Slides to Accompany *Programming Languages and Methodologies*

R. J. Schalkoff

Chapter 8, Part 1: Lambda Calculus

The lambda Calculus and Functional Programming

- **the lambda calculus**:
  - A basis for computation based upon defining and using functions (functional programming).
  - A very simple syntax and very powerful semantics.
  - Resulted from the work of Alonzo Church.

- Church's work was motivated by the desire to create a calculus that incorporated the computational aspect of functions.
- The lambda calculus allows functions to be applied to themselves. Thus, recursion occurs naturally in the lambda calculus (and consequently in functional programming).
- We consider Lisp (and ML and OCAML) as 'sugared' implementations of and extensions to the lambda calculus.
- The lambda calculus has a very simple syntax, consisting of only 4 productions.

More History: The Roots of Lisp

Big Result:

- In 1960, John McCarthy showed how a small number of (possibly recursive) functions and a data structure (a list) for both code and data could form the basis for a programming language. *You (can and) should read this paper.*
- A remarkable corollary to this result is the fact that these functions (alone) allow one to write an interpreter for Lisp **in Lisp**.
- This is an example of the *self-definition* of the semantics of a programming language.
- Lisp has had a significant impact on the subsequent development of other functional languages, such as ML and Haskell.

The Turing Machine Analogy to Computation

- Alan Turing: proposed an idealized mathematical structure for a writing machine. The Turing machine is one of the key abstractions used in modern computability theory (basically, the study of what computers can and cannot do).
- The Turing machine provides a formalism for expressing (procedural) computation.
- By analogy, the lambda calculus provides a formalism for the expression of a computation in a purely functional language.

Review: Relations and Functions

- A (binary) relation is a mapping between elements of 2 sets:

$$R: \ A \to B$$

- A function is a special case of a relation:

$$f: \ X \to Y$$

  such that for each $x \in X$ we can specify a unique element in Y, denoted y or $f(x)$.

- Functions can return objects (a larger class than numbers) which can be almost anything, e.g., sets, lists, strings, function definitions, void, ...

Examples

- Consider the (numerical) function *square*, with signature:

$$square : \ Integer \rightarrow Integer$$

  which is commonly written as $square(n) = n^2$. In this case, the function is given a name (*square*).

- Alternately, consider the definition of an anonymous function which has the same input-output mapping. In the syntax of the λ calculus (ignoring the n^2 notation, which is not permitted by the syntax) this function is defined using the abstraction:

$$\lambda n.n^2$$

Polymorphic Example

- The function abstraction for the identity mapping is:

$$(\lambda n.n)$$

- For example, the combination

$$((\lambda n.n)\ 4)$$

returns the value 4, whereas

$$((\lambda n.n)\ 'hi\ mom')$$

returns the string *'hi mom'*.

- The function behaves similarly with different types. This is called *polymorphic* behavior.

4 Productions Go a Long Way

<expression> ::=

<variable>

| **<constant>**

| **(<expression> <expression>)**

| **(λ <variable> . <expression>)**

Notes:

- In the syntactic specification, the parentheses are not for clarity of reading; *they are part of the syntax.*
- The last two productions are self-embedding or recursive.
- Variables are denoted by *lower case* letters.

Remarks on the lambda Calculus

- The role of a variable in the λ calculus is quite different from the notion of a variable as used in an imperative language.
- The set of constants includes the names of (assumed) built-in functions, such as **add** or $+$.

The Combination (Prelude to Reduction)

<expression> ::= (<expression1> <expression2>)

- `expression1` and `expression2` are each derived from `<expression>`.
- The corresponding semantics of this string indicate application of `expression1` to `expression2`.
- Often, `expression1` is referred to as the **operator** (or **rator**), and `expression2` is referred to as the **operand** (or **rand**).
- `expression1` must be (or evaluate to) a function, either predefined (i.e., a constant) or an abstraction.

Reduction Examples #1(a)-1(b)

Suppose we are given:

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

- Is this expression syntactically correct?
- If so, can it be reduced?

Now consider:

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

- Is this expression syntactically correct?
- If so, can it be reduced?

