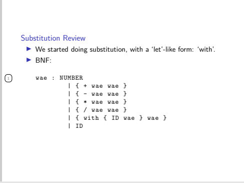
Substitution Review

- We started doing substitution, with a ‘let’-like form: ‘with’.
- BNF:

```
wae : NUMBER
    | { + wae wae }
    | { - wae wae }
    | { * wae wae }
    | { / wae wae }
    | { with { ID wae } wae }
    | ID
```



Substitution Review

- We started doing substitution, with a 'let'-like form: 'with'.
- BNF:

```
wae : NUMBER
        | { + wae wae }
        | { - wae wae }
        | { * wae wae }
        | { / wae wae }
        | { with { ID wae } wae }
        | ID
```

1. + Avoid writing expressions twice.
   + More expressive language (can express identity).
   + Avoid redundant redundancy.

- After lots of attempts, we defined substitution:
  e[v/i] – To substitute an identifier ‘i’ in an expression ‘e’ with an expression ‘v’, replace all instances of ‘i’ that are free in ‘e’ with the expression ‘v’.
- We then extended the AE evaluation rules:

```
2  eval(...) = ... same as the AE rules ...
   eval({with {x E1} E2}) = eval(E2[eval(E1)/x])
   eval(ID) = error!
```

- One thing we glossed over last time is the type error in this scheme. What's wrong with this picture:

3
```
(has-type subst : (WAE Symbol WAE -> WAE ))
(has-type eval : (WAE -> Number))
```

- We hacked around the type problem by wrapping the results of subst:

4
```
(define (eval expr)
  (type-case WAE expr
       ⋮
    [(With bound-id named-expr bound-body)
     (eval (subst bound-body
                  bound-id
                  (Num (eval named-expr))))]))
```

Formal Specs

Between plain English and code: (‘N’ is a NUMBER, ‘E1’, ‘E2’ are WAEs, ‘x’ is some ID, ‘y’ is a **different** ID)

```
N[v/x]                  = N

{+ E1 E2}[v/x]          = {+ E1[v/x] E2[v/x]}

{- E1 E2}[v/x]          = {- E1[v/x] E2[v/x]}
```

```
{* E1 E2}[v/x]           = {* E1[v/x] E2[v/x]}

{/ E1 E2}[v/x]           = {/ E1[v/x] E2[v/x]}

y[v/x]                   = y
x[v/x]                   = v

{with {y E1} E2}[v/x] = {with {y E1[v/x]} E2[v/x]}

{with {x E1} E2}[v/x] = {with {x E1[v/x]} E2}
```

- a formal definition of ‘eval’:

```
eval(N)          = N
eval({+ E1 E2}) = eval(E1) + eval(E2)
eval({- E1 E2}) = eval(E1) - eval(E2)
eval({* E1 E2}) = eval(E1) * eval(E2)
eval({/ E1 E2}) = eval(E1) / eval(E2)
eval(x)         = error!
eval({with {x E1} E2}) = eval(E2[eval(E1)/x])
```

Lazy vs Eager Evaluation

- There are two basic approaches for evaluation: eager and lazy.
- In lazy evaluation (without sharing), bindings are used for unevaluated code; computation is still duplicated.
- Which evaluation method did our evaluator use? The relevant piece of formalism (and corresponding code) is:

```
eval({with {x E1} E2}) = eval(E2[eval(E1)/x])

[(With bound-id named-expr bound-body)
 (eval (subst bound-body
              bound-id
              (Num (eval named-expr))))]
```

Making evaluation lazy

- We can do the substitution purely syntactically:

9

```
eval({with {x E1} E2}) = eval(E2[E1/x])
```

- and in the code:

