Introduction

- $\lambda$-calculus fully expresses computations of any programming language

- Is $\lambda$-calculus sufficiently expressive to express itself?
Implementing $\lambda$-calculus

- What would be required to automate $\lambda$-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 $\lambda$-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

$$(\lambda x \mid (\text{CONS } (\text{CONS } 2 \ x) \ (\text{CONS } 3 \ x)) \ \ (\text{CONS } 1 \ \text{nil}))$$

$\ x \ \text{shared?}$

- 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?

\[
\langle \text{expression} \rangle := \langle \text{identifier} \rangle \mid \langle \text{application} \rangle \mid \langle \text{function} \rangle
\]

\[
\langle \text{identifier} \rangle := a \mid b \mid c \mid \cdots
\]

\[
\langle \text{application} \rangle := \text{"} \langle \text{expression} \rangle \ \text{"} \langle \text{expression} \rangle \text{"}
\]

\[
\langle \text{function} \rangle := \text{"}(\lambda\ \langle \text{identifier} \rangle \ \text{"}\mid \langle \text{expression} \rangle \text{"})"
\]
Primitive λ-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
    
    \[
    \begin{align*}
    [0 \ \Phi] & \quad ;; \text{an identifier} \\
    [1 \ \Phi] & \quad ;; \text{an application of functions} \\
    [2 \ \Phi] & \quad ;; \text{a function definition}
    \end{align*}
    \]

Primitive λ-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. $0 \equiv \text{cons}( \_0, \_0 )$, $1 \equiv \text{cons}( \_0, \_1 )$
    
    \[
    \begin{align*}
    \text{type} & \quad \text{id} \\
    \text{type} & \quad \text{id}
    \end{align*}
    \]

- 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 $(a \ b)$

\[
\equiv $(0 \ 1)
\equiv \text{cons}( \_1, \text{cons}( \text{cons}( \_0, \_0 ), \text{cons}( \_0, \_1 )))
\]

\[
\begin{align*}
\text{type} & \quad \text{id} \\
\text{type} & \quad \text{id}
\end{align*}
\]
Again, use CONS cell for function definition:

\[ (\lambda a \mid (a \ b)) \]

\[ \equiv (\lambda 0 \mid (0 \ 1)) \]

\[ \equiv \text{cons}(2, \text{cons}(\text{cons}(0,0)\text{type},0\text{id}),\text{cons}(1,\text{type}),\text{cons}(\text{cons}(0,0),\text{cons}(0,1))))\text{type} \]

\[ \text{cons}(\text{cons}(0,0)\text{type},0\text{id}),\text{cons}(1,\text{type}),\text{cons}(\text{cons}(0,0),\text{cons}(0,1))))\text{type id} \text{id} \text{id} \]

Creating Representations I

Using abstract programming idioms

\text{new-id}(\text{last-id})

\[ \equiv \text{cons}(0,\text{successor(\text{second(last-id))}}) \]

\text{new-app}(\text{function,argument})

\[ \equiv \text{cons}(1,\text{cons(function,argument)}) \]

\text{new-def}(\text{parameter, body})

\[ \equiv \text{cons}(2,\text{cons(parameter, body)}) \]
Creating Representations II

\((\lambda a \mid (a \ b))\ c\)
\equiv\ LET\ a = 0 \ IN\n    \ LET\ b = \text{new-id}(a) \ IN\n        \ LET\ c = \text{new-id}(b) \ IN\n            \text{new-app(}\n                \text{new-def}(a, \text{new-app}(a, b)),\n                c)\n
Representation of Type Predicates

Predicates using abstract programming idioms

- Recall: all datatypes are of the form: (\text{type, value})

\(\text{is-id}(\langle E \rangle)\)
\text{;; True if } \langle E \rangle \text{ is constant identifier}
\equiv IF \text{car}(\langle E \rangle)=0 \ THEN \ T \ ELSE \ F

\(\text{is-app}(\langle E \rangle)\)
\text{;; True if } \langle E \rangle \text{ is constant identifier}
\equiv IF \text{car}(\langle E \rangle)=1 \ THEN \ T \ ELSE \ F

