse. 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 do this in Java? You'll get a 1 and a 2, but which is which depends on the order that these two classes are initialized.