# COMP302 - Midterm 3 Review Tutorial Notes (with Solutions)

L Denney

April 2025

## Parsing

We define the syntax of a small portion of MiniCaml with a new extension: The ability to delay computation with the keyword `delay`.
`delay` $e$ allows the programmer delay the evaluation of the expression $e$. We say `delay` $e$ has type Susp $A$ when $e$ has type $A$. We can allow $e$ to be evaluated by using a "Let-style matching elimination" on an expression $e_1$ of type Susp $A$ with `let delay` $x = e_1$ `in` $e_2$. This lets us use the expression wrapped in the delay freely in $e_2$.

```
type ident = string

type tp =
| Int
| Arrow of tp * tp
| Susp of tp

type exp =
| ConstI of int
| Var of ident                    (* x *)
| Fn of ident * tp * exp          (* fn x : t => e *)
| Delay of exp                    (* delay e *)
| LetDelay of ident * exp * exp   (* let delay x = e1 in e2 *)
```

### Exercise

Let's try to parse this into an AST (just by hand)!

1. `fun (x : Susp int) -> let delay y = x in y`

   Answer: Fn ( "x", Susp Int , LetDelay ("y", Var ("x"), Var ("y"))

2. `fun (x : int => int) -> fun (y: int) -> 5`

   Answer: Fn ("x" , Arrow (Int, Int), Fn ("y", Int, ConstI 5))

3. `let delay x = delay (fun (y:int) -> y) in x`

   Answer: LetDelay ("x", Delay (Fn ("y", Int, Var "y")), Var "x")

## Type Inference

Static Semantics for `delay` (typing rules)

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash \texttt{delay}\ e : \text{Susp}\ A}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Susp}\ A \qquad \Gamma, x : A \vdash e_2 : B}{\Gamma \vdash \texttt{let delay}\ x = e_1\ \texttt{in}\ e_2 : B}$$

### Exercise

Let's extend `infer` from class with the cases for the new typing rules above.

```
exception TypeError of string
type ident = string
type ctx = (ident * tp) list

let rec infer (c : ctx) (e: exp) : tp =
  match e with
  | ConstI i -> Int
  | Var x -> (match List.assoc_opt x c with
    | None -> raise (TypeError "unbound variable")
    | Some t -> t)
  | Fn (x, t, e) -> Arrow( t, infer ((x, t):: c) e)
  | Delay e -> Susp (infer c e)
  | LetDelay (x, e1, e2 ) ->
      (match infer c e1 with    (* check that e1 has type Susp A *)
      | Susp a -> infer ( (a, t) :: c ) e2
      | _ -> raise (TypeError "LetDelay requires e1 to be of type Susp t"))
```

## Evaluation

Operational Semantics for `delay` (evaluation rules)

$$\frac{t \Downarrow \texttt{delay}\ t' \qquad [t'/x]s \Downarrow v}{\texttt{let delay}\ x = t\ \texttt{in}\ s \Downarrow v}$$

$$\frac{}{\texttt{delay}\ e \Downarrow \texttt{delay}\ e}$$

### Exercise

We are given a function `subst` that does our substituting for us as expected. Using the above operational semantics and `subst` (when needed), implement the cases for our new expression constructors to `eval` from Homework 9.

```
(* outputs the result of [d/z]e *)
let rec subst ((d, z) : exp * ident) (e : exp) : exp =
    failwith "imagine this is implemented"

exception EvaluationStuck

let rec eval (e : exp): exp =
  match e with
  | Var _ -> raise EvaluationStuck
  | ConstI i -> e
  | Fn (_,_,_) -> e         (* values evaluate to themselves!*)
  | Delay e -> Delay e      (* delay e is a value (it shouldnt take any steps because
                               we \want\ the computation to be suspended) :) *)
  | LetDelay (x, e1, e2 ) ->
    (match eval e1 with
    | Delay e1' -> eval (subst (e1', x)  e2)
    | _ -> raise EvaluationStuck)
```

# Substitution

We have seen that the names of bound variables don't matter, but why can't we say the same thing about free variables? We can build an intuition for this by remembering that valid programs must have all of their variables bound (to convince yourself, consider what would happen if you encounter a variable that refers to nothing when running a program). So we can imagine that when we encounter a free variable during substitution we know, assuming we are looking at a valid program, it must be bound somewhere. But here it *appears* free because we are in a sub-term of the whole program and cannot see it's binder. If we were to rename the free variable, we wouldn't be able to rename its binder to match! This would have the effect of "unbinding" it from its binder (bad).

## Renaming a free variable (Bad)

Here is an example demonstrating the issue with renaming a free variable.
Imagine we want to do some substituting on this example:

$$\texttt{fun}\ x \rightarrow (\texttt{fun}\ y \rightarrow x + y)$$

We can ignore the actual substituting going on and instead consider what the substitution can "see".
Now imagine we are in the recursive call of the substitution looking at the body of the first function

$$(\texttt{fun}\ y \rightarrow x + y)$$

It looks like x is free right now! What if we were to rename it to some new name like z? Then we would get

$$(\texttt{fun}\ y \rightarrow z + y)$$

But once we return from this recursive call, we will be left with

$$\texttt{fun}\ x \rightarrow (\texttt{fun}\ y \rightarrow z + y)$$

and this is **not** equivalent to the one we started with!!

**Remember:** substitutions only target free variables

## Renaming a bound variable (Ok)

Here is an example demonstrating that there is no issue renaming a bound variable given that the new name is not that of a free variable of the expression being subbed in ($[e/x]$ s.t. the new name $\notin FV(e)$ ).

