The Lambda Calculus

CS510

The Lambda Calculus

- The result of distilling the essence of all functional programming languages (FPLs)
- A concise and rigorous testbed for exploring properties of programming languages
- Introduced in 1932 by Church (Alonzo $\sim$, 1903-1995)
  - These properties are in fact not limited to FPL
  - Examples are:
    - Static and dynamic scoping
    - Type-checking and type inference
    - Abstract machines for program execution
    - Judicious strategies for program execution (eg. sharing)
    - Garbage collection
    - Secure compilation

The Lambda Calculus

- Is thus a concise FPL
- There are only two operations:
  - Build functions
  - Apply them to arguments
- Numbers, booleans, lists, trees, pairs, etc. can all be encoded as functions
- We shall represent the above two operations directly in OCaml
- It is fair to say that OCaml is an extension of the Lambda Calculus

The Lambda Calculus

- There are two important aspects of the Lambda Calculus
  - Syntax: How to build expressions (or programs)
  - Semantics: How to execute the expressions (what it means to run a program)
- First we present an example
- Then we address syntax and semantics
- Note:
  - We can execute the lambda directly in OCaml

The Lambda Calculus: A Sample Computation

A "program" in the Lambda Calculus:

$$(\lambda x.x + x)((\lambda y.2 * y)\, 4)$$

Execution of a program:

$$\begin{array}{ll} & (\lambda x.x + x)((\lambda y.2 * y)\, 4) \\ \rightarrow & (\lambda x.x + x)(2 * 4) \\ \rightarrow & (\lambda x.x + x)8 \\ \rightarrow & 8 + 8 \\ \rightarrow & 16 \end{array}$$

- 16 is the result of running or evaluating the program

The Syntax of the λ-calculus

$$
\begin{array}{rcll}
\langle \text{exp} \rangle & ::= & \langle \text{identifier} \rangle & \textit{variable} \\
 & | & \lambda \langle \text{identifier} \rangle . \langle \text{exp} \rangle & \textit{abstraction} \\
 & | & \langle \text{exp} \rangle \ \langle \text{exp} \rangle & \textit{application}
\end{array}
$$

Examples:

- y
- $\lambda x.x$
- yz
- $(\lambda x.x)\ (\lambda y.y)$
- $(\lambda x.(xx))\ (\lambda x.(xx))$

- The λ-expressions are an inductive set
- In an abstraction $\lambda x.s$ we call x a binding variable

Free and Bound Variables Occurrences

- Bound: x is bound in an expression E if it refers to a formal parameter introduced in E
- Free: x is free in E if it is not declared in E

Example:

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

At run-time, all variables must be either

1. lexically bound: bound by a formal parameter, or
2. globally bound: bound by a top-level definition or supplied by the system

Examples

- $\lambda x.x$
- $\lambda y.(id\ y)$
- $(\lambda x.x)\ ((\lambda y.y)\ z)$
- $y(\lambda y.y)$
- $(\lambda id.(id\ id))\ (\lambda y.y)$

Renaming Bound Variables

- Bound variables can be renamed without changing the meaning of an expression
- Eg. $\lambda x.x$ can be renamed to $\lambda y.y$
- In the λ-calculus, renaming is called α-conversion
- But renaming requires caution

Renaming

$$M^x_y$$

Replace every free occurrence of x with y

- Example
  - $(x\,y)^x_z = z\,y$
  - $(\lambda y.x)^x_z = \lambda y.z$
  - $(\lambda y.x)^x_y = \lambda y.y$ Wrong??
    - Renaming can capture variables
    - y is typically required not to be bound in M

α-Equivalence

$$\frac{y \notin FV(M) \text{ and } y \text{ not a binding variable in } M}{\lambda x.M =_\alpha \lambda y.M^x_y}$$

$$\frac{M =_\alpha M'}{\lambda x.M =_\alpha \lambda x.M'} \qquad \frac{M =_\alpha M' \quad P =_\alpha P'}{M\,P =_\alpha M'\,P'}$$

Examples:

- $\lambda x.x =_\alpha \lambda y.y$
- $\lambda x.y =_\alpha \lambda z.y$
- $\lambda x.y \neq_\alpha \lambda x.z$
- $\lambda x.\lambda x.x \neq_\alpha \lambda y.\lambda x.y$

