A) this.x = 3

B) o.x = 3

→ C) x = 3

object mutation

Variable mutation

x = 4 ← var mut

Aliasing {

y = o

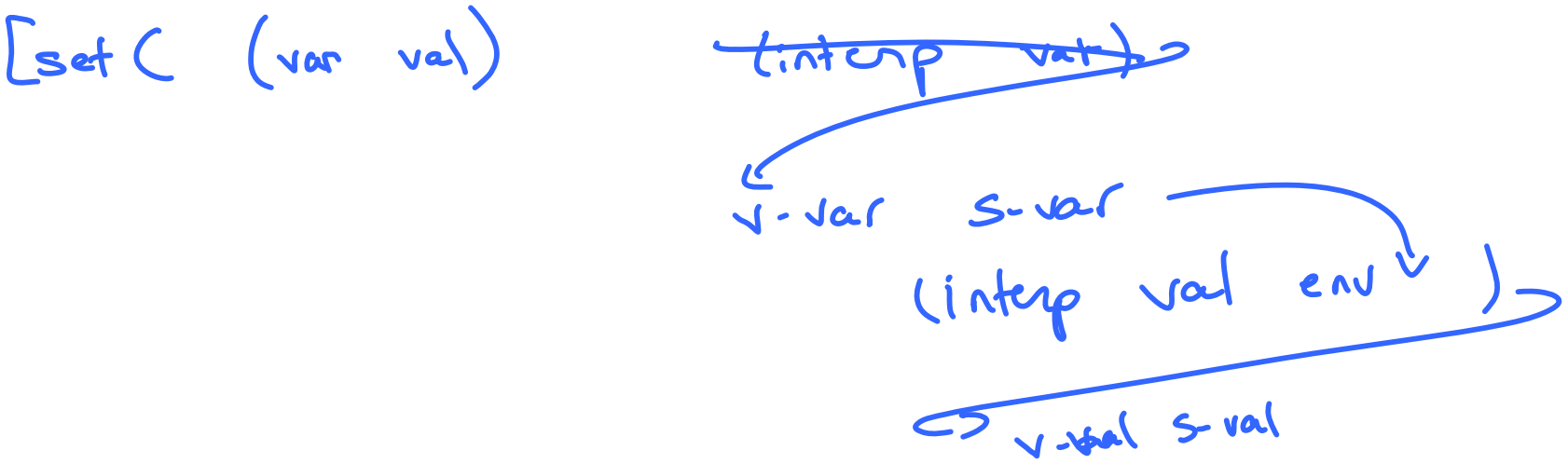
o.w = 4;

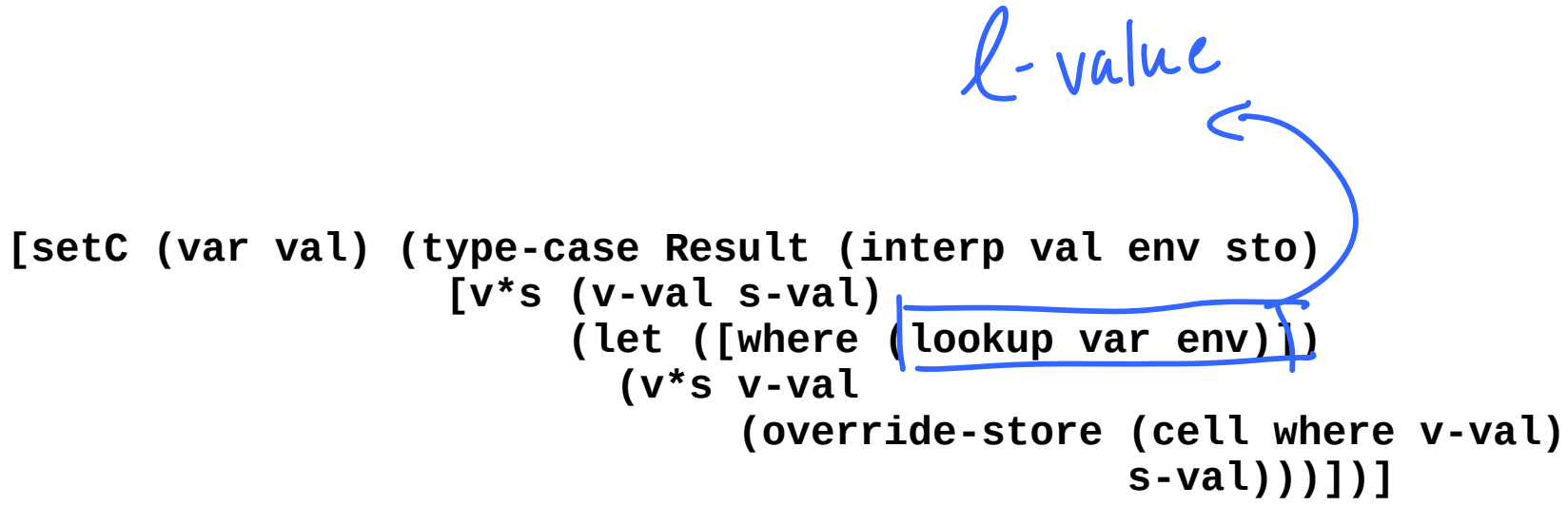
y.w

```
(define-type Result [v*s (v : Value) (s : Store)])

(define (interp [expr : ExprC] [env : Env] [sto : Store]) : Result

  [varC (s : symbol)]
```

