More Lambda Calculus

Meeting 17, CSCI 5535, Spring 2009

Announcements

- Work on your project (probably background reading)
- I am looking at your proposals, but come talk to me if you have concerns

Plan

- Last Time (Introduce Lambda Calculus)
  - Syntax
  - Substitution
- Today (Lambda Calculus in "Real Life")
  - Operational Semantics
  - Evaluations strategies
  - Equality
  - Encodings
  - Fixed Points

Recall

Goal: Come up with a “core” language that’s as small as possible and still Turing complete

This will give a way of illustrating important language features and algorithms

Lambda Syntax

- The $\lambda$-calculus has 3 kinds of expressions (terms)
  - $\mathit{e} ::= \mathit{x}$ Variables
  - $\lambda \mathit{x}. \mathit{e}$ Functions (abstractions)
  - $\mathit{e}_1 \mathit{e}_2$ Application
- $\lambda \mathit{x}. \mathit{e}$ is a one-argument anonymous function with body $\mathit{e}$
- $\mathit{e}_1 \mathit{e}_2$ is a function application

Combinators

- A $\lambda$-term without free variables is closed or a combinator
- Some interesting combinators:
  - $\mathit{I} = \lambda \mathit{x}. \mathit{x}$
  - $\mathit{K} = \lambda \mathit{x}. \mathit{y}. \mathit{x}$
  - $\mathit{S} = \lambda \mathit{f}. \lambda \mathit{g}. \lambda \mathit{x}. \mathit{f} \mathit{x} (\mathit{g} \mathit{x})$
  - $\mathit{D} = \lambda \mathit{x}. \mathit{x} \mathit{x}$
  - $\mathit{Y} = \lambda \mathit{f}. (\lambda \mathit{x}. \mathit{f} (\mathit{x} \mathit{x})) (\lambda \mathit{x}. \mathit{f} (\mathit{x} \mathit{x}))$
- Theorem: any closed term is equivalent to one written with just $\mathit{S}, \mathit{K}$ and $\mathit{I}$
  - Example: $\mathit{D} \equiv \mathit{S} \mathit{S} \mathit{I}$
  - (we’ll discuss this form of equivalence later)

Explain these informally
Informal Semantics

• All we’ve got are functions, so all we can do is call them!

• The evaluation of \((\lambda \; x. \; e) \; e'\)
  - Binds \(x\) to \(e'\)
  - Evaluates \(e\) with the new binding
  - Yields the result of this evaluation

• Like a function call, or like "let \(x = e'\) in \(e\)"

• Example:
  \((\lambda \; f. \; f \; (f \; e)) \; g\) evaluates to   

Operational Semantics

• Many operational semantics for the \(\lambda\)-calculus

• All are based on the equation
  \((\lambda \; x. \; e_1) \; e_2 \rightarrow_\beta [e_2/x]e_1\)
  usually read from left to right

• This is called the \(\beta\)-rule and the evaluation step a \(\beta\)-reduction

• The subterm \((\lambda \; x. \; e_1) \; e_2\) is a \(\beta\)-redex

• We write \(e \rightarrow_\beta e'\) to say that \(e\) \(\beta\)-reduces to \(e'\) in one step

• We write \(e \rightarrow_\beta^* e'\) to say that \(e\) \(\beta\)-reduces to \(e'\) in 0 or more steps
  – Remind you of the small-step opsem term rewriting?

Examples of Evaluation

• The identity function:
  \((\lambda \; x. \; x) \; E \rightarrow [E/x] \; x = E\)

• Another example with the identity:
  \((\lambda \; f. \; f \; (\lambda \; x. \; x)) \; (\lambda \; x. \; x) \rightarrow\)
Examples of Evaluation

• The identity function:
  $$(\lambda x. x) E \rightarrow [E / x] x = E$$

• Another example with the identity:
  $$(\lambda f. f (\lambda x. x)) (\lambda x. x) \rightarrow$$
  $$[\lambda x. x / f] f (\lambda x. x) =$$
  $$[\lambda x. y / x] x = \lambda y. y$$

Examples of Evaluation

• A non-terminating evaluation:
  $$(\lambda x. xx) (\lambda y. yy) \rightarrow$$
  $$[\lambda y. yy / x] xx = (\lambda y. yy) (\lambda y. yy) \rightarrow ...$$

• Try T T, where $T = \lambda x. x x x$

Evaluation and the Static Scope

• The definition of substitution guarantees that evaluation respects static scoping:
  $$(\lambda x. (\lambda y. y x)) (y (\lambda x. x)) \rightarrow^\beta \lambda y. z (y (\lambda v. v))$$
  ($y$ remains free, i.e., defined externally)

