Programming Languages

Lambda Calculus and Scheme

CSCI-GA.2110-001
Fall 2013

λ-Calculus

- invented by Alonzo Church in 1932 as a model of computation
- predated any electronic digital computers
- even came before the Turing Machine (A.M. Turing)
- defines the same class of functions as Turing Machines
- later turned out to be Turing complete
- basis for functional languages (e.g., Lisp, Scheme, ML, Haskell)
- typed and untyped variants
- has *syntax* and *reduction rules*

Syntax

We will discuss the *pure*, *untyped* variant of the λ-calculus.

The syntax is simple:

$$
\begin{array}{lcll}
M & ::= & x & \text{variable} \\
 & | & \lambda x \,.\, M & \text{function} \\
 & | & M\, M & \text{function application}
\end{array}
$$

Shorthands:

- We can use parentheses to indicate grouping
- We can omit parentheses when intent is clear
- $\lambda x\, y\, z \,.\, M$ is a shorthand for $\lambda x \,.\, (\lambda y \,.\, (\lambda z \,.\, M))$
- $M_1\, M_2\, M_3$ is a shorthand for $(M_1\, M_2)\, M_3$

Free and bound variables

- In a term $\lambda x \,.\, M$, the scope of x is M.
- We say that x is *bound* in M.
- Variables that are not bound are *free*.

Example:

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

  - The z is free.
  - The last y is free.
  - The x and remaining y are bound.

We can perform α-conversion at will:

$$\lambda y \,.\, (\ldots y \ldots) \quad \longrightarrow_\alpha \quad \lambda w \,.\, (\ldots w \ldots)$$

β-reduction

The main reduction rule in the λ-calculus is function application:

$$(\lambda x\,.\,M)\,N \quad \longrightarrow_\beta \quad [x \mapsto N]M$$

The notation $[x \mapsto N]M$ means:

M, *with all* ***bound*** *occurrences of* x *replaced by* N.

Restriction: N should not have any free variables which are bound in M.

Example:

$$(\lambda x\,.\,(\lambda y\,.\,(x\,y)))\,(\lambda y\,.\,y) \quad \longrightarrow_\beta \quad \lambda y\,.\,((\lambda y.y)\ y)$$

An expression that cannot be β-reduced any further is a *normal form*.

β-reduction

Not everything has a normal form:

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

reduces to itself. Application rule can be applied infinitely.

Evaluation strategies

We have the β-rule, but if we have a complex expression, where should we apply it first?

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

Two popular strategies:

- **normal-order**: Reduce the outermost "redex" first.

  $$[x \mapsto (\lambda x\,.\,x)(\lambda y\,.\,z)](\lambda y\,.\,y\,x\,x) \;\longrightarrow_\beta\; \lambda y\,.\,y\,((\lambda x\,.\,x)(\lambda y\,.\,z))\,((\lambda x\,.\,x)(\lambda y\,.\,z))$$

- **applicative-order**: Arguments to a function evaluated first, from left to right.

  $$(\lambda x\,.\,\lambda y\,.\,y\,x\,x)\,([x \mapsto (\lambda y\,.\,z)]x) \;\longrightarrow_\beta\; (\lambda x\,.\,\lambda y\,.\,y\,x\,x)\,((\lambda y\,.\,z))$$

Evaluation strategies

Some observations:

- Some lambda expressions do not terminate when reduced.
- 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.

Computational power

Fact: The untyped λ-calculus is Turing complete. (Turing, 1937)

But how can this be?

- There are no built-in types other than “functions” (e.g., no booleans, integers, etc.)
- There are no loops
- There are no control structures (e.g., if-then-else, switch)
- There are no recursive definitions
- There are no imperative features (e.g., side effects)

Numbers and numerals

- *number*: an abstract idea
- *numeral*: the representation of a number

Example: 15, fifteen, XV, `0F`

These are different numerals that all represent the same *number*.

