Evaluation Order

Dr. Mattox Beckman

University of Illinois at Urbana-Champaign
Department of Computer Science

Objectives

- Demonstrate the difference between *normal order* and *applicative order* evaluation.
- Demonstrate the difference between *normal form* and *weak head normal form*.

Things We Didn't Mention Last Time ...

- If there is more than one β-reduction, which one do you do first?
- Do you always have to do all β-reductions, or should some be left alone?

Applicative Order

- *Applicative order* is like call-by-value in many programming languages.
- Start with the *left-most outer-most* application.
- Evaluate the argument *before* doing the β-reduction.

$$(\lambda xf.fx)(\lambda z.z)((\lambda q.q)(\lambda r.r))$$

Applicative Order

- *Applicative order* is like call-by-value in many programming languages.
- Start with the *left-most outer-most* application.
- Evaluate the argument *before* doing the β-reduction.

$$\begin{array}{rl} & (\lambda xf.fx)(\lambda z.z)((\lambda q.q)(\lambda r.r)) \\ \Rightarrow & (\lambda f.f\lambda z.z)((\lambda q.q)(\lambda r.r)) \end{array}$$

Applicative Order

- *Applicative order* is like call-by-value in many programming languages.
- Start with the *left-most outer-most* application.
- Evaluate the argument *before* doing the β-reduction.

$$\begin{array}{ll} & (\lambda xf.fx)(\lambda z.z)((\lambda q.q)(\lambda r.r)) \\ \Rightarrow & (\lambda f.f\lambda z.z)((\lambda q.q)(\lambda r.r)) \\ \Rightarrow & (\lambda f.f\lambda z.z)(\lambda r.r) \end{array}$$

Applicative Order

- *Applicative order* is like call-by-value in many programming languages.
- Start with the *left-most outer-most* application.
- Evaluate the argument *before* doing the β-reduction.

$$
\begin{array}{ll}
 & (\lambda xf.fx)(\lambda z.z)((\lambda q.q)(\lambda r.r)) \\
\Rightarrow & (\lambda f.f\lambda z.z)((\lambda q.q)(\lambda r.r)) \\
\Rightarrow & (\lambda f.f\lambda z.z)(\lambda r.r) \\
\Rightarrow & (\lambda r.r)(\lambda z.z) \\
\Rightarrow & (\lambda z.z)
\end{array}
$$

Normal Order

- *Normal order* is like call-by-name, which is what the C preprocessor uses.
- Again, start with the *left-most outer-most* application.
- But this time do the β-reduction right away.

$$(\lambda xf.fx)(\lambda z.z)((\lambda q.q)(\lambda r.r))$$

Normal Order

- *Normal order* is like call-by-name, which is what the C preprocessor uses.
- Again, start with the *left-most outer-most* application.
- But this time do the β-reduction right away.

$$\begin{array}{rl} & (\lambda xf.fx)(\lambda z.z)((\lambda q.q)(\lambda r.r)) \\ \Rightarrow & (\lambda f.f\lambda z.z)((\lambda q.q)(\lambda r.r)) \end{array}$$

Normal Order

- *Normal order* is like call-by-name, which is what the C preprocessor uses.
- Again, start with the *left-most outer-most* application.
- But this time do the β-reduction right away.

$$\begin{array}{ll} & (\lambda xf.fx)(\lambda z.z)((\lambda q.q)(\lambda r.r)) \\ \Rightarrow & (\lambda f.f\lambda z.z)((\lambda q.q)(\lambda r.r)) \\ \Rightarrow & ((\lambda q.q)(\lambda r.r))\lambda z.z \end{array}$$

Normal Order

- *Normal order* is like call-by-name, which is what the C preprocessor uses.
- Again, start with the *left-most outer-most* application.
- But this time do the β-reduction right away.

$$
\begin{aligned}
& (\lambda xf.fx)(\lambda z.z)((\lambda q.q)(\lambda r.r)) \\
\Rightarrow\ & (\lambda f.f\lambda z.z)((\lambda q.q)(\lambda r.r)) \\
\Rightarrow\ & ((\lambda q.q)(\lambda r.r))\lambda z.z \\
\Rightarrow\ & (\lambda r.r)\lambda z.z
\end{aligned}
$$

Normal Order

- *Normal order* is like call-by-name, which is what the C preprocessor uses.
- Again, start with the *left-most outer-most* application.
- But this time do the β-reduction right away.

$$\begin{array}{ll} & (\lambda xf.fx)(\lambda z.z)((\lambda q.q)(\lambda r.r)) \\ \Rightarrow & (\lambda f.f\lambda z.z)((\lambda q.q)(\lambda r.r)) \\ \Rightarrow & ((\lambda q.q)(\lambda r.r))\lambda z.z \\ \Rightarrow & (\lambda r.r)\lambda z.z \\ \Rightarrow & \lambda z.z \end{array}$$

Interesting Effects

- Applicative order often has fewer reductions.
  E.g., $(\lambda xf.fxxxx)((\lambda a.a)(\lambda b.b))$

Interesting Effects

- Applicative order often has fewer reductions.
  E.g., $(\lambda xf.fxxxx)((\lambda a.a)(\lambda b.b))$
- Normal order can win sometimes.
  E.g., $(\lambda xf.fffff)((\lambda a.a)(\lambda b.b))$

Interesting Effects

- Applicative order often has fewer reductions.
  E.g., $(\lambda xf.fxxxx)((\lambda a.a)(\lambda b.b))$
- Normal order can win sometimes.
  E.g., $(\lambda xf.fffff)((\lambda a.a)(\lambda b.b))$
- *If it terminates,* applicative order will yield the same result as normal order.
  E.g., $(\lambda xf.fffff)((\lambda a.aa)(\lambda b.bb))$

When Can We Stop?

- Consider this function definition.
- When do you expect the `(\z.z) y` function call to occur?

```
foo x y =
   x + (\z . z) y
```

Weak Head Normal Form

- If the “head node” (root node of the syntax tree) is a lambda, then everything inside is the body of the function.
- This is *weak head normal form*.
- This form more closely resembles what “real programming languages” do.

$$\begin{tikzcd}
& \lambda a \arrow[d] & \\
& @ \arrow[dl] \arrow[dr] & \\
\lambda x \arrow[d] & & \lambda y \arrow[d] \\
x & & y
\end{tikzcd}$$

$$\lambda a.(\lambda x.x)(\lambda y.y)$$

Normal Form

- In normal form, once the outermost node is a lambda, you descend into the body and continue there.
- You get maximally reduced expressions: “normalized”
- It’s possible to have α-capture though.
  E.g., $\lambda y.(\lambda xy.x)y$

$$\begin{tikzcd}
\lambda a \arrow[d] \\
\lambda y \arrow[d] \\
y
\end{tikzcd}$$

$$\lambda a.\lambda y.y$$

In Our Class

- We will tend to prefer *normal form* to *weak head normal form*.
  Why? Because this better reveals the structure of the resulting evaluations.
- We will want you to know both applicative order and normal order.
  Why? That difference will come up again later in the course!
  We will let you know if we care which one you use.