CSC547: Type Systems for OO Languages

Recursion

Abadi and Cardelli, Chapter 9

Recursive types

Recursive types and subtyping
Recursion

So far, many examples have been rather unsatisfactory:

\[ P \triangleq [ x : \text{int}, y : \text{int} ] \]

\[ Point \triangleq [ x : \text{int}, y : \text{int}, \text{dotprod} : P \rightarrow \text{int} ] \]

\[ \text{prototypePoint} \triangleq [ \]
\[ \quad x = 0, y = 0, \]
\[ \quad \text{dotprod} = \varsigma(this : Point) \lambda(other : P) ( \]
\[ \quad \quad (this.x * other.x) + (this.y * other.y) \]
\[ \quad ) \]
\[ ]

\[ newPoint \triangleq \lambda(x : \text{int}) \lambda(y : \text{int}) ( \]
\[ \quad \text{prototypePoint} . x := x . y := y \]
\[ ) \]

We don’t really want \( P \) here!

We’d like \( Point \) to be a recursive type.
Recursion

In the object calculus, recursive types are written \( \mu(X)A \) for example:

\[
Point \triangleq \mu(P)[x : \text{int}, y : \text{int}, \text{dotprod} : P \rightarrow \text{int}]
\]

The recursive type comes with two operations fold and unfold:

(Val Fold)
If \( E \vdash a : UPoint \)
then \( E \vdash \text{fold}(Point, a) : Point \)

(Val Unfold)
If \( E \vdash a : Point \)
then \( E \vdash \text{unfold}(a) : UPoint \)

where:

\[
UPoint \triangleq [x : \text{int}, y : \text{int}, \text{dotprod} : Point \rightarrow \text{int}]
\]

Do the following type-check?

\[
p : \text{Point} \vdash p.x : \text{int}
p : \text{Point} \vdash p.\text{dotProd}(p) : \text{int}
p : \text{Point} \vdash p.x := 3 : \text{Point}
\]
Recursion

For example:

\[
\text{prototypePoint} \triangleq \text{fold} (\text{Point}, [ \\
\quad x = 0, y = 0, \\
\quad \text{dotprod} = \zeta (this : \text{UPoint}) \lambda (other : \text{Point}) ( \\
\quad \quad (this.x * \text{unfold} (other).x) + \\
\quad \quad (this.y * \text{unfold} (other).y) \\
\quad ) \\
\])
\]

\[
\text{newPoint} \triangleq \lambda (x : \text{int}) \lambda (y : \text{int}) (\text{fold} (\text{Point}, \\
\quad \text{unfold} (\text{prototypePoint}) \ . \ x := x \ . \ y := y \\
\))
\]
Recursion

What about a Cell example in pseudo-Java:

```java
interface Cell {
    contents : int;
    get () : int;
    set (x : int) : Cell;
}
```

What about a type for the untyped lambda-calculus:

```java
interface Function {
    apply (f : Function) : Function;
}
```
Recursion

Formal treatment of recursive types...

Assume a grammar of type variables \( X, Y, Z \)...

The formal system \( \Delta_X \) allows type variables to be used for types:

\[
(\text{Env } X) \text{ (where } X \text{ not in } E) \\
\text{If } E \vdash \diamond \text{ then } E, X \vdash \diamond
\]

\[
(\text{Type } X) \\
\text{If } E, X, E' \vdash \diamond \text{ then } E, X, E' \vdash X
\]

For example, how can we show

\[
P \vdash [x : \text{int}, y : \text{int}, \text{dotprod} : P \rightarrow \text{int}]
\]
Recursion

The formal system $\Delta_{\mu}$ allows us to create recursive types and fold and unfold:

(Type Rec)
If $E, X \vdash A$
then $E \vdash \mu(X)A$

(Val Fold)
If $E \vdash a : B \{\ X \leftarrow A \} \}$
then $E \vdash \text{fold}(A, a) : A$

(Val Unfold)
If $E \vdash a : A$
then $E \vdash \text{unfold}(a) : B \{\ X \leftarrow A \} \}$

where $A \equiv \mu(X)B$.

