Lambda Calculus: Implementation Techniques and a Proof

COS 441 Slides 15

Last Time: The Lambda Calculus

A language of pure functions:

values

e ::= x | \x.e | e e   v ::= \x.e

With a call-by-value operational semantics:

single step:

$$\frac{}{\text{(\textbackslash x.e) v --> e [v/x]}}\ \text{(beta)}$$

$$\frac{\text{e1 --> e1'}}{\text{e1 e2 --> e1' e2}}\ \text{(app1)}$$

$$\frac{\text{e2 --> e2'}}{\text{v e2 --> v e2'}}\ \text{(app2)}$$

multi-step:

$$\frac{}{\text{e -->* e}}\ \text{(reflexivity)}$$

$$\frac{\text{e1 --> e2} \qquad \text{e2 -->* e3}}{\text{e1 -->* e3}}\ \text{(transitivity)}$$

Examples

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

$$
\dfrac{\dfrac{}{(\backslash x.\backslash y.\ x\ y)\ \ (\backslash w.w)\ \ \text{-->}\ \ \backslash y.(\backslash w.w)\ y}\ \text{(beta)}}{((\backslash x.\backslash y.\ x\ y)\ \ (\backslash w.w))\ \ (\backslash z.z)\ \ \text{-->}\ (\backslash y.(\backslash w.w)\ y)\ (\backslash z.z)}\ \text{(app1)}
$$

- 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

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

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

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

FIRST-ORDER SYNTAX

Examples

- Data structure:

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

- Examples:

```
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
```

Values

- 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

```
eval :: Lam -> Lam

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

$$\frac{}{(\backslash x.e)\ v \dashrightarrow e\ [v/x]}\ \text{(beta)}$$

$$\frac{e1 \dashrightarrow e1'}{e1\ e2 \dashrightarrow e1'\ e2}\ \text{(app1)}$$

$$\frac{e2 \dashrightarrow e2'}{v\ e2 \dashrightarrow v\ e2'}\ \text{(app2)}$$

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'
```

$$\frac{}{(\backslash x.e)\ v \dashrightarrow e\ [v/x]}\ \text{(beta)}$$

$$\frac{e1 \dashrightarrow e1'}{e1\ e2 \dashrightarrow e1'\ e2}\ \text{(app1)}$$

$$\frac{e2 \dashrightarrow e2'}{v\ e2 \dashrightarrow v\ e2'}\ \text{(app2)}$$

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
```

$$\frac{}{(\backslash x.e)\ v \dashrightarrow e\ [v/x]}\ \text{(beta)}$$

$$\frac{e1 \dashrightarrow e1'}{e1\ e2 \dashrightarrow e1'\ e2}\ \text{(app1)}$$

$$\frac{e2 \dashrightarrow e2'}{v\ e2 \dashrightarrow v\ e2'}\ \text{(app2)}$$

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

-- forms that don't match LHS; no rule exists
eval (Abs x e)                   = error "Value!"
eval (Var x)                     = error "Stuck!"
```

$$\frac{}{(\backslash x.e)\ v \dashrightarrow e\,[v/x]}\ \text{(beta)}$$

$$\frac{e1 \dashrightarrow e1'}{e1\ e2 \dashrightarrow e1'\ e2}\ \text{(app1)}$$

$$\frac{e2 \dashrightarrow e2'}{v\ e2 \dashrightarrow v\ e2'}\ \text{(app2)}$$

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

expressions 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

```
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

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

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

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

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

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

Abs makes a Haskell function f into a Lam

Higher-Order Abstract Syntax

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

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

e :: Lam
e = Abs f
```

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

Higher-Order Abstract Syntax

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

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

e :: Lam
e = Abs f

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

ide :: Lam
ide = Abs id
```

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

Higher-Order Abstract Syntax

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

```
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
```

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

evaluating (App e ide):

inside e, we have f = \x -> App x x

applying f to ide, we get:
App ide ide

Higher-Order Abstract Syntax

- 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
```

$$\frac{}{(\backslash x.e)\ v \dashrightarrow e\ [v/x]}\ \text{(beta)}$$

$$\frac{e1 \dashrightarrow e1'}{e1\ e2 \dashrightarrow e1'\ e2}\ \text{(app1)}$$

$$\frac{e2 \dashrightarrow e2'}{v\ e2 \dashrightarrow v\ e2'}\ \text{(app2)}$$

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

GETTING STUCK

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

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?

