CMSC 330: Organization of Programming Languages

Lambda Calculus
Turing Machine

Infinite Tape

1 0 0 0 1 1 1 0 ...

Read / Write Head

Control Unit
State: Y

START

HALT

b : b, R

3

4

START

b : b, R

START

e : e, R

START

b : b, R
Turing Completeness

- Turing machines are the most powerful description of computation possible
  - They define the Turing-computable functions
- A programming language is Turing complete if
  - It can map every Turing machine to a program
  - A program can be written to emulate a Turing machine
  - It is a superset of a known Turing-complete language
- Most powerful programming language possible
  - Since Turing machine is most powerful automaton
Programming Language Expressiveness

- So what language features are needed to express all computable functions?
  - What’s a minimal language that is Turing Complete?
- Observe: some features exist just for convenience
  - Multi-argument functions  
    - Use currying or tuples
  - Loops  
    - Use recursion
  - Side effects  
    - Use functional programming pass “heap” as an argument to each function, return it when with function’s result:
      
      ```
      effectful : `a → `s → ( `s * `a )
      ```

It is not difficult to achieve Turing Completeness
- Lots of things are ‘accidentally’ TC

Some fun examples:
- x86_64 `mov` instruction
- Minecraft
- Magic: The Gathering
- Java Generics

There’s a whole cottage industry of proving things to be TC

But: What is a “core” language that is TC?
Lambda Calculus (λ-calculus)

- Proposed in 1930s by
  - Alonzo Church
    (born in Washington DC!)
- Formal system
  - Designed to investigate functions & recursion
  - For exploration of foundations of mathematics
- Now used as
  - Tool for investigating computability
  - Basis of functional programming languages
    - Lisp, Scheme, ML, OCaml, Haskell…
Why Study Lambda Calculus?

- It is a “core” language
  - Very small but still Turing complete
- But with it can explore general ideas
  - Language features, semantics, proof systems, algorithms, …
- Plus, higher-order, anonymous functions (aka *lambdas*) are now very popular!
  - C++ (C++11), PHP (PHP 5.3.0), C# (C# v2.0), Delphi (since 2009), Objective C, Java 8, Swift, Python, Ruby (Procs), … (and functional languages like OCaml, Haskell, F#, …)
  - Excel, as of 2021!
Lambda Calculus Syntax

- A lambda calculus expression is defined as
  \[ e ::= x \text{ variable} \]
  \[ | \lambda x.e \text{ abstraction (fun def)} \]
  \[ | e \ e \text{ application (fun call)} \]

  - This grammar describes ASTs; not for parsing - ambiguous!
  - Lambda expressions also known as lambda terms

- \( \lambda x.e \) is like \texttt{(fun x -> e)} in OCaml

That’s it! Nothing but higher-order functions
Lambda Calculus Syntax Ambiguity

- How is parsing ambiguous?
- Let’s try: \( \lambda x.x \ x \)
Lambda Calculus Syntax Ambiguity

- How is parsing ambiguous?
- Let’s try: $\lambda x.x \ x$

\[
\begin{align*}
E & \rightarrow V \mid L \mid A \\
L & \rightarrow \lambda V.E \\
A & \rightarrow E \ E \\
V & \rightarrow v \mid \epsilon
\end{align*}
\]
While this means that our grammar is not so useful for *parsing*, it is still useful for write LC terms if we follow some conventions.

Almost all literature you will find uses two syntactic conventions.

We add a third convention that is very common ‘syntactic sugar’ for ease of reading larger LC terms.
Disambiguating: Three Conventions

- Scope of $\lambda$ extends as far right as possible
  - Subject to scope delimited by parentheses
  - $\lambda x. \lambda y. x \ y$ is same as $\lambda x. (\lambda y. (x \ y))$

- Function application is left-associative
  - $x \ y \ z$ is $(x \ y) \ z$
  - Same rule as OCaml

- As a convenience, we use the following “syntactic sugar” for local declarations
  - let $x = e1$ in $e2$ is short for $(\lambda x. e2) \ e1$
Warmup Quiz

- Revisiting \( \lambda x.x \) considering our conventions
- Which parse tree is it?

\[
E \rightarrow V | L | A
\]
\[
L \rightarrow \lambda V.E
\]
\[
A \rightarrow E E
\]
\[
V \rightarrow v | \varepsilon
\]
Warmup Quiz

