G22.2110-003 Programming Languages - Fall 2012

Lecture 5

Thomas Wies

New York University

Review

Last week

- Subprograms
- Calling Sequences
- Parameter Passing
- Recursion

Outline

- Function Languages
- Lambda Calculus
- Scheme

Sources:

- Jung, Achim. “A short introduction to the Lambda Calculus”, available from the class website.
- Ch. 5 of Types and Programming Languages by Benjamin Pierce, MIT Press, 2002
- PLP, ch. 10

Functional Programming

Functional Programming refers to a programming style in which every procudure is *functional*, i.e. it computes a function of its inputs with no side effects.

Functional programming languages are based on this idea, but they also provide a number of interesting features that are often missing in imperative languages.

One of the most important and interesting features of functional programming languages is that functions are *first-class* values.

The means that programs can create new functions at run-time.

This can be leveraged to build powerful *higher-order* functions: a higher-order function either takes a function as an argument or returns a function as a result (or both).

Functional languages draw heavily on the *λ-calculus* for inspiration.

λ-Calculus

- invented by Alonzo Church in 1932 as a model of computation
- 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:

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

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 consistent renaming of bound variables at will (α-conversion):

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

β-reduction

The main reduction rule in the λ-calculus is *β-reduction*, which is just function application.

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

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

M, with all free occurrences of x replaced by N.

Restriction: N should not have any free variables which are bound in M. We can always use α-conversion to comply with this restriction.

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 has been reduced to *normal form*.

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.

$$\begin{aligned}(\lambda x.\, \lambda y.\, y\, x\, x)\, ((\lambda x.\, x)(\lambda y.\, z)) \; &\longrightarrow_\beta \\ &[x \mapsto (\lambda x.\, x)(\lambda y.\, z)](\lambda y.\, y\, x\, x) = \\ &\lambda y.\, y\, ((\lambda x.\, x)(\lambda y.\, z))\, ((\lambda x.\, x)(\lambda y.\, z))\end{aligned}$$

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

$$\begin{aligned}(\lambda x.\, \lambda y.\, y\, x\, x)\, ((\lambda x.\, x)(\lambda y.\, z)) \; &\longrightarrow_\beta \\ &(\lambda x.\, \lambda y.\, y\, x\, x)\, ([x \mapsto (\lambda y.\, z)]x) \; = \\ &(\lambda x.\, \lambda y.\, y\, x\, x)\, ((\lambda y.\, z))\end{aligned}$$

Order of Evaluation: Does it Matter?

Church-Rosser Theorem

If a term M can be reduced (in 0 or more steps) to terms N and P, then there exists a term Q such that both N and P can be reduced to Q.

$$\begin{tikzcd}
 & M \arrow[dl, "*"'] \arrow[dr, "*"] & \\
N \arrow[dr, dashed, "*"'] & & P \arrow[dl, dashed, "*"] \\
 & Q &
\end{tikzcd}$$

Order of Evaluation

The property expressed in the Church-Rosser theorem is called *confluence*. We say that β-reduction is *confluent*.

Corollary

Every term has at most one normal form.

Order of Evaluation

The property expressed in the Church-Rosser theorem is called *confluence*. We say that β-reduction is *confluent*.

Corollary

Every term has at most one normal form.

Why at most one?

Order of Evaluation

The property expressed in the Church-Rosser theorem is called *confluence*. We say that β-reduction is *confluent*.

Corollary

Every term has at most one normal form.

Why at most one?

There are some terms with no normal form!

Order of Evaluation

The property expressed in the Church-Rosser theorem is called *confluence*. We say that β-reduction is *confluent*.

Corollary

Every term has at most one normal form.

Why at most one?

There are some terms with no normal form!

Example:

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

Computability

The notion of *computability* relies on formal models of computation.

