

Programming Languages, CPSC 604—Slides 6
Lambda Calculus

Jaakko Järvi

February 12, 2009

Outline

- 1 Introduction
- 2 Programming with lambda calculus
- 3 Recursion
- 4 Summarizing

Lambda calculus — what is it?

- Formal mathematical system by Alonzo Church (from 1920s)
 - for studying functions, function application, recursion
 - goal was to use it as foundation of mathematics
- **Lambda definability**
 - A function is lambda definable if it can be expressed using lambda calculus
 - Church's thesis: all intuitively computable functions are lambda definable
 - Set of Lambda definable functions = Turing computable functions = General Recursive Functions

Lambda calculus — what's in it for us?

- Mathematical system to study programming languages in a pure, simplified form
- Fairly direct mapping to evaluation of functional languages.
- Lambda calculi can be seen as prototype programming languages.
- Variations
 - Implicit and explicit typing
- More complex systems built on top of simply typed lambda calculus
 - System F — for studying polymorphism
 - $\lambda_{<}$ — for studying subtyping
 - System $F_{<}$ — for studying polymorphism together with subtyping
 - ...
 - Used widely in design, specification, and implementation of programming languages, and in study of type systems

Lambda calculus — definition

- Abstract syntax

$$M ::= x \mid M M \mid \lambda x. M$$

- M is a lambda term
- An infinite set of variables x, y, z, \dots is assumed
- $M N$ is an **application**
 - “function M applied to the argument N ”
- $\lambda x. M$ is an **abstraction**
 - “function that assigns to x the value M ”

About syntax

- Extremely simple
 - only three constructs
- Lambda functions are *anonymous*
- Simple syntactic sugar and conventions
 - $M N_1 \dots N_k$ means $(\dots ((M N_1) N_2) \dots N_k)$
 - Function application groups from left to right
 - $\lambda x. x y$ is $(\lambda x. (x y))$
 - Function application has higher precedence (= lambda abstraction scope reaches as far as possible)
 - $\lambda x_1 x_2 \dots x_k. M$ means $\lambda x_1. (\lambda x_2. (\dots (\lambda x_k. (M)) \dots))$

Examples

- Some lambda terms

 x $\lambda x.x$ $\lambda f.\lambda g.\lambda x.f(g\ x)$

- Compare lambda and, say, Haskell constructs:

 $\lambda x.M$ $(\backslash x \rightarrow M)$

- Aside: where does the λ (lambda) come from ?

Variable binding

- **Free occurrence** of variable in some expression means that variable not defined in that expression
- **Bound occurrence** = not free
- λ is a *binding operator*
 - Binds a variable in the scope where the *lambda* reaches
- Examples:
 - $\lambda x.M$
 - x bound in $\lambda x.M$
 - and in particular in M
 - red x is the **binding occurrence** of x
 - $\lambda x.x y$
 - y is free

Variable binding — precise definition

- $FV(M)$ defines the set of free variables in the term M

$$FV(x) = \{x\}$$

$$FV(MN) = FV(M) \cup FV(N)$$

$$FV(\lambda x.M) = FV(M) \setminus \{x\}$$

- Spot free and bound occurrences here:

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

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

- **Combinator** — a term with no free variables (also **closed term**)

About bound variables

- Bound variable is a *placeholder*
 - name of the bound variable is insignificant
- $\lambda x.x$ defines the same function as $\lambda y.y$
- Two lambda terms that differ only on the names of bound variables are said to be **α -equivalent** ($=_\alpha$)
- Equational axiom:

$$\lambda x.M = \lambda y.[y/x]M$$

where y does not appear in M

- substitution applies to free occurrences only
- Also, **α -conversion**: Rewriting a term changing bound variable names

Substitution

- Lambda calculus computes by substitution
- Defined by the equational axiom:

$$(\lambda x.M)N = [N/x]M$$

- β -equivalence
- Think of this as invoking a function
 - $(\lambda x.M)$ is the function
 - N is the argument
 - substitution takes care of parameter passing