Alien numerals:

frobnitz $-$ frobnitz $=$ wedgleb
wedgleb $+$ taksar $=$?

Booleans in the λ-calculus

How can a value of "true" or "false" be represented in the λ-calculus?

Any way we like, as long as we define all the boolean operations correctly.

One reasonable definition:

- `true` takes two values and returns the first
- `false` takes two values and returns the second

$$
\begin{aligned}
\texttt{TRUE} &\equiv \lambda a\,.\,\lambda b\,.\,a \\
\texttt{FALSE} &\equiv \lambda a\,.\,\lambda b\,.\,b \\[1ex]
\texttt{IF} &\equiv \lambda c\,.\,\lambda t\,.\,\lambda e\,.\,(c\;t\;e) \\[1ex]
\texttt{AND} &\equiv \lambda m\,.\,\lambda n\,.\,\lambda a\,.\,\lambda b\,.\,m\,(n\;a\;b)\;b \\
\texttt{OR} &\equiv \lambda m\,.\,\lambda n\,.\,\lambda a\,.\,\lambda b\,.\,m\;a\,(n\;a\;b) \\
\texttt{NOT} &\equiv \lambda m\,.\,\lambda a\,.\,\lambda b\,.\,m\;b\;a
\end{aligned}
$$

Booleans in the λ-calculus

Let's try passing `TRUE` to `IF`.
We'll use `1`,`0` as shorthand to represent λ functions.

Evaluate the expression to `1` if `TRUE`, or `0` otherwise:

$$
\begin{aligned}
\texttt{TRUE} &\equiv \lambda a\,.\,\lambda b\,.\,a \\
\texttt{IF} &\equiv \lambda c\,.\,\lambda t\,.\,\lambda e\,.\,(c\;t\;e)
\end{aligned}
$$

$$
\begin{aligned}
&\lambda c\,.\,\lambda t\,.\,\lambda e\,.\,(c\;t\;e)(\lambda a\,.\,\lambda b\,.\,a)\;1\;0 \\
&\longrightarrow_\beta \lambda t\,.\,\lambda e\,.\,((\lambda a\,.\,\lambda b\,.\,a)\;t\;e)\;1\;0 \\
&\longrightarrow_\beta \lambda e\,.\,((\lambda a\,.\,\lambda b\,.\,a)\;1\;e)\;0 \\
&\longrightarrow_\beta ((\lambda a\,.\,\lambda b\,.\,a)\;1\;0) \\
&\longrightarrow_\beta ((\lambda b\,.\,1)\;0) \\
&\longrightarrow_\beta 1
\end{aligned}
$$

Arithmetic in the λ-calculus

We can represent the number n in the λ-calculus by a function which maps f to f composed with itself n times: $f \circ f \circ \ldots \circ f$.

Some numerals:

$$\begin{aligned}
\ulcorner 0 \urcorner &\equiv \lambda f x \,.\, x \\
\ulcorner 1 \urcorner &\equiv \lambda f x \,.\, f x \\
\ulcorner 2 \urcorner &\equiv \lambda f x \,.\, f (f x) \\
\ulcorner 3 \urcorner &\equiv \lambda f x \,.\, f (f (f x))
\end{aligned}$$

Some operations:

$$\begin{aligned}
\texttt{ISZERO} &\equiv \lambda n \,.\, n \,(\lambda x \,.\, \texttt{FALSE})\, \texttt{TRUE} \\
\texttt{SUCC} &\equiv \lambda n\, f\, x \,.\, f\, (n\, f\, x) \\
\texttt{PLUS} &\equiv \lambda m\, n\, f\, x \,.\, m\, f\, (n\, f\, x) \\
\texttt{MULT} &\equiv \lambda m\, n\, f \,.\, m\, (n\, f) \\
\texttt{EXP} &\equiv \lambda m\, n \,.\, n\, m \\
\texttt{PRED} &\equiv \lambda n \,.\, n\, (\lambda g\, k \,.\, (g \ulcorner 1 \urcorner)\, (\lambda u \,.\, \texttt{PLUS}\, (g\, k) \ulcorner 1 \urcorner)\, k)\, (\lambda v \,.\, \ulcorner 0 \urcorner) \ulcorner 0 \urcorner
\end{aligned}$$

