CSC312 Principles of Programming Languages :

Lambda Calculus in Further Depth

Lambda Calculus

A clean, concise way to express a function.

Example:

A square function expressed in Python:

definitions:

```
def squareFunction (x):
        y = x * x;
        return y;
```

invocation:

```
squareFunction (100)
```

The same function expressed in Lambda Calculus:

definitions:

$\lambda x . x * x$

specify the formal parameter (λx)

specify the function body ($x * x$)

invocation:

$(\lambda x . x * x) \, 100$

Lambda Calculus

- There are many programming languages we could talk about
- But pretty much all real languages are complex, large and obscure many important issues in irrelevant details
- We want: "as simple as possible" language to study properties of programming languages
- This language is known as lambda calculus

Definition of Lambda Calculus

- *Principle Components of a programming language?*
- *Syntax*
- *Semantics*

Lambda Calculus Syntax

- There are only four expressions in lambda calcus:
- Expression 1: constants
  - 1, 7, "yourName" are all valid expressions in lambda calculus
- Expression 2: identifiers
  - Will usually use x, y, etc for those
- Expression 3: lambda abstraction
  - written as $\lambda x.e$
- Expression 4: application
  - written as $e_1 \; e_2$

Lambda Calculus Syntax

- Or, more concisely, the syntax of a lambda calculus expression as context-free grammar is given by:

$$e = c \mid \mathsf{id} \mid \lambda \mathsf{id}.e \mid e_1 \; e_2$$

With it, we can now check whether an expression is a lambda calculus.

How? What do we need to check?
(Think of your Grammar project.)

Example:

- Consider the expression: $A = (\lambda x.x)\ 3$
- Now, recalling the syntax

$$e = c \mid \text{id} \mid \lambda\text{id}.e \mid e_1\ e_2$$

we can give a derivation proving that A is valid

- $e \rightarrow e_1\ e_2 \rightarrow e_1\ 3 \rightarrow (\lambda x.e)\ 3 \rightarrow (\lambda x.x)\ 3$
- Any expression for which we can find a derivation is syntactically valid lambda calculus

Are we done?

- We can now decide if any string is lambda calculus

Lambda calculus semantics

- Let's define the meaning for each expression in our production:
  - Constant c: The meaning of c is the value of c
  - Identifier id: The meaning of id is id
  - Lambda $\lambda x.e$: The meaning: $\lambda x.e$
  - Application $\lambda x.e\ e_2$: The meaning: $e[e_2/x]$
- $e[e_2/x]$ is substitution. We replace all free occurrences of x by e_2 in expression e
- An occurrence of a variable is free if it is not bound by a λ
  Example: $(\lambda x.x)[2/x] = \lambda x.x$
- Upshot: We can define anonymous functions with binding operator λ.

Examples

- Meaning (or value) of $(\lambda x.x)$ 1?
- $(\lambda x.x)\ 1 \rightarrow x[1/x] \rightarrow 1$
- $(\lambda x.(\lambda x.x)x)1 \rightarrow ((\lambda x.x)x)[1/x] \rightarrow (\lambda x.x)1 \rightarrow ...$
- Substitution is capture-avoiding: Does not replace variables bound by other λ's
- Convention: We assume that λ-bindings extend as far to the right as possible
- We read $\lambda x.\lambda y.xy$ as $(\lambda x.(\lambda y.xy))$ But use parenthesis to be safe

More Examples

- To make lambda calculus slightly more interesting, we will also allow arithmetic operators with their usual meaning.
- We could give them precise semantics, but too boring. We all know their semantics
- $(\lambda x.5 * x)\ 1 \rightarrow (5 * x)[1/x] \rightarrow (5 * 1) \rightarrow 5$
- $(\lambda x.\lambda y.x + y)\ 3\ 5 \rightarrow ((\lambda y.x + y)[3/x])\ 5 \rightarrow (\lambda y.3 + y)\ 5 \rightarrow (3 + y)[5/y] \rightarrow (3 + 5) \rightarrow 8$

Quick Review: Lambda Calculus

- *Syntax*
- *Semantics*

$$
\begin{aligned}
&x[x \leftarrow y] = y \\
&(xx)[x \leftarrow y] = (yy) \\
&(zw)[x \leftarrow y] = (zw) \\
&(zx)[x \leftarrow y] = (zy) \\
&(\lambda x \cdot (zx))[x \leftarrow y] = (\lambda u \cdot (zu))[x \leftarrow y] = (\lambda u \cdot (zu)) \\
&(\lambda x \cdot (zx))[y \leftarrow x] = (\lambda u \cdot (zu))[y \leftarrow x] = (\lambda u \cdot (zu))
\end{aligned}
$$

