Lambda Calculus

COS 441 Slides 12

read:  3.4, 5.1, 5.2, 3.5 Pierce
the lambda calculus

• Originally, the lambda calculus was developed as a logic by Alonzo Church in 1932 at Princeton
  – Church says: “There may, indeed, be other applications of the system than its use as a logic.”
  – Dave says: “There sure are!”

• The lambda calculus is a language of pure functions

• It serves as the semantic basis for languages like Haskell that are based around functions, but also pretty much every other language that includes some notion of function

• It is just as powerful as a Turing Machine (lambda terms can compute anything a Turing Machine can) and provides an alternate basis for understanding computation

• Pierce Text, Chap 3, 5
Operational Semantics

- **Denotational semantics** for a language provides a function that translates from program syntax into mathematical objects like sets, functions, lists or even some other programming language
  - a denotational semantics acts like a compiler

- **Operational semantics** works by rewriting or executing programs step-by-step
  - it uses only one program syntax to explain how a program runs

- As languages become more complicated, it is often easier to define operational semantics than denotational semantics
  - it requires less math to do so
  - but you might not be able to prove particularly strong theorems using the semantics

- Starting with the lambda calculus, we will look at operational semantics
Operational Rules

- Operational rules typically look like this:

  \[
  \text{condition}_1 \ldots \text{condition}_k \quad \text{subprogram} \rightarrow \text{subprogram}' \\
  \text{prog} \rightarrow \text{prog}'
  \]

- Read \text{prog} \rightarrow \text{prog}' as prog "steps to" prog'
- \text{prog} \rightarrow \text{prog}' is a new kind of judgement (aka property/Assertion/Claim)
Operational Rules

• Operational rules typically look like this:

\[
\text{condition}_1 \ldots \text{condition}_k \quad \text{subprogram} \rightarrow \text{subprogram}' \\
\quad \text{prog} \rightarrow \text{prog}'
\]

• Read \text{prog} \rightarrow \text{prog}' as prog "steps to" prog'

• \text{prog} \rightarrow \text{prog}' is a new kind of judgement (aka property/assertion/claim)

• An example, defining evaluation of if statements:

\[
\text{e} \rightarrow \text{e}' \\
\quad \text{if } \text{e} \text{ then } \text{c}_1 \text{ else } \text{c}_2 \rightarrow \text{if } \text{e}' \text{ then } \text{c}_1 \text{ else } \text{c}_2
\]

\[
\text{if True then } \text{c}_1 \text{ else } \text{c}_2 \rightarrow \text{c}_1 \\
\text{if False then } \text{c}_1 \text{ else } \text{c}_2 \rightarrow \text{c}_2
\]
LAMBDA CALCULUS
syntax

e ::= x  
    | \x.e  (a variable)
    | e e   (a function; in Haskell: \x -> e)
    | e e   (function application)

[ “\” will be written “λ” in a nice font and pronounced "lambda"]
syntax

• the identity function:
  • \( \lambda x.x \)

• 2 notational conventions:
  • applications associate to the left (like in Haskell):
  • “\( y z x \)” is “\( (y z) x \)”
  • the body of a lambda extends as far as possible to the right:
  • “\( \lambda x.x \lambda z.x z x \)” is “\( \lambda x.\lambda z.(x z x) \)”
the **scope** of $x$ is the entire body of the function
(ie: the $x$’s that appear in the body of the function refer to that particular argument)

$\\backslash x.x.x$

$\backslash x.x.y$

$y$ is **free** in the term $\backslash x.x.y$

$x$ is **bound** in the term $\backslash x.x.y$
the scope of the right-most $x$ includes the body of the function; the scope of the left-most $x$ does not.

If you wanted to refer to the first $x$, above, well you can't. You should have chosen a different variable name in your programs.