• If we forget to rename the bound $y$:
  $$(\lambda x. (\lambda y. y x)) (y (\lambda x. x)) \rightarrow^\beta * \lambda y. y (y (\lambda v. v))$$
  ($y$ was free before but is bound now)

Another View of Reduction

• The application

Normal Forms

• A term without redexes is in normal form
• A reduction sequence stops at a normal form

• If $e$ is in normal form and $e \rightarrow^\beta * e'$ then $e$ is identical to $e'$

• $K = \lambda x. y. x$ is in normal form
• $KI$ is not in normal form

Structural Operational Semantics

• We define a small-step reduction relation

• This is a non-deterministic semantics
• Note that we evaluate under $\lambda$ (where?)
Lambda Calculus Contexts

- Define contexts with one hole
  \[ H ::= \hole | \lambda x. H | H e | e H \]

- Write \[ H[e] \] to denote the filling of the hole in \( H \) with the expression \( e \)
- Example:
  - \[ H = \lambda x. x \] 
  - \[ H[\lambda y. y] = \lambda x. x (\lambda y. y) \]
  - Filling the hole allows variable capture!

- Example:
  - \[ H = \lambda x. x \] 
  - \[ H[x] = \lambda x. x x \]

Contextual Operational Semantics

- Contexts allow concise formulations of congruence rules (application of local reduction rules on subterms)
- Reduction occurs at a \( \beta \)-redex that can be anywhere inside the expression
- The latter rule is called a congruence or structural rule
- The above rules to not specify which redex must be reduced first

- \( (\lambda x. e_1) e_2 \rightarrow [e_2/x] e_1 \)
  - \( H[e] \rightarrow H[e'] \)

The Order of Evaluation

- In a \( \lambda \)-term there could be more than one instance of \( (\lambda x. e_1) e_2 \), as in:
  \[ (\lambda y. (\lambda x. x) y) E \]
  - Could reduce the inner or outer \( \lambda \)
  - Which one should we pick?
  \[ (\lambda y. (\lambda x. x) y) E \]

- The Diamond Property

  - A relation \( R \) has the diamond property if whenever \( e R e_1 \) and \( e R e_2 \) then there exists \( e' \) such that \( e_1 R e' \) and \( e_2 R e' \)
  - \( \rightarrow_\beta \) does not have the diamond property
  - \( \rightarrow_\beta^* \) has the diamond property (Church-Rosser property, confluence property)
    - simplest known proof is quite technical
A Diamond In The Rough

- Languages defined by non-deterministic sets of rules are common
  - Logic programming languages
  - Expert systems
  - Constraint satisfaction systems
- And thus most pointer analyses...
  - Dataflow systems
  - Makefiles
- It is useful to know whether such systems have the diamond property

Beta Equality

- Let $\beta$ be the reflexive, transitive and symmetric closure of $\rightarrow_\beta$
  $\beta$ is $(\rightarrow_\beta \cup \leftarrow_\beta)^*$
- That is, $e \beta e'$ if $e$ converts to $e'$ via a sequence of forward and backward $\rightarrow_\beta$

The Church-Rosser Theorem

- If $e_1 \beta e_2$ then there exists $e'$ such that
  $e_1 \rightarrow_\beta e'$ and $e_2 \rightarrow_\beta e'$

- Proof (sketch): apply the diamond property as many times as necessary

Corollaries

- If $e_1 \beta e_2$ and $e_1$ and $e_2$ are normal forms then $e_1$ is identical to $e_2$
  - From CR we have $\exists e'. e_1 \rightarrow_\beta e'$ and $e_2 \rightarrow_\beta e'$
  - Since $e_1$ and $e_2$ are normal forms they are identical to $e'$
- If $e \rightarrow_\beta e_1$ and $e \rightarrow_\beta e_2$ and $e_1$ and $e_2$ are normal forms then $e_1$ is identical to $e_2$
  - “All terms have a unique normal form.”

Evaluation Strategies

- Church-Rosser says that independent of the reduction strategy we will find $\leq 1$ normal form
- But some reduction strategies might find 0
  - $(x. z)((y. y y)(y. y y)) \rightarrow$
    $((y. y y)(y. y y)) \rightarrow$
  - $(x. z)((y. y y)(y. y y)) \rightarrow z$
- There are three traditional strategies
  - normal order (never used, always works)
  - call-by-name (rarely used, cf. TeX)
  - call-by-value (amazingly popular)

Evaluation Strategies Summary

- Normal Order
  - Evaluates the left-most redex not contained in another redex
  - If there is a normal form, this finds it
  - Not used in practice: requires partially evaluating function pointers and looking “inside” functions
- Call-By-Name (“lazy”)
  - Don’t reduce under $\lambda$, don’t evaluate a function argument (until you need to)
  - Does not always evaluate to a normal form
- Call-By-Value (“eager” or “strict”)
  - Don’t reduce under $\lambda$, always evaluate a function argument right away
  - Finds normal forms less often than the other two
Bonus: Evaluation Strategies

