PROGRAMMING LANGUAGES RECITATION

-Monika Dagar

1st October 2020

Agenda

- Lambda Calculus
- Lambda Calculus –Free and Bound Variables
- Lambda Calculus – Reduction
- Lambda Calculus – Evaluation Strategy
- Scheme Programming
- Tail Recursion
- Resources

Lambda Calculus

- Introduced by the mathematician ‘Alonzo Church’ in the 1930s.
- Definition: a mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution.
- It is a Turing complete language.

Lambda Calculus Syntax

- The lambda calculus consists of a language of lambda terms, which is defined by a certain formal syntax, and a set of transformation rules, which allow manipulation of the lambda terms.

- A valid lambda term:
  - *A variable, x , is itself a valid lambda term.*
  - *If M is a lambda term, and x is a variable, then (λx.M) is a lambda term (called an* ***abstraction****).*
  - *If M and N are lambda terms, then (MN) is a lambda term (called an* ***application****).*

- Every abstraction in lambda calculus takes a single argument.

Precedence and associative

- Precedence: application has higher precedence than lambda abstraction.
- Associative:
  - *Applications are left associative*
  - *Abstractions are right associative*

Shorthand

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

$\rightarrow t1\ t2\ t3 \Rightarrow (t1\ t2)\ t3$

$\rightarrow \lambda x\, y.\, t1\ t2 \Rightarrow (\lambda x\, y.\ t1\ t2)$

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

Free and Bound Variables

- The free variables of a term are those variables that are not bound by an abstraction. The set of free variables of an expression is defined inductively:
  - *The free variables of x are just x.*
  - *The set of free variables of λx.M is the set of free variables of M, but with x removed*
  - *The set of free variables of MN is the union of the set of free variables of M and the set of free variables of N.*

Reduction

α-conversion

- α conversion states that any bound variable is a placeholder and can be replaced (*renamed*) with a different variable, provided there are no clashes.
- Renaming rules:
  - *Only bound variables can be renamed, not free variables.*
  - *Renaming consistency: if we rename x in a term λ x. M, all occurrences of x in M must be replaced by y.*

    $$\lambda x.\ M\ \langle \text{rename } x \text{ to } y \rangle \rightarrow \lambda y.\ M[y/x]$$

  - *Renaming capture-avoiding: if we rename x in a term λ x. M, for every sub term M' inside M, if M' has a variable x that is bound to by current λ x. M, then y must be free in term M' by the renaming. Otherwise, you should do renaming for y firstly to free y.*

β-reduction

- Definition - A technique to evaluate lambda expression or to reduce it to a simplified one.
- Reduction rule: (λ x. t) s = t[s/x]
- t[s/x]: for the term t, substitute all occurrences of x that are bounded by current λ x. t to the term s.

η-reduction

- We use η-reduction to eliminate useless variables:
  - *Since:*
    - $(\lambda x.Mx) -\!\!\rightarrow_\eta M$
    - $(\lambda x.Mx)N \Leftrightarrow MN$
- This type of reduction is mostly for notational convenience and does not add any expressive power to the calculus.

Evaluation strategy

- **Normal order**: reduce the outermost “redex” first.
- **Applicative order**: arguments to a function application are evaluated first, from left to right before the function application itself is evaluated.

$$(\lambda x.(\lambda y.xy))((\lambda x.x)z) = \lambda y.((\lambda x.x)z)y$$
$$= \lambda y.zy$$

$$(\lambda x.(\lambda y.xy))((\lambda x.x)z) = (\lambda x.(\lambda y.xy))z$$
$$= \lambda y.zy$$

- An expression that can't be β-reduced any further is a normal form.
- Not everything has a normal form.
- If a lambda reduction terminates, it terminates to the same reduced expression regardless of reduction order.
- If a terminating lambda reduction exists, normal order evaluation will terminate.