Booleans in the λ-calculus

Let's try passing $\ulcorner 0 \urcorner$ to $\mathtt{SUCC}$:

$$\begin{array}{lcl} \ulcorner 0 \urcorner & \equiv & \lambda f x \,.\, x \\ \mathtt{SUCC} & \equiv & \lambda n\, f\, x \,.\, f\,(n\, f\, x) \end{array}$$

$$\begin{array}{l} \lambda n f x \,.\, f(n f x)(\lambda f x \,.\, x) \\ \longrightarrow_\beta \lambda f x \,.\, f((\lambda f x \,.\, x)\ f x) \\ \longrightarrow_\beta \lambda f x \,.\, f((\lambda x \,.\, x)\ x) \\ \longrightarrow_\beta \lambda f x \,.\, f x \\ \equiv \ulcorner 1 \urcorner \end{array}$$

Recursion

How can we express recursion in the λ-calculus?

Example: the factorial function

$$fact(n) \equiv \texttt{if } n = 0 \texttt{ then } 1 \texttt{ else } n * fact(n-1)$$

In the λ-calculus, we can start to express this as:

$$fact \equiv \lambda n \,.\, (\texttt{ISZERO}\, n) \ulcorner 1 \urcorner (\texttt{MULT}\, n\, (fact\, (\texttt{PRED}\, n)))$$

But we need a way to give the factorial function a name.

Idea: Pass in $fact$ as an extra parameter somehow:

$$\lambda fact \,.\, \lambda n \,.\, (\texttt{ISZERO}\, n) \ulcorner 1 \urcorner (\texttt{MULT}\, n\, (fact\, (\texttt{PRED}\, n)))$$

We want the *fix-point* of this function:

$$\texttt{FIX}(f) \equiv f(\texttt{FIX}(f))$$

Fix point combinator, rationale

Definition of a fix-point operator:

$$\mathtt{FIX}(f) \equiv f(\mathtt{FIX}(f))$$

One step of `fact` is: $\lambda f\,.\,\lambda x\,.\,(\mathtt{ISZERO}\,x)\,\ulcorner 1 \urcorner\,(\mathtt{MULT}\,x\,(f\,(\mathtt{PRED}\,x)))$

Call this F. If we apply FIX to this, we get

$$\begin{aligned}
\mathtt{FIX}(F)(n) &= F\,(\mathtt{FIX}(F))\,(n)\\
\mathtt{FIX}(F)(n) &= \lambda x\,.\,(\mathtt{ISZERO}\,x)\,\ulcorner 1 \urcorner\,(\mathtt{MULT}\,x\,(\mathtt{FIX}(F)\,(\mathtt{PRED}\,x)))(n)\\
\mathtt{FIX}(F)(n) &= (\mathtt{ISZERO}\,n)\,\ulcorner 1 \urcorner\,(\mathtt{MULT}\,n\,(\mathtt{FIX}(F)\,(\mathtt{PRED}\,n)))
\end{aligned}$$

If we rename "$\mathtt{FIX}(F)$" as "`fact`", we have exactly what we want:

$$\mathtt{fact}(n) = (\mathtt{ISZERO}\,n)\,\ulcorner 1 \urcorner\,(\mathtt{MULT}\,n\,(\mathtt{fact}\,(\mathtt{PRED}\,n)))$$

Conclusion: $\mathtt{fact} = \mathtt{FIX}(F)$. (But we still need to define FIX.)