Inductive definition of substitution in λ calculus

$$[N/x]x = N$$

$$[N/x]y = y, y \text{ any variable different from } x$$

$$[N/x](M_1 M_2) = ([N/x]M_1) ([N/x]M_2)$$

$$[N/x](\lambda x.M) = \lambda x.M$$

$$[N/x](\lambda y.M) = \lambda y.([N/x]M), y \text{ not free in } N$$

- Alpha renaming can be applied freely, variables drawn from the infinite pool of variable names
- Examples:

$$[z/x]x$$

$$[z/x](\lambda x.x x)$$

$$[z/x](\lambda y.y x)$$

$$[z/x](\lambda z.x z)$$

Inductive definition of substitution in λ calculus

$$[N/x]x = N$$

$$[N/x]y = y, y \text{ any variable different from } x$$

$$[N/x](M_1 M_2) = ([N/x]M_1) ([N/x]M_2)$$

$$[N/x](\lambda x.M) = \lambda x.M$$

$$[N/x](\lambda y.M) = \lambda y.([N/x]M), y \text{ not free in } N$$

- Alpha renaming can be applied freely, variables drawn from the infinite pool of variable names
- Examples:

$[z/x]x$	z
$[z/x](\lambda x.x x)$	$\lambda x.x x$
$[z/x](\lambda y.y x)$	$\lambda y.y z$
$[z/x](\lambda z.x z)$	$\lambda a.z a$

Computing in λ calculus

- Reading β -equivalence from left to right gives β -reduction

$$(\lambda x.M)N \rightarrow [N/x]M$$

- Similarly for α -reduction
- Computing with λ calculus is then a series of β - and α -reductions
 - maybe towards a simpler term

Terminology and notation

- $M \rightarrow N$
 - M β -reduces to N
- **Redex** (reducible expression)

$(\lambda x.M)N$

- **Normal form**
 - a term that cannot be reduced further
 - For example: $\lambda x.x$

Example reduction

- We assume a little extension to lambda calculus: operator $+$

$$(\lambda f. \lambda x. f (f x)) (\lambda y. y + 1) 2$$

Example reduction

- We assume a little extension to lambda calculus: operator $+$

$$(\lambda f.\lambda x.f (f x)) (\lambda y.y + 1) 2$$

$$\rightarrow (\lambda x.(\lambda y.y + 1)((\lambda y.y + 1) x)) 2$$

Example reduction

- We assume a little extension to lambda calculus: operator $+$

$$(\lambda f.\lambda x.f (f x)) (\lambda y.y + 1) 2$$

$$\rightarrow (\lambda x.(\lambda y.y + 1)((\lambda y.y + 1) x)) 2$$

$$\rightarrow (\lambda x.(\lambda y.y + 1)(x + 1)) 2$$

Example reduction

- We assume a little extension to lambda calculus: operator $+$

$$(\lambda f. \lambda x. f (f x)) (\lambda y. y + 1) 2$$

$$\rightarrow (\lambda x. (\lambda y. y + 1) ((\lambda y. y + 1) x)) 2$$

$$\rightarrow (\lambda x. (\lambda y. y + 1) (x + 1)) 2$$

$$\rightarrow (\lambda x. (x + 1 + 1)) 2$$

Example reduction

- We assume a little extension to lambda calculus: operator $+$

$$(\lambda f. \lambda x. f (f x)) (\lambda y. y + 1) 2$$

$$\rightarrow (\lambda x. (\lambda y. y + 1) ((\lambda y. y + 1) x)) 2$$

$$\rightarrow (\lambda x. (\lambda y. y + 1) (x + 1)) 2$$

$$\rightarrow (\lambda x. (x + 1 + 1)) 2$$

$$\rightarrow (2 + 1 + 1)$$

Confluence

- In general, there can be more than one redex in a given expression
- Consider an alternative reduction order for the previous example:
 - $(\lambda x. (\lambda y. y + 1) (x + 1)) 2$
 - $(\lambda y. y + 1) (2 + 1)$
 - $((2 + 1) + 1)$
 - $(2 + 1 + 1)$
- **Confluence**: evaluation order is not significant for the final value
- **Consequence**: there can be only one normal form of a given expression