Important note: The names of bound variables don’t matter to the semantics of lambda calculus programs, so you can rename bound variables (provided you do so consistently) whenever you want.

\[
\lambda x.x \quad == \quad \lambda y.y \quad == \quad \lambda z.z
\]

\[
\lambda x.\lambda y.x \ y \quad == \quad \lambda y.\lambda x.y \ x \quad == \quad \lambda z.\lambda w.z \ w
\]
Call-by-value operational semantics

- single-step, call-by-value operational semantics:

  \[ e \rightarrow e' \]

- In English, we say “e steps to e’”

- This is a new kind of “judgement”, just like a Hoare triple was a judgement and there were rules that allowed us to conclude when it was a valid judgement
Call-by-value operational semantics

• single-step, call-by-value operational semantics: $e \rightarrow e'$
  – values are $v ::= \lambda x.e$
  – primary rule (beta reduction):

$$\lambda x.e \, v \rightarrow e [v/x]$$

  – $e [v/x]$ is the expression in which all free occurrences of $x$ in $e$ are replaced with $v$
  – this replacement operation is called substitution
  – implementing substitution for the lambda calculus properly is actually tougher than it would seem at first

   call-by-value
   since argument is a
   value rather than
general expression
• beta rule:

\[
\begin{align*}
\lambda x.e & \rightarrow e[v/x] \\
\text{(beta)}
\end{align*}
\]

• is used together with search rules:

\[
\begin{align*}
e_1 & \rightarrow e_1' \\
\text{(app1)}
\end{align*}
\]

\[
\begin{align*}
e_2 & \rightarrow e_2' \\
\text{(app2)}
\end{align*}
\]

• notice, because of the rules, evaluation is left to right

• and that's it -- 3 rules -- that is all you need to know about evaluating expressions in the lambda calculus!
Example

- Program:

\[ (\lambda x. \lambda y. x\ y\ y)\ (\lambda w. w)\ (\lambda z. z)\]

- Proof that it can take a step:

\[
\frac{\text{app1}}{e_1 \to e_1'\quad e_1\ e_2 \to e_1'\ e_2}
\]
\[
\frac{\text{app2}}{e_2 \to e_2'\quad v\ e_2 \to v\ e_2'}
\]
Example

- Program:
  \[ (((\lambda x. \lambda y. x y) \ (\lambda w. w)) \ (\lambda z. z)) \]

- Proof that it can take a step:
  \[
  \frac{\quad \frac{e_1 \rightarrow e_1'} {e_1 e_2 \rightarrow e_1' e_2}}{\frac{\quad \frac{e_2 \rightarrow e_2'} {v e_2 \rightarrow v e_2'}}{\frac{\quad \frac{(\lambda e) v \rightarrow e [v/x]}{(\lambda x. \lambda y. x y) (\lambda w. w) \rightarrow \lambda y. (\lambda w. w) y}}{(\lambda x. \lambda y. x y) (\lambda z. z) \rightarrow (\lambda y. (\lambda w. w) y) (\lambda z. z)}}}{e_1 e_2 \rightarrow e_1' e_2}} \]
Example

- **Program:**

  \[
  (((\lambda x. \lambda y. x y) \lambda w. w) \lambda z. z)
  \]

- **Proof that it can take a step:**

  \[
  \frac{\lambda x. \lambda y. x y \lambda w. w \rightarrow \lambda y. (\lambda w. w) y}{\lambda x. \lambda y. x y \lambda w. w \rightarrow \lambda y. (\lambda w. w) y \lambda z. z \rightarrow (\lambda y. (\lambda w. w) y) \lambda z. z} \text{(app1)}
  \]

- **Proof it can take a second step:**

  \[
  \frac{\lambda y. (\lambda w. w) y \lambda z. z \rightarrow (\lambda w. w) \lambda z. z}{\lambda y. (\lambda w. w) y \lambda z. z \rightarrow (\lambda w. w) \lambda z. z \rightarrow (\lambda w. w) (\lambda z. z)} \text{(app2)}
  \]

- **So we typically write (without explicit proofs):**

  \[
  (((\lambda x. \lambda y. x y) \lambda w. w) \lambda z. z \rightarrow (\lambda y. (\lambda w. w) y) \lambda z. z \rightarrow (\lambda w. w) (\lambda z. z))
  \]
Example

\((x.x\ x)\ (y.y)\)
Example

\((\lambda x.x x) (\lambda y.y)\)

--> x x [\(\lambda y.y / x\)]
Example

$$(\lambda x.x \ x) \ (\lambda y. y)$$

$->$ $x \ x \ [\lambda y. y / x]$  

$==$ $$(\lambda y. y) \ (\lambda y. y)$$
Example

\((\lambda x.x x) (\lambda y.y)\)

--> \(x x [\lambda y.y / x]\)

== \((\lambda y.y) (\lambda y.y)\)

--> \(y [\lambda y.y / y]\)
Example

\((\lambda x.x \ x) \ (\lambda y.y)\)

--> \(x \ x \ [\lambda y.y / x]\)