- Revisiting $\lambda x.x \, x$ considering our conventions
- Which parse tree is it?

$$E \rightarrow V \mid L \mid A$$
$$L \rightarrow \lambda V.E$$
$$A \rightarrow E \, E$$
$$V \rightarrow v \mid \varepsilon$$
Quiz #1

\[ \lambda x. (y z) \] and \[ \lambda x. y z \] are equivalent

A. True
B. False
Quiz #1

\( \lambda x. (y \ z) \) and \( \lambda x. y \ z \) are equivalent

A. True
B. False
Quiz #2

This term is equivalent to which of the following?

\[ \lambda x. x \ a \ b \]

A. \( (\lambda x. x) \ (a \ b) \)
B. \( (((\lambda x. x) \ a) \ b) \)
C. \( \lambda x. (x \ (a \ b)) \)
D. \( (\lambda x. ((x \ a) \ b)) \)
Quiz #2

This term is equivalent to which of the following?

\[ \lambda x. x \ a \ b \]

A. \((\lambda x. x) \ (a \ b)\)
B. \(( ((\lambda x. x) \ a) \ b)\)
C. \(\lambda x. \ (x \ (a \ b))\)
D. \((\lambda x. \ ((x \ a) \ b))\)
But what does it mean?

- Many ways to define the semantics of LC
- We will look at two
  - Operational Semantics
  - Definitional Interpreter
Lambda Calculus Semantics

- Evaluation: All that’s involved are function calls
  \((\lambda x. e1) \, e2\)
  - Evaluate \(e1\) with \(x\) replaced by \(e2\)
- This application is called **beta-reduction**
  - \((\lambda x. e1) \, e2 \rightarrow e1[x:=e2]\)
    - \(e1[x:=e2]\) is \(e1\) with occurrences of \(x\) replaced by \(e2\)
    - This operation is called **substitution**
      - Replace formals with actuals
      - Instead of using environment to map formals to actuals
- We allow reductions to occur **anywhere** in a term
  - Order reductions are applied does not affect final value!
- When a term **cannot be reduced further** it is in beta normal form
### Beta Reduction Example

- \((\lambda x.\lambda z.x \ z) \ y\)
  - \(\rightarrow (\lambda x.(\lambda z. (x \ z))) \ y\) \hspace{1cm} \text{// since \(\lambda\) extends to right}
  - \(\rightarrow (\lambda x.(\lambda z. (x \ z))) \ y\) \hspace{1cm} \text{// apply \((\lambda x. e1) \ e2 \rightarrow e1[x:=e2]\)}
  - \(\rightarrow (\lambda z. (y \ z))\) \hspace{1cm} \text{// where \(e1 = \lambda z. (x \ z), e2 = y\)}
  - \(\rightarrow \lambda z. (y \ z)\) \hspace{1cm} \text{// final result}

#### Equivalent OCaml code
- \((\mathtt{\text{fun}} \ x \rightarrow (\mathtt{\text{fun}} \ z \rightarrow (x \ z)))) \ y \rightarrow \mathtt{\text{fun}} \ z \rightarrow (y \ z)\)
Big-Step Operational Semantics

- Beta reduction says how to evaluate a single call
  - It doesn’t say how to evaluate a term with many function calls in it
- We can use operational semantics to “fully evaluate” a term in one “big step”

![Reduction Diagram]

Beta reduction, here

\[
(\lambda x.e_1) \Downarrow (\lambda x.e_1)
\]

\[
e_1 \Downarrow (\lambda x.e_3)
\]

\[
e_2 \Downarrow e_4
\]

\[
e_3[x:=e_4] \Downarrow e_5
\]

\[
e_1 \ e_2 \Downarrow e_5
\]
Two Varieties

- There are two common variants of big-step semantics
  - *Eager* evaluation (aka *strict*, or *call by value*)
  - *Lazy* evaluation (aka *call by name*)
Eager

- Notice that we evaluated the argument $e_2$ before performing the beta-reduction
  - This is the first version we saw
- Hence, *eager*

\[
\begin{align*}
(\lambda x. e_1) & \Downarrow (\lambda x. e_1) \\
\end{align*}
\]

\[
\begin{align*}
e_1 & \Downarrow (\lambda x. e_3) & e_2 & \Downarrow e_4 & e_3[x:=e_4] & \Downarrow e_5 \\
e_1 \; e_2 & \Downarrow e_5 \\
\end{align*}
\]
Lazy