\(\text{is-func}(\langle E \rangle)\)
\text{;; True if } \langle E \rangle \text{ is constant identifier}
\equiv IF \text{car}(\langle E \rangle)=2 \ THEN \ T \ ELSE \ F
Accessing Representations

Abstract idioms for datatypes of the form \((\text{type}, \text{value})\)

- **Application Accessors for** \((\text{type} \ (\text{function \ argument}))\)

  \[
  \text{get-func}(A) \equiv \text{car}(\text{cdr}(A)) ; \text{ie funct of application}
  \]

  \[
  \text{get-arg}(A) \equiv \text{cdr}(\text{cdr}(A)) ; \text{ie arg of application}
  \]

- **Function Definition Accessors for** \((\text{type} \ (\text{parameter \ body}))\)

  \[
  \text{get-parm}(F) \equiv \text{car}(\text{cdr}(F)) ; \text{ie get } \lambda \text{ parameter}
  \]

  \[
  \text{get-body}(F) \equiv \text{cdr}(\text{cdr}(F))
  \]

\[\lambda\text{-calculus Evaluation Function}\]

- **Implement \(\lambda\)-evaluation as 3 functions:**
  - eval: takes a \(\lambda\)-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
\( \lambda \)-Calculus Eval Function

\[
eval(\langle E \rangle) \equiv
\]

IF is-id(e)
THEN \( \langle E \rangle \equiv f : a \text{ constant} \)
ELSE IF is-app(e)
THEN \( \langle E \rangle \equiv (\langle F \rangle \langle A \rangle) : \text{ application} \)
apply(get-func(e), get-arg(e))
ELSE \( \langle E \rangle \equiv (\lambda x | \langle \text{BODY} \rangle) : \text{ definition} \)
new-func(get-parm(e), eval(get-body(e)))

\[\Rightarrow\text{ Note: body of definitions are evaluated before use}\]

Applicative-Order Apply Function

\[
\text{apply}(\langle F \rangle, \langle A \rangle) \equiv \langle E \rangle ;; \text{ apply function } \langle F \rangle \text{ to argument } \langle A \rangle
\]
LET \( b = \text{eval}(\langle A \rangle) \) IN
IF is-id(\( \langle F \rangle \))
THEN \( \langle F \rangle \langle A \rangle \equiv (f \langle A \rangle) \)
ELSE IF is-app(\( \langle F \rangle \))
THEN \( \langle F \rangle \langle A \rangle \equiv (\langle G \rangle \langle C \rangle) \langle A \rangle \)
IF is-id(get-func(\( \langle F \rangle \)))
THEN new-app(\( \langle F \rangle \), b)
ELSE apply(\( \text{eval}(\langle F \rangle) \), b)
ELSE \( \langle (\lambda x | \langle G \rangle) \langle A \rangle \rangle \)
eval(subs(b, get-parm(\( \langle F \rangle \)), get-body(\( \langle F \rangle \))))
\textbf{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 \(\lambda y\) must be renamed

- In an application like \((\lambda y \mid y) \; x\)
  - formal parameter \(\lambda y\) does not have to be renamed
  - But, renaming \(\lambda y\) does not alter meaning

- Simplification: Do not check for free parameters — always rename formal parameters

\textbf{λ-Calculus Substitution II}

\texttt{subs(s,v,⟨E⟩) ;; substitute s for var v in expression ⟨E⟩}

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

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

(continued on next slide ...)
ELSE ;; Definition (\(f/B\)) - 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

Applying \(\lambda\)-Calculus Evaluation

▶ To evaluate: \((\lambda x \mid x) a\)

LETREC zero = (\(\lambda sz \mid z\))
AND successor = (\(\lambda x (\lambda sz \mid s(xsz))\))
AND add =
  :
AND zerop =
  :
AND eval = \(\langle\text{BODY}\rangle\)
AND apply = \(\langle\text{BODY}\rangle\) AND subs = \(\langle\text{BODY}\rangle\) IN