Many formal models have been proposed:

1. General recursive functions defined by means of an equation calculus (Gödel-Herbrand-Kleene)
2. μ-recursive functions and partial recursive functions (Gödel-Kleene)
3. Functions computable by Turing machines (Turing)
4. Functions defined from canonical deduction systems (Post)
5. Functions given by certain algorithms over finite alphabets (Markov)
6. Universal Register Machine-computable functions (Shepherdson-Sturgis)
7. Any function you can write in your favorite programming language

Fundamental Result

All of these (and many other) models of computation are equivalent. That is, they give rise to the same class of computable functions.

Any such model of computation is said to be *Turing complete*.

Computational power

Theorem

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 imperative features
- There are no recursive definitions

Numbers and numerals

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

Example

Consider the following expressions:

- 15
- fifteen
- *XV*
- `0F`

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

Booleans in the λ-calculus

How can we represent `TRUE` and `FALSE` in the λ-calculus?

One reasonable definition:

- `true` is a function that takes two values and returns the first
- `false` is a function that takes two values and returns the second

$$
\begin{array}{lcl}
\texttt{true} & \equiv & \lambda\, a\, b.\, a \\
\texttt{false} & \equiv & \lambda\, a\, b.\, b \\
\\
\texttt{and} & \equiv & \lambda\, m\, n.\, \lambda\, a\, b.\, m\,(n\, a\, b)\, b \\
\texttt{or} & \equiv & \lambda\, m\, n.\, \lambda\, a\, b.\, m\, a\,(n\, a\, b) \\
\texttt{not} & \equiv & \lambda\, m.\, \lambda\, a\, b.\, m\, b\, a
\end{array}
$$

Arithmetic in the λ-calculus: Church Numerals

The number n is represented in the λ-calculus by a function which maps a successor function s and a zero z to n applications of s to z: $s \circ s \circ \ldots \circ s(z)$.

Some numerals:

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

Some operations:

$$\begin{aligned} \texttt{iszero} &\equiv \lambda\, n.\, n\,(\lambda\, x.\, \texttt{false})\, \texttt{true} \\ \texttt{succ} &\equiv \lambda\, n\, s\, z.\, s\,(n\, s\, z) \\ \texttt{plus} &\equiv \lambda\, m\, n\, s\, z.\, m\, s\,(n\, s\, z) \\ \texttt{mult} &\equiv \lambda\, m\, n\, s.\, m\,(n\, s) \\ \texttt{exp} &\equiv \lambda\, m\, n.\, n\, m \\ \texttt{pred} &\equiv \lambda\, n.\, n\,\big(\lambda\, g\, k.\, (g\, \ulcorner 1 \urcorner)\,(\lambda\, u.\, \texttt{plus}\,(g\, k)\, \ulcorner 1 \urcorner)\, k\big)\,(\lambda\, v.\, \ulcorner 0 \urcorner)\, \ulcorner 0 \urcorner \end{aligned}$$

Recursion

How can we express recursion in the λ-calculus?

Example: the factorial function

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

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

$$fact = \lambda\, n.\, (\texttt{iszero}\, n) \ulcorner 1 \urcorner (\texttt{mult}\, n\, (fact\, (\texttt{pred}\, n)))$$

Recursion

How can we express recursion in the λ-calculus?

Example: the factorial function

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

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

$$\mathit{fact} = \lambda\, n.\, (\texttt{iszero}\, n) \ulcorner 1 \urcorner (\texttt{mult}\, n\, (\mathit{fact}\, (\texttt{pred}\, n)))$$

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

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

$$\mathit{FactFun} = \lambda\, \mathit{fact}.\, \lambda\, n.\, (\texttt{iszero}\, n) \ulcorner 1 \urcorner (\texttt{mult}\, n\, (\mathit{fact}\, (\texttt{pred}\, n)))$$

Now, the function *fact* we are looking for satisfies the following equation:

$$\mathit{fact} = \mathit{FactFun}\, \mathit{fact}$$

Fixpoint Combinator

We have:

$$\mathit{FactFun} = \lambda\, \mathit{fact}.\, \lambda\, n.\, (\texttt{iszero}\, n)\, \ulcorner 1 \urcorner\, (\texttt{mult}\, n\, (\mathit{fact}\, (\texttt{pred}\, n)))$$

and we are looking for a solution to:

$$\mathit{fact} = \mathit{FactFun}\, \mathit{fact}$$

A solution to this equation is called a *fixpoint* for *FactFun*.

