Lambda Calculus in “Real Life”

Meeting 17, CSCI 5535, Spring 2009

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.

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

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.

Referential Transparency

- In "pure" functional programs, we can reason equationally, by substitution.
  - Called "referential transparency".
    - $\text{let } x = e_1 \text{ in } e_2 \iff [x/e_1]e_2$
- 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:
  $$e_1 \rightarrow_{\beta}^* e \text{ and } e_2 \rightarrow_{\beta}^* e$$

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: $e_1 \equiv^* e_2$ 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 $= \text{def} \lambda x. \lambda y. x$
  - false $= \text{def} \lambda x. \lambda y. y$
  - if $E_1$ then $E_2$ else $E_3$ $= \text{def} E_1 \ E_2 \ E_3$
  - Example: "if true then u else v" is
    $$(\lambda x. \lambda y. x) \ u \rightarrow_{\beta} (\lambda y. u) \ v \rightarrow_{\beta} u$$

Let's try to define or

- Recall:
  - true $= \text{def} \lambda x. \lambda y. x$
  - false $= \text{def} \lambda x. \lambda y. y$
  - if $E_1$ then $E_2$ else $E_3$ $= \text{def} E_1 \ E_2 \ E_3$
- Intuition:
  - or $a \ b = \text{if } a \text{ then true else } b$
- Either of these will work:
  - or $= \text{def} \lambda a. \lambda b. \lambda x. \lambda y. a \ x \ (b \ y)$
This is getting painful to check …

• Let’s use OCaml …

More Boolean Encodings

• Think about how to do and and not
• Without peeking!

Encoding and and not

• and a b = if a then b else false
  - and =def λa. λb. a b false
  - and =def λa. λb. λx. λy. a (b x y) y

• not a = if a then false else true
  - not =def λa. a false true
  - not =def λa. λx. λy. a y x

Encoding Pairs in λ-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
  mkpair x y =def λb. b x y
  fst p =def p true
  snd p =def p false
• fst (mkpair x y) →β (mkpair x y) true
  →β true x y

Encoding Numbers λ-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 =def λf. λz. z
  1 =def λf. λz. f z
  2 =def λf. λz. 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
  succ n =def λf. λs. f (n f s)
  or succ n =def λf. λs. λf. n (s f)
• Addition
  plus n1 n2 =def n1 succ n2
• Multiplication
  mult n1 n2 =def n1 (add n2) 0
• Testing equality with 0
  iszero n =def n (λb. false) true
• Subtraction
  - Is not instructive, but makes a fun exercise …
Computation Example

- What is the result of the application \(\text{add } 0\)?

\[
\begin{align*}
&\lambda n_1.\lambda n_2.\ n_1\ \text{succ}\ n_2\ 0 \rightarrow^\beta \\
&\lambda n_2.\ 0\ \text{succ}\ n_2 \rightarrow^\beta \\
&\lambda n_2.\ (\lambda f.\ \lambda s.\ s)\ \text{succ}\ n_2 \rightarrow^\beta \\
&\lambda n_2.\ n_2 \rightarrow^\beta \\
&\lambda x.\ x
\end{align*}
\]

- 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 "find" such that "find \(p\ n\)" is the smallest natural number which is at least \(n\) and satisfies \(p\)

- Ideas? How do we begin?

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

\[
\text{find } p\ n = \text{if } p\ n\ \text{then } n\ \text{else find } p\ (\text{succ } n)
\]

- 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 \(Y\ F \approx F\ (Y\ F)\)

- 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