# CS 4110

# Programming Languages & Logics

## Lecture 27
## Recursive Types

# Recursive Types

Many languages support data types that refer to themselves:

Java

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

# Recursive Types

Many languages support data types that refer to themselves:

Java

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

OCaml

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

# Recursive Types

Many languages support data types that refer to themselves:

**Java**

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

**OCaml**

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

**$\lambda$-calculus?**

$$tree = \textbf{unit} + \textbf{int} \times tree \times tree$$

# Recursive Type Equations

We would like **tree** to be a solution of the equation:

$$\alpha = \textbf{unit} + \textbf{int} \times \alpha \times \alpha$$

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

# Unwinding Equations

We could *unwind* the equation:

$$\alpha = \textbf{unit} + \textbf{int} \times \alpha \times \alpha$$

# Unwinding Equations

We could *unwind* the equation:

$$\begin{aligned}\alpha &= \textbf{unit} + \textbf{int} \times \alpha \times \alpha \\ &= \textbf{unit} + \textbf{int} \times \\ &\qquad (\textbf{unit} + \textbf{int} \times \alpha \times \alpha) \times \\ &\qquad (\textbf{unit} + \textbf{int} \times \alpha \times \alpha)\end{aligned}$$

# Unwinding Equations

We could *unwind* the equation:

$$
\begin{aligned}
\alpha &= \textbf{unit} + \textbf{int} \times \alpha \times \alpha \\
&= \textbf{unit} + \textbf{int} \times \\
&\qquad\qquad (\textbf{unit} + \textbf{int} \times \alpha \times \alpha) \times \\
&\qquad\qquad (\textbf{unit} + \textbf{int} \times \alpha \times \alpha) \\
&= \textbf{unit} + \textbf{int} \times \\
&\qquad\qquad (\textbf{unit} + \textbf{int} \times \\
&\qquad\qquad\qquad (\textbf{unit} + \textbf{int} \times \alpha \times \alpha) \times \\
&\qquad\qquad\qquad (\textbf{unit} + \textbf{int} \times \alpha \times \alpha)) \times \\
&\qquad\qquad (\textbf{unit} + \textbf{int} \times \\
&\qquad\qquad\qquad (\textbf{unit} + \textbf{int} \times \alpha \times \alpha) \times \\
&\qquad\qquad\qquad (\textbf{unit} + \textbf{int} \times \alpha \times \alpha))
\end{aligned}
$$

# Unwinding Equations

We could *unwind* the equation:

$$
\begin{aligned}
\alpha &= \textbf{unit} + \textbf{int} \times \alpha \times \alpha \\
&= \textbf{unit} + \textbf{int} \times \\
&\qquad\quad (\textbf{unit} + \textbf{int} \times \alpha \times \alpha) \times \\
&\qquad\quad (\textbf{unit} + \textbf{int} \times \alpha \times \alpha) \\
&= \textbf{unit} + \textbf{int} \times \\
&\qquad\quad (\textbf{unit} + \textbf{int} \times \\
&\qquad\qquad\quad (\textbf{unit} + \textbf{int} \times \alpha \times \alpha) \times \\
&\qquad\qquad\quad (\textbf{unit} + \textbf{int} \times \alpha \times \alpha)) \times \\
&\qquad\quad (\textbf{unit} + \textbf{int} \times \\
&\qquad\qquad\quad (\textbf{unit} + \textbf{int} \times \alpha \times \alpha) \times \\
&\qquad\qquad\quad (\textbf{unit} + \textbf{int} \times \alpha \times \alpha)) \\
&= \cdots
\end{aligned}
$$

# Unwinding Equations

We could *unwind* the equation:

$$
\begin{aligned}
\alpha &= \textbf{unit} + \textbf{int} \times \alpha \times \alpha \\
&= \textbf{unit} + \textbf{int}\times \\
&\qquad\qquad (\textbf{unit} + \textbf{int} \times \alpha \times \alpha)\times \\
&\qquad\qquad (\textbf{unit} + \textbf{int} \times \alpha \times \alpha) \\
&= \textbf{unit} + \textbf{int}\times \\
&\qquad\qquad (\textbf{unit} + \textbf{int}\times \\
&\qquad\qquad\qquad (\textbf{unit} + \textbf{int} \times \alpha \times \alpha)\times \\
&\qquad\qquad\qquad (\textbf{unit} + \textbf{int} \times \alpha \times \alpha))\times \\
&\qquad\qquad (\textbf{unit} + \textbf{int}\times \\
&\qquad\qquad\qquad (\textbf{unit} + \textbf{int} \times \alpha \times \alpha)\times \\
&\qquad\qquad\qquad (\textbf{unit} + \textbf{int} \times \alpha \times \alpha)) \\
&= \cdots
\end{aligned}
$$

If we take the limit of this process, we have an infinite tree.

