Lambda Calculus: Implementation Techniques and a Proof

COS 441 Slides 15
A language of pure functions:

\[ e ::= x | \lambda x. e | e \, e \]  
\[ v ::= \lambda x. e \]

With a call-by-value operational semantics:

**single step:**

\[ (\lambda x. e) \, v \rightarrow e \, [v/x] \] (beta)

\[ e_1 \rightarrow e'_1 \] (app1)

\[ e_1 \, e_2 \rightarrow e'_1 \, e_2 \]

\[ e_2 \rightarrow e'_2 \] (app2)

\[ v \, e_2 \rightarrow v \, e'_2 \]

**multi-step:**

\[ e \rightarrow^* e \] (reflexivity)

\[ e_1 \rightarrow e_2 \quad e_2 \rightarrow^* e_3 \] (transitivity)
Examples

• We used the formal rules to build proofs that lambda terms could take steps:

\[ \frac{(x.\ y.\ x\ y)\ (\w.\w)}{\rightarrow\ y.(\w.\w)\ y} \]  \hspace{2cm} \text{(beta)}

\[ \frac{(x.\ y.\ x\ y)\ (\w.\w)\ (z.\ z)}{\rightarrow\ (y.(\w.\w)\ y)\ (z.\ z)} \] \hspace{2cm} \text{(app1)}

• We showed it was possible to encode several simple kinds of data structures or computations
  – booleans
  – pairs
  – numbers
  – looping
  – and I claimed you could code up anything else since the untyped lambda calculus is Turing-complete
IMPLEMENTING THE LAMBDA CALCULUS
Two Options

• **First-order Abstract Syntax:** build a data structure to represent a program

```haskell
data Lam =
    Var String
  | Abs String Lam
  | App Lam Lam
```

• **Higher-order Abstract Syntax:** use functions in Haskell to represent lambda calculus functions directly

```haskell
data Lam =
    Abs (Lam -> Lam)
  | App Lam Lam
  | FreeVar String
```
FIRST-ORDER SYNTAX
Examples

• Data structure:

```haskell
data Lam =
    Var String
    | Abs String Lam
    | App Lam Lam
```

• Examples:

```haskell
i = Abs "x" (Var "x")  -- \x.x

tru = Abs "t" (Abs "f" (Var "t"))  -- \t.\f.t

fls = Abs "t" (Abs "f" (Var "f"))  -- \t.\f.f
```

• i, tru, fls all have type Lam
Substitution

- Substitution:

  -- subst e x v == e[v/x]  -- v must be closed (no free variables)
Substitution

- Substitution:
  
  -- subst e x v == e[v/x]       -- v must be closed (no free variables)

  subst (Var y) x v = if x == y then v else Var y
Substitution

- Substitution:

  -- subst e x v == e[v/x]   -- v must be closed (no free variables)
  subst (Var y) x v = if x == y then v else Var y
  subst (Abs y e) x v = if x == y then (Abs y e) else (Abs y (subst e x v))
Substitution

- Substitution:

  -- subst e x v == e[v/x]    -- v must be closed (no free variables)

  subst (Var y) x v = if x == y then v else Var y
  subst (Abs y e) x v = if x == y then (Abs y e) else (Abs y (subst e x v))
  subst (App e1 e2) x v = App (subst e1 x v) (subst e2 x v)
Substitution

• Substitution:

-- subst e x v == e[v/x]     -- v must be closed (no free variables)

subst (Var y) x v = if x == y then v else Var y
subst (Abs y e) x v = if x == y then (Abs y e) else (Abs y (subst e x v))
subst (App e1 e2) x v = App (subst e1 x v) (subst e2 x v)

• Example:

y    = Var "y"
x    = Var "x"

id   = Abs "x" x
foo  = App (Abs "y" y) y

subst foo "y" id  ==  App (Abs "y" y) id
Code to determine if an expression is a value:

```
-- is a value? --

value (Var s)    = False
value (Abs x e)  = True
value (App e1 e2) = False
```
Evaluation

\[ \text{eval :: Lam} \rightarrow \text{Lam} \]

-- beta rule
\[ \text{eval} (\text{App} (\text{Abs} \ x \ e) \ v) \mid \text{value} \ v = \text{subst} \ e \ x \ v \]
Evaluation

eval :: Lam -> Lam

-- beta rule
eval (App (Abs x e) v) | value v = subst e x v

-- app2 rule
eval (App v e2) | value v = let e2' = eval e2 in
  App v e2'
Evaluation

eval :: Lam -> Lam

-- beta rule
eval (App (Abs x e) v) | value v = subst e x v

-- app2 rule
eval (App v e2) | value v = let e2' = eval e2 in App v e2'

