

Lecture notes for CS 6110 (Spring'09) taught by Andrew Myers at Cornell; edited by Amal Ahmed, Fall'09.

## 1 Introduction

Industrial-strength programming languages have a feature we haven't looked at yet: 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 variants (aka datatypes), with declarations like the following:

```
type 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 = 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 permits an empty tree to be represented using **null**, just as we might in Java.

## 2 Recursive Types as Regular Labeled Trees

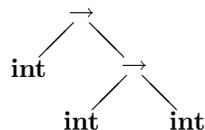
By expanding the equation for **tree**, we can see that

$$\begin{aligned} \alpha &= 1 + \alpha * \alpha * \mathbf{int} \\ &= 1 + (1 + \alpha * \alpha * \mathbf{int}) * (1 + \alpha * \alpha * \mathbf{int}) * \mathbf{int} \\ &= 1 + (1 + (1 + \alpha * \alpha * \mathbf{int}) * (1 + \alpha * \alpha * \mathbf{int}) * \mathbf{int}) * (1 + (1 + \alpha * \alpha * \mathbf{int}) * (1 + \alpha * \alpha * \mathbf{int}) * \mathbf{int}) * \mathbf{int} \\ &= \dots \end{aligned}$$

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

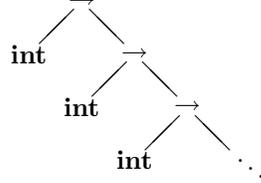
If we take the limit of this process, we have an infinite tree. We can think of this as an infinite labeled graph whose nodes are labeled with the type constructors  $*$ ,  $+$ , **int**, and 1. This is very much like an ordinary type expression, except that it is infinite. There are no more  $\alpha$ '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  $\Sigma$  be a signature consisting of several type constructors of various arities. For example,  $\Sigma$  might consist of the type constructors  $\rightarrow$ ,  $*$ ,  $+$ , 1, and **int**. We can form the set of (finite) types over  $\Sigma$  inductively in the usual way. Each such type can be regarded as a finite labeled tree. For example, the type **int**  $\rightarrow$  **int**  $\rightarrow$  **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  $\Sigma$ ; 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 as  $1$ , 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**,  $1$ , **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  $\tau_i$  is a finite type over the type constructors  $\Sigma$  and type variables  $\alpha_1, \dots, \alpha_n$ . This system has a solution  $\sigma_1, \dots, \sigma_n$  in which each  $\sigma_i$  is a regular tree. Moreover, if no right-hand side is a variable, then the solution is unique.

### 3 The $\mu$ 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):

$$\begin{aligned} F(X) &= \lambda \alpha :: \text{type}. 1 + \alpha * \alpha * \text{int} \\ \mathbf{tree} &= F(\mathbf{tree}) \end{aligned}$$

Let us posit 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 **tree**  $= \mu \alpha. 1 + \alpha * \alpha * \mathbf{int}$ . If we write  $\tau$  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  $\alpha$  inside  $\tau$  “unfolds”  $\tau$ . Going from one side to another of the following equality we get the **fold** and **unfold** operations.

$$\begin{aligned} \mu \alpha. \tau &= \tau\{\mu \alpha. \tau / \alpha\} \\ \mathbf{unfold} &: \mu \alpha. \tau \longrightarrow \tau\{\mu \alpha. \tau / \alpha\} \\ \mathbf{fold} &: \mu \alpha. \tau \longleftarrow \tau\{\mu \alpha. \tau / \alpha\} \end{aligned}$$

The solutions  $\sigma_1, \dots, \sigma_n$  to any finite system of the form (1) can be expressed in terms of  $\mu$ . For example, suppose  $\tau_1$  and  $\tau_2$  are finite type expressions over the type variables  $\alpha_1, \alpha_2$  such that neither  $\tau_1$  nor  $\tau_2$  is a variable. The system

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

has a unique solution  $\sigma_1, \sigma_2$  specified by

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

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  $\mu$  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 but not equal, and the fold and unfold operations are explicit. Most programming languages handle it roughly in this way, by combining fold and unfold with an existing operation that is supported by the kind of type that supports recursion. For example, ML has recursive variants, but a type is not equal to its unfolding. The **case** (or **match**) construct signals to the compiler that a recursive datatype needs to be unfolded. The constructors for datatypes, conversely, implicitly fold the type.
- **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,
                   y: REF bar
                 END
END
```

It is possible to decide the equality of the types **foo** and **bar** efficiently because the recursive type definitions stand for regular types. There are therefore only a finite number of type expressions that need to be compared.