Normal-Order Reduction

- A redex is **outermost** if it is not contained inside another redex
- Example:
  \[ S (K x y) (K u v) \]
  
  \[ K x, K u \text{ and } S (K x y) \text{ are all redexes} \]
  
  Both \( K u \) and \( S (K x y) \) are outermost
  
  Normal order always reduces the **leftmost outermost** redex first
  
  Theorem: If \( e \) has a normal form \( e' \) then normal order reduction will reduce \( e \) to \( e' \)

Why Not Normal Order?

- In most (all?) programming languages, functions are considered values (fully evaluated)
- Example:
  \[ \lambda x. D \ D = \bot \] (with normal order)
  
  Thus, no reduction is done under lambda
  
  No popular programming language uses normal order

Call-by-Name

- Don’t reduce under \( \lambda \)
  
  Don’t evaluate the argument to a function call
  
  A value is an abstraction
  
  \[ \frac{e_1 \rightarrow^* \lambda x. e_2}{\lambda x. e_1 \rightarrow^* e_2} \]
  
  Call-by-name is demand-driven: an expression is not evaluated unless needed
  
  It is normalizing: converges whenever normal order converges
  
  Call-by-name does not necessarily evaluate to a normal form. Example: \( D \ D \)

Call-by-Value Evaluation

- Don’t reduce under \( \lambda \)
  
  Do evaluate the arguments to a function call
  
  A value is an abstraction
  
  \[ \frac{e_1 \rightarrow^* \lambda x. e_2 \rightarrow^* [e_2/x]e_1 \rightarrow^* e}{\lambda x. e_1 \rightarrow^* e} \]
  
  Most languages are primarily call-by-value
  
  But CBV is not normalizing: \( (\lambda x. I) (D D) \)
  
  CBV diverges more often than normal order or even CBN
Call by Value Example

- Example:

\[(\lambda y. (\lambda x. x) y) ((\lambda u. u) (\lambda v. v)) \rightarrow_{\beta} (\lambda v. v)\]

Evaluation Strategy Considerations

- Call-by-value:
  - easy to implement
  - well-behaved (predictable) with respect to side-effects
- Call-by-name:
  - More difficult to implement (must pass unevaluated expressions)
  - The order of evaluation is harder to predict (e.g., difficulty with side-effects)
  - Has a simpler theory than call-by-value
  - Allows the natural expression of infinite data structures (e.g., streams)
  - Terminates more often than call-by-value

Which is Better?

- The debate about whether languages should be strict (CBV) or lazy (CBN) is nearly 20 years old
- This debate is typically confined to the functional programming community (where it is sometimes intense)
- Outside the functional community CBN is rarely considered (but remember TeX)

Caveats

- The terms lazy and strict are not used consistently in the literature
- Call-by-value and call-by-name are well defined
- There are parameter passing mechanisms besides call-by-value and call-by-name that cannot be naturally expressed in the lambda calculus
  - by reference
  - value-result
- These additional mechanisms deal with side-effects
- Many languages have a mixture of parameter passing mechanisms

How is \(\lambda\)-calculus related to “real life”? 

- 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

Referential Transparency

• In “pure” functional programs, we can reason equationally, by substitution
  - Called “referential transparency”
  
  \[
  \text{let } x = e_1 \text{ in } e_2 \equiv \left( e_1/x \right) e_2
  \]
• In an imperative language a side-effect in \( e_1 \) might invalidate the above equation.
  - Why?
• 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 Complex 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: \( e_1 \rightarrow e_2 \) is undecidable
• That means we can encode any computation we want in it ... if we’re sufficiently clever ...

Encodings

• 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” exp)
• A boolean is a function that, given two choices, selects one of them:
  - true \( \equiv_{\text{def}} \lambda x. \lambda y. x \)
  - false \( \equiv_{\text{def}} \lambda x. \lambda y. y \)
  - if \( E_1 \) then \( E_2 \) else \( E_3 \) \( \equiv_{\text{def}} \)
Encoding Booleans in \(\lambda\)-Calculus

- What can we do with a boolean?
  - we can make a binary choice (= "if" exp)
- A boolean is a function that, given two choices, selects one of them:
  - \(\text{true} = \lambda x. \lambda y. x\)
  - \(\text{false} = \lambda x. \lambda y. y\)
- \(\text{if } E_1 \text{ then } E_2 \text{ else } E_3 = E_1 \text{ or } E_3\)
- Example: "if true then \(u\) else \(v\)" is \((\lambda x. \lambda y. x) u v \rightarrow \beta (\lambda y. u) v \rightarrow \beta u\)

