COMPUT325: Meta-interpretation

Dr. B. Price and Dr. R. Greiner

21st October 2004

Introduction

- λ-calculus fully expresses computations of any programming language
- Is λ-calculus sufficiently expressive to express itself?

Implementing λ-calculus

- What would be required to automate λ-calculus representation and evaluation
  - representation for constants, applications and function definitions
  - Function for checking types of data
  - Functions for creating and accessing components of representations
  - Functions for λ-calculus evaluation
    - checking for free variables
    - renaming variables
    - performing substitutions
  - garbage collection

Why Garbage Collection

- No imperative assignment $\rightarrow$ no side-effects
- Efficiency maintained by shared references
- Sharing $\rightarrow$ function arguments may be shared by others

$(\lambda$x | (CONS $\underbrace{\texttt{(CONS 2 x)}}_{\texttt{x shared?}}$ (CONS 3 x)) $\underbrace{\texttt{(CONS 1 nil)}}_{\texttt{allocated CONS}}$

- Function cannot tell if it is safe to modify arguments (i.e. cannot deallocate!)
- But functions must allocate memory for new values
- Recursive loops could quickly consume all memory
- Garbage collectors analyze *global* pattern of dependencies to safely deallocate data

More on Memory Management

- "primitive values" with no shared sub-components can be passed by value - eliminating memory allocation
- static analysis of programs can detect arguments that are used only once (linearity)
- programs can then be optimized to do
  - imperative in-place modification when it is safe
  - deterministic deallocation of memory to avoid garbage generation
- For small toy examples, we can ignore garbage collection issues

Representation: λ-Calculus BNF

- What do we have to represent?

⟨expression⟩:=⟨identifier⟩ | ⟨application⟩ | ⟨function⟩

⟨identifier⟩ := a | b | c | ···

⟨application⟩:= "(" ⟨expression⟩ ⟨expression⟩ ")"

⟨function⟩ := "(λ" ⟨identifier⟩ "|" ⟨expression⟩ ")"

Primitiveλ-Calculus Representation I

- In λ-calculus, all data types are represented as λ expressions
- Need a way to distinguish: identifier, application, function
- Use a cons cell where `FIRST` is type, and `SECOND` is data
- Let the integers 0, 1, 2 denote identifiers, applications and function defs respectively
  - Let Φ be the appropriate λ-calculus representation

```
[0  Φ]   ;; an identifier
[1  Φ]   ;; an application of functions
[2  Φ]   ;; a function definition
```


Primitiveλ-Calculus Representation II

- Use cons cell type marker with Church integers for identifiers
  - Instead of `x,y,z` we use integer identifiers
  - To discriminate from numeric integers, write `$0, $1, $2, ...`
  - Where `$0` is type-marked identifier with church number 0
    i.e. `$0`$\equiv$`cons( 0 , 0 )`, `$1`$\equiv$`cons( 0 , 1 )`
    (0 = type, 0 = id; 0 = type, 1 = id)
- Use cons cell type marker with cons cell for applications
  - Consider application of `a` to `b`, (`a b`)
  - To discriminate from lists, write application (a b) as $(a b)

$\equiv$`$($0 $1)`
$\equiv$`cons( 1 ,cons( cons( 0 , 0 ) ,cons( 0 , 1 )))`
(1 = type; cons(0 , 0) = type, id; cons(0 , 1) = type, id)

Primitiveλ-Calculus Representation III

- Again, use CONS cell for function definition:

```
(λa | (a b))
≡$(λ$0 | $( $0 $1) )
≡cons( 2 ,        ;; Type marker for function def
       type
    cons(                  ;; Cons of parm and body
      cons( 0 , 0 )           ;; Parameter a
            type  id
      cons( 1 ,               ;; Type for application
            type
        cons(cons( 0 , 0 ) ,cons( 0 , 1 )) ))) ;; Body (
                   type  id        type  id
```


Creating Representations I

Using abstract programming idioms

```
new-id(last-id)
  ;; create a new identifier with type marker
  ≡ cons(0,successor(second(last-id)))

new-app(function,argument)
  ;; create a new function application with type
  ≡cons(1,cons(function,argument))

new-def(parameter, body)
  ;;create a new function definition with type
  ≡ cons(2,cons(parameter,body))
```

