What is the type of a cons (singly-linked) list of integers? How about a binary tree of integers? In a language like Java, we might write

```
class IntList {
  int data;
  IntList next;
}
```

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

In OCaml, we might write

```
type intList = Empty | Cons of int * intList
type intTree = Leaf | Node of int * intTree * intTree
```

How would we write these types in a language we have seen before, such as STLC with extra datatypes as defined in Lecture 18? We might hope to write something like

$$\mathsf{intList} \;=\; \mathsf{unit} + (\mathsf{int} \times \mathsf{intList})$$

as this would precisely mirror the OCaml definition above. However, that is not a well-defined type in our language; it is self-referential. Even if we take it to be an equation that intList must satisfy, we do not yet have the constructs to build a type satisfying that equation. Indeed, we will need a new construct in our space of types to handle it.

1 The μ Constructor

As in a variety of previous contexts, to solve the equation $\alpha = \mathsf{unit} + (\mathsf{int} \times \alpha)$, we need to find a fixed point of the function $\mathcal{T}\ \alpha = \mathsf{unit} + (\mathsf{int} \times \alpha)$. To handle this, we introduce a *fixed point type constructor* $\mu\alpha.\,\tau$ that defines the *least fixed point* of the function $\mathcal{T}\ \alpha = \tau$. Such a fixed point exists and is unique as long as $\tau \neq \alpha$. Note that if τ does not reference α, then the result is simply τ.

To make this work, we use type variables, just as we did with polymorphism in Lecture 20, and the type constructor $\mu\alpha.\,\tau$ functions as a binder for the variable α.

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

$$\mu\alpha.\,\tau \;=\; \tau[\alpha \mapsto \mu\alpha.\,\tau].$$

This construct therefore allows us to build types like the intList described above. We can define

$$\mathsf{intList} \;\triangleq\; \mu\alpha.\,\mathsf{unit} + (\mathsf{int} \times \alpha).$$

To see why, we can unroll this one level, noting that

$$\mathsf{intList} \;=\; \mathsf{unit} + (\mathsf{int} \times (\mu\alpha.\,\mathsf{unit} + (\mathsf{int} \times \alpha))) \;=\; \mathsf{unit} + (\mathsf{int} \times \mathsf{intList}).$$

Recursive types also plays nicely with polymorphism. If we wanted to define a polymorphic list type $\forall\alpha.\,\mathsf{list}\ \alpha$, we could do that simply as

$$\forall\alpha.\,\mathsf{list}\ \alpha \;\triangleq\; \forall\alpha.\,\mu\beta.\,\mathsf{unit} + (\alpha \times \beta).$$

This structure allows us to define classic polymorphic list operators like map, filter, and fold.

The μ constructor is even sufficient to build mutually-recursive types. For example, if $\alpha_1 = \tau_1$ and $\alpha_2 = \tau_2$ where both τ_1 and τ_2 may refer to either or both of α_1 and α_2, we can define the mutually-recursive types σ_1 and σ_2 by:

$$\sigma_1 \;\triangleq\; \mu\alpha_1.\,(\tau_1[\alpha_2 \mapsto \mu\alpha_2.\,\tau_2]) \qquad \sigma_2 \;\triangleq\; \mu\alpha_2.\,(\tau_2[\alpha_1 \mapsto \mu\alpha_1.\,\tau_1]).$$

2 Recursive Types in a Language

To use the fixed point type constructor in a type system, we need rules for it. There are two approaches to using recursive types in languages: *equirecursive* and *isorecursive*.

2.1 Equirecursive Types

With equirecursive types, we consider $\mu\alpha.\,\tau$ and $\tau[\alpha \mapsto \mu\alpha.\,\tau]$ to be *the same type*. A term of one may freely be used as the other. To support this in the type system, we would include the following two rules.

$$[\mu\text{-INTRO}]\ \frac{\Gamma \vdash e : \tau[\alpha \mapsto \mu\alpha.\,\tau]}{\Gamma \vdash e : \mu\alpha.\,\tau} \qquad [\mu\text{-ELIM}]\ \frac{\Gamma \vdash e : \mu\alpha.\,\tau}{\Gamma \vdash e : \tau[\alpha \mapsto \mu\alpha.\,\tau]}$$

This is the approach taken by languages like Haskell and OCaml. It is generally simpler and easier to program with, but it can be more challenging to reason about formally.