-- app1 rule
eval (App e1 e2) = let e1' = eval e1 in App e1' e2

\( (\lambda x.e) v \rightarrow e[v/x] \) (beta)

\[ \frac{e1 \rightarrow e1'}{e1 \; e2 \rightarrow e1' \; e2} \] (app1)

\[ \frac{e2 \rightarrow e2'}{v \; e2 \rightarrow v \; e2'} \] (app2)
eval :: Lam -> Lam

-- beta rule
eval (App (Abs x e) v) | value v = subst e x v

-- app2 rule
eval (App v e2) | value v = let e2' = eval e2 in App v e2'

-- app1 rule
eval (App e1 e2) = let e1' = eval e1 in App e1' e2

-- forms that don't match LHS; no rule exists
eval (Abs x e) = error "Value!"
eval (Var x) = error "Stuck!"
HIGHER-ORDER SYNTAX
Higher-Order Abstract Syntax

• Key idea: use functions in Haskell to represent lambda calculus functions directly

```
data Lam =
    Abs (Lam -> Lam)
  | App Lam Lam
  | FreeVar String
```

needed for printing, and analysis of expressions

not needed for evaluation

equations should not have free variables if you want to execute them
Higher-Order Abstract Syntax

- Key idea: use functions in Haskell to represent lambda calculus functions directly

```haskell
data Lam = Abs (Lam -> Lam) | App Lam Lam
```

remember, Abs is a converter:

- it takes a Lam -> Lam function and puts it in an "object" with 1 field that has type Lam

  Abs :: (Lam -> Lam) -> Lam

  App :: Lam -> Lam -> Lam
Higher-Order Abstract Syntax

• Key idea: use functions in Haskell to represent lambda calculus functions directly

```haskell
data Lam = Abs (Lam -> Lam)  
  | App Lam Lam
```

```haskell
f :: Lam -> Lam
f = \x -> App x x
```

```haskell
e :: Lam
e = Abs f
```

Abs makes a Haskell function f into a Lam

a Haskell variable with type Lam appears in the places a lambda calculus variable would appear

think of the body of f like an expression with holes it:

```
App [ ] [ ]
```

during evaluation, we'll plug the holes with the argument to the function
Higher-Order Abstract Syntax

- Key idea: use functions in Haskell to represent lambda calculus functions directly

```haskell
data Lam = Abs (Lam -> Lam) | App Lam Lam

f :: Lam -> Lam
f = \x -> App x x

e :: Lam
e = Abs f
```
Higher-Order Abstract Syntax

- Key idea: use functions in Haskell to represent lambda calculus functions directly

```haskell
f :: Lam -> Lam
f = \x -> App x x

e :: Lam
e = Abs f

id :: Lam -> Lam
id = \x -> x

ide :: Lam
ide = Abs id
```

```haskell
data Lam = Abs (Lam -> Lam) | App Lam Lam
```
Higher-Order Abstract Syntax

- Key idea: use functions in Haskell to represent lambda calculus functions directly

```
data Lam = Abs (Lam -> Lam) | App Lam Lam
```

```
f :: Lam -> Lam
f = \x -> App x x
```

```
e :: Lam
e = Abs f
```

```
id :: Lam -> Lam
id = \x -> x
```

```
ide :: Lam
ide = Abs id
```

```
e' :: Lam
e' = App e ide
```

Evaluating (App e ide):
- Inside e, we have $f = \lambda x . \text{App } x \ x$
- Applying $f$ to $\text{ide}$, we get: $\text{App } \text{ide } \text{ide}$
• Key idea: use functions in Haskell to represent lambda calculus functions directly

id :: Lam -> Lam
id = \x -> x

ide :: Lam
ide = Abs ide

fls = Abs (\t -> Abs (\f -> f))

tru = Abs (\t -> Abs (\f -> t))

An Alternative:

id = \f -> f
ide = Abs id
flsf = \t -> Abs id
fls = Abs flsf

An Alternative:

truf = \t -> Abs (\f -> t)
tru = Abs truf
Evaluation

eval :: Lam -> Lam

eval (App (Abs f) v) | value v = f v -- beta rule

eval (App v e2) | value v = -- app2 rule
let e2' = eval e2 in
  App v e2'

eval (App e1 e2) = -- app1 rule
let e1' = eval e1 in
  App e1' e2

eval (Abs f) = error "Value!"