LET x = 0 IN
LET a = new-id(x) IN
  eval( new-app(new-func(x,x),a) )
Here, we ignore underlying representation

Just examine how Eval, Apply and Subs work together

Square brackets avoid confusion with $\lambda$-C arguments

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

$\lambda$-Calculus Evaluation Example II

eval[((\lambda y | s) x)] ;; Case: application

apply[ get-fun[((\lambda y|s) x)],
       get-arg[((\lambda y|s) x)] ]

≡apply[(\lambda y | s), x] ;; (definition, arg)

eval[
    subs[ eval[x],
          get-id[ (\lambda y | s) ]
          get-body[ (\lambda y| s) ] ]
    eval[ subs[ x, y, s ] ]
    eval[ s]
→ s
\[ \text{eval} \left[ \left( (\lambda \ y | s) \ ((\lambda \ y | s) \ x) \right) \right] ;; \text{Case: application} \]

\[ \text{apply} \left[ (\lambda \ y | s) , ( (\lambda \ y | s) \ x) \right] ;; \text{Case: (Def, Arg)} \]

\[ \text{eval} \left[ \text{subs} \left[ \text{eval} \left[ ( (\lambda \ y | s) \ x) \right] , \ y , \ s \right] \right] \] ; case: application

\[ \text{apply} \left[ (\lambda \ y | s) , \ x \right] ;; \text{(definition, arg)} \]

\[ \text{eval} \left[ \text{subs} \left[ \text{eval} \left[ x \right] , \ y , \ s \right] \right] \]

\[ \text{eval} \left[ x \right] \rightarrow x ;; \text{constant identifier} \]

\[ \text{eval} \left[ \text{subs} \left[ x , \ y , \ s \right] \right] \]

\[ \rightarrow s \]

\[ \text{eval} \left[ \text{subs} \left[ s , y , s \right] \right] \]

\[ \text{subs} \left[ s , y , s \right] \rightarrow s \]

\[ \text{eval} \left[ s \right] \rightarrow s \]

---

\[ \text{\textit{\lambda-Calculus Evaluation as Function}} \]

\[ \text{\textbullet\ A normal order version of apply is required for recursive functions} \]

\[ \text{\textbullet\ Need to add accumulator variables to pass forward next identifier number} \]

\[ \text{\textbullet\ Not conceptually difficult, but messes up code} \]

\[ \text{\textbullet\ And that’s it:} \]

\[ \text{\textbullet\ \lambda-calculus evaluation can be written as a \lambda-calculus expression} \]

\[ \text{\textbullet\ Therefore, \lambda-calculus evaluation is just another function} \]

\[ \text{\textbullet\ \lambda-calculus can be used to implement \lambda-calculus} \]
Bootstrapping

- Functional languages can be written in abstract programming language
- Abstract programming has a simple translation to $\lambda$-calculus
- $\lambda$-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

$\lambda$-Calculus Evaluation in Lisp

- Possible to implement $\lambda$-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 $\lambda$-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

\[
(\lambda x \mid \text{IF } T \text{ THEN } ((\lambda y \mid (\lambda z \mid y \ z) \ x) \ x) \\
\text{ELSE } ((\lambda y \mid y) \ x)) \ z \\
\beta \rightarrow \left[ z/x \right] \text{IF } T \text{ THEN } ((\lambda y \mid (\lambda z \mid y \ z) \ x) \ x) \ \text{ELSE } ((\lambda y \mid y) \ x)
\]

▶ Followed generic \(\beta\)-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

▶ \((\lambda z \mid y \ z)\) rebuilt even though no \(x\)

▶ In \((\lambda x \mid (\lambda y \mid (\lambda z \mid \langle E \rangle)))\), expression \(\langle E \rangle\) 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

\[
\text{eval}[(\lambda x\ |\ \text{IF } T \text{ THEN } (\lambda z\ |\ x) \text{ ELSE } (\lambda z\ |\ z\ x))\ y)]
\[
[y/x] \ (\text{IF } T \text{ THEN } (\lambda z\ |\ x) \text{ ELSE } (\lambda z\ |\ z\ x))
\rightarrow \text{IF } T \text{ THEN } (\lambda z\ |y) \text{ ELSE } (\lambda z\ |z\ y)
\]

