Lecture 28
Recursive Types

7 November 2014
Announcements

- Foster office hours 11-12pm
- Guest lecture by Fran on Monday
Recursive Types

Many languages support recursive data types

Java

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

Many languages support recursive data types

Java

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

OCaml

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

Many languages support recursive data types

Java

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

OCaml

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

Simple Types

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

We would like the type \texttt{tree} to satisfy

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

We would like the type \texttt{tree} to satisfy

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

In other words, 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 the equation for tree, we have:

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

Unwinding the equation for tree, we have:

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

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

Unwinding the equation for \texttt{tree}, we have:

\[
\alpha = \texttt{unit} + \texttt{int} \times \alpha \times \alpha \\
= \texttt{unit} + \texttt{int} \times \\
\hspace{1cm} (\texttt{unit} + \texttt{int} \times \alpha \times \alpha) \times \\
\hspace{1cm} (\texttt{unit} + \texttt{int} \times \alpha \times \alpha) \\
= \texttt{unit} + \texttt{int} \times \\
\hspace{1cm} (\texttt{unit} + \texttt{int} \times \alpha \times \alpha) \times \\
\hspace{2cm} (\texttt{unit} + \texttt{int} \times \alpha \times \alpha) \times \\
\hspace{3cm} (\texttt{unit} + \texttt{int} \times \alpha \times \alpha) \\
\hspace{3cm} (\texttt{unit} + \texttt{int} \times \alpha \times \alpha))
\]
Unwinding Equations

Unwinding the equation for tree, we have:

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

Unwinding the equation for tree, we have:

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

At each level, we have a finite type with variables \( \alpha \) and we obtain the next level by substituting the right-hand side for \( \alpha \)
Infinite Types

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 $\times$, $+$, $\text{int}$, and $\text{unit}$.

This infinite tree is a solution of our equation, and this is what we take as the type $\text{tree}$.

More generally, over standard type constructors such as $\to$, $\times$, $+$, $\text{unit}$, and $\text{int}$, we can form the set of (finite) types inductively in the usual way.
Example

For example, the type \texttt{int} \rightarrow \texttt{int} \rightarrow \texttt{int} can be viewed as the labeled tree

```
    int
   /   \
int   int
   \   /
int  int
```
Example

A (finite or infinite) expression with only finitely many subexpressions (up to isomorphism) is called regular.

For example, the infinite type

\[ \text{int} \rightarrow \text{int} \rightarrow \text{int} \rightarrow \ldots \]

is regular, since it has only two subexpressions up to isomorphism, namely itself and int.
We can specify infinite solutions to systems of equations using a finite syntax involving the fixpoint type constructor $\mu$.
μ Types

We can specify infinite solutions to systems of equations using a finite syntax involving the fixpoint type constructor $\mu$.

Given an equation $\alpha = \tau$ such that the right-hand side is not $\alpha$, there is a unique solution, which is a finite or infinite regular tree.
We can specify infinite solutions to systems of equations using a finite syntax involving the \textit{fixpoint type constructor} $\mu$.

Given an equation $\alpha = \tau$ such that the right-hand side is not $\alpha$, there is a unique solution, which is a finite or infinite regular tree.

The solution will be infinite if $\alpha$ occurs in $\tau$ and will be finite (in fact it will just be $\tau$) if $\alpha$ does not occur in $\tau$. 
We can specify infinite solutions to systems of equations using a finite syntax involving the fixpoint type constructor $\mu$.

Given an equation $\alpha = \tau$ such that the right-hand side is not $\alpha$, there is a unique solution, which is a finite or infinite regular tree.

The solution will be infinite if $\alpha$ occurs in $\tau$ and will be finite (in fact it will just be $\tau$) if $\alpha$ does not occur in $\tau$.

We denote this unique solution by $\mu\alpha.\tau$. 

\( \mu \) Types

We can specify infinite solutions to systems of equations using a finite syntax involving the \textit{fixpoint type constructor} \( \mu \).

Given an equation \( \alpha = \tau \) such that the right-hand side is not \( \alpha \), there is a unique solution, which is a finite or infinite regular tree.

The solution will be infinite if \( \alpha \) occurs in \( \tau \) and will be finite (in fact it will just be \( \tau \)) if \( \alpha \) does not occur in \( \tau \).

We denote this unique solution by \( \mu \alpha. \tau \).

Note that \( \mu \) acts as a binding operator in type expressions.
Example

To get a tree type satisfying our original equation, we can define

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

...and it is straightforward to extend this to mutually recursive types.
In *equirecursive types* we take a recursive type to be equal to its (potentially infinite) unfolding.

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

$$\mu \alpha. \tau = \tau\{\mu \alpha. \tau/\alpha\}.$$
In *equirecursive types* we take a recursive type to be equal to its (potentially infinite) unfolding.

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

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

...and so the typing rules are simple:

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

$$\Gamma \vdash e : \mu \alpha. \tau \quad \mu\text{-intro}$$

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

$$\Gamma \vdash e : \tau \{\mu \alpha. \tau / \alpha\} \quad \mu\text{-elim}$$

Equivalently, we can just allow substitution of equals for equals.
Another approach is to work with *isorecursive types*. 

