CS 611
Advanced Programming Languages

Andrew Myers
Cornell University

Lecture 9: Reduction orders and normal forms

Reductions

- So far, two reductions that preserve the meaning of a lambda calculus expression:

$(\lambda\, x\, e) \xrightarrow{\alpha} (\lambda\, x'\, e\{x'/x\})$ (if $x' \notin FV[\![e]\!]$)

$((\lambda\, x\, e_1)\, e_2) \xrightarrow{\beta} e_1\{e_2/x\}$

Extensionality

- Two functions are equal by *extension* if they have the same meaning: they give the same result when applied to the same argument
- With lazy evaluation, expressions $(\lambda\, x\, (e\, x))$ and e are equal by extension

  $(\lambda\, x\, (e\, x))\, e' = e\, e'$ (if $x \notin FV[\![e]\!]$)

- η-reduction: $(\lambda\, x\, (e\, x)) \xrightarrow{\eta} e$ (if $x \notin FV[\![e]\!]$)

Reductions

- Three reductions that preserve the meaning of a lambda calculus expression (open *or* closed)

$(\lambda\, x\, e) \xrightarrow{\alpha} (\lambda\, x'\, e\{x'/x\})$ (if $x' \notin FV[\![e]\!]$)
$((\lambda\, x\, e_1)\, e_2) \xrightarrow{\beta} e_1\{e_2/x\}$
$(\lambda\, x\, (e\, x)) \xrightarrow{\eta} e$ (if $x \notin FV[\![e]\!]$)

Normal form

- A lambda expression is in *normal form* when no reductions can be performed on it or on any of its sub-expressions
- Normal form is defined relative to a set of allowed reductions – is a value
- Reducible expressions are called *redexes*
- What is the normal form for
  LOOP = ((λ x x) (λ x x)) ?

Normal order

- Lazy evaluation (call-by-name)

$$\frac{e_0 \Downarrow (\lambda\, x\, e_2)}{(e_0\, e_1) \Downarrow e_2\{e_1/x\}}$$

- *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

- (single-argument) call-by-value: only β-substitute when the argument is fully reduced: argument evaluated before call

$$\frac{e_0 \rightarrow e'_0}{(e_0\ e_1) \rightarrow (e'_0\ e_1)}$$

$$\frac{e_1 \rightarrow e'_1}{(v\ e_1) \rightarrow (v\ e'_1)}$$

$$\frac{}{((\lambda\, x\, e)\ v) \rightarrow e\{v/x\}}$$

Divergence

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

  ((λb c) *LOOP*)
- Need special *non-strict* if form:

  (*IF TRUE* 0 Y)
- What if we allow any arbitrary order of evaluation?

Non-deterministic evaluation

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

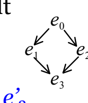
$$\frac{e_1 \rightarrow e'_1}{(e_0\ e_1) \rightarrow (e_0\ e'_1)}$$

$$((\lambda\, x\, e_1)\ e_2) \rightarrow e_1\{e_2/x\} \qquad (\beta)$$

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

Church-Rosser theorem

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



  $(e_0 \rightarrow^* e_1\ \wedge\ e_0 \rightarrow^* e_2)$

  $\Rightarrow\ \exists\, e_3\, .\, e_1 \rightarrow^* e_3 \wedge e_2 \rightarrow^* e'_3 \wedge e_3 = e'_3$
- Implies: only one normal form for an expression
- Transition relation → has the *Church-Rosser property* or *diamond property* if this theorem is true
- β+η, β–only evaluation have this property

Concurrency

- Transition rules for application permit parallel evaluation of operator and operand
- Church-Rosser: any allowed interleaving gives 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)

$$\frac{e_0 \rightarrow e'_0}{(e_0\ e_1) \rightarrow (e'_0\ e_1)}$$

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

Evaluation Contexts

- Let context C be an expression with a hole [·] where a redex may be reduced
- C[*e*] with redex *e* reduces to some C[*e*′]
- Normal order: C = [·] | C *e*

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

  $C[\lambda\, x\, (e\ x)] \rightarrow C[e] \quad (\text{if } x \notin FV[\![e]\!])$
- Applicative order: C= [·] | C *e* | (λ *x e*) C

  $C[(\lambda\, x\, e)\ v] \rightarrow C[e\{v/x\}]$

Simplifying λ 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!)
  - can reduce even to single combinator (X)
  - 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
- Need extra argument $N : Var \to \omega$ to keep track of indices of each identifier

DB ⟦e⟧ = T ⟦e, Ø⟧
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∈ Var. if x = y then 0 else 1+N(y)))

Evaluation tradeoffs

- Normal order reduction always finds normal form – but requires substitution of arbitrary expressions
- Applicative order reduction substitutes only values – but may diverge "unnecessarily"
- Can we do better?