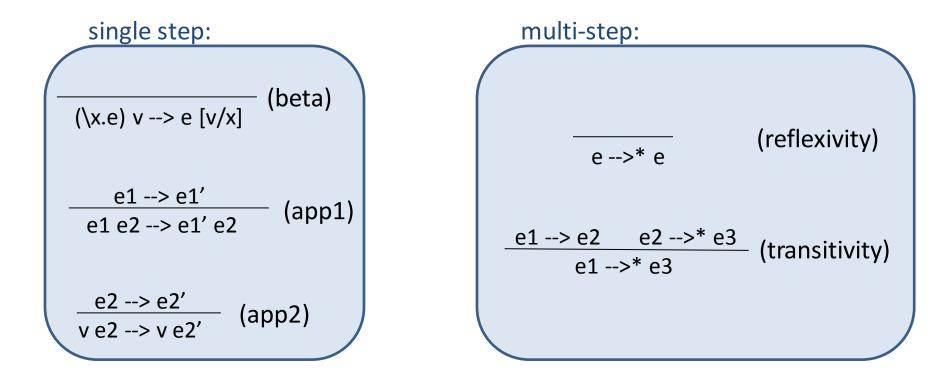
Lambda Calculus: Implementation Techniques and a Proof COS 441 Slides 15

Last Time: The Lambda Calculus

A language of pure functions:

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

With a call-by-value operational semantics:



Examples

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

$$\frac{(\langle x. \langle y. x y \rangle (\langle w. w \rangle) --> \langle y. (\langle w. w \rangle y \rangle) (\langle z. z \rangle) (\langle w. w \rangle) (\langle z. z \rangle) (\langle y. \langle w. w \rangle y \rangle) (\langle z. z \rangle)}{(\langle x. \langle y. x y \rangle (\langle w. w \rangle) (\langle z. z \rangle) --> (\langle y. (\langle w. w \rangle y \rangle) (\langle z. z \rangle) (\langle$$

- 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, tru, fls all have type Lam

• Substitution:

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

• 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:

-- 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:

-- 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:

-- 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

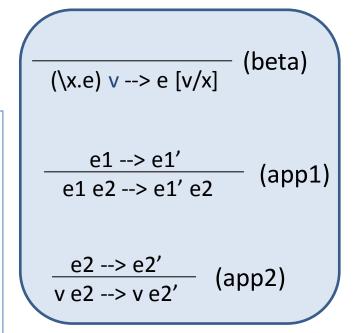
• Code to determine if an expression is a value:

-- is a value? --

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

eval :: Lam -> Lam

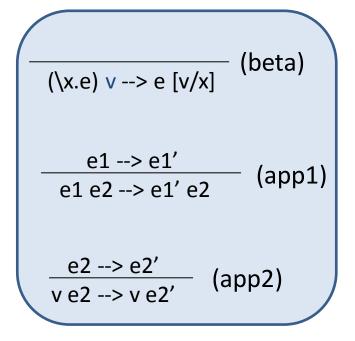
```
-- beta rule
eval (App (Abs x e) v) | value v = subst e x v
```



eval :: Lam -> Lam

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

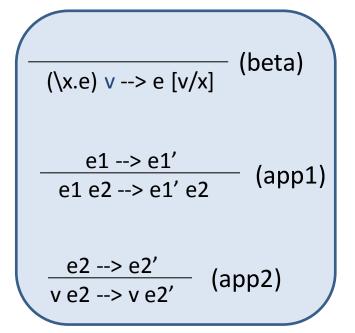
```
-- app2 rule
eval (App v e2) | value v = let e2' = eval e2 in
App v e2'
```



```
eval :: Lam -> Lam
```

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

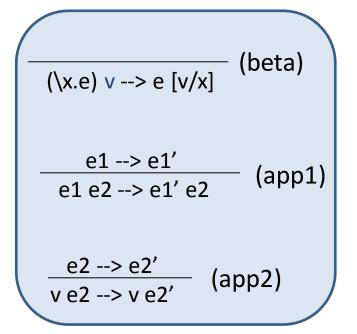
```
-- app2 rule
eval (App v e2) | value v = let e2' = eval e2 in
App v e2'
-- app1 rule
eval (App e1 e2) = let e1' = eval e1 in
App e1' e2
```



```
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
                                = error "Value!"
eval (Abs x e)
eval (Var x)
                                = error "Stuck!"
```



