1 Introduction

Many programming languages have the ability to define recursive data types. For example, suppose we want to define binary trees with integer data at the nodes. In Java we can write

```
class Tree {
  Tree leftChild, rightChild;
  int data;
}
```

A binary tree is an object of this class. In OCaml we can write

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

These types are *recursive* because they are defined in terms of themselves.

In the typed λ-calculus, we do not yet have any mechanism to define recursive types. We would like the type tree to satisfy

$$\mathsf{tree} = 1 + \mathsf{tree} * \mathsf{tree} * \mathsf{int}, \tag{1}$$

where 1 (aka unit) represents the empty tree; in other words, we would like the set of data objects inhabiting the type tree to be a solution of the equation

$$\alpha = 1 + \alpha * \alpha * \mathsf{int}. \tag{2}$$

However, no such solution exists among the types we have seen so far.

How might we augment our type system to include solutions to recursive type equations such as (2)? There are two basic approaches, called the *equirecursive* and *isorecursive* approach, respectively.

2 Recursive Types as Regular Labeled Trees—The Equirecursive View

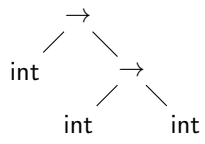
By unwinding (2), we can see that

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

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 (2) for α. This gives a sequence of deeper and deeper finite trees, where each successive tree is a substitution instance of the previous tree.

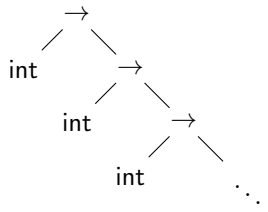
If we take the limit of this process, we have an infinite tree. We can think of this as an infinite labeled graph whose nodes are labeled with the type constructors *, +, int, and 1. This is very much like an ordinary type expression, except that it is infinite. There are no more α's, because we have substituted for all of them all the way down. This infinite tree is a solution of (2), 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 $\rightarrow$, $*$, $+$, 1, 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 $\mathsf{int} \rightarrow \mathsf{int} \rightarrow \mathsf{int}$ 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 1, then any node labeled with that symbol must be a leaf. Within these constraints, the tree may be infinite.[1]

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 unwinding of (2) 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 of the form (2). Suppose we have n type equations in n variables:

$$\alpha_1 = \tau_1, \ \ldots, \ \alpha_n = \tau_n \tag{3}$$

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.

2.1 The μ Constructor

We can specify the infinite solutions to systems of type equations (3) using a finite syntax involving a new type constructor μ, the *fixpoint type constructor*. If we have an equation $\alpha = \tau$ such that the right-hand side is not α, then there is a unique solution, which is a finite or infinite regular tree. The solution will be infinite if α occurs in τ and will be finite (in fact it will just be τ) if α does not occur in τ. We denote this unique solution by $\mu\alpha.\,\tau$.

Syntactically, μ acts as a binding operator in type expressions as λ does in λ-terms, with the same notions of scope, free and bound variables, α-conversion, and safe substitution.

[1] For a more formal account of this construction, see Handout 2A, §1.2 (27 January).

Since $\mu\alpha.\,\tau$ is a solution to $\alpha = \tau$, we have

$$\mu\alpha.\,\tau = \tau\{\mu\alpha.\,\tau/\alpha\}. \tag{4}$$

For example, to get a tree type satisfying (1), we can define

$$\mathsf{tree} \stackrel{\triangle}{=} \mu\alpha.\,1 + \alpha * \alpha * \mathsf{int}.$$

The desired equation (1) is just (4) for this case.