Creating Representations II

```
(λa | (a b) ) c
≡ LET a = 0 IN
     LET b = new-id(a) IN
       LET c = new-id(b) IN
         new-app(
           new-def(a, new-app(a,b)),
           c)
```


Representation of Type Predicates

Predicates using abstract programming idioms

- Recall: all datatypes are of the form: `(type, value)`

```
is-id(⟨E⟩)
  ;; True if ⟨E⟩ is constant identifier
  ≡ IF car(⟨E⟩)=0 THEN T ELSE F

is-app(⟨E⟩)
  ;; True if ⟨E⟩ is constant identifier
  ≡ IF car(⟨E⟩)=1 THEN T ELSE F

is-func(⟨E⟩)
  ;; True if ⟨E⟩ is constant identifier
  ≡ IF car(⟨E⟩)=2 THEN T ELSE F
```

Accessing Representations

Abstract idioms for datatypes of the form `(type,value)`

- Application Accessors for `(type (function argument))`

  `get-func(A)` $\equiv$ `car(cdr(A)) ;ie funct of application`

  `get-arg(A)` $\equiv$ `cdr(cdr(A)) ;ie arg of application`

- Function Definition Accessors for `(type (parameter body))`

  `get-parm(F)` $\equiv$ `car(cdr(F)) ;ie get` λ `parameter`

  `get-body(F)` $\equiv$ `cdr(cdr(F))`

λ-calculus Evaluation Function

- Implement λ-evaluation as 3 functions:
  - eval: takes a λ-calculus expression and returns its evaluation
  - apply: applies a function to an argument
  - subs: substitutes an expression for a constant in an expression
- Implementations are given in abstract programming notation

λ-Calculus Eval Function

```
eval(⟨E⟩) ≡

  IF is-id(e)
  THEN ;;⟨E⟩≡f  : a constant
    e

  ELSE IF is-app(e)
    THEN  ;;⟨E⟩≡(⟨F⟩ ⟨A⟩) : application
      apply(get-func(e), get-arg(e))

    ELSE ;;⟨E⟩≡(λx | ⟨BODY⟩ ) : definition
      new-func(get-parm(e), eval(get-body(e)))
```

- Note: body of definitions are evaluated before use

Applicative-Order Apply Function

```
apply(⟨F⟩,⟨A⟩) ≡  ;; apply function ⟨F⟩ to argument ⟨A⟩
  LET b=eval(⟨A⟩) IN
    IF is-id(⟨F⟩)
    THEN ;;(⟨F⟩⟨A⟩)≡(f⟨A⟩)
  d    new-app(⟨F⟩, b)
    ELSE IF is-app(⟨F⟩)
      THEN ;;(⟨F⟩⟨A⟩)≡((⟨G⟩⟨C⟩)⟨A⟩)
        IF is-id(get-func(⟨F⟩))
        THEN new-app(
              new-app(get-func(⟨F⟩,eval(⟨C⟩)),
              b)
        ELSE apply(eval(⟨F⟩), b)

      ELSE ;;((λx|⟨G⟩)⟨A⟩)
        eval(subs(b, get-parm(⟨F⟩), get-body(⟨F⟩)))
```

λ-Calculus Substitution I

- In an application like $(\lambda x \mid (\lambda y \mid x))\ y$
  - argument x is a free variable that would get bound on substitution
  - so, formal parameter λy must be renamed
- In an application like $(\lambda y \mid y)\ x$
  - formal parameter λy does not have to be renamed
  - But, renaming λy does not alter meaning
- Simplification: Do not check for free parameters
  — always rename formal parameters

λ-Calculus Substitution II

```
subs(s,v,⟨E⟩)  ;; substitute s for var v in expression ⟨E⟩

  IF is-id(⟨E⟩)
  THEN ;; base case, either constant matches or not
    IF ⟨E⟩=v THEN s ELSE ⟨E⟩

  ELSE IF is-app(⟨E⟩)
  THEN ;; application, substitute within (⟨F⟩ ⟨A⟩)
    new-app( subs(s,v,get-func(⟨E⟩)),
             subs(s,v,get-arg(⟨E⟩)))
```

(continued on next slide ...)