10

```
(define (eval expr)
  (type-case WAE expr
        ⋮
     [With (bound-id named-expr bound-body)
      (eval (subst bound-body
                   bound-id
                   named-expr))] ; <- no eval
                                 ; no wrapping
        ⋮
                                 ))
```

testing our lazy evaluator

11

```
(test (run `5) 5)
(test (run `{+ 5 5}) 10)
(test (run `{with {x {+ 5 5}} {+ x x}}) 20)
(test (run `{with {x 5} {+ x x}}) 10)
(test (run `{with {x {+ 5 5}} {with {y {- x 3}} {+
   y y}}}) 14)
(test (run `{with {x 5} {with {y {- x 3}} {+ y
   y}}}) 4)
(test (run `{with {x 5} {+ x {with {x 3} 10}}}) 15)
(test (run `{with {x 5} {+ x {with {x 3} x}}}) 8)
(test (run `{with {x 5} {+ x {with {y 3} x}}}) 10)
(test (run `{with {x 5} {with {y x} y}}) 5)
(test (run `{with {x 5} {with {x x} x}}) 5)
(test/exn (run `{with {x 1} y}) "free identifier")
```

Comparing the lazy and eager evaluators:

12
```
; lazy version
 (trace eval)
 (run `{with {x {+ 1 2}} {* x x}})
```

13
```
; eager version
 (trace eval)
 (run `{with {x {+ 1 2}} {* x x}})
```

- What can we guess about efficiency?
- Is there any program that will run differently in the two languages?
- The main feature of the lazy evaluator is that it is not evaluating the named expression until it is actually needed.

- What can we guess about efficiency?
- Is there any program that will run differently in the two languages?
- The main feature of the lazy evaluator is that it is not evaluating the named expression until it is actually needed.

1. We saw that our simple lazy evaluation strategy can cause duplicate work at runtime. Languages with rely heavily on lazy evaluation will typically introduce some sharing mechanism for unevaluated (or partially evaluated) expressions to avoid this.

Unused bound identifiers

14
```
(run `{with {x {/ 8 0}} 7})
```

- What about?

15
```
(run `{with {x y} 7})
```

- Why might we think about these as different? Consider

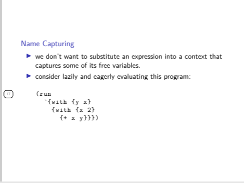
16
```
(run `{with {y 0}
          {with {z y}
                {with {x {/ 8 z}} 7}}})
```

Name Capturing

- we don't want to substitute an expression into a context that captures some of its free variables.
- consider lazily and eagerly evaluating this program:

```
(run
  `{with {y x}
    {with {x 2}
      {+ x y}}})
```

2019-02-09



Name Capturing

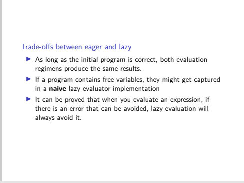
- we don't want to substitute an expression into a context that captures some of its free variables.
- consider lazily and eagerly evaluating this program:

```
(run
  `{with {y x}
    {with {x 2}
      {+ x y}}})
```

1. With the eager evaluator, name capture is not a problem because by the time we do the substitution, the named expression should not have free variables that need to be replaced.

Trade-offs between eager and lazy

- As long as the initial program is correct, both evaluation regimens produce the same results.
- If a program contains free variables, they might get captured in a **naive** lazy evaluator implementation
- It can be proved that when you evaluate an expression, if there is an error that can be avoided, lazy evaluation will always avoid it.



Trade-offs between eager and lazy

- As long as the initial program is correct, both evaluation regimens produce the same results.
- If a program contains free variables, they might get captured in a **naive** lazy evaluator implementation
- It can be proved that when you evaluate an expression, if there is an error that can be avoided, lazy evaluation will always avoid it.

1. This capturing is a bug
2. + On the other hand, lazy evaluators are usually slower than eager evaluator, so it's a trade-off.
3. Note that with lazy evaluation we say that an identifier is bound to an expression rather than a value. (Again, this is why the eager version needed to wrap 'eval's result in a 'Num' and this one doesn't.)

