Computing Science (CMPUT) 325

Nonprocedural Programming

Department of Computing Science
University of Alberta

Reductions in Lambda Calculus

- Goal: reduce lambda expression
  to its simplest possible form
- How exactly does this process work?
- It is called “operational semantics” of lambda calculus
- In lambda calculus, computation is the process
  of reductions from one expression
  to another expression
- We have already seen some examples

Comment on Built-in Functions

- We have freely used built-in, or primitive functions such as `+, -, *, null, eq`, and more
- In lambda calculus we do **not need any** of them!
- All these functions can be defined in pure lambda calculus
- We can even represent numbers such as 0,1,2,3,... by lambda expressions
- These are deep results, some of the definitions are highly nontrivial. But it can be done.

Eliminating Built-in Functions (continued)

- We will look at some examples and ideas later
- For now, just believe me...
- Analogy: A Turing machine can compute anything, using only very primitive operations on a tape
- For convenience, we will often still use built-in functions

Important Questions about Reduction

- What types of reductions are there?
- How do we do them?
- Is there a simplest form for a given expression?
- Is there always a simplest form?
- Is it unique?
- How can we compute it?
- Can we compute it efficiently?

Beta-Reduction

- Several types of reduction have been studied in lambda calculus
- The most intuitive and important one is what we called function application
- It is called beta-reduction (or β-reduction) in the theory
- We write $\rightarrow^{\beta}$ to indicate such a reduction
- Rule: given an expression `((lambda (x)` *`body`*`) a)`,
- Reduce it to *body*, but:
- Replace all occurrences of `x` in *body* by `a`
- Example: `((lambda (x) (x (x 1))) 5)` $\rightarrow^{\beta}$ `(5 (5 1))`

Beta-Reduction Comments

- The expression we reduce could be a sub-expression nested within some complex expression
- Some complications may arise when we replace, due to name conflicts
  - We will discuss that soon
- Sometimes, the result after a reduction is actually more complex than before...
- ...Imagine a function body where x occurs many times, and gets substituted by some complex expression each time
- Also imagine how recursion will go through a long sequence of lambda expressions, with some reduction steps simply producing the next recursive call

Beta-Reduction Example

- `( (lambda (x) (x 2))`
  `(lambda (z) (+ z 1)) )`
- The body is `(x 2)`
- The x gets replaced by the argument given,
  `(lambda (z) (+ z 1))`
- The result of this beta-reduction is
  `((lambda (z) (+ z 1)) 2)`
- Now we can do another beta-reduction on the result...

Beta-Reduction Example continued

- Continuing with `((lambda (z) (+ z 1)) 2)`

- The body is `(+ z 1)`
- The z gets replaced by the argument given, 2
- The result of this beta-reduction is `(+ 2 1)`
- Now, if we had the pure lambda calculus definition of +, 2, and 1, we could reduce this further, and end up with...
- ...the definition of 3.

Alpha-Reduction Intuition

- Alpha-Reduction means renaming variables
- Intuition: changing the name of local variables in a function does not change the meaning
- Lisp example: all three are the same:
  - `(defun f (x y) (- x y))`
  - `(defun f (y z) (- y z))`
  - `(defun f (y x) (- y x))`
- However, we should not produce a “name conflict”:
- `(defun f (x x) (- x x))`
- In sbcl, compile-time error: The variable X occurs more than once in the lambda list.

Alpha-Reduction in lambda calculus

- In lambda calculus, a bound variable can be replaced by another if the latter doesn't cause any name conflict
- It is always safe if you use a new name, that does not occur anywhere else in the whole lambda expression
- Example: `(lambda (x) (+ x y))`
- x is **bound** in the scope of (lambda (x) ...)
- y is **free**
- You can rename x to anything, except y (name conflict)
- You can NOT rename y (it would change the result)