Properties of lambda expressions

- We have seen that to compute the value of lambda expressions, we only needed to define application: $\lambda x.e\ e_2$ as $e[e_2/x]$
- In lambda calculus, this is called β-reduction.
- Confluence: Order of reductions is provably irrelevant
- Other property of lambda expressions: $\lambda x.e \Leftrightarrow \lambda y.(e[y/x])$
- This is called $\alpha-$reduction
- Simply encodes that the name of lambda bound variables is irrelevant
- Analogy: $\int_0^\infty e^{-x}\, \mathrm{d}x \equiv \int_0^\infty e^{-y}\, \mathrm{d}y$

Expression Equivalence

- Using $\alpha-$ and $\beta-$reductions, we can prove equivalence of expressions by computing their values using $\beta-$reduction and (if necessary) applying $\alpha-$reductions.
- Example: $e_1 = (\lambda x.x + 1)$ and $e_2 = (\lambda z.z + 1)$.
- Using $\alpha-$reduction, we can rewrite
  $e_1' = (\lambda x.x + 1) \rightarrow^{\alpha} (\lambda z.z + 1)$
- Have now proven that e_1 and e_2 are equivalent

Is Lambda Calculus expressive enough as a programming language?

- Lambda calculus looks very far from a real programming language.
- On the face of it, many features missing.
  - Multi-argument functions
  - Declarations
  - Conditionals
  - Named Functions
  - Recursion
  - ...
- Next: How to express these features in basic lambda calculus

Multi-argument functions

- How can we express adding two numbers?
- Recall earlier example: $(\lambda x.\lambda y.x + y)3\ 5$
- Here, we first reduce to $(\lambda x.\lambda y.x + y)\ 3\ 5 \rightarrow ((\lambda y.x + y)[3/x])\ 5 \rightarrow (\lambda y.3 + y)\ 5$
- In other words, we partially evaluate λx, resulting in a new function $(\lambda y.3 + y)$.
- This is equivalent to having a λ-binding with multiple arguments
- This is known as Currying

Declarations

- We want to be able to give names to subexpressions
- Equivalence in typical programming languages: Local declarations
- Specifically, we want to add a let-construct of the following form to lambda calculus
- let $x = e_1$ in e_2
- Insight: Can define meaning of let-construct in in terms of basic lambda calculus: *How?*

Declarations

- One possibility: let $x = e_1$ in e_2 means $e_2[e_1/x]$
- Or equivalently: let $x = e_1$ in e_2 means $(\lambda x.e_2)e_1$
- Why are these definitions equivalent?

Conditionals

- Conditional: if x then e_1 else e_2

x e1 e2

that is,

if x is True, return e1
if x is False, return e2.

that is,

When applying True on e1 and e2, return e1;
When applying False on e1 and e2, return e2.

So, can we represent True and False as lambda expressions?

Conditionals

- Conditional: if x then e_1 else e_2
- Trick: We first define true and false as functions:
  let true $= (\lambda x \lambda y.x)$ let false $= (\lambda x \lambda y.y)$
- Recall: λ-bindings extend as far to the right as possible:
  $(\lambda x \lambda y.x) \equiv (\lambda x(\lambda y.x))$
- Then define conditional as:
  if p then e_1 else $e_2 \rightarrow (\lambda p \lambda e_1 \lambda e_2.p \ e_1 \ e_2)$
- Here, p is a predicate, i.e. function evaluating to true or false
- Example predicates are EQZ, GTZ, etc.
- Observation: If we define numbers carefully in λ calculus, we can also define those precisely, but we won't in class

Example:

if (w>v) then return w, else return v.

$(\lambda p\ \lambda x\ \lambda y.\ p\ x\ y)\ (w>v)\ w\ v$ gives

$(w>v)\ w\ v$ (called S1)

If (w>v) then (w>v) gives True, that is,

$\lambda x\ \lambda y.\ x$

So, S1 becomes

$(\lambda x\ \lambda y.\ x)\ w\ v \Rightarrow w.$

O.w., (w>v) gives False, that is,

$\lambda x\ \lambda y.\ y$

So, S1 becomes

$(\lambda x\ \lambda y.\ y)\ w\ v \Rightarrow v.$

Named Functions

- We want to add functions with names
- Solution: Use the let-construct to name anonymous λ terms:

To define f as the name of the following function:

$\lambda x.\ e1$

in the context of e2.

Use let-construct:

let $f = \lambda x.\ e1$ in e2.

Use lambda calculus:

$(\lambda f.\ e2)\ (\lambda x.\ e1)$

Example:

x =2
y = 3
if (x>y) then return x, else return y.

$$\lambda x.\ \lambda y.\ ((\lambda p.\ \lambda w.\ \lambda v.\ p\ w\ v)\ (x>y)\ x\ y)\ 2\ 3$$