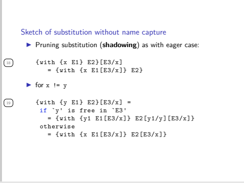
Sketch of substitution without name capture

- Pruning substitution (**shadowing**) as with eager case:

18
```
{with {x E1} E2}[E3/x]
   = {with {x E1[E3/x]} E2}
```

- for `x != y`

19
```
{with {y E1} E2}[E3/x] =
 if `y' is free in `E3'
   = {with {y1 E1[E3/x]} E2[y1/y][E3/x]}
 otherwise
   = {with {x E1[E3/x]} E2[E3/x]}
```



Sketch of substitution without name capture

- Pruning substitution (**shadowing**) as with eager case:

18
```
{with {x E1} E2}[E3/x]
   = {with {x E1[E3/x]} E2}
```

- for `x != y`

19
```
{with {y E1} E2}[E3/x] =
 if `y' is free in `E3'
   = {with {y1 E1[E3/x]} E2[y1/y][E3/x]}
 otherwise
   = {with {x E1[E3/x]} E2[E3/x]}
```

1. You can see that this is much more complicated, and probably not correct yet.

de Bruijn Indices

- Name capture is a problem that should be avoided.
- Note that the only thing we use names for are for references.
- We don't really care what the name is:

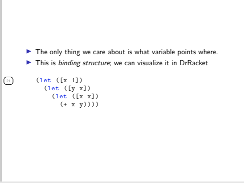
```
{with {x 5} {+ x x}}
{with {y 5} {+ y y}}

(define (foo x) (list x x))
(define (foo y) (list y y))
```

- The only thing we care about is what variable points where.
- This is *binding structure*; we can visualize it in DrRacket

```
(let ([x 1])
  (let ([y x])
    (let ([x x])
      (+ x y))))
```

2019-02-09



- The only thing we care about is what variable points where.
- This is *binding structure*; we can visualize it in DrRacket

21
```
(let ([x 1])
  (let ([y x])
    (let ([x x])
      (+ x y))))
```

1. The sense in which these expressions are "obviously the same". is is called "alpha-equality").

- Idea: if all we care about is where the arrows go, then simply get rid of the names
- Instead of referencing a binding through its name, just specify which of the surrounding scopes we want to refer to.

- For example, instead of:

  22
  ```
  {with {x 5} {with {y 6} {+ x y}}}
  ```

  we can use a new 'reference' syntax – "[N]" – and use this instead of the above:

  23
  ```
  {with 5 {with 6 {+ [1] [0]}}}
  ```

- The rules for [N] are – [0] is the value bound in the current scope, [1] is the value from the next one up etc.
  Of course, to do this translation, we have to know the precise scope rules.

- More complicated example:

24
```
{with {x 5} {+ x {with {y 6} {+ x y}}}}
```

is translated to:

25
```
{with 5 {+ [0] {with 6 {+ [1] [0]}}}}
```

(note how ‘x’ appears as a different reference based on where it appeared in the original code.)

- Even more subtle:

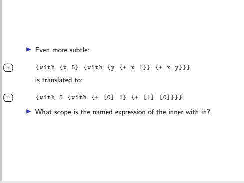
```
26    {with {x 5} {with {y {+ x 1}} {+ x y}}}
```

  is translated to:

```
27    {with 5 {with {+ [0] 1} {+ [1] [0]}}}
```

- What scope is the named expression of the inner with in?

2019-02-09



- Even more subtle:

```
{with {x 5} {with {y {+ x 1}} {+ x y}}}
```

is translated to:

```
{with 5 {with {+ [0] 1} {+ [1] [0]}}}
```

- What scope is the named expression of the inner with in?

1. The inner ‘with’ does not have its own named expression in its scope, so the named expression is immediately in the scope of the outer ‘with’.

de Bruijn Indices

- instead of referencing identifiers by their name, we use an index into the surrounding binding context.
- The major disadvantage, as can be seen in the above examples, is that the transformed code is not easy for humans to understand.
- Specifically, the same identifier is referenced using different numbers, which makes it hard to understand what some code is doing.
- Practically all compilers use de Bruijn indices for compiled code (think about stack pointers).