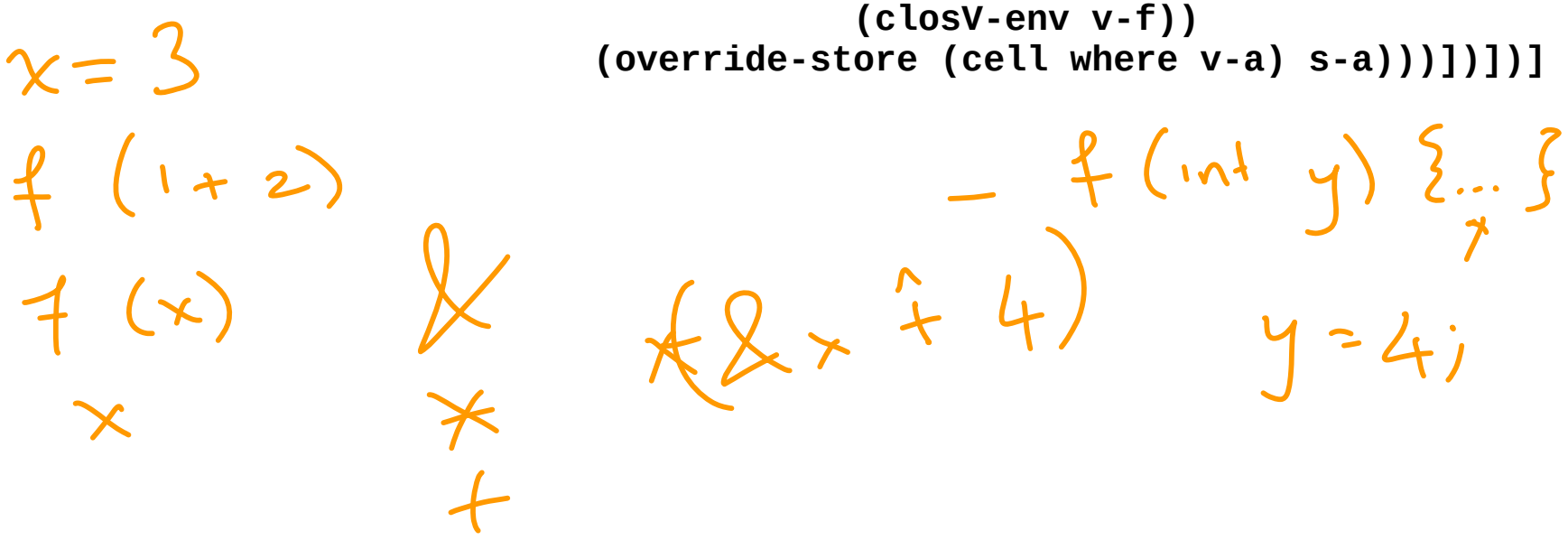
[setC [var : symbol] [val : ExprC]]

```
[idC (n) (v*s (fetch (lookup n env) sto) sto)]

[varC (n) (v*s (fetch (lookup n env) sto) sto)]

[setboxC (b v) (type-case Result (interp b env sto)
                 [v*s (v-b s-b)
                      (type-case Result (interp v env s-b)
```