2.2 Isorecursive Types

The other option is isorecursive types, where $\mu\alpha.\,\tau$ and $\tau[\alpha \mapsto \mu\alpha.\,\tau]$ are not considered the same type, but merely isomorphic. That means we can convert terms of one into terms of the other in both directions, but we must do so explicitly. The explicit conversions are known as *fold* and *unfold*, with the types:

$$\mathsf{fold}_{\mu\alpha.\tau} : \tau[\alpha \mapsto \mu\alpha.\,\tau] \to \mu\alpha.\,\tau \qquad \mathsf{unfold}_{\mu\alpha.\tau} : \mu\alpha.\,\tau \to \tau[\alpha \mapsto \mu\alpha.\,\tau]$$

We will omit the subscripts on these operations when there is no ambiguity.

To use folding and unfolding in programs, we add fold and unfold to our language syntax, where fold serves as an introduction form for a recursive type and unfold serves as the corresponding elimination form.

$$\begin{array}{rcl} e & ::= & \cdots \mid \mathsf{fold}\ e \mid \mathsf{unfold}\ e \\ v & ::= & \cdots \mid \mathsf{fold}\ v \end{array}$$

As always when adding new syntactic forms, we need to define their operational semantics and provide typing rules. The typing rules follow directly from the idea that these are isomorphisms on recursive types.

$$[\textsc{Fold}]\ \frac{\Gamma \vdash e : \tau[\alpha \mapsto \mu\alpha.\,\tau]}{\Gamma \vdash \mathsf{fold}\ e : \mu\alpha.\,\tau} \qquad [\textsc{Unfold}]\ \frac{\Gamma \vdash e : \mu\alpha.\,\tau}{\Gamma \vdash \mathsf{unfold}\ e : \tau[\alpha \mapsto \mu\alpha.\,\tau]}$$

The semantic rules for these new operations consider them both evaluation contexts, and the introduction and elimination forms cancel each other, as normal.

$$E \quad ::= \quad \cdots \mid \mathsf{fold}\ E \mid \mathsf{unfold}\ E \qquad \frac{}{\mathsf{unfold}\ (\mathsf{fold}\ v) \longrightarrow v}$$

3 Using Recursive Types

To see how we can use recursive types to build interesting datatypes, we will look at some examples.

3.1 Lists

Perhaps the simplest place to see the value of recursive types is in defining lists. In Section 1 we saw that we could define the type

$$\mathsf{intList} \triangleq \mu\alpha.\, \mathsf{unit} + (\mathsf{int} \times \alpha).$$

Using isorecursive types, this guides us in how to define the basic list operators. For instance, we want empty to have type intList and correspond to having no elements. Using the injection functions for the sum type, we note that

$$\begin{aligned} () &: \mathsf{unit} \\ \mathsf{inl}\ () &: \mathsf{unit} + (\mathsf{int} \times \mu\alpha.\, \mathsf{unit} + (\mathsf{int} \times \alpha)) \\ \mathsf{fold}\ (\mathsf{inl}\ ()) &: \mu\alpha.\, \mathsf{unit} + (\mathsf{int} \times \alpha) = \mathsf{intList} \end{aligned}$$

Therefore, we can define

$$\mathsf{empty} \triangleq \mathsf{fold}\ (\mathsf{inl}\ ()).$$

Similarly, to define cons we will need to fold an injection of a pair.

$$\mathsf{cons}\ h\ t \triangleq \mathsf{fold}\ (\mathsf{inr}\ (h, t))$$

To access elements of the list, we need to *unfold* the list and then match.

$$\begin{aligned} \mathsf{isempty}\ l &\triangleq \mathsf{match}\ (\mathsf{unfold}\ l)\ \mathsf{with}\ \mathsf{inl}(\_).\, \mathsf{true} \mid \mathsf{inr}(\_).\, \mathsf{false} \\ \mathsf{head}\ l &\triangleq \mathsf{match}\ (\mathsf{unfold}\ l)\ \mathsf{with}\ \mathsf{inl}(\_).\, \mathsf{error} \mid \mathsf{inr}(p).\, \mathsf{proj}_1\ p \\ \mathsf{tail}\ l &\triangleq \mathsf{match}\ (\mathsf{unfold}\ l)\ \mathsf{with}\ \mathsf{inl}(\_).\, \mathsf{error} \mid \mathsf{inr}(p).\, \mathsf{proj}_2\ p \end{aligned}$$

3.2 Numbers as Recursive Types

The most basic types in many contexts are unit, nat, and bool. We have already seen how to encode bool using unit and sum types. With the presence of recursive types, we no longer need a primitive type for nat.