Let's try to define or

- Recall:
  - \(\text{true} = \lambda x. \lambda y. x\)
  - \(\text{false} = \lambda x. \lambda y. y\)
  - \(\text{if } E_1 \text{ then } E_2 \text{ else } E_3 = E_1 \text{ or } E_3\)
- Intuition:
  - \(\text{or } a \text{ b = if } a \text{ then true else } b\)
- Either of these will work:
  - \(\text{or} = \lambda a. \lambda b. a \text{ true } b\)
  - \(\text{or} = \lambda a. \lambda b. \lambda x. \lambda y. a \times (b \times y)\)

This is getting painful to check ...

- Let's use OCaml ...

More Boolean Encodings

- Think about how to do \(\text{and}\) and \(\text{not}\)
- Without peeking!

Encoding and and not

- \(\text{and} \ a \ b = \text{if } a \text{ then } b \text{ else } \text{false}\)
  - \(\text{and} = \lambda a. \lambda b. a \text{ and } b\)
  - \(\text{and} = \lambda a. \lambda b. \lambda x. \lambda y. (b \times y)\)
- \(\text{not} \ a = \text{if } a \text{ then false else true}\)
  - \(\text{not} = \lambda a. \text{ a not } \text{ true}\)
  - \(\text{not} = \lambda a. \lambda x. \lambda y. a \times x\)

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:
  - \(\text{mkpair} \times y = \lambda b. b \times y\)
  - \(\text{fst} \times p = \text{true}\)
  - \(\text{snd} \times p = \text{false}\)
  - \(\text{fst} (\text{mkpair} \times y) \rightarrow \beta (\lambda b. b \times y) \rightarrow \beta (\text{true} \times y) \rightarrow \beta \text{true} \times y\)
  - \(\text{snd} (\text{mkpair} \times y) \rightarrow \beta (\lambda b. b \times y) \rightarrow \beta \text{false} \times y\)
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 \(z\), applies \(f\) a number of times to \(z\):
  
  \[
  0 = \text{def} \lambda f. \lambda z. z \\
  1 = \text{def} \lambda f. \lambda z. f z \\
  2 = \text{def} \lambda f. \lambda z. f (f z)
  
  - Very similar to List.fold_left and friends
- These are numerals in a unary representation
- Called Church numerals

Computing with Natural Numbers

- The successor function
  
  \[
  \text{succ } n = \text{def} \lambda f. \lambda s. f (n f s)
  \]
  or
  
  \[
  \text{succ } n = \text{def} \lambda f. \lambda s. n f (f s)
  \]
- Addition
  
  \[
  \text{plus } n_1 \ n_2 = \text{def} \ n_1 \ \text{succ } n_2
  \]
- Multiplication
  
  \[
  \text{mult } n_1 \ n_2 = \text{def} \ n_1 \ \text{(add } n_2 \ 0)
  \]
- Testing equality with 0
  
  \[
  \text{iszero } n = \text{def} \ n \ (\lambda b. \text{false}) \ \text{true}
  \]
- Subtraction
  
  Is not instructive, but makes a fun exercise …

Computation Example

- What is the result of the application \(\text{add} \ 0\)?
  
  \[
  (\lambda n_1. \lambda n_2. n_1 \ \text{succ } n_2) \ 0 \rightarrow \beta
  \]

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

Encoding Recursion

- Given a predicate \(p\), encode the function “find” such that “find \(p\ n\)” is the smallest natural number which is at least \(n\) and satisfies \(p\)
- find satisfies the equation
  
  \[
  \text{find } \ n = \text{if } \ p \ \text{n then n else find } (\text{succ } n)
  \]
- Define
  
  \[
  F = \lambda f. \lambda p. \lambda n. (p \ n) \ (f \ (\text{succ } n))
  \]
- We need a fixed point of \(F\)
  
  \[
  \text{find } = F \ \text{find}
  \]
  or
  
  \[
  \text{find } \ n = F \ \text{find } \ n
  \]

Toward Recursion

- Given a predicate \(p\), encode the function “find” such that “find \(p\ n\)” is the smallest natural number which is at least \(n\) and satisfies \(p\)
- Ideas? How do we begin?

The Fixed-Point Combinator \(Y\)

- Let \(Y = \lambda F. (\lambda y. F (y \ y)) (\lambda x. F (x \ x))\)
  
  - This is called the \text{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 \(Y\ F\ = F (Y\ F)\)
- Given any function in \(\lambda\)-calculus we can compute its fixed-point (!)
  
  - 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 types