Renaming requires Caution

- We must not capture existing references

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

cannot be renamed to

$$\lambda y.(y\,y)$$

- We must not rename bound uses

$\lambda x.(\lambda x.(z\ x)))\ (z\ x)$ can be renamed to $\lambda y.(\lambda x.(z\ x))\ (z\ y)$

$\lambda x.(\lambda x.(z\ x)))\ (z\ x)$ cannot be renamed to
$\lambda y.(\lambda x.(z\ y))\ (z\ y)$

Free Variables Defined Formally

$$
\begin{array}{rcll}
\langle \text{exp} \rangle & ::= & \langle \text{identifier} \rangle & \textit{variable} \\
& | & \lambda \langle \text{identifier} \rangle . \langle \text{exp} \rangle & \textit{abstraction} \\
& | & \langle \text{exp} \rangle \ \langle \text{exp} \rangle & \textit{application}
\end{array}
$$

$FV(\cdot) : \langle \text{exp} \rangle \to \wp \langle \text{identifier} \rangle$

$$
\begin{array}{rcl}
FV(x) & = & \{x\} \\
FV(\lambda x.E) & = & FV(E) - \{x\} \\
FV(E_1\ E_2) & = & FV(E_1) \cup FV(E_2)
\end{array}
$$

Bound Variables Defined Formally

$$
\begin{array}{lcll}
\langle \text{exp} \rangle & ::= & \langle \text{identifier} \rangle & \textit{variable} \\
& | & \lambda \langle \text{identifier} \rangle . \langle \text{exp} \rangle & \textit{abstraction} \\
& | & \langle \text{exp} \rangle \ \langle \text{exp} \rangle & \textit{application}
\end{array}
$$

$BV(\cdot) : \langle \text{exp} \rangle \rightarrow \wp \langle \text{identifier} \rangle$

$$
\begin{array}{rcl}
BV(x) & = & \emptyset \\
BV(\lambda x.E) & = & BV(E) \cup (\{x\} \cap FV(E)) \\
BV(E_1\ E_2) & = & BV(E_1) \cup BV(E_2)
\end{array}
$$

The Lambda Calculus

Scoping

Semantics

Parsing

Declaration vs. Reference

$$\lambda x.x$$

The two occurrences of x are used differently:

- x a declaration or formal parameter:
  - introduces the variable as a name for some value (the value shall be supplied when the procedure is called)
- x is a reference
  - represents variable use.

A similar example but in OCaml:

```
let f x  = x+x
```

Scoping

Determining which declaration is associated with a particular reference

- A declaration may be one of
  - formal parameter list
  - `define` construct
- A reference is a variable reference

Example:

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

Scoping Rules

Two Rules:

- Static: determining which declaration is associated with each reference by observing program text
- Dynamic: we can only determine which declaration is associated with a reference at run-time

Notes:

- Examples of dynamic scoping will be seen later
- For now, we use the standard, static scoping approach

Static Scoping – Region vs Scope

Each declaration determines a

- region: area of program text in which the declaration is in effect
- scope: area of program text in which uses of the defined variable refer to the declaration

Region and scope may not be the same due to shadowing (local redeclaration)

$$\lambda x.(\lambda x.(x + 1))$$

The inner declaration of `x` shadows the outer declaration. It creates a hole in the scope of the outer declaration.

The Lambda Calculus

Scoping

Semantics

Parsing

Semantics

- How terms are evaluated or executed?
- There are various ways of defining the semantics of a PL rigorously:
  - Operational
  - Denotational
  - Axiomatic
- We will define a (small-step) operational semantics
- This requires introducing evaluation judgements

$$M \to N \quad \text{“}\textit{term } M \textit{ reduces, in one step, to term } N\text{''}$$

Substitution

$$M\{x \leftarrow N\}$$

"Substitute all free occurrences of x in M with N"

$$
\begin{aligned}
x\{x \leftarrow N\} &\stackrel{\text{def}}{=} N \\
y\{x \leftarrow N\} &\stackrel{\text{def}}{=} y, \quad x \neq y \\
(M_1\, M_2)\{x \leftarrow N\} &\stackrel{\text{def}}{=} M_1\{x \leftarrow N\}\, M_2\{x \leftarrow N\} \\
(\lambda y.M)\{x \leftarrow N\} &\stackrel{\text{def}}{=} \lambda y.M\{x \leftarrow N\}\ x \neq y,\ y \notin FV(N)
\end{aligned}
$$