== \((\lambda y.y) \ (\lambda y.y)\)

--> \(y \ [\lambda y.y / y]\)

== \(\lambda y.y\)
A Non-Example

• Given:

$$(((\lambda x)(\lambda y)) ((\lambda w)(\lambda z)))$$

• One might think that:

$$(((\lambda x)(\lambda y)) ((\lambda w)(\lambda z)) \rightarrow (((\lambda x)(\lambda y))(\lambda z))$$

• Since:  \( (\lambda w)(\lambda z) \rightarrow (\lambda z) \)

• But that would require the presence of this rule:

$$\frac{e_2 \rightarrow e'_2}{e_1 e_2 \rightarrow e_1 e'_2} \quad \text{(app3)}$$
Another example

(\textbf{x.x} \textbf{x}) (\textbf{x.x} \textbf{x})
Another example

\((\lambda x.x) (\lambda x.x)\)

\(\rightarrow x \; x \; [\lambda x.x \; x/x]\)
Another example

\((\lambda x.x \ x) \ (\lambda x.x \ x)\)

\[\rightarrow x \ x \ [\lambda x.x \ x/x]\]

\[== (\lambda x.x \ x) \ (\lambda x.x \ x)\]

• In other words, it is simple to write non-terminating computations in the lambda calculus

• So, what else can we do with the lambda calculus?
We can do everything

• The lambda calculus can be used as an “assembly language”
• We can show how to compile useful, high-level operations and language features into the lambda calculus
  – Result = adding high-level operations is convenient for programmers, but not a computational necessity
  – Result = make your compiler intermediate language simpler

• Translations that show how to implement various useful programming features in the lambda calculus are typically called "Church encodings" after Alonzo Church
Aside

- Single-step reduction, one by one, gets pretty tedious, so we can make up a new notation for multi-step evaluation (and give the new notation a formal definition!)
- To say a program takes 0, 1 or many steps, we write:

  \[ e \rightarrow^* e' \]

- Rules:

  \[
  \begin{align*}
  e \rightarrow^* e & \quad \text{(reflexivity)} \\
  e_1 \rightarrow e_2 & \quad e_2 \rightarrow^* e_3 \\
  e_1 \rightarrow^* e_3 & \quad \text{(transitivity)}
  \end{align*}
  \]
• A multi-step proof:

\[
\begin{align*}
\text{e} & \to e \\
\text{e}_1 & \to \text{e}_2 \\
\text{e}_2 & \to e_3 \\
\text{e}_1 & \to e_3
\end{align*}
\]  

(Reflexivity)

\[
\begin{align*}
\text{a} & \to \text{b} \\
\text{b} & \to e
\end{align*}
\]  

(Transitivity)

\[
\text{a} \to e
\]
• A multi-step proof:
A multi-step proof:

\[
\begin{align*}
\text{e1} & \rightarrow \text{e2} & \text{e2} & \rightarrow^{*} \text{e3} & \text{e1} & \rightarrow^{*} \text{e3} \\
\text{d} & \rightarrow \text{e} & \text{e} & \rightarrow^{*} \text{e} \\
\text{c} & \rightarrow \text{d} & \text{d} & \rightarrow^{*} \text{e} \\
\text{b} & \rightarrow \text{c} & \text{c} & \rightarrow^{*} \text{e} \\
\text{a} & \rightarrow \text{b} & \text{b} & \rightarrow^{*} \text{e} & \text{a} & \rightarrow^{*} \text{e}
\end{align*}
\]
Aside

A multi-step proof:

- A proof that
  - $e \rightarrow^* e$ (reflexivity)
- $e_1 \rightarrow e_2$, $e_2 \rightarrow^* e_3$ (transitivity)
- $e_1 \rightarrow^* e_3$

- $a \rightarrow b$, $b \rightarrow^* e$, $c \rightarrow d$, $d \rightarrow^* e$, $e \rightarrow^* e$

- Proof that
  - $a \rightarrow b$
CHURCH ENCODINGS
Let Expressions

• It is useful to bind intermediate results of computations to variables:
  
  \[
  \text{let } x = e_1 \ \text{in} \ e_2
  \]

• Question: can we implement this idea in the lambda calculus?

\[
\begin{align*}
\text{source} &= \text{lambda calculus} + \text{let} \\
\text{translate/compile} & \\
\text{target} &= \text{lambda calculus}
\end{align*}
\]
Let Expressions