Fix point combinator, definition

There are many fix-point combinators. Here is the simplest, due to Haskell Curry:

$$\texttt{FIX} = \lambda f\,.\,(\lambda x\,.\,f\,(x\,x))\,(\lambda x\,.\,f\,(x\,x))$$

Let's prove that it actually works:

$$\begin{aligned}
\texttt{FIX}(g) = \quad & (\lambda f\,.\,(\lambda x\,.\,f\,(x\,x))\,(\lambda x\,.\,f\,(x\,x)))\,g \\
\longrightarrow_\beta \quad & ((\lambda x\,.\,g\,(x\,x))\,(\lambda x\,.\,g\,(x\,x))) \\
\longrightarrow_\beta \quad & g\,((\lambda x\,.\,g\,(x\,x))\,(\lambda x\,.\,g\,(x\,x)))
\end{aligned}$$

But this is exactly $g(\texttt{FIX}(g))$!

Scheme overview

- related to Lisp, first description in 1975
- designed to have clear and simple semantics (unlike Lisp)
- statically scoped (unlike Lisp)
- dynamically typed
  - types are associated with values, not variables
- functional: first-class functions
- garbage collection
- simple syntax; lots of parentheses
  - homogeneity of programs and data
- continuations
- hygienic macros

A sample Scheme session

```
(+ 1 2)
⇒ 3
(1 2 3)
⇒ procedure application: expected procedure; given: 1
a
⇒ reference to undefined identifier: a
(quote (+ 1 2))   ; a shorthand is '(+ 1 2)
⇒ (+ 1 2)
(car '(1 2 3))
⇒ 1
(cdr '(1 2 3))
⇒ (2 3)
(cons 1 '(2 3))
⇒ (1 2 3)
```

Uniform syntax: lists

- expressions are either atoms or lists
- atoms are either constants (e.g., numeric, boolean, string) or symbols
- lists nest, to form full trees
- syntax is simple because programmer supplies what would otherwise be the internal representation of a program:

```
(+ (* 10 12) (* 7 11))   ; means (10*12 + 7*11)
```

- a program is a list:

```
(define (factorial n)
        (if (= n 0)
            1
            (* n (factorial (- n 1)))))
```

Scheme supports λ expressions

```
(lambda (x) (+ x x))
⇒ #<procedure>

((lambda (x) (+ x x)) 5)
⇒ 10

(define doubleit (lambda (x) (+ x x)))
(doubleit 5)
⇒ 10

(define quadrupleit (lambda (x) (* (doubleit x) 2)))
(quadrupleit 4)
⇒ 16

(define doubleit (lambda (x) 7))
(quadrupleit 4)
⇒ 14
```

List manipulation

Three primitives and one constant:

- `car`: get head of list
- `cdr`: get rest of list
- `cons`: prepend an element to a list
- `'()` null list

Add equality (`=` or `eq`) and recursion, and you've got yourself a universal model of computation

Rules of evaluation

- a *number* evaluates to itself
- an *atom* evaluates to its current binding
- a *list* is a computation:
  - must be a form (e.g., if, lambda), or
  - first element must evaluate to an operation
  - remaining elements are actual parameters
  - result is the application of the operation to the evaluated actuals

Quoting data

Q: If every list is a computation, how do we describe data?

A: Another primitive: `quote`

```
(quote (1 2 3 4))
⇒ (1 2 3 4)
(quote (Baby needs a new pair of shoes)
⇒ (Baby needs a new pair of shoes)
'(this also works)
⇒ (this also works)
```

List decomposition

```
(car '(this is a list of symbols))
⇒ this

(cdr '(this is a list of symbols))
⇒ (is a list of symbols)

(cdr '(this that))
⇒ (that) ; a list

(cdr '(singleton))
⇒ () ; the empty list

(car '())
⇒ car: expects argument of type <pair>; given ()
```

List building

```
(cons 'this '(that and the other))
⇒ (this that and the other)
(cons 'a '())
⇒ (a)
```

useful shortcut:

```
(list 'a 'b 'c 'd 'e)
⇒ (a b c d e)
```

equivalent to:

```
(cons 'a
      (cons 'b
            (cons 'c
                  (cons 'd
                        (cons 'e '())))))
```

List decomposition shortcuts

Operations like:

```
(car (cdr xs))
(cdr (cdr (cdr ys)))
```

are common. Scheme provides shortcuts:

`(cadr xs)`	is	`(car (cdr xs))`
`(cdddr xs)`	is	`(cdr (cdr (cdr xs)))`

Up to 4 `a`'s and/or `d`'s can be used.