1. Condition $x \neq y,\ y \notin FV(N)$ can always be upheld using renaming
2. It is there to avoid variable capture

Operational Semantics - Functions

Values:

$$V \quad ::= \quad \lambda x.M$$

Rules:

$$\frac{M_1 \to M_1'}{M_1\ M_2 \to M_1'\ M_2}\ (\text{E-App1})$$

$$\frac{M_2 \to M_2'}{(\lambda x.M_1)\ M_2 \to (\lambda x.M_1)\ M_2'}\ (\text{E-App2})$$

$$\frac{}{(\lambda x.M)\ V \to M\{x \leftarrow V\}}\ (\text{E-AppAbs})$$

Examples

- $(\lambda y.y)\, z \to z$
- $(\lambda x.x\, z)\, (\lambda y.y) \to (\lambda y.y)\, z$
- $(\lambda z.z)\, ((\lambda y.y)\, \mathit{true}) \to (\lambda z.z)\, \mathit{true}$
- There is no E s.t. $\lambda x.x \to E$
- There is no E s.t. $x \to E$
  - We say x is in normal form
  - Note that normal forms need not be values

The Lambda Calculus

Scoping

Semantics

Parsing

Parsing

- The syntax we've been using to talk about the Lambda Calculus

$$
\begin{array}{rcll}
\langle \text{exp} \rangle & ::= & \langle \text{identifier} \rangle & \textit{variable} \\
 & | & \lambda \langle \text{identifier} \rangle . \langle \text{exp} \rangle & \textit{abstraction} \\
 & | & \langle \text{exp} \rangle \ \langle \text{exp} \rangle & \textit{application}
\end{array}
$$

- We are now interested in representing these expressions as code in a PL

PL Syntax

The syntax of a PL usually has two parts

- Concrete syntax
  - Syntax in which the programmer codes
- Abstract syntax
  - Internal representation of the concrete syntax
  - Used by the interpreter or compiler for running the program

Parser

- A program that transforms concrete syntax to abstract syntax

Concrete Syntax

- Typically described in terms of BNF grammars
  - A set of syntactic rules that identify well-formed expressions
- The one we use for the Lambda Calculus

$$
\begin{array}{lcll}
\langle \text{exp} \rangle & ::= & \langle \text{identifier} \rangle & \textit{variable} \\
 & | & \texttt{lam}\ (\langle \text{identifier} \rangle)\langle \text{exp} \rangle & \textit{abstraction} \\
 & | & \texttt{app}\ \langle \text{exp} \rangle\ \langle \text{exp} \rangle & \textit{application} \\
 & | & (\langle \text{exp} \rangle) &
\end{array}
$$

Concrete Syntax

$$
\begin{array}{lcll}
\langle \text{exp} \rangle & ::= & \langle \text{identifier} \rangle & \textit{variable} \\
 & \mid & \texttt{lam}\ (\langle \text{identifier} \rangle)\langle \text{exp} \rangle & \textit{abstraction} \\
 & \mid & \texttt{app}\ \langle \text{exp} \rangle\ \langle \text{exp} \rangle & \textit{application} \\
 & \mid & (\langle \text{exp} \rangle) &
\end{array}
$$

- Components of a BNF grammar
  - Productions:
    $\langle \text{exp} \rangle ::= \langle \text{identifier} \rangle$
    $\langle \text{exp} \rangle ::= \texttt{lam}\ (\langle \text{identifier} \rangle)\langle \text{exp} \rangle$
    $\langle \text{exp} \rangle ::= \texttt{app}\ \langle \text{exp} \rangle\ \langle \text{exp} \rangle$
    $\langle \text{exp} \rangle ::= (\langle \text{exp} \rangle)$
  - Non-terminals: $\langle \text{exp} \rangle$, $\langle \text{identifier} \rangle$
  - Terminals: "(", ")", "`lam`", "`app`"'
- Notice that the rules talk about specific syntactic elements such as parenthesis etc. Eg.
  - `lam lam` is not a valid λ-expression

