CMSC 330: Organization of Programming Languages

Lambda Calculus Introduction

Introduction

• We’ve seen that several language conveniences aren’t strictly necessary
  – Multi-argument functions: use currying or tuples
  – Loops: use recursion
  – Side-effects: we don't need them either

• 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
Revised Rule for Application

\[
\begin{align*}
A; E_1 &\rightarrow (A', \lambda x.E) & A; E_2 &\rightarrow v \\
\hline
A, A', x:v; E &\rightarrow v' \\
A; (E_1 \ E_2) &\rightarrow v'
\end{align*}
\]

- To apply something to an argument:
  - Evaluate it to produce a closure
  - Evaluate the argument (call-by-value)
  - Evaluate the body of the closure, in
    - The current environment, extended with the closure’s
      environment, extended with the binding for the parameter

Example

\[
\begin{align*}
\cdot; (\text{fun } x = (\text{fun } y = + x y)) &\rightarrow (\cdot, \lambda x.(\text{fun } y = + x y)) \\
\cdot; 3 &\rightarrow 3 \\
x:3; (\text{fun } y = + x y) &\rightarrow (x:3, \lambda y.(+ x y)) \\
\cdot; (\text{fun } x = (\text{fun } y = + x y)) \ 3 &\rightarrow (x:3, \lambda y.(+ x y))
\end{align*}
\]
Lambda Calculus

- A lambda calculus expression is defined as

\[ e ::= x \quad \text{variable} \]
\[ \mid \lambda x . e \quad \text{function} \]
\[ \mid e \; e \quad \text{function application} \]

- \( \lambda x . e \) is like \((\text{fun } x \rightarrow e)\) in OCaml

- That’s it! Only higher-order functions

Intuitive Understanding

- Before we work more with the mathematical notation of lambda calculus, we’re going to play a puzzle game!

- From: http://worrydream.com/AlligatorEggs/
Puzzle Pieces

- Hungry alligators: eat and guard family
- Old alligators: guard family
- Eggs: hatch into new family

Example Families

- Families are shown in columns
- Alligators guard families below them
Puzzle Rule 1: The Eating Rule

- If families are side-by-side the top left alligator eats the entire family to her right
- The top left alligator dies
- Any eggs she was guarding of the same color hatch into what she just ate

Eating Rule Practice

- What happens to these alligators?
  Puzzle 1:  Puzzle 2:

Answer 1:  Answer 2:
Puzzle Rule 2: The Color Rule

- If an alligator is about to eat a family and a color appears in both families then we need to change that color in one of the families.

- If a color appears in both families, but only as an egg, no color change is made.

Puzzle Rule 3: The Old Alligator Rule

- When an old alligator is only guarding one family it dies.
Challenging Puzzles!

- Try to reduce these groups of alligators as much as possible using the three puzzle rules:

- Challenge your neighbors with puzzles of your own.

More Puzzles

- When Family Not eats Family True it becomes Family False and when Not eats False it becomes True… what color should the white eggs be?

- What do the AND and OR families look like?
Lambda Calculus

- A lambda calculus expression is defined as

  \[ e ::= x \text{ variable} \quad (e: \text{egg}) \]

  \[ | \lambda x.e \text{ function} \quad (\lambda x: \text{alligator}) \]

  \[ | e e \text{ function application} \quad \text{(adjacency of families)} \]

- \( \lambda x.e \) is like \( \text{fun x} \rightarrow e \) in OCaml

- That’s it! Only higher-order functions

Three Conveniences

- Syntactic sugar for local declarations
  - \text{let } x = e1 \text{ in } e2 \text{ is short for } (\lambda x.e2) e1

- The scope of \( \lambda \) extends as far to the right as possible
  - \( \lambda x. \lambda y.x \, y \text{ is } \lambda x.(\lambda y.(x \, y)) \)

- Function application is left-associative
  - \( x \, y \, z \text{ is } (x \, y) \, z \)
  - Same rule as OCaml
Operational Semantics

- All we’ve got are functions, so all we can do is call them
- To evaluate \((\lambda x. e_1) \ e_2\)
  - Evaluate \(e_1\) with \(x\) bound to \(e_2\)
- This application is called “beta-reduction”
  - \((\lambda x. e_1) \ e_2 \rightarrow e_1[x/e_2]\) (the eating rule)
    - \(e_1[x/e_2]\) is \(e_1\) where occurrences of \(x\) are replaced by \(e_2\)
    - Slightly different than the environments we saw for Ocaml
      - Do substitutions to replace formals with actuals, instead of carrying around environment that maps formals to actuals
      - We allow reductions to occur anywhere in a term

Examples  (try with alligators too)

- \((\lambda x. z) \ z \rightarrow z\)
- \((\lambda x. y) \ z \rightarrow y\)
- \((\lambda x. y) \ z \rightarrow zy\)
  - A function that applies its argument to \(y\)
- \((\lambda x. y) \ (\lambda z. z) \rightarrow (\lambda z. z) \ y \rightarrow y\)
- \((\lambda x. \lambda y. x \ y) \ z \rightarrow \lambda y. z \ y\)
  - A curried function of two arguments that applies its first argument to its second
- \((\lambda x. \lambda y. x \ y) \ (\lambda z. z) \ x \rightarrow \lambda y. ((\lambda z. z) y) x \rightarrow (\lambda z. z) x \rightarrow xx\)
Static Scoping and Alpha Conversion

- Lambda calculus uses static scoping

- Consider the following
  - \((\lambda x. (\lambda x.) x) \ z \rightarrow ?\)
    - 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 \((\lambda x. (\lambda y.) y)\)
  - Renaming bound variables consistently is allowed
    - This is called alpha-renaming or alpha conversion (color rule)
  - Ex. \(\lambda x. x = \lambda y. y = \lambda z. z\)

Static Scoping (cont’d)

- How about the following?
  - \((\lambda x. (\lambda y.) y \ x) \ y \rightarrow ?\)
    - When we replace y inside, we don’t want it to be “captured” by the inner binding of y

- This function is “the same” as \((\lambda x. (\lambda z.) z)\)