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

\[
\text{eval}[(\lambda x\ |\ \text{IF } T \text{ THEN } (\lambda z\ |\ x) \text{ ELSE } (\lambda z\ |\ z\ x))\ y]]
\]

**Binding Parameters to Expressions**

- Parameter value may in turn be an expression

\[
\text{eval}[\ (\lambda x\ |\ (*\ 2\ x))\ (+\ 3\ 2),\ \{}\ ]
\]
\[
\text{eval}[\ (*\ 2\ x),\ \{x\leftarrow(+\ 3\ 2)\}]\ ]
\]
\[
\text{eval}[2,{x\leftarrow(+\ 3\ 2)}] \rightarrow 2
\]
\[
\text{eval}[x,{x\leftarrow(+\ 3\ 2)}]
\]
\[
\text{eval}[\ (+\ 3\ 2)] \rightarrow 5
\]
Bindings and Multiple Arguments

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

\[
\text{eval[ } \lambda x \mid \lambda y \mid ( + x y ) \text{ ] 3 5, \{\} ]}
\]
\[
\text{eval[ } \lambda y \mid ( + x y ) \text{ ] 5, \{ x ←3 \} ]}
\]
\[
\text{eval[ } ( + x y ) \text{ ] 3 5, \{ y ←5, x ←3 \} ]}
\]
\[
\text{eval[ } x \text{ ] 5, \{ y ←5, x ←3 \} ]}
\]
\[
\text{eval[ } 3 \text{ ] → 3}
\]
\[
\text{eval[ } 5 \text{ ] → 5}
\]
\[
\text{eval[ } ( + 3 5 ) \text{ ] →5}
\]

Bindings and Shadowed Arguments

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

\[
\text{eval[ } \lambda x \mid + \left( \lambda x \mid ( + x x ) \text{ ] 5 \right) x \text{ ] 3, \{\} ]}
\]
\[
\text{eval[ } ( + ( \lambda x \mid ( + x x ) \text{ ] 5 \right) x \text{ ] 3, \{ x ←3 \} ]}
\]
\[
\text{eval[ } ( \lambda x \mid ( + x x ) \text{ ] 5, \{ x ←3 \} ]}
\]
\[
\text{eval[ } ( + x x ) \text{ ] 5, \{ x ←3 \} ] \rightarrow 10
\]
\[
\text{eval[ } x \text{ ] 3, \{ x ←3 \} ] \rightarrow 10
\]
\[
\text{eval[ } 10 \text{ ] 3, \{ x ←3 \} ] \rightarrow 10
\]
\[
\text{eval[ } x \text{ ] 3, \{ x ←3 \} ] \rightarrow 13
\]
Problems with Bindings and Free Variables I

\[ \text{eval}[(\lambda y | (\lambda x | + x y)) 4, \{\}] \]
\[ \text{eval}[(\lambda x | + x y), \{y\leftarrow 4\}] \]

- No application here — cannot evaluate \((\lambda x | + x y)\) further
- But, should have \(y\) bound to 4
- Our simple interpreter actually handles this (but poorly):
  - evaluate \(\lambda\)-body: \(+ x y\) in environment \((y\leftarrow 4)\)
  - create new function with evaluated body \((\lambda x | \langle \text{BODY} \rangle)\)
    \[ \text{eval}[(+ x y, \{y\leftarrow 4\}] \rightarrow (+ x 4) \]
    \[ \rightarrow (\lambda x | + x 4) \]
- Above solution breaks: See next slide!

