CSCI 740 - Programming Language Theory

Lecture 8

The λ-calculus Semantics

Instructor: Hossein Hojjat

September 18, 2017

Recap

- A λ-calculus expression is defined as

$$\begin{array}{rcll} e & ::= & x & \text{variable} \\ & | & \lambda x.e & \text{abstraction} \\ & | & e\ e & \text{application} \end{array}$$

α-conversion

$$\lambda x.e \ \rightarrow_\alpha \ \lambda y.e[x \mapsto y] \qquad \text{if } y \notin \mathsf{FV}(e)$$

β-reduction

$$(\lambda x.e)\ e' \ \rightarrow_\beta \ e[x \mapsto e']$$

η-conversion

$$\lambda x.(e\ x) \ \rightarrow_\eta \ e \qquad \text{if } \ x \notin \mathsf{FV}(e)$$

Meaning of an Expression

- Giving meaning to syntax is called **semantics**
- Semantics must distinguish between expressions that should be unequal
- **Example:**

$$\begin{array}{ll} c_0 & \lambda s.\lambda z.z \\ c_1 & \lambda s.\lambda z.s\ z \\ c_2 & \lambda s.\lambda z.s\ (s\ z) \end{array}$$

- Then $c_0 \neq c_1 \neq c_2$
- Semantics must equate expressions that should be equal
- **Example:** expressions corresponding to (PLUS c_0 c_1) and c_1 must have the same meaning
- A semantics is fully abstract if it distinguishes expressions just when, in some context, they show observably different behavior

Information Content

- Instantaneous information: An expression obtained by replacing each redex in a term by $\perp$ where $\perp$ stands for no information

	Term	Instantaneous information
	$(\lambda x.\lambda y.y\ (x\ t))(\lambda z.z)$	$\perp$
$\rightarrow_\beta$	$\lambda y.y\ ((\lambda z.z)\ t)$	$\lambda y.(y\ \perp)$
$\rightarrow_\beta$	$\lambda y.y\ t$	$\lambda y.y\ t$

- β-reductions monotonically increase information
- The meaning of an expression is the **maximum information** that can be obtained by β-reductions

Normal Forms

- e is in normal form if there is no e' such that $e \to_\beta e'$
- e is normalizable if there is some e' such that $e \to_\beta^* e'$ and e' is in normal form
- Normal forms correspond to the results of computations
- Is the meaning of a term simply its normal form?

Normal Forms

- e is in normal form if there is no e' such that $e \to_\beta e'$
- e is normalizable if there is some e' such that $e \to_\beta^* e'$ and e' is in normal form
- Normal forms correspond to the results of computations
- Is the meaning of a term simply its normal form?
- What if a term does not have any normal form?

$$(\lambda x.xx)(\lambda x.xx) \to_\beta (\lambda x.xx)(\lambda x.xx) \to_\beta \cdots$$

$$(\lambda x.xxx)(\lambda x.xxx) \to_\beta (\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx) \to_\beta \cdots$$

Normal Forms

- $\Omega = (\lambda x.x\ x)(\lambda x.x\ x)$ does not have a normal form
  - Meaning of Ω is $\bot$
- What about $\lambda x.x\ \Omega$?
- It does not have a normal form, but its meaning is $\lambda x.x\ \bot$

Normal Forms

- $\Omega = (\lambda x.x\ x)(\lambda x.x\ x)$ does not have a normal form
  - Meaning of Ω is $\bot$
- What about $\lambda x.x\ \Omega$?
- It does not have a normal form, but its meaning is $\lambda x.x\ \bot$

- What if an expression has more than one normal form?
- It would have two meanings then!
- Can the choice of redexes lead to produce different expressions?
  - e.g., x versus $\lambda y.y$?
- **Example.** Does it matter for the normal form to start with e_1 or e_2?

