More Lambda Calculus
and
Intro to Type Systems
Plan

• Heavy Class Participation
  - Thus, wake up! *(not actually kidding)*

• Lambda Calculus
  - How is it related to real life?
  - Encodings
  - Fixed points

• Type Systems
  - Overview
  - Static, Dynamic
  - Safety, Judgments, Derivations, Soundness
Lambda Review

• $\lambda$-calculus is a calculus of functions

\[ e := x | \lambda x. \ e | e_1 \ e_2 \]

• Several evaluation strategies exist based on $\beta$-reduction

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

• How does this simple calculus relate to real programming languages?
Functional Programming

• The $\lambda$-calculus is a prototypical functional language with:
  - no side effects
  - several evaluation strategies
  - lots of functions
  - nothing but functions (pure $\lambda$-calculus does not have any other data type)

• How can we program with functions?
• How can we program with only functions?
Programming With Functions

- **Functional programming** is a programming style that relies on lots of functions
- A typical functional paradigm is *using functions as arguments or results of other functions*
  - Called “**higher-order programming**”
- Some “impure” functional languages permit side-effects (e.g., Lisp, Scheme, ML, Python)
  - references (pointers), in-place update, arrays, exceptions
  - Others (and by “others” we mean “Haskell”) use monads to model state updates
Variables in Functional Languages

- We can introduce new variables:
  \[ \text{let } x = e_1 \text{ in } e_2 \]
  - \( x \) is bound by \( \text{let} \)
  - \( x \) is statically scoped in (a subset of) \( e_2 \)

- This is pretty much like \((\lambda x. e_2) e_1\)

- In a functional language, variables are never updated
  - they are just names for expressions or values
  - e.g., \( x \) is a name for the value denoted by \( e_1 \) in \( e_2 \)

- This models the meaning of “let” in math (proofs)
Referential Transparency

- In “pure” functional programs, we can reason equationally, by substitution
  - Called “referential transparency”
    
    \[
    \text{let } x = e_1 \text{ in } e_2 \quad \text{===} \quad [e_1/x]e_2
    \]

- In an imperative language a side-effect in $e_1$ might invalidate the above equation

- The behavior of a function in a “pure” functional language depends only on the actual arguments
  - Just like a function in math
  - This makes it easier to understand and to reason about functional programs
How Tough Is Lambda?

• Given $e_1$ and $e_2$, how complex (a la CS theory) is it to determine if:

\[ e_1 \rightarrow_\beta^* e \quad \text{and} \quad e_2 \rightarrow_\beta^* e \]
Expressiveness of $\lambda$-Calculus

- The $\lambda$-calculus is a minimal system but can express:
  - data types (integers, booleans, lists, trees, etc.)
  - branching
  - recursion
- This is enough to encode Turing machines:
  - We say the lambda calculus is Turing-complete
- Corollary: $e_1 =_\beta e_2$ is undecidable
- Still, how do we encode all these constructs using only functions?
- Idea: *encode the “behavior” of values* and not their structure
Encoding Booleans in $\lambda$-Calculus

- What can we *do* with a boolean?
  - we can make a binary choice (= “if” statement)

- A boolean is a function that, given two choices, selects one of them:

  - **true** = $\lambda x. \lambda y. x$
  - **false** = $\lambda x. \lambda y. y$
  - if $E_1$ then $E_2$ else $E_3$ = $E_1 \ E_2 \ E_3$

- Example: “if true then $u$ else $v$” is
  
  $$(\lambda x. \lambda y. x) \ u \ v \rightarrow_{\beta} (\lambda y. u) \ v \rightarrow_{\beta} u$$
More Boolean Encodings

• Let’s try to do boolean or together

• Recall:
  - true = \text{def} \lambda x. \lambda y. x
  - false = \text{def} \lambda x. \lambda y. y
  - if E_1 then E_2 else E_3 = \text{def} E_1 E_2 E_3

• We want or to take in two booleans and yield a result that is a boolean

• How can we do this?
A *Trying Ordeal*

**Recall:**
- **true** = \( \lambda x. \lambda y. x \)
- **false** = \( \lambda x. \lambda y. y \)
- if \( E_1 \) then \( E_2 \) else \( E_3 \) = \( E_1 \ E_2 \ E_3 \)

**Intuition:**
- **or** \( a \ b \) = if \( a \) then **true** else **false**

**Either of these will work:**
- **or** = \( \lambda a. \lambda b. a \ \text{true} \ b \)
- **or** = \( \lambda a. \lambda b. \lambda x. \lambda y. a \ x \ (b \ x \ y) \)
Final Boolean Encodings

- Think about how to do **and** and **not**
- Without peeking! Now come up and do it!
Another Demand

• How to do **and** and **not**
  
  • **and** a b
    - **and**
      = **def** \( \lambda a. \lambda b. a \ b \ \text{false} \)
    - **and**
      = **def** \( \lambda a. \lambda b. \lambda x. \lambda y. a \ (b \ x \ y) \  y \)