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: (\x.y x) TM
  - The above expression gets stuck by running in to 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?
  - ie: 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?

guarantee == sound

  - 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:
  - \y.\x.y   is closed:
  - closed (Abs "y" (Abs "x" (Var "y"))) == True

  - y (\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:
  - \y.\x.y   is closed:
  - closed (Abs "y" (Abs "x" (Var "y"))) == True

  - y (\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
  - (\x.\y.y) (\y.z) (\w.w) --> (\y.y) (\w.w) --> \w.w

A Scope Checker for HOAS Expressions

```
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)
```

A Scope Checker for HOAS Expressions

```
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) = …
```

A Scope Checker for HOAS Expressions

```
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
```

A Scope Checker for HOAS Expressions

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

$$\frac{\text{clos (x:env) e}}{\text{clos env \textbackslash x.e}}$$

-- if e is closed in (x:env) then
\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"

$$\frac{\text{clos (x:env) e}}{\text{clos env \textbackslash x.e}}$$

-- if e is closed in (x:env) then
\x.e is closed in env

$$\frac{\text{clos env e1} \qquad \text{clos env e2}}{\text{clos env (e1 e2)}}$$

-- if e1 and e2 are closed in env then
e1 e2 is closed in env

Closed Expressions

env ::= x1 : x2 : …. : []

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

$$\frac{\text{clos (x:env) e}}{\text{clos env \textbackslash x.e}}$$

-- if e is closed in (x:env) then
\x.e is closed in env

$$\frac{\text{clos env e1} \qquad \text{clos env e2}}{\text{clos env (e1 e2)}}$$

-- if e1 and e2 are closed in env then
e1 e2 is closed in env

$$\frac{\text{lookup env x == true}}{\text{clos env x}}$$

-- if x is in env then
x is closed in env

A PROOF

Evaluation Preserves Closedness

- Theorem: If clos [] e and e --> e' then clos [] e'.

Evaluation Preserves Closedness

- Theorem: If clos [] e and e --> e' then clos [] e'.
- Requires a lemma that substitution preserved Closedness
  - Lemma: If clos [] (\x.e) and clos [] v then 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 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:

$$\frac{}{(\backslash x.e)\ v \dashrightarrow e\ [v/x]}\ (\text{beta})$$

(1) clos [] ((\x.e) v) (given)

(2) clos [] (\x.e) (by 1, def of clos)

(3) clos [] v (by 1, def of clos)

(4) clos [] (e[v/x]) (by 2, 3)

$$\frac{\text{clos (x:env) e}}{\text{clos env } \backslash\text{x.e}} \qquad \frac{\text{clos env e1} \quad \text{clos env e2}}{\text{clos env (e1 e2)}} \qquad \frac{\text{lookup env x == true}}{\text{clos env x}}$$

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:

$$\frac{\texttt{e1 --> e1'}}{\texttt{e1 e2 --> e1' e2}}\ \text{(app1)}$$

(1)	clos [] (e1 e2)	(given)
(2)	clos [] e1	(by 1, def of clos)
(3)	clos [] e2	(by 1, def of clos)
(4)	clos [] e1'	(by IH, 2)
(5)	clos [] (e1' e2)	(by 4, 3, def of clos)

$$\frac{\texttt{clos (x:env) e}}{\texttt{clos env \textbackslash x.e}} \qquad \frac{\texttt{clos env e1} \quad \texttt{clos env e2}}{\texttt{clos env (e1 e2)}} \qquad \frac{\texttt{lookup env x == true}}{\texttt{clos env x}}$$

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:

$$\frac{\text{e2 --> e2'}}{\text{v e2 --> v e2'}} \quad \text{(app2)}$$

(1) clos [] (v e2)	(given)
(2) clos [] v	(by 1, def of clos)
(3) clos [] e2	(by 1, def of clos)
(4) clos [] e2'	(by IH, 3)
(5) clos [] (v e2')	(by 2, 4, def of clos)

$$\frac{\text{clos (x:env) e}}{\text{clos env \textbackslash x.e}} \qquad \frac{\text{clos env e1} \quad \text{clos env e2}}{\text{clos env (e1 e2)}} \qquad \frac{\text{lookup env x == true}}{\text{clos env x}}$$

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