Today's Session

- We have the room for an hour – but I’ll be around after
- I haven’t seen this year’s paper
- One request: structural induction
- I have slides working through two further types of questions:
  - “Is this substitution correct?”
  - “Is this system sound?”
- ...but we can go through anything on the board
Exam Information

→ Your exam:
  → **Time:** Friday, 4th May 2018, 14:30 to 16:30
  → **Location:** Patersons Land - G.21
  → (Be sure to check closer to the time – these sometimes change!)

→ Exam format:
  → Two hours
  → Question 1 is compulsory, then you have a choice between questions 2 and 3.

→ Revision Exercises:
  → Four papers:
    → Mock exam (on EPL course page)
    → 2015/16 exam
    → 2015/16 resit exam
    → 2016/17 exam
  → Tutorial questions
Consider the following BNF grammar:

\[ e ::= 0 | e_1 + e_2 \]

(i) Define a Scala type called `Expr` using case classes to represent the above abstract syntax

(ii) The size of an expression in this grammar is the number of symbols in the expression (excluding parentheses, if any). Define a Scala function `size` that computes the size of an expression.

(iii) The size of an expression in the above grammar is always odd. Sketch a proof of this by induction on the structure of expressions (explaining the base case and induction step).
\[ e ::= 0 | e_1 + e_2 \]

(i) Define a Scala type called `Expr` using case classes to represent the above abstract syntax

```scala
abstract class Expr
case object Zero extends Expr
case class Plus(e1: Expr, e2: Expr) extends Expr
```
(ii) The size of an expression in this grammar is the number of symbols in the expression (excluding parentheses, if any). Define a Scala function size that computes the size of an expression.

```scala
def size(e: Expr): Int = e match {
  case Zero => 1
  case Plus(e1, e2) => size(e1) + size(e2) + 1
}
```
(iii) The size of an expression in the above grammar is always odd. Sketch a proof of this by induction on the structure of expressions (explaining the base case and induction step).

→ **Structural induction:** assume that a certain property is true of each subterm. Use this knowledge to prove that each term also satisfies the property.

→ **Base case:** a constructor without any subterms (the 0 expression)

→ **Inductive case:** a constructor containing subterms \((e_1 + e_2)\)
Theorem

Let $e$ be an expression in the above grammar. The size of $e$ is always odd.

Proof.

By structural induction on $e$.

Case $e = 0$: $\text{size}(0) = 1$, which is odd, as required.

Case $e = e_1 + e_2$:

\[ \begin{align*}
\text{By the induction hypothesis, } \text{size}(e_1) \text{ is odd} \\
\text{By the induction hypothesis, } \text{size}(e_2) \text{ is odd} \\
\text{Two odd numbers added together make an even number} \\
\text{(can write } \text{size}(e_1) = 2j + 1 \text{ and } \text{size}(e_2) = 2k + 1) \\
(2j + 1) + (2k + 1) = 2(j + k + 1) \\
\text{Extra symbol } +, \text{ so we have } 2(j + k + 1) + 1, \text{ which is odd, as required.}
\end{align*} \]
Consider the following substitutions:

\[ \rightarrow (\lambda x. x y)[x/ y] = \lambda z. z x \]

\[ \rightarrow (\lambda x. \lambda y. (x, y, z))[y, z/ x] = \lambda x. \lambda y. ((y, z), y, z) \]

\[ \rightarrow (\lambda x. x + ((\lambda y. y) z))[y/ z] = \lambda x. x + ((\lambda y. y) y) \]

\[ \rightarrow (\lambda x. x + ((\lambda y. y) z))[x/ z] = \lambda x. x + ((\lambda y. y) x) \]

For each one, explain whether the substitution has been performed correctly or not. If not, give the correct answer for the right-hand side.

[8 marks]
This is correct.

Substituting $x$ for $y$ naively would result in $x : x x$. Here, $x$ would be captured by the $x$ binder, changing the meaning of the program.

Instead, it is always safe to perform substitution by choosing fresh variables for the binders, and then performing the substitution:

$$((\lambda x.x y)[x/y] = \lambda z.z x$$
This is correct.

→ Substituting $x$ for $y$ naively would result in $\lambda x.x x$. Here, $x$ would be captured by the $\lambda x$ binder, changing the meaning of the program.

