Announcements

• My office hours are at the normal time today but canceled on Monday

• Guest lecture by Seung Hee Han on Monday
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

\[-\text{calculus}\]

tree = unit + int + tree
Recursive Types

Many languages support data types that refer to themselves:

Java

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

OCaml

```ocaml
type tree = Leaf | Node of tree * tree * int
```
Recursive Types

Many languages support data types that refer to themselves:

**Java**

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

**OCaml**

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

**$\lambda$-calculus?**

\[
\text{tree} = \text{unit} + \text{int} \times \text{tree} \times \text{tree}
\]
Recursive Type Equations

We would like \texttt{tree} to be a solution of the equation:

\[
\alpha = \texttt{unit} + \texttt{int} \times \alpha \times \alpha
\]

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

We could *unwind* the equation:

\[ \alpha = \text{unit} + \text{int} \times \alpha \times \alpha \]
Unwinding Equations

We could *unwind* the equation:

\[
\alpha = \text{unit} + \text{int} \times \alpha \times \alpha \\
= \text{unit} + \text{int} \times \\
\quad (\text{unit} + \text{int} \times \alpha \times \alpha) \times \\
\quad (\text{unit} + \text{int} \times \alpha \times \alpha)
\]
Unwinding Equations

We could *unwind* the equation:

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

$$= \text{unit} + \text{int} \times$$

$$(\text{unit} + \text{int} \times \alpha \times \alpha) \times$$

$$(\text{unit} + \text{int} \times \alpha \times \alpha)$$

$$= \text{unit} + \text{int} \times$$

$$(\text{unit} + \text{int} \times$$

$$(\text{unit} + \text{int} \times \alpha \times \alpha) \times$$

$$(\text{unit} + \text{int} \times \alpha \times \alpha)) \times$$

$$(\text{unit} + \text{int} \times$$

$$(\text{unit} + \text{int} \times \alpha \times \alpha) \times$$

$$(\text{unit} + \text{int} \times \alpha \times \alpha))$$

If we take the limit of this process, we have an infinite tree.
We could *unwind* the equation:

\[ \alpha = \text{unit} + \text{int} \times \alpha \times \alpha \]
\[ = \text{unit} + \text{int} \times \]
\[ (\text{unit} + \text{int} \times \alpha \times \alpha) \times \]
\[ (\text{unit} + \text{int} \times \alpha \times \alpha) \]
\[ = \text{unit} + \text{int} \times \]
\[ (\text{unit} + \text{int} \times \]
\[ (\text{unit} + \text{int} \times \alpha \times \alpha) \times \]
\[ (\text{unit} + \text{int} \times \alpha) \times \]
\[ (\text{unit} + \text{int} \times \alpha \times \alpha) \]
\[ = \ldots \]
Unwinding Equations

We could *unwind* the equation:

\[
\alpha = \text{unit} + \text{int} \times \alpha \times \alpha \\
= \text{unit} + \text{int} \times \\
\quad (\text{unit} + \text{int} \times \alpha \times \alpha) \times \\
\quad (\text{unit} + \text{int} \times \alpha \times \alpha) \\
= \text{unit} + \text{int} \times \\
\quad (\text{unit} + \text{int} \times \\
\quad \quad (\text{unit} + \text{int} \times \alpha \times \alpha) \times \\
\quad \quad (\text{unit} + \text{int} \times \alpha \times \alpha) \times \\
\quad \quad (\text{unit} + \text{int} \times \alpha \times \alpha)) \times \\
\quad \quad (\text{unit} + \text{int} \times \alpha \times \alpha)) \\
= \ldots
\]

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

We’ll specify potentially-infinite solutions to type equations using a finite syntax based on the \textit{fixed-point type constructor} \( \mu \).

\[ \mu \alpha. \tau \]
μ 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:

\[ \text{tree} \triangleq \mu \alpha. \text{unit} + \text{int} \times \alpha \times \alpha. \]
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:

\[ \mu \alpha. \tau \text{ is a solution to } \alpha = \tau, \text{ so:} \]

\[ \mu \alpha. \tau = \tau \{\mu \alpha. \tau / \alpha\} \]

Two typing rules let us switch between folded and unfolded:

\[ \frac{\Gamma \vdash e : \tau \{\mu \alpha. \tau / \alpha\} \quad \mu\text{-INTRO}}{\Gamma \vdash e : \mu \alpha. \tau} \]

\[ \frac{\Gamma \vdash e : \mu \alpha. \tau \quad \mu\text{-ELIM}}{\Gamma \vdash e : \tau \{\mu \alpha. \tau / \alpha\}} \]
Isorecursive Types

Alternatively, *isorecursive types* avoid infinite type trees.

