# CS 611: Lecture 10

More Lambda Calculus
September 20, 1999

Cornell University Computer Science Department
Andrew Myers

## Why is substitution hard?

- Substitution on lambda expressions:

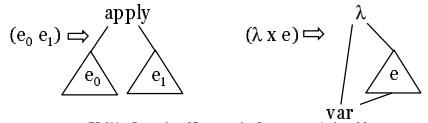
$$(\lambda\, y\, e_0)\,[e_1 / x\,] \triangleq\ (\lambda\, y' e_0[y'/y][e_1/x\,])$$

*where*

$$y' \notin FV[\![e_0]\!]$$

$$y' \notin FV[\![e_1]\!]$$

- Abstract syntax DAGs without names:





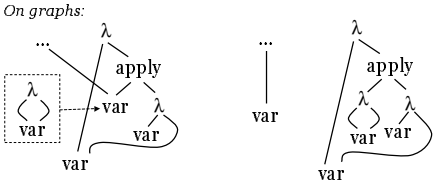

## Example

*On expressions:*

$$(\lambda\, y\, e_0)\,[e_1 / x\,] \triangleq\ (\lambda\, y'\, e_0[y'/y][e_1/x\,])$$

$$(\lambda\, a\, (b\, (\lambda\, b\, a))\,)\,[\,(\lambda\, a\, a)\,/\,b\,] = (\lambda\, a'\, (b\, (\lambda\, b\, a'))\,)\,[(\lambda\, a\, a)/\,b]) = (\lambda\, a'\, (b\, (\lambda\, b\, a'))\,)\,[(\lambda\, a\, a)\,/\,b]) = (\lambda\, a'\, ((\lambda\, a\, a)\,(\lambda\, b\, a'))\,)$$

*On graphs:*






## Holes in scope

- Reuse of an identifier creates a *hole in scope*

(λ a ((b (λ a (b a))) a))

scope of outer "a"

- Variable is inaccessible within the region in which it is shadowed
- Common source of programming errors



## Reductions

- Lambda calculus reductions
  - alpha (change of variable)
    $(\lambda\, x\, e) \xrightarrow{\alpha} (\lambda\, x'\, e[x'/x])$ if $(x' \notin FV[\![e]\!])$
  - beta (application)
    $((\lambda\, x\, e_1)\, e_2) \xrightarrow{\beta} e_1[e_2 / x]$
  - eta (removal of extra abstraction)
    $(\lambda\, x\, (e\, x)) \xrightarrow{\eta} e$ if $(x \notin FV[\![e]\!])$
- Expressions are equivalent if they
  - have the same Stoy diagram/abstract DAG
  - can be converted to each other using alpha renaming on sub-expressions
- Evaluation is a sequence of reductions



## Normal order

- Currently defined operational semantics: lazy evaluation (call-by-name)

$$\frac{e_0 \rightarrow (\lambda\, x\, e_2)}{(e_0\, e_1) \rightarrow e_2\,[\,x\,/\,e_1\,]}$$

- *Normal order evaluation*: apply β (or η) reductions to leftmost redex till no reductions can be applied (*normal form*)
- Always finds a normal form if there is one
- Substitutes *unevaluated* form of actual parameters
- Hard to understand, implement with imperative lang.

## Applicative order

- Only β-substitute when the argument is fully reduced: argument evaluated before call

$$\frac{e_1 \rightarrow_1 e'_1}{(e_0\ e_1) \rightarrow_1 (e_0\ e'_1)}$$

$$\frac{e_0 \rightarrow_1 e'_0}{(e_0\ v) \rightarrow_1 (e'_0\ v)}$$

$$\frac{}{((\lambda\, x\ e)\ v) \rightarrow_1 e[v/x]}$$

- Almost call-by-value



## Applicative order

- Applicative order may diverge even when a normal form exists
- Example:

((λb c) ((λa (a a)) (λa (a a))))

- Need special *non-strict* if form:

(*IF TRUE* 0 Y)

- What if we allow any arbitrary order of evaluation?



## Non-deterministic evaluation

$$\frac{e_0 \rightarrow_1 e'_0}{(e_0\ e_1) \rightarrow_1 (e'_0\ e_1)} \qquad \frac{e \rightarrow_1 e'}{(\lambda\, x\ e) \rightarrow_1 (\lambda\, x\ e')}$$

$$\frac{e_1 \rightarrow_1 e'_1}{(e_0\ e_1) \rightarrow_1 (e_0\ e'_1)}$$

$$((\lambda\, x\ e_1)\ e_2) \rightarrow_1 e_1[e_2 / x] \qquad (\beta)$$

$$\frac{x \notin FV[\![e]\!]}{(\lambda\, x\ (e\ x)) \rightarrow_1 e} \qquad (\eta)$$



## Church-Rosser theorem

- Non-determinism in evaluation order does not result in non-determinism of result
- Formally:

$$(e_0 \rightarrow^* e_1 \wedge e_0 \rightarrow^* e_2)$$
$$\Rightarrow \exists e_3 .\ e_1 \rightarrow^* e_3 \wedge e_2 \rightarrow^* e'_3 \wedge e_3 \stackrel{\alpha}{=} e'_3$$

- Implies: only one normal form for an expression
- Transition relation has the *Church-Rosser property* or *diamond property* if this theorem is true



## Concurrency

- Transition rules for application permit parallel evaluation of operator and operand
- Church-Rosser: any interleaving of computation yields same result
- Many commonly-used languages do not have Church-Rosser property
  C: int x=1, y = (x = 2)+x
- Intuition: lambda calculus is functional; value of expression determined locally (no store)

$$\frac{e_0 \rightarrow_1 e'_0}{(e_0\ e_1) \rightarrow_1 (e'_0\ e_1)} \qquad \frac{e_1 \rightarrow_1 e'_1}{(e_0\ e_1) \rightarrow_1 (e_0\ e'_1)}$$



## Simplifying lambda calculus

- Can we capture essential properties of lambda calculus in an even simpler language?
- Can we get rid of (or restrict) variables?
  - S & K combinators: closed expressions are trees of applications of only S and K (no variables or abstractions!)
  - de-Bruijn indices: all variable names are integers

## DeBruijn indices

- Idea: name of formal argument of abstraction is not needed

$$e ::= \lambda\, e_0 \mid\ e_0\, e_1 \mid n$$

- Variable name $n$ tells how many lambdas to walk up in AST

*IDENTITY* ≜ (λ a a) = (λ 0)

*TRUE* ≜ (λ x (λ y x)) = (λ (λ 1))

*FALSE* = 0 ≜ (λ x (λ y y)) = (λ (λ 0))

2 ≜ (λ f (λ a (f (f a))) = (λ (λ (1 (1 0))))



## Translating to DeBruijn indices

- A function DB ⟦$e$⟧ that compiles a closed lambda expression $e$ into DeBruijn index representation (closed = no free identifiers)
- Need extra argument $N : symbol \rightarrow integer$ to keep track of indices of each identifier
- DB ⟦$e$⟧ = T ⟦$e$, ∅⟧
- T ⟦($e_0\ e_1$), $N$⟧ = (T ⟦$e_0$, $N$⟧ T ⟦$e_1$, $N$⟧)
- T ⟦$x$ , $N$⟧ = $N(x)$
- T ⟦(λ $x$ $e$) , $N$⟧ =
  (λ T ⟦$e$, (λ $y$ . if $x = y$ then 0 else 1+$N(y)$)⟧)