λ-Calculus Substitution III

```
ELSE ;; Definition (λf|⟨B⟩) - check variable issues!

    LET f = get-parm(⟨E⟩) IN

      IF f=v
      THEN ;; var shadowed by formal parameter -> done!
        ⟨E⟩

      ELSE ;; always rename binding variable
         LET z=new-id() AND b = get-body(⟨E⟩) IN
            new-func(
              z, subs(s,v,        ;; beta substitution
                      subs(z,f,b))) ;; alpha renaming
```


Appyling λ-Calculus Evaluation

- To evaluate: (λ x | x) a

```
LETREC zero = (λ sz| z)
AND    successor = (λx (λsz|s(xsz))
AND    add =
⋮
AND    zerop =
⋮
AND    eval = ⟨BODY⟩
AND    apply = ⟨BODY⟩ AND subs = ⟨BODY⟩  IN

LET x = 0 IN
LET a = new-id(x) IN
   eval( new-app(new-func(x,x),a) )
```

λ-Calculus Evaluation Example I

- Here, we ignore underlying representation
- Just examine how Eval, Apply and Subs work together
- Square brackets avoid confusion with λ-C arguments

```
eval[ (λy | s) ]   ;; Case: function def
  new-func[  get-id[ (λy | s) ]
             eval[s] ]
    get-id[ (λy | s) ] → y
    get-body[ (λy | s) ] → s
  new-func[ y, s ]
→(λy | s)
```


λ-Calculus Evaluation Example II

```
eval[ ((λy | s) x) ]  ;; Case: application

   apply[ get-fun[ ((λy|s) x) ],
          get-arg[ ((λy|s) x) ]  ]

 ≡apply[ (λy | s), x ] ;; (definition, arg)

      eval[
         subs[  eval[x],
                get-id[ (λy | s) ]
                get-body[ (λy| s) ]  ]
      eval[ subs[ x, y, s ] ]
      eval[ s]
→ s
```

λ-Calculus Evaluation Example III

```
eval [ ( (λy|s) ((λy|s) x) ) ] ;; Case: application
  apply[ (λy|s), ((λy|s) x)  ] ;; Case: (Def, Arg)
    eval[ subs[ eval[((λy|s) x)], y, s ]
      eval[((λy|s) x)] ; case: application
        apply[ (λy | s), x ] ;; (definition, arg)
          eval[subs[eval[x], y ,s)]]
            eval[x] → x  ;; constant identifier
          eval[subs[ x, y, s ] ]
      → s
    eval[ subs[s,y,s] ]
      subs[s,y,s] → s
    eval[s]
→ s
```


λ-Calculus Evaluation as Function

- A *normal order version of apply* is required for recursive functions
- Need to add accumulator variables to pass forward next identifier number
  - Not conceptually difficult, but messes up code
- And that's it:
  - λ-calculus evalulation can be written as a λ-calculus expression
  - Therefore, λ-calculus evaluation is just another function
  - λ-calculus can be used to implement λ-calculus

Bootstrapping

- Functional languages can be written in abstract programming language
- Abstract programming has a simple translation to λ-calculus
- λ-calculus has simple syntax, evaluation rules and semantics
  - Simple to implement
  - Easy to show correctness
- Easy to prototype a new language
  - Define translation from new language to abstract programming language
  - Run new language on top of abstract programming layer
  - Write native code compiler in new language
  - Now can compile new language directly to platform
- Called bootstrapping

λ-Calculus Evaluation in Lisp

- Possible to implement λ-Calculus Evaluator in Lisp
- But: Lisp has:
  - basic datatypes: numbers, lists, constants
  - a type system with predicates: `'atom', 'consp'`
  - primitive functions: `+, -, cons, car, cdr`
- Can replace low-level λ-calculus idioms for numbers and lists with high-level Lisp implementations
- Do not need separate structure to represent type of data
- Requires
  - rewrite of creators, accessors and predicates
  - extra case in interpreter to intercept and call built-in functions directly
  - minor changes to other components

Efficiency Issues

- Consider the following example

```
(λx | IF T  THEN ((λy| (λz| y z) x) x)
            ELSE ((λy| y) x))            ) z
```

$\xrightarrow{\beta}$`[ z/x ] IF T THEN ((λy| (λz| y z) x) x) ELSE ((λy| y)`

- Followed generic β-reduction. Notice anything odd?
  - Substituted for both halves of IF statement
    even though ELSE is *never* used
  - Substitution involves rebuilding a copy of the expression
    - `(λz| y z)` rebuilt even though no x
- In `(λx| ( λ y| ( λz | ⟨E⟩)))`, expression ⟨E⟩ is rebuilt 3 times!