• **not** a
  - **not**
    = **def** \( \lambda a. \ a \ \text{false} \ \text{false} \ \text{true} \)
  - **not**
    = **def** \( \lambda a. \lambda x. \lambda y. a \ y \ x \)
Encoding Pairs in \( \lambda \)-Calculus

- What can we do with a pair?
  - we can access one of its elements (= “.field access”)

- A pair is a function that, given a boolean, returns the first or second element

  \[
  \text{mkpair } x \; y \; = \; \text{def} \; \lambda b. \; b \; x \; y \\
  \text{fst } p \; = \; \text{def} \; p \; \text{true} \\
  \text{snd } p \; = \; \text{def} \; p \; \text{false}
  \]

- \( \text{fst } (\text{mkpair } x \; y) \rightarrow_\beta (\text{mkpair } x \; y) \; \text{true} \)

  \( \rightarrow_\beta \; \text{true } x \; y \rightarrow_\beta \; x \)
Encoding Numbers in $\lambda$–Calculus

- What can we *do* with a natural number?
  - What do you, the viewers at home, think?
Encoding Numbers \( \lambda \)-Calculus

• What can we do with a natural number?
  - we can iterate a number of times over some function (= “for loop”)

• A natural number is a function that given an operation \( f \) and a starting value \( s \), applies \( f \) a number of times to \( s \):
  
  \[
  \begin{align*}
  0 &= \text{def } \lambda f. \lambda s. s \\
  1 &= \text{def } \lambda f. \lambda s. f s \\
  2 &= \text{def } \lambda f. \lambda s. f (f s)
  \end{align*}
  \]
  
  - Very similar to \texttt{List.fold_left} and friends

• These are numerals in a unary representation

• Called \texttt{Church numerals}
Test Time!

- How would you encode the **successor function** (succ x === x+1)?
- How would you encode more general **addition**?
- Recall: $4 =_{\text{def}} \lambda f. \lambda s. f f f (f s)$
Computing with Natural Numbers

- The successor function
  \[ \text{succ } n \quad = \quad \text{\texttt{def}} \quad \lambda f. \lambda s. f (n f s) \]
  or
  \[ \text{succ } n \quad = \quad \text{\texttt{def}} \quad \lambda f. \lambda s. n f (f s) \]

- Addition
  \[ \text{add } n_1 \; n_2 \quad = \quad \text{\texttt{def}} \quad n_1 \; \text{succ } n_2 \]

- Multiplication
  \[ \text{mult } n_1 \; n_2 \quad = \quad \text{\texttt{def}} \quad n_1 \; (\text{add } n_2) \; 0 \]

- Testing equality with 0
  \[ \text{iszero } n \quad = \quad \text{\texttt{def}} \quad n \; (\lambda b. \; \text{false}) \; \text{true} \]

- Subtraction
  - Is not instructive, but makes a fun exercise ...
Computation Example

• What is the result of the application `add 0`?

\[(\lambda n_1. \lambda n_2. \text{succ } n_2) \ 0 \rightarrow_\beta \]

\[\lambda n_2. \text{succ } n_2 = \]

\[\lambda n_2. (\lambda f. \lambda s. s) \text{ succ } n_2 \rightarrow_\beta \]

\[\lambda n_2. n_2 = \]

\[\lambda x. x \]

• By computing with functions we can express some optimizations
  - But we need to reduce under the lambda
  - Thus this “never” happens in practice
Toward Recursion

- Given a predicate \( P \), encode the function “\( \text{find} \)” such that “\( \text{find} \ P \ n \)” is the smallest natural number which is larger than \( n \) and satisfies \( P \).
- Claim: with \( \text{find} \) we can encode all recursion.

*Intuitively, why is this true?*
Encoding Recursion

- Given a predicate $P$ encode the function “find” such that “find $P$ n” is the smallest natural number which is larger than $n$ and satisfies $P$
- **find** satisfies the equation
  $$\text{find } p \ n = \text{if } p \ n \ \text{then } n \ \text{else } \text{find } p \ (\text{succ } n)$$
- Define
  $$F = \lambda f. \lambda p. \lambda n. (p \ n) \ n \ (f \ p \ (\text{succ } n))$$
- We need a **fixed point** of $F$
  $$\text{find} = F \ \text{find}$$
  or
  $$\text{find } p \ n = F \ \text{find } p \ n$$
The Fixed-Point Combinator $Y$

- Let $Y = \lambda F. (\lambda y. F(y y)) (\lambda x. F(x x))$
  - This is called the **fixed-point combinator**
  - Verify that $Y F$ is a fixed point of $F$
    
    $Y F \rightarrow_\beta (\lambda y. F(y y)) (\lambda x. F(x x)) \rightarrow_\beta F(Y F)$
  - Thus $Y F =_\beta F(Y F)$

- Given any function in $\lambda$-calculus we can compute its fixed-point (w00t! why do we not win here?)
- Thus we can define “find” as the fixed-point of the function $F$ from the previous slide
- Essence of recursion is the self-application “$y y$”
Expressiveness of Lambda Calculus

- Encodings are fun
  - Yes! Yes they are!
- But programming in pure \( \lambda \)-calculus is painful

- So we will add constants (0, 1, 2, ..., true, false, if-then-else, etc.)