- Alternatively, we could have performed beta reduction *without* evaluating \( e_2 \); use it as is
  - Hence, *lazy*

\[
\begin{align*}
(\lambda x. e_1) & \Downarrow (\lambda x. e_1) \\
\text{e1} \Downarrow (\lambda x. e_3) & \quad \text{e3}[x:=\text{e2}] \Downarrow \text{e4} \\
\text{e1 e2} & \Downarrow \text{e4}
\end{align*}
\]
Small Step Semantics

- Operational semantics rules we have seen have always been “big step”, i.e., complete evaluation
  - $e \downarrow e'$ says that $e$ will terminate as $e'$
- This is a little unsatisfying
  - It doesn’t account for nontermination
  - It doesn’t identify where a program fails to progress
- Small-step semantics addresses these problems
  - $e \rightarrow e'$ in small-step says $e$ takes one step to $e'$
  - We say a term $e_1$ can be \textit{beta-reduced} to term $e_2$ if $e_1$ steps to $e_2$ after one or more steps
Small-Step Rules of LC

- Here are the “small-step” (→) rules:

\[
\begin{align*}
\frac{e_1 \to e_2}{\lambda x.e_1 \to \lambda x.e_2} \\
\frac{e_2 \to e_3}{e_1 \cdot e_2 \to e_1 \cdot e_3} \\
\frac{e_3}{e_1 \cdot e_2 \to e_1 \cdot e_3 \cdot e_2} \\
\frac{e_1 \to e_3}{(\lambda x.e_1) \cdot e_2 \to e_1[x := e_2]}
\end{align*}
\]
Evaluation Strategies

- These rules are highly flexible
  - It might be that for a given program, there are several possible rules that could apply

- Typically, a programming language will choose an *evaluation strategy* which is described by using only a **subset of these rules**. Examples:
  - Call by Value
  - Call by Need
  - Partial Evaluation
Call by Value

- Before doing a beta reduction, we make sure the argument cannot, itself, be further evaluated
- This is known as call-by-value (CBV)
  - This is the Eager big step approach

\[
\begin{align*}
e1 & \rightarrow e3 \\
e1 \ e2 & \rightarrow e3 \ e2
\end{align*}
\]

\[
\begin{align*}
e2 & \rightarrow e3 \\
e1 \ e2 & \rightarrow e1 \ e3
\end{align*}
\]

\[
e = (\lambda x.e2) \text{ or } e = y
\]

\[
(\lambda x.e1) \ e \rightarrow e1[x:=e]
\]
Beta Reductions (CBV)

- $(\lambda x. x) \ z \rightarrow z$

- $(\lambda x. y) \ z \rightarrow y$

- $(\lambda x. x \ y) \ z \rightarrow z \ y$
  - A function that applies its argument to $y$
Beta Reductions (CBV)

- \((\lambda x. x \ y) \ (\lambda z. z) \rightarrow (\lambda z. z) \ y \rightarrow y\)

- \((\lambda x. \lambda y. x \ y) \ z \rightarrow \lambda y. z \ y\)
  - A curried function of two arguments
  - Applies its first argument to its second

- \((\lambda x. \lambda y. x \ y) \ (\lambda z. zz) \ x \rightarrow (\lambda y. (\lambda z. zz)y) x \rightarrow (\lambda z. zz) x \rightarrow x \ x\)
Quiz #3

\((\lambda x. y) \ z\) can be beta-reduced to

A. \(y\)
B. \(y \ z\)
C. \(z\)
D. cannot be reduced
Quiz #3

$$(\lambda x. y) \; z \text{ can be beta-reduced to}$$

A. $y$
B. $y \; z$
C. $z$
D. cannot be reduced
Quiz #4

Which of the following reduces to $\lambda z. z$?

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

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

c) $(\lambda y. y) (\lambda x. \lambda z. z) w$

d) $(\lambda y. \lambda x. z) z (\lambda z. z)$
Quiz #4

Which of the following reduces to $\lambda z. z$?