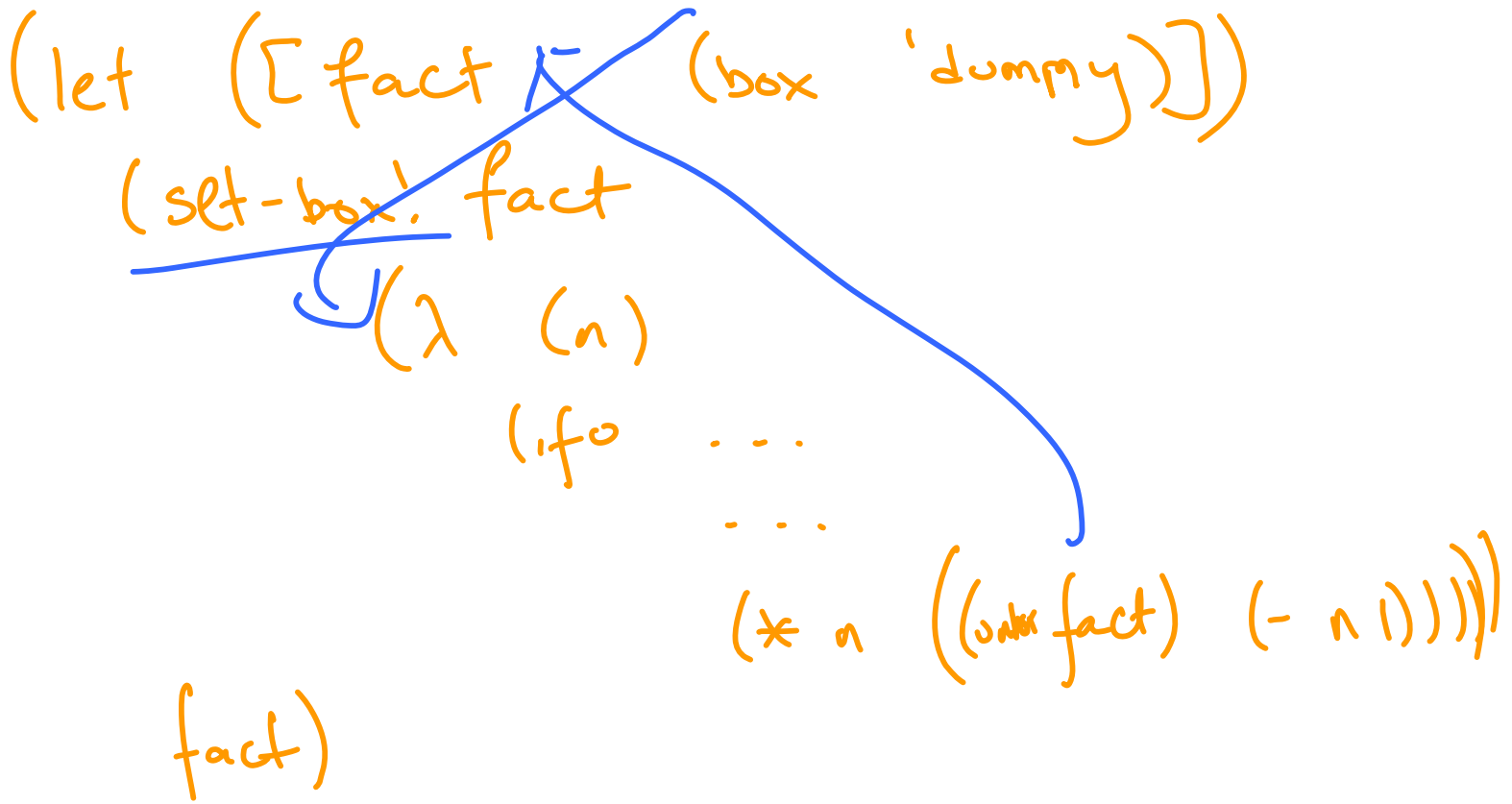



l-value
[setC (var val) (type-case Result (interp val env sto)
[v*s (v-val s-val)
(let ([where (lookup var env)])
(v*s v-val
(override-store (cell where v-val)
s-val)))])]

```
[appC (f a)
      (type-case Result (interp f env sto)
        [v*s (v-f s-f)
             (type-case Result (interp a env s-f)
               [v*s (v-a s-a)
                    (let ([where (new-loc)])
                      (interp (closV-body v-f)
                              (extend-env (bind (closV-arg v-f)
                                                where)
                                          (closV-env v-f))
                              (override-store (cell where v-a) s-a)))])])]
```



```
(let ([fact (λ (n)
              (if0 n
                   1
                   (* n (fact (+ n -1)))))])
  (fact 0))
```


(let ([fact (box 'dummy)])
(set-box! fact
(λ (n)
(if ...
...
(* n ((unbox fact) (- n 1))))))
fact)