- Next we will add \textit{types}
Still Going!

- One minute break
- Stretch!
In this 1943 Antoine de Saint-Exupery novel the title character lives on an asteroid with a rose but eventually travels to Earth.
Springfield, Illinois

- Animaniacs
- Two player game
- Players take turns naming State Capitals until one player says “Springfield, Illinois”
- The player to say “Springfield, Illinois” receives points equal to the number of previously-named Capitals
- Why is this game interesting?
Q: Computer Science (姚期智)

- This Shanghai-born Turing Award winner is known for contributions to the theory of computation. He formulated the Millionaire's Problem and stated this minimax principle: “the expected cost of any randomized algorithm for solving a given problem, on the worst case input for that algorithm, can be no better than the expected cost, for a worst-case random probability distribution on the inputs, of the deterministic algorithm that performs best against that distribution.”
Types

- A program variable can assume a range of values during the execution of a program.

- An upper bound of such a range is called a type of the variable.
  - A variable of type “bool” is supposed to assume only boolean values.
  - If x has type “bool” then the boolean expression “not(x)” has a sensible meaning during every run of the program.
Typed and Untyped Languages

- **Untyped languages**
  - Do *not* restrict the range of values for a given variable
  - Operations might be applied to inappropriate arguments. The behavior in such cases might be unspecified
  - The pure $\lambda$-calculus is an extreme case of an untyped language (however, its behavior is completely specified)

- **(Statically) Typed languages**
  - Variables are assigned (non-trivial) types
  - A type system keeps track of types
  - Types might or might not appear in the program itself
  - Languages can be explicitly typed or implicitly typed
The Purpose Of Types

- The foremost **purpose of types** is *to prevent certain types of run-time execution errors*

- Traditional trapped execution errors
  - Cause the computation to stop immediately
  - And are thus well-specified behavior
  - Usually enforced by hardware
  - e.g., Division by zero, floating point op with a NaN
  - e.g., Dereferencing the address 0 (on most systems)

- Untrapped execution errors
  - Behavior is **unspecified** (depends on the state of the machine = this is very bad!)
  - e.g., accessing past the end of an array
  - e.g., jumping to an address in the data segment
Execution Errors

• A program is deemed safe if it does not cause untrapped errors
  - Languages in which all programs are safe are safe languages

• For a given language we can designate a set of forbidden errors
  - A superset of the untrapped errors, usually including some trapped errors as well
    • e.g., null pointer dereference

• Modern Type System Powers:
  - prevent race conditions (e.g., Flanagan TLDI ‘05)
  - prevent insecure information flow (e.g., Li POPL ’05)
  - prevent resource leaks (e.g., Vault, Weimer)
  - help with generic programming, probabilistic languages, ...
  - ... are often combined with dynamic analyses (e.g., CCured)
Preventing Forbidden Errors - Static Checking

• Forbidden errors can be caught by a combination of static and run-time checking

• Static checking
  - Detects errors early, \textit{before testing}
  - Types provide the necessary static information for static checking
  - e.g., ML, Modula-3, Java
  - Detecting certain errors statically is \textit{undecidable} in most languages
Preventing Forbidden Errors - Dynamic Checking

- Required when static checking is undecidable
  - e.g., array-bounds checking
- Run-time encodings of types are still used (e.g. Lisp)
- Should be limited since it delays the manifestation of errors
- Can be done in hardware (e.g. null-pointer)
Safe Languages

- There are typed languages that are not safe ("weakly typed languages")
- All safe languages use types (static or dynamic)

<table>
<thead>
<tr>
<th></th>
<th>Typed</th>
<th></th>
<th>Untyped</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td></td>
<td>Dynamic</td>
<td></td>
</tr>
<tr>
<td>Safe</td>
<td>ML, Java, Ada, C#, Haskell, ...</td>
<td>Lisp, Scheme, Ruby, Perl, Smalltalk, PHP, Python, ...</td>
<td>λ-calculus</td>
</tr>
<tr>
<td>Unsafe</td>
<td>C, C++, Pascal, ...</td>
<td>?</td>
<td>Assembly</td>
</tr>
</tbody>
</table>

- We focus on statically typed languages
Why Typed Languages?

• Development
  - *Type checking catches early many mistakes*
  - Reduced debugging time
  - Typed signatures are a powerful basis for design
  - Typed signatures enable separate compilation

• Maintenance
  - Types act as checked specifications
  - Types can enforce abstraction

• Execution
  - Static checking reduces the need for dynamic checking
  - *Safe languages are easier to analyze statically*
    - the compiler can generate better code
Homework

- Read Cardelli article
- Read Wright & Matthias article
- Homework 5 Due Soon