Amazingly, we can find a solution not only for *fact* but for *any* function.

A *fixpoint combinator* Y is a term that satisfies the following equation for any M:

$$Y\, M = M\, (Y\, M)$$

Fixpoint combinator, definition

There are many fixpoint combinators. Here is the simplest, due to Haskell Curry:

$$Y = \lambda f.\,(\lambda x.\,f\,(x\,x))\,(\lambda x.\,f\,(x\,x))$$

Let's check that it actually works:

$$\begin{aligned} YM = \quad & (\lambda f.\,(\lambda x.\,f\,(x\,x))\,(\lambda x.\,f\,(x\,x)))\,M \\ \longrightarrow_\beta \quad & (\lambda x.\,M\,(x\,x))\,(\lambda x.\,M\,(x\,x)) \\ \longrightarrow_\beta \quad & M\,((\lambda x.\,M\,(x\,x))\,(\lambda x.\,M\,(x\,x))) \end{aligned}$$

But this is exactly $M\,(Y\,M)$!

Therefore, we can define $\mathit{fact} = Y\,\mathit{FactFun}$

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

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 numeric or symbols
- lists nest, to form full trees
- syntax is simple: all expressions use *prefix* notation (i.e. 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 (fact n)
        (if (eq? n 0)
            1
            (* n (fact (- n 1)))))
```

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 a function
  - remaining elements are actual parameters
  - result is the application of the function to the evaluated actuals
  - used evaluation strategy: applicative-order

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)
```

Booleans

SCHEME has true and false values:

- `#t` – true
- `#f` – false

However, when evaluating a condition, 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.

List manipulation

Three primitives and one constant:

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

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

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 ys)))
```

Up to 4 (total) `a`'s and `d`'s can be used.

What lists are made of

```
(cons 'a '(b))    ⇒   (a b)       a list
(car '(a b))      ⇒   a
(cdr '(a b))      ⇒   (b)

(cons 'a 'b)      ⇒   (a . b)     a dotted pair
(car '(a . b))    ⇒   a
(cdr '(a . b))    ⇒   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))`

Recursion on lists

```
(define (member elem lis)
   (cond
      ((null? lis) #f)
      ((eq? 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 '(1 2 3 4))
⇒ (1 4 9 16)
(map sqr (map sqr '(1 2 3 4)))
⇒ (1 16 81 256)
```

In fact, there is an inbuilt version of `map` that is even more general:

```
(map * '(1 2 3 4) '(1 2 3 4))
⇒ (1 4 9 16)
```

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

What is a tail-recursive procedure?

One in which no computation is done after any recursive call.

Is the following tail-recursive?

```
(define (fact n)
  (if (zero? n) 1
      (* n (fact (- n 1)))))
```

No, it is not, but we can define a tail-recursive version:

```
(define (fact n)
  (letrec
    ((fact-aux (lambda (n prod)
       (if (zero? n) prod
           (fact-aux (- n 1) (* prod n))))))
    (fact-aux n 1)))
```

Tail recursion

What is a tail-recursive procedure?

One in which no computation is done after any recursive call.

Is the following tail-recursive?

```
(define (fact n)
  (if (zero? n) 1
      (* n (fact (- n 1)))))
```

No, it is not, but we can define a tail-recursive version:

```
(define (fact n)
  (letrec
    ((fact-aux (lambda (n prod)
       (if (zero? n) prod
           (fact-aux (- n 1) (* prod n))))))
    (fact-aux n 1)))
```

Many versions of SCHEME ensure that tail-recusrion is implemented efficiently (i.e. no additional stack space for recursive calls)