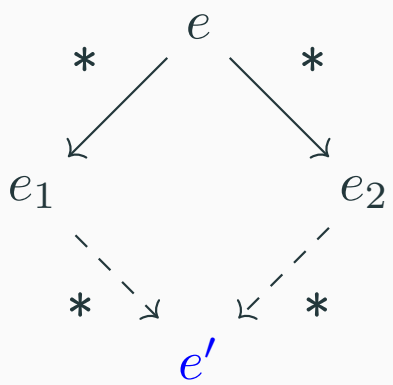
$$\underbrace{\big((\lambda x.M)\ A\big)}_{e_1}\ \underbrace{\big((\lambda x.N)\ B\big)}_{e_2}$$

Confluence

Church-Rosser Theorem (Confluence)

For any lambda expressions e, e_1, and e_2, if $e \rightarrow_\beta^* e_1$ and $e \rightarrow_\beta^* e_2$, then there exists a λ-expression e' such that

$e_1 \rightarrow_\beta^* e'$ and $e_2 \rightarrow_\beta^* e'$



Confluence

Corollary: no λ-expression can be reduced to two distinct normal forms

Proof.

- Suppose $e \rightarrow_\beta^* e_1$ and $e \rightarrow_\beta^* e_2$ and e_1, e_2 are normal forms
- Church-Rosser theorem says there must be an expression e' such that e_1 and e_2 are each reducible to it
- Since e_1 and e_2 are in normal form, they cannot have any redexes so $e_1 = e_2 = e'$

- All reduction sequences that terminate will always yield the same result and that result must be a normal form

Evaluation Strategies

- Question: Which redex should be picked?
- **Good news:** (Church-Rosser Theorem) independent of strategy, there is at most one normal form
- **Bad news:** some strategies may fail to find a normal form

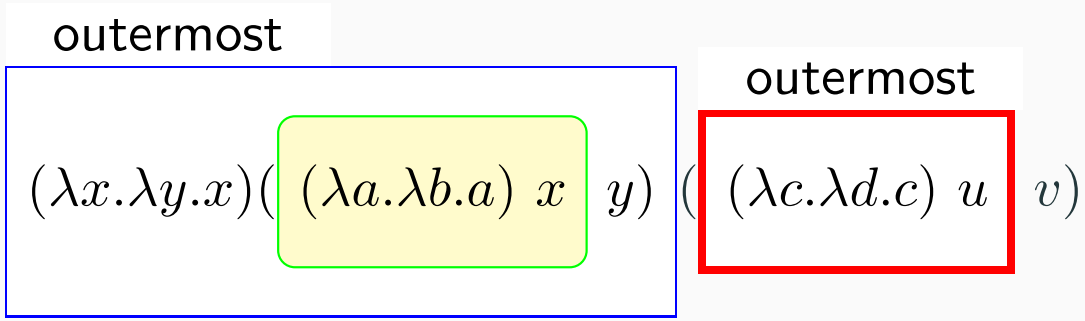
Example

- $(\lambda x.y)\big((\lambda x.x\ x)\ (\lambda x.x\ x)\big)$
- This is an application of the function $\lambda x.y$ to the argument $(\lambda x.x\ x)\ (\lambda x.x\ x)$
- The evaluation order that tries to evaluate the argument before calling the outermost function will never terminate
- Outermost function ignores it's argument and just returns y

$$(\lambda x.y)\big((\lambda x.x\ x)\ (\lambda x.x\ x)\big)\ \rightarrow_\beta\ y$$

Normal-Order Reduction

- **Outermost redex:** a redex not contained inside another redex



- **Normal order:** leftmost outermost redex first
- **Theorem:** If e has a normal form e' then normal order reduction will reduce e to e'

Normal-Order Reduction

- Why not normal order?
- Normal order reduces under λ

$$\lambda x.((\lambda y.y\ y)(\lambda y.y\ y)) \ \rightarrow_\beta \ \lambda x.((\lambda y.y\ y)(\lambda y.y\ y)) \ \rightarrow_\beta \ \cdots$$

- In most programming languages, functions (λ-abstractions) are considered "values" (fully evaluated)
- **Weak Reduction:**
  - No reduction done under λ's (i.e. inside a function body)
  - Functional programming languages implement only weak reduction
- **Strong Reduction:**
  - Reduction is done under λ's
  - Practical applications in partial evaluation of functions and type checking
  - See e.g. Benjamin Grégoire and Xavier Leroy: "A Compiled Implementation of Strong Reduction" (ICFP'02)

Call-by-Name (CBN) Reduction

- Don't reduce under λ
- Don't evaluate the argument to a function call
- Directly substitute; evaluate when reducing body
- Demand-driven
  - an expression is not evaluated unless needed in body
- Converges whenever normal order converges
- But does **not** always evaluate to a normal form