Lazy Substitution

- How do we avoid redundant substitutions?
  1. Note any parameter substitutions introduced by applications
     (keep in ordered list)
  2. Start processing the expression
  3. Perform substitution only if parameter encountered

Binding Lists

- Bindings list are a simple approach to efficient substitution
- Naive eval: substitute everything first, then eval

```
eval[ ( (λx| IF T THEN (λz|x) ELSE (λz | z x ))  y)]
[y/x] ( IF T THEN (λz|x) ELSE (λz| z x) )
→ IF T THEN (λz|y) ELSE (λz | z y)
eval[ IF T THEN (λz|y) ELSE (λz | z y) ] →(λz|y)
```

- Smart substitution: eval until substitution is needed, then substitute

```
eval[ ( (λx| IF T THEN (λz|x) ELSE (λz | z x ))  y)]
eval[ IF T THEN (λz|x) ELSE (λz | z x ), {x←y} ]
   eval[ T, {x←y}]
   eval[ (λz|x), {x←y}]
         eval[ x, {x←y}]→(λz|y)
```


Binding Parameters to Expressions

- Parameter value may in turn be an expression

```
eval[ (λx | (* 2 x)) (+ 3 2), {} ]
  eval[ (* 2 x) , {x←(+ 3 2)} ]
    eval[2,{x←(+ 3 2)}] → 2
    eval[x,{x←(+ 3 2)}]
       eval[ (+ 3 2) ] → 5
```

Bindings and Multiple Arguments

- Multiple bindings are added to bindings list in order of occurence

```
eval[ (λx | (λy |(+ x y)) ) 3 5, {} ]
  eval[ (λy |(+ x y)) 5, {x←3} ]
    eval[ (+ x y), {y←5, x←3} ]
      eval[x, {y←5, x←3}]
         eval[3]→ 3
      eval[y, {y←5, x←3}]
         eval[5]→ 5
    eval[ (+ 3 5) ]→5
```


Bindings and Shadowed Arguments

- Bindings looked up from left to right. First value found is used

```
eval[ (λx | (+ ((λx |(+ x x)) 5) x) 3, {} ]
eval[ (+ ((λx |(+ x x)) 5) x), {x←3} ]
  eval[ ((λx |(+ x x)) 5), {x←3} ]
    eval[(+ x x), {x←5, x←3} ]
       eval[x, {x←5, x←3} ] →5
       eval[x, {x←5, x←3} ] →5
    →10
  →10
  eval[ (+ 10 x), {x←3} ]
    eval[10, {x←3}]→10
    eval[x, {x←3}] → 3
  →13
```

Problems with Bindings and Free Variables I

```
eval[ (λy | (λx | + x y)) 4, {}]
eval[ (λx | + x y), {y←4}]
```

- No application here — cannot evaluate (λx| + x y) further
- But, should have y bound to 4
- Our simple interpreter actually handles this (but poorly):
  - evaluate λ-body: + x y in environment (y←4)
  - create new function with evaluated body (λx| ⟨BODY⟩)

```
eval[+ x y, {y←4}] → (+ x 4)
→ (λx| + x 4)
```

- Above solution breaks: See next slide!

Problems with Bindings and Free Variables II

```
eval[ (λy | (λy| (y y))) 4, {} ]
eval[ (λy| (y y)), {y←4} ]
  DO NOT DO THIS!
  → (λy | eval[ ( y y), {y←4} ]  )
   ≡ (λy | 4 4)
```

- Dynamic binding results in wrong answer! The "funarg" problem
- Could try to represent fact that y is bound in inner λ

```
eval[ (λy| (y y)), {y←4} ]
    DO NOT DO THIS!
   → (λy | eval[ ( y y), {y←y, y←4} ]  )
   → (λy | ( y y))
```