Now imagine we wanted to renamed a **bound** variable (say y to w) in our rec call ($\texttt{fun}\ y \rightarrow x + y$) as above. Then after renaming we will have

$$(\texttt{fun}\ w \rightarrow x + w)$$

And after we return from the rec call we get

$$\texttt{fun}\ x \rightarrow (\texttt{fun}\ w \rightarrow x + w)$$

Notice that the program with the bound variable renaming and the initial program are the *same*. (They preform the same when used)

$$\texttt{fun}\ x \rightarrow (\texttt{fun}\ w \rightarrow x + w) = \texttt{fun}\ x \rightarrow (\texttt{fun}\ y \rightarrow x + y)$$

Conclusion: the names of bound variables don't matter.

## How to do substitution

$[e_1/x]e_2$ is read "substitute $e_1$ for $x$ in $e_2$" and it means replace all occurrences of $x$ in $e_2$ with $e_1$.

But there are two special cases to look out for when doing substitution where things can get weird:

1. When the **target** of the substitution is a **bound variable**.
   This one case is concerned with the right side of the substitution [e/**x**]
   Solution → We drop the substitution.

   e.g. $x$ is bound in $e_2$ by the `let-in` expression, so we don't propagate the substitution in there.

   $$[e/x](\texttt{let}\ x = e_1\ \texttt{in}\ e_2) \rightarrow \texttt{let}\ x = [e/x]e_1\ \texttt{in}\ e_2$$

2. Variable capture: the expression **being substituted in** contains a variable that will become bound once the substitution is complete.
   This one looks at the subject of the substitution [**e**/x] , and considers it's free variables.
   Solution → We rename any bound variables that would capture any free variables in $e$ after the substitution.
   i.e. in $[e/x]e'$, rename any bound variables in $e'$ that share a name with a free variable in $e$

Big idea: preform a substitution without making free variables bound or bound variables free or changing what the bound variables are bound to.

We know that we can always rename bound variables (along with their binder) safely to avoid variable capture, but we want to do this efficiently by renaming the fewest bound variables required.

## Exercise

For each example below

1. circle the free variables
2. preform capture avoiding substitution with minimal renaming

(a) $[4/y](\texttt{let}\ x = y * 5\ \texttt{in}\ (x + y + z, 10))$

(b) $[8/b](\texttt{let}\ a = \texttt{fun}\ c \rightarrow (\texttt{if}\ x\ \texttt{then}\ b\ \texttt{else}\ c)\ \texttt{in}\ a\ (b + 100))$ a is free

(c) $[y/x](\texttt{let}\ x = \texttt{fun}\ y \rightarrow y\ \texttt{in}\ x\ y)$

(d) $[z/x](\texttt{let}\ x = y + x\ \texttt{in}\ x + 3)$

**Answer:**

(a) $[4/y](\texttt{let}\ x = \textcircled{y} * 5\ \texttt{in}\ (x + \textcircled{y} + \textcircled{z}, 10))$
$\longrightarrow \texttt{let}\ x = 4 * 5\ \texttt{in}\ (x + 4 + z, 10)$

(b) $[8/b](\texttt{let}\ a = \texttt{fun}\ c \rightarrow (\texttt{if}\ \textcircled{x}\ \texttt{then}\ \textcircled{b}\ \texttt{else}\ c)\ \texttt{in}\ a\ (\textcircled{b} + 100))$
$\longrightarrow (\texttt{let}\ a = \texttt{fun}\ c \rightarrow (\texttt{if}\ x\ \texttt{then}\ 8\ \texttt{else}\ c)\ \texttt{in}\ a\ (8 + 100))$

(c) $[y/x](\texttt{let}\ x = \texttt{fun}\ x \rightarrow x\ \texttt{in}\ x\ y)$
$\longrightarrow \texttt{let}\ x = \texttt{fun}\ z \rightarrow z\ \texttt{in}\ x\ y$ (we rename x to z to avoid variable capture)

(d) $[z/x](\texttt{let}\ x = y + \textcircled{x}\ \texttt{in}\ x + 3)$
$\longrightarrow \texttt{let}\ x = y + z\ \texttt{in}\ x + 3$

# Subtyping

## Arrow Types

The most complicated part of subtyping is arrow types so that's what we will focus on.

$$\frac{T_1 \leq S_1 \quad S_2 \leq T_2}{S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2}$$

**Intuition for arrows:** $S = S_1 \rightarrow S_2$ is a subtype of $T = T_1 \rightarrow T_2$ if at runtime the interpreter can safely use $S$ when $T$ is expected. This means:

1. for all types of stuff that T can output, S outputs at least one of them *and* for all the types of stuff S can output, T can also output something of that type.
2. S can take in inputs of all the types T can (and more if it wants)

The result of this is we can use S anywhere that is asking for a T.

## Exercise

Consider these types with the following relations to each other:

$$\texttt{a} \leq \texttt{b} \quad \texttt{d} \leq \texttt{a} \quad \texttt{d} \leq \texttt{c} \quad \texttt{e} \leq \texttt{d}$$

Now consider the type $T := \texttt{a} \rightarrow \texttt{d} \rightarrow \texttt{b}$
What is its relationship with each of the following 4 types: (unrelated, $\leq$, or $\geq$ )

(a) $\texttt{a} \rightarrow \texttt{d} \rightarrow \texttt{a}$

(b) $\texttt{c} \rightarrow \texttt{d} \rightarrow \texttt{a}$

(c) $\texttt{a} \rightarrow \texttt{b} \rightarrow \texttt{b}$

(d) $\texttt{b} \rightarrow \texttt{e} \rightarrow \texttt{b}$

**Answer:**

(a) (a) $\leq$ T

(b) unrelated

(c) (c) $\leq$ T

(d) T $\leq$ (d)