Abstract Syntax

- An abstraction over the concrete syntax
- It may be seen as the underlying inductive definition resulting from abstracting away syntactic elements such as parenthesis and the use of the word "lam"
- In essence it identifies the rule associated with each syntactic component

```
type term =
    | Var of string
    | Lambda of string*term
    | App of term*term
```

Concrete vs. Abstract

Concrete syntax:

$$\texttt{lam}\ (x)\ x$$

- Designed for human consumption

Abstract syntax:

```
LamExp ("x", VarExp "x")
```

- Highlights structure e.g., to enable processing by meta-programs
- Elements of an abstract syntax are called Abstract Syntax Trees (ASTs)

Parsing

- Process of deriving the corresponding AST from some concrete syntax representation
- Two steps
  - Lexer: Identify list of tokens from source file
  - Parser: Identify expression tree from the list of tokens

Lexer

Tokens: identifier, parenthesis, “lam”, “app”

```
type token =
  | OpenPar
  | ClosePar
  | Lambda
  | App
  | Id of string
  | Eof
```

```
 # tokenize "(lam (x) x)";;
- : token list =
[OpenPar; Lambda; OpenPar; Id "x"; ClosePar; Id "x";
    ↪ ClosePar; Eof]
```

Lexer

```
(* char list -> token list *)
let rec tokenize_list  = function
  | [] -> [Eof]
  | ' '::xs -> tokenize_list xs
  | '('::xs -> OpenPar::tokenize_list xs
  | ')'::xs -> ClosePar::tokenize_list xs
  | 'l'::xs ->
    if ((List.length xs > 4) && (take xs 2 = ['a';'m']))
    then Lambda::tokenize_list (drop xs 2)
    else failwith "Unrecognized token: did you mean lam?"
  | 'a'::xs ->
    if ((List.length xs > 4) && (take xs 2 =  ['p';'p']))
    then App::tokenize_list (drop xs 2)
    else failwith "Unrecognized token: did you mean app?"
  | c::xs when is_letter c ->
    Id (char_list_to_string (c::take_while xs is_letter)):: tokenize_list (drop_while xs is_letter)
  | _ -> failwith "Unrecognized token"
```

Parser - Parenthesis

```
(* token list -> token * token list *)
let parseCloseParen xs =
  match xs with
  | ClosePar::ys -> (ClosePar,ys)
  | _ -> failwith "parseCloseParen: expected closing
      ↪ parenthesis "

(* token list -> token * token list *)
let parseOpenParen xs =
  match xs with
  | OpenPar::ys -> (OpenPar,ys)
  | _ -> failwith "parseOpenParen: expected opening
      ↪ parenthesis"
```

Parser - Identifier and Formal Parameter to Abstraction

```
(* token list -> string * token list *)
let parseIdent xs =
  match xs with
  | Id s::ys -> (s,ys)
  | _ -> failwith "parseIdent: expected identifier
      ↪ parenthesis"

(* token list -> string * token list *)
let  parseFormParam xs =
  let (_,xs1) = parseOpenParen xs
  in let (var,xs2) =  parseIdent xs1
  in let (_,xs3) = parseCloseParen xs2
  in (var,xs3)
```

Parser - Expression

```
(* token list -> expr * token list *)
let rec parseExp = function
  | [] -> (VarExp "eof",[])
  | Id s::xs -> (VarExp s, xs)
  | OpenPar::xs ->
    let (e1,xs1) = parseExp xs
    in let (_,xs2) = parseCloseParen xs1
    in (e1,xs2)
  | App::xs ->
    let (e1,xs1) = parseExp xs
    in let (e2,xs2) = parseExp xs1
    in (AppExp(e1,e2),xs2)
  | Lambda::xs ->
    let (id,xs1) = parseFormParam xs
    in let (pBody,xs2) =  parseExp xs1
    in (LamExp(id,pBody),xs2)
  | xs -> failwith "parse: Invalid concrete syntax "
```

$\langle \text{exp} \rangle ::= \langle \text{identifier} \rangle \mid \texttt{lam}\ (\langle \text{identifier} \rangle)\langle \text{exp} \rangle \mid, \text{app}\ \langle \text{exp} \rangle\ \langle \text{exp} \rangle$

$\langle \text{exp} \rangle ::= (\langle \text{exp} \rangle)$