# Infinite Types

Think of this as an infinite labeled graph whose nodes are labeled with the type constructors $\times$, $+$, **int**, and **unit**.

This infinite tree is a solution of our equation, and this is what we take as the type **tree**.

# $\mu$ Types

We'll specify potentially-infinite solutions to type equations using a finite syntax based on the *fixed-point type constructor* $\mu$.

$$\mu\alpha.\, \tau$$

# $\mu$ Types

We'll specify potentially-infinite solutions to type equations using a finite syntax based on the *fixed-point type constructor* $\mu$.

$$\mu\alpha.\,\tau$$

Here's a **tree** type satisfying our original equation:

$$\textbf{tree} \triangleq \mu\alpha.\,\textbf{unit} + \textbf{int} \times \alpha \times \alpha.$$

# Static Semantics (Equirecursive)

We'll define two treatments of recursive types. With *equirecursive types*, a recursive type is equal to its unfolding:

$\mu\alpha.\,\tau$ is a solution to $\alpha = \tau$, so:

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

# Static Semantics (Equirecursive)

We'll define two treatments of recursive types. With *equirecursive types*, a recursive type is equal to its unfolding:

$\mu\alpha.\,\tau$ is a solution to $\alpha = \tau$, so:

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

Two typing rules let us switch between folded and unfolded:

$$\frac{\Gamma \vdash e : \tau\{\mu\alpha.\,\tau/\alpha\}}{\Gamma \vdash e : \mu\alpha.\,\tau}\ \mu\text{-INTRO}$$

$$\frac{\Gamma \vdash e : \mu\alpha.\,\tau}{\Gamma \vdash e : \tau\{\mu\alpha.\,\tau/\alpha\}}\ \mu\text{-ELIM}$$

# Isorecursive Types

Alternatively, *isorecursive types* avoid infinite type trees.

The type $\mu\alpha.\,\tau$ is distinct but transformable to and from $\tau\{\mu\alpha.\,\tau/\alpha\}$.

# Isorecursive Types

Alternatively, *isorecursive types* avoid infinite type trees.

The type $\mu\alpha.\,\tau$ is distinct but transformable to and from $\tau\{\mu\alpha.\,\tau/\alpha\}$.

Converting between the two uses explicit **fold** and **unfold** operations:

$$\begin{array}{rcrcl} \textbf{unfold}_{\mu\alpha.\,\tau} & : & \mu\alpha.\,\tau & \rightarrow & \tau\{\mu\alpha.\,\tau/\alpha\} \\ \textbf{fold}_{\mu\alpha.\,\tau} & : & \tau\{\mu\alpha.\,\tau/\alpha\} & \rightarrow & \mu\alpha.\,\tau \end{array}$$

# Static Semantics (Isorecursive)

The typing rules introduce and eliminate $\mu$-types:

$$\frac{\Gamma \vdash e : \tau\{\mu\alpha.\,\tau/\alpha\}}{\Gamma \vdash \textbf{fold}\ e : \mu\alpha.\,\tau}\ \mu\text{-INTRO}$$

$$\frac{\Gamma \vdash e : \mu\alpha.\,\tau}{\Gamma \vdash \textbf{unfold}\ e : \tau\{\mu\alpha.\,\tau/\alpha\}}\ \mu\text{-ELIM}$$

# Dynamic Semantics

We also need to augment the operational semantics:

$$\frac{}{\textbf{unfold}\ (\textbf{fold}\ e) \to e}$$

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

# Example

Here's a recursive type for lists of numbers:

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

# Example

Here's a recursive type for lists of numbers:

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

Here's how to add up the elements of an **intlist**:

$$
\begin{array}{l}
\text{let sum} = \\
\quad \text{fix}\,(\lambda f{:}\,\textbf{intlist} \to \textbf{intlist} \\
\qquad\quad \lambda l{:}\,\textbf{intlist}.\ \text{case}\ \textbf{unfold}\ \ell\ \text{of} \\
\qquad\qquad\qquad (\lambda u : \textbf{unit}.\, 0) \\
\qquad\qquad\quad\ \mid (\lambda p : \textbf{int} \times \textbf{intlist}.\, (\#1\, p) + f\,(\#2\, p)))
\end{array}
$$

# Encoding Numbers

Recursive types let us encode the natural numbers!

# Encoding Numbers

Recursive types let us encode the natural numbers!

