Reductions

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

$$\lambda x e \rightarrow \lambda x' e(x'/x) \quad \text{(if } x' \notin FV\{e\})$$

$$((\lambda x e_1) e_2) \rightarrow \eta e_1(e_2/x)$$

---

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
  $$(\lambda x (e x)) e' = e e' \quad \text{(if } x \notin FV\{e\})$$
- $$\eta$$-reduction: $$(\lambda x (e x)) \eta e \quad \text{(if } x \notin FV\{e\})$$

---

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 – it is a value.
- Reducible expressions are called redexes.
- What is the normal form for $$\text{LOOP} = ((\lambda x (\lambda x x))$$?
Applicative order

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

\[
\begin{align*}
    e_0 & \rightarrow e'_0 \\
    (e_0, e_1) & \rightarrow (e'_0, e_1) \\
    e_1 & \rightarrow e'_1 \\
    (v, e_1) & \rightarrow (v, e'_1) \\
    (\lambda x e) v & \rightarrow e[v/x]
\end{align*}
\]

Divergence

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

\[
(\lambda b c) \text{LOOP}
\]
- Need special non-strict if form:

\[
(\text{IF TRUE } 0 \ Y)
\]
- What if we allow any arbitrary order of evaluation?

Non-deterministic evaluation

- 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
- Intuition: lambda calculus is functional; value of expression determined locally (no store)

\[
\begin{align*}
    e_0 & \rightarrow e'_0 \\
    (e_0, e_1) & \rightarrow (e'_0, e_1) \\
    e_1 & \rightarrow e'_1 \\
    (e_0, e_1) & \rightarrow (e'_0, e'_1) \\
    (\lambda x e) e_2 & \rightarrow e_1[e_2/x] \quad (\beta) \\
    (\lambda x (e x)) & \rightarrow e \quad (\eta)
\end{align*}
\]

Church-Rosser theorem

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

\[
\begin{align*}
    (e_0 \rightarrow^* e_1 \land e_0 \rightarrow^* e_2) \\
    \Rightarrow \exists e_3, e_1 \rightarrow^* e_3 \land e_2 \rightarrow^* e'_3 \land e_3 = e'_3
\end{align*}
\]
- Implies: only one normal form for an expression
- Transition relation $\rightarrow$ has the Church-Rosser property or diamond property if this theorem is true
- $\beta+\eta, \beta-$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
- Intuition: lambda calculus is functional; value of expression determined locally (no store)

\[
\begin{align*}
    e_0 & \rightarrow e'_0 \\
    (e_0, e_1) & \rightarrow (e'_0, e_1) \\
    e_1 & \rightarrow e'_1 \\
    (e_0, e_1) & \rightarrow (e'_0, e'_1)
\end{align*}
\]

Evaluation Contexts

- Let context C be an expression with a hole $[\cdot]$ where a redex may be reduced
- C[e] with redex e reduces to some C[e']
- Normal order: C = [\cdot] | C e

\[
\begin{align*}
    C[(\lambda x e) e_1] & \rightarrow C[e_1[e_2/x]] \\
    C[(\lambda x (e x))] & \rightarrow C[e] \quad (\text{if } x \notin \text{FV}[e])
\end{align*}
\]
- Applicative order: C= [\cdot] | C e | (\cdot x e) C

\[
\begin{align*}
    C[(\lambda x e) v] & \rightarrow C[e[v/x]]
\end{align*}
\]
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
  \[ e ::= \lambda e_0 | e_0 e_1 | n \]
- Variable name \( n \) tells how many lambdas to walk up in AST

\[ \text{IDENTITY} \triangleq (\lambda \, a \, a) = (\lambda \, 0) \]
\[ \text{TRUE} \triangleq (\lambda \, x \, (\lambda \, y \, x)) = (\lambda \, (\lambda \, 1)) \]
\[ \text{FALSE} = 0 \triangleq (\lambda \, x \, (\lambda \, y \, y)) = (\lambda \, (\lambda \, 0)) \]
\[ 2 \triangleq (\lambda \, f \, (\lambda \, a \, (f \, (f \, a)))) = (\lambda \, (\lambda \, (1 \, (1 \, 0)))) \]

Translating to DeBruijn indices

- A function \( \text{DB}[e] \) that compiles a closed lambda expression \( e \) into DeBruijn index representation
- Need extra argument \( N : \text{Var} \to \omega \) to keep track of indices of each identifier

\[ \text{DB}[e] = T[e, \emptyset] \]
\[ T[(e_0 \, e_1)]N = (T[e_0]_N \, T[e_1]_N) \]
\[ T[x]_N = N(x) \]
\[ T[(\lambda \, x \, e)]_N = (\lambda \, T[e]((\lambda \, y \in \text{Var.} \, \text{if } x = y \text{ then } 0 \text{ else } 1 + N(y))) \]

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?