Computing Science (CMPUT) 325

Nonprocedural Programming

Department of Computing Science
University of Alberta

Normal Form

- A lambda expression that cannot be reduced further (by beta-reduction) is called a normal form
- If a lambda expression E can be reduced to a normal form, we then say that E has a normal form
- In general, a lambda expression may not have a normal form
- See counterexample next slide

A Lambda Expression without a Normal Form

- Example:
- `((lambda (x) (x x)) (lambda (z) (z z)))`
- Body `(x x)`
- Given argument `(lambda (z) (z z))`
- β-reduction: Substitute given arg. for x in body:
- `((lambda (z) (z z)) (lambda (z) (z z))))`
- α-reduction: rename first z to x
- `((lambda (x) (x x)) (lambda (z) (z z)))`
- Same...

Lambda Expression without a Normal Form continued

- One step of reduction (plus renaming) has led to an identical lambda expression
- We can reduce this again and again, infinitly often
- We never reach a normal form that can no longer be reduced
- This proves that not all lambda expressions have a normal form
- There are other examples, where the expression just grows and grows with each “reduction”

Lambda Expression without a Normal Form continued

- Similar, “almost self-replicating” lambda expressions are useful (actually indespensable) for encoding recursive functions
- We will see this later
- Note: no functional language is sufficiently powerful, if it cannot express recursive functions

Order of Reduction

- If we have nested function applications, in which order should we reduce them?
- This is a general question for function evaluation
- Any programming language has to deal with this issue
- Usually we evaluate all the arguments first, then call the function on the evaluated arguments
- We have already seen one exception: the `if` statement does not evaluate all arguments, and delays the evaluation of the `then`, `else` parts

Two Important Orders of Reduction

- Normal Order Reduction (NOR): evaluate leftmost **outermost** application
- Applicative Order Reduction (AOR): evaluate leftmost **innermost** application
- Examples and discussion on next slides

Example for Normal Order Reduction (NOR)

- Example in Fun:
- Function application `f(g(2))`
- With `f(x) = x + x`
- `g(x) = x + 1`
- Normal Order Reduction (NOR): **outermost first**
- `f(g(2))` $\longrightarrow$ `g(2) + g(2)` $\longrightarrow$ `3 + g(2)` $\longrightarrow$ `3 + 3` $\longrightarrow$ `6`
- Note: actually, the outermost function is the +. But if the built-in + requires evaluated arguments, then we need to evaluate them first

Example for Applicative Order Reduction (AOR)

- Function application `f(g(2))`
- With `f(x) = x + x`
- `g(x) = x + 1`
- Applicative Order Reduction (AOR): **innermost first**
- `f(g(2))` $\longrightarrow$ `f(3)` $\longrightarrow$ `3 + 3` $\longrightarrow$ `6`

Tie-breaking Rules

- What if there is more than one outermost or innermost function that is applicable?
- Standard tie-breaking rule: choose the leftmost one
- Example: `f(g(2)) + f(g(4))`
- Applicative Order: There are two innermost applications, `g(2)` and `g(4)`.
  So we choose `g(2)` as leftmost innermost.
- Normal Order: The outermost application is the +.
  If we cannot evaluate + until its arguments are reduced, then `f(g(2))` and `f(g(4))` are outermost,
  we start with the leftmost outermost `f`, in `f(g(2))`

Efficiency

- Normal Order Reduction: `f(g(2))` $\longrightarrow$ `g(2) + g(2)` ...
- Applicative Order Reduction: `f(g(2))` $\longrightarrow$ `f(3)` ...
- In NOR, g(2) is evaluated twice
- In AOR, only once
- AOR is generally more efficient
- However, NOR terminates more often...

An Example where NOR Terminates and AOR Does Not

- `g(x) = cons(x, g(x+1))`
  infinite nested call, trouble...
- `f(x) = 5` a constant function
- Reduce `f(g(0))`
- NOR: `f(g(0))` $\longrightarrow$ `5`
- AOR: `f(g(0))` $\longrightarrow$ `f(cons(0, g(1)))` $\longrightarrow$ `f(cons(0, cons(1, g(2))))` $\longrightarrow$ ...