Reduction Example #2

Reduce:

$$(((\lambda f.(\lambda x.(f(f\ x))))\ square)\ 2)$$

where *square* is assumed to be the predefined squaring function.

Steps:

- Recognize this string is an (overall) *combination*; specifically $((\lambda f.(\lambda x.(f(f\ x))))\ square)$ applied to 2
- $((\lambda f.(\lambda x.(f(f\ x))))\ square)$ is also a combination. Evaluation of this part yields:

$$((\lambda f.(\lambda x.(f(f\ x))))\ square) \Rightarrow (\lambda x.(square\ (square\ x)))$$

- This resulting *abstraction* is then applied to 2, yielding

$$((\lambda x.(square(square\, x)))\, 2) \Rightarrow (square\, (square\, 2)) \Rightarrow (square\, 4) \Rightarrow 16$$

- So the overall reduction is:

$$(((\lambda f.(\lambda x.(f(f\ x))))\ square)\ 2) \Rightarrow$$
$$((\lambda x.(square(square\ x)))\ 2) \Rightarrow$$
$$(square\ (square\ 2)) \Rightarrow (square\ 4) \Rightarrow 16$$

Prelude to Lisp: Reduction (Evaluation) Examples

A lambda function with the (extended syntax) abstraction:

$$(\lambda x.(+\ 3\ x))$$

Could be written in Lisp as:

```
> (lambda (x) (+ 3 x))
#<closure :lambda (x) (+ 3 x)>
```

Notice Lisp calls this a *lexical closure*.

Consider the lambda calculus combination:

$$((\lambda x.(+\ 3\ x))\ 9)$$

In Lisp this becomes:

```
> ((lambda (x) (+ 3 x)) 9)
12
```

Consider the use of the (extended syntax) abstraction:

$$(\lambda n.n^3)$$

The corresponding Lisp is:

```
> ((lambda (n) (* n n n)) 3)
27
```

The λ-calculus and ocaml

$$(\lambda p.sqrt\ p)$$

```
# function p -> sqrt p;;
- : float -> float = <fun>

# ((function p -> sqrt p) 2.0);;
- : float = 1.4142135623730951
```

Prelude to naming functions in ocaml

Consider the use of the (extended syntax) *abstraction*:

$$(\lambda n.n^3)$$

The corresponding ocaml is:

```
# fun n -> n*n*n;;
- : int -> int = <fun>
# let f1=  fun n -> n*n*n;;
val f1 : int -> int = <fun>
# f1 3;;
- : int = 27
#
```

Consider the lambda calculus (extended syntax) *combination*:

$$((\lambda x.(+\ 3\ x))\ 9)$$

In ocaml this becomes:

```
# fun x -> 3+x;;
- : int -> int = <fun>
# let f2=fun x -> 3+x;;
val f2 : int -> int = <fun>
# f2 9;;
- : int = 12
#
```

Our Old Friend

REcall:

$$(((\lambda f.(\lambda x.(f(f\ x))))\ square)\ 2)$$

where *square* is assumed to be the predefined squaring function.

First, we define and explore $square(x)$, first as a lambda function:

```
# function s -> s*s;;
- : int -> int = <fun>

# ( (function s -> s*s) 4);;
- : int = 16
```

Now, without 'abusing' let, let's give the lambda function a name and use it:

```
# let square = function s -> s*s;;
val square : int -> int = <fun>

# square 4;;
- : int = 16
# square(4);;
- : int = 16
# (square 4);;
- : int = 16
#
```

Now, for our example, it gets a little complicated ...

```
;; first a warmup--
# function f -> (function x -> (f x));;
- : ('a -> 'b) -> 'a -> 'b = <fun>

;; now showing the returned function--
# ((function f -> (function x -> (f x))) square);;
- : int -> int = <fun>

;; now use it--
# (((function f -> (function x -> (f x))) square) 2);;
- : int = 4

;; now the more specific result we wanted (the previous reduction)--
# function f -> (function x -> (f (f x)));;
- : ('a -> 'a) -> 'a -> 'a = <fun>
#  (((function f -> (function x -> (f (f x)))) square) 2);;
- : int = 16
```