Confluence

- Notation: $M \rightarrow^* N$ means M reduces to N in zero or more steps
- Confluence
 - If $M \rightarrow^* N$ and $M \rightarrow^* N'$, then there exists some P such that $N \rightarrow^* P$ and $N' \rightarrow^* P$
- **Strong normalization property**
 - Every term reduces to a normal form in finite number of steps
 - Question: are all lambda terms strongly normalizable

Example

- No

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

- We do have strong normalization, however, in *simply-typed lambda calculus*
- Note, different evaluation strategies can have different termination behavior
 - call-by-value vs. call-by-name
- Reduction strategies:
 - *Full beta reduction*—reduce anywhere
 - *Normal order*—reduce the leftmost outermost redex
 - *Call by name*—normal order, but never reduce inside abstractions
 - *Call by value*—reduce outermost redexes only, and reduce the argument first

Outline

- 1 Introduction
- 2 Programming with lambda calculus**
- 3 Recursion
- 4 Summarizing

Extending the calculus

- All values in pure lambda calculus are functions
- Often convenient to add syntax for numbers, Booleans, tuples, records, etc.

→ *applied lambda calculus*

- We already saw an example of lambda calculus extended with integers
- More interesting extensions possible: cells, exceptions, ...
- Extensions can, in theory, be expressed in terms of pure lambda calculus, but often very tedious

Representing Booleans

- The following definitions encode Boolean values (called *Church Booleans*)

$$\text{true} = \lambda t.\lambda f.t$$

$$\text{false} = \lambda t.\lambda f.f$$

- Here is a conditional operation

$$\text{test} = \lambda l.\lambda m.\lambda n.l m n$$

- We want
 - $\text{test } b \ v \ w \rightarrow v$, if $b = \text{true}$
 - $\text{test } b \ v \ w \rightarrow w$, if $b = \text{false}$

Example

$(\lambda l.\lambda m.\lambda n.l\ m\ n)\ \text{true}\ v\ w$

Example

$$\begin{aligned} & (\lambda l. \lambda m. \lambda n. l\ m\ n)\ \text{true}\ v\ w \\ \rightarrow & (\lambda m. \lambda n. \text{true}\ m\ n)\ v\ w \\ \rightarrow & (\lambda n. \text{true}\ v\ n)\ w \\ \rightarrow & \text{true}\ v\ w \\ \rightarrow & (\lambda t. \lambda f. t)\ v\ w \\ \rightarrow & (\lambda f. v)\ w \\ \rightarrow & v \end{aligned}$$

Other extensions

- Similarly Boolean functions, e.g.:

$\text{and} = \lambda b.\lambda c.b\ c\ \text{false}$

- Pairs

$\text{pair} = \lambda f.\lambda s.\lambda b.b\ f\ s$

$\text{first} = \lambda p.p\ \text{true}$

$\text{second} = \lambda p.p\ \text{false}$

- Intent is that Boolean value serves as a tag to pick either first or the second value of the pair

Church numerals

- One can encode numbers as follows

$$c_0 = \lambda s. \lambda z. z$$

$$c_1 = \lambda s. \lambda z. s z$$

$$c_2 = \lambda s. \lambda z. s (s z)$$

$$c_3 = \lambda s. \lambda z. s (s (s z))$$

...

- and functions for numbers:

$$\text{succ} = \lambda n. \lambda s. \lambda z. s (n s z)$$

$$\text{plus} = \lambda m. \lambda n. \lambda s. \lambda z. m s (n s z)$$

$$\text{times} = \lambda m. \lambda n. m (\text{plus } n) c_0$$

...

Local variables

- Syntactic sugar for local variables:

$$\text{let } x = M \text{ in } N \equiv (\lambda x.N) M$$

- A real programming language starts to emerge...
- Example: write this C program in lambda calculus:

```
int g(int x) { return x; }
code_that_follows;
```

⇓

$$\text{let } g = (\lambda x.x) \text{ in } \textit{code_that_follows}$$

⇓

$$(\lambda g.\textit{code_that_follows})(\lambda x.x)$$

Outline

- 1 Introduction
- 2 Programming with lambda calculus
- 3 Recursion**
- 4 Summarizing

Recursion

- Quite remarkably, recursion can be encoded in pure lambda calculus
- Not necessarily in a way that maps well to practical programming languages, though
- Defined via self-application
- Example...