What lists are made of

`(cons 'a '(b))`	$\Rightarrow$	`(a b)`	*a list*
`(car '(a b))`	$\Rightarrow$	`a`	
`(cdr '(a b))`	$\Rightarrow$	`(b)`	
`(cons 'a 'b)`	$\Rightarrow$	`(a . b)`	*a dotted pair*
`(car '(a . b))`	$\Rightarrow$	`a`	
`(cdr '(a . b))`	$\Rightarrow$	`b`	

A list is a special form of dotted pair, and can be written using a shorthand:

`'(a b c)` is shorthand for `'(a . (b . (c . ())))`

We can mix the notations:

`'(a b . c)` is shorthand for `'(a . (b . c))`

A list not ending in `'()` is an *improper list*.

Booleans

Scheme has true and false values:

- `#t` – true
- `#f` – false

However, when evaluating a condition (e.g., in an `if`), any value not equal to `#f` is considered to be true.

Simple control structures

- Conditional

```
(if condition expr1 expr2)
```

- Generalized form

```
(cond
  (pred1 expr1)
  (pred2 expr2)
  ...
  (else exprn))
```

Evaluate the `pred`'s in order, until one evaluates to true. Then evaluate the corresponding `expr`. That is the value of the `cond` expression.

`if` and `cond` are not regular functions

Global definitions

`define` is also special:

```
(define (sqr n) (* n n))
```

The body is not evaluated; a binding is produced: `sqr` is bound to the body of the computation:

```
(lambda (n) (* n n))
```

We can `define` non-functions too:

```
(define x 15)
(sqr x)
⇒ 225
```

`define` can only occur at the top level, and creates global variables.

Recursion on lists

```
(define (member elem lis)
   (cond
      ((null? lis) #f)
      ((= elem (car lis)) lis)
      (else (member elem (cdr lis)))))
```

Note: every non-false value is true in a boolean context.

Convention: return rest of the list, starting from `elem`, rather than `#t`.

Standard predicates

If variables do not have associated types, we need a way to find out what a variable is holding:

- `symbol?`
- `number?`
- `pair?`
- `list?`
- `null?`
- `zero?`

Different dialects may have different naming conventions, e.g., `symbolp`, `numberp`, etc.

Functional arguments

```
(define (map fun lis)
   (cond
      ((null? lis) '())
      (else (cons (fun (car lis))
                  (map fun (cdr lis))))))

(map sqr (map sqr '(1 2 3 4)))
⇒ (1 16 81 256)
```

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

> "A Scheme implementation is properly tail-recursive if it supports an unbounded number of active tail calls."

```
(define (factorial n)
  (if (zero? n) 1
      (* n (factorial (- n 1))))) ; not tail recursive
                                  ; stack grows to size n
(define (fact-iter prod count var)
  (if (> count var) prod
      (fact-iter (* count prod)  ; tail recursive
                 (+ count 1)     ; implemented as loop
                 var)))
(define (factorial n) (fact-iter 1 1 n)) ; OK
```

Acid test for proper tail-recursion

```
(define (increment x) (increment x))
(increment 1)
```

An implementation utilizing tail recursion optimization will compute indefinitely, but will not run out of resources. Otherwise, an upper stack memory limit will be reached (stack overflow) and the program will terminate.

Most functional languages optimize, but so do some imperative languages. For example when compiled on the "Any CPU" target and run on a 64-bit version of the .NET runtime, C# will be optimized for proper tail-recursion. (The optimization is performed at run time, not compile time.)