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

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

c) $(\lambda y. y) (\lambda x. \lambda z. z) w$

d) $(\lambda y. \lambda x. z) z (\lambda z. z)
Evaluation Order

- The CBV rules we saw permit small-stepping either the function part or the argument part
  - If both are possible, the rules allow either one
    
    | e1 → e3 | e2 → e3 |
    |--------|--------|
    | e1 e2 → e3 e2 | e1 e2 → e1 e3 |

- Here’s how we would require left-to-right order

<table>
<thead>
<tr>
<th>e1 → e3</th>
<th>e1 = y or e1 = λx.e</th>
</tr>
</thead>
<tbody>
<tr>
<td>e1 e2 → e3 e2</td>
<td>e2 → e3</td>
</tr>
<tr>
<td></td>
<td>e1 e2 → e1 e3</td>
</tr>
</tbody>
</table>

- The second rule prohibits evaluating e2 except when e1 cannot be evaluated further
Call by Name

- Instead of the CBV strategy, we can specifically choose to perform beta-reduction before we evaluate the argument
- This is known as call-by-name (CBN)
  - This is the Lazy small-step approach

\[
\begin{align*}
\text{e1} \rightarrow \text{e3} \\
\text{e1 e2} \rightarrow \text{e3 e2} \\
(\lambda x.\text{e1}) \text{e2} \rightarrow \text{e1}[x:=\text{e2}] \\
\end{align*}
\]
CBN Reduction

- CBV
  - \((\lambda z.z) ((\lambda y.y) x) \rightarrow (\lambda z.z) x \rightarrow x\)

- CBN
  - \((\lambda z.z) ((\lambda y.y) x) \rightarrow (\lambda y.y) x \rightarrow x\)
Beta Reductions (CBN)

\[(\lambda x. x (\lambda y. y)) (u \, r) \rightarrow\]

\[(\lambda x. (\lambda w. x \, w)) (y \, z) \rightarrow\]
Beta Reductions (CBN)

\[(\lambda x. x (\lambda y. y)) \ (u \ r) \rightarrow (u \ r) \ (\lambda y. y)\]

\[(\lambda x. (\lambda w. x \ w)) \ (y \ z) \rightarrow (\lambda w. \ (y \ z) \ w)\]
Why Does This Matter?

- The rules we just showed are very common for programming languages based on LC
  - CBV is the most common (e.g. OCaml, Java)
  - CBN does come up (Haskell uses a variant known as “call-by-need”) but is much less common

- Interestingly: more programs terminated under call-by-name. Can you think of why?
  - Consider: \((\lambda x. e2) \ e1\),
  - What if \(e1\) would never terminate, but \(e2\) would?
Evaluating Within a Function

- Our original rules had evaluation under the lambda
- Where does this help us?

\[
\begin{align*}
e_1 \rightarrow e_2 \\
(\lambda x. e_1) \rightarrow (\lambda x. e_2)
\end{align*}
\]

\[
\begin{align*}
e_2 \rightarrow e_3 \\
e_1 e_2 \rightarrow e_1 e_3
\end{align*}
\]

\[
\begin{align*}
e_1 \rightarrow e_3 \\
e_1 e_2 \rightarrow e_3 e_2
\end{align*}
\]

\[
(\lambda x. e_1) e_2 \rightarrow e_1[x:=e_2]
\]
Partial Evaluation

- That rule is useful when you have a beta-reduction *under* a lambda:
  - \((\lambda y.(\lambda z.z) \ y \ x) \rightarrow (\lambda y.y \ x)\)

- Called *partial evaluation*
  - Can combine with CBN or CBV (just add in the rule)
  - In practical languages, this evaluation strategy is employed in a limited way, as compiler optimization

```plaintext
int foo(int x) {
    return 0+x;
}
```

```plaintext
int foo(int x) {
    return x;
}
```
Static Scoping & Alpha Conversion

- Lambda calculus uses static scoping

- Consider the following
  - $(\lambda x. x (\lambda x. x)) z \rightarrow ?$
    - The rightmost “x” refers to the second binding
  - This is a function that
    - Takes its argument and applies it to the identity function

- This function is “the same” as $(\lambda x. x (\lambda y. y))$
  - Renaming bound variables consistently preserves meaning
    - This is called alpha-renaming or alpha conversion
  - Ex. $\lambda x. x = \lambda y. y = \lambda z. z \quad \lambda y. \lambda x. y = \lambda z. \lambda x. z$
Quiz #5