Isorecursive Types
Another approach is to work with *isorecursive types*. Here we do not have any infinite types, but rather the expression $\mu \alpha. \tau$ is itself a type that is distinct, but isomorphic to $\tau \{ \mu \alpha. \tau / \alpha \}$.
Another approach is to work with *isorecursive types*.

Here we do not have any infinite types, but rather the expression $\mu \alpha. \tau$ is itself a type that is distinct, but isomorphic to $\tau\{\mu \alpha. \tau/\alpha\}$

The step of substituting $\mu \alpha. \tau$ for $\alpha$ in $\tau$ is called *unfolding*, and the reverse operation is called *folding*
Isorecursive Types

Another approach is to work with isorecursive types.

Here we do not have any infinite types, but rather the expression $\mu \alpha. \tau$ is itself a type that is distinct, but isomorphic to $\tau\{\mu \alpha. \tau/\alpha\}$

The step of substituting $\mu \alpha. \tau$ for $\alpha$ in $\tau$ 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.

\[
\text{unfold}_{\mu \alpha. \tau} : \mu \alpha. \tau \rightarrow \tau\{\mu \alpha. \tau/\alpha\}
\]
\[
\text{fold}_{\mu \alpha. \tau} : \tau\{\mu \alpha. \tau/\alpha\} \rightarrow \mu \alpha. \tau
\]
In the isorecursive view, the typing rules consist of a pair of introduction and elimination rules for $\mu$-types that explicitly mention `fold` and `unfold`:

\[
\begin{align*}
\Gamma \vdash e : \tau\{\mu\alpha.\tau/\alpha\} \\
\Gamma \vdash \text{fold } e : \mu\alpha.\tau \\
\Gamma \vdash e : \mu\alpha.\tau \\
\Gamma \vdash \text{unfold } e : \tau\{\mu\alpha.\tau/\alpha\}
\end{align*}
\]
Dynamic Semantics

We also need to augment the operational semantics:

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

Intuitively, to access data in a recursive type \( \mu \alpha. \, \tau \), we need to \textbf{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 \textbf{fold}
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

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

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

Now suppose we want to add up the elements of an \text{intlist}. This will be a recursive function, so we would need to take a fixpoint

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

Now that we have recursive types, we no longer need to take \texttt{int} as primitive, but we can define it as a recursive type.
Encoding Numbers

Now that we have recursive types, we no longer need to take \texttt{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:
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:

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

Now that we have recursive types, we no longer need to take \texttt{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:

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

\[
0 \triangleq \text{fold} (\text{inl}_{\texttt{nat}} () )
\]
Encoding Numbers

Now that we have recursive types, we no longer need to take \texttt{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:

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

Now that we have recursive types, we no longer need to take \texttt{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:

\[
\text{nat} \triangleq \mu \alpha. \text{unit} + \alpha \\
0 \triangleq \text{fold} (\text{inl}_{\text{nat}} ()) \\
1 \triangleq \text{fold} (\text{inr}_{\text{nat}} 0) \\
2 \triangleq \text{fold} (\text{inr}_{\text{nat}} 1),
\]
Encoding Numbers

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:

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

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

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

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

The successor function is:

\[
(\lambda x : \text{nat}. \text{fold}(\text{inr}_{\text{nat}}x)) : \text{nat} \rightarrow \text{nat}.
\]
Self-Application and $\Omega$

Recall $\Omega$ defined as:

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

We can now give these terms recursive types!
Self-Application and $\Omega$

Recall $\Omega$ defined as:

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

We can now give these terms recursive types!

$x$ is used as a function, so it must have a type, say $\sigma \rightarrow \tau$
Self-Application and $\Omega$

Recall $\Omega$ defined as:

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

We can now give these terms recursive types!

$x$ is used as a function, so it must have a type, say $\sigma \rightarrow \tau$

But $x$ is applied to itself, so it must also have type $\sigma$
Recall $\Omega$ defined as:

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

We can now give these terms recursive types!

$x$ is used as a function, so it must have a type, say $\sigma \rightarrow \tau$

But $x$ is applied to itself, so it must also have type $\sigma$

Hence, the type of $x$ must satisfy the equation $\sigma = \sigma \rightarrow \tau$
Putting all these pieces together, the fully typed \(\omega\) term is:

\[
\omega \equiv (\lambda x : \mu \alpha. (\alpha \to \tau). (\text{unfold } x) \ x) : (\mu \alpha. (\alpha \to \tau)) \to \tau.
\]
Self-Application and $\Omega$

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

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

We can also 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
```
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, which leads to the type:

$$U \triangleq \mu \alpha. \alpha \rightarrow \alpha$$
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, which leads to the type:

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

The full translation is as follows

$$[[x]] \triangleq x$$

$$[[e_0 e_1]] \triangleq (\text{unfold} \; [[e_0]]) \; [[e_1]]$$

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

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