## 4 Typing Rules

Let's explore the more explicit 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  $\mu$ -types.

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

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

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

We can see how recursive types work by writing some code. 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 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., **int** \*  $\alpha$ . So we can define

$$\mathbf{intlist} \triangleq \mu\alpha. 1 + \mathbf{int} * \alpha$$

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:

**let sum = rec f : intlist  $\rightarrow$  int.  $\lambda l$  : 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  $\mu$ -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

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

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

**case  $l'$  of**  
 $u : 1. 0$   
 $| p : \mathbf{int} * \mathbf{intlist}. (\#1 p) + f (\#2 p)$

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**). Both of them are eager:

$$E ::= \dots \mid \mathbf{fold}_{\mu\alpha.\tau} E \mid \mathbf{unfold} E$$

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** (**fold** <sub>$\mu\alpha.\tau$</sub>   $v$ ). And what does this step to? No surprise: just  $v$ .

$$\mathbf{unfold} (\mathbf{fold}_{\mu\alpha.\tau} v) \longrightarrow v$$

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 $\Omega$

Recall the self-application and  $\Omega$  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:

$$x : \mu\alpha. \alpha \rightarrow \tau$$

for some type  $\tau$ .

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

$$\mathbf{unfold} x : (\mu\alpha. \alpha \rightarrow \tau) \rightarrow \tau$$

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**  $x$ )  $x$ ?

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

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

So now we can write the fully typed  $SA$  term.

$$SA \triangleq \lambda x : \mu\alpha. \alpha \rightarrow \tau. (\mathbf{unfold} x) x : (\mu\alpha. \alpha \rightarrow \tau) \rightarrow \tau$$

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

$$\mathbf{fold}_{\mu\alpha. \alpha \rightarrow \tau} SA : \mu\alpha. \alpha \rightarrow \tau$$

Therefore, we can take  $SA$  and apply it to **fold**  $SA$ , and get a  $\tau$ .

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

And this, you'll notice, is just the same as the  $\Omega$  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  $\tau$  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  $\lambda$ -calculus started out with 1, **boolean**, and **int**. We already saw how to get rid of **boolean**, by translating it to the sum  $1 + 1$ .

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:

$$\mathbf{nat} \triangleq \mu\alpha. 1 + \alpha$$

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

$$\begin{aligned} 0 &\triangleq \mathbf{fold}_{\mathbf{nat}} \mathbf{inl}(\mathbf{null}) \\ 1 &\triangleq \mathbf{fold}_{\mathbf{nat}} \mathbf{inr}(0) \\ 2 &\triangleq \mathbf{fold}_{\mathbf{nat}} \mathbf{inr}(1) \\ &\dots \end{aligned}$$

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

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

So all we really need is the 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 1 type.

## 9 Untyped to Typed $\lambda$ -Calculus

Now that we have all the expressive power we want in types, we can encode the (untyped)  $\lambda$ -calculus with types. Taking an arbitrary  $\lambda$ -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  $\lambda$ -calculus?

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

$$\mathbf{U} \triangleq \mu\alpha. \alpha \rightarrow \alpha$$

which will satisfy the isomorphism.

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

$$\begin{aligned} \llbracket x \rrbracket &= x \\ \llbracket e_0 e_1 \rrbracket &= (\mathbf{unfold} e_0) e_1 \\ \llbracket \lambda x. e \rrbracket &= \mathbf{fold}_{\mathbf{U}} \lambda x : \mathbf{U}. \llbracket e \rrbracket \end{aligned}$$

Note that every translation gives us something of type **U**.

This translation can be implemented directly in SML and OCaml by defining recursive types.

```
type u = Fun of f
and f = u->u
...
```

Then **fold** is implemented using the **Fun** constructor, and **unfold** is implemented using **case/match**.