The Lambda Calculus

CS496

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 Scheme
- It is fair to say that Scheme 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)
- We will mainly focus on syntax
- For the semantics we will rely on Scheme, given that it is an implementation itself of the Lambda Calculus
  - We can execute the lambda directly in Scheme

The Lambda Calculus: A Sample Computation

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

- $(\lambda x.x + x)((\lambda y.2 * y)\, 4)$ is the program or expression
- 16 is the result of running or evaluating the program

The Syntax of the λ-calculus

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

Examples:

- y
- (λx. x)
- (y z)
- ((λx. x) (λy. y))
- ((λx. (x x)) (λx. (x x)))

- The λ-expressions are an inductive set

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 requires Caution

We must not capture existing references

$$(\lambda x.(cons\ x\ '()))$$
;; cannot be renamed to
$$(\lambda cons.(cons\ cons\ '()))$$

We must not rename bound uses

$$(\lambda x.((\lambda x.(cons\ x\ '()))(cons\ x\ '())))$$
;; can be renamed to
$$(\lambda y.((\lambda x.(cons\ x\ '()))(cons\ y\ '())))$$
;; cannot be renamed to
$$(\lambda y.((\lambda x.(cons\ y\ '()))(cons\ y\ '())))$$

Free Variables Defined Formally

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

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

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

Bound Variables Defined Formally

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

$BV(\cdot) : \langle \mathrm{exp} \rangle \to \wp \langle \mathrm{var} \rangle$

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

Static vs. Dynamic Properties

The notion of free/bound variable is a static one

- Static properties: Determined by analyzing text of program without considering inputs
  - names of variables
  - list of procedures called in block of code
  - declaration associated with a particular variable
  - whether a variable occurrence is a declaration or reference
  - free and bound variables

- Dynamic properties: Determined by run-time inputs
  - whether a numerical expression will evaluate to a positive or negative number.

The Lambda Calculus

Scoping

Encoding the Lambda Expressions in Scheme

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 Scheme:

```
(define (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.

Matching Use/Declaration

To find associated declaration, start at variable and move outward

- $\lambda x.(\lambda y.(\lambda x.(x\ y))\ x)$
- $\lambda z.((\lambda a.\lambda b.\lambda c.(a\ (\lambda a.(a + c))\ b))(\lambda f.\lambda x.(f\ (z\ x))))$

Lexical Addressing

Number of declarations (contours) passed along journey outward.

$$\begin{array}{l}\lambda x.\lambda y.\\ \quad ((\lambda a.\\ \quad\quad (x\ (a\ y)))\\ \quad\quad x)\end{array}$$

$$\begin{array}{l}\lambda x.\lambda y.\\ \quad ((\lambda a.\\ \quad\quad ((x : 1\ 0)((a : 0\ 0)(y : 1\ 1))))\\ \quad\quad (x : 0\ 0))\end{array}$$

$$\begin{array}{l}\lambda 2\\ \quad ((\lambda 1\\ \quad\quad ((: 1\ 0)((: 0\ 0)(: 1\ 1))))\\ \quad\quad (: 0\ 0))\end{array}$$

Representing λ-calculus in Scheme

- The lambda calculus is just another example of an inductive set
- We shall see two possible representations
  1. Using lists
  2. Using a `define-datatype`
- We could also encode them using Scheme functions themselves

Representing λ-calculus using Lists

Syntax of Lambda Calculus

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

Encoding in Scheme

$$\begin{array}{rcll} \langle \text{exp} \rangle & ::= & \langle \text{identifier} \rangle & \textit{variable} \\ & \mid & (\texttt{lambda}\,(\langle \text{identifier} \rangle)\,\langle \text{exp} \rangle) & \textit{abstraction} \\ & \mid & (\langle \text{exp} \rangle\,\langle \text{exp} \rangle) & \textit{application} \end{array}$$

Representing λ-calculus using Lists

Functions for constructing lambda terms

```
(define make-vref                      ;; constructors
  (lambda (id) id))

(define make-lam
  (lambda (id exp)
    (list 'lambda (list id) exp)))

(define make-ap
  (lambda (fun arg)
    (list fun arg)))
```

Representing λ-calculus using Lists(cont.)

```
(define lam->var caadr)            ;; destructors
(define lam->exp caddr)
(define ap->fun car)
(define ap->arg cadr)

(define vref?                      ;; predicates
  (lambda (e) (not (pair? e))))

(define lam?
  (lambda (e) (and (pair? e) (equal? (car e)
    'lambda))))

(define ap?
  (lambda (e) (and (pair? e) (not (lam? e)))))
```

- Example on the board

Compute Free Variables – An Exercise

```scheme
(define (fv exp)
    (if (vref? exp)
        (list exp)
        (if (lam? exp)
            ...
            (if (ap? exp)
                ...
                (error "bad exp"))))))
```

Examples:

```scheme
> (fv (make-vref 'x))
(x)

> (fv (make-lam 'x (make-vref 'x)))
()

