Scribe: Arjun Rao, Yogeshwer Sharma Lecturer: Andrew Myers

1 Introduction

In the previous lectures, we have seen many types but none of the types supports an unbounded data-structure. For example, it is not possible to make a list in the types considered till now, as the list can grow to an arbitrary size and our type system does not support that. These types are quite common in modern programming languages. Consider, for example, the following Java code:

```
class Tree {
      Tree left                        //reference to the left subtree
      Tree right                       //reference to the right subtree
      int value
}
```

or the following C code:

```
struct Tree {
       struct Tree *left                //pointer to the left subtree
       struct Tree *right               //pointer to the right subtree
       int value
}
```

or the following SML code

```
type Tree = empty
          | Node of Tree*Tree*int
```

In all these cases, Tree is defined in terms of itself and hence it can grow to an arbitrary size (although finite). To define these structures, we need a general mechanism by which we can define complex structures in terms of smaller and simpler structures. This mechanism is called *recursive types* and today we will see this mechanism which support this type of unbounded data structures.

2 μ-types

Consider the above case of a tree, each of whose node contains a *pointer* to left subtree, a *pointer* to the right subtree and a value in the node, say an integer. In such a case, the type of the tree becomes (calling the type of tree to be τ)

$$\tau = 1 + \tau * \tau * \text{int}$$

We can express this type using a new kind of type constructor: $\mu X.1 + X * X * \mathsf{int}$. In fact, we're looking for a kind of fixed point of a function F where $F(X) = 1 + X * X * \mathsf{int}$. The μ constructor finds this fixed point. When we have $F(X) = \tau$, then if $\mu X.F(X)$ is a fixed point of F, then $F(\mu X.F(X)) = \tau\{\mu X.F(X)/X\}$ should be the "same type" as $\mu X.F(X) = \mu X.\tau$. Thus, we expect the following equation to hold for any type τ, at least to within an isomorphism:

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

In this equation, going from left to right is called *unfolding* while going from right to left is called *folding*.

There are two ways to treat recursive types in a programming language, differing in the way in which the above conversion takes place.

- If the above conversion takes place implicitly, then the types are called *equirecursive* types. Not many languages support equirecursive types. One language that does support them is Modula-3.
- If the conversion above takes place explicitly via fold and unfold, then the resulting type is called *isorecursive* type. In the case of fold,

$$\text{fold}_{\mu X.\tau} : \quad \tau\{\mu X.\tau/X\} \to \mu X.\tau$$

and in the case of unfold

$$\text{unfold}_{\mu X.\tau} : \quad \mu X.\tau \to \tau\{\mu X.\tau/X\}$$

Now we give the introduction and elimination rules for the fold and unfold.

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

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

3 Examples

3.1 Trees

After these definitions, now we are ready to give an example of a function that manipulate these data-structures. We will write a method to compute the sum of all the number stored in the nodes of the tree.

```
let sumtree:tree → int =
    rec f:tree → int
        λt:tree.  let t':1+tree*tree*int = unfold_tree t
                  in
                    case t' of
                         λu:1.  0
                       | λn:tree*tree*int.  (#3 n) + f(#1 n) + f(#2 n)
                  end
```

Here we see that there are actually two types of recursions going on. One is at the term level (rec) and another is at the type level (fold and unfold).

3.2 Natural numbers

Using recursive types, we can encode the type of natural numbers as follows:

$$\text{Nat} = 1 + \text{Nat}$$
$$\text{Nat} = \mu N.(1 + N)$$

In these settings,

$$0 = \text{fold}_{\text{Nat}}(inl_{1+\text{Nat}}(unit))$$
$$1 = \text{fold}_{\text{Nat}}(inl_{1+\text{Nat}}(0))$$

3.3 Self-application

With recursive types, we can construct non-terminating programs without recursive functions. Define SA:

$$SA \stackrel{def}{=} \lambda x.(xx)$$

and note that $\text{fold}_{\mu X.X \to \tau}(\lambda x : \mu X.X \to \tau.(\text{unfold}_{\mu X.X \to \tau} x)x) : (\mu X.X \to \tau) \to \tau$. What is $\Omega = \text{SA SA}$ then? $(\text{unfold}_{\mu X.X \to Y} SA)SA : \tau$. Thus, we can give Ω any type we like. This is sound as a type system because Ω diverges, never disappointing the evaluation context around it. This is unsound if mapped onto a logic since every type is inhabited by Ω, even types which correspond to untrue propositions. This is what happens when we add enough expressive power into programming languages!

3.4 Translation from untyped to typed lambda calculus

With recursive types, we want to show that, now, we are not at a loss for computational power, and the best way to accomplish this is by translation from the untyped to the typed-lambda calculus.

First of all, we need a type satisfying $D \cong D \to D$, which is the type $U \stackrel{\triangle}{=} \mu D.D \to D$. Then the translation is as follows:

$$\begin{aligned} [\![x]\!] &= x && : U \\ [\![e_0\ e_1]\!] &= (\text{unfold}_U [\![e_0]\!])[\![e_1]\!] && : U \\ [\![\lambda x.e]\!] &= \text{fold}_U(\lambda x : U.[\![e]\!]) && : U \end{aligned}$$

4 Type recursion in real languages

What about other languages? Languages, such as SML and C, are *nominal* in that the fixed points corresponding to recursive data types all have explicit names. This allows the necessary fixed point to be taken in a less obtrusive fashion than in our core language. In each of these languages, fixed points are associated with some other type constructor, and fold and unfold are performed at the same time as some necessary operation associated with those types. This correspondence is shown in the following table:

$\lambda^{\to}$	recursive types	fold	unfold
SML	datatypes	constructors	case (pattern matching)
C, C++	structs	new, &	*

An example of a language which takes a more *structural* approach to type recursion is Modula-3, where the following two types can be used interchangeably:

```
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
```

We'll see more on this *equirecursive* approach to recursive types soon.

4.1 Mutual recursion

Most languages support types that can be defined in terms of each other; that is, they are mutually recursive. For example, in SML we might want Node and Edge types for representing graphs:

```
datatype node = Node of edge array
and datatype edge = Edge of node * node
```

To interpret this as a fixed point, we need to take a fixed point over a tuple of types rather than on a single type. We've already seen how to take such fixed points at the term level; at the type level it's similar.

4.2 Closed vs. open recursion

SML provides *closed recursion* on types—closed in the sense that the scope of the fixed point is limited to a sequence of datatype declarations separated by `and`. Some languages, such as Java, provide *open recursion* in which types (classes) can freely refer to other types even if they are in other packages. And cycles are permitted freely. For example, classes A and B can have fields of each other's type:

```
class A {
    ... B ...
}
class B {
    ... A ...
}
```

It may not be possible to understand the meaning of a Java type without looking at every other type in the system. Essentially, the set of types is produced by an implicit fixed point over the whole system, exploiting the nominal type system of Java to know what that fixed point is.

Open recursion can create some interesting problems, however. Consider this example:

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

What does Java do in such static initialization? The result depends on which class is loaded first by the interpreter. If class A is loaded first, then it will temporarily put 0 in the location for x (as B is still not loaded, so $B.y$ is not defined). While loading B, it sees that $A.x = 0$, and makes $B.y = 1$. After that, it backpatches the value of x in class A by putting $A.x = B.y + 1 = 1 + 1 = 2$. Refer to "Eager Class Initialization for Java" by Dexter Kozen and Matthew Stillerman for how to catch circularities in initialization that are missed by this lazy implementation.