CSC547: Type Systems for OO Languages

A Concurrent Object Calculus


Motivation

Concurrent objects

Synchronization

Results
Motivation

Concurrency!

Multiple threads, object locking...
Concurrent objects

Sample pseudo-Java code:

class Server implements Runnable {
    final ServerSocket ss;
    Server (ServerSocket ss) {
        this.ss = ss;
    }
    void run () {
        while (true) {
            final Socket s = ss.accept ();
            new Thread () { public void run () {
                handle (s);
            }}.start ();
        }
    }
    void handle (Socket s) { ... }
}

What does this code do?
Concurrent objects

Translated into the concurrent object calculus:

\[
\text{server} \triangleq \lambda (ss : \text{ServerSocket}) \[
ss = ss, \\
run = \varsigma (this : \text{Server}) ( \\
let s = ss.accept in \\
this.handle (s) \uparrow this.run \\
), \\
handle = \varsigma (this : \text{Server}) \lambda (s : \text{Socket}) ( ... ) 
\]

The important new gadget: \( a \uparrow b \) which spawns off \( a \) as a new thread.

How is \( a \uparrow b \) written in Java?

What are the types \textit{Server}, \textit{ServerSocket} and \textit{Socket}?

How can we implement the dining philosophers in Java?

How can we implement the dining philosophers in the concurrent object calculus?
Concurrent objects

In the imperative object calculus, we modelled a program as a heap and an term: $H | a$.

What did a heap look like?

What were the reduction rules for imperative objects?
**Concurrent objects**

In the concurrent object calculus, we consider terms and heap to be the same:

\[ a, b, c ::= \ldots \text{ as before } \ldots \]

\[ a \uparrow b \]

\[ p \mapsto O \]

The imperative object calculus configuration:

\[ p \mapsto [\text{ foo}=0, \text{ bar}=q ], q \mapsto [\text{ foo}=0, \text{ bar}=p ] | \]
\[ p.\text{bar}.\text{bar}.\text{foo} \]

corresponds to what concurrent object calculus program?

What do the reduction rules for imperative objects look like in the concurrent object object calculus?
Concurrent objects

Well almost...

In the concurrent object calculus we keep track of shared secrets: object references which are private to some threads:

\[ a \vdash (\nu p)( p \mapsto O \vdash b ) \]

Here, \( p \) is a secret which \( b \) knows about but \( a \) does not.

The important properties about secrets is that their name is hidden from the outside world:

(Alpha equivalence)
\[ (\nu p)b \equiv (\nu q)(b \{ p \leftarrow q \}) \]

(Scope extrusion)
\[ ((\nu q)a) \vdash b \equiv (\nu q)(a \vdash b) \]

(when \( q \) is not in \( b \).)

For example, drag the \( \nu s \) up to top level in:

\[ ( (\nu p)( p \mapsto O \vdash a ) ) \vdash ( (\nu p)( p \mapsto O' \vdash a' ) ) \]
Concurrent objects

The concurrent object calculus doesn’t have objects as primitive!

They’re now derived:

\[ O \triangleq (\nu p)(p \mapsto O \uparrow p) \]

For example:

\[ \text{let } x = [ \text{foo}=0, \text{bar}=\varsigma(x)x ] \text{ in } a \]
Concurrent objects

The semantics is defined using a structural congruence \( \equiv \) which includes alpha equivalence, scope extrusion and:

- We can consider the heap as a ‘soup’ of threads and objects:

  (Struct Par Assoc)
  \[
  (a \overset{\alpha}{\implies} b) \overset{\beta}{\implies} c \equiv a \overset{\alpha}{\implies} (b \overset{\beta}{\implies} c)
  \]

  (Struct Par Comm)
  \[
  (a \overset{\alpha}{\implies} b) \overset{\beta}{\implies} c \equiv (b \overset{\alpha}{\implies} a) \overset{\beta}{\implies} c
  \]

- Heap can be dragged up through let:

  (Struct Let Assoc) \((y \text{ not in } c)\)
  \[
  (\text{let } x = (\text{let } y = a \text{ in } b) \text{ in } c)
  \equiv \text{let } y = a \text{ in } (\text{let } x = b \text{ in } c)
  \]

  (Struct Res Let) \((p \text{ not in } b)\)
  \[
  (\nu p)(\text{let } x = a \text{ in } b) \equiv \text{let } x = (\nu p)a \text{ in } b
  \]

  (Struct Par Let)
  \[
  a \overset{\alpha}{\implies} \text{let } x = b \text{ in } c \equiv \text{let } x = (a \overset{\alpha}{\implies} b) \text{ in } c
  \]

For example, drag all the heap up to top level in:

\[
\text{let } x = [ \text{foo}=0, \text{bar}=\varsigma(x)x ] \text{ in } a
\]
Concurrent objects

The reduction semantics is then the same (apart from syntax) as the imperative object calculus, for example:

(Red Clone)

\[
(p \mapsto O) \mapsto clone(p) \\
\mapsto (p \mapsto O) \mapsto (\nu q) \mapsto ((q \mapsto O) \mapsto q)
\]

(Red Let Result)

\[
let x = v in b \\
\mapsto b \{ \{ x \leftarrow v \} \}
\]

We allow \((\nu p)-\), \((- \mapsto b)\) and \((a \mapsto -)\) as evaluation contexts.

For example:

\[
let x = [ foo=0, bar=\varsigma(x)x ] in \\
let y = clone (x) in \\
x.bar:=y; \\
y.bar:=x; \\
x.bar.bar.foo
\]
Synchronization

To do anything useful, we need to add object locking...

In the concurrent object calculus, we add lock objects, which are either \textit{locked} or \textit{unlocked} together with operations \textit{acquire(lock)} and \textit{release(lock)}.

How can we code up the dining philosophers (with just two philosophers and two forks)?
Synchronization

Additional rules for locks are:

(Red Acquire)
\[(p \mapsto unlocked) \xrightarrow{\tau} (acquire(p))\]
\[(p \mapsto locked) \xrightarrow{\tau} (p)\]

(Red Release)
\[(p \mapsto O) \xrightarrow{\tau} (release(p))\]
\[(p \mapsto unlocked) \xrightarrow{\tau} (p)\]

How can we get the philosophers to deadlock?
Synchronization

We can code examples using locks, e.g. streams:

\[
\text{newChan} \triangleq \\
\quad \text{let } rd = \text{locked in} \\
\quad \text{let } wr = \text{unlocked in} [ \\
\quad \quad \text{reader} = rd, \\
\quad \quad \text{writer} = wr, \\
\quad \quad \text{val} = \text{rubbish}, \\
\quad \quad \text{read} = \varsigma (\text{this})( \\
\quad \quad \quad \text{acquire (this.reader);} \\
\quad \quad \quad \text{let } x = \text{this.val in} \\
\quad \quad \quad \text{release (this.writer);} \\
\quad \quad \quad x \\
\quad \quad ), \\
\quad \quad \text{write} = \varsigma (\text{this}) \lambda (x)( \\
\quad \quad \quad ( \\
\quad \quad \quad \text{acquire (this.writer);} \\
\quad \quad \quad \text{this.val} = x; \\
\quad \quad \quad \text{release (this.reader)} \\
\quad \quad \quad ) \triangleright x \\
\quad \quad ) \\
\quad ] \\
\]

What happens:

\[
\text{let } c = \text{newChan in } ( \text{c.write (3)} \triangleright \text{c.read} )
\]
Results

We can adapt Abadi and Cardelli’s type system for the imperative object calculus to the concurrent object calculus.

We get subject reduction.
Next week

Thoughts about future versions of the class (optional!)...