Problems with Bindings and Free Variables II

\[ \text{eval}[(\lambda y | (\lambda y | (y y))) 4, \{\}] \]
\[ \text{eval}[(\lambda y | (y y)), \{y\leftarrow 4\}] \]

DO NOT DO THIS!
\[ \rightarrow (\lambda y | \text{eval}[( y y), \{y\leftarrow 4\}] ) \]
\[ \equiv (\lambda y | 4 4) \]

- Dynamic binding results in wrong answer! The “funaarg” problem
- Could try to represent fact that \(y\) is bound in inner \(\lambda\)

\[ \text{eval}[(\lambda y | (y y)), \{y\leftarrow 4\}] \]
DO NOT DO THIS!
\[ \rightarrow (\lambda y | \text{eval}[( y y), \{y\leftarrow y, y\leftarrow 4\}] ) \]
\[ \rightarrow (\lambda y | ( y y)) \]

- Solution Breaks in more complex cases
Imagine we define a function in a local context and return it:

\[
(\text{LET } x=1 \text{ IN LET } f(y)=x+y \text{ IN } f) \rightarrow f
\]

Now apply this function to \(x\), where \(x=2\)

\[
\text{LET } x = 2 \text{ IN (LET } x=1 \text{ IN LET } f(y)=x+y \text{ IN } f) \ x)
\]

Translates into \(\lambda\)-calculus as:

\[
(\lambda x \ | \ ((\lambda x | (\lambda y | + x y)) \ 1) \ x) \ 2
\]

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[ (\lambda x | ((\lambda x| (\lambda y| + x y)) 1) x)  2 ,{}]
```

```
Regular apply, make binding
```

```
eval[ ((\lambda x | (\lambda y | + x y)) 1) x, \{x←2\}]
```

```
Apply (f a): eval f1, then eval a1, then apply f1 to a1
```

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

```
Apply (f a): eval f2, then eval a2, then apply f2 to a2
```

f2=eval[(\lambda x | (\lambda y | + x y)), \{x←2\}]

