CSE114A lecture 6

Agenda:

- Recap: variable scope, free and bound variables ✓
- Computing the set of free variables of an expression ✓
- Revisiting the β-rule ✓
- If time: capture-avoiding substitution
- If time: Decrementing numbers!

Variable scope

A **binding** is an association of a *name* to some *entity*.

The *scope* of a binding in a program is the part of the program where the name can be used to refer to the entity.

```
\x -> e
```

`\x`: binder — `e`: body: scope of x!

Any occurrence of x in e is *bound* by the binder `\x`.

An occurrence of a variable is *free* if it's not bound!

```
x y
```

(x: Free occurrence of x.)

```
\x -> x
```

```
\x -> (\y -> x)
```

x is bound by `\x`.

```
(\x y -> y) x
```

(end of the scope of `\x`! — x: Free occurrence of x)

Remember:

```
  \x y -> y
= \x -> (\y -> y)
```

Computing the set of free variables of an expression

(without *evaluating* the expression!)

$$e ::= x \mid \backslash x \to e \mid e_1\ e_2$$

- x: variables
- $\backslash x \to e$: abstractions, aka function definitions
- $e_1\ e_2$: applications, aka function calls

Let's define $FV(e)$ to be the set of variables that occur free in e.

$$FV(x) = \{x\}$$

$$FV(\backslash x \to e) = FV(e) - \{x\}$$

($\backslash x$: the binder; $-$: set difference)

$$FV(e_1\ e_2) = FV(e_1) \cup FV(e_2)$$

($\cup$: set union)

> Quiz question 1:
> What do you think $FV(e_1\ e_2)$ is?
> Hint: you can use set operations

What we just did was a *static analysis* of e!
(static: without running the program)

An expression with no free variables is said to be *closed*.

A closed expression is also known as a *combinator*.

> Quiz question 2:
> Write the simplest combinator you can think of!

`\x -> x` !

Revisiting the beta rule

$$(\backslash x \to e_1)\ e_2 \ \texttt{=b>}\ e_1[x := e_2]$$

What if you had...

```
(\x -> (\x -> x)) y
```

(the x in the body refers to: not this... (the outer `\x`) — this! (the inner `\x`))

What should we get?

If we replaced *all* occurrences of x in the function's body with y, we'd get

```
=b> \x -> y
```

which isn't right!

What we really want is for $e_1[x := e_2]$ to mean "replace all *free* occurrences of x with e_2 in e_1".

Wait... is that it? What about this?

```
(\x -> (\y -> x)) y
=b> \y -> y
```

(y: argument)

What went wrong?
y occurred free in the *argument*, but it got *captured* by the `\y` binder in the body of the function!

So we need to be even more subtle about what $e_1[x := e_2]$ means:
(substitution)

It means:

"e_1, but with all free occurrences of x in e_1 replaced with e_2,
** *as long as* no variables that occur free in e_2 get captured by binders in e_1."

If any variables *do* get captured, the substitution operation is *undefined*!

So if we have an expression like

```
\x -> (\y -> x)
```

and we want to apply it to y, we have to *rename* the formal parameter y in the binder `\y` so that we can take a `=b>` step.

E.g. we could rename y to z.

```
(\x -> (\y -> x)) y
```

rename:

```
(\x -> (\z -> x)) y
```

now step:

```
\z -> y
```

We can, if we want, now formally define capture-avoiding substitution.

Let's do it

$$e_1[x := e_2]$$

(from $(\backslash x \to e_1)\ e_2$)

- case: e_1 is a variable.

  if e_1 is x:

  $(\backslash x \to x)\ e_2$ `=b>` e_2

  otherwise (e_1 isn't x):

  $(\backslash x \to y)\ e_2$ `=b>` y

- case: e_1 is an application $e_3\ e_4$:

  $$(e_3\ e_4)[x := e_2] = (e_3[x := e_2])\ (e_4[x := e_2])$$

- case: e_1 is a function $\backslash x \to e_3$ with x as its formal parameter:

  $$(\backslash x \to e_3)[x := e_2] = \backslash x \to e_3$$

  (e_3: no free occurrences of x in here!)

> Quiz question 3:
> what's $(\backslash x \to e_3)[x := e_2]$?

($(\backslash x \to e_1)\ e_2$ `=b>` ?)

Finally: if e_1 is a function with something other than x as its formal parameter:

$$(e_1 == \backslash y \to e_3)$$

we have to check if the formal parameter is actually in the free variables of e_2.