Which of the following expressions is \textit{alpha equivalent} to \((\lambda x. \lambda y. x y) y\)

a) \(\lambda y. y y\)
b) \(\lambda z. y z\)
c) \((\lambda x. \lambda z. x z) y\)
d) \((\lambda x. \lambda y. x y) z\)
Quiz #5

Which of the following expressions is alpha equivalent to \((\lambda x. \lambda y. x y) y\)

a) \(\lambda y. y y\)
b) \(\lambda z. y z\)
c) \((\lambda x. \lambda z. x z) y\)
d) \((\lambda x. \lambda y. x y) z\)
Getting Serious about Substitution

- We have been thinking informally about substitution, but the details matter.
- So, let’s carefully formalize it, to help us see where it can get tricky!
Defining Substitution

- Use recursion on structure of terms
  - $x[x:=e] = e$  // Replace $x$ by $e$
  - $y[x:=e] = y$  // $y$ is different than $x$, so no effect
  - $(e_1 e_2)[x:=e] = (e_1[x:=e]) (e_2[x:=e])$
    // Substitute both parts of application
  - $(\lambda x. e')[x:=e] = \lambda x. e'$
    - In $\lambda x. e'$, the $x$ is a parameter, and thus a local variable that is different from other $x$'s. Implements static scoping.
    - So the substitution has no effect in this case, since the $x$ being substituted for is different from the parameter $x$ that is in $e'$
  - $(\lambda y. e')[x:=e] = ?$
    - The parameter $y$ does not share the same name as $x$, the variable being substituted for
    - Is $\lambda y.(e'[x:=e])$ correct? No…
Variable Capture

- How about the following?
  - \((\lambda x.\lambda y. x \; y) \; y \mapsto ?\)
  - When we replace \(y\) inside, we don’t want it to be captured by the inner binding of \(y\), as this violates static scoping
  - I.e., \((\lambda x.\lambda y. x \; y) \; y \neq \lambda y. y \; y\)

- Solution
  - \((\lambda x.\lambda y. x \; y)\) is “the same” as \((\lambda x.\lambda z. x \; z)\)
    - Due to alpha conversion
  - So alpha-convert \((\lambda x.\lambda y. x \; y) \; y\) to \((\lambda x.\lambda z. x \; z) \; y\) first
    - Now \((\lambda x.\lambda z. x \; z) \; y \mapsto \lambda z. y \; z\)
Completing the Definition of Substitution

- Recall: we need to define \((\lambda y.e')[x:=e]\)
  - We want to avoid capturing (free) occurrences of \(y\) in \(e\)
  - Solution: alpha-conversion!
    - Change \(y\) to a variable \(w\) that does not appear in \(e'\) or \(e\)
      (Such a \(w\) is called fresh)
    - Replace all occurrences of \(y\) in \(e'\) by \(w\).
    - Then replace all occurrences of \(x\) in \(e'\) by \(e\)!

- Formally:
  \[ (\lambda y.e')[x:=e] = \lambda w.((e' [y:=w]) [x:=e]) \] (\(w\) is fresh)
Beta-Reduction, Again

Whenever we do a step of beta reduction

- \((\lambda x. e_1) e_2 \rightarrow e_1[x:=e_2]\)
- We must alpha-convert variables as necessary
- Sometimes performed implicitly (w/o showing conversion)

Examples

- \((\lambda x. \lambda y. x \ y) \ y = (\lambda x. \lambda z. x \ z) \ y \rightarrow \lambda z. y \ z \quad // \ y \rightarrow \ z\)
- \((\lambda x. x \ (\lambda x. x)) \ z = (\lambda y. y \ (\lambda x. x)) \ z \rightarrow z \ (\lambda x. x) \quad // \ x \rightarrow \ y\)
Quiz #6

Beta-reducing the following term produces what result?

\[(\lambda x. x \lambda y. y \ x) \ y\]

A. \(y \ (\lambda z. z \ y)\)
B. \(z \ (\lambda y. y \ z)\)
C. \(y \ (\lambda y. y \ y)\)
D. \(y \ y\)
Quiz #6

Beta-reducing the following term produces what result?

\[(\lambda x. x \ \lambda y. y \ x) \ y\]

A. \(y \ (\lambda z. z \ y)\)
B. \(z \ (\lambda y. y \ z)\)
C. \(y \ (\lambda y. y \ y)\)
D. \(y \ y\)