- *Solution Breaks in more complex cases*

Problems with Bindings and Free Variables III

- Imagine we define a function in a local context and return it:

```
(LET x=1 IN LET f(y)=x+y IN f) → f
```

- Now apply this function to x, where x=2

```
LET x = 2 IN
  (  (LET x=1 IN
        LET f(y)=x+y IN f)  x )
```

- Translates into λ-calculus as:

```
(λx |
    (  ((λx | (λy | + x y))  1)  x)
) 2
```

- Expected answer? Increment of 2 = 3

Problems with Bindings and Free Variables IV

```
LET x =2 IN ( (LET x=1 IN LET f(y)=x+y IN f)  x )

eval[ (λx |  ((λx|(λy|+x y)) 1) x)   2 ,{}]
Regular apply, make binding
eval[ ((λx | (λy | + x y)) 1) x, {x←2}]
  Apply (f a): eval f1, then eval a1, then apply f1 to a1
  f1=eval[ ((λx | (λy | + x y)) 1), {x←2}]
  Apply (f a): eval f2, then eval a2, then apply f2 to a2
    f2=eval[(λx | (λy | + x y)), {x←2}]
    Def:eval body (λy | + x y) for local bindings
      eval[ (λy | + x y), {x←2}]
       Def:eval body (+x y) for local bindings
      eval[ + x y, {x←2}] → + 2 y   OOPS!!
```

Problems with Bindings and Free Variables IV

- Having made the wrong substitution, we get the wrong answer!

```
    new-func(y,+ 2 y)⟶(λy|+2 y)
  f2=new-func(x,(λy|+2 y))⟶(λx|(λy|+2 y))
  a2=eval[1,{x⟵2}]=1
  apply[(λx|(λy|+2 y)),1]
f1=(λy|+2 y)
a1=eval[x,{x⟵2}]=2
apply[ (λy|+2 y), 2 ] ⟶4   WRONG!
```


Lexical Contex I

- Can you see why early Lisp's has Dynamic scoping?
  - Implementation used simple bindings lists.
- Dynamic scoping does not match λ-calculus semantics
- Difficult to debug and understand

Lexical Contex I

- In static scoping, need to save the context *in which a λ is defined*
- Otherwise we can get into a situtation where we make the wrong substitution

```
LET x = 2 IN
  ( (LET x=1 IN
       LET f(y)=x+y IN f)  x )
```

- In the above example `x=2` got substituted for the `x` in the inner λ
- This is because the function was *used* in a different context than it was *defined*
  - Defined in the magenta context
  - Used in the blue context

Closures

- The set of bindings that are active for a definition is called its *environment* or *context*
- An expression is "executed in" an environment
- An expression together with its environment is called a *closure*
- <closure>={expression, environment}
- By saving a closure with a λ we can ensure it evaluates to the same thing whenever and wherever it is executed
- Should be no free variables in a closure

Simple Application with Closures

```
eval[ (λx | x) 2 ,{}]
```
Regular apply: eval f1, eval a1, apply f1 to a1

```
f1 = eval[ (λx | x) ,{}]
```
Definition:make closure
```
f1 = <(λx | x) ,{}>

a1 = eval[ 2 ] = 2

apply[ f1, a1 ]
```
Eval f1 body in environment
with x=a1 and context of f1={}
```
eval[ x, {x←2}+{} ]
```

→2

- Seems like extra machinery, but useful in complex cases

Forming and Applying Closures

- Forming closures
  - Given definition (λp |⟨BODY⟩) defined in environment E
  - We form the closure <(λp |⟨BODY⟩),E>
- To apply closure <(λp |⟨BODY⟩),E> to argument A in context G
  - evaluate ⟨BODY⟩
  - in an environment = { p←A + E + G}

Trickier Application with Closures I

```
LET x=1 IN LET y=(λz|z+x) IN y(3)
eval[(λx|(λy|(y 3)) (λz|z+x)) 1, {}]
Regular apply, eval f1, eval a1, apply f1 to a1
  f1=eval[(λx|(λy|(y 3)) (λz|z+x)), {}]
  Definition:make closure
  f1=<(λx|(λy|(y 3)) (λz|z+x)),{}>
  a1=eval[ 1, {}] = 1
  apply(f1,a1)
  Eval f1 body with a1 and context of f1
  eval[(λy|(y 3)) (λz|z+x) ,{x=1}]
```