Example of NOR in Lambda Calculus

- `((lambda (x) (+ 1 x))`
  `((lambda (z) (+ 1 z)) 3))`
- Normal order reduction:

$\longrightarrow$ `(+ 1 ((lambda (z) (+ 1 z)) 3))`
$\longrightarrow$ `(+ 1 (+ 1 3))`
$\longrightarrow$ `(+ 1 4)`
$\longrightarrow$ 5

Same Example with AOR

- `((lambda (x) (+ 1 x))`
  `((lambda (z) (+ 1 z)) 3))`
- Applicative order reduction:

$\longrightarrow$ `((lambda (x) (+ 1 x)) (+ 1 3))`

$\longrightarrow$ `((lambda (x) (+ 1 x)) 4)`

$\longrightarrow$ `(+ 1 4)`

$\longrightarrow$ 5

Second Example of NOR

- `((lambda (x) (+ x x)) ((lambda (z) (+ 3 z)) 2))`
- Normal order reduction:

$\longrightarrow$ `(+ ((lambda (z) (+ 3 z)) 2) ((lambda (z) (+ 3 z)) 2))` $\longrightarrow$ `(+ (+ 3 2) ((lambda (z) (+ 3 z)) 2))` $\longrightarrow$ `(+ 5 ((lambda (z) (+ 3 z)) 2))` $\longrightarrow$ `(+ 5 (+ 3 2))` $\longrightarrow$ `(+ 5 5)` $\longrightarrow$ `10`

Same Example with AOR

- `((lambda (x) (+ x x))`
  `((lambda (z) (+ 3 z)) 2))`
- Applicative order reduction:

$\longrightarrow$ `((lambda (x) (+ x x)) (+ 3 2))` $\longrightarrow$ `((lambda (x) (+ x x)) 5)` $\longrightarrow$ `(+ 5 5)` $\longrightarrow$ `10`

Church Rosser Theorem

- Church and Rosser proved two important properties of reductions and normal forms
- In this theorem, $\longrightarrow$ means a sequence of zero or more reduction steps.
- Two parts:

1. If A $\longrightarrow$ B and A $\longrightarrow$ C
   then there exists an expression D such that
   B $\longrightarrow$ D and C $\longrightarrow$ D
2. If A has a normal form E, then
   there is a normal order reduction A $\longrightarrow$ E.

Church Rosser Theorem Part 1 Comments

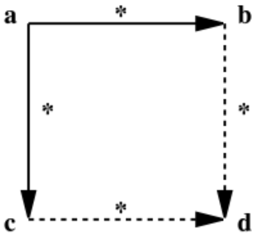

Image source:

`en.wikipedia.org/wiki/Church-Rosser_theorem`

- If A $\longrightarrow$ B and A $\longrightarrow$ C then there exists an expression D such that B $\longrightarrow$ D and C $\longrightarrow$ D
- No matter what reduction strategies are used initially to get to B and C...
- ...there is always a way to converge from both B and C back to the same expression D
- Note: this is true even if there is no normal form for A
- (Easy) exercise: prove that there is **at most** one normal form.

Church Rosser Theorem Part 2 Comments

- If A has a normal form E, then there is a normal order reduction A $\longrightarrow$ E.
- Normal order reduction guarantees termination if the given expression has a normal form
- Note: NOR can be a very inefficient and slow process in some cases. But it always works if there is a normal form.
- Note: the Theorem does not tell us whether there **is** a normal form, or how many reduction steps we would need to reach it.
- Compare with the halting problem - does a Turing machine halt on a given input? Undecidable in general.

Summary and Outlook

- Studied abstract model of computation of lambda calculus
- Clarifies foundations of functional programming
- A model that is equivalent to Turing machines (both express the same computations)
- Next steps: interpreter and "compiler" for functional programming language
- Based on reductions in lambda calculus
- Assume we have some useful built-ins
- We will explain a bit how primitive functions work, if we have time in last class before reading week
- If not, I will just post them as optional notes.