The solutions $\sigma_1, \ldots, \sigma_n$ to any finite system of the form (3) 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:

```
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 fixpoint when assigning types.

2.2 Typing Rules

In the equirecursive view, since $\mu\alpha.\,\tau = \tau\{\mu\alpha.\,\tau/\alpha\}$, the typing rules are quite simple:

$$\frac{\Gamma \vdash e : \tau\{\mu\alpha.\,\tau/\alpha\}}{\Gamma \vdash e : \mu\alpha.\,\tau}\ (\mu\text{-intro}) \qquad\qquad \frac{\Gamma \vdash e : \mu\alpha.\,\tau}{\Gamma \vdash e : \tau\{\mu\alpha.\,\tau/\alpha\}}\ (\mu\text{-elim})$$

Equivalently, we can just allow substitution of equals for equals in type expressions.

3 Folding and Unfolding—The Isorecursive View

There is another approach to recursive types, the *isorecursive* approach. Here we do not have any infinite types, but rather the expression $\mu\alpha.\,\tau$ is itself a type. In this approach, $\mu\alpha.\,\tau$ and $\tau\{\mu\alpha.\,\tau/\alpha\}$ are considered different (but isomorphic) types.

The step of substituting $\mu\alpha.\,\tau$ for α in τ is called *unfolding*, and the reverse operation is called *folding*. The conversion of elements between these two types is accomplished by explicit fold and unfold operations.

$$\mathsf{unfold}_{\mu\alpha.\,\tau} : \mu\alpha.\,\tau \ \rightarrow\ \tau\{\mu\alpha.\,\tau/\alpha\} \qquad\qquad \mathsf{fold}_{\mu\alpha.\,\tau} : \tau\{\mu\alpha.\,\tau/\alpha\} \ \rightarrow\ \mu\alpha.\,\tau$$

(we suppress the subscripts when there is no ambiguity). In this view, the equality symbol in (4) is not really an equality, but just an isomorphism.

3.1 Typing Rules

In the isorecursive view, the typing rules consist of a pair of introduction and elimination rules for μ-types that explicitly mention fold and unfold:

$$\frac{\Gamma \vdash e : \tau\{\mu\alpha.\,\tau/\alpha\}}{\Gamma \vdash \mathsf{fold}\ e : \mu\alpha.\,\tau}\ (\mu\text{-intro}) \qquad \frac{\Gamma \vdash e : \mu\alpha.\,\tau}{\Gamma \vdash \mathsf{unfold}\ e : \tau\{\mu\alpha.\,\tau/\alpha\}}\ (\mu\text{-elim})$$

3.2 Structural Operational Semantics

In the isorecursive view, we also need to augment the operational semantics. We only need one rule:

$$\mathsf{unfold}\ (\mathsf{fold}\ e)\ \rightarrow\ e.$$

Intuitively, to access the data in a recursive type $\mu\alpha.\,\tau$, we need to unfold it first; but the only way that values of type $\mu\alpha.\,\tau$ could have been created in the first place is via a fold.

3.3 An Example

Suppose we want to write a program to add a list of numbers. The list type is a recursive type, which we can define as

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

The type 1 (aka unit) represents the empty list. It has a single inhabitant null. The other case is for a nonempty list consisting of a head, which is an int, followed by the tail of the list, which is an intlist; that is, int * intlist.

Now we can write a function sum to add up an intlist. This will be a recursive function, so we will need to take a fixpoint.

$$\mathsf{let\ sum} = \mathsf{rec}\ f : \mathsf{intlist} \rightarrow \mathsf{int}.\,\lambda\ell : \mathsf{intlist}.\ \ldots\ \mathsf{in}\ \ldots$$

In the body of this function, we would like to do a match on the intlist ℓ. But to do a match, we need a sum type, and ℓ is a μ-type, so we will have to unfold it first. (OCaml does this automatically when it sees a match.) So the body would be

$$\begin{array}{l}\mathsf{match\ unfold}\ \ell : 1 + \mathsf{int} * \mathsf{intlist\ with}\\ \quad \mid \lambda u : 1.\,0\\ \quad \mid \lambda p : \mathsf{int} * \mathsf{intlist}.\,(\mathsf{fst}\ p) + f\ (\mathsf{snd}\ p)\end{array}$$

This is just the same code that you would write in OCaml, except we have broken out some of the things that Ocaml hides for you. In particular, we have explicitly shown the recursion in the definition of the intlist type and the unfold that is needed to get the exploded view of the type.

4 Equirecursive vs. Isorecursive

Programming languages deal with recursive types in different ways. Java and Modula-3 take the equirecursive approach, in which the folded and unfolded types are considered equal, and the fold/unfold operations are just the identity functions. Recursive types and their unfoldings are fully substitutable for each other.

```
class E {
  String x;
  E e;
  public String toString() {
    return e.e.e.e.e.e.e.e.e.e.e.e.e.e.e.x;
  }
}
```

On the other hand, the ML family, CLU, and C use isorecursive types, in which $\mu\alpha.\,\tau$ and $\tau\{\mu\alpha.\,\tau/\alpha\}$ are considered different (but isomorphic) types, and the casting operators fold and unfold are required to go between them. CLU uses up and down instead of fold and unfold. In OCaml, the unfold operator is performed automatically and implicitly by the match and let statements and the pattern-matching mechanism. The type constructors in a recursive datatype definition, applied to arguments, act as fold operations.