Trickier Application with Closures II

```
  eval[(λy|(y 3)) (λz|z+x) ,{x=1}]
    Regular apply, eval f2, eval a2, apply f2 to a2
    f2=eval[(λy|(y 3)),{x=1}]
    Definition:make closure
    f2=<(λy|(y 3)),{x=1}>
    a2=eval[(λz|z+x) ,{x=1}]
    Definition:make closure
    a2=<(λz|z+x) ,{x=1}>
    apply(f2,a2)
    Eval f2 body with a2 and context of f2
    a2 is a closure— parm y is bound to a closure
    eval[(y 3),{y=<(λz|z+x),{x=1}>,x=1}]
```

Trickier Application with Closures III

```
eval[(y 3),{y=<(λz|z+x),{x=1}>,x=1}]
```
Regular apply, eval f3, eval a3, apply f3 to a3
```
  f3=eval[y,{y=<(λz|z+x),{x=1}>,x=1}]
  f3=<(λz|z+x),{x=1}>
  a3=eval[3]=3
  apply[f3,a3]
```
Eval f2 body with a2 and context of f2
```
    Eval[z+x,{z=3,x=1}]
```
Regular apply...
```
      f4=eval[z,{z=3,x=1}]=3
      a4=eval[x,{z=3,x=1}]=1
      apply[f4,a4]
      eval[+ 3 1] → 4
```


Applications with Closures I

```
LET x = 2 IN ( (LET x=1 IN LET f(y)=x+y IN f)  x )

eval[ (λx| ((λx|(λy|+x y)) 1) x)   2 ,{}]
```
Regular apply: eval f1, eval a1, apply f1 to a1
```
  f1=eval[ (λx|((λx | (λy | + x y)) 1) x), {}]
```
Definition:make closure
```
  f1=<(λx |((λx | (λy | + x y)) 1) x),{}>

  a1=eval[2]=2

  apply(f1,a1)
```
f1 is closure:eval f1 body with arg a1 and context of f1
```
  f1=eval[((λx | (λy | + x y)) 1) x ,{x←2}+{}]
```

Applications with Closures II

f1=eval[((λx | (λy | + x y)) 1) x ,{x←2}+{}]

Regular apply: eval f2, eval a2, apply f2 to a2

f2=eval[((λx | (λy | + x y)) 1,{x←2}]

Regular apply: eval f3, eval a3, apply f3 to a3

f3=eval[((λx | (λy | + x y)),{x←2}]

Definition:make closure

f3=<((λx | (λy | + x y)),{x←2}>

a3=eval[1,{x←2}]

apply(f3,a3)

Closure:eval f3 body with arg a3 and context of f3

Applications with Closures III

eval[(λy | + x y),{x←1, x←2}]

Definition:make closure

<(λy | + x y),{x←1, x←2}>

f2 = <(λy | + x y),{x←1, x←2}>

a2 = eval[x,{x←2}] = 2

apply(f2,a2)

Closure: eval f2 body

in env with arg=a2 and f2 context

eval[+ x y,{y←2,x←1, x←2}]

Apply: eval f3, eval a4, a5, and apply f3

a4=eval[x, {y←2,x←1, x←2}] = 1

a5=eval[y, {y←2,x←1, x←2}] = 2

apply(+,1,2)

→3 ...

→3

Other Uses for Closures

- Closures can be used for creating delayed computations
  - Delay and force predicates covered earlier
- Making recursion more efficient

Bindings and Recursion I

- Applicative order reduction blows up with Combinator Y

  $\longrightarrow$ R<YR> $\longrightarrow$ RR<YR> $\longrightarrow$ RRR<YR>
- Normal order is inefficient in general - but suppose we use it
- Bindings evaluate Fixed-Point Combinator correcty

```
F≡(λf | (λn | zerop(n) 0 f(n-1)))
YF ≡(λf | (λx| f (x x))  (λx| f (x x)) )
eval[(λf | (λx| f (x x))  (λx| f (x x)) ) F,{}]
eval[(λx| f (x x))  (λx| f (x x)), {f←F}]
⋮
→(λx| F (x x))  (λx| F (x x)) ≡⟨YF⟩
```