-- note: we never had to implement
-- substitution ourselves; Haskell did it for us
Can Evaluation Ever Get Stuck?

• Values are lambda expressions that have “properly finished” being evaluated – there is nothing more to do.
  – In the pure lambda calculus, the only values are functions
  – “\x.x” is a value. It can’t be evaluated any further.
  – “\x.\y.x y” is also a value

• Are there lambda terms that aren’t values but can’t be evaluated any further using the rules?

• If there were, we’d call those things stuck expressions
Can Evaluation Ever Get Stuck?

- Values are lambda expressions that have "properly finished" being evaluated – there is nothing more to do.
  - In the pure lambda calculus, the only values are functions.
  - "\x.x" is a value. It can't be evaluated any further.
  - "\x.\y.x y" is also a value.
- Are there lambda terms that aren't values but can't be evaluated any further using the rules?
- If there were, we'd call those things stuck expressions.
- Expressions with free variables can be stuck! Eg:
  - x
  - x (\y.y)
  - (\y. x y) (\w.w) isn't stuck right away, but will be after an evaluation step.
Given a lambda term, is it possible to create an automatic analyzer that decides, yes or no, whether or not a lambda term will ever get stuck?
Stuckness testing

• Given a lambda term, is it possible to create an automatic analyzer that decides, yes or no, whether or not a lambda term will ever get stuck?
  – No! The lambda calculus is Turing-Complete. It can encode any Turing Machine.
  – Suppose TM is a lambda term that simulates a Turing Machine
  – Consider: $(\lambda x.y \ x) \ TM$
  – The above expression gets stuck by running into free variable $y$ if the TM halts; does not get stuck if the TM does not halt. We can’t decide if TMs halt, so we can’t decide if the lambda term ever gets stuck.
Stuckness testing

• Given a lambda term, is it possible to create an automatic analyzer that soundly but conservatively decides whether or not a lambda term will ever get stuck?
  – i.e.: can we design an algorithm that given a lambda term,
    • says “no the lambda term is not stuck” if it can guarantee the lambda term is not stuck?
    • says “yes, maybe” if it isn’t sure?
  – of course! the algorithm could always cop out and say “yes, maybe”

• But it turns out we can also define a principled, non-trivial analyzer that is sound and conservative, but for all practical purposes does a “good enough” job
  – such an analyzer is called a scope checker
  – and it is the simplest kind of type system
A SIMPLE SCOPE CHECKER
A Scope Checker for FOAS Expressions

data Lam =
   Var String     -- variables
   | Abs String Lam -- \"x\". e
   | App Lam Lam    -- e1 e2

closed :: Lam -> Bool
closed e = clos [] e
where
clos env (Abs x e) = clos (x:env) e
clos env (App e1 e2) = clos env e1 && clos env e2
clos env (Var x) = lookup env x

lookup [] x = False
lookup (y:env) x = x == y || lookup env x
Scope Checking Examples

• A closed lambda expression:
  – \( \lambda y. \lambda x. y \) is closed:
  – closed (Abs "y" (Abs "x" (Var "y"))) == True

  – \( y (\lambda y.y) \) is not closed:
  – closed (App (Var "y") (Abs "y" (Var y))) == False

• Can you come up with a lambda term that is not closed according to our Haskell definition but that evaluates safely without encountering a free variable?
  – there must be one because I told you that it is undecidable whether execution encounters a free variable
Scope Checking Examples

• A closed lambda expression:
  – \( \lambda y. \lambda x.y \) is closed:
  – closed (Abs "y" (Abs "x" (Var "y"))) == True

  – \( y (\lambda y. y) \) is not closed:
  – closed (App (Var "y") (Abs "y" (Var y))) == False

• Can you come up with a lambda term that is not closed according to our Haskell definition but that evaluates safely without encountering a free variable?
  – there must be one because I told you that it is undecidable whether execution encounters a free variable
  – \( (\lambda x. \lambda y.y) (\lambda z. \lambda w.w) \rightarrow (\lambda y. \lambda w.w) \rightarrow \lambda w.w \)
module Lambda (Lam (Abs, App),  -- only Abs App constructors useable by clients
        freevar,            -- freevar function useable
        ... ) where

data Lam =  Abs (Lam -> Lam) | App Lam Lam | FreeVar String

freevar :: String -> Lam
freevar s = FreeVar ("!" ++ s)
module Lambda (Lam (Abs,App),  -- only Abs App constructors useable by clients
    freevar,                     -- freevar function useable ...
    ... ) where