A natural number is either $0$ or the successor of a natural number:

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

# Encoding Numbers

Recursive types let us encode the natural numbers!

A natural number is either $0$ or the successor of a natural number:

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

$$0 \triangleq \textbf{fold}\,(\mathsf{inl}_{\textbf{unit}+\textbf{nat}}\,())$$

# Encoding Numbers

Recursive types let us encode the natural numbers!

A natural number is either $0$ or the successor of a natural number:

$$
\begin{aligned}
\textbf{nat} &\triangleq \mu\alpha.\,\textbf{unit} + \alpha \\
0 &\triangleq \textbf{fold}\ (\mathsf{inl}_{\textbf{unit}+\textbf{nat}}\ ()) \\
1 &\triangleq \textbf{fold}\ (\mathsf{inr}_{\textbf{unit}+\textbf{nat}}\ 0) \\
2 &\triangleq \textbf{fold}\ (\mathsf{inr}_{\textbf{unit}+\textbf{nat}}\ 1), \\
&\vdots
\end{aligned}
$$

# Encoding Numbers

Recursive types let us encode the natural numbers!

A natural number is either $0$ or the successor of a natural number:

$$\begin{aligned} \textbf{nat} &\triangleq \mu\alpha.\, \textbf{unit} + \alpha \\ 0 &\triangleq \textbf{fold}\;(\mathsf{inl}_{\textbf{unit}+\textbf{nat}}\;()) \\ 1 &\triangleq \textbf{fold}\;(\mathsf{inr}_{\textbf{unit}+\textbf{nat}}\;0) \\ 2 &\triangleq \textbf{fold}\;(\mathsf{inr}_{\textbf{unit}+\textbf{nat}}\;1), \\ &\vdots \end{aligned}$$

The successor function has type $\textbf{nat} \to \textbf{nat}$:

$$(\lambda x : \textbf{nat}.\, \textbf{fold}\;(\mathsf{inr}_{\textbf{unit}+\textbf{nat}}\;x))$$

# Self-Application and $\Omega$

Recall $\Omega$ defined as:

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

$\Omega$ was impossible to type... until now!

# Self-Application and $\Omega$

Recall $\Omega$ defined as:

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

$\Omega$ was impossible to type... until now!

$x$ is a function. Let's say it has the type $\sigma \to \tau$.

# Self-Application and $\Omega$

Recall $\Omega$ defined as:

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

$\Omega$ was impossible to type... until now!

$x$ is a function. Let's say it has the type $\sigma \to \tau$.

$x$ is used as the argument to this function, so it must have type $\sigma$.

# Self-Application and $\Omega$

Recall $\Omega$ defined as:

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

$\Omega$ was impossible to type... until now!

$x$ is a function. Let's say it has the type $\sigma \to \tau$.

$x$ is used as the argument to this function, so it must have type $\sigma$.

So let's write a type equation:

$$\sigma = \sigma \to \tau$$

# Self-Application and $\Omega$

Putting these pieces together, the fully typed $\omega$ term is:

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

# Self-Application and $\Omega$

Putting these pieces together, the fully typed $\omega$ term is:

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

The type of $\omega$ is $(\mu\alpha.\,(\alpha \to \tau)) \to \tau$.

So the type of **fold** $\omega$ is $\mu\alpha.\,(\alpha \to \tau)$.

# Self-Application and $\Omega$

Putting these pieces together, the fully typed $\omega$ term is:

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

The type of $\omega$ is $(\mu\alpha.\, (\alpha \to \tau)) \to \tau$.

So the type of **fold** $\omega$ is $\mu\alpha.\, (\alpha \to \tau)$.

Now we can define $\Omega = \omega\ (\textbf{fold}\ \omega)$. It has type $\tau$.

# Self-Application and $\Omega$

We can even write $\omega$ 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);;
...runs forever until you hit control-c
```

# Encoding $\lambda$-Calculus

With recursive types, we can type everything in the untyped lambda calculus!

# Encoding $\lambda$-Calculus

With recursive types, we can type everything in the untyped lambda calculus!

Every $\lambda$-term can be applied as a function to any other $\lambda$-term. So let's define an "untyped" type:

$$U \triangleq \mu\alpha.\, \alpha \to \alpha$$

# Encoding $\lambda$-Calculus

With recursive types, we can type everything in the untyped lambda calculus!

Every $\lambda$-term can be applied as a function to any other $\lambda$-term. So let's define an "untyped" type:

$$U \triangleq \mu\alpha.\, \alpha \to \alpha$$

The full translation is:

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

Every untyped term maps to a term of type $U$.