How can we show:

$\vdash \text{Point}$
$\vdash \text{prototypePoint} : \text{Point}$
Recursion

Operational semantics for recursion is given by:

\[ \text{unfold} \left( \text{fold} \left( A, a \right) \right) \rightarrow a \]

For example, what happens when we run:

\[ \text{newPoint} \left( 5 \right) \left( 37 \right) \]

recall:

\[ \text{prototypePoint} \triangleq \text{fold} \left( \text{Point}, \left[ \begin{array}{l}
x = 0, y = 0, \\
dotprod = \zeta (this : \text{UPoint}) \lambda (other : \text{Point}) (\\n \hspace{1cm} (this.x \ast \text{unfold (other).x}) + \\
 \hspace{1cm} (this.y \ast \text{unfold (other).y})\\n\end{array} \right]\right) \]

\[ \text{newPoint} \triangleq \lambda (x : \text{int}) \lambda (y : \text{int}) \left( \text{fold} \left( \text{Point}, \text{unfold (prototypePoint)} \right) . x := x . y := y \right) \]
Recursion and Subsumption

OK, let’s try another example:

\[
Cell \triangleq \mu(C) \left[ \\
\text{get}^+ : \text{int}, \text{set}^+ : \text{int} \rightarrow C \\
\right]
\]

How would we implement a prototype cell with a contents field?

Hint:

\[
PrivateCell \triangleq \mu(PC) \left[ \\
\text{contents} : \text{int}, \text{get}^+ : \text{int}, \text{set}^+ : \text{int} \rightarrow PC \\
\right]
\]
Recursion and Subsumption

A problem: we need that $\text{PrivateCell} <: \text{Cell}$, which we can’t derive yet!

What happens if we just add the rule:

(Sub Rec Failed)
If $E \vdash \mu(X)A$
and $E \vdash \mu(Y)B$
and $E, Y, X \vdash A <: B$
then $E \vdash \mu(X)A <: \mu(Y)B$

now try to derive $\text{PrivateCell} <: \text{Cell}$
Recursion and Subsumption

The problem is you need \( PC <: C \) in order to show that \( PrivateCell <: Cell \).

So we add these constraints into the environment:

\[
(\text{Sub Rec})
\]

If \( E \vdash \mu(X)A \)

and \( E \vdash \mu(Y)B \)

and \( E, Y, X <: Y \vdash A <: B \)

then \( E \vdash \mu(X)A <: \mu(Y)B \)

now try to derive \( PrivateCell <: Cell \)
Recursion and Subsumption

The formal system for environments with type variable bounds $\Delta_{<:X}$ is:

(Env $X::<$) (where $X$ not in $E$)
If $E \vdash A$
then $E, X <: A \vdash \diamond$

(Type $X::<$)
If $E, X <: A, E' \vdash \diamond$
then $E, X <: A, E' \vdash X$

(Sub $X$)
If $E, X <: A, E' \vdash \diamond$
then $E, X <: A, E' \vdash X <: A$

We can write $X$ for the environment $X <: Top$ so this is an extension of $\Delta_X$.

How can we derive:

$C, PC <: C \vdash$
[ contents : int, get$^+$ : int, set$^+$ : int $\rightarrow$ PC ]
$ <: [ get^+ : int, set^+ : int \rightarrow C ]$
Recursion and Subsumption

The formal system for recursion and subtyping $\Delta <: \mu$ extends $\Delta_{\mu}$ with:

(Sub Rec)
If $E \vdash \mu(X)A$
and $E \vdash \mu(Y)B$
and $E, Y, X <: Y \vdash A <: B$
then $E \vdash \mu(X)A <: \mu(Y)B$

How can we now derive $PrivateCell <: Cell$?
Recursion and Subsumption

We now have some quite realistic languages:

\[ \text{Ob}_1^{<:\mu} \triangleq \text{Ob}_1^{<} \cup \Delta_{<:\chi} \cup \Delta_{<:\mu} \]

\[ \text{F}_1^{<:\mu} \triangleq \text{F}_1^{<} \cup \Delta_{<:\chi} \cup \Delta_{<:\mu} \]

\[ \text{FOb}_1^{<:\mu} \triangleq \text{FOb}_1^{<} \cup \Delta_{<:\chi} \cup \Delta_{<:\mu} \]

This is enough to code up a lot of Java!
Some properties of $\text{Ob}_1 < : \text{mu}$

Proposition (Ob$_1 < :$ has minimum types in an empty environment)

If $\vdash a : B$ then there exists a minimal $\vdash a : A$.

Proof see Section 9.3.1.

Proposition (Subject reduction for Ob$_1 < : \text{mu}$)

If $\vdash a : A$ and $a \rightsquigarrow v$ then $\vdash v : A$

Proof see Section 9.3.2.
End of immutable objects

Some missing features:

Assignment.

F-bounded polymorphism.

Concurrency and distribution.