data Lam =  Abs (Lam -> Lam) | App Lam Lam | FreeVar String

freevar :: String -> Lam
freevar s = FreeVar ("!") ++ s

boundname = "bound"

closed :: Lam -> Bool
closed (Abs f) = ...
module Lambda (Lam (Abs, App), -- only Abs App constructors useable by clients
    freevar, -- freevar function useable
    ... ) where

    data Lam = Abs (Lam -> Lam) | App Lam Lam | FreeVar String

    freevar :: String -> Lam
    freevar s = FreeVar ("!" ++ s)

    boundname = "bound"

    closed :: Lam -> Bool
    closed (Abs f) =
        let body = f (FreeVar boundname) in
        closed body
module Lambda (Lam (Abs, App), -- only Abs App constructors useable by clients
       freevar, -- freevar function useable
       ... ) where

data Lam = Abs (Lam -> Lam) | App Lam Lam | FreeVar String

freevar :: String -> Lam
freevar s = FreeVar (!"!") ++ s

boundname = "bound"

closed :: Lam -> Bool
closed (Abs f) =
    let body = f (FreeVar boundname) in
closed body
closed (App e1 e2) = closed e1 && closed e2
closed (FreeVar s) = s == boundname
ONE MORE WAY TO DESCRIBE CLOSED EXPRESSIONS
Closed Expressions

env ::= x1 : x2 : .... : []

judgement form: clos env e -- "e has no free variables except those in env"
Closed Expressions

env ::= x1 : x2 : .... : []

judgement form:  \( \text{clos env e} \)  -- "e has no free variables except those in env"

\( \text{clos (x:env) e} \)  -- if e is closed in (x:env) then \\
\( \text{clos env \( \backslash x.e \) } \)  \( \backslash x.e \) is closed in env
Closed Expressions

env ::= x1 : x2 : .... : []

judgement form: clos env e  -- "e has no free variables except those in env"

\[
\begin{align*}
\text{clos (x:env) e} & \quad \text{-- if e is closed in (x:env) then} \\
\text{clos env \ x.e} & \quad \text{\ x.e is closed in env} \\
\text{clos env e1  clos env e2} & \quad \text{-- if e1 and e2 are closed in env then} \\
\text{clos env (e1 e2)} & \quad \text{e1 e2 is closed in env}
\end{align*}
\]
Closed Expressions

\[\text{env ::= } x_1 : x_2 : \ldots : []\]

judgement form: \(\text{clos env } e\) -- "e has no free variables except those in env"

\[\begin{align*}
\text{clos } x:\text{env } e & \quad \text{-- if } e \text{ is closed in } (x:\text{env}) \text{ then} \\
\text{clos env } \backslash x.e & \quad \text{\backslash } x.e \text{ is closed in env} \\
\text{clos env } e_1 & \quad \text{clos env } e_2 \quad \text{-- if } e_1 \text{ and } e_2 \text{ are closed in env then} \\
\text{clos env } (e_1 \ e_2) & \quad e_1 \ e_2 \text{ is closed in env} \\
\text{lookup env } x == \text{true} \quad \text{-- if } x \text{ is in env then} \\
\text{clos env } x & \quad x \text{ is closed in env}
\end{align*}\]
A PROOF
• Theorem: If \text{clos} \ e \text{ and } e \rightarrow e' \text{ then } \text{clos} \ e'.
Evaluation Preserves Closedness

• Theorem: If $\text{clos} [] \ e$ and $e \rightarrow e'$ then $\text{clos} [] \ e'$.
• Requires a lemma that substitution preserved Closedness
  – Lemma: If $\text{clos} [] (\lambda x. e)$ and $\text{clos} [] \ v$ then $\text{clos} [] (e[v/x])$
Evaluation Preserves Closedness

• Theorem: If clos [] e and e --> e' then clos [] e'.
• Proof: By induction on the derivation that e --> e'
  – proofs by induction on the derivation of e --> e' have 1 case for each rule
  – use the induction hypothesis on when subprog --> subprog in the premise of the rule.
  – use lemma: If clos [] (\x.e) and clos [] v then clos [] (e[v/x])
Evaluation Preserves Closedness