> (fv '((lambda (x) x) (lambda (y) x)))
(x)
```

Representing λ-calculus using `define-datatype`

Syntax of Lambda Calculus

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

Encoding using `define-datatype`

```
(define-datatype expression expression?
  (var-exp
     (id symbol?))
  (lambda-exp
     (id symbol?)
     (body expression?))
  (app-exp
     (rator expression?)
     (rand expression?)))
```

The Lambda Calculus

Scoping

Encoding the Lambda Expressions in Scheme

Parsing

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
- Here is an example we've already seen

```
<expr> ::= <identifier>
         | (lambda (<identifier>) <expr>
         | (<expr> <expr>)
```

- Other examples were given when we introduced notations for inductive sets

Concrete Syntax

```
<expr> ::= <identifier>
         | (lambda (<identifier>) <expr>
         | (<expr> <expr>)
```

- Components of a BNF grammar
  - Productions:
    `<expr> ::= <identifier>`
    `<expr> ::= (lambda (<identifier>) <expr>`
    `<expr> ::= (<expr> <expr>)`
  - Non-terminals: `<expr>`, `<identifier>`
  - Terminals: "(", ")", "lambda"
- Notice that the rules talk about specific syntactic elements such as parenthesis, the word "lambda", etc. Eg.
  - `((x y))` is not a valid λ-expression, nor is `(lam (x) x)`

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 "lambda"
  - In essence it identifies the rule associated with each syntactic component
- Here is the abstract syntax corresponding the the concrete syntax for the λ-calculus

Concrete vs. Abstract

Concrete syntax:

```
(lambda (x) (x y))
```

- Designed for human consumption

Abstract syntax:

```
(lambda-exp 'x
            (app-exp (var-exp 'x)
                     (var-exp 'y)))
```

- 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
- Issues:
  - programs are typically represented as strings of characters

```
'(#\( #\l #\a #\m #\b #\a
  #\( #\x #\) #\( #\x #\y #\) #\))
```

  - we often represent programs as lists of symbols in Scheme

```
'(lambda (x) (x y))
```

  - if we represent ASTs using datatypes, we must be able to convert from these forms to datatype value form.

Parsing

```
<expr> ::= <identifier>
         | (lambda (<identifier>) <expr>)
         | (<expr> <expr>)
```

Examples:

```
'x

'(x z)

'(lambda (x) (x y))
```

Idea: look at datum...

- if symbol, then it's a `identifier`
- if pair, then it's either a `lambda` or `app` (check for `'lambda` tag)

Parse

BNF definition:

```
<expr> ::= <identifier>
         | (lambda (<identifier>) <expr>
         | (<expr> <expr>)
```

Scheme parsing procedure:

```
(define (parse datum)
  (cond
    ((symbol? datum) (var-exp datum))
    ((pair? datum)
     (if (eq? (car datum) 'lambda)
         (lambda-exp (caadr datum)
                     (parse (caddr datum)))
         (app-exp (parse (car datum))
                  (parse (cadr datum)))))
    (else (error "parse: Invalid concrete syntax"
   datum))))
```

Parse – Example

```
(define (parse datum)
  (cond
    ((symbol? datum) (var-exp datum))
    ((pair? datum)
     (if (eq? (car datum) 'lambda)
         (lambda-exp (caadr datum)
                     (parse (caddr datum)))
         (app-exp (parse (car datum))
                  (parse (cadr datum)))))
    (else (error "parse: Invalid concrete syntax" datum))))

> (parse 'x)
(var-exp 'x)

> (parse '(x y))
(app-exp (var-exp 'x) (var-exp 'y))

> (parse '(lambda (x) x))
(lambda-exp 'x (var-exp 'x))
```

Parse – Caveat

- This is an oversimplified example of a parser
- In particular, note that

```
> (parse '(x y z))
(app-exp (var-exp 'x) (var-exp 'y))
```

- In this case, the parser should in fact give a parsing error
- Parsers typically work in two phases:
  1. Dividing a sequence of symbols into a list of tokens
  2. Constructing an AST from the list of tokens
- We will not go into further details in this course

Unparse

- Reverse process: going from abstract to concrete syntax
- Unparsing is usually called pretty printing
- Here is an unparsing functions for our running example

```
(define unparse
  (lambda (exp)
    (cases expression exp
      (var-exp (datum) datum)
      (lambda-exp (id body)
        (list 'lambda
              (list id)
              (unparse body)))
      (app-exp (rator rand)
           (list (unparse rator)
                 (unparse rand)))
      (else (error "unparse: Invalid abstract syntax"
   exp)))))

```

Unparse

```
(define unparse
  (lambda (exp)
    (cases expression exp
      (var-exp (datum) datum)
      (lambda-exp (id body)
        (list 'lambda
              (list id)
              (unparse body)))
      (app-exp (rator rand)
           (list (unparse rator)
                 (unparse rand)))
      (else (error "unparse: Invalid abstract syntax" exp)))))

> (unparse (lambda-exp 'x (var-exp 'x)))
'(lambda (x) x)
```

- `compose unparse parse` is not always the identity in practice