```
Def: eval body (\lambda y | + x y) for local bindings
```

```
eval[ (\lambda y | + x y), \{x←2\}]
```

```
Def: eval body (+x y) for local bindings
```

```
eval[ + x y, \{x←2\}] \rightarrow + 2 y  \text{ OOPS!!}
```
Problems with Bindings and Free Variables IV

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

\[
\begin{align*}
\text{new-}\text{func}(y,+2\ y) & \rightarrow (\lambda y|+2\ y) \\
\text{f2=} & \text{new-}\text{func}(x,(\lambda y|+2\ y)) \rightarrow (\lambda x|(\lambda y|+2\ y)) \\
a2= & \text{eval}[1,\{x\leftarrow2\}] = 1 \\
\text{apply}[(\lambda x|(\lambda y|+2\ y)),1] & = \\
\text{f1=} & (\lambda y|+2\ y) \\
a1= & \text{eval}[x,\{x\leftarrow2\}] = 2 \\
\text{apply}[ (\lambda y|+2\ y), 2 ] & \rightarrow 4 \quad \text{WRONG!}
\end{align*}
\]

Lexical Context

- Can you see why early Lisp’s has Dynamic scoping?
  - Implementation used simple bindings lists.
  - Dynamic scoping does not match $\lambda$-calculus semantics
  - Difficult to debug and understand
Lexical Context

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

\[
\text{LET } x = 2 \text{ IN } \\
( \text{LET } x = 1 \text{ IN } \\
\quad \text{LET } f(y) = x + y \text{ IN } f \quad x )
\]
- 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.
- \(<\text{closure}> = \{\text{expression, environment}\}\)
- 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

\[ \text{eval}\left[(\lambda x \mid x) \ 2\right] ,\{}\right] \]

Regular apply: \( \text{eval} \ f_1, \ \text{eval} \ a_1, \ \text{apply} \ f_1 \ \text{to} \ a_1 \)

\[ f_1 = \text{eval}\left[(\lambda x \mid x) \right] ,\{}\right] \]

Definition: make closure
\[ f_1 = <(\lambda x \mid x) ,\{}\right] > \]

\[ a_1 = \text{eval}\left[2\right] = 2 \]

\[ \text{apply}\left[ f_1, \ a_1 \right] \]

Eval \( f_1 \) body in environment
with \( x=a_1 \) and context of \( f_1={} \)
\[ \text{eval}\left[ x, \{x\leftarrow 2\}+\{} \right] \]

\[ \rightarrow 2 \]

▶ Seems like extra machinery, but useful in complex cases

Forming and Applying Closures

▶ Forming closures
▶ Given definition \( (\lambda p \mid \langle \text{BODY} \rangle) \) defined in environment \( E \)
▶ We form the closure \( <(\lambda p \mid \langle \text{BODY} \rangle) ,E> \)

▶ To apply closure \( <(\lambda p \mid \langle \text{BODY} \rangle) ,E> \) to argument \( A \) in context \( G \)
▶ evaluate \( \langle \text{BODY} \rangle \)
▶ in an environment = \{ \( p\leftarrow A + E + G\} \)
Trickier Application with Closures I

LET \( x=1 \) IN LET \( y=(\lambda z|z+x) \) IN \( y(3) \)

\[\text{eval}[(\lambda x|(\lambda y|(y \ 3)) (\lambda z|z+x)) \ 1, \ {}]\]

**Regular apply, eval f1, eval a1, apply f1 to a1**

\( f1=\text{eval}[(\lambda x|(\lambda y|(y \ 3)) (\lambda z|z+x)), \ {}]\)

*Definition: make closure*

\( f1=<(\lambda x|(\lambda y|(y \ 3)) (\lambda z|z+x)>, \{}\>

\( a1=\text{eval}[\ 1, \ {}] = 1\)

apply(\( f1,a1 \))

**Eval f1 body with a1 and context of f1**

\[\text{eval}[(\lambda y|(y \ 3)) (\lambda z|z+x), \{x=1\}]\]

---

Trickier Application with Closures II

\[\text{eval}[(\lambda y|(y \ 3)) (\lambda z|z+x), \{x=1\}]\]

**Regular apply, eval f2, eval a2, apply f2 to a2**

\( f2=\text{eval}[(\lambda y|(y \ 3), \{x=1\}]\)

*Definition: make closure*

\( f2=<(\lambda y|(y \ 3), \{x=1}\>

\( a2=\text{eval}[(\lambda z|z+x), \{x=1\}]\)

*Definition: make closure*

\( a2=<(\lambda z|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

\[\text{eval}[(y \ 3), \{y=<(\lambda z|z+x), \{x=1\}> ,x=1\}]\]
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 | (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

\( f_1 = \text{eval} \left[ \left( \lambda x \mid (\lambda y \mid + x \; y) \right) 1 \right] x, \{x \leftarrow 2\} + \{\} \)

Regular apply: \( \text{eval} \; f_2, \; \text{eval} \; a_2, \; \text{apply} \; f_2 \; \text{to} \; a_2 