A natural number is either 0 or the successor of a natural number. We can therefore define

$$\mathsf{nat} \triangleq \mu\alpha.\, \mathsf{unit} + \alpha \qquad 0 \triangleq \mathsf{fold}\ (\mathsf{inl}\ ()) \qquad 1 \triangleq \mathsf{fold}\ (\mathsf{inr}\ 0) \qquad 2 \triangleq \mathsf{fold}\ (\mathsf{inr}\ 1) \qquad \cdots$$

Using this encoding, it is straightforward to define successor and predecessor functions.

$$\begin{aligned} \mathsf{succ}\ n &\triangleq \mathsf{fold}\ (\mathsf{inr}\ n) \\ \mathsf{pred}\ n &\triangleq \mathsf{match}\ (\mathsf{unfold}\ n)\ \mathsf{with}\ \mathsf{inl}(\_).\, 0 \mid \mathsf{inr}(m).\, m \end{aligned}$$

So all we really need as primitive types and type constructors are 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.

3.3 Self-Application and Ω

Recall the infinite loop Ω defined by $\omega = \lambda x.\, x\ x$ and $\Omega = \omega\ \omega$.

We saw before that it was impossible to give a type to ω (and therefore Ω) in STLC. The reason for this was we needed types τ_1 and τ_2 such that $\tau_1 = \tau_1 \to \tau_2$. In STLC that was impossible. With polymorphism, we could give ω the (very strange) type $\forall\beta.\, (\forall\alpha.\, \alpha) \to \beta$, but we still could not self-apply it to get an infinite loop.

However, with recursive types, there is a more straightforward way to type ω: we can give x type $\mu\alpha.\,\alpha \to \tau$ for any type τ! To actually apply x to something we need to unfold it, and the resulting type 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. This insight lets us define

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

which has type $(\mu\alpha.\,\alpha \to \tau) \to \tau$. If we fold that, we get back something of the same type as the argument, thereby allowing us to define

$$\Omega \triangleq \omega\ (\mathsf{fold}\ \omega).$$

This is a well-typed term that will never terminate. It does not immediately step to itself because of the folding and unfolding involved, but it doesn't take very long.

$$\begin{aligned}\Omega = \omega\ (\mathsf{fold}\ \omega) &= (\lambda x{:}\mu\alpha.\,\alpha \to \tau.\,(\mathsf{unfold}\ x)\ x)\ (\mathsf{fold}\ \omega)\\ &\longrightarrow (\mathsf{unfold}\ (\mathsf{fold}\ \omega))\ (\mathsf{fold}\ \omega)\\ &\longrightarrow \omega\ (\mathsf{fold}\ \omega) = \Omega\end{aligned}$$

4 Untyped to Typed λ-Calculus

Recursive types not only allow for Ω, they bring back the full expressive power of untyped λ-calculus. To prove this, it is possible to translate any untyped λ-calculus term into a well-typed term with the *universal type*: $U \triangleq \mu\alpha.\,\alpha \to \alpha$. This type satisfies the equation $U = U \to U$. Since all pure λ-calculus terms are functions, we can represent them all as this type.

The translation is as follows.

$$\begin{aligned}\mathcal{D}[\![x]\!] &\triangleq x\\ \mathcal{D}[\![e_1\ e_2]\!] &\triangleq (\mathsf{unfold}\ \mathcal{D}[\![e_1]\!])\ \mathcal{D}[\![e_2]\!]\\ \mathcal{D}[\![\lambda x.\,e]\!] &\triangleq \mathsf{fold}\ (\lambda x{:}U.\,\mathcal{D}[\![e]\!])\end{aligned}$$

You can prove by induction that, for any untyped λ-calculus term e, $\Gamma \vdash \mathcal{D}[\![e]\!] : U$ where Γ maps all free variables in e to U.