CS 4110

Programming Languages & Logics

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

```
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

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

Recursive Type Equations

We would like the type **tree** to satisfy

$$\mathit{tree} = \mathbf{unit} + \mathbf{int} \times \mathit{tree} \times \mathit{tree}$$

Recursive Type Equations

We would like the type **tree** to satisfy

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

In other words, 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

Unwinding the equation for **tree**, we have:

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

Unwinding Equations

Unwinding the equation for **tree**, we have:

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

Unwinding Equations

Unwinding the equation for **tree**, we have:

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

Unwinding Equations

Unwinding the equation for **tree**, we have:

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

Unwinding Equations

Unwinding the equation for **tree**, we have:

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

At each level, we have a finite type with variables α and we obtain the next level by substituting the right-hand side for α

Infinite Types

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

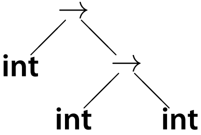
We can think of this as an infinite labeled graph whose nodes are labeled with the type constructors $\times$, $+$, **int**, and **unit**.

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

More generally, over standard type constructors such as $\rightarrow$, $\times$, $+$, **unit**, and **int**, we can form the set of (finite) types inductively in the usual way

Example

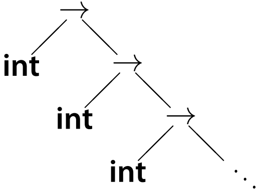
For example, the type **int** $\rightarrow$ **int** $\rightarrow$ **int** can be viewed as the labeled tree



Example

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

For example, the infinite type



is regular, since it has only two subexpressions up to isomorphism, namely itself and **int**

μ Types

We can specify infinite solutions to systems of equations using a finite syntax involving the *fixpoint type constructor* μ

μ Types

We can specify infinite solutions to systems of equations using a finite syntax involving the *fixpoint type constructor* μ

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

μ Types

We can specify infinite solutions to systems of equations using a finite syntax involving the *fixpoint type constructor* μ

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

The solution will be infinite if α occurs in τ and will be finite (in fact it will just be τ) if α does not occur in τ

μ Types

We can specify infinite solutions to systems of equations using a finite syntax involving the *fixpoint type constructor* μ

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

The solution will be infinite if α occurs in τ and will be finite (in fact it will just be τ) if α does not occur in τ

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

μ Types

We can specify infinite solutions to systems of equations using a finite syntax involving the *fixpoint type constructor* μ

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

The solution will be infinite if α occurs in τ and will be finite (in fact it will just be τ) if α does not occur in τ

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

Note that μ acts as a binding operator in type expressions

Example

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

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

...and it is straightforward to extend this to mutually recursive types

Static Semantics (Equirecursive)

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

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

Static Semantics (Equirecursive)

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:

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

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

Equivalently, we can just allow substitution of equals for equals

Isorecursive Types

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

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 α in τ 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 α in τ 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.

$$\begin{array}{rcl} \textbf{unfold}_{\mu\alpha.\,\tau} & : & \mu\alpha.\,\tau \;\to\; \tau\{\mu\alpha.\,\tau/\alpha\} \\ \textbf{fold}_{\mu\alpha.\,\tau} & : & \tau\{\mu\alpha.\,\tau/\alpha\} \;\to\; \mu\alpha.\,\tau \end{array}$$

Static Semantics (Isorecursive)

In the isorecursive view, the typing rules consist of a pair of introduction and elimination rules for μ-types that explicitly mention **fold** and **unfold**:

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

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

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; but the only way that values of type $\mu\alpha.\,\tau$ could have been created in the first place is via a **fold**

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

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

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

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

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

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

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

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:

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:

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

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:

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

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

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:

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

$$0 \triangleq \textbf{fold}\ (\mathrm{inl}_{\textbf{nat}}\ ())$$

$$1 \triangleq \textbf{fold}\ (\mathrm{inr}_{\textbf{nat}}\ 0)$$

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:

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

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:

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

The successor function is:

$$(\lambda x : \textbf{nat}.\, \textbf{fold}\ (\mathrm{inr}_{\textbf{nat}}\ x)) : \textbf{nat} \to \textbf{nat}.$$

Self-Application and Ω

Recall Ω defined as:

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

We can now give these terms recursive types!

Self-Application and Ω

Recall Ω defined as:

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

We can now give these terms recursive types!

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

Self-Application and Ω

Recall Ω defined as:

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

We can now give these terms recursive types!

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

But x is applied to itself, so it must also have type σ

Self-Application and Ω

Recall Ω defined as:

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

We can now give these terms recursive types!

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

But *x* is applied to itself, so it must also have type σ

Hence, the type of *x* must satisfy the equation $\sigma = \sigma \to \tau$

Self-Application and Ω

Putting all these pieces together, the fully typed ω term is:

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

Self-Application and Ω

Putting all these pieces together, the fully typed ω term is:

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

We can also write ω 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 λ-Calculus

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

Encoding λ-Calculus

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

Every λ-term can be applied as a function to any other λ-term, which leads to the type:

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

Encoding λ-Calculus

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

Every λ-term can be applied as a function to any other λ-term, which leads to the type:

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

The full translation is as follows

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

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