Call-by-Name (CBN) Reduction

$$
\begin{aligned}
& (\lambda y.(\lambda x.x)\ y)\ \underbrace{((\lambda u.u)(\lambda v.v))} \\
\to_\beta\ & (\lambda x.x)\ \underbrace{((\lambda u.u)(\lambda v.v))} \\
\to_\beta\ & (\lambda u.u)\ \underbrace{(\lambda v.v)} \\
\to_\beta\ & \lambda v.v
\end{aligned}
$$

Call-by-Name (CBN) Reduction

- ALGOL 60 allows the call-by-name evaluation strategy
- The Algol 60 report describes call-by-name as follows:

> 1. Actual parameters are textually substituted for the formals. Possible conflicts between names in the actuals and local names in the procedure body are avoided by renaming the locals in the body.
> 2. The resulting procedure body is substituted for the call. Possible conflicts between nonlocals in the procedure body and locals at the point of call are avoided by renaming the locals at the point of call.

Call-by-Name (CBN) Reduction

- C/C++ macros effectively pass parameters using call by name
- **Macro Expansion:**
  A "call" to the macro replaces the macro by the body of the macro
- `max(func(),0)` is replaced by `((func())>(0)?(func()):(0))`
- Notice that the argument expressions get duplicated (and hence re-evaluated) wherever the body refers to the argument name

```
#include <stdio.h>
#define max(a,b) ( (a)>(b) ? (a) : (b) )
int func() {
  printf("here!\n");
  return 10;}
int main ( int argc , char *argv[] ) {
  max(func(),0);
  return 0;}
```

- What does the program print?

Call-by-Value (CBV) Reduction

- Don't reduce under λ
- Do evaluate argument to function call
- Most languages are call-by-value
- Not normalizing

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

- Diverges, but normal order (or Call-by-Name Reduction) converges

Call-by-Value (CBV) Reduction

Example:

evaluate arg

$$
\begin{aligned}
& (\lambda y.(\lambda x.x)\ y)\ \boxed{((\lambda u.u)(\lambda v.v))} \\
\to_\beta\ & (\lambda y.(\lambda x.x)\ y)(\lambda v.v) \\
\to_\beta\ & (\lambda y.y)(\lambda v.v) \\
\to_\beta\ & \lambda v.v
\end{aligned}
$$

CBV vs. CBN

Call-by-value (CBV):

- Easy to implement
- May diverge

Call-by-name (CBN):

- More difficult to implement
  - Must pass unevaluated expressions
  - Arguments multiply-evaluated inside function body
- Simpler theory than call-by-value
- Terminates more often (always if normal form exists)
- e.g. if arg is non-terminating, but not used...

There are various other reduction strategies (not covered in this course)

Big Step Operational Semantics

- Model the execution in an abstract machine

Basic Notation: **Judgments**

$$\langle \text{configuration} \rangle \Downarrow \text{result}$$

- Describes how a program configuration is evaluated into a result
- Configuration is usually a program fragment together with any state

Basic Notation: **Inference rules**

- Define how to derive judgments for an arbitrary program
- Also called derivation rules or evaluation rules
- Usually defined recursively

$$\frac{\langle c_1 \rangle \Downarrow r_1 \qquad \langle c_2 \rangle \Downarrow r_2 \qquad \cdots \qquad \langle c_k \rangle \Downarrow r_k}{\langle \text{configuration} \rangle \Downarrow \text{result}}$$

Call-by-Name (CBN) Reduction

$$\frac{}{x \Downarrow x}$$

$$\frac{}{\lambda x.e \Downarrow \lambda x.e}$$

$$\frac{e_1 \Downarrow \lambda x.e_1' \qquad e_1'[x \mapsto \alpha(e_2)] \Downarrow e_3}{e_1\ e_2 \Downarrow e_3}$$

Call-by-Value (CBV) Reduction

$$\frac{}{x \Downarrow x}$$

$$\frac{}{\lambda x.e \Downarrow \lambda x.e}$$

$$\frac{e_1 \Downarrow \lambda x.e_1' \qquad e_2 \Downarrow e_2' \qquad e_1'[x \mapsto \alpha(e_2')] \Downarrow e_3}{e_1\ e_2 \Downarrow e_3}$$