HIGHER-ORDER 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

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

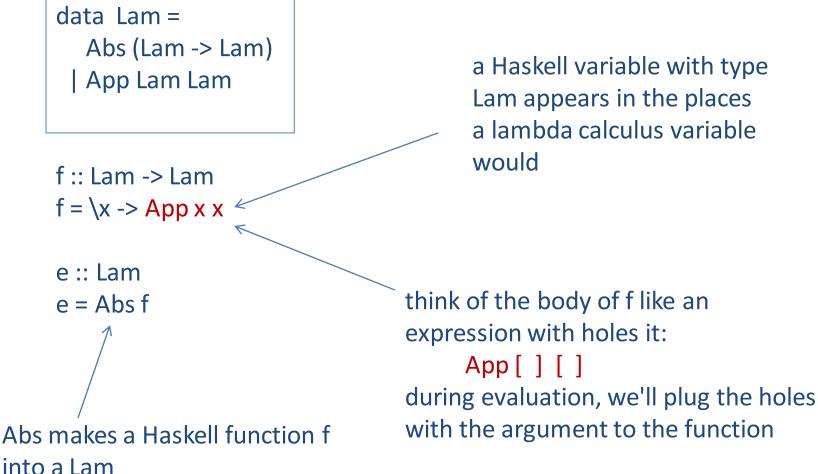


in an "object" with 1 field that has type Lam

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

App :: Lam -> Lam -> Lam

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



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

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

f :: Lam -> Lam f = x -> App x x data Lam = Abs (Lam -> Lam) | App Lam Lam

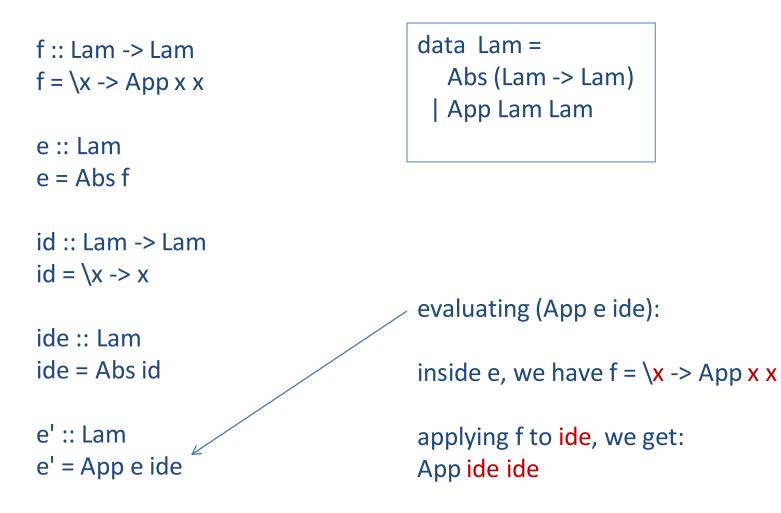
e :: Lam e = Abs f

• 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
```

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



• 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 ($\f -> f$))

An Alternative: $id = \langle f \rightarrow f$ ide = Abs id $flsf = \langle t \rightarrow Abs id$ fls = Abs flsf

An Alternative: truf = $t \rightarrow Abs (f \rightarrow t)$ tru = Abs truf

```
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
 |et 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
```

$$(x.e) \lor --> e [v/x] (beta)$$

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

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

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?
 guarantee == sound
 - 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:
 - 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
 - $(x.y.y) (y.z) (w.w) \longrightarrow (y.y) (w.w) \longrightarrow w.w$

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

env ::= x1 : x2 : : []

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

env ::= x1 : x2 : : []

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

clos (x:env) e clos env \x.e

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

env ::= x1 : x2 : : []

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

clos (x:env) e clos env \x.e

clos env e1 clos env e2 clos env (e1 e2)

- -- if e is closed in (x:env) then \x.e is closed in env
- -- if e1 and e2 are closed in env then e1 e2 is closed in env

env ::= x1 : x2 : : []

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

clos (x:env) e clos env \x.e

clos env e1 clos env e2 clos env (e1 e2)

lookup env x == true clos env x

- -- if e is closed in (x:env) then \x.e is closed in env
- -- if e1 and e2 are closed in env then e1 e2 is closed in env

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

A PROOF

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

- 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])

- 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])

- 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:

(\x.e) v --> e [v/x] (beta)

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

_clos (x:env) e	clos env e1 clos env e2	lookup env x == true
clos env \x.e	clos env (e1 e2)	clos env x

- 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:

<u>e1 --> e1'</u> e1 e2 --> e1' e2 (app1)

(1) clos [] (e1 e2)

(given)

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

(by 1, def of clos) (by IH, 2)

(by 4, 3, def of clos)

clos (x:env) e	clos env e1 clos env e2	lookup env x == true
clos env \x.e	clos env (e1 e2)	clos env x

- 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])

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

clos (x:env) e	clos env e1 clos env e2	lookup env x == true
clos env \x.e	clos env (e1 e2)	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