• It is useful to bind intermediate results of computations to variables:
  
  let x = e1 in e2

• Question: can we implement this idea in the lambda calculus?

  translate (let x = e1 in e2) =
Let Expressions

• It is useful to bind intermediate results of computations to variables:
  
  \[ \text{let } x = e_1 \text{ in } e_2 \]

• Question: can we implement this idea in the lambda calculus?

  \[ \text{translate } (\text{let } x = e_1 \text{ in } e_2) = (\lambda x. \text{translate } e_2) \text{ (translate } e_1) \]
Let Expressions

• It is useful to bind intermediate results of computations to variables:
  
  let x = e1 in e2

• Question: can we implement this idea in the lambda calculus?
  
  translate (let x = e1 in e2) =
  
  (\x. translate e2) (translate e1)

  translate (x) = x

  translate (\x.e) = \x. translate e

  translate (e1 e2) = (translate e1) (translate e2)
ENCODING BOOLEANS
booleans

• we can encode booleans
  – we will represent “true” and “false” as functions named “tru” and “fls”
  – how do we define these functions?
  – think about how “true” and “false” can be used
  – they can be used by a testing function:
    • “test b then else” returns “then” if b is true and returns “else” if b is false
    • the only thing the implementation of test is going to be able to do with b is to apply it
    • the functions “tru” and “fls” must distinguish themselves when they are applied
booleans

• the encoding:

\[ \text{tru} = \lambda t.\, \lambda f.\, t \]

\[ \text{fls} = \lambda t.\, \lambda f.\, f \]

\[ \text{test} = \lambda x.\, \lambda \text{then}.\, \lambda \text{else}.\, x \, \text{then} \, \text{else} \]
booleans

\( \text{tru} = \lambda t. \lambda f. t \quad \text{fls} = \lambda t. \lambda f. f \)
\( \text{test} = \lambda x. \lambda \text{then}. \lambda \text{else}. x \quad \text{then} \quad \text{else} \)

eg:

test \text{tru} a b
booleans

\[ \text{tru} = \lambda t. \lambda f. t \quad \text{fls} = \lambda t. \lambda f. f \]
\[ \text{test} = \lambda x. \lambda \text{then}. \lambda \text{else}. x \, \text{then} \, \text{else} \]

eg:

test \, \text{tru} \, a \, b
\[= (\lambda x. \lambda \text{then}. \lambda \text{else}. x \, \text{then} \, \text{else}) \, (\lambda t. \lambda f. t) \, a \, b \]
booleans

definitions:

\[
\text{tru} = \lambda t. \lambda f. \, t \\
\text{fls} = \lambda t. \lambda f. \, f
\]

\[\text{test} = \lambda x. \lambda \text{then.} \lambda \text{else.} \, x \, \text{then else}\]

eg:

demonstration:

test \, \text{tru} \, a \, b

\[= (\lambda x. \lambda \text{then.} \lambda \text{else.} \, x \, \text{then else}) \, (\lambda t. \lambda f. \, t) \, a \, b\]

\[\rightarrow^* (\lambda t. \lambda f. \, t) \, a \, b\]
booleans

\[
\text{tru} = \lambda t. \lambda f. t \quad \text{fls} = \lambda t. \lambda f. f
\]
\[
\text{test} = \lambda x. \lambda \text{then}. \lambda \text{else}. \lambda x \text{ then } \text{else}
\]

eg:

\[
\text{test \ tru \ a \ b} = (\lambda x. \lambda \text{then}. \lambda \text{else}. \lambda x \text{ then } \text{else}) (\lambda t. \lambda f. t) \ a \ b
\]
\[
\rightarrow^* (\lambda t. \lambda f. t) \ a \ b
\]
\[
\rightarrow^* a
\]
tru = \t.\f. t  
fls = \t.\f. f 

test = \x.\then.\else. x then else

create a function "and" in the lambda calculus that mimics conjunction. It should have the following properties.

and tru tru -->* tru
and fls tru -->* fls
and tru fls -->* fls
and fls fls -->* fls
SUMMARY
The Lambda Calculus involves just 3 things:
- variables \( x, y, z \)
- function definitions \( \lambda x.e \)
- function application \( e_1 e_2 \)

Despite its simplicity, despite the apparent lack of if statements or loops or any data structures other than functions, it is Turing complete.

Church encodings are translations that show how to encode various data types or linguistic features in the lambda calculus.