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

\[
\begin{align*}
\text{unfold}_{\mu \alpha. \tau} & : \mu \alpha. \tau \to \tau\{\mu \alpha. \tau / \alpha\} \\
\text{fold}_{\mu \alpha. \tau} & : \tau\{\mu \alpha. \tau / \alpha\} \to \mu \alpha. \tau
\end{align*}
\]
Static Semantics (Isorecursive)

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

$$\Gamma \vdash e : \tau\{\mu \alpha. \tau / \alpha\}$$

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

$$\Gamma \vdash e : \mu \alpha. \tau$$

$$\frac{}{\Gamma \vdash \text{unfold } e : \tau\{\mu \alpha. \tau / \alpha\}} \quad \mu\text{-ELIM}$$
Dynamic Semantics

We also need to augment the operational semantics:

\[ \text{unfold} \ (\text{fold} \ e) \rightarrow e \]

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:

\[
\text{intlist} \triangleq \mu \alpha. \text{unit} + \text{int} \times \alpha.
\]
Example

Here’s a recursive type for lists of numbers:

\[
\text{intlist} \triangleq \mu \alpha. \text{unit} + \text{int} \times \alpha.
\]

Here’s how to add up the elements of an \text{intlist}:

\[
\begin{align*}
\text{let sum} &= \\
&= \text{fix} \ (\lambda f : \text{intlist} \rightarrow \text{intlist} \\
&\quad \quad \lambda l : \text{intlist}. \ \text{case unfold} \ l \ \text{of} \\
&\quad \quad \quad (\lambda u : \text{unit}. \ 0) \\
&\quad \quad \quad \mid (\lambda p : \text{int} \times \text{intlist}. \ (#1 \ p) + f (#2 \ p)))
\end{align*}
\]
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:

\[
\text{nat} \triangleq \mu \alpha. \text{unit} + \alpha
\]
Encoding Numbers

Recursive types let us encode the natural numbers!

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

\[
\text{nat} \triangleq \mu \alpha. \text{unit} + \alpha \\
\text{0} \triangleq \text{fold} (\text{inl}_{\text{unit}} + \text{nat} ())
\]
Encoding Numbers

Recursive types let us encode the natural numbers!

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

\[ \text{nat} \triangleq \mu \alpha. \text{unit} + \alpha \]

\[ 0 \triangleq \text{fold} (\text{inl}_{\text{unit} + \text{nat}} ()) \]

\[ 1 \triangleq \text{fold} (\text{inr}_{\text{unit} + \text{nat}} 0) \]

\[ 2 \triangleq \text{fold} (\text{inr}_{\text{unit} + \text{nat}} 1), \]

\[ \vdots \]
Recursive types let us encode the natural numbers!

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

\[
\text{nat} \triangleq \mu \alpha. \text{unit} + \alpha
\]

\[
0 \triangleq \text{fold} (\text{inl}_{\text{unit} + \text{nat}} () )
\]

\[
1 \triangleq \text{fold} (\text{inr}_{\text{unit} + \text{nat}} 0)
\]

\[
2 \triangleq \text{fold} (\text{inr}_{\text{unit} + \text{nat}} 1),
\]

\[\vdots\]

The successor function has type \text{nat} \rightarrow \text{nat}:

\[
(\lambda x : \text{nat}. \text{fold} (\text{inr}_{\text{unit} + \text{nat}} x))
\]
Recall $\Omega$ defined as:

$$
\omega \triangleq \lambda x. x \; x \quad \Omega \triangleq \omega \; \omega .
$$

$\Omega$ was impossible to type... until now!
Self-Application and $\Omega$

Recall $\Omega$ defined as:

$$\omega \triangleq \lambda x. x \ x \quad \Omega \triangleq \omega \ \omega.$$ 

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

$x$ is a function. Let’s say it has the type $\sigma \rightarrow \tau$. 
Self-Application and $\Omega$

Recall $\Omega$ defined as:

$$\omega \triangleq \lambda x. x \ x \quad \Omega \triangleq \omega \ \omega.$$ 

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

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

$x$ is used as the argument to this function, so it must have type $\sigma$. 
Self-Application and $\Omega$

Recall $\Omega$ defined as:

$$\omega \triangleq \lambda x. x \ x \quad \Omega \triangleq \omega \ \omega.$$ 

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

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

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

So let’s write a type equation:

$$\sigma = \sigma \rightarrow \tau$$
Self-Application and $\Omega$

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

$$\omega \triangleq \lambda x : \mu \alpha. (\alpha \rightarrow \tau). (\text{unfold } x) \ x$$
Putting these pieces together, the fully typed $\omega$ term is:

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

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

So the type of $\text{fold } \omega$ is $\mu \alpha. (\alpha \to \tau)$. 

Self-Application and $\Omega$
Putting these pieces together, the fully typed $\omega$ term is:

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

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

So the type of $\text{fold } \omega$ is $\mu\alpha. (\alpha \to \tau)$.

Now we can define $\Omega = \omega \; (\text{fold } \omega)$. It has type $\tau$. 
Self-Application and $\Omega$:

We can even write $\omega$ in OCaml:

```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
```
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:

$$ U \triangleq \mu \alpha. \alpha \rightarrow \alpha $$
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:

$$U \triangleq \mu\alpha. \alpha \rightarrow \alpha$$

The full translation is:

$$[x] \triangleq x$$

$$[e_0 e_1] \triangleq \text{unfold} \ [e_0] \ [e_1]$$

$$[\lambda x. e] \triangleq \text{fold} \ \lambda x : U. \ [e]$$

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