CSCI 334: Principles of Programming Languages

Lecture 7: Evaluation by Rewriting

Instructor: Dan Barowy

Williams

Topics

Lambda calculus—how to evaluate it

Your to-dos

1. Lab 3, **due Sunday 2/27** (individual lab)
2. Reading response, **due Wednesday 3/2**.

Lambda calculus: relevance

Fundamental technique for building programming languages that work **correctly** (and **intuitively**!).

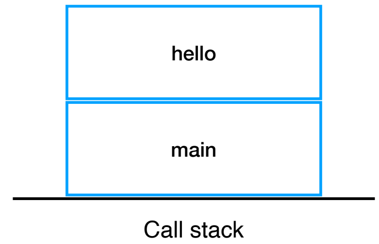
But it can also be leveraged to do some **seemingly magical** things, like **type inference**:

```
Vector<Association<String,FrequencyList>> table =
   new Vector<Association<String,FrequencyList>>();

Vector<Association<String,FrequencyList>> table = new Vector<>();

let table = new Vector<>();

…
```

Class Lambda Grammar

```
<expr>    ::= <value>
            |  <abs>
            |  <app>
            |  <parens>
<var>     ::= α ∈ { a ... z }
<abs>     ::= λ<var>.<expr>
<app>     ::= <expr><expr>
<parens>  ::= (<expr>)
<value>   ::= v ∈ ℕ
            |  <var>
```

Evaluation: You know how C does it




```c
#include <stdio.h>

void hello() {
  printf("Hello world!\n");
}

int main() {
  hello();
  return 0;
}
```

Evaluation: Lambda calculus is like algebra

`(λx.x)x`

Evaluation consists of simplifying an expression using text substitution.

Only two simplification rules:

α-reduction

β-reduction

α-Reduction

`(λx.x)x`

This expression has two **different** `x` variables

Which should we rename?

Rule:

$\lambda x.\langle expr\rangle =_{\alpha} \lambda y.[y/x]\langle expr\rangle$

`[y/x]<expr>` means “substitute `y` for `x` in `<expr>`”

α-Reduction

(λ**x**.**x**) x	given
(λ**y**.[**y**/**x**]**x**) x	α-reduce **x** with **y** (binding)
(λ**y**.**y**) x	α-reduce **x** with **y** (expr)

β-Reduction

(λx.x) y

How we “call” or **apply** a function to an argument

Rule:

(λx.<expr>) y $=_\beta$ [y/x]<expr>

Let’s reduce this

(λx.x) x

How far do we go?

We keep going until there is **nothing left to simplify**.

x	←	done
xx	←	done
λx.y	←	done
(λx.xy) z	←	not done

That “most simplified” expression is called a **normal form**.

An expression that can be simplified is a called a **redex**.

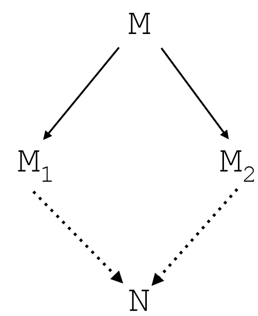
Try this one with a partner

`(λx.λy.yx)xy`

(don't forget precedence/associativity rules)

Sometimes multiple simplifications

Order (mostly) does not matter



If $M \rightarrow M_1$ and $M \rightarrow M_2$
then $M_1 \rightarrow^* N$ and $M_2 \rightarrow^* N$
for some N

"confluence"

Activity

Leftmost reduction:

`(λf.λx.f(f x))(λz.(+ x z))2`

Activity

Rightmost reduction:

`(λf.λx.f(f x))(λz.(+ x z))2`

Recap & Next Class

Today:

Lambda calculus: how to evaluate

Next class:

LISP