$\Rightarrow$ $(\lambda z.zz)(\lambda z.zz)$

$\Rightarrow$

- iszero = (λ n. n (λ x. false) true)
- 0 = (λ s z. z)
- 1 = (λ s z. s z)
- true = (λ x y. x)
- false = (λ x y. y)
- **Question: How do we compute iszero 1 to get false via beta reduction?**

```
    iszero 1                                #|Evaluate by normal order|#
=> iszero 1                                 ; by def of iszero
=> (λ n. n (λ x. false) true) 1             ; do one step reduction for λ n
=> 1 (λ x. false) true                      ; by def of 1
=> (λ s z. s z) (λ x. false) true           ; application are left associative
=> ((λ s z. s z) (λ x. false)) true         ; do one step reduction for λ s
=> (λ z. (λ x. false) z) true               ; do one step reduction for λ z
=> (λ x. false) true                        ; do one step reduction for λ x
=> false
```

Scheme Programming

Syntax

- For scheme, an expression is either an atom or a list. All expressions use prefix notation.
- Atom: constants (numbers and Booleans) or symbols (variables and inbuilt functions)
- List: be nested to form trees.

'( ) → empty list

list → construct a list from given data

```
; single line comment
#|
comment multiple lines
|#

; Constants
1 ; integer
#t; boolean

; Symbols
(define x 1)
> x
1
> (let ((x 1)) x)
1
```

Semantics

The rules for evaluating Scheme programs:

- A constant evaluates to itself
- A symbol evaluates to its current binding
- A list must be:
  - *A form (e.g. if, lambda), or*
  - *A function application:*
    - the first element of the list must evaluate to a function
    - the remaining elements are the actual parameters

List manipulation

=>(cons 1 (cons 2 (cons 3 '())))
(1 2 3)

- cons: prepend an element to a list
- car: get the head of a list
- cdr: get the tail of a list

→(car '(1 2 3))
1
→(cdr '(1 2 3))
(2 3)

Lambda expression

- Scheme supports implementing anonymous functions that are similar to lambda terms in the lambda calculus.
  - *(lambda (x y) (* x y))*

Control constructs

=> (if condition expr1 expr2)

=> (cond
(pred1 expr1)
(pred2 expr2)
.
.
.
(else exprn))

Locals

Basic `let` skeleton:

```
(let
    ((v1 init1) (v2 init2) ... (vn initn))
    body)
```

To declare locals, use one of the `let` variants:

- `let` : Evaluate all the *inits* in the current environment; the *vs* are bound to fresh locations holding the results.
- `let*` : Bindings are performed sequentially from left to right, and each binding is done in an environment in which the previous bindings are visible.
- `letrec` : The *vs* are bound to fresh locations holding undefined values, the *inits* are evaluated in the resulting environment (in some unspecified order), each *v* is assigned to the result of the corresponding *init*. This is what we need for mutually recursive functions.

Tail Recursion

- Def. it is a recursion in which no additional computation ever follows a recursive call.
- The recursive function returns what the next recursive call returns. **No more computations** after the recursive call.

```
(define (factorial n) ; standard recursion
        (if (= n 0)
           1                          ; return 1
          (* n (factorial (- n 1))) ; return n * factorial (n - 1)
        )
)
(factorial 100000)
```

```
(define (fac x) ; tail recursion
  (letrec
    ( ; rec param
     (fac-tr (lambda (x acc)
       (if (zero? x) acc                    ; return acc
         (fac-tr (- x 1) (* x acc)))))  ; return fac-tr(x - 1, x * acc)
    )
    (fac-tr x 1) ; rec body
  )
)
(fac 100000) ; This one is faster than standard recursion
```

- The most benefit for tail recursion is that the compiler can reuse the current activation record at the time of the recursive call, eliminating the need to allocate a new one, i.e. constant stack space.
- The Scheme compiler will detect the tail recursion fac and performs tail-call elimination to ensure that fac will run in constant stack space.
- Effectively, tail call elimination yields an implementation that is equivalent to one that uses a loop:

```
(define (fact x)
  (let
    (
     (fact_loop (lambda (x acc)
         (do ((i x (- i 1))           ;   for (i = x; i != 0; i--)
              (acc acc (* acc i)))    ;       acc = acc * i;
              ((zero? i) acc)         ;   return acc;
          )))
    )
    (fact_loop x 1)
  )
)
```

Resources

- Installation: follow this link to download DrRacket.
- If you want to learn more about scheme programming, here is an online wiki you could take a look.