Example

- Consider the definition of the factorial (we are assuming some extensions):

$$f(n) \equiv \text{if } n == 1 \text{ then } 1 \text{ else } n * f(n - 1)$$

- If we make the right hand side a function, left side becomes just f :

$$f \equiv \lambda n. \text{if } n == 1 \text{ then } 1 \text{ else } n * f(n - 1)$$

- Problem is that f is not bound. Repeat the trick:

$$G \equiv \lambda f. \lambda n. \text{if } n == 1 \text{ then } 1 \text{ else } n * f(n - 1)$$

- G takes a function and returns a function
- What happens if we call G with the factorial function (f)?

Example

$$f \equiv \lambda n. \text{if } n == 1 \text{ then } 1 \text{ else } n * f(n - 1)$$
$$G \equiv \lambda f. \lambda n. \text{if } n == 1 \text{ then } 1 \text{ else } n * f(n - 1)$$

- Check that:

$$G(f) = f$$

- f is a **fixed point** of function G

Fixed points

- Function maps values of the domain set to values of the range set
- A fixed point of some function G is a value f , such that $f = G(f)$
- A function can have more than one fixed point...
- or none at all, e.g. if intersection of domain set and range set is empty
- Examples:

$$f(x) = -x + 1$$

$$f(x) = x$$

$$f(x) = x + 1$$

Finding fixed points

- Certain combinators can find a fixed point

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

- Let's check that $g (Y g) = Y g$:

Finding fixed points

- Certain combinators can find a fixed point

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

- Let's check that $g (Y g) = Y g$:

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

$$Y g = (\lambda x. g(x x)) (\lambda x. g(x x))$$

$$Y g = g((\lambda x. g(x x)) (\lambda x. g(x x)))$$

$$Y g = g(Y g)$$

- $Y g$ is a fixed point of g
- $Y g$ results in a function that satisfies the original recursive definition
- Y is called the **Y-combinator**
- Not the only possible fixed point operator

Using Y

- Now look at **G** again
- Define `factorial = Y G`
- Compute `factorial 2`

```
(Y G) 2 =
```

```
G(Y G) 2 =
```

```
(λ f.λ n.if n == 1 then 1 else n*f(n - 1)) (Y G) 2
```

```
(λ n.if n == 1 then 1 else n*((Y G)(n - 1))) 2
```

```
if 2 == 1 then 1 else 2*((Y G)(2 - 1))
```

```
if 2 == 1 then 1 else 2*((Y G)(1))
```

```
2*((Y G)(1))
```

- Note in particular that **G** is not recursive but rather just a function that returns a function
- Recursion follows rules of lambda calculus
- Intuition: each round expands one level more of the computation

Outline

- 1 Introduction
- 2 Programming with lambda calculus
- 3 Recursion
- 4 Summarizing**

Summarizing the system

- The grammar and (small-step) operational semantics of lambda-calculus

- Syntax

$$t ::= x$$

$$\lambda x. t$$

$$t t$$

$$v ::= \lambda x. t$$

- Evaluation rules (call-by-name)

$$\frac{t_1 \rightarrow t_1'}{t_1 t_2 \rightarrow t_1' t_2} \quad \frac{t \rightarrow t'}{v t \rightarrow v t'}$$

$$(\lambda x. t) v \rightarrow [v/x]t$$

- \rightarrow is the smallest binary relation on terms satisfying the rules

Summary of lambda calculus

- Lambda calculus is a Turing complete language, and extremely simple
- Serves as a foundational calculus for studying programming languages
- With not too many extensions (which can be expressed in the calculus itself), becomes similar to practical functional languages
 - Imperative languages require a bit more stretching
- Very important tool in programming language design, in particular for type systems

Some pointers

- Good introductory material:
 - Pierce: Types in Programming Languages, MIT Press, 2002.
 - Mitchell: Concepts in Programming Languages, Cambridge University Press, 2003.
- Heavy duty classic:
 - H.P. Barendregt: The Lambda Calculus