```
# type tree = Leaf | Node of tree * tree * int;;
type tree = Leaf | Node of tree * tree * int
# Node (Leaf, Leaf, 4);;
- : tree = Node (Leaf, Leaf, 4)
```

5 Numbers as a Recursive Type

We started with primitive types 1, boolean, and int. We have already seen that the type boolean can be represented as $1 + 1$ with values false and true represented by inL null and inR null, respectively.

Now that we have recursive types, we no longer need to take int as primitive, but we can define it as a recursive type. A natural number is either 0 or a successor of a natural number. Thus we can take

$$\mathsf{nat} \triangleq \mu\alpha.\,1 + \alpha \qquad 0 \triangleq \mathsf{fold}\ (\mathsf{inL}\ \mathsf{null}) \qquad 1 \triangleq \mathsf{fold}\ (\mathsf{inR}\ 0) \qquad 2 \triangleq \mathsf{fold}\ (\mathsf{inR}\ 1),$$

and so on. Here $\mathsf{fold} = \mathsf{fold}_{\mathsf{nat}}$, $\mathsf{inL} = \mathsf{inL}_{1+\mathsf{nat}}$, and $\mathsf{inR} = \mathsf{inR}_{1+\mathsf{nat}}$.

We can use the recursive type nat to code up all of the usual arithmetic, and all these operations are well-typed. For example, the successor function would be

$$(\lambda x : \mathsf{nat}.\,\mathsf{fold}\ (\mathsf{inR}\ x)) : \mathsf{nat} \to \mathsf{nat}.$$

So all we really need as primitive types and type constructors are 1 (unit), recursive types, products, and sums, and of course $\to$. With these we can build all the other types like natural numbers, integers, lists, trees, floating point numbers, and so on.

6 Self-Application and Ω

Recall the *paradoxical combinator* Ω defined by

$$\omega \triangleq \lambda x.\,xx \qquad\qquad \Omega \triangleq \omega\omega.$$

We can now give these terms recursive types, provided we insert some folding and unfolding. Since x is applied as a function, it must have some kind of function type, say $\sigma \to \tau$. But since it is applied to itself as an argument, it must also have type σ. This seems to indicate that the type of x must satisfy the equation $\sigma = \sigma \to \tau$. The recursive type $\mu\alpha.\,(\alpha \to \tau)$ appears to be in order (here τ can be anything).

To actually apply x to x, we have to unfold it. The type of $\mathsf{unfold}\ x$ is

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

This is a function with domain $\mu\alpha.\,(\alpha \to \tau)$, which is the type of x, so we can apply it to x. The type of the result $(\mathsf{unfold}\ x)\ x$ is τ. Thus the fully typed ω term is

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

If we now fold this, we get

$$\mathsf{fold}\ \omega : \mu\alpha.\,(\alpha \to \tau).$$

Therefore, we can apply ω as a function to $\mathsf{fold}\ \omega$, and the result is

$$\omega\ (\mathsf{fold}\ \omega) : \tau.$$

This is the same as the original Ω term, but with explicit folding and unfolding.

We can do this in OCaml:

```
# type u = Fold of (u -> u);;
type u = Fold of (u -> u)
# let omega = fun x -> match x with Fold f -> f x;;
val omega : u -> u = <fun>
# omega (Fold omega);;
```

(*and at this point it will just sit there forever until you hit control-c*)

```
^CInterrupted.
```

So we were finally able to introduce nontermination. But the point is that it passed typechecking, so the program was well-typed.

7 Untyped to Typed λ-Calculus

With recursive types, we can type everything in the pure untyped λ-calculus. Every λ-term can be applied as a function to any other λ-term, so every λ-term (with appropriate folds and unfolds inserted) has type $U \triangleq \mu\alpha.\,\alpha \to \alpha$. The translation is

$$\begin{aligned}
\mathcal{D}[\![x]\!] &\triangleq x \\
\mathcal{D}[\![e_0\,e_1]\!] &\triangleq (\mathsf{unfold}\ \mathcal{D}[\![e_0]\!])\ \mathcal{D}[\![e_1]\!] \\
\mathcal{D}[\![\lambda x.\,e]\!] &\triangleq \mathsf{fold}\ \lambda x : U.\,\mathcal{D}[\![e]\!].
\end{aligned}$$

Note that every untyped term maps to a term of type U.