→ Instead, it is always safe to perform substitution by choosing fresh variables for the binders, and then performing the substitution:
   
   $$(\lambda z.z y)[x/y] = (\lambda z.z x)$$
(\lambda x. \lambda y. (x, y, z))[\langle y, z \rangle / x] = \lambda x. \lambda y. \langle y, z \rangle, y, z)
\[(\lambda x. \lambda y. (x, y, z))[(y, z)/x] = \lambda x. \lambda y. ((y, z), y, z)\]

→ This is incorrect.
→ We can only substitute for free variables – the \(x\) here was bound.
→ Even if we could: whereas the \(y\) in \((y, z)\) was free before the substitution, \(y\) has been captured by the \(\lambda y\) afterwards.
→ To correct the substitution, freshen the binders beforehand:

\[(\lambda a. \lambda b. (a, b, z))[(y, z)/x] = \lambda a. \lambda b. (a, b, z)\]
\((\lambda x.x + ((\lambda y.y) z))[y/z] = \lambda x.x + ((\lambda y.y) y)\)
\[(\lambda x.x + ((\lambda y.y) z))[y/z] = \lambda x.x + ((\lambda y.y) y)\]

→ This is correct.

→ z is not in the scope of the \(\lambda y\) binder, so \(y\) is not captured when it is substituted.
\[(\lambda x.x + ((\lambda y.y) z))[x/z] = \lambda x.x + ((\lambda y.y) x)\]

→ This is incorrect.
→ \(z\) is in the scope of \(\lambda x\) before the substitution, so \(x\) is captured by the binder.
→ As ever, this can be solved by freshening the binder before substituting:

\[(\lambda a.a + ((\lambda y.y) z)[x/z] = \lambda a.a + ((\lambda y.y) x)\]
“Type soundness is often proved using two properties, called \textit{preservation} and \textit{progress}”. Define the \textit{preservation} property.
“Type soundness is often proved using two properties, called preservation and progress”. Define the preservation property.

→ **Preservation**: Typing is preserved under reduction.
   
   → More formally, if \( \vdash e : \tau \) and \( e \mapsto e' \), then \( \vdash e' : \tau \).

→ **Progress**: A well-typed term is either a value, or can take a reduction step (evaluation doesn’t get “stuck”)

   → More formally, if \( \vdash e : \tau \), then either \( e \) is a value \( v \), or there exists some \( e' \) such that \( e \mapsto e' \).

→ **Soundness**: A system is **sound** if it satisfies preservation and progress.

These seem to come up a lot – they’re worth knowing!
Consider the following rules which we might add to handle random number generation to a language that already has basic arithmetic:

\[
\begin{align*}
  e &\mapsto e' \\
  \text{randInt}(e) &\mapsto \text{randInt}(e') \\
  0 \leq n < \nu &\Rightarrow \text{randInt}(\nu) \mapsto n \\
  \nu \leq 0 &\Rightarrow \text{randInt}(\nu) \mapsto 0 \\
\end{align*}
\]

\[\Gamma \vdash e : \tau\]

\[\Gamma \vdash e : \text{int} \Rightarrow \Gamma \vdash \text{randInt}(e) : \text{int}\]

Is this system sound? Briefly explain why or why not.
Does the system satisfy preservation? If something reduces, does it have the same type?

→ Yes: the type is int before and after reduction.

Does the system satisfy progress? Can we always reduce?

→ Yes: if `randInt` is evaluating a value, then all values accounted for by the last two rules. If evaluating a subexpression, we can assume it takes a step, and thus conclude with the first rule.
15/16 Resit Paper: 2(e)

$e \leadsto e'$

\[
\frac{e \leadsto e'}{} \quad \frac{0 \leq n < v}{\text{randInt}(v) \mapsto n} \quad \frac{v \leq 0}{\text{randInt}(v) \mapsto 0}
\]

$\Gamma \vdash e : \tau$

$\Gamma \vdash e : \text{int}$

$\Gamma \vdash \text{randInt}(e) : \text{int}$

How would we prove this formally?

→ Preservation: by induction on $e \leadsto e'$.

→ Progress: by induction on $\vdash e : \tau$.  

Is this system sound?
Is this system sound?

→ No.

→ Preservation holds: if we take a reduction step, we still end up with a float.

→ Progress **does not hold**: we cannot reduce $v_1 \div 0$ since no rules match, yet $v_1 \div 0$ is not a value.