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

\[
\text{tree} \triangleq \text{unit} + \text{int} \times \text{tree} \times \text{tree}
\]
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?

\[\text{tree} = \text{unit} + \text{int} \times \text{tree} \times \text{tree}\]
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:

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

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

We could *unwind* the equation:

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

We could *unwind* the equation:

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

We could *unwind* the equation:

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

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

\[
= \ldots
\]

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

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

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

$$\mu \alpha . \tau$$

$$\alpha = \nu \tau$$

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

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

\[
\begin{array}{c}
\Gamma \vdash e : \tau_2 \\
\Gamma \vdash \text{inv} \ ((1, \text{inl} \ 1), \text{inl} 0) : \mu \alpha. \text{unit} + \text{int} \times \alpha + \alpha \\
\Gamma \vdash \text{inv} \ ((1, \text{inl} \ 1), \text{inl} 0) : \text{unit} + \text{int} \times \mu \alpha. \text{unit} + \text{int} \times \alpha + \alpha \\
\Gamma \vdash \text{inv} \ ((1, \text{inl} \ 1), \text{inl} 0) : \mu \alpha. \text{unit} + \text{int} \times \alpha + \alpha \\
\Gamma \vdash \text{inv} \ ((1, \text{inl} \ 1), \text{inl} 0) : \mu \alpha. \text{unit} + \text{int} \times \alpha + \alpha = \text{Evec}
\end{array}
\]
Static Semantics (Equirecursive)

We’ll define two treatments of recursive types. With \textit{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:

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

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

Alternatively, isorecursive types avoid infinite type trees.

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

$\text{fold} : \tau \rightarrow \mu \alpha. \tau \rightarrow (\mu \alpha. \tau) \rightarrow (\mu \alpha. \tau) \rightarrow \ldots$

$\text{unfold} : \mu \alpha. \tau \rightarrow \tau \rightarrow \tau \rightarrow \ldots$

$\text{fold \ (\text{nil} \ ())) : \text{tree}$

unif + ...
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:

$$\text{unfold}_{\mu \alpha. \tau} : \mu \alpha. \tau \rightarrow \tau\{\mu \alpha. \tau / \alpha\}$$

$$\text{fold}_{\mu \alpha. \tau} : \tau\{\mu \alpha. \tau / \alpha\} \rightarrow \mu \alpha. \tau$$

\[\Gamma \vdash e : \exists \alpha. \exists \beta. \tau \]  $\mu$-intro

\[\Gamma \vdash \text{fold } e : \mu \alpha. \tau\]
Static Semantics (Isorecursive)

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

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

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

$$
e' \xrightarrow{*} \text{unfold} (\text{fold } e) \leadsto x \in \mathbb{R}
$$

$$
\Gamma \vdash x : \forall\alpha.\beta
$$
Dynamic Semantics

We also need to augment the operational semantics:

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

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

Intuitively, to access data in a recursive type \( \mu \alpha. \tau \), we need to \text{unfold} it first. And the only way that values of type \( \mu \alpha. \tau \) could have been created is via \text{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}:

let sum =
fix (\lambda f : \text{intlist} \to \text{intlist}
\lambda l : \text{intlist}. \text{case unfold}(l) of
(\lambda u : \text{unit}. 0)
| (\lambda p : \text{int} \times \text{intlist}. (#1 p) + f(#2 p)))
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:

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

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

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

\[
\vdots
\]
Encoding Numbers

Recursive types let us encode the natural numbers!

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

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

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

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

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

\[
\vdots
\]

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

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

Recall $\Omega$ defined as:

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

$\Omega$ 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 $\sigma \rightarrow \tau. = \alpha$

$$\lambda x: \mu \alpha. \alpha \rightarrow \tau$$
Self-Application and $\Omega$

Recall $\Omega$ defined as:

\[
\omega \triangleq \lambda x. \, x \, x \quad \quad \Omega \triangleq \omega \, \omega.
\]

$\Omega$ 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:

$$\sigma = \sigma \rightarrow \tau$$  

$$\lambda x : \mu \alpha. \alpha \rightarrow \tau. \ (\text{unfold} \ x) \ (x^{\ell}) : \mu \alpha. \alpha \rightarrow \tau$$
Self-Application and $\Omega$

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

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

$$\mu \alpha. (\alpha \to \tau) \to \tau$$

$$\text{fold } \omega : \mu \alpha. \alpha \to \tau$$
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 $\mu \alpha. (\alpha \to \tau)$.
Self-Application and $\Omega$

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

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

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

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

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);;
let 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!

\[ U \rightarrow U = U \]

\[ \mu\alpha. \alpha \rightarrow \alpha \]
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. So let’s define an “untyped” type:

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

$$[\text{e}_1, \text{e}_2] \triangleq \text{unfold } \text{e}_1 \text{ e}_2$$

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

$$[\text{x}] \triangleq x$$
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:

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

The full translation is:

$$\text{let } \alpha \text{ : } \text{tree} \text{ in } \lambda x : \text{tree} . e$$

$$[x] \triangleq x$$

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

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

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