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
Lambda Calculus

• A lambda calculus expression is defined as

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

• \( \lambda x. e \) is like \( \texttt{(fun x -> e)} \) 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:
Eating Rule Practice

- What happens to these alligators?

**Puzzle 1:**

**Answer 1:**

**Puzzle 2:**

**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.
  – In the picture below, green and red appear in both the first and second families. So, in the second family, we switch all of the greens to cyan, and all of the reds to blue.

• If a color appears in both families, but *only* as an egg, no color change is made.
Puzzle Rule 3: The Old Alligator Rule

• An old alligator that is guarding only one family 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.
Another Puzzle
YAP (Yet Another Puzzle)
Naming Families

• When Family **Not** eats Family **True** it becomes Family **False** and when **Not** eats **False** it becomes **True**
Lambda Calculus

• A lambda calculus expression is defined as

\[ e ::= x \quad \text{variable} \quad (e: \text{egg}) \]
\[ \mid \lambda x. e \quad \text{function} \quad (\lambda x: \text{alligator}) \]
\[ \mid e \; e \quad \text{function application} \quad (\text{adjacency of families}) \]

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

• That’s it! Only higher-order functions
Three Conveniences

• Syntactic sugar for local declarations
  – let x = e1 in e2 is short for (λx.e2) e1

• The scope of λ extends as far to the right as possible
  – λx.λy.x y is λx.(λy.(x y))

• Function application is left-associative
  – x y z 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”
  – \((\lambda x.e_1) \ e_2 \rightarrow e_1[x/e_2] (\text{the eating rule})\)
    • \(e_1[x/e_2]\) is \(e_1\) where occurrences of \(x\) are replaced by \(e_2\)
    • Slightly different than the environments we saw for Ocaml
      – Substitutions instead of environments
Examples

• \((\lambda x.x) \ z \rightarrow z\)
• \((\lambda x.y) \ z \rightarrow y\)
• \((\lambda x.x \ y) \ z \rightarrow z \ y\)
  – A function that applies its argument to \(y\)
• \((\lambda x.x \ y) \ (\lambda z.z) \rightarrow (\lambda z.z) \ y \rightarrow y\)
• \((\lambda x.\lambda y.x \ y) \ z \rightarrow \lambda y.z \ y\)
  – A curried function of two arguments that applies its first argument to its second
• \((\lambda x.\lambda y.x \ y) \ (\lambda z.z \ z) \ x \rightarrow\)
  \[\lambda y.((\lambda z.z \ z)y)x \rightarrow (\lambda z.z \ z)x \rightarrow x \ x\]
Static Scoping and 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 is allowed
    • This is called alpha-renaming or alpha conversion (color rule)
    – Ex. \(\lambda x.x = \lambda y.y = \lambda z.z\) \hspace{1cm} \lambda y.\lambda x.y = \lambda z.\lambda x.z\)
Static Scoping (cont’d)

• How about the following?
  – \((\lambda x. \lambda y. x \ y) \ y \rightarrow ?\)
  – 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 \((\lambda x. \lambda z. x \ z)\)