Bindings and Recursion II

```
eval[ (λf | (λn | zerop(n) 0 f(n-1))) ⟨YF⟩ 1, {}]
eval[ (λn | zerop(n) 0 f(n-1)) 1, {f←⟨YF⟩}]
eval[ zerop(n) 0 f(n-1), {n←1,f←⟨YF⟩}]
   eval[ zerop(n), {n←1,f←⟨YF⟩}] → F
   eval[ f(n-1), {n←1,f←⟨YF⟩}]
     eval[ ⟨YF⟩, {n←1,f←⟨YF⟩}] → F ⟨YF⟩
     eval[ n-1, {n←1,f←⟨YF⟩} ] → 0
   eval[ F <YF> 0, {n←1,f←⟨YF⟩} ]
```


Bindings and Recursion III

- Process repeats

```
eval[ (λf | (λn | zerop(n) 0 f(n-1))) ⟨YF⟩ 0,
        {n←1,f←⟨YF⟩} ]

   eval[ (λn | zerop(n) 0 f(n-1)) 0,
        {f←⟨YF⟩,n←1,f←⟨YF⟩}]

   eval[ zerop(n) 0 f(n-1) 0,
        {n←0,f←⟨YF⟩,n←1,f←⟨YF⟩}]
      eval[ zerop(n), {n←0,f←⟨YF⟩,n←1,f←⟨YF⟩}]
      eval[ zerop(0),
            {n←0,f←⟨YF⟩,n←1,f←⟨YF⟩}] →0
   eval[0] →0
```

Closures and Recursion I

- We end up with many copies of the function in the environment
- Closures can be used to eliminate duplicate copies
- Every instance of a recuring evaluation uses the same closure
  - The body is the same
  - The lexical definition is the same
- Imperatively modify closure so that it points to itself
- Eliminates combinators and the need for normal order reduction
- Imperative operation is internal so it does not affect referential transparency

Closures and Recursion II

E≡LETREC f=⟨BODY⟩ IN ⟨EXPR⟩

C ≡ <⟨BODY⟩,{f←C}><⟨EXPR⟩,{ f←{⟨BODY⟩,C} }>

E ≡<⟨EXPR⟩,{f←C}>

Closures and Recursion III

```
LETREC z(n)=zerop(n) 0 z(n-1) IN z(1)

C ≡ <(λn|zerop(n) 0 z(n-1)), {z←C}>

E ≡ <(z 1), {z←C}>

eval[E,{}]
eval[ <(z 1), {z←C}>, {}]
eval[ (z 1), {z←C}]
  f1=eval[ <(λn|zerop(n) 0 z(n-1)), {z←C}>, {z←C}]
  f1=<(λn|zerop(n) 0 z(n-1)), {z←C}>
  a1=eval[1]
  apply[f1,a1]
  eval[ zerop(n) 0 z(n-1),{n←1,z←C} ]
```


Closures and Recursion IV

```
  eval[ zerop(n) 0 z(n-1),{n←1,z←C} ]
    eval[ zerop(n) , {n←1,z←C} ]
    F
```

NOTE: Under applicative, all args evaluated!
Have to treat IF as special form to avoid infinite regress! Well, assume we did:

```
    eval[ z(n-1), {n←1,z←C} ]
```

Clearly, z will get value from context again.
Recursion emerges from self-reference

Lazy Execution

- Lazy substitution made substitution more efficient
- Still required inefficient normal-order reduction for recursion
- Can more laziness further increase efficiency?
- Add lazy evaluation of functions
- How? What properties do we want
  - Delayed execution of a function should not change its value
  - Execution of a closure at any time always returns the same answer
  - It should be *referentially transparent*
  - Evaluation of an expression can only be affected by free variables
  - Need to store these bindings

Closure-based Execution

- Replace all applications in expression with closures
  - Form the closure: <expression, current environment>
  - Replace application in expresion with closure
- Some applications will never be needed
- Evaluate any closures whose values we need

Meta-Interpreter for Pure Lisp

- A λ-calculus interpreter can be expressed in λ-calculus
- A Pure Lisp interpreter can be expressed in Pure Lisp
- Internal use of imperative operations can improve efficiency without compromising referential transparency — code executes the same way regardless of where or when it is executed