- Theorem: If \( \text{clos} [\ ] e \) and \( e \rightarrow e' \) then \( \text{clos} [\ ] e' \).
- Proof: By induction on the derivation that \( e \rightarrow e' \)
  - Lemma: If \( \text{clos} [\ ] (\lambda x. e) \) and \( \text{clos} [\ ] v \) then \( \text{clos} [\ ] (e[v/x]) \)
- case:

\[
\begin{align*}
(\lambda e. v) & \rightarrow e[v/x] \quad \text{(beta)} \\
(1) & \text{clos} [\ ] ((\lambda x. e) v) \quad \text{(given)} \\
(2) & \text{clos} [\ ] (\lambda x. e) \quad \text{(by 1, def of clos)} \\
(3) & \text{clos} [\ ] v \quad \text{(by 1, def of clos)} \\
(4) & \text{clos} [\ ] (e[v/x]) \quad \text{(by 2, 3)}
\end{align*}
\]

\[
\begin{align*}
\text{clos } (x: \text{env}) e & \quad \text{clos env } e_1 \quad \text{clos env } e_2 \\
\text{clos env } \lambda x. e & \quad \text{clos env } (e_1 \ e_2) \\
\text{lookup env } x \ &= \ true \quad \text{clos env } x
\end{align*}
\]
Evaluation Preserves Closedness

- **Theorem:** If clos [] e and e --> e' then clos [] e'.
- **Proof:** By induction on the derivation that e --> e'
  - **Lemma:** If clos [] (\x.e) and clos [] v then clos [] (e[v/x])
- **case:**

  \[
  \frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2} \quad \text{(app1)}
  \]

  (1) clos [] (e1 e2) \quad \text{(given)}
  (2) clos [] e1 \quad \text{(by 1, def of clos)}
  (3) clos [] e2 \quad \text{(by 1, def of clos)}
  (4) clos [] e1' \quad \text{(by IH, 2)}
  (5) clos [] (e1' e2) \quad \text{(by 4, 3, def of clos)}
Evaluation Preserves Closedness

- **Theorem:** If \( \text{clos} \ [\ e \] \) and \( e \rightarrow e' \) then \( \text{clos} \ [\ e' \] \).
- **Proof:** By induction on the derivation that \( e \rightarrow e' \)
  - **Lemma:** If \( \text{clos} \ [\ \lambda x. e \] \) and \( \text{clos} \ [\ v \) then \( \text{clos} \ [\ (e[v/x]) \]
- **case:**

\[
\frac{e_2 \rightarrow e_2'}{\underbrace{v \ e_2 \rightarrow v \ e_2'}} \quad \text{(app2)}
\]

\[
\begin{align*}
(1) & \quad \text{clos} \ [\ (v \ e_2) \] & \quad \text{(given)} \\
(2) & \quad \text{clos} \ [\ v \] & \quad \text{(by 1, def of clos)} \\
(3) & \quad \text{clos} \ [\ e_2 \] & \quad \text{(by 1, def of clos)} \\
(4) & \quad \text{clos} \ [\ e_2' \] & \quad \text{(by IH, 3)} \\
(5) & \quad \text{clos} \ [\ (v \ e_2') \] & \quad \text{(by 2, 4, def of clos)}
\end{align*}
\]

\[
\begin{array}{ccc}
\text{clos} \ (x:\text{env}) \ e & \text{clos env} \ e_1 & \text{clos env} \ e_2 \\
\text{clos env} \ \lambda x. e & \text{clos env} \ (e_1 \ e_2) & \text{lookup env} \ x = \text{true} \\
\text{clos env} \ x
\end{array}
\]
Why do we care?

• Why do we care if closure is preserved by execution?
Why do we care?

• Why do we care if closure is preserved by execution?
• The initial motivation was that programs could get "stuck" when executing by running in to a free variable. We wanted to prevent that.
• In a real language implementations, getting "stuck" often means all hell breaks loose and random bad stuff ensues:
  – derefencing a dangling pointer in C is another way to "get stuck"
• If we checked a program was closed, but then after 3 steps of evaluation a free variable appeared, then closure checking wouldn't be helpful -- it wouldn't prevent programs from getting stuck
• Moral: closure checking is a useful kind of static program analysis because if you check a program once before it executes, you never, ever have to worry about it getting stuck on a free variable, no matter how long it runs
SUMMARY
Summary

• There are at least two ways to implement the lambda calculus
  – higher-order abstract syntax uses Haskell functions to implement lambdas and Haskell variables to implement lambda variables
  – first-order abstract syntax uses strings to represent variables and does not use functions

• Unfortunate Fact: Almost every non-trivial property of how a lambda expression evaluates is undecidable

• Optimistic Perspective: We can approximate many properties

• Example:
  – do we encounter a free var during execution: undecideable
  – we can still design a useful scope checker
  – the closure property is robust and highly useful because it is preserved by execution