Alpha-Reduction Good and Bad Examples

- Good: `(lambda (x) (+ x y))`
  $\rightarrow^{\alpha}$ `(lambda (z) (+ z y))`
- Bad, name conflict: `(lambda (x) (+ x y))`
  $\rightarrow^{\alpha}$ `(lambda (y) (+ y y))`
  - Bad because it changes the meaning:
    the formerly free variable y in `(+ x y)`
    has become bound in
    `(lambda (y) (+ y y))`
- Also bad: `(lambda (x) (+ x y))`
  $\rightarrow^{\alpha}$ `(lambda (x) (+ x z))`
  - Cannot rename free variables, can only rename bound variables

Free vs Bound Variables - More Details

- Free and bound are not absolute concepts, they depend on the scope
- Example:
- `(lambda (z) (lambda (x) (+ x z)))`
- z is free in the scope of `(lambda (x) (+ x z))`
- z is bound in the scope of `(lambda (z) ...)`
- Compare with other programming languages: local variable vs variable from surrounding block (or even global variable)

Avoiding Name Conflicts

- Avoid name conflicts in alpha-reduction:
- Always use a new variable name
- Direct substitution
  (without first using an alpha-reduction)
  may not always work correctly!
- The following example shows why.

Example Where Direct Substitution Goes Wrong

- `((lambda (x) (lambda (z) (x z))) z)`
- This lambda expression applies:
  - A function of x,
  - with body `(lambda (z) (x z))`,
  - to the argument `z`
- There is a name conflict between
  - the z bound in `(lambda (z) (x z))`
  - the (free) argument z in `((lambda (x) ...) z)`

Direct Substitution Without Renaming

- Let's see what happens if we blindly do beta-reduction
- `((lambda (x) (lambda (z) (x z))) z)`
- Replace x by z in body gives:
- `(lambda (z) (z z))`
- This is wrong! The former free (within scope) x got changed into z, but in this scope z is a bound variable.

Do Alpha-reduction First

- `((lambda (x) (lambda (z) (x z))) z)`
- Which z can we rename? Only the one in `(lambda (z) ...)`
- Rename that z to u in the whole scope of this function:
- `((lambda (x) (lambda (u) (x u))) z).`
- Now, the bound variable is called u and will not conflict with the argument z. With beta-reduction we now get the answer:
- `(lambda (u) (z u))`

A Demonstration of the Difference

- Here, we show that
  `(lambda (z) (z z))` and
  `(lambda (u) (z u))` are different functions
- Lets apply each of them to the same argument, say `a`.
- `((lambda (z) (z z)) a)` $\rightarrow^{\beta}$ `(a a)`
- `((lambda (u) (z u)) a)` $\rightarrow^{\beta}$ `(z a)`

Scope of Variables and Beta-Reduction

- The scope of a variable should be preserved by variable renaming to ensure that reduction is correct
- `((lambda (x) (lambda (z) (x z))) z)`
  $\rightarrow^{\beta}$ `(lambda (u) (z u))`
- where u is some new variable
- Correct beta reductions can always be achieved by
  - renaming (alpha-reduction), if needed
  - followed by a beta-reduction using direct substitution
- `((lambda (x) (lambda (z) (x z))) z)`
  $\rightarrow^{\alpha}$ `((lambda (x) (lambda (u) (x u))) z)`
  $\rightarrow^{\beta}$ `(lambda (u) (z u))`

Summary of Reductions

- One β-reduction corresponds to one-step function application
- The substitution of the formal variable by the argument must be done carefully to avoid name conflicts
- α-reduction renames function arguments
- After using such renaming where necessary, a simple substitution gives a correct beta-reduction
- To be safe we can always use α-reduction with brand-new names for bound variables

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

Compare: Infinite "Reduction" in Lisp

- Simple example in Lisp:
  `(defun f (x) (+ (f x) (f x)))`
- `* (f 3)`
- `... Control stack exhausted (no more space for function call frames). This is probably due to heavily nested or infinitely recursive function calls, or a tail call that SBCL cannot or has not optimized away.`
- Even simpler: `(defun g (x) (g x))`
- `* (g 3)`
- SBCL optimizes away the tail recursion, and just goes into an infinite loop...

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

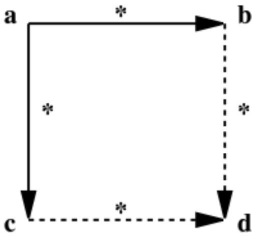

Image source:

`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.