\)

\( f_2 = \text{eval} \left[ \left( \lambda x \mid (\lambda y \mid + x \; y) \right) 1 \right] \{x \leftarrow 2\} \)

Regular apply: \( \text{eval} \; f_3, \; \text{eval} \; a_3, \; \text{apply} \; f_3 \; \text{to} \; a_3 
\)

\( f_3 = \text{eval} \left[ \left( \lambda x \mid (\lambda y \mid + x \; y) \right) \right] \{x \leftarrow 2\} \)

Definition: make closure

\( f_3 = < \left( \lambda x \mid (\lambda y \mid + x \; y) \right), \{x \leftarrow 2\} > \)

\( a_3 = \text{eval} \left[ 1, \{x \leftarrow 2\} \right] \)

apply\( (f_3, a_3) \)

Closure: eval \( f_3 \) body with arg \( a_3 \) and context of \( f_3 \)

Applications with Closures III

\( \text{eval} \left[ (\lambda y \mid + x \; y) \right], \{x \leftarrow 1, \; x \leftarrow 2\} \)

Definition: make closure

\( < (\lambda y \mid + x \; y), \{x \leftarrow 1, \; x \leftarrow 2\} > \)

\( f_2 = < (\lambda y \mid + x \; y), \{x \leftarrow 1, \; x \leftarrow 2\} > \)

\( a_2 = \text{eval} \left[ x, \{x \leftarrow 2\} \right] = 2 \)

apply\( (f_2, a_2) \)

Closure: eval \( f_2 \) body

in env with arg = \( a_2 \) and \( f_2 \) context

\( \text{eval} \left[ + \; x \; y, \{y \leftarrow 2, x \leftarrow 1, \; x \leftarrow 2\} \right] \)

Apply: eval \( f_3 \), eval \( a_4 \), \( a_5 \), and apply \( f_3 \)

\( a_4 = \text{eval} \left[ x, \{y \leftarrow 2, x \leftarrow 1, \; x \leftarrow 2\} \right] = 1 \)

\( a_5 = \text{eval} \left[ y, \{y \leftarrow 2, x \leftarrow 1, \; x \leftarrow 2\} \right] = 2 \)

apply\( (+, 1, 2) \)

\( \rightarrow 3 \ldots \)
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
  \[ R<YR> \rightarrow RR<YR> \rightarrow RRR<YR> \]
- Normal order is inefficient in general - but suppose we use it
- Bindings evaluate Fixed-Point Combinator correctly
  \[ F \equiv (\lambda f \mid (\lambda n \mid \text{zerop}(n) \ 0 \ f(n-1))) \]
  \[ YF \equiv (\lambda f \mid (\lambda x \mid f (x \ x)) \ (\lambda x \mid f (x \ x)) \ ) \ F,\{} \]
  \[ \text{eval}[(\lambda x \mid f (x \ x)) \ (\lambda x \mid f (x \ x)), \{f \leftarrow F\}] \]
  \[ \ldots \]
  \[ \rightarrow (\lambda x \mid F (x \ x)) \ (\lambda x \mid F (x \ x)) \equiv (YF) \]
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 recurring 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

\[
E \equiv \text{LETREC } f = \langle \text{BODY} \rangle \text{ IN } \langle \text{EXPR} \rangle
\]

\[
C \equiv <\langle \text{BODY} \rangle, \{f \leftarrow C\}> <\langle \text{EXPR} \rangle, \{f \leftarrow \{\langle \text{BODY} \rangle, C\} \}>
\]

\[
E \equiv <\langle \text{EXPR} \rangle, \{f \leftarrow C\}>
\]
Closures and Recursion III

\[ \text{LETREC } z(n) = \text{zerop}(n) \ 0 \ z(n-1) \ \text{IN } z(1) \]

\[ C \equiv \langle (\lambda n | \text{zerop}(n) \ 0 \ z(n-1)), \{z \leftarrow C\} \rangle \]

\[ E \equiv \langle (z \ 1), \{z \leftarrow C\} \rangle \]

eval[E,{}

eval[ \langle (z \ 1), \{z \leftarrow C\} \rangle, \{}

eval[ \langle (z \ 1), \{z \leftarrow C\} \rangle

f1=eval[ \langle (\lambda n | \text{zerop}(n) \ 0 \ z(n-1)), \{z \leftarrow C\} \rangle, \{z \leftarrow C\} \]

f1=\langle (\lambda n | \text{zerop}(n) \ 0 \ z(n-1)), \{z \leftarrow C\} \rangle

a1=eval[1]

apply[f1,a1]

eval[ \text{zerop}(n) \ 0 \ z(n-1), \{n \leftarrow 1, z \leftarrow C\} ]

\]

Closures and Recursion IV

\[ \text{eval}[ \text{zerop}(n) \ 0 \ z(n-1), \{n \leftarrow 1, z \leftarrow C\} ] \]

\[ \text{eval}[ \text{zerop}(n), \{n \leftarrow 1, z \leftarrow C\} ] \]

F

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

\[ \text{eval}[ z(n-1), \{n \leftarrow 1, z \leftarrow C\} ] \